UNIVERSITY OF LEEDS

Solid State Informatics Studies to Address Challenges in Pharmaceutics Development

Jakub Piotr Janowiak

University of Leeds

School of Chemical and Process Engineering

Submitted in accordance with the requirements for the degree of

Doctor of Philosophy

September 2020

Intellectual property statement

The candidate confirms that the work submitted is his own and that appropriate credit has been given where reference has been made to the work of others.

Acknowledgements

I would like to express my gratitude to everyone that has helped me on my journey without whom I would not have reached this point.

I would like to thank my supervisors – Elaine Martin and Kevin Roberts, for their patience and invaluable guidance. Special thanks go to Richard Marchese Robinson, who has helped me with the technical and scientific understanding. I am grateful for the insights provided by Andrew Maloney, Ilenia Giangreco, and Klimentina Pencheva.

I would like to acknowledge the incredible services provided by the University of Leeds HCP, without which I would still be waiting for my compute jobs to finish.

CP3 CDT has given me an incredible opportunity to do research, for which I am grateful. I would like to extend my thanks to all my colleagues from the CDT, with special mention to Alexandru Moldovan, Benjamin Tayler-Barrett, Thomas Hardcastle for their support and for the many memorable moments.

Although far away, my family has kept me motivated during the years of research.

Thank you to Hannah Stacey for all the support she has given me.

Finally, I would like to thank my late granddad for inspiring my curiosity in the natural world from a young age, something that stayed with me my whole life and led me to do a PhD.

Abstract

Cheminformatics methods such as Matched Molecular Pair Analysis (MMPA) and Quantitative Structure-Property Relationship (QSPR) models based on molecular structure have been widely used to address challenges faced during the Discovery stage of pharmaceutical product development. This thesis builds upon these concepts by including the solid state consideration to address challenges associated with the Development stage.

Polymorph propensity of molecules and solid state specific melting point (as a surrogate for solubility) were focused upon in the thesis. Matched Molecular Pair Analysis (MMPA) was used for the propensity study. However, no statistically significant molecular transformations were identified due to the small number of MMPs identified and the limited size and quality of polymorphism data.

The issue of the small number of MMPs was further analysed by constructing a Matched Molecular Graph. The graph approach allowed the comparison of the properties of datasets from different stages of the pharmaceutical development process. Datasets taken from Development stage contain fewer molecules with at least one MMP (25.1 %) and the lower total number of MMPs (2,776) compared to Discovery datasets of the same size (58.2 % and 10,321), making the analysis method less suitable.

A benchmarking dataset for crystal structure classification (into polymorphs and redeterminations) was curated, and the developed machine-learning based method (F1=0.910) along with existing methods (F1=0.780) of classification were compared.

A Message Passing Neural Network was used to develop a QSPR model using molecular and crystal information. The best model that only used molecular information achieved R^2 of 0.628 on the validation set, while the model trained with the crystal information obtained 0.649. The improvements were limited when compared to the QSPR model that only utilised molecular information; likely due to the limited polymorphic data and the typically small effect the crystal packing differences causes. The best model achieved test set R^2 value of 0.550.

This thesis provides partial solutions to the challenges of solid form informatics and forms a starting point for further research in the area.

Contents

List of figures

List of tables

Abbreviations

(G)RU – (Gated) Recurrent Unit

(R)MSE – (Root) Mean Squared Error

(U)FF – (Universal) ForceField

∇C – Partial derivative of the cost function (loss function) with respect to all the weights

$\mathcal{A}$ – machine learning algorithm

$\boldsymbol{a}$ – output vector of a neuron

Adam - Adaptive Moment Estimation

ANN – Artificial Neural Network

API – Active Pharmaceutical Ingredient, or Application Programming Interface

b – bais term

CCDC – Cambridge Crystallographical Data Centre

CSD – Cambridge Structural Database

D – Dataset (D_{tr} - training, D_{v} - validation, D_{te} - test)

DFT – Density Functional Theory

DHK – Matched Molecular Pairs Database designed by Dalke, Hert, and Kramer

$\mathcal{E}$ - set of edges of a graph

EI – Expected improvement

EMA - European Medicines Agency

f_{acq} – Acquisition function used during Sequential model-based optimisation

FDA - Food and Drug Administration

f_{grad} – function to compute the gradients during gradient-based training

f_{init} – initiation function

$f_{message}$ – message-passing function of graph-embedding process

f_{readout} – function to generate a fix-length of graph-embedding process

f_{update} – hidden update function of graph-embedding process, weight update function during gradient-based training

$\mathcal{G}$ – a graph (a set of vertices and edges)

GBO – gradient-based optimisation

GM – Graph Model

GSE – General Solubility Equation

GSK TCAKS – Tres Cantos Anti-Kinetoplastids dataset

$\boldsymbol{h}$ – hidden state vector of a neuron

$\mathcal{H}$ – machine learning model

HGP - Hierarchical Gaussian Process

HRF - Hussain and Rea Fragmentation method of Matched Molecular Pair identification

ICH - International Conference on Harmonisation

KDE – Kernel Density Estimates

kNN – k-Nearest Neighbours

$\mathcal{L}$ – Loss function

LIME – Local Interpretable Model-agnostic Explanations

MAE - Mean Absolute Error

MCS - Maximum Common Substructure method of Matched Molecular Pair identification

MLP – Multi-Layer Perceptron

MMG – Matched Molecular Graph

MMP(s) – Matched Molecular Pair(s)

MMPA – Matched Molecular Pair Analysis

MP – Melting Point

MPNN - Message Passing Neural Network

MST – Material Science Tetrahedron

NN – Neural Network

p – input vector to a neuron

PC(A) – Principal Component (Analysis)

PI – Probability of improvement

PL – Prediction Layer

QbD - Quality by desgin

QSPR – Quantitative Structure Property Relationship

ReLu – Rectified Linear function

RF – Random Forest

RMSD - Root Mean Squared Distance

SGD – Stochastic Gradient Descent

SMAC – Sequential Model based Algorithm Configuration

SMBO – Sequential model-based optimisation

SMILES – Simplified Molecular-Input Line-Entry System

STM - pre-Specified Transformation Methods of Matched Molecular Pair identification

SVM – Support Vector Machine

TPE – Tree-structured Parzen Estimator

UTM - Unspecified Transformation Methods of Matched Molecular Pair identification

$\mathcal{V}$ - set of vertices of a graph

W, ***w*** – weight matrix or vector.

X, ***X***, *x* – feature set, feature matrix, feature vector

Y, *y* – set of, or individual target value

γ – width of margins or quantile

η – learning rate during gradient-based optimisation

θ – parameters of a model

λ – hyperparameters of a model

σ^2 – variance

Chapter 1

Introduction

1.1 Context

Nearly half of the 7.5 years increase in life expectancy during the last half-century of the 20th century can be related to improvement in medical care [1]. Medicines in the form of tablets for oral administration play an essential role in improving the quality and expectancy of life with approximately half of the drugs on WHO's 'List of Essential Medicine'[2] being orally administrated [3]. The majority of these are of solid form. Engineering frameworks have been developed to better understand the behaviour of solid state products, thereby enabling the sustainable development of medical care.

The Material Science Tetrahedron (MST) is one example of such a framework, and it emphasises the importance of the links between structure, property, processing, and performance [4]. The objective of the framework is to optimise the Performance by adjusting the properties of the studied system; this is achieved by processing the material to alter its structure. The property is determined by the structure of the product. Key to the successful implementation of the Material Science Tetrahedron lies in understanding the structure-property relationship. The relationship can be further expanded to take into account structure at different scales such as molecular, crystal, and particle. These, in turn, affect properties to a varying degree. The MST has been utilised when undertaking challenges in pharmaceutical product development [5].

Pharmaceutical product development can be divided into two stages: Discovery and Development. During the Discovery stage, a large number of molecules are screened and optimised for specific properties such as molecular efficacy and toxicity [6,7]. The work focuses on molecular structure alteration to optimise relevant properties. Once a project reaches the Development stage, the molecular structure is set and work is performed on structures at a larger scale, such as solid form and particle. Many key properties that determine the ultimate performance of a drug are dependent on the solid state structure. Polymorphs, structures with the same molecule but different crystal arrangements, can exhibit different solubilities [8]. Therefore, it is highly desirable to be able to predict how molecular and crystal structures contribute to key properties.

The pharmaceutical product development process generates large amounts of data due to the trial and error approach, as well as its regulatory obligations [9]. In particular, a large number of molecules are screened during the Discovery stage. For this reason, many empirical methods such as Quantitative Structure-Property Relationship (QSPR) modelling [6,7,10] and Matched Molecular Pair Analysis (MMPA) [11,12] have been used to utilise the volume of data generated to better guide the process. In Discovery settings, the primary objective of the emprical models is to predict the activity of a molecule – hence the modeling is often called Quantitative Structure-Activity Relationship. However, in this thesis, activity is considered a property, so QSPR is used as a collective term for empirical models that predict activity or other property. Because of the relatively smaller volumes of data (structured data in particular), the Development side of the process has not utilised methods such as QSPR to a similar extent. The thesis aims to build upon the limited work in the area of solid state informatics to address key challenges in Development.

1.2 Aim and objectives

The primary aim of the thesis is to investigate the extent to which techniques deployed during the Discovery stage can be applied to the Development stage datasets to address challenges encountered at this stage. The intention is that this will contribute to increasing the efficiency of the Development stage, as well as facilitate the interaction between the two phases by allowing the Development stage challenges to be better anticipated and addressed during Discovery.

Molecular and crystal structures are studied in the thesis as they represent the interface between Discovery and Development. There are several relevant properties for the pharmaceutical product development. In the thesis, the emphasis is placed on polymorph propensity and solid state-specific melting point. Polymorph propensity refers to the propensity of a molecule to exhibit polymorphism which is of great importance during Development [9,13]. An empirical model of the propensity could be used as an additional consideration during lead optimisation (Discovery). The second property of interest in the thesis is the solid state-specific melting point. Melting point can be related to solubility, which is one of the key properties of a drug product due to its influence on bioavailability. A novel method of capturing crystal information as well as the significance of the solid state information was investigated.

The methods used in the thesis are MMPA and QSPR modelling. Within empirical modelling, there is typically a trade-off between the model’s ability to capture complex relationships and ease with which a model's prediction can be explained. Complex model explainability is an active area of research [14,15] and is also touched upon in Chapter 7. The two techniques selected represent explainability (MMPA) and the ability to capture complexity (QSPR).

1.3 Structure of the thesis

Chapter 2 expands on the context introduced in 1.1 and provides the theoretical background necessary for the rest of the thesis. Approach chapter (pink in Figure 1.1) focuses on the different iterations of the development and explains the principles of the method. Result chapters (gold in Figure 1.1) provide the specific context and aims, followed by the methodology used and the discussion of results. Discussion chapters (grey in Figure 1.1) are used to collate the topics presented in the previous chapters and offer an overarching discussion of them.

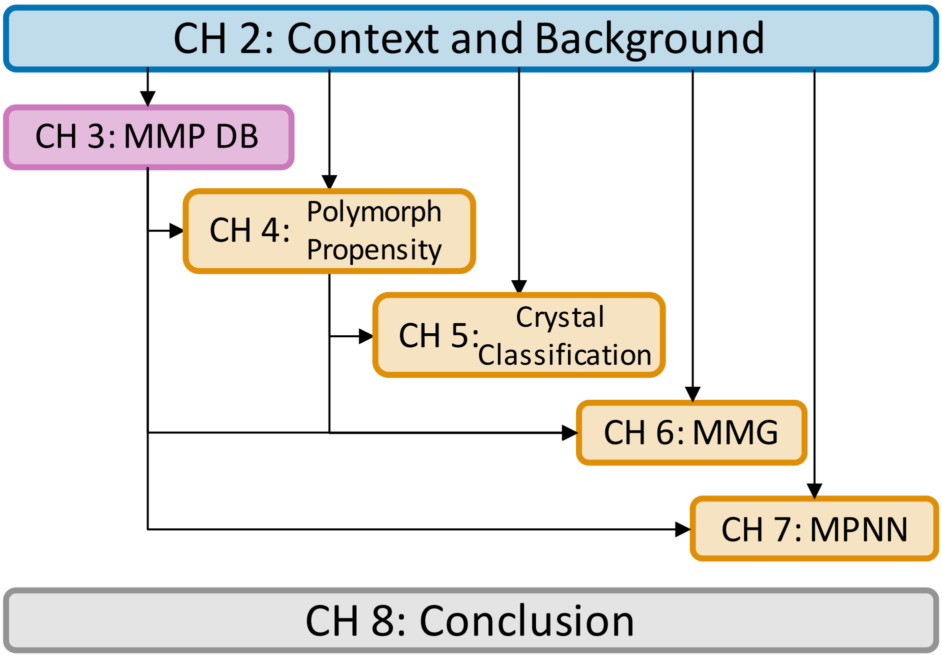

CH 2: Context and Background
CH 3: MMP DB
CH 4: Polymorph Propensity
CH 5: Crystal Classification
CH 6: MMG
CH 7: MPNN
CH 8: Conclusion

Figure 1.1: Overview of the structure of the thesis.

Context and theory – dark blue (chapter 2). Approach development – pink (chapter 3). Results – gold (chapters 4 – 7). Conclusion – grey (chapter 8). Abbreviated chapter names are used in the figure to provide an overview of the content. MMP DB – Matched Molecular Pair Database. MMG – Matched Molecular Graph. MPNN – Message Passing Neural Networks

Chapter 3 presents the method development performed to produce a database of Matched Molecular Pairs. The developed methodology was applied to the work presented in Chapter 4, Chapter 6, and Chapter 7. Chapter 4 presents the work on polymorph propensity prediction. It identifies several issues associated with the study of this phenomenon which are addressed in subsequent chapters (5 and 6). Chapter 5 addresses the need for a robust, automated method for the classification of pairs of crystal structures as different or same polymorphs by benchmarking an existing method and comparing it to the novel machine learning-based approaches. Chapter 6 describes the work done on assessing the suitability of datasets for Matched Molecular Pair Analysis. Crystal structure-specific melting point prediction is reported in Chapter 7. Chapter 8 collates the findings from the previous chapters, provides a discussion on how the work addressed the aims presented in 1.2, and sets out the direction for future research in this area.

Chapter 2

Literature Context and Theoretical Background

2.1 Pharmaceutical product development

Medicine has been an essential aspect of human civilisation since its beginnings. As large population centres developed, the need for a systematic approach to healing and the development of remedies increased [16–18]. Centuries of improvements led us to the modern drug development process that has brought us numerous treatments and increased the length and quality of life [1]. However, the process is currently riddled with a lack of productivity [19]. Identification of the critical challenges and development of potential solutions is essential to secure a sustainable healthcare system for future generations [19,20]. The Material Science Tetrahedron is presented as a framework for addressing these issues. Consequently, the key relationship between structure and property is identified, and the areas of focus for the thesis are discussed.

2.1.1 History

The profession of a physician was already established by 3,000 BCE in Mesopotamia [18,21]. At the time, observation-based treatments like pharmaceutics and surgical procedures were inseparable from superstitious healing rituals [18]. Pharmaceutical prescriptions dating back to 3,000 BCE were found to contain botanic, mineral and alcohol-based ingredients [16,18]. Many of these remedies were developed by religious reasoning and non-systemic trial-and-error approaches and were documented on clay tablets [18]. One of the ingredients mentioned in these tablets is willow leaves, which contains salicylic acid, a precursor to the active ingredient of aspirin (acetylsalicylic acid) used over 5,000 years later [21]. The fact that laws existed for punishment for mistreatment suggests that some degree of confidence in the treatments existed at the time [18]. Despite the limited scientific understanding and the absence of the modern scientific method, the ancient physicians were able to develop effective medicine using a rudimentary trial-and-error approach; thus establishing trial-and-error as a critical element of treatment development which is used to this day.

The introduction of a more robust, scientific approach to study illnesses and development of treatments is credited to Hippocrates, who was born around 500 BCE [22]. He was responsible for removing the assumption of the divine origin of disease, thus paving the way for a scientific approach based on the observation and understanding of nature [22,23]. The Hippocratic School of medicine is based on the principles of rationality, experiments, patient observation, and deduction [22,23]. The approach also aims to eliminate the presumptions and biases that the researchers may have [22,23]. Unfortunately, many of Hippocrates' followers partially abandoned these principles [22]. Nonetheless, the approach is evident in modern pharmaceutics development and forms the basis of the frameworks used. Indeed, due to his principles forming the basis of modern medicine, Hippocrates is often referred to as the "father of modern medicine" [23].

A significant step towards the modern approach to the development of medicine was made at the turn of the 18th and 19th centuries [17]. With the advancement of chemistry, Friedrich Wilhelm Adam Serturner was able to isolate morphine crystals from poppy seed juice in 1804 [24]. This was followed by the isolation of other active ingredients such as quinine (1820), atropine (1833), and cocaine (1860)[17]. In 1869, the first synthetic drug, chloral hydrate (discovered in 1832) was introduced into the pharmacopoeia [16]. These advances led drug development into a new era, focused on molecules as the basis of pharmacological effects.

The history of medicine is as long as the history of humankind itself. Diseases tend to propagate in areas of high population density, which put pressure on early civilisations to tackle this challenge [18]. Over the millennia, the techniques used to develop remedies evolved, resulting in the current pharmaceutical product development process.

2.1.2 Modern approach

The modern pharmaceutical product development process is the product of millennia of human ingenuity. It relies on the same principles developed over the ages: trial and error, the scientific method, and our understanding and mastery of the natural world. The term "pharmaceutical product development" is sometimes used to refer to only the late stage of the process of developing new medicine; however, in this thesis, the term is used to describe the entirety of the process. The modern framework can be divided into two main stages: Discovery and Development (Figure 2.1) [19,25,26]. The majority of the cost is incurred during the Development stage [19]. An overview of the process, with an emphasis on orally administrated drugs in the tablet form, is presented as the majority of drug products are of this form [3]. For the process to be initiated, a suitable target needs to be identified [25]. A target is a cell type, enzyme, gene, receptor, or pathway that has been shown to have an effect on a disease.

2.1.2.1 Discovery

Once a target is identified, high throughput screening is used to identify potential molecules that may affect it. Millions of molecules are screened using automated High Throughput setups at rates of 10,000s a day until promising compounds (Hits) are identified [25]. Due to the large number of data generated, the data management as well as the false positive rate need to be considered [20,27]. These are then screened further (Hit to Lead) to reduce the number of compounds of interest to 10 – 15 [19,25].

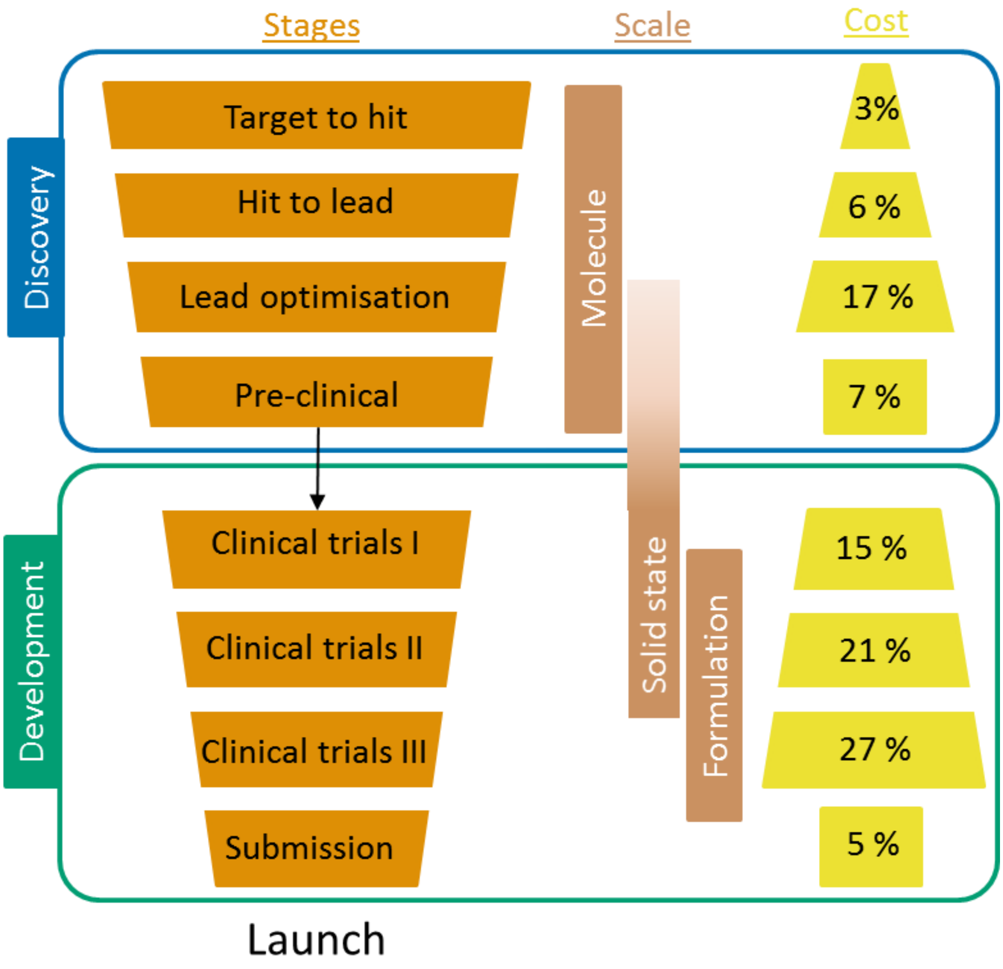


Figure 2.1: Pharmaceutical product development overview.

With each subsequent stage, the number of molecules considered decreases. Majority of work in Discovery is done on the scale of molecules. Solid state considerations are made during late discovery and Development. At the same time, formulation of the final drug product is investigated. The cost of each stage is based on values found in literature [19]**.**

During the Hit-to-Lead stage, assays are repeated to confirm the results of the initial screening stage, and further experiments are performed to obtain pharmacodynamic and pharmacokinetic profiles as well as toxicity data [27]. This is followed by lead optimisation, where small changes to the compound are made to improve the desired properties further (e.g. IC_{50} – concentration required to achieve 50 % inhibition [28]) and limit the undesired properties (e.g. toxicity) [25]. Some pre-clinical experiments are also performed to collect more data. At these early stages, the molecular efficacy of the potential drug is the key focus. In parallel, ways of synthesis and the delivery method of the drug candidate for the clinical trials is investigated. During the discovery stage, most of the work is performed on a molecular scale; traditionally, solid state considerations, such as polymorph screening and identification, were not taken into account at this stage [29]. However, this began to change at the dawn of the 21st century, where efforts were made to closely align the Discovery and Development efforts [29]. The details of this are explained in the Development section (2.1.2.2) and further considered as a solution to key challenges of the product development process in 2.1.3. At the end of Discovery, a decision is made whether to continue with the drug candidate and preceed to the Development phase. Performance and manufacturability are considered amongst other commercial considerations. To avoid costly late stage attrition, candidates are often dropped at this stage [30].

2.1.2.2 Development

The Development stage consists of parallel branches; clinical trials, the product formulation, and the manufacture process development [26,31]. Three stages of clinical trials are carried out on an increasingly large number of patients to determine the safety and efficacy of the drug candidate [32,33]. Phase 1 mainly focuses on safety and dosage, confirming the pre-clinical results. In Phases 2, the efficacy of the drug is tested on a larger number of patients. The phase 3 trials are the largest, where the drug performance is compared to a benchmark treatment if it is already available [34].

The second branch of the Development process is the formulation. In this branch, the key objectives are to ensure adequate stability and Performance of the final drug product [35]. The structure of the API is determined by the time it reaches the Development stage. However, the decisions regarding its solid form, particle characteristics, and final tablet composition leave room for optimisation for the key objectives. The drug candidates are screened for possible solid forms such as: hydrates, solvates, co-crystals, and polymorphs. Different solid forms can exhibit vastly different properties such as the case of Ritonavir where two polymorphs had different solubilities (170mg/mL and 30 mg/mL at 5 ^{o}C in ethanol-water mixture (3:1)) [8]. The screening is usually completed by repeatedly crystallising the drug candidate under different conditions such as varied cooling rates and solvents [36]. Based on the properties of the solid form, formulation at the larger scale is undertaken; optimisation of particle properties, and finally, the design of the tablet composition.

The formulation process works in tandem with the manufacturing process design. As the size of the clinical trials increases and the need for large-scale manufacture approaches, the manufacturing process is developed [37]. Quality by Design (QbD) framework is applied to the manufacturing process to ensure consistent quality of the drug product [38]. The guiding principle of this framework is the need for the scientific understanding of the underlying phenomena when designing the manufacturing route.

Successful clinical trials, formulation with the desired performance and stability, along with the manufacturing process is submitted to the governing agency for approval. In Europe, this falls under the European Medicines Agency (EMA), while in the United States, it is the Food and Drug Administration (FDA). The end of the pharmaceutical product development is marked by the granting of approval for the drug product.

2.1.3 Key challenge

The modern product development framework has had success in developing many drugs. However, in recent years, the productivity of the approach came into question [19,20]. The cost of each stage of discovery has decreased many-fold [19,20,34], yet the cost of the successful introduction of a novel drug has doubled every nine years since the 1950s [34]. The main reason for the decrease in productivity is said to be late-stage attrition, namely, the failures of drug candidates during the Development process. In fact, in the decade from 1998, 54 % of drug candidates that entered the Development stage failed to get approval from the FDA [33].

Several strategies have been proposed to address the low productivity within pharmaceutical product development. These include but are not limited to: human factor mitigation [39], organisational [19,34], predictive tool improvement [20], and integration of Discovery and Development processes for better performance optimisation [29]. The first two strategies, although important, fall outside of the scope of the thesis. The remaining two strategies, (1) improvement of predictive tools and (2) better integration of Discovery and Development, are focused upon in the thesis.

The most common cause of failure is poor efficacy of the drug candidate during the clinical trials [33]. Unless otherwise stated, efficacy refers to the clinical efficacy resulting from the bioavailability and molecular efficacy at the target site. Ability to predict the efficacy of a drug product in humans is the Holy Grail of pharmaceutical research. However, the problem is difficult due to the number of factors that affect it. These factors range from difficulty in predicting the biological effect of a molecule [40,41] to prediction of ADMET (absorption, distribution, metabolism, excretion and toxicity) [42–44] . The Material Science Tetrahedron (MST) is used to decompose some of the complexities of the efficacy prediction into simpler components. In particular, the emphasis is placed on the structures and properties that are focused upon at the interface of Discovery and Development. The MST framework is used to contextualise the research focus of this thesis.

2.1.4 Material Science Tetrahedron

The Material Science Tetrahedron is a framework that emphasises the importance of the links between structure, property, processing, and Performance (Figure 2.2). The origin of the tetrahedron can be traced back to a National Academy report from 1989, “Materials Science and Engineering for the 1990’s”, where it was proposed as a framework for a holistic view of the developments in the area [4]. The key elements of the MST are discussed in 2.1.4.1. In 2008, a paper highlighting the usefulness of the MST in pharmaceutical research and development was published [5]. Emphasis was placed on the relationship between two elements of the tetrahedron: structure and property (2.1.4.2). The structures and properties that are focused on in the thesis are presented in 2.1.4.3. Methodologies for studying the relationship between these are also discussed (2.1.4.4).

2.1.4.1 Vertices and edges of the tetrahedron

The MST consists of four vertices: structure, property, processing, and Performance [4,5]. Performance is the primary element of interest and is the reason for the development of a new product. In the case of the pharmaceutical product development, the Performance encompasses factors such as manufacturability, bioavailability,

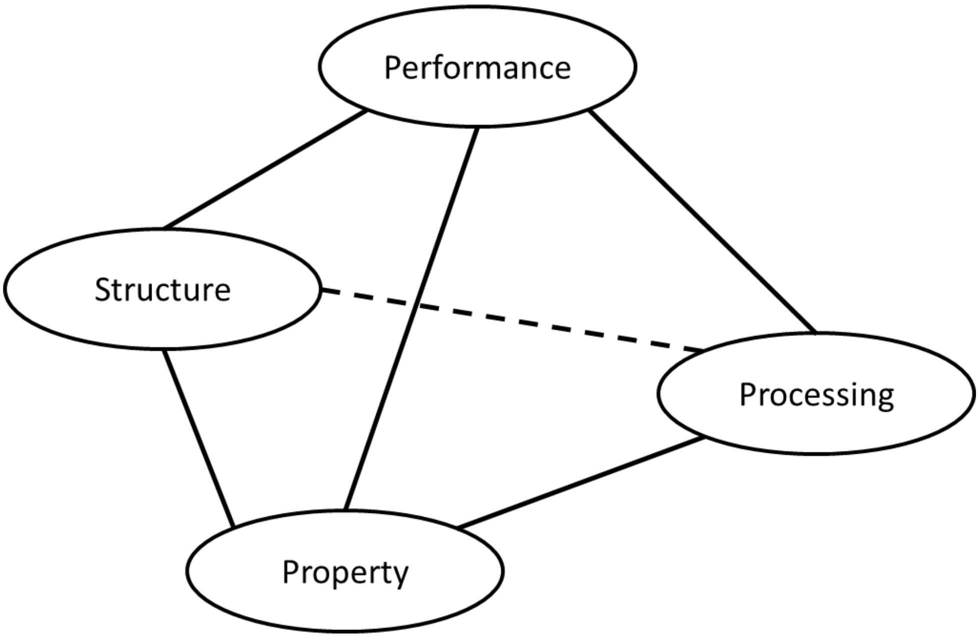


Figure 2.2: Material Science Tetrahedron.

The tetrahedron illustrates the interdependence of structure, property, processing, and performance.

toxicity (lack thereof), and molecular efficacy. Property is another vertex of the tetrahedron; it represents the different properties of the system studied, such as its physiochemical and mechanical properties. The structure of the system can be considered at different scales, from molecular through crystal to particle structure and beyond (formulation and the tablet form). Processing indicates the various actions that can be carried out on the product to alter it, such as synthesis of the API or milling.

The six edges of the MST represent the relationships between the four elements [4,5]. When attempting to improve Performance, the most trivial approach is to see what processing can achieve this. The processing-performance relationship is akin to Hippocratian physicians observing the effects of prepared remedies on patients during the classical era [23]. However, it does not provide any insight into the reason why certain processing affects Performance. This is not sufficient for modern-day regulatory bodies that require a Quality by Design (QbD) approach to be applied to the pharmaceutical product development [26,38]. For this reason, it is beneficial to focus on the relationship between processing and Performance via structure and property; namely, processing – structure, structure – property, and property – Performance [5].

Moving backwards from Performance, the first relationship is that between property and Performance. Understanding how properties affect Performance is paramount in being able to consistently develop methods of improving it. A drug candidate may fail clinical trials due to insufficient efficacy (Performance) caused by many factors, such as poor aqueous solubility (property). Beyond knowing which properties to improve, it is necessary to understand how these arise. This is the purpose of the structure – property relationship. A crystal structure that forms strong intermolecular interactions requires more energy to dissociate, thus reducing the solubility [45]. Knowing this, one can design a process that alters the crystal structure – for example, by forming a co-crystal. The way in which processing can be used to modify the structure of the system is captured by the processing – structure edge of the MST. It is possible to skip the structure and directly map the processing – property relationship. However, this does not contribute to the understanding of the underlying science behind the property [5]. The structure – property relationship forms the basis of the scientific understanding of the studied system’s behaviour.

In summary, the objective of the MST framework is to optimise the Performance by adjusting the properties of the studied system. This is achieved by processing the material to alter its structure. The property is determined by the structure of the product. Key to the successful implementation of the Material Science Tetrahedron lies in understanding the structure-property relationship. This relationship is further explored below.

2.1.4.2 Focus on Structure-Property Relationship (SPR)

Pharmaceutical products are complex, multi-component systems where the Performance of the product depends on many properties. In such systems, the

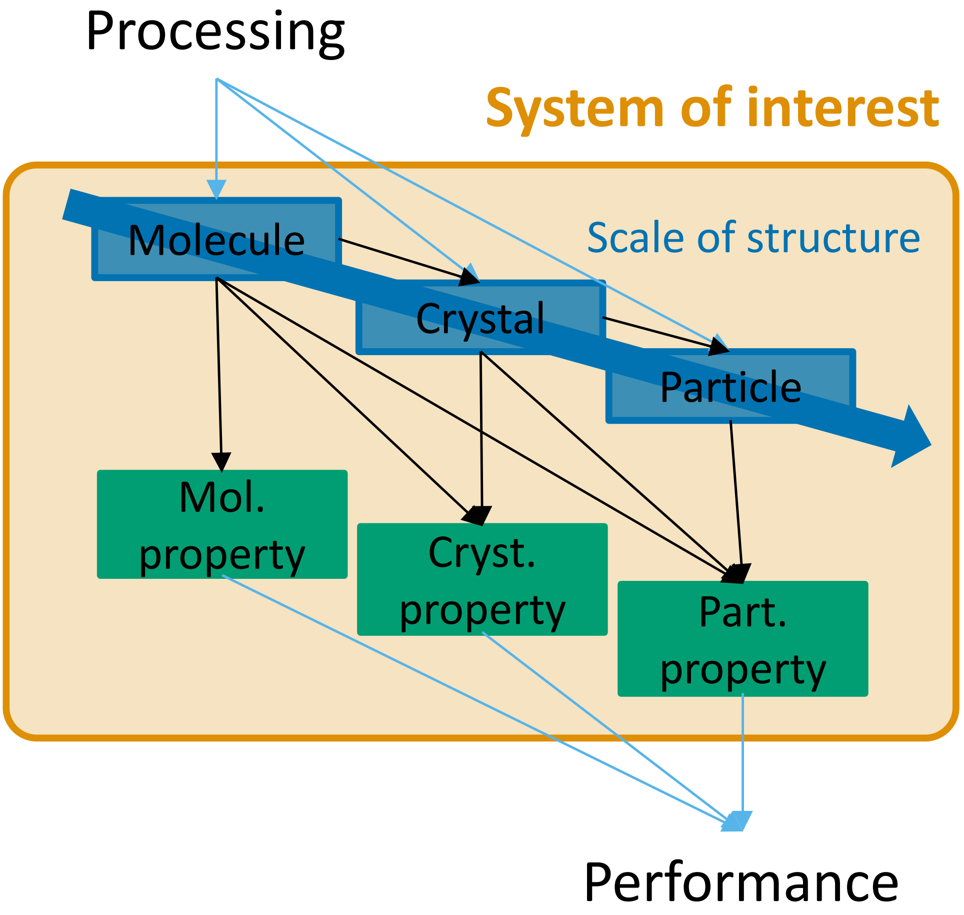


Figure 2.3: Structure Property Relationship (SPR) at different scales.

A system of interest (gold box) is affected by processing and determines the performance (blue arrows). Relationships within the system are indicated by black arrows. Structure (dark blue) exists at different scales where each subsequent scale of structure is determined by the prior scale of structures and the processing carried out. Properties (dark green) at each scale are determined by structures at that scale and all prior scales of structure.

relationship between structure and property exists at different scales. Furthermore, the structures at progressively larger scales are affected by the smaller structures of their component. Similarly, the properties at each scale are determined by the structure at that scale and the structures at the smaller scales. Different processing techniques can be applied to modify the structure at different scales as required. This multifaceted relationship between structures and properties is depicted in Figure 2.3.

An example of a drug that has insufficient efficacy (a key performance indicator) can be used to illustrate the complexity of this relationship. The majority of modern pharmaceutical product development is based on the molecular scale; hence, this is the smallest scale considered in this thesis. One possible reason for the poor performance is low bioavailability of the drug as a drug even with high molecular efficacy, cannot produce the desired effect if its bioavailability is too low. To reach the target, a molecule must dissolve from the tablet into the gastrointestinal tract, and permeate across the lipid bilayer. The permeability can be determined based on the molecular structure [46]. The solubility is a function of molecular and crystal structures. Restricting the consideration to the thermodynamic solubility, it depends on the energy change of combining solute and solvent molecules and the free energy needed to remove the molecule from the given crystal structure [45]. Thermodynamic solubility may refer to the equilibrium between the solution and the most stable crystal form for a given condition. Here however, thermodynamic solubility is used to refer to the equilibrium between the solution and any crystal form. The rate of dissolution is affected by the particle, crystal and molecular structure. The ratio of surface area to volume decreases as particle size increases [47]. Since dissolution is surface dependent, particle size affects the rate [48]. The crystal structure determines the surface chemistry, which affects the rate of dissolution as well. Beyond the three structures discussed here (molecule, crystal, and particle), bioavailability can be affected by tablet structure (excipients used).

2.1.4.3 Structures and properties of interest

The relationship between structures and properties exists at different scales. For a complex system such as a drug product, several scales and properties are important. In this thesis, the focus is placed on properties that are relevant to the interface between

Discovery and Development, with the aim of contributing to the integration of the two stages.

The two scales of structure that are focused upon in this thesis are molecular and crystal. The Discovery process is focused on molecules. Hence it was selected as one of the scales of structure. The crystal structure is the other structure that is focused upon in the thesis as it plays a vital role during Development. Many important properties are solid form specific [49,50], and earlier incorporation of the solid form consideration has already been shown to be beneficial [29]. Detailed consideration of these different scales of structure is presented in section 2.2.

The ability of a molecule to exhibit different packing arrangements in crystal form is referred to as polymorphism [51]. Different polymorphs have different properties, such as solubility and stability [8]. The ability to control polymorphism is an essential task within pharmaceutical product development [8,13,36,49,52]. The tendency of a compound to form polymorphs is called the polymorph propensity. Solubility has a direct impact on the bioavailability of the orally administrated drug product and is also an important property for processing (crystallisation). Bioavailability is defined as the amount of drug substance found in the circulatory system as a fraction of the amount of drug administrated where intravenous administration has 100 % bioavailability [36]. Both properties form part of the decision trees for quality management adopted by all major drug regulators around the world [9]. More details concerning each property and the data sources used in the thesis are described in 2.3.

2.1.4.4 Structure-Property Relationship (SPR) methodologies

The relationship between structure and property lies at the heart of the MST, and significant research effort has been dedicated to developing an understanding of it. The first principle approach aims to derive the relationship from the scientific understanding of the phenomena. In cases where the first principle approach is unfeasible, either due to lack of information, deficient theory, or limited resources due to computational expense, an empirical approach may be adopted [53].

Two categories of empirical methods are used in the thesis: Quantitative Structure Property Relationship (QSPR)[54] and Matched Molecular Pair Analysis (MMPA) [55]. In this thesis, QSPR is defined as any model, derived from the application of statistical or machine learning algorithms to relevant data, that maps a relationship between the structure of a system and a property of the system. The MMPA approach aims to map a relationship between a change in the structure of a molecule to a change in a property of the system. The principles of QSPR models are presented in 2.4 and applied in Chapter 5 and Chapter 7. Similarly, for the MMPA approach, the fundamental aspects are presented in 2.5 and its specific implementation in Chapter 3 and Chapter 4.

2.2 Scales of structure

Structure is one of the vertices of the MST and is a key focus of this thesis. Within the scope of pharmaceutical product development and the challenges identified in 2.1.3, structures can be found on various scales (Figure 2.3). The following sub-sections describe the different levels of structure associated with a pharmaceutical product. As discussed in 2.1.4.3, molecular and crystal structures are the main focus of the thesis and are presented in 2.2.1 and 2.2.2. To better contextualise the structures, crystal habit and is discussed (2.2.3). Larger structures, such as particles and tablets, are not discussed here as these fall outside of the scope of the thesis.

2.2.1 Molecules

A molecule is the smallest scale of the system that is considered in this thesis. A molecule consists of atoms that are covalently bonded to form functional groups and molecules. The molecular structure determines factors such as molecular toxicity and molecular efficacy at the site of biological action. It is at this scale that the majority of the Discovery work is carried out.

Molecules are defined by the identities of their constituent atoms and the way in which these are covalently bonded together, along with the shape the overall structure takes (conformers). The primary method used in the thesis to describe molecules is the simplified molecular-input line-entry system (SMILES) notation [56]. Molecular structures can also be expressed as graphs[57]; this is elaborated upon in 2.4.1 and Chapter 7.

2.2.2 Crystal structure

Once the scale of the system is increased to include several molecules, these can be arranged to create a new scale of the structure. Such groups of molecules exist in three different states of matter; gas, liquid, and solid. The focus of the thesis is on the solid state as most pharmaceutical products are of this state [3]. Solids can be further categorised into an amorphous and crystalline form. Crystalline structures are characterised by the regularity of the molecular arrangement. Amorphous solids have a random arrangement of molecules and fall outside of the scope of the thesis.

2.2.2.1 Crystal lattice

The regularity of crystals can be described using a lattice. An *n*-dimensional lattice is an infinite set of points defined by *n* linearly independent vectors such that

$$\boldsymbol{p} = \sum_{i}^{n} a_i \boldsymbol{x_i} + \boldsymbol{c} \qquad \text{Equation 2.1}$$

where the $\boldsymbol{x_i}$ is the ith basis vector, $a_i \in \mathbb{Z}$, $\boldsymbol{c}$ is an offset vector that is equal to 0 for lattice points (origin is one of the lattice points), and $\boldsymbol{p}$ is the vector representing a point on the lattice [58]. In essence, each lattice point is related to every other point by translation. This also implies that every point has the exact same environment. Crystal structures have lattices across three-dimensional space while the graphical example in Figure 2.4 illustrates the two-dimensional case for the sake of clarity.

The smallest repeating unit of the lattice is called the unit cell (shaded area in Figure 2.4). Each point within the unit cell has an equivalent point in every other unit cell by translation using Equation 2.1, where $\boldsymbol{c}$ is the point of interest in the unit cell. Several potential unit cells can be defined for a given system, but it is common practice in crystallography to define it as the smallest repeating unit that clearly captures the symmetry of the lattice [58]. In the case of the example in Figure 2.4, a rectangular unit cell can be used to define the repeating pattern. However, a parallelogram without right angles may be chosen if it represents the internal symmetry of the unit cell better.

Many crystals exhibit symmetry beyond translational symmetry parallel to the basis vector set. This is captured using 230 space groups (for a 3D system) and the respective symmetry operators [59]. The minimum motif required to recreate the full crystal structure is called the asymmetric unit [60].

2.2.2.2 Crystal packing

The arrangement of molecules, the crystal packing, is determined by the balance between *intra*- and *inter*-molecular interactions [61]. Intramolecular interactions include the covalent bonds formed between atoms to form a molecule. Intermolecular interactions consist of Van der Waals (VdW), Hydrogen bonds (H-bond), and electrostatic interactions [62]. VdW interactions occur between two dipoles, either permanent-induced or induced-induced. Hydrogen bonding is a directional interaction between two dipoles [63]. It forms between a hydrogen that is connected to an electronegative atom (H-bond donor) and an electronegative atom with a lone pair of electrons (H-bond acceptors). As a result of the directionality of hydrogen bond, they most commonly occur at around 180° with a lower limit of 110° [63]. Smaller angles are possible, but may indicate that a more stable crystal packing exists [64].

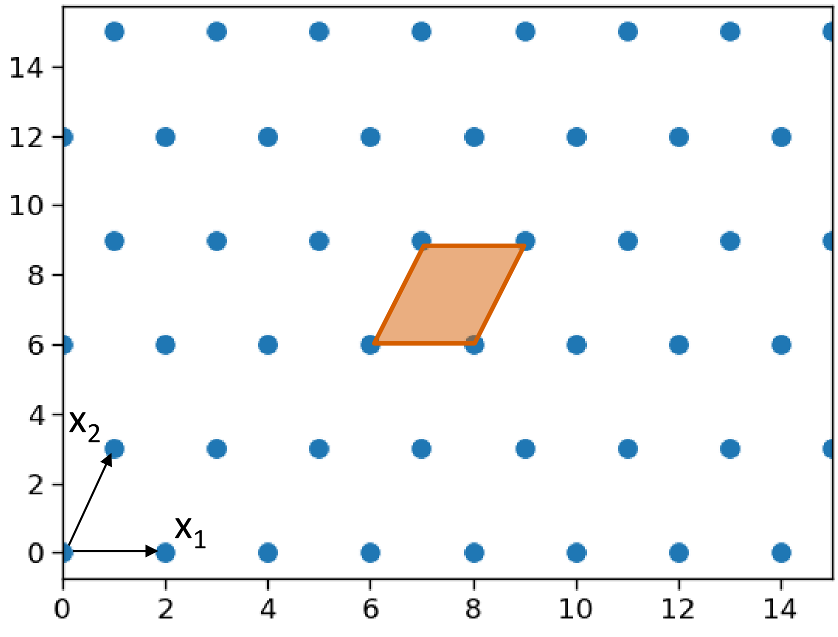


Figure 2.4: Example lattice with illustration of a unit cell.

The two vectors that define the lattice are x_1 = [2,0] and x_2 = [1,3]. The shaded area is the unit cell with dimensions equal to the magnitude of each vector.

Electrostatic interaction occurs between ions or between partially charged fragments of the molecules. An energetically favourable crystal packing is characterised by maximisation of these intermolecular interactions, while minimising the energy penalty due to disruption of the intramolecular interaction. It is important to note that Hydrogen bonds may also occur within the molecule itself, stabilising a particular conformer. Although intermolecular interactions are weaker than the intramolecular counterparts, the sum of intermolecular forces may be sufficient to induce a conformational change [8]. Due to the large number of possible geometries resulting from the interactions, a number of packing arrangements corresponding to local energy minima are often possible [50]. The ability of a molecule to exhibit multiple arrangements in the crystal state is called polymorphism.

2.2.2.3 Polymorphism

Although polymorphs have the same molecular structure, they may exhibit different physical properties that can lead to differences in the Performance of the chemical product. It is important to note that in some cases, solid forms such as solvates and co-crystal are wrongly termed polymorphs (or pseudo-polymorphs). Here, the term is strictly applied to crystals with the same molecular composition but different arrangements of these molecules.

Paracetamol is an example of a polymorphic molecule [65]. Figure 2.5 (a and c) show two polymorphs of the compound. Different intermolecular interactions govern the crystal packing of the two polymorphs [65]. The two unit cells shown in Figure 2.5 (a and b) also differ slightly – by an average of 1.7 % change in the unit cell dimensions. However, these two structures are the same polymorph (I) at different temperatures (-150.15 ^{o}C and room temperature). The two structures are not considered to be polymorphic due to the same crystal packing seen across the two structures. The importance of considerations of polymorphism within the context of the Development process is further discussed in 2.3.1.

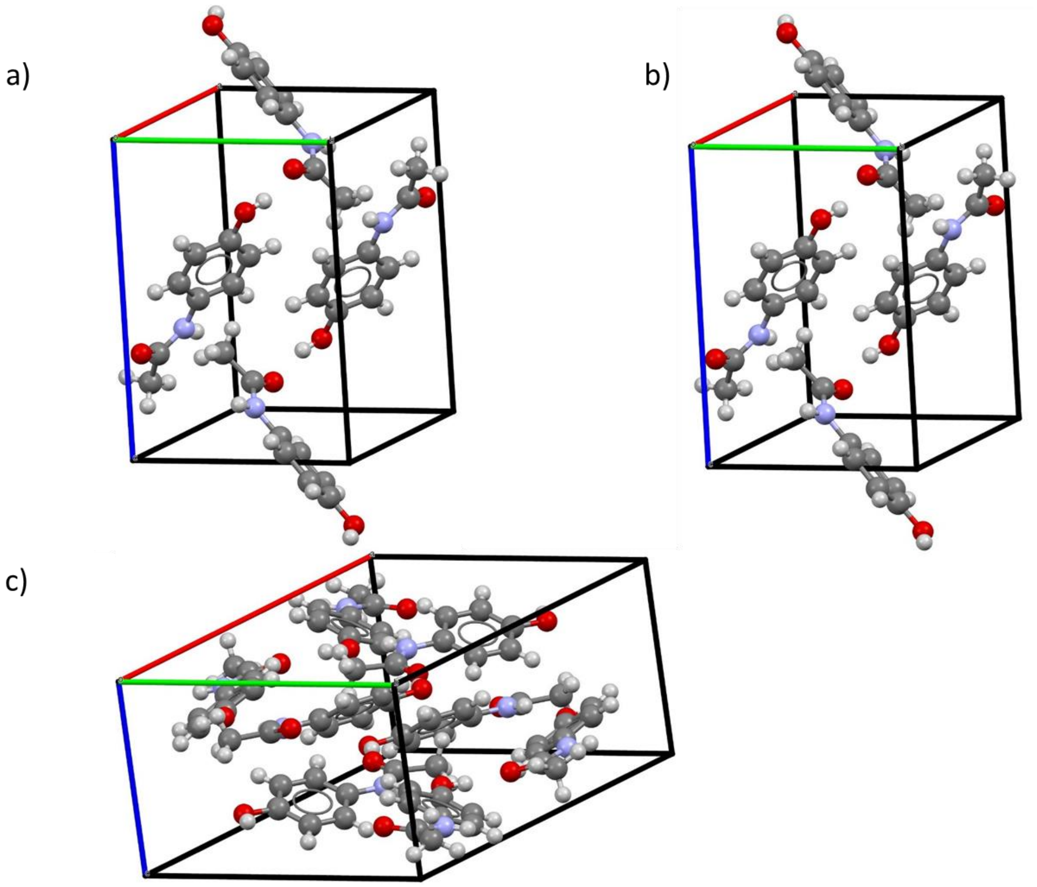


Figure 2.5: Three crystal structures of paracetamol, denoted by their CSD refcodes. a - polymorph 1 (HXACAN07), b – another experimental determination (“redetermination”) of polymorph 1 (HXACAN09), c - polymorph 2 (HXACAN08)

2.2.3 Crystal habit

The unit cell describes the way in which molecules pack to form solid state crystals. Crystals however, rarely grow in the same shape as the underlying unit cell [66]. The external structure of a crystal is called the crystal habit. The crystal habit is influenced by the internal structure of the crystal (packing, discussed in 2.2.2) and the crystallisation conditions [67]. The set of possible surfaces of the crystal (facets) are determined by the way in which the molecules pack. The crystallisation conditions, along with the surface chemistry of the facets, govern the growth rate of the facets. The crystal habit is defined by the relative growth rates of the facets. Presence of additives or impurities may selectively inhibit the growth of certain facets, thus changing the crystal habit [68]. Similarly, the solvent selection affects the binding at different facets, potentially leading to changes in crystal habit [69]. Other factors such as the temperature, the rate of change of temperature, and the degree of agitation can all have an impact on the resulting crystal habit [67]. The shape of the crystal affects the structure and properties at larger scales of structures such as particles, powder, and the final drug product. However, these fall outside of the scope of the thesis.

2.3 Properties and data sources

Polymorph propensity and solubility were identified in 2.1.4.3 as the main properties of interest in the thesis due to their relevance for the Development. Details of polymorphism and the data source for acquisition of the relevant information is presented in 2.3.1 with equivalent discussion for solubility presented in 2.3.2. However, the amount of solid state specific solubility data is limited [70]. Melting point can be related to solubility, and some solid state specific data is available for this property [71]. Details of the ways in which melting point data can be related to solubility as well as information regarding data acquisition are presented in 2.3.3.

2.3.1 Polymorph propensity

Polymorphism is of great interest within the pharmaceutical product development. For example, a previously unknown, more stable polymorph of ritonavir appeared two years after the product launch [8]. The higher stability caused the product to fail dissolution tests and consequently was removed from the market. Polymorphs may exhibit different physiochemical properties [36,49]. To avoid such problems, the regulatory bodies around the world have adopted strict requirements for identification and control of the solid form of drug products [9,13,52]. Polymorphism also plays an important role in patent litigation. Separate patents were granted for the two polymorphs of ranitidine (Zantac) [72].

International Conference on Harmonisation (ICH) of Technical Requirements for Registration of Pharmaceuticals for Human Use produced a series of guidelines concerning the requirements for approval of drug products which were subsequently adopted by all major regulatory bodies. Guideline on Quality Management Q6a decision tree 4 sets out the acceptance criteria for polymorphism in drug products and substances [9]. The guideline lists the requirement for identification of all solid forms.

The ability to accurately predict the likely number of solid forms would be of benefit, if available ahead of time. The propensity for polymorph formation is defined as the likelihood that a given compound will form polymorphs. This property of molecules is investigated in Chapter 4.

The Cambridge Structural Database (CSD) is a curated repository of small organic and metal-organic crystal structures that contains over 1,000,000 entries and is maintained by the Cambridge Crystallographic Data Centre (CCDC)[73,74]. The database contains 3D crystal structures and to varying degree the information concerning the conditions under which the structure was obtained. Several entries corresponding to a single molecular composition may be present. These entries correspond to the polymorphs and redeterminations. A redetermination is an experimental determination of the crystal structure of a given polymorph with a slightly different structure. The difference can arise from different conditions or the way in which the structure was resolved from the experimental data. Issues surrounding the differentiation between redeterminations of the same polymorphs and polymorphic structures is discussed in Chapter 5. The data contained within the CSD formed the basis of the polymorph propensity study (Chapter 4).

2.3.2 Solubility

Solubility is one of the key properties that contribute to the efficacy of the drug product as it affects the bioavailability of the drug [75]. True thermodynamic solubility indicates the maximum amount of the dissolved solute possible for a given state for the most stable crystal structure [76]. Due to kinetic barriers of formation of the most stable polymorph, the equilibrium between meta-stable polymorph and solution may be more relevant for drug absorption upon drug administration. Kinetic solubility is the solubility for a specific crystal structure. Processes such as dissolution and absorption are relatively quick, thus can be affected by differences in crystal packing [75].

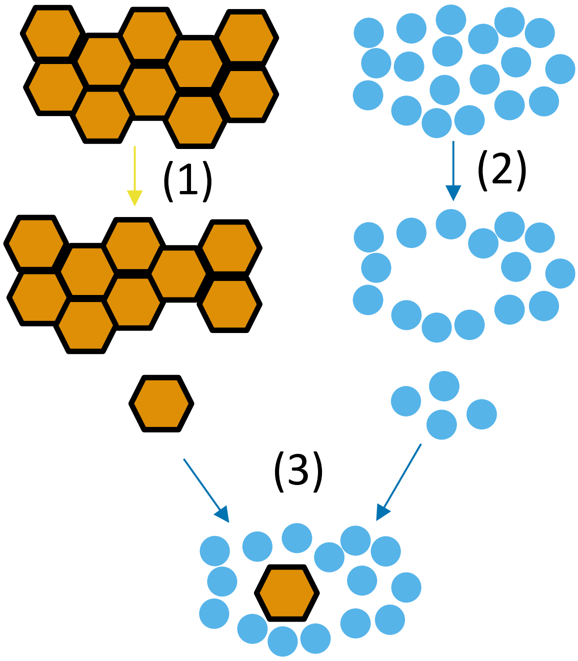


Figure 2.6: Simplified model of dissolution of a crystalline structure.

A molecule of the solute (gold hexagon) is removed from the crystal lattice (1). A void in the solvent (blue circle) is created (2). The solute molecule solvates into the void in the solvent (3). Yellow arrows indicate processes where the crystal structure of the solute plays an dominant role. [75]

To understand the factors that influence solubility, it is useful to deconstruct the relevant thermodynamic processes – dissolution (Figure 2.6) [75]. Here, it is assumed that the true thermodynamic solubility and the kinetic solubility is derived via equilibrium with the specific crystalline form studied. Hence, solubility can be related to the Gibbs free energy change associated with the dissolution process [77]. The first step of the process is the release of the solute molecule from the crystal lattice. The associated free energy change can be related to the strength of the intermolecular interactions within the crystal. The second step is the creation of a cavity within the solvent. The size of the solute molecule affects the free energy change associated with this process. The final step is the solvation of the free solute molecule. In reality, the dissolution involves formation of molecular clusters that disperse in the solvent [78]. However, this falls outside of the scope of the thesis, so the simplified model is used hereafter. The process can be split into two factors: (1) molecular and crystal structure-dependent strength of packing and (2) molecular structure-dependent solvation.

The two factors can be used to construct a grid that indicates the relative importance of the molecular and crystal features [71]. The grid is illustrated in Figure 2.7. The solubility decreases as molecules move diagonally from quadrant I to IV. In quadrant II the solubility is limited by solvation and in quadrant III, molecules exhibit solubility limited by crystal packing. Molecules in quadrant IV have poor solvation properties as well as strong intermolecular interactions within the crystal structure. The authors of the paper claim that many of the literature datasets underrepresent quadrant III, while many of the compounds within the Development stage fall here [71].

Different strategies for solubility improvement can be adopted based on the limiting factor. For crystal packing limited solubility, the intermolecular interactions can be weakened by altering the packing arrangement. Different solid form (polymorph, solvate, or co-crystal) may be selected to achieve this [75]. For solvation-limited compounds, alternative formulation changes can be used to improve solubility [35]. Therefore, the ability to predict solid form specific solubility, along with the limiting factor is highly desirable.

Informatics approaches have been shown to be useful tools for solid form selection [79]. Such informatics-based methods require large datasets to develop. Solubility is recognised as an important property within pharmaceutical product development, so

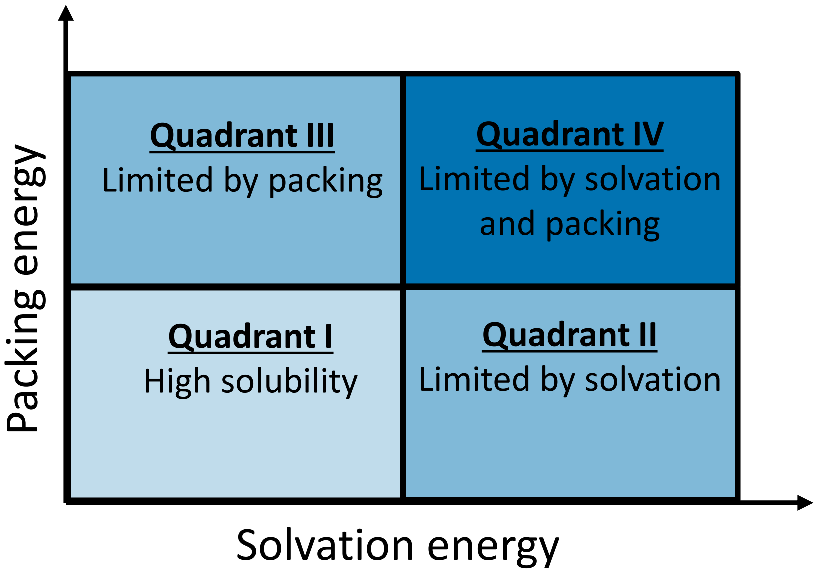


Figure 2.7: solvation - packing grid.

The grid is used to identify structures with crystal packing limited and solvation limited solubility. [71]

curated datasets are available. The Handbook of Aqueous Solubility contains 16,000 data points for 4000 compounds [80]. However, the dataset comprises very limited crystal structure information. Hence, rather than focusing on modelling solid state specific solubility, the focus was placed on melting point. This approach does not take into account the solvation aspect shown in Figure 2.6. However, there is value in developing better understanding of the solid state-limited solubility [71]. Solid state specific melting data is available, and the property can be related to the crystal packing contributions to solubility [70,71].

2.3.3 Melting point

Melting point is the temperature at which a phase transition between solid state and liquid is thermodynamically favourable. It corresponds to the energy required for the molecules to break the intermolecular interactions within the crystal structure. The melting point can be used as an indicator of the energy required to remove a molecule from the crystal lattice (process (1) in Figure 2.6) [75] and the packing energy in Figure 2.7 [35,71]. Furthermore, the melting point is used for solubility prediction via the General Solubility Equation (GSE) [81].

Melting point data is widely available. However, similarly to the solubility data, the melting points are rarely associated with a specific polymorph. Open Melting Point Data resource contains 13,000 curated data points for a diverse range of temperatures [82]. The dataset, along with its subsets, has been used to assess the Performance of several models [81,83]. A larger, less curated dataset was generated by text mining patent literature [83]. The Patent Dataset contains 289,379 datapoints, and it was shown that accurate models can be trained on less curated datasets as well. Based on the analysis of the datasets, the typical error within literature datasets was estimated to be approximately 32 - 35 ^{o}C [83,84]. The error is not due to instrument error but rather due to impurities and polymorphism, which is not controlled in many cases. These datasets do not contain any solid state information.

In contrast to the limited polymorph specific solubility data in curated datasets, CSD contains some polymorph specific melting point data. Approximately 17 % of single component structures in the CSD have melting point data reported along with the crystal structure. A dataset of 53,756 crystal structure specific melting point was extracted from the CSD. The CSD MP dataset is further discussed in Chapter 7. The

CSD contains a wide range of structures. To ensure the MP dataset was relevant to the pharmaceutics, the drug-likeness was investigated. The strict definition of drug-likeness is not set; however, it is generally accepted that a molecule is “drug-like” if it is similar to available drug molecules [85]. A number of metrics can be used to describe “drug-likeliness” such as molecular weight (typically within 300 and 400 g/mol), rotatable bond count (5 and 6), and other molecular properties [85]. Approximately 93 % of structures with the CSD MP dataset fall within the drug-like melting point range of 50 °C – 250 °C [85] (Figure 2.8).

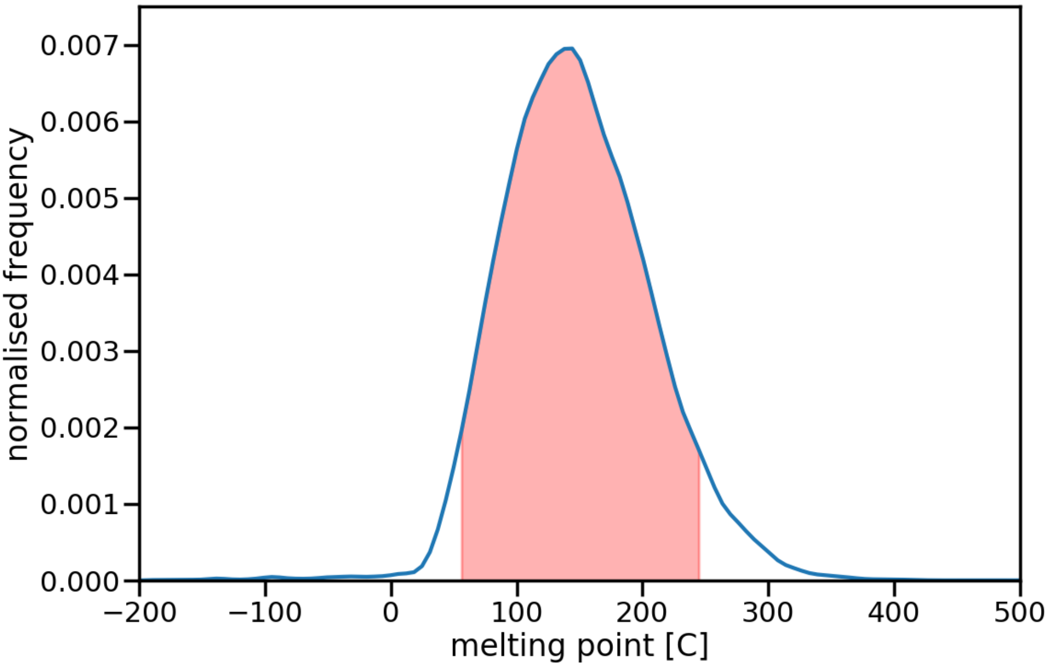


Figure 2.8: Distribution of melting points of structures with the CSD single component melting point dataset.

The drug-like melting point range (50 °C – 250 °C) is highlighted. Kernel Density Estimate (KDE) using gaussian kernel with kde factor of 1 was used to construct the plot.

2.4 Quantitative Structure Property Relationship (QSPR)

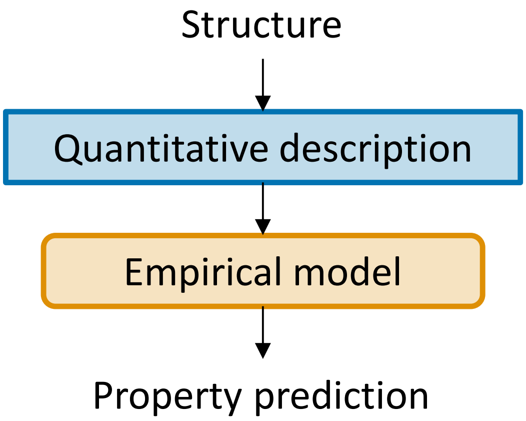


Figure 2.9: Quantitative Structure Property Relationship overview.

Quantitative structure property relationship (QSPR) is a paradigm that aims to develop accurate empirical models that predict a property based on the structure of the system studied (Figure 2.9). A distinction is often made for models of biological activity, referred to as Quantitative structure activity relationship (QSAR); however, the two are considered to be same and are referred to as QSPR models in this thesis. Structures themselves cannot be used to map the relationship to the property of interest, so some quantitative description is needed (2.4.1). This description takes the form of a feature vector. The variables in this feature vector are typically termed “descriptors” within the QSPR community. Features and descriptors are used interchangeably in the thesis. The empirical model aims to find a function that accurately maps features onto targets ($f : \boldsymbol{X} \to Y$). The principles of how this is accomplished, by training the models, is presented in 2.4.2. Section 2.4.3 explains the way in which the Performance of the models is measured. The machine learning algorithms used in the thesis are discussed in the subsequent sections – Random Forest in 2.4.4, Support Vector Machine in 2.4.5, and Neural Networks 2.4.6. Many models also have hyperparameters which are a set of parameters that remain constant during training but affect the Performance. The hyperparameters are adjusted by the process of hyperparameter optimisation, which is discussed in 2.4.7. An overview of the application of QSPR models is presented in 2.4.8.

2.4.1 A quantitative description of the structure

The decision on the descriptor set used for a QSPR is crucial, as it has an effect on the possible performance of the model. The descriptors need to capture relevant and generalisable structural characteristics which are relevant to modelling the property of interest. In this thesis, the descriptors are divided into two categories: structure descriptors and graph embedding. Molecules can be seen as graphs (sets of connected nodes – atoms); as such graph embedding techniques can be used to generate a descriptor representation of the structure (2.4.1.2). In the thesis, graph embedding techniques are defined as techniques that use graph representation of the entire structure to produce a fixed-length representation of the given structure. Fingerprint methods also use graph representation of structure; however, only a substructure is often used to develop several fingerprints followed by one-hot key encoding for the presence of a given substructure. One-hot key encoding is a process where a feature is constructed for each unique value of interest (in this case one feature is constructed for each fingerprint). The value of the generated feature is set to 1 where a given sample has that specific unique value and set to 0 otherwise. The fingerprint approaches, along with any other method of encoding the structure as a set of numerical variables are collectively referred to as a structure descriptor in this thesis (2.4.1.1).

2.4.1.1 Structure descriptors

For a QSPR model to accurately map the relationship between structure and the property of interest, the structure needs to be adequately described. The primary objective of a structure descriptor is to capture some information about the structure while remaining invariant to artificial differences in the structure representation. This means that calculation of a feature should always return the same value for the same molecule, irrespective whether the molecule has been rotated or its atoms numbered in a different order.

Historically, simple molecular descriptors such as the number of atoms in a molecule were used to find correlations with properties [86–88]. As the computation and experimental capabilities increased, so did the number of usable molecular descriptors [89]. Thousands of molecular descriptors have been developed that have demonstrated a correlation with some property of interest [89–93].

The same principles can be extended to crystal descriptors. As discussed in 2.1.4.2 and 2.3.2, properties such as solubility are affected by the solid structure. An inability to adequately capture the solid state information was given as one of the reasons for the difficulty of predicting solubility [94]. Experimental melting point data, lattice energy, and 3D molecular descriptors calculated for structures found in the solid state were used as crystal descriptors for solubility prediction [70,95].

Further to these descriptors, fingerprints can also be used for QSPR modelling. Typically, fingerprints encode the presence of a specific molecular feature (e.g. molecular substructures) [96,97]. The fingerprint approach has been used extensively to construct QSPR models for solubility [98] and other properties such as activity [99].The presence of these substructures can then be used as a descriptor. The task-specific descriptors used in the thesis are discussed in the respective chapters (Chapter 5 and Chapter 7).

2.4.1.2 Graph embedding

Many applications within the broader machine learning community require an effective representation of graph data [100]. For this reason, a number of graph embedding techniques have been developed for different tasks. The definition of a graph is presented, followed by graph embedding methods that can be used to generate a fixed-length representation of such a graph.

A graph $\mathcal{G}$ is defined by a set of vertices $\mathcal{V}$, and edges $\mathcal{E}$; $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. An edge $e \in \mathcal{E}$ is defined by the two vertices it links; $e_{ij} = (v_i, v_j)$ [101]. Only undirected graphs are used in this thesis. Undirected graphs are graphs such that $e_{ij} \equiv e_{ji}$: $e_{ij} \in \mathcal{E}$. For undirected graphs a set of all neighbouring vertices for a given vertex $\mathcal{v}_k$ is giving by

$$NBR(v_k) = \{v_i : (v_k, v_i) \in \mathcal{E}\} \qquad \text{Equation 2.2}$$

Additional information concerning vertices and edges properties can be embedded as labels l_v and $l_{e,}$ respectively. Graphs can be further categorised into homogenous – graphs such that $|l_v| = |l_e| = 1$ for all vertices and edges – and heterogeneous graphs. As molecules tend to have more than one vertex label (atom type) and edge label (bond type), only heterogeneous graph embedding methods were considered.

A comprehensive review of graph embedding techniques can be found elsewhere [100]. Here, the focus is placed on Message Passing Neural Network (MPNN) framework [102]. The general principles of neural networks are presented in 2.4.6. MPNN is a class of neural networks that generate a fixed-length representation by consecutively updating the state of each vertex $\mathcal{V}$ ($h_v^{(t)}$) followed by pooling all the states. Each vertex is initialised based on the vertex label (l_v),

$$h_v^{(0)} = f_{init}(l_v) \qquad \text{Equation 2.3}$$

The initialisation (f_{init}) can also be random. The message passed to the vertex ($m_v^{(t+1)}$) is a function of the current state of the vertex ($h_v^{(t)}$) and its neighbours using a message function ($f_{message}$).

$$m_v^{(t+1)} = \sum_{w \in NBR(v)} f_{message}(h_v^{(t)}, h_w^{(t)}, l_{vw}, t) \qquad \text{Equation 2.4}$$

The relationship can also be dependent on the timestep *t* and the edge type that connects the vertex with its neighbour (l_{vw}). The message is then used to update the state of the vertex using an update function (f_{update}),

$$h_v^{(t+1)} = f_{update}(h_v^{(t)}, m_v^{(t+1)}, t) \qquad \text{Equation 2.5}$$

At each time step, each vertex receives message only from its neighbours. However, as the state of the neighbours is updated based on their neighbours, effectively the information concerning each vertex is propagated through the graph. At time step *t*, information from vertex *t*-connections away reaches its vertex. After *T* iterations of message passing, the fixed-length graph representation is computed based on the state of all the vertices.

$$p_{structure} = f_{readout}\left(\left\{h_v^{(T)} : v \in \mathcal{V}\right\}\right) \qquad \text{Equation 2.6}$$

The difference between the approaches within the MPNN framework comes from the functions used ($f_{message}$, f_{update}, $f_{readout}$) [97,102–107]. The details of the MPNN method used in the thesis to capture molecular and crystal structures are discussed in Chapter 7.

2.4.2 Principles of machine learning

For the empirical models to make useful predictions, some parameters need to be optimised. In the case of machine learning algorithms, this is referred to as learning or training and can be split into three categories: supervised, unsupervised, and reinforcement. In case of supervised learning, a dataset that contains pairs of features (***X***) and targets (*Y*) is fed to the learning algorithm which then aims to find a mapping between the two ($f\colon \boldsymbol{X} \to Y$) [108]. Reinforcement learning is a method by which an agent (the model) is given a reward for an action based on observations with the aim to learn actions that maximises the reward value [109]. Unsupervised learning does not require labelled data nor rewards for actions; instead, it draws inferences from the datasets such as clustering or dimensionality reduction [110]. Supervised learning is used in this thesis. The general learning task is defined below (2.4.2.1), and the key factor in determining the usefulness of a model – generalisability is defined in 2.4.2.2.

2.4.2.1 Definition of the task

Sets of N instances (examples) of features ($X = \{\boldsymbol{x_1}, \boldsymbol{x_2} \dots \boldsymbol{x_N}\}$) and corresponding target values ($Y = \{y_1, y_2 \dots y_N\}$) are arranged in pairs to form a dataset for supervised learning ($D = \{(x_i,\ y_i)\colon x_i \in X, y_i \in Y, 1 \leq i \leq N\}$). The target can be a discrete label, a continuous number, or a vector (of labels or continuous values); the type of the target is used to categorise empirical models. In the case where the target is a label, the task is called classification. In the case of a regression task, the model outputs a predicted value(s). The empirical model ($\mathcal{H}_\theta$) is a function constructed from parameters (θ) given a set of hyperparameters (λ) using the selected algorithm $\mathcal{A}$. Machine learning models such as Random Forest (RF), Support Vector Machine (SVM), Artificial Neural Networks (ANN or NN), k-Nearest Neighbours, and Naïve Bayes are commonly used within the field of cheminformatics. In this chapter, the emphasis is placed on RF (2.4.4), SVM (2.4.5), and NN (2.4.6) as these are used in the thesis (Chapter 5 and Chapter 7). Whilst random forest is typically considered a non-parametric model, rather than one which constructs a function based upon a pre-defined functional form and a fixed set of adjustable parameters, the set of split criteria for the trees can be seen as parameters.

$$\mathcal{H}_\theta = \mathcal{A}(\theta; \lambda) \qquad \text{Equation 2.7}$$

Hyperparameters are kept constant during training. The performance of many machine learning algorithm is greatly affected by the selection of hyperparameter. The details of performance measures are presented in 2.4.3 while hyperparameter tuning is discussed in 2.4.7. The cost function is used to compute the loss of the model for a specific set of parameters.

$$C(\theta;\lambda) = \mathcal{L}(X, Y;\, \mathcal{H}_\theta) \qquad \text{Equation 2.8}$$

$\mathcal{L}$ is the loss function and is selected prior to training. Several loss functions for regression and classification tasks are presented in 2.4.3.1 and 2.4.3.2, respectively. The value of the cost function may be computed for each iteration of training (per batch in case of Neural Networks) or independently for each branch of a Random Forest model. The aim of the training is to find a set of parameters (θ^*) such that the cost function is minimised;

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, C(\theta;\lambda) \qquad \text{Equation 2.9}$$

In case of Random forest, this can be seen as the optimum tree structure for each tree. Several strategies for finding the optimum parameters (θ^*) exists. Each algorithm involves a different approach to finding the optimum set of parameters. The algorithm-specific considerations are presented in the respective sections below (2.4.4 – 2.4.6). Although the optimum set of parameters is based on the training set, it is important to note that the aim of any supervised model is to generate useful predictions on unseen data. Some considerations on how this can be estimated are discussed below (2.4.2.2).

2.4.2.2 Generalisation

The ability of a model to make accurate predictions on unseen data taken from the same distribution as the data used for training is referred to as generalisability [108]. To get an estimate of model generalisability, the available dataset (D) is commonly split into training (D_{tr}), test (D_{te}), and potentially validation (D_v) sets as needed. To correctly measure generalisability, each dataset should have the same distribution.

The test set is used as the unseen data at the end of the development process and is split from the remaining data first. If only a single model is trained (one algorithm and one set of hyperparameters), there is no need to split the remaining data into training and validation as all of it can be used for training. However, when a number of models are to be trained, it is beneficial to split the dataset further into training and validation sets. This is because a particular set of features or hyperparameters may yield good predictions on a particular set of data due to chance. Hence, predictions made on data used to select the best model may be optimistically biased [111,112]. Hence, a separate validation set should be employed for the model section. The datasets can be split before the training commences into a training and validation datasets or cross-validation procedure can be used [113].

Once the datasets are prepared, the model development process can be initiated. The examples from the training set are used to fit parameters of the model (or construct decision trees in case of RF). A solution to the minimisation problem defined in Equation 2.9 is approximated using this dataset.

The validation set is then used to give an estimate of how well the model performs on unseen examples. The loss or other performance measure is computed for the predictions made by the model on the validation set. If the performance on the validation set is significantly worse than on the training set, it is likely that the model is overfitted [114]. Overfitting is the result of the model mapping the noise within the training set. The performance on the validation set is typically used for selection of the best model.

Although no parameter adjustment is directly performed based on the validation set, the model was adjusted (or selected) based on the validation set performance; hence the dataset is no longer “unseen”. The test set provides an unbiased evaluation of the performance of the model. No adjustments to the model are made after it is run on this dataset. The performance represents the ability of the model to predict unseen examples. In the section below (2.4.3), methods for assessing the performance of empirical models are discussed.

2.4.3 Performance measures

Performance measures are required to quantitatively assess the quality of a model. This is needed for the training process (Equation 2.8) as well as for the understanding of how well the model generalises. To monitor the iterative training process, easily computable functions are preferred. These are referred to as loss functions and are presented for regression and classification task in 2.4.3.1 and 2.4.3.2, respectively. The measure of generalisability of the trained models is calculated on validation or test sets. The loss function used for training the algorithm can be used to assess this; however, it is sometimes not easily interpretable and may not fully capture the Performance of the model. For this reason, a number of metrics have been developed to allow comparison of models. These are also discussed in the respective sections (2.4.3.1 and 2.4.3.2).

2.4.3.1 Regression task

The aim of a regression task is to predict a value for a given feature vector that closely corresponds to the actual value. The 'closeness' of the prediction can be captured using different loss functions. For each of the functions, the predicted value for the ith input is defined as,

$$\hat{y}_i = \mathcal{H}_\theta(x_i) \qquad \text{Equation 2.10}$$

and the error is,

$$e_i = y_i - \hat{y}_i \qquad \text{Equation 2.11}$$

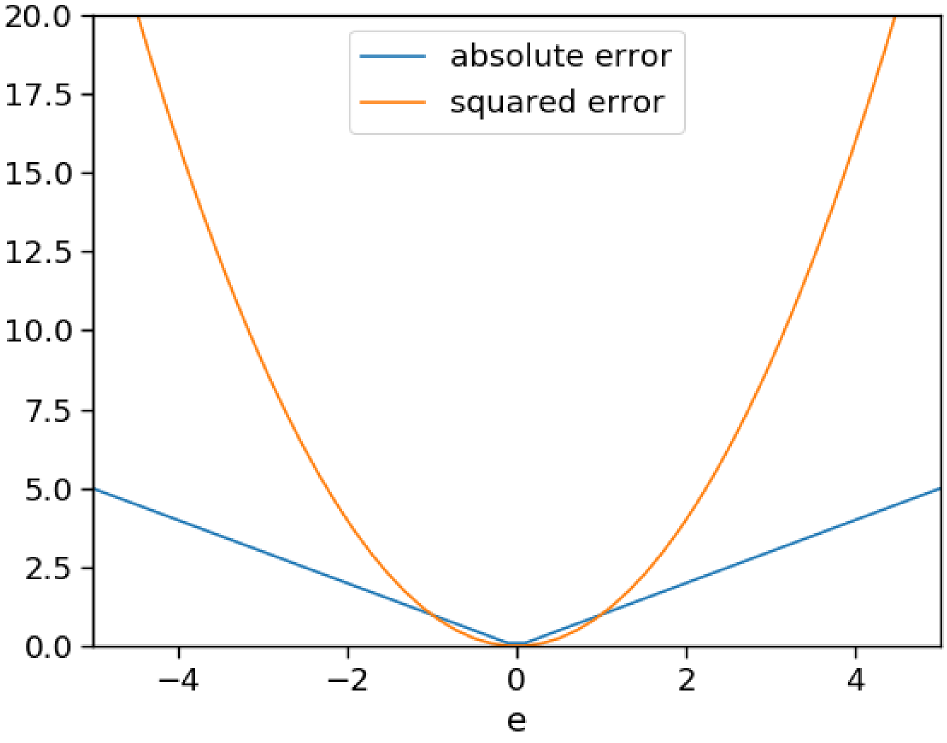


Figure 2.10: Comparison of absolute and squared errors.

L1 and L2 are the two basic loss functions which are used in this thesis. For a dataset of n items, L1 function, also known as the mean absolute error (MAE) is defined as,

$$MAE = \frac{\sum_{i=1}^{n} |e_i|}{n} \qquad \text{Equation 2.12}$$

The L2 function, mean squared error (MSE), is defined as,

$$MSE = \frac{\sum_{i=1}^{n} (e_i)^2}{n} \qquad \text{Equation 2.13}$$

The two loss functions are compared in Figure 2.10. The value of squared error increases fast as the error increases compared to the absolute error ($as\ e \rightarrow \infty, e^2 \gg |e|$). As a result, an outlier can have a disproportionally large effect on the MSE. This may result in unstable MSE value when working with datasets that have outliers. MAE is not affected by the outliers to the same degree. However, the gradient is constant and independent of the error value. This may affect the training as the correction made to the model is proportional to the magnitude of the loss function. To address the shortcomings of the L1 and L2 loss function, other loss functions have been developed that attempt to capture advantages of each. Huber loss [115] (smooth mean absolute error) and log cosh loss [116] approximate the behaviour of L2 for small e and L1 for large e.

A square root may be taken of MSE to convert it to the same units as the target; this is called root mean squared error (RMSE). Comparison between performances on different datasets can be made using R^{2,} which is computed by scaling RMSE based on the distribution of the target values in the dataset (total sum of squares) [117].

$$R^2 = 1 - \frac{RMSE}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \qquad \text{Equation 2.14}$$

The performance measures defined in this section can be used for training and performance comparison of regression tasks. In this thesis, only neural networks were used for regression.

2.4.3.2 Classification task

Classification tasks aim to predict a correct label based on a given feature vector. In this thesis, only a binary classification (two labels) was performed so this section focuses on loss functions that are used for this purpose. Algorithms such as SVM classifier calculates the confidence of the model in the particular classification. For this reason the model output $\mathcal{H}_\theta(x)$ is not restricted to just the class label ({-1,1}). The Hinge loss is commonly used for measuring the performance of SVM classifiers [118,119].

$$L_{hinge(i)} = \begin{cases} 1 - y_i\mathcal{H}_\theta(x_i) & if\ y_i\mathcal{H}_\theta(x_i) < 1 \\ 0 & otherwise \end{cases} \qquad \text{Equation 2.15}$$

The multiplication of y_i and $\mathcal{H}_\theta(x_i)$ is done to reward predictions where the signs of $\mathcal{H}_\theta(x_i)$ and y_i align (correct prediction) and penalise cases where the signs are different (misclassification).

Entropy and Gini index are commonly used measures of impurity of nodes in a decision tree [120]. For binary classification, the two measures are defined in Equation 2.16 and Equation 2.17, respectively,

$$Entropy = 1 - \sum_{c}^{2} p_c \log_2 p_c \qquad \text{Equation 2.16}$$

$$Gini = 1 - \sum_{c=1}^{2} p_c^2 \qquad \text{Equation 2.17}$$

where p_c is the fraction of elements with the label c. This is used in Random Forest models to decide the best split (more details in 2.4.4).

For binary classification, a confusion matrix is a useful tool to analyse the performance of the model (Table 2.1). Based on the confusion matrix, several performance measures can be defined.

$$precision = \frac{TP}{TP + FP} \qquad \text{Equation 2.18}$$

$$recall = \frac{TP}{TP + FN}$$ Equation 2.19

The same ratios can also be constructed for the negative class. The overall performance can be described using accuracy.

$$accuracy = \frac{TN + TP}{TP + FP + TN + FN}$$ Equation 2.20

In cases where the number of positives and negatives is not similar, the F-score may be used which focuses on how well the positive class is predicted. The F1 score ($\beta = 1$) is the harmonic average of recall and precision. If more emphasis needs to be placed on one of the measures, an appropriate β may be selected such that $\beta \in \mathbb{R}^+$ [121].

$$F_\beta = (1 + \beta^2) \frac{1}{\beta^2 \frac{1}{recall} + \frac{1}{precision}}$$ Equation 2.21

Table 2.1: Confusion matrix

		Predicted	
		Negative	**Positive**
Actual	**Negative**	True negative (TN)	False positive (FP)
	Positive	False negative (FN)	True Positive (TP)

2.4.4 Random Forest (RF)

2.4.4.1 Algorithm description

Random Forest (RF) is a machine learning algorithm constructed from an ensemble of decision trees [122]. Each decision trees makes an independent prediction based on the input feature vector ($\boldsymbol{x}$). The predictions made by the decision trees are pooled together to generate a single prediction. In the original publication of the algorithm, this was achieved by a majority vote where each decision tree has a single vote [122]. In the implementation used in the thesis, the probabilities for each label are summed across the entire forest and the label with the highest total value is taken as the prediction of the RF model [123]. In summary, the prediction is made using a pooling function on M decision trees ($h_i(\boldsymbol{x})$);

$$\hat{y} = f_{pool}(\{h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \ldots, h_M(\boldsymbol{x})\}) \qquad \text{Equation 2.22}$$

2.4.4.2 Training methodology

For each tree, a subset of the training dataset is selected randomly and independently, typically using the bootstrap method. In the case of the bootstrap method, a predefined number of samples (n) is selected from the dataset with replacement. At each node of each tree, a subset of features is selected based on which the node is split. All possible split points are considered, and for each, the impurity of the split is computed according to Gini (Equation 2.16) or entropy (Equation 2.15) impurity measures[124]. The best split point is selected, based on the largest reduction in impurity compared to the parent node. The process is repeated for each of the two child nodes. The process terminates when: (1) the child nodes are pure (i.e. the node only contains samples of a single class), (2) the number of elements in the node is smaller than a predefined number, or (3) the maximum depth of the tree is reached. Once the set number of trees are trained, the training of the random forest model is complete.

2.4.5 Support Vector Machine (SVM)

2.4.5.1 Algorithm description

In this thesis, Support Vectors Machines (SVM) are used for a classification task (Chapter 5). An SVM classifier constructs a hyperplane (decision boundary) to divide a multidimensional feature space, into two classes (above and below) [53,119,125]. The dimensionality of the feature space corresponds to the number of features with potentially additional dimensions due to the transformation of the features (kernel trick). The decision boundary is defined by the weight vector (**w**) and the bias term b.

$$\boldsymbol{w}\,\boldsymbol{x} + b = 0 \qquad \text{Equation 2.23}$$

Figure 2.11 (A and B) illustrates how a decision boundary can be used to linearly separate examples. Margins (dotted lines in the figure) are parallel to the decision boundary, constructed using data points closest to the boundary (the support vectors), indicate how well the classes are divided.

However, not all data is linearly separable. In such cases, the data can be transformed into a linearly separable form. This is illustrated in Figure 2.11C for 1-dimensional feature space. By adding another dimension, where$\varphi\left(x_{(1)}\right) = \sin\left(x_{(1)}\right)$, the two classes can be easily separated by a linear decision boundary (Figure 2.11D).

The prediction is made using the transformed feature vector and the decision boundary (Equation 2.23),

$$\hat{y} = \boldsymbol{w}\,\varphi(\boldsymbol{x}) + b \qquad \text{Equation 2.24}$$

A label is assigned based on whether the output of the function is positive or negative.

In Equation 2.24, only the dot product of the transformed feature vector and the weight vector needs to be computed [119]. The mapping transformation from the original

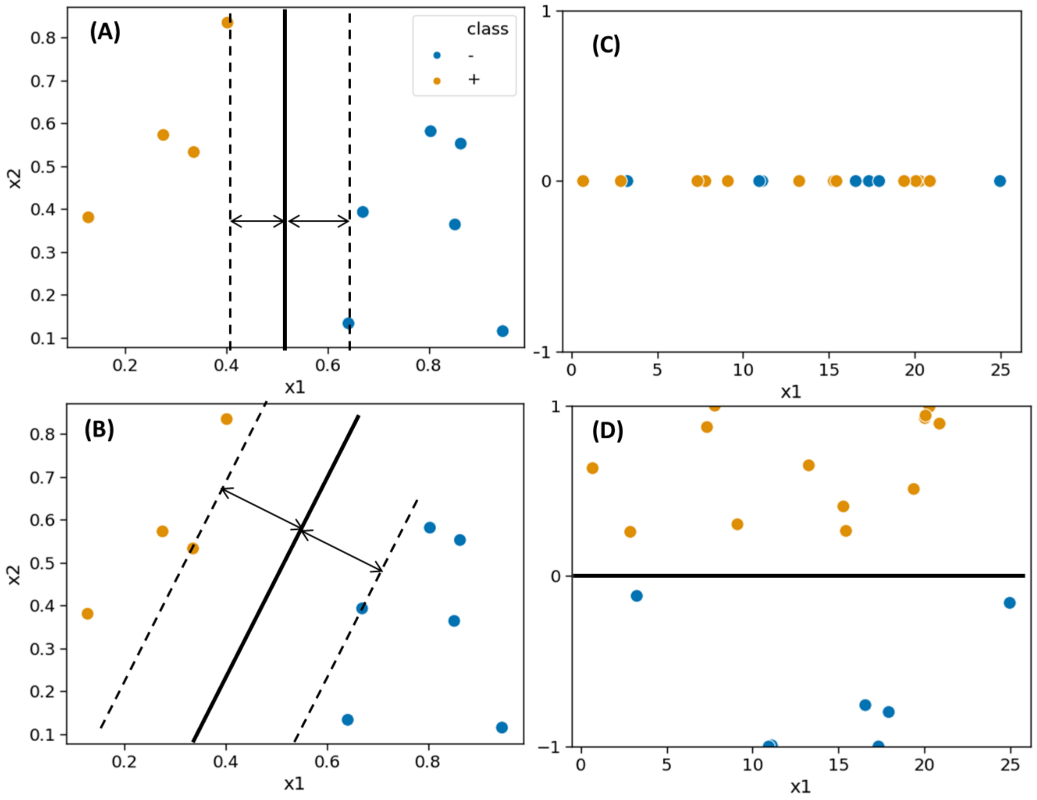


Figure 2.11: support vector machine classification

A possible decision boundary for separating the two classes (A). Another decision boundary that increases the separation between the two classes (B). Kernel trick can be used to transform a linearly inseparable points (C). In the example in the figure, passing the x_1 feature through a sine function transforms the data into linearly separable distribution (D).

features and the transformed feature vector does not need to be known. Rather, the dot product can be computed using a kernel function requiring vectors in the original descriptor space as input. This is referred to as the kernel trick.

2.4.5.2 Training methodology

In this thesis, SVM is only used as a binary classifier, so only the training procedure for binary classification is presented. Training of an SVM consists of finding a decision boundary (hyperplane) that separates the two classes (positive and negative represented by 1 and -1 respectively). The feature vectors (of size d) can be projected onto d-dimensional space where the (d-1)-dimensional decision boundary can be defined. This is illustrated for a feature vector of length of 2 in Figure 2.12 (the three points that lie on the dotted lines are the support vectors). The aim of training is twofold:

1) Minimise the amount of misclassifications – performance
2) Maximise the margins that separate the two classes – avoid overfitting

The cost function (Equation 2.9) of SVM can be written as the following.

$$C(\boldsymbol{w}, b) = C_{margin}(\boldsymbol{w}, b) + K\ C_{class}(\boldsymbol{w}, b) \qquad \text{Equation 2.25}$$

K is a hyperparameter responsible for determining the relative importance of minimisation of misclassification and maximisation of the margins. It is usually denoted as C, however to avoid confusion with the cost function, K is used here. Hinge loss (Equation 2.15) is commonly used for C_{class} [126].

The aim 2 (maximisation of margin) can be expressed as the following equation.

$$w = \underset{w}{\operatorname{argmax}} \left(\min_{i} \gamma_i \right) \qquad \text{Equation 2.26}$$

where γ_i corresponds to the margin in the ith dimension. The two lines that define the margin (dotted lines in Figure 2.12) can be defined as follows.

$$\boldsymbol{w}\,\boldsymbol{x} + b = 1 \qquad \text{Equation 2.27}$$

$$\boldsymbol{w}\,\boldsymbol{x} + b = -1 \qquad \text{Equation 2.28}$$

Let **x**$_1$ and **x**$_2$ denote two points that lie on the two margin lines. The following relationship can be constructed.

$$\boldsymbol{x_1} = \boldsymbol{x_2} + 2\gamma\left(\frac{\boldsymbol{w}}{|\boldsymbol{w}|}\right) \qquad \text{Equation 2.29}$$

Using Equation 2.27, Equation 2.28, and Equation 2.29, an expression for the width of the margin (γ) can be derived.

$$\boldsymbol{x_1} = \boldsymbol{x_2} + 2\gamma\left(\frac{\boldsymbol{w}}{|\boldsymbol{w}|}\right)$$

$$\boldsymbol{w}\left(\boldsymbol{x_2} + 2\gamma\left(\frac{\boldsymbol{w}}{|\boldsymbol{w}|}\right)\right) + b = 1$$

$$(\boldsymbol{w}\,\boldsymbol{x_2} + b) + \ 2\gamma\,\boldsymbol{w}\left(\frac{\boldsymbol{w}}{|\boldsymbol{w}|}\right) = 1$$

$$2\gamma\,\boldsymbol{w}\left(\frac{\boldsymbol{w}}{|\boldsymbol{w}|}\right) = 2$$

$$\gamma = \ \frac{|\boldsymbol{w}|}{\boldsymbol{ww}}$$

$$\gamma = \frac{1}{|\boldsymbol{w}|} \qquad \text{Equation 2.30}$$

Therefore, the margins can be maximised by minimising$|\boldsymbol{w}|$, or more conveniently, by minimising $\frac{1}{2}|\boldsymbol{w}|^2$. The cost function component associated with the aim 2 of SVM training (C$_{\text{margin}}$) can be calculated by

$$C_{margin}(\boldsymbol{w}) = \frac{1}{2}|\boldsymbol{w}|^2 \qquad \text{Equation 2.31}$$

The full cost function (Equation 2.25) can thus be expressed by substituting the respective cost function components (Equation 2.15 and Equation 2.31) as follows

$$C(\boldsymbol{w}, b) = \frac{1}{2}|\boldsymbol{w}|^2 + K\sum_{i=1}(1 - y_i f(\boldsymbol{x_i}))_+ \qquad \text{Equation 2.32}$$

Gradient-based optimisation (see 2.4.6.2) can be used to optimise w, b for the given set of hyperparameters.

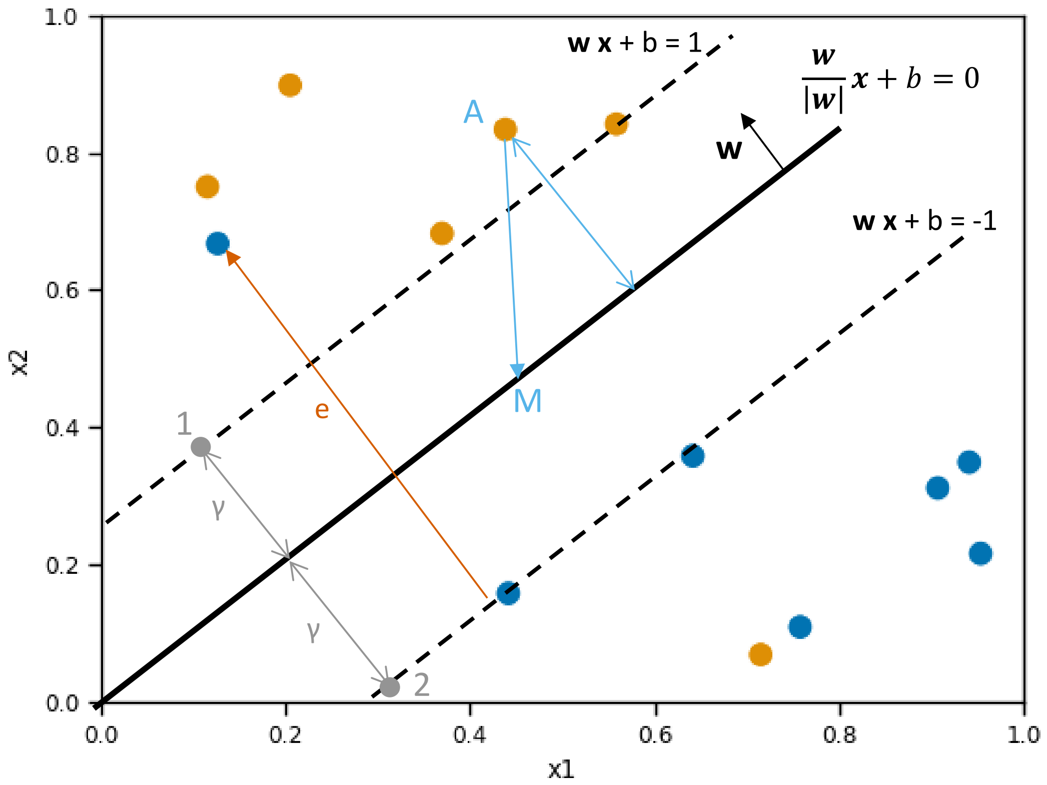


Figure 2.12: Training support vector classification.

The decision boundary (black) and margins (black dotted) separate two classes (gold – positive, dark blue – negative). Point A and M used for calculation of confidence of prediction (blue). Point 1 and 2 used for margin maximisation (grey). Handling of misclassification illustrated using e (orange).

2.4.6 Neural Networks (NN)

2.4.6.1 Algorithm description

First proposed in 1943, artificial neural networks are a type of function approximators that are inspired by the nervous activity of animal brains [127]. The term Neural Networks (NN) and Artificial Neural Networks are used interchangeably. Similarly to their biological counterparts, artificial neural networks are composed of neurons. The basic functionality of a neuron is expressed mathematically in Equation 2.33 and graphically in Figure 2.13 .

$$a = f(\boldsymbol{wp} + b) \qquad \text{Equation 2.33}$$

The input vector (**p**) is usually either the input to the whole network (**x**) or the output of the previous layer ($\mathbf{a}_{\mathbf{k-1}}$). It is multiplied by the weight vector (**w**) where each element (w_i) represents the sensitivity of the neuron to the respective input (p_i). A bias term (b) is usually added to allow 0 values in **p** to have an influence on the neuron. The resulting scalar is then passed through an activation function which enables the output (a) to have a non-linear relation to the input.

In principle, any differentiable function can be used as the activation function. Table 2.2 summarises the activation functions used in this thesis, along with some other commonly used functions. Rectified Linear function (ReLu) is the most commonly used activation function in deep neural networks due to its computational efficiency and reduced chance of encountering the vanishing gradient problem in multi-layered neural networks [128].

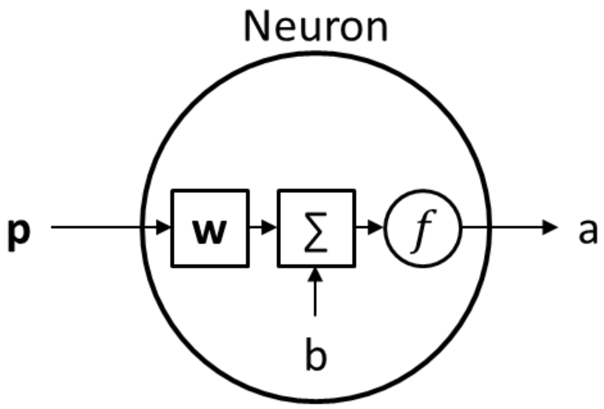
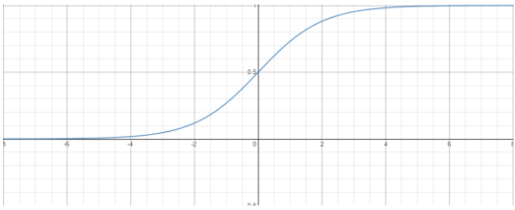
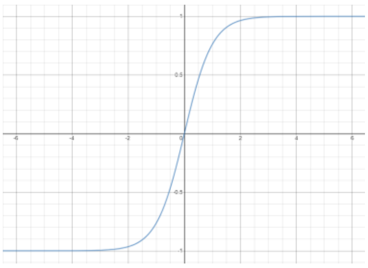
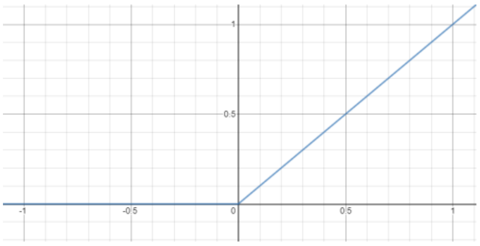


Figure 2.13: Simple neuron design.
Vector input (p), is multiplied by a weight vector (w), a bias term (b) is added, and the result is passed through an activation function (*f*) which results in a scalar output (a).

Table 2.2: Activation functions commonly used in neural networks [130]

Name	**Function**	**Visualisation**
Sigmoid	$y = \frac{1}{1 + e^{-x}}$	
Tanh	$y = \tanh(x)$	
ReLU	$y = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$	

The neurons form the basic unit of neural networks. More sophisticated units can be developed and arranged in a specific manner to suit a specific task (such as Long Short Term Memory [129]). The neurons are usually arranged in layers, where each may contain one or more neurons. Three main classes of the arrangement are shown in Figure 2.14: fully connected, convolutional, and recursive [130].

In a fully connected layer (Figure 2.14 top left), each neuron is fed all the available inputs (either model inputs or outputs of the previous layer). Each of the neurons has a different weight vector ($\mathbf{w}_\mathbf{j}$), and so the weights of a fully connected layer can be expressed as a single weight matrix ($\mathbf{W}_\mathbf{k}$). For a layer k, containing S neurons, the weight matrix can be expressed as shown in Equation 2.34.

$$W_k = \begin{bmatrix} w_{1k} \\ \vdots \\ w_{jk} \\ \vdots \\ w_{Sk} \end{bmatrix} \qquad \text{Equation 2.34}$$

This type of layer is the most common in neural networks and its main advantage is that it utilises all available inputs.

A convolutional layer (Figure 2.14 top right) consists of neurons that have a restricted *receptive field* (i.e. can only "see" a subset of all inputs). Inputs within the *receptive field* are fed into the neuron to compute a single value output. The *receptive field* is then shifted by a predetermined amount (movement in Figure 2.14 to the right), and the action is repeated. In case part of the *receptive field* falls outside of the input vector, padding with zeroes is sometimes used to ensure the dimensionality of the instance

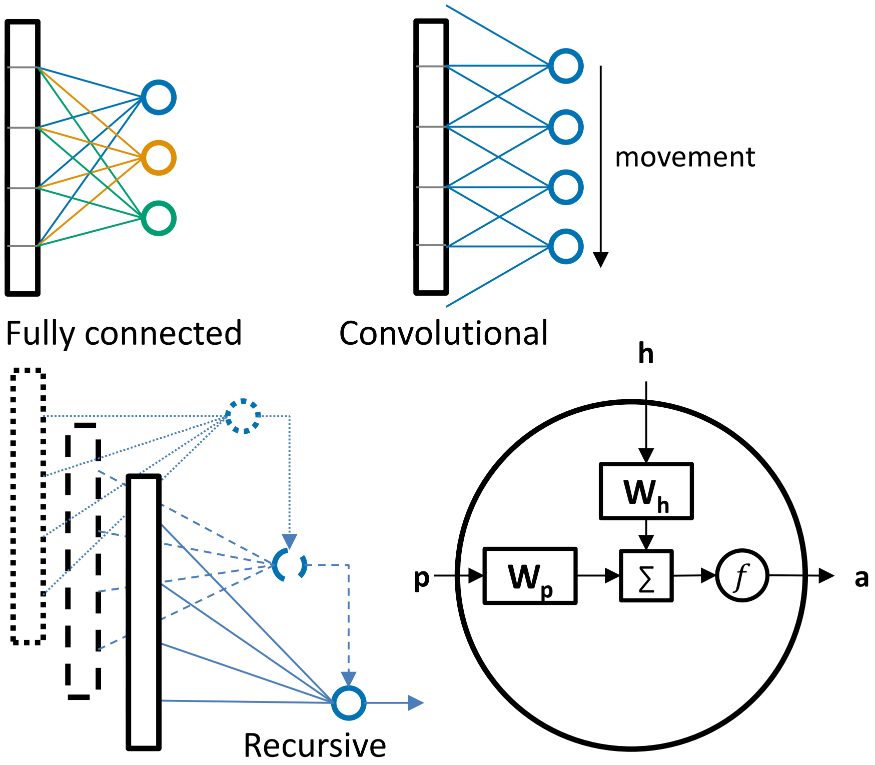


Figure 2.14: Examples of fully connected, convolutional, and recursive layers of neural networks, along with a schematics of a simple recursive neuron.

Weights of each neuron are different and are connected to every input in case of a fully connected layer (top left). In a convolutional layer, a neuron with same weights is applied to a subset of the input (top right). All of the inputs and the previous hidden state (h) are fed into a recursive neuron (bottom left). A simple recursive neuron design (bottom right).

input is consistent. This type of layer is used to reduce the number of connections within a network. It is also used to capture information where there is value in the segment of the input regardless of where it appears, such as a curved line when attempting to classify hand-written digits [131]. Convolutional layers are not used in this thesis. A more detailed explanation of this type of layer can be found elsewhere [128,130].

In the case of a recursive layer (Figure 2.14 bottom left), the neuron is connected to the input, but also to itself. For this reason, the basic neuron introduced in the Figure 2.13 needs to be modified to handle two inputs (Figure 2.14 bottom right). The state that is passed from the neuron to itself is referred to as the hidden state ($\boldsymbol{h_t}$). The dimensions of $\boldsymbol{h}$, $\boldsymbol{p}$, and $\boldsymbol{a}$ are the same to ensure that the neuron can be applied recursively. The equation describing the operation of a simple recursive neuron is shown in Equation 2.35 [130].

$$\boldsymbol{a} = f\left(\boldsymbol{W_h h} + \boldsymbol{W_p}\, \boldsymbol{p}\right) \qquad \text{Equation 2.35}$$

The first time a recursive neuron is applied, an initial hidden state (**h**$_0$) needs to be given. The order in which the input vectors (**p**) are fed is dependent on the structure of the data. In a special case where the inputs are fed sequentially, the layer is referred to as recurrent.

The activation function used in these layers may be different for each neuron. However, it is common to see the same activation function being used for all neurons of layers of a particular type. Different types of layers are sometimes combined within the same neural network. The output of one layer becomes the input of the following layer; hence the neural network can be expressed as a composite function of functions for each layer

$$\boldsymbol{y} = \left(f_{L(N)} \circ f_{L(N-1)} \circ \ldots \circ f_{L(1)}\right)(\boldsymbol{x}) \qquad \text{Equation 2.36}$$

where $f_{L(i)}$ is the function of the *i*th layer of the network and contains the required weight multiplication and activation as exemplified in Equation 2.33 and Equation 2.35. Neural Network design is a substantial field, so only the architectures relevant to the thesis are discussed further in Chapter 7. The work in the chapter utilises a recurrent layer for graph embedding (as introduced in 2.4.1.2) and fully connected layers for predictions.

2.4.6.2 Training methodology

Neural Network design often results in models with many parameters (θ) that require tuning for the model to make meaningful predictions. The training of a Neural Network is usually carried out by iteratively updating the parameters based on the gradients of the loss (∇C) until the predefined convergence criteria is met. This approach is commonly referred to as gradient descent. However, in this thesis, the term gradient descent is used exclusively for the simplest form of the algorithm (computation of the full gradient and update with constant learning rate; explained in more details below) while the whole family of algorithms are referred to as gradient-based optimisation (GBO) algorithms [132]. The algorithm A.1 describes the basic components of GBO algorithms. The initial set of parameters θ_0 is determined by an initiation function (f_{init}) and the hyperparameters of the model (λ_{init}). Some initiation strategies are discussed below. The training is initiated with the L being set to arbitrary, large number. Predictions are made using the initial set of parameters and the respective loss is calculated using the cost function C. Although loss is a function of θ parameterised by the hyperparameters λ (as defined in Equation 2.9), for the sake of clarity, λ is omitted in the description as these remain constant for the duration of training. The differences in algorithms within the GBO family arises from the differences in implementation of gradient computation ($f_{grad}(L, \theta_t)$) and the parameter update strategy ($f_{update}(\theta_t\ , \nabla C)$). The process is repeated until the convergence criteria (f_{conv}) is met.

Algorithm A.1: Gradient-based optimisation

```
$\theta_0 \leftarrow f_{init}(\lambda_{init})$
$t \leftarrow 0$
$Loss \leftarrow +\infty$
while $f_{conv}(C, t) = False$ do
    $L \leftarrow C(\theta_t)$
    $\nabla C \leftarrow f_{grad}(L, \theta_t)$
    $\theta \leftarrow f_{update}(\theta_t, \nabla C)$
    $t \leftarrow t + 1$
```

For many neural networks, the initial parameters (weights and biases) have an effect on how well the network can be trained [133–135]. One generic weight initialisation technique is to generate a random weight matrix that is orthogonal (i.e. $\boldsymbol{W}^T = \boldsymbol{W}^{-1}$) [136]. A number of initialisation strategies were developed according to the design of the network; particularly based on the activation functions used. The weights are randomly selected from a truncated normal distribution with a mean of 0 and variance according to,

$$\sigma^2 = \frac{m}{n_{in} + k\, n_{out}} \qquad \text{Equation 2.37}$$

where n_{in} and n_{out} are the sizes of the input and output vectors to the layer, *m* is 1 or 2, and *k* is 0 or 1 depending on the initiation strategy. The values of *m* and *k* for different

Table 2.3: coefficients to various initialisation strategies.

Name	***m***	***k***
Lecun [134]	1	0
He [264]	2	0
Glorot[133]	2	1

initialisation techniques are presented in Table 2.3. A robust comparison of the initialisation techniques can be found elsewhere [135].

The GBO methods require partial derivatives of the loss with respect to each parameter. For a model with M parameters, the gradients are defined as

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial \theta_1} \\ \vdots \\ \frac{\partial C}{\partial \theta_i} \\ \vdots \\ \frac{\partial C}{\partial \theta_M} \end{bmatrix} \qquad \text{Equation 2.38}$$

Neural networks are differentiable composite functions as defined in Equation 2.36. Therefore, the partial derivatives with respect to each parameter can be computed using the chain rule. This is computationally expensive and gives rise to several key issues which are explored below.

Firstly, all loss functions defined in 2.4.3 involve summation of individual loss over the entire dataset. This is often computationally expensive so the loss may be computed for a subset of the dataset (single data point or multiple data points). Although this is only an approximation of the true gradient, the reduction in the computational cost is large enough to warrant the use of this method [132,137,138]. As the subsets are often selected at random, the method is referred to as Stochastic Gradient Descent (SGD).

Another issue arises if the Neural Network has many layers or a recursive layer. In a recursive network, the input at timestep *t* depends on outputs for all previous timesteps (0, 1, …, *t*-1). When *t* is a large number, the differential becomes long, potentially resulting in the “vanishing gradient” problem [139]. However, no more than four timesteps of a recursive layer were used in this thesis (Chapter 7), so no measure to counter this was necessary.

The computed gradients are then used to update each of the parameters of the model. The basic equation for this is given below, where $\boldsymbol{\theta}_{(t+1)}$ are the updated parameters and η is the learning rate.

$$\boldsymbol{\theta}_{(t+1)} = \boldsymbol{\theta}_{(t)} - \eta \nabla \mathrm{C} \qquad \text{Equation 2.39}$$

The learning rate controls how much the weights are modified by and is predetermined in case of GD and SGD. However, selection of the appropriate learning rate is often challenging; too small and the optimisation will take a long time, too large and the optimisation will not find a stable minima [132]. In theory, an adjustable learning rate that decreases as it approaches minima would address this issue. Several methods that attempt to achieve this have been developed such as Adagrad [140] and Adam [137]. Adam (Adaptive Moment Estimation) optimiser is de facto the standard optimisation procedure in Neural Network training [132]. It provided adequate convergence performance in Chapter 7 so no alternatives were used.

Adam uses two variables ($\boldsymbol{m_t}, \boldsymbol{v_t}$), parameterised by $\beta_1, \beta_2 \in [0,1)$ to update model parameters ($\boldsymbol{\theta}$) [137]. The two variables are decaying averages of gradients and squared gradients, defined as

$$\boldsymbol{m_t} = \beta_1 \boldsymbol{m_{t-1}} + (1 - \beta_1)\boldsymbol{\nabla \mathrm{C}_t} \qquad \text{Equation 2.40}$$

$$\boldsymbol{v_t} = \beta_2 \boldsymbol{v_{t-1}} + (1 - \beta_2)\boldsymbol{\nabla \mathrm{C}_t}^2 \qquad \text{Equation 2.41}$$

The terms ($\boldsymbol{v_t}, \boldsymbol{m_t}$) are biased towards the initiation values ($\boldsymbol{v_0}, \boldsymbol{m_0}$), usually set to 0. A bias-correction is applied before the weights are updated.

$$\boldsymbol{m_t'} = \frac{\boldsymbol{m_t}}{1 - \beta_1^t} \qquad \text{Equation 2.42}$$

$$\boldsymbol{v_t'} = \frac{\boldsymbol{v_t}}{1 - \beta_2^t} \qquad \text{Equation 2.43}$$

Finally, the model parameters are updated according to the following equation.

$$\boldsymbol{\theta}_{(t+1)} = \boldsymbol{\theta}_{(t)} - \frac{\eta}{\sqrt{\boldsymbol{v'_t}} + \epsilon} \boldsymbol{m_t'}$$

The ϵ term is a small number to ensure that the denominator is not zero (usually of the order of 10^{-8}). Multiplication between vectors in this equation is done element-wise where each element corresponds to a parameter of the model [137]. The training procedure is continued until the convergence criteria is met which is chosen before the training is initiated. This is usually defined as a number of consecutive iterations with loss function reduction below a specified amount. In the thesis, this is specified when training procedure is discussed (Chapter 7). The three methods introduced in this section are summarised in Table 2.4.

Table 2.4: Comparison of the three learning algorithms.

	GD	**SGD**	**Adam**
Gradient computation	Full gradient	subset	Subset
Algorithm hyperparameters	η	η	η β_1, β_2 ϵ
Model parameter update	$\boldsymbol{\theta}_{(t+1)} = \boldsymbol{\theta}_{(t)} - \eta \boldsymbol{\nabla}\mathbf{C}$	$\boldsymbol{\theta}_{(t+1)} = \boldsymbol{\theta}_{(t)} - \eta \boldsymbol{\nabla}\mathbf{C}_{\boldsymbol{subset}}$	$\boldsymbol{\theta}_{(t+1)} = \boldsymbol{\theta}_{(t)} - \frac{\eta}{\sqrt{\boldsymbol{v'}_{\boldsymbol{t}}} + \epsilon} \boldsymbol{m}'_{\boldsymbol{t}}$
comment	Slow computation of full gradient	Learning rate unadjustable	Adjusted update rate per parameter

2.4.7 Hyperparameter optimisation

Most machine learning algorithms require some hyperparameters that can greatly affect the performance of the model [141]. The purpose of the hyperparameters is to control the balance between under- and over-fitting [141]. The principles of hyperparameter optimisation are similar to the training procedure discussed in the training sections of each introduced algorithm. $\mathcal{H}_{\lambda}^{*}$ denotes an empirical model based on a learning algorithm $\mathcal{A}$, optimised with respect to its parameters on a training set D_{tr} with a given set of hyperparameters λ.

$$\mathcal{H}_{\lambda}^{*} = \mathcal{A}(D_{tr};\ \lambda) \qquad \text{Equation 2.44}$$

The cost function with respect to the hyperparameter set (λ) can be expressed similarly to Equation 2.8;

$$C(\lambda) = \mathcal{L}(D_{v};\ \mathcal{H}_{\lambda}^{*}) \qquad \text{Equation 2.45}$$

The same loss function $\mathcal{L}$ can be used as with training. Alternatively, any quantitative performance measure (2.4.3) can also be used. In cases such as with Gaussian Process Regression, hyperparameter inference can be accomplished during training [142]. However, such algorithms were not used in the thesis, thus fall out of scope. The hyperparameter optimisation task can thus be expressed in an analogous way to the training task[141];

$$\lambda^{*} = \underset{\lambda}{\operatorname{argmin}}\, \mathcal{L}(\ D_{v}, \mathcal{A}(D_{tr}, \lambda)) \qquad \text{Equation 2.46}$$

A key difference between hyperparameter tuning (Equation 2.46) and training (Equation 2.8) is the computation cost. $\mathcal{A}(D_{tr}, \lambda)$ is an optimisation problem in itself and can be expensive to compute [141]. Furthermore, many of the models have complex, conditional hyperparameter search spaces [141]. This is particularly true in the case of Neural Networks where the number of neurons in a layer as well as the number of layers may be hyperparameters [143]. Gradients are usually not available for hyperparameter optimisation so the GBO methods discussed in 2.4.6.2 are not suitable.

The most basic approach is to tune the hyperparameters manually, using rule of thumb and experience. However, this can be tedious and is often outperformed by the methods presented below [113,141]. These methods can be divided into informed and uninformed approaches. In the thesis, uninformed methods are defined as an approach where the choice of subsequent hyperparameters is not affected by the previous step. Grid search and random search falls into this category and are elaborated upon in 2.4.7.1. On the other hand, in informed methods the set of hyperparameters proposed at each iteration is based on the previous steps. This is a hot topic in machine learning research and several methods have been developed; these are presented in 2.4.7.2. An informed method used in Chapter 7, Sequential model-based optimisation (SMBO), is explained in more details in 2.4.7.3.

2.4.7.1 Uninformed methods

To perform a grid search, a search space of hyperparameters consisting of discrete values is constructed. A combination of the hyperparameters is used for training of the model and its performance on the validation set is recorded. The process is repeated for all combinations of the hyperparameters and the combination that yields a model with the best performance is selected. This brute force approach is easy to set up and can be easily parallelised for reduced computation time [113].

However, the grid search does not provide even coverage of the entire search space as it focuses on the selected discrete values (Figure 2.15). For this reason, and particularly for high dimension search space, a random search often outperforms grid search [113]. In the case of random search, upper and lower boundaries for each hyperparameter are defined. The algorithm randomly samples the search space for a predefined number of iterations. The best performing hyperparameter combination is then selected. By not being limited by a defined grid, the algorithm is able to sample more varied values for each of the hyperparameters (Figure 2.15). The random search is often sufficient when hyperparameter optimising on relatively simple search spaces. The method is often used as a benchmark for assessment of the performance of the informed methods of hyperparameter optimisation [113].

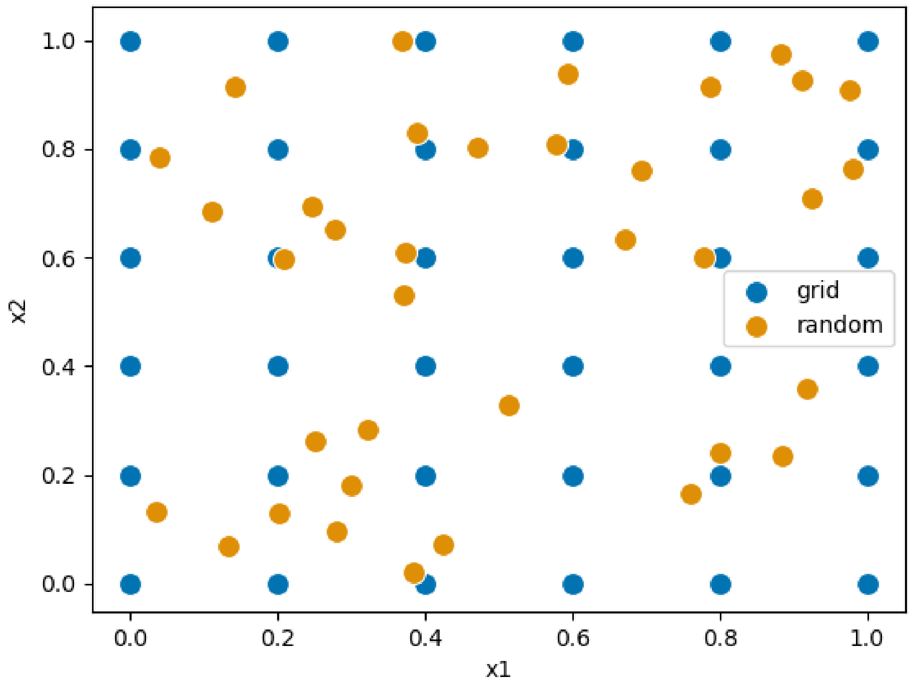


Figure 2.15: Sampling of a search space using Grid search and Random search.

In case of the grid method, the preselected values of each variable are sampled. The 36 samples only cover ten distinct values (0.0, 0.2,…1) of each variable. The same number of samples using the random search method cover 36 distinct values of each.

2.4.7.2 Informed approaches

The high computation cost of each iteration of the hyperparameter optimisation makes methods that can reduce the number of required steps highly desirable. The computational cost of selecting a new set of hyperparameters to try is negligible compared to the cost of the evaluation. As a result, a number of algorithms that attempt to find the global minimum of the hyperparameter search space were developed. Evolutionary algorithms are based on minor random mutations at each iteration from the best performing set of models from the previous iteration [143]. Particle swarm uses a set of "particles" (set of models trained with a set of hyperparameters) that traverse the search space [144]. The movement of each of these particles is determined by the location of its current best performing set of hyperparameters and that of the entire swarm as well as a random component. Bayesian optimisation constructs a surrogate function that predicts the performance of the trained model for a given set of hyperparameters [145]. The subsequent combination is selected based on the balance of exploration and exploitation. An exploration step is taken to improve the accuracy of the surrogate function while exploitation attempts to find the minima based on the current surrogate function. Algorithms that create a surrogate function that is sequentially updated are collectively termed Sequential Model Based Optimisation (SMBO) algorithms.

2.4.7.3 Sequential model-based optimisation (SMBO)

SMBO algorithms were originally developed for experiment design and oil exploration work. In both of these applications, the computation of the performance (i.e. yield of a reaction by doing the experiment or test drilling to measure the amount of oil in an area) is significant; hence the algorithm attempts to minimise the number of steps required to reach the optimum value [146]. The applicability of these methods to hyperparameter optimisation is evident based on the similarity of the challenges of the tasks. Four elements need to be defined for the SMBO algorithms: hyperparameter search space, objective function (Equation 2.45), surrogate model, and acquisition function (Figure 2.16).

The definition of the hyperparameter search space is performed similarly to the other methods introduced. Unlike the grid search method, each variable can be defined using a truncated distribution function (Gaussian, uniform). Conditional variables can also be supported by SMBO algorithms [141,146]. Conditional variables refer to variables that only exist when a specific condition is met; for example, the number of neurons in the second layer of a Neural Network is only a valid hyperparameter when the number of layers (another hyperparameter) is two or higher.

The acquisition function is used to determine the following hyperparameter combination to test which reflects the relative importance of exploration and exploitation. The hyperparameters are chosen at random based on a distribution defined by the acquisition function. The general form of the acquisition function can be defined as,

$$f_{acq}(\lambda; C^*) = \int_{-\infty}^{\infty} u(\lambda; C^*)\, p_M(C|\lambda)\, dC \qquad \text{Equation 2.47}$$

where C denotes the cost function for the set of hyperparameters λ as defined in Equation 2.45, and C^* denotes the baseline value of the cost function [147]. The baseline value C^* can be the lowest observed value so far or some other defined value. The $u(\lambda; C^*)$ is a utility function and the $p_M(C|\lambda)$ corresponds to the probability of obtaining performance C given set of hyperparameters λ based on the surrogate model M. Probability of improvement (PI)[148] is one of the possible acquisition functions that can be used and the corresponding utility function is defined as

$$u(\lambda) = \begin{cases} 0 & C(\lambda) > C^* \\ 1 & C(\lambda) \leq C^* \end{cases} \qquad \text{Equation 2.48}$$

Expected improvement (EI)[148] is another acquisition function for which the utility function is defined as follows.

$$u(\lambda) = \begin{cases} 0 & C(\lambda) > C^* \\ C^* - C(\lambda) & C(\lambda) \leq C^* \end{cases} \qquad \text{Equation 2.49}$$

The main difference between the two utility functions is the value of reward for finding a value below the benchmark value (C*). In the case of PI, the same reward is given regardless of the size of improvement, which may lead to overexploitation of a local minima found [148]. The reward for EI is scaled based on the size of improvement. EI is used in this thesis (Chapter 7) for which the acquisition function can be written as follows (substitution of Equation 2.49 into Equation 2.47).

$$EI(\lambda; C^*) = \int_{-\infty}^{C^*} (C^* - C) p_M(C|\lambda)\, dC \qquad \text{Equation 2.50}$$

Other acquisition functions such as entropy-based functions have also been developed [149].

The difference in SMBO algorithms arises from the different approaches to constructing the surrogate model (Figure 2.16). Sequential Model-based Algorithm Configuration (SMAC) uses a random forest for the construction of the surrogate function [150]. The Hierarchical Gaussian Process (HGP) approach uses Gaussian Processes which are updated at each iteration [147]. The method used in this thesis (Chapter 7) is the Tree-structured Parzen Estimator (TPE) approach, details of which are presented below. A comparison of SMAC and TPE can be found here [151], while comparison between HGP and TPE is presented here [147].

HGP and other Gaussian Process-based approaches model $p_M(C|\lambda)$ directly, but the TPE method instead models $p_M(\lambda|C)$ via two kernel density estimates (KDE) as follows. The term p_{M} is abbreviated to p hereafter for clarity.

$$p(\lambda|C) = \begin{cases} \ell(\lambda) & C(\lambda) < C^* \\ g(\lambda) & C(\lambda) \geq C^* \end{cases}$$ Equation 2.51

The two KDE functions are constructed from the predefined hyperparameter distributions and the previous observations of pairs of λ and $C(\lambda)$ [147]. A quantile γ can be defined that corresponds to the probability of $C(\lambda) < C^*$.

$$\gamma = p(C < C^*)$$ Equation 2.52

For TPE, Bayes' Theorem is used to modify Equation 2.50 to give

$$EI(\lambda; C^*) = \int_{-\infty}^{C^*} (C^* - C) \frac{p(\lambda|C)p(C)}{p(\lambda)}\, dC$$

$$= \frac{\int_{-\infty}^{C^*} (C^* - C) p(\lambda|C) p(C)\, dC}{p(\lambda)}$$ Equation 2.53

The denominator of Equation 2.53 can be expressed using the Law of Total Probability, Equation 2.51, and Equation 2.52 as follows.

$$p(\lambda) = \int_{-\infty}^{\infty} p(\lambda|C)p(C)dC =$$

$$= \int_{-\infty}^{C^*} p(\lambda|C)p(C)dC + \int_{C^*}^{\infty} p(\lambda|C)p(C)dC$$

$$= \gamma\ell(\lambda) + (1 - \gamma)g(\lambda)$$ Equation 2.54

The integral in the nominator of Equation 2.53 is within the range where $p(\lambda|C) = \ell(\lambda)$, hence it can be written as

$$\int_{-\infty}^{C^*} (C^* - C) p(\lambda|C) p(C)\, dC =$$

$$= C^*\ell(\lambda) \int_{-\infty}^{C^*} p(C)\, dC - \ell(\lambda) \int_{-\infty}^{C^*} C\, p(C)\, dC$$

$$= C^*\ell(\lambda)\gamma - \ell(\lambda) \int_{-\infty}^{C^*} C\, p(C)\, dC$$ Equation 2.55

Substituting Equation 2.54 and Equation 2.55 back into Equation 2.53 and rearranging to aggregate the terms affected by λ, the following equation is obtained [147]

$$EI(\lambda; C^*) = \left(\gamma + (1-\gamma)\frac{g(\lambda)}{\ell(\lambda)}\right)^{-1}\left(C^*\gamma - \int_{-\infty}^{C^*} C\, p(C)\, dC\right) \qquad \text{Equation 2.56}$$

As can be seen from Equation 2.56, the EI can be maximised by minimising the ratio $g(\lambda)/\ell(\lambda)$. In essence, this is achieved by probing areas with high probability of achieving a score above the target (high $\ell(\lambda)$), and low probability of scores below the target (low $g(\lambda)$). In a typical implementation of the algorithm, a number of samples are drawn from $\ell(\lambda)$ and each of the candidate hyperparameter combinations are evaluated by the ratio $g(\lambda)/\ell(\lambda)$. The λ with the highest EI is then selected to obtain $C(\lambda)$ according to Equation 2.45. The surrogate model is updated based on the new $\lambda, C(\lambda)$ pair and the process is repeated [147].

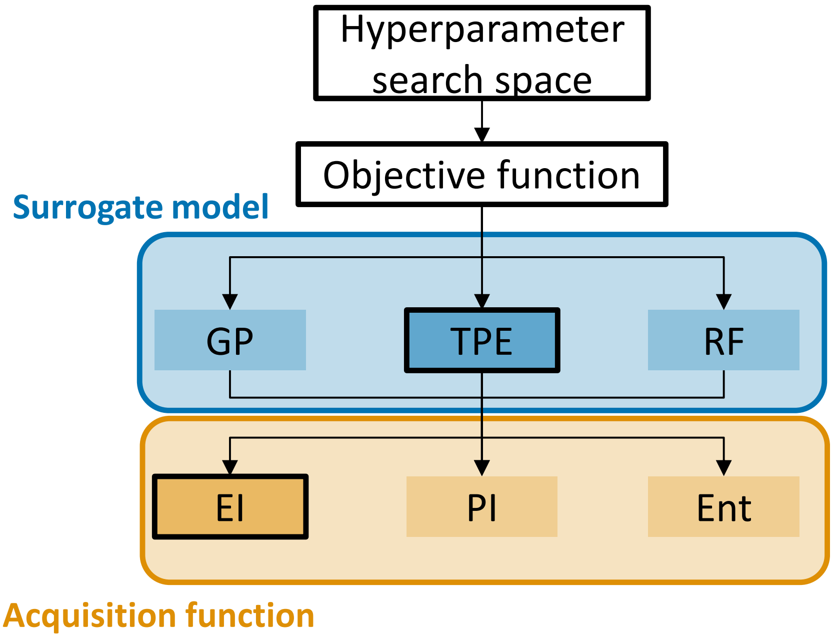


Figure 2.16: Overview of SMBO algorithms.

Several surrogate models such as Gaussian Process (GP), Tree-structured Parzen Estimator (TPE), and Random Forest (RF) are available. Expected Improvement (EI), Probability of Improvement (PI), and Entropy (Ent) are some of the possible acquisition functions. TPE and EI was used in the thesis.

2.4.8 Application of QSPR

QSPR methods are often used in situations where a theoretical chemistry approach is not suitable. Historically, such empirical models tended to be simple with very limited scope of applicability such as only hydrocarbons [152]. As the amount of data and modelling algorithms improve, more complicated relationships with ‘global’ applicability domain were mapped [53]. Although empirical models aim to be generalisable, the models cannot be expected to reliably extrapolate to instances which are not represented in the training data. Exact definition and estimation of the applicability domain remains an active area of research [153,154].

One of the earliest works of QSPR can be traced back to the mid-19th century with work by Hermann Kopp on the relationship between the molecule size and boiling point of alkanes [86,155]. Over the following decades, attempts were made to find relationships between descriptors of the molecular structure and other physical properties. In the 1930s, a melting point prediction model was published that used number of atoms and the density of the compound as descriptors [87]. Development of early descriptors can also be traced to this time period [156]. Cheminformatics approaches, and the notion that an empirical relationship between structures of molecules and their properties exists, faced many objections from other chemists [86]. However, these objections were shown to have been misplaced based on the development of modern QSPR models [6,7,10,40,53,157–161].

Work done in the 1960s is considered to be the origin of the modern QSPR methodologies [10,153,162–165]. QSPR models have been used extensively within the areas related to the pharmaceutical product development. Models mapping molecular structure biological activity, selectivity, and toxicity have been developed [166,167]. Likewise, predictions of solubility [45,70,98,157,168,169] and melting point [81,83,170–173] are also of considerable interest within the QSPR community.

The development of QSPR model has been facilitated by the increase in data availability. In the case of melting point, the datasets were limited to 10s of compounds in the early 20th century [87]. By the turn of the 21st century, models were typically trained on datasets in the 100s of data points [171,174,175]. The Patent Dataset (introduced in 2.3.3), which contains 289,379 datapoints, is the largest melting point dataset available at the time of writing [83].

The increase in dataset size allows more complex models with larger applicability domains. Early melting point models were often restricted to a specific class of molecules such as rigid, non-hydrogen bonding aromatics [171] or aliphatics with certain functional groups [175]. QSPR models that can be applied to a wider range of molecules were later developed [83,172]. This trend is expected to continue in the future as the dataset sizes increase and the available models become better at capturing complex relationships. The dataset size can be expected to represent a more diverse chemical space. The development of more complex algorithms has facilitated improvements in performance in other fields such as Natural Language Processing [176], so a similar trend can be expected to extend to QSPR research. The increase in computational resource efficiency (both the computation and the economic cost) has also been identified as one of the drivers of progress in the field of predictive modelling [177].

Solubility and melting points are both dependent on the molecular structure, but also the crystal structure. However, the crystal structure information is usually not available and not used. This is cited as one of the limitations of these QSPR models [172]. Work has been carried out investigating the extent to which incorporation of crystal descriptors improve the model performance [70,95,178]. No good evidence of significant improvement was observed. However, this may be attributed to limitations in the calculated crystal descriptors and uncertainty in the specific polymorph for which the experimental training set data were measured [70]. The issue of developing adequate descriptors of the solid state and their importance to accurate property prediction is addressed in Chapter 7.

2.5 Matched Molecular Pair Analysis (MMPA)

Matched Molecular Pair Analysis (MMPA) is a statistical method for studying the effects of molecular changes on a property of interest. Molecules that differ only by one chemical transformation are considered to be a Matched Molecular Pair (MMP). Changes in properties across these pairs, for a given transformation, are statistically analysed to infer the effects of the molecular transformations. The analysis procedure (2.5.1) and its application within cheminformatics (2.5.2) are discussed here.

2.5.1 Identification of pairs and analysis procedure

2.5.1.1 Terminology

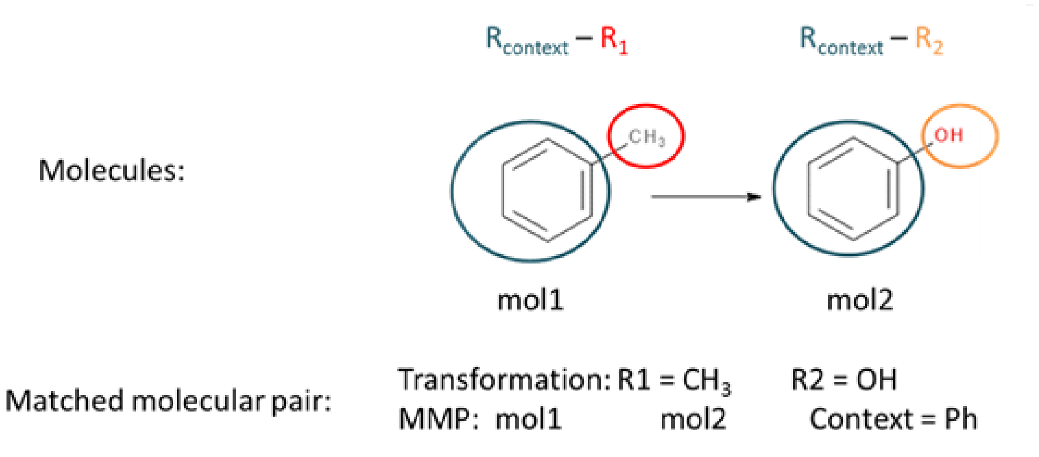


Figure 2.17: Example of a matched molecular pair.

MMPA is based on pairs of molecules that differ by one chemical transformation. An example of an MMP is toluene and phenol, as shown in Figure 2.17. In this case, the ‘transformation’ is the change from a methyl group (R1) to a hydroxyl group (R2). These two groups are also referred to as cores. A ring substitution (e.g. phenyl to Pyridyl group change) can also be considered to be a transformation. In the given example, phenyl group is named ‘context’ as it is the common molecular substructure across the two molecules. The two molecules are an MMP with transformation $–CH_3$ → –OH. For the analysis, other MMPs with the same transformation would be used to study the effect of substitution of a methyl group to hydroxyl group.

2.5.1.2 Identification

The simplest way of identification of MMPs is for a chemist to manually compare molecules. This method becomes unfeasible as the number of molecules increases. Two categories of automated methods for MMP identification have been developed; pre-specified transformation methods (STM) and unspecified transformation methods (UTM). STM can be useful in limiting the computational power required by narrowing down the search to only the transformations of interest [179]. However, this is also a weakness of the method – no new transformations can be identified. UTM identify transformations and MMPs from a given set of molecules.

UTM can be further subdivided into fragmentation [180], maximum common substructure (MCS) [181,182], and hybrid approaches [183,184]. UTM is used in this project in order to be able to identify transformations that have an impact on the properties studied (Chapter 6). In particular, the Hussain and Rea Fragmentation method (HRF) was selected as it is computationally efficient and can be easily implemented within the workflow. The details of the algorithm are presented in Chapter 3.

2.5.1.3 Analysis

Once the transformations of interest have been identified and all corresponding MMPs identified, the property change is calculated for each of the pairs. Each change becomes a single datapoint for the analysis. All changes are grouped by the transformation. For example, all MMPs where the transformation involves a methyl group being swapped for a hydroxyl group are grouped together to study the effects of that transformation on a property of interest. Statistical analysis is then performed to infer the effect of the transformation. Typically, the averages and paired t-test scores are calculated [179,185,186]. In many cases, the t-test is repeated multiple times (once for each transformation within the dataset). As such, measures need to be taken to account for multiple statistical testing [187,188]. The fraction of MMPs that have positive / negative effect on the property has also been used for the analysis [189]. In some cases the information about the context of the specific MMP is also included in analysis [190].

2.5.2 Application of MMPA

The MMPA framework can be applied to study any property of interest that is affected by molecular change. It has been widely used for properties that are relevant to the drug Discovery process [179]. Properties relating to ADMET (absorption, distribution, metabolism, excretion, and toxicity) such as aqueous solubility [191] and plasma protein[186] binding were studied using MMPA. Molecular fragment contributions to melting point, which is a property used in many solubility predictions (e.g. General Solubility Equation) were investigated using this method as well [192]. MMPA has also been used to study the effects of chemical transformations on binding to a particular biological receptors [193,194](such as CYP inhibition) as well as to study promiscuity [195] (the ability for a molecule to interact with several biological macromolecules). MMPA was demonstrated to be a versatile method (in terms of studied properties) which provides easily interpretable results that can be used by chemists during Discovery [11,55,196].

MMPA has also been used in tandem with other statistical approaches such as QSPR models. Two ways of combining QSPR with MMPA have been developed; QSPR-by-MMP [197] and prediction driven MMP [198]. QSPR descriptors were calculated for the chemical transformations (rather than individual chemical as it is the usual case for QSAR models). The developed model predicted the activity change for the chemical transformations. It was noted that for smaller sets of molecules, the number of well represented transformations in the training set was too low in many cases, limiting the model to the more commonly occurring transformations [197]. In case of the prediction driven MMP, a QSPR model was developed for a set of molecules. MMPA was carried out on the dataset using the predicted values [198]. Application of MMPA to the output of the QSPR model allowed for a more easily interpretable results; an increase or decrease in aquatic toxicity associated with a particular chemical transformation [198]. The study demonstrated that useful knowledge can be extracted by applying MMPA to calculated property values.

MMPA is a versatile method that can be used for multi-parameter optimisation [55,199]. This can be achieved by determining the effects of chemical transformation on multiple properties (such as solubility and plasma binding) and selecting the transformations that provide the optimum change in the properties [186]. However, MMPA typically has some shortcomings that need to be addressed. Firstly, the results of MMPA can be significantly affected by the errors in the data[185]. Since each data point used on the analysis is the difference between property values of two molecules, the errors compound. Secondly, investigation into the varied effect within a transformation (different MMPs of the same transformation have different effects) has not been widely studied [185]. Some studies have taken into account the contextual information (molecular structure surrounding the site of transformation) [190,197]. It was noted that some transformations had a context chemotype (similar structure) specific effect that was undistinguishable without this approach. Thirdly, the application of MMPA has been mostly limited to properties of interest during the Discovery stage of drug development.

2.6 Summary of the chapter

The modern pharmaceutical product development process was developed as a result of millennia of human struggle against disease. The process has successfully contributed to longevity and quality of life improvements. In recent years, the productivity of drug product development has been decreasing; primarily due to failures during clinical trials caused by insufficient human efficacy. With the ultimate goal of predicting the efficacy (out of scope of the thesis), the relationships underpinning the performance of pharmaceutics was analysed using the framework of the Material Science Tetrahedron. Molecular structure – polymorph propensity, and molecular and crystal structure – melting point were identified as the two structure property relationships that the thesis focuses on. The theoretical framework for two empirical approaches: Quantitative Structure Property Relationship (QSPR) and Matched Molecular Pair Analysis (MMPA) were also presented.

In the following chapter, the development of a Matched Molecular Pair Database (MMPDB) for streamlined MMPA of properties related to performance of pharmaceutical products is discussed. The chapter focuses on the method development and explains the design decision behind the database schema. The approach is contextualised within literature works and its applicability to further research in the thesis is discussed.

Chapter 3

Matched Molecular Pair Database

3.1 Introduction

Matched Molecular Pair Analysis (MMPA) has been widely used within the Discovery stage of the Pharmaceutical Product Development [11]. The analysis provides an easy way to interpret results that can be used to assist in molecular optimisation. More recently, the MMPA methodology was applied to the Cambridge Structural Database (CSD) to investigate the effects of molecular transformation on crystal packing [189]. This is the first application of MMPA to a dataset that resembles a Development stage dataset. The work in the thesis aims to further this research by focusing on transformations that affect polymorph propensity. The purpose of applying empirical methods commonly used in Discovery to address Development challenges is to allow for better integration of the two stages and to enable the prediction of Development challenges while still in Discovery.

For systematic MMPA, a database can be a useful method of MMP storage to avoid repeated, computationally expensive MMP identification (3.1.1). Several MMP identification algorithms exist (as discussed in 2.5.1.2); the selected algorithm is presented in 3.1.2. This work (along with work shown in Chapter 6) was presented at UK QSAR conference [200]. Shortly after the work on this chapter was completed, a similar MMP database was published [201]. Comparison of the database developed for this thesis, and the one available in the literature is presented later in the chapter (3.3).

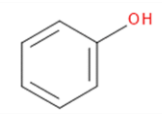
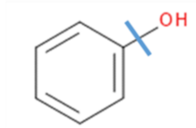
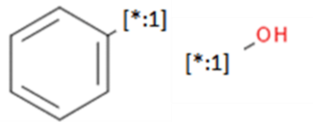
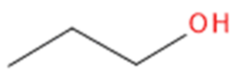
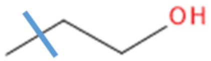
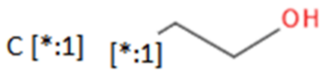
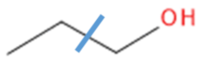
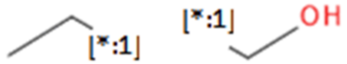
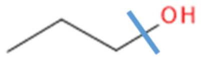
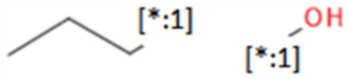
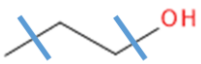
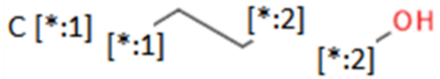
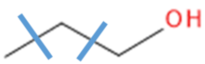
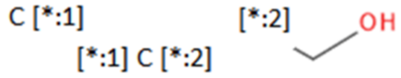
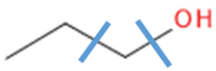
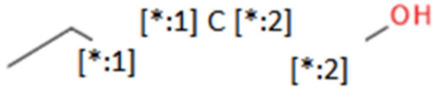
3.1.1 Need for database

Matched Molecular Pair identification is a relatively slow process. Due to the $O(n^2)$ nature of many algorithms (i.e. the computational cost increases with the square of the number of samples), repeated identification of MMPs at large scale is computationally expensive [180]. The CSD has quarterly updates with new structures. To avoid the necessity to repeat the identification process, a database is desirable. A database approach allows the molecular fragments generated from the previously seen structures, matching of which is necessary to identify MMPs, to be stored and indexed rather than having to generate these each time the dataset used for MMPA is updated. Beyond storing MMP information, the database can also be used to store a number of properties. The database also needs to store some crystallographic information in order to effectively interact with the CSD.

3.1.2 Hussain and Rea Fragmentation (HRF) method

Hussain and Rea Fragmentation (HRF) method [180] was selected for MMP identification due to its computational efficiency and the ability to easily integrate it within Python workflow via its RDkit implementation [202]. Several improvements to the algorithm were introduced during the development of the workflow for the database population (3.2.3). The HRF algorithm is an automated MMP identification algorithm that does not require pre-specification of transformations. It uses SMILES (simplified molecular-input line-entry system) representations of molecular structure. The chemical notation system developed to allow computer processing and efficient substructure searching [56] (e.g. paracetamol is represented by CC(=O)Nc1ccc(O)cc1). Rdkit was used for SMILES generation [202]. The algorithm can be divided into three steps, (1) fragmentation, (2) indexing, and (3) MMP identification. Fragmentation of the input SMILES is performed by one, two, or three cuts (see Table 3.1 for example of single and double cuts). The cuts are limited to acyclic bonds between non-hydrogen atoms and it is ensured that predefined functional groups are not cut. This ensures that groups such as a carboxylic acid group(R-COOH) are not fragmented into a ketone group (R-C(=O)-R’) and a hydroxyl group (R’-OH). All the fragments are then indexed which includes all the possible ways in which a given molecule can be fragmented. A matched molecular pair is then identified by grouping molecules that share the same fragment (context). MMP identification is made by identifying all molecules that share the same context as the molecule of interest. The core is the fragment of the molecule that changes across an MMP. Transformation is the change defined by the core of each of the molecules. The larger a formed core is, the less likely it is to occur in multiple instances, reducing the likelihood that any results obtained from it will be statistically significant. From a chemistry perspective, the study of MMPs with large changes is uninformative as the two molecules are chemically too different. An example of such MMP is paracetamol and ethanol (hydroxyl group is the context; the change is from Acetanilide to methyl group). Therefore, a size limit is imposed to eliminate MMPs where the change is too big to be meaningfully included in the analysis. The identification step can be repeated for all molecules to identify all MMPs within a dataset. This can also be accomplished by limiting the ratio of the change to the molecule size.

Table 3.1: Fragmentation of molecules for MMP identification.

Molecule	Fragmentation	Fragments	Comment
			• Cyclic bonds uncut • No double / triple cuts can be performed on this molecule
			• Three different single cuts can be performed
			
			
			• Three unique double cuts can be performed • Two fragments on either side are the context (part that does not change in a MMP) • The fragment in the middle (one with [*:1] and [*:2]) is the core (part that changes in a MMP • Triple cuts are not possible
			
			

3.2 Database design

The database schema and the process to populate it was developed and is discussed in this section. The aim of the schema is to store MMP data as well as additional property data to enable easy MMPA. The workflow is based on the RDkit implementation of the HRF algorithm introduced in 3.1.2. The algorithm was expanded upon in the current work, with the differences discussed in 3.2.3.

3.2.1 Schema

The database schema was proposed to store the molecular and crystal information along with the identified MMPs. The schema (presented in Figure 3.1) has three types of tables, based on their primary purpose. The grey tables (fragments context_table, and core_table) are used solely for MMP identification process (see 3.2.2 for details of HRF algorithm implementation). The fragments table contains the fragmented molecules (single row per cut per molecule). The table also stores information relating to the resulting fragments such as the fragment size, its size ratio, and whether the fragmentation was done by a single cut (this distinction is needed for handling transformations including hydrogen). The context and core tables contain context and core fragment information, respectively.

All_smiles, MMP, and Transformation are the second type of tables; these contain information needed to perform MMPA. All_smiles table holds all the molecular structure information such as SMILES, size, and flag columns used to keep track

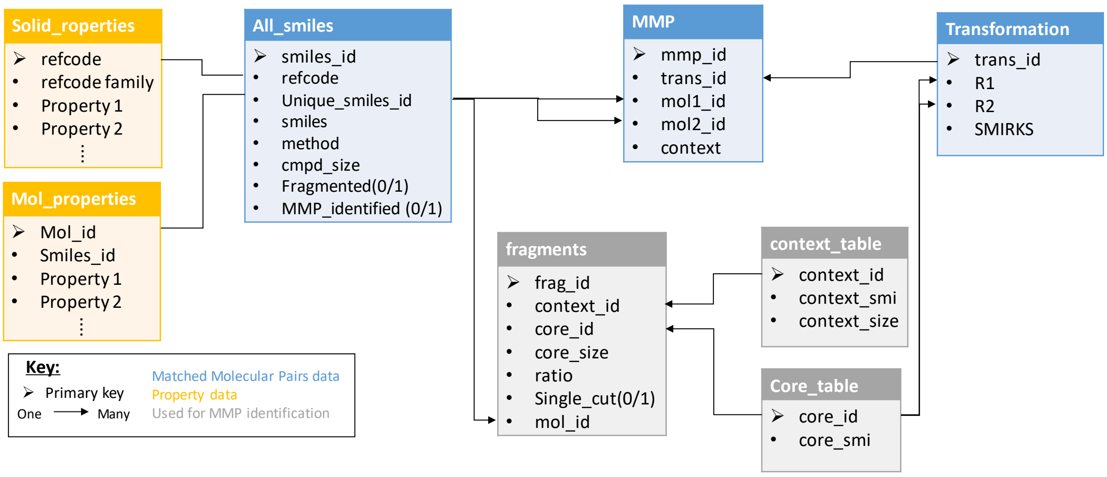


Figure 3.1: MMP Database schema.

A larger image of the schema is available in Appendix 1

whether a given molecule has been fragmented and its MMPs identified. It also contains refcode, which is used to associate a crystal structure to the given molecule. Transformation table contains the unique transformations identified within the dataset. The core_ids for the cores of the transformations are stored in R1 and R2 columns respectively. Smirks are generated to allow for each specific transformation. MMP table contains all the identified MMPs. The smiles_ids of the two molecules are stored in mol1_id and mol2_id. The transformation identifier (trans_id from Transformation table) and context identifier (context_id from context_table) complete the information that is stored for each MMP. This allows for each selection of all MMPs for a given transformation or the context of interest. By parsing through mol1_id and mol2_id, all MMPs of a given molecule can be retrieved as well.

All molecular and crystal properties are stored in two respective tables (Mol_properties and Solid_properties). The properties in these tables are used for MMPA. For the purposes of the polymorph propensity study discussed in Chapter 4, the number of known polymorphs was considered a molecular property, so this was added to the Mol_properties table.

3.2.2 Workflow for population of the database

The workflow for the generation of the MMPs consists of three stages: fragmentation, indexing, and MMP identification. However, for the process to begin, molecular structures, expressed as SMILES, are required. In case of a dataset with only molecular structures, a file containing SMILES and optionally molecule id can be used as the input. If the dataset contains crystal information (as it was the case for the work discussed in this thesis), a file containing CSD refcodes can be used as the input.

If refcodes are supplied, the CSD Python API is used to access the molecular structure of the crystal. Canonised SMILES are generated using a script supplied by the CCDC. The canonisation is a process that ensures a molecule structure is always represented in the same way (for example; ethanol could be written as OCC, C(O)C, or CCO). In case of a multi-component crystal structure, all distinct molecular structures are retained.

The added SMILES are compared against all molecules already in the database. This step is skipped if a new database is created. Two types of identifiers are added for each molecule. Firstly, smiles_id is assigned to every molecule that is added. Additionally, unique_smiles_id is assigned to every new molecular structure that is added. The unique_smiles_id is set equal to the smiles_id, the first time a molecular structure is encountered, and the unique_smiles_id of the first instance is used for subsequent entries with the molecular structure. For example, in case of a dataset of two hydrates (as defined by the CSD API), the first main component (non-water molecule) is assigned smiles_id and unique_smiles_id of 0. The water molecule from the first hydrate is assigned 1 for both identifiers. The main component of the second hydrate is similarly assigned 2 for both. However, the second water molecule is assigned smiles_id of 3 but unique_smiles_id of 1 (same as the first occurrence of the water molecule). In this way, all distinct molecular structures can be selected by specifying the condition that smiles_id must equal unique_smiles_id. This is quicker than selection based on SMILES string comparison.

The newly added, distinct molecular structures are fragmented using HRF method (3.1.2). The output of the fragmentation is referred to as ‘rfrag’.These results are stored in memory and are not inserted as-is into the database. ‘fragmented’ from ‘all_smiles’

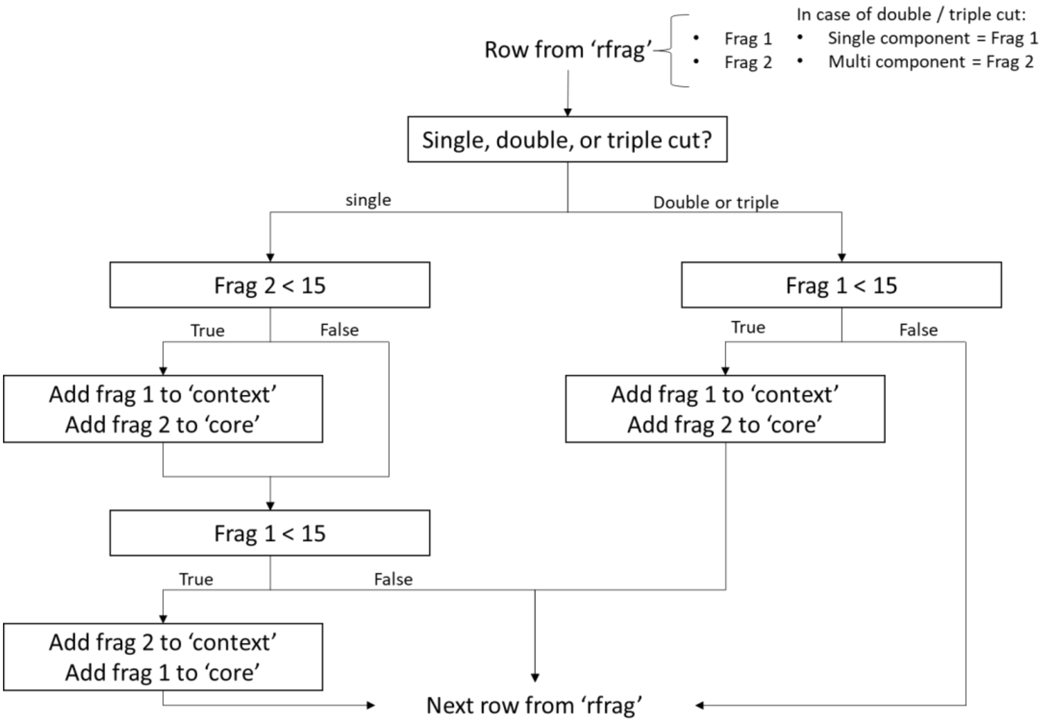


Figure 3.2: Schema explaining the indexing processes.

Series of logical tests are done before the fragment data is inserted into the 'context' and 'core' tables. 15 indicates the heavy atom count (non-hydrogen atoms) that is set as cut off for too large transformations.

table is set to True (1) for each molecule that was inputted into the fragmentation algorithm even if fragmentation failed. This ensures that the script can continue to run even if some errors were encountered. Most errors occur due to the fact that the molecules are un-fragmentable (such as water). The process creates a large number of fragments (over 319,000 from 8,879 molecules that are part of the drug subset[85]). It is impractical to attempt to identify MMPs from this, hence indexing is performed.

Indexing is the stage where the fragments are reorganised to allow easier MMP identification (Figure 3.2). The 'rfrag' data that is stored in memory is iterated over. Rows with single cut molecules are treated differently to double or triple cut molecules. The first fragment is selected and its heavy atom count (non-hydrogen atoms) is compared to the set cut off (typically set to 15). If the fragment is within the set size, it is inserted into the 'core' table. The other fragment is inserted into the context table. The step is repeated with the other combination of fragments. For single cut molecules, both "halves" of the molecule may be used as the core or context. For example for ethanol, the hydroxyl group may be used as the core and methyl group as the context and vice versa. Due to the fact that SMILES do not explicitly include

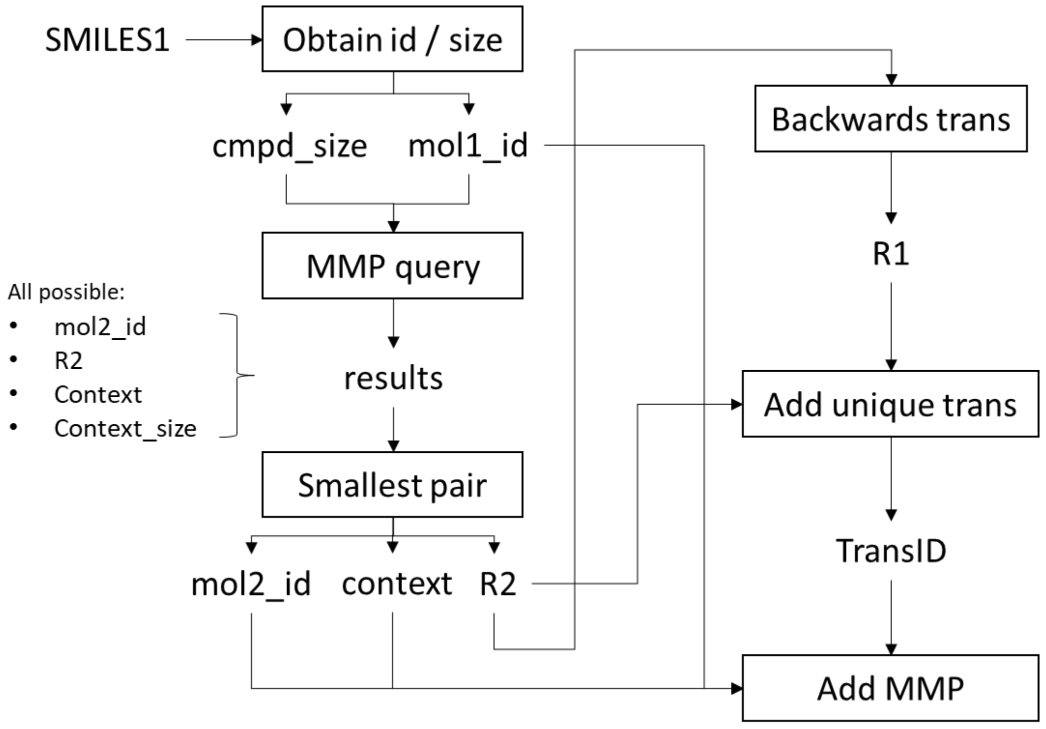


Figure 3.3: MMP identification stage.

All molecules are iterated over to identify all relevant MMPs.

hydrogens, transformations including it are handled separately. For all fragments resulting from a single cut, a hydrogen is added to where the cut was made and it is checked if that forms a valid molecule. If valid molecule is formed, all molecules within the dataset are searched to see if this molecule is present. If the molecule is in the dataset, a new row is added to fragments table. For example, using the hydroxyl group fragment of ethanol, a hydrogen is attached to it forming water. If water is present in the dataset, this will result in a new entry where the core is hydrogen, context is hydroxyl group, and the molecule is water. In case of double or triple cut, the fragment that contains the single component is inserted into the 'core' table if the other fragment meets the size requirement. Apart from size, the ratio of the heavy atoms of the core to the molecule may be used (either separately or in tandem).

The final stage is MMP identification where the MMPs are identified for each unique molecule that has been fragmented. The process involves several steps that are illustrated in Figure 3.3 For each input SMILES, the heavy atom count (cmpd_size) and its smiles_id (mo11_id) is retrieved. All possible mol2_id are identified (all possible MMPs within the dataset for the given mol1) by MMP query. This query returns several context- R2 combinations for the same pair of molecules. For example, in case of butane and butanol the following combinations would be returned: context = butane R2 = hydroxyl group, context = propane R2 = methanol, context = ethane R2 = ethanol, and context = methane R2 = propanol. The context and R2 are selected such that the context_size is the largest (therefore, smallest change). In the aforementioned example, this would be context = butane and R2 = hydroxyl group. For each of the identified MMP, R1 is retrieved from the database. The combination of R1 and R2 are checked in 'Transformation' table (Both R1, R2 and R2, R1). If the combination already exists, the corresponding TransID is retrieved. If Transformation R2, R1 was already in the database, the molecules are reordered (mol1id becomes mol2id and vice versa). Otherwise, the newly identified transformation is inserted into the database and the TransID is retrieved. The combination of mol1id and mol2id are searched in the 'MMP' table, and if the pair is already in the database, the script proceeds to the next pair of molecules. Otherwise, the pair of molecules, along with context and TransID are inserted into the table. This is repeated for all pairs of molecules identified. Once all pairs are evaluated, the 'all_smiles' table is updated by setting 'MMPidentified' to True (1) for the given molecule. 'MMPidentified' is set to True even if no MMP were identified for the molecule. The process is repeated for all fragmented molecules in the 'all_smiles' table.

3.2.3 Modifications to the MMP identification

The MMP database generation process follows a similar procedure to the original implementation of the HRF method. However, some changes were made to address the shortcomings of the original method. Firstly, using a database allows for addition of new molecular structures without rerunning the entire process (3.2.3.1). Secondly, some instances where the HRF method generates multiple MMPs of the same pair of molecules were addressed (3.2.3.2).

3.2.3.1 Updatability

The updatability of the database was compared to deploying the original implementation of the HRF method. A dataset of 50,000 randomly selected, organic molecules with no disorders in the crystal structure (this often broke the SMILES generation step of the flow) were processed by both methods. The dataset was consequently increased by 10,000 molecules three times (to a total of 80,000 randomly selected molecules). In case of the HRF method, the fragmentation was performed only on the additional molecules, and the MMP identification on the entirety of the fragments. This was necessary because the implementation of the HRF algorithm does not support MMP identification between the added dataset and the original dataset. In case of the database approach, fragmentation is performed on the added molecules and the MMP identification is only performed on them as well. The identification includes MMPs between the additional molecules and the molecules in the original dataset. The benchmarking was performed on a Windows 7 machine with Intel Xeon E3-1226 v3 3.00 GHz 4 core processor and 16GB of RAM. Only a single core was used in the processing.

The processing times for the two methods are shown in Figure 3.4. For the initial identification of MMPs in the 50,000 molecule is twice as long for the database method compared to the HFR method. This is because both methods perform the same computation, while the database also performs database read and write actions. The majority of the HFR algorithm is implemented in a lower-level language (C with Python wrapper) compared to the majority of the processing logic being implemented in Python for the database approach (C-based libraries were utilised). However, the processing time for further 10,000 molecules is shorter for the database method compared to the HFR method. Despite being computationally less efficient, the database method has less computation to do. In this particular case, the break-even point is after the second 10,000 molecule update. The database method shows some advantages in terms of computation time for rapidly growing sources of data such as the CSD, which was a key source of data for work presented in the thesis. Alternative approaches to reducing the processing time by improving the efficiency of the algorithm itself are likely possible, but fall outside of the scope of the thesis.

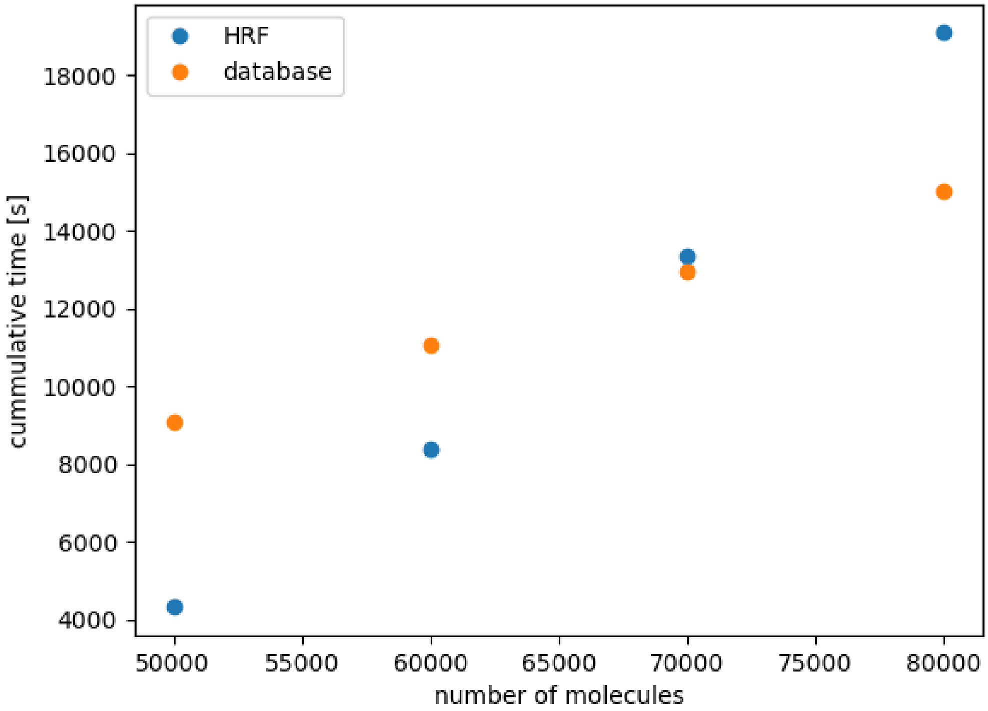


Figure 3.4: Performance comparison between HRF and database method of MMP identification for an increasing dataset.

Comparison performed on a Windows 7 machine with Intel Xeon E3-1226 v3 3.00 GHz 4 core processor and 16GB of RAM. Only single core was used.

3.2.3.2 Elimination of duplicate MMPs

The database method also addresses some of the shortcomings of the original HRF implementation. For molecules that can be cut at different points, multiple MMPs for a given pair of molecules may be identified (illustrated in Figure 3.5). The MMP resulting from cut 1 (OH>>CH_3) in the figure corresponds to the smallest, and likeliest to repeat across the dataset. The remaining two possible MMPs, although valid, are not as useful in terms of MMPA artificially increasing the number of MMPs within a dataset. The database approach addresses this issue because MMP identification is performed per molecule. Once all MMPs for a given molecule are identified, any duplicates based on the matching molecules are eliminated. For a given pair of molecules, the MMP with the largest context (smallest change) is kept. This

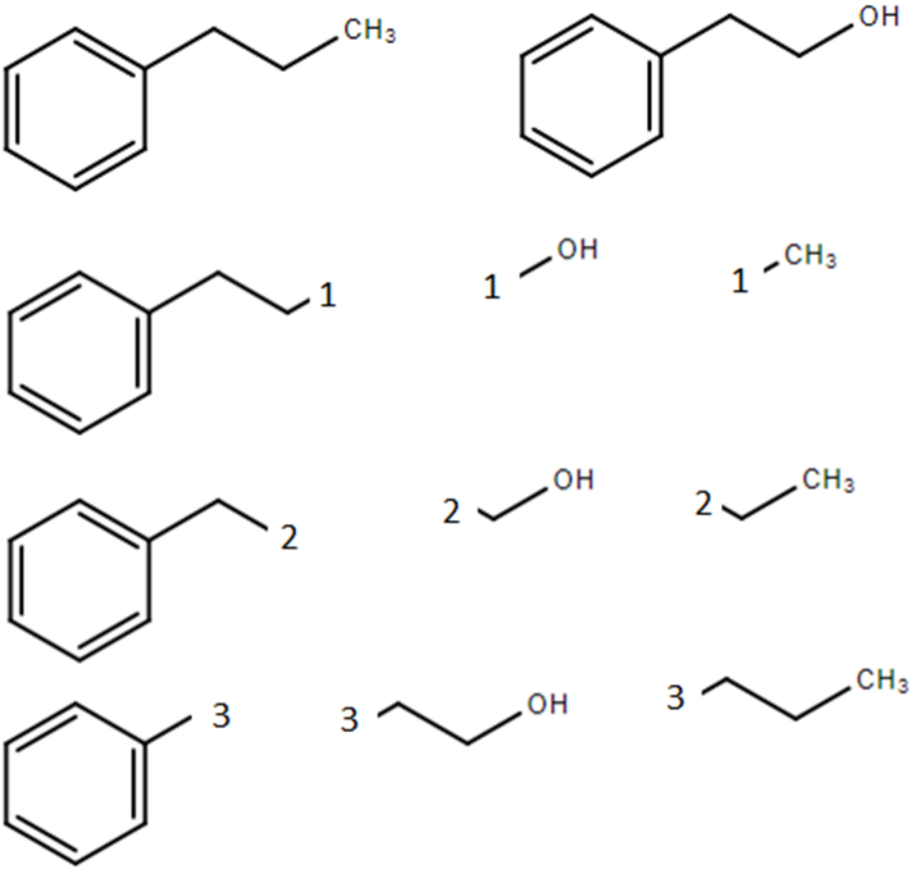


Figure 3.5: Multiple MMPs that can be identified from the same pair of molecules.

The molecules can be cut at different points (1-3). All the cuts are valid (only a single C-C bond is cut, the resulting fragment size ratio is within limits).

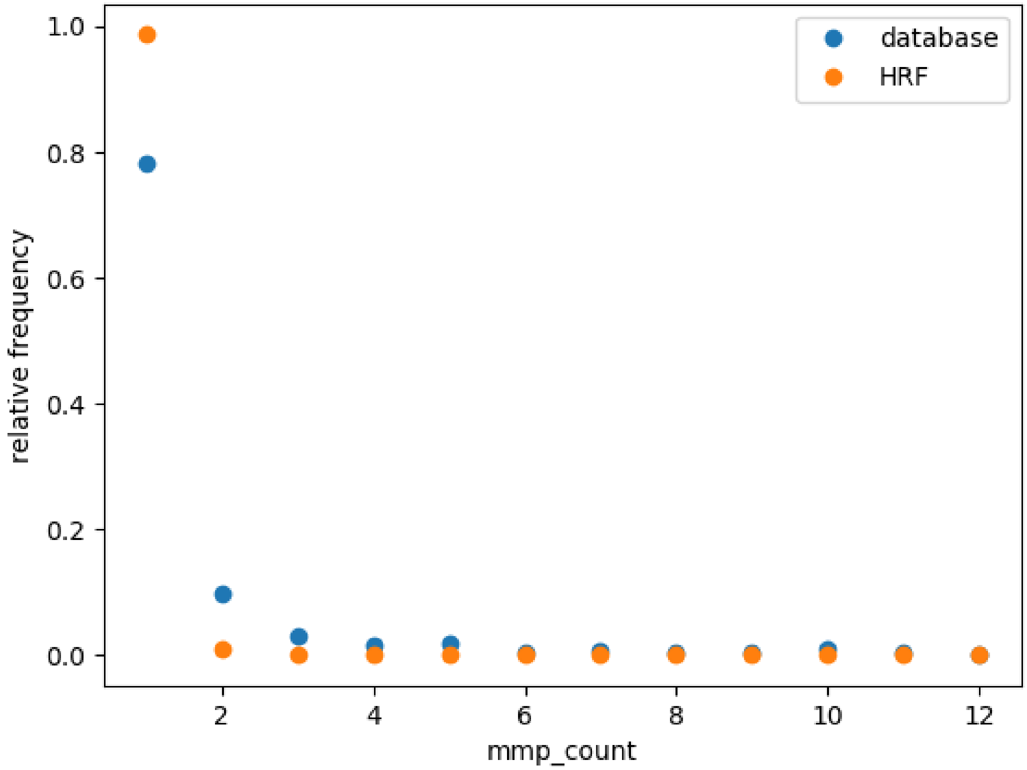


Figure 3.6: Comparison of frequency of occurrences of transformations.

The database method reduces the number of transformation that do not occur often by removing duplicate MMPs for the same pair of molecules.

transformation is the most common and meaningful in terms of MMPA. This procedure reduces the number of MMPs that rarely occur as shown in Figure 3.6.

3.3 Comparison to another MMP databases approach

Shortly after completion of the work discussed in this chapter and presentation at UK QSAR in March of 2018 [200], a similar MMP database approach was published in May of the same year [201] (hereafter referred as DHK method for the names of the authors). Both, the work presented above and the DHK method address the same issue regarding MMPA; the extensive processing required to be carried out to identify all MMPs within a dataset, and aims to aid systematic use of MMPA. The paragraphs below present a comparison between the work presented in this chapter and the published approach.

The HRF method is the basis of MMP identification used by both methods (one presented in the thesis and the DHK method). However, the DHK approach expands this to handle transformations involving chirality. This is achieved by the “welding” technique developed as part of the publication [201]. For double-cut MMPs, the order of attachments is stored, and canonicalization of the re-connected fragments is checked to ensure it matches that of the original molecule. The approach provides a useful mechanism for differentiation of stereoisomers.

Another difference between the DHK compared to the work in the thesis is the inclusion of local environments. In the case of MMP, environment refers to the atoms that surround the location where a cut is made during fragmentation. This information is stored in the database and can be used to select MMPs with only the same environment when conducting the analysis.

However, whilst the DHK approach offers some potential advantages over the database approach developed, at the same time, in this thesis, it should be noted that only the approach presented in this chapter was integrated with the population of a database for analysis of solid state data. Matched Molecular Graph (introduced in Chapter 6) construction was also added to the capabilities of the database presented in the thesis. Due to the small number of MMPs, environmental consideration could not be conducted (Chapter 4).

3.4 Summary

This chapter presented the method used to generate a database of Matched Molecular Pairs that was subsequently used for the study of the effects of molecular changes on solid state properties (Chapter 4). The database facilitates repeated analysis with growing dataset without having to repeat MMP identification. Another advantage of the method presented here is the ability to limit the MMPs that are not useful. Namely, repeated MMPs for the same pair of molecules, and reduction in number of rare transformations (ones that occur only a few times and hence are statistically not significant). An interactive analysis tool was also created to complement the database and allow MMPA to be carried out routinely. The scripts written to generate the database and carry out analysis is available in Appendix 1.

A similar database was published during the course of this work. The work offers several advantages in terms of handling of chirality and storing of transformation environments [201]. However, due to the ease of integration with the CSD and Matched Molecular Graph capabilities (detailed in Chapter 6), the method presented in here was used in subsequent research. In the following chapter, this method and database are used to study the effects of molecular transformation on the propensity for molecules to exhibit polymorphism.

Chapter 4

Polymorph Propensity Prediction

4.1 Introduction

The ability to predict the propensity to form polymorphs is valuable to the pharmaceutical industry [9]. Unexpected polymorphism of the drug compound necessitated the removal of ritonavir from the market [8]. Polymorph screening is typically carried out during the Development stage of drug product development with the aim to find all polymorphs within the range of applicable conditions. The ability to predict the polymorph propensity may potentially allow to anticipate the magnitude of challenges likely to be faced during the Development stage.

A number of studies have been done to better understand polymorphism. These typically focus on examining individual crystal structures [8]. Individual intermolecular interactions are assessed to see whether the structure is stable [64,79,203]. If the structure does not satisfy all potential intermolecular synthons, it is likely that other polymorphs exist. The assessment of the synthons can be done based on the statistically favourable interaction based on the analysis of the CSD [64]. Density Functional Theory (DFT) based approaches have also been used for this purpose [204]. Considerable work has been carried out in the area of crystal structure prediction using a range of methods such as DFT and Forcefield (FF) for calculation of structure stability coupled with search algorithms to explore the set of potential structures [205–207]. Blind tests for structure prediction have been periodically organised by CCDC since 1999 with the most recent one taking place in 2020 [208]. There has also been work published on the overall trends in polymorphism [50,209,210].

In this chapter, the issue of polymorph propensity is studied from the perspective of molecular transformations. The statistical approach to this is performed using MMPA (see Chapter 3 for details) on the CSD. The intention for the work is to allow polymorph propensity to be considered during the drug optimisation stage during Discovery.

4.2 Method and Data

4.2.1 Dataset

4.2.1.1 CSD single component dataset

The polymorph propensity study focused on single component organic structures. Single component structures were identified by checking the number of separate molecular components. If the number was one, the structure was considered a single component. If more than one molecular component was identified, SMILES strings were generated for each of the components. If all of the strings matched, the crystal structure was considered to be a single component structure. No organometallics were considered. Based on these criteria, a dataset of 155,040 crystal structures was identified. This dataset excluded all hydrates and co-crystals.

The CCDC publishes a list of crystal structures with the best R factor for each unique crystal structure (polymorph) in the CSD. The list is generated based on the comparison of generated spectra [211]. The details and the effectiveness of the method are discussed in Chapter 5. The number of occurrences of each of the refcode within the best R factor list corresponds to the number of polymorphs of that molecular composition. The number of redeterminations was calculated by subtracting the number of polymorphs from the total number of refcodes for the specific refcode family within the CSD.

4.2.1.2 Monomorphic adjustment

The CSD is a repository of published crystal structure so it reflects the research trends within the scientific community. A number of reasons exist for determining the crystal structure of a compound. This may be done to confirm the molecular structure and the crystal information is of secondary importance. In such cases, it is unlikely that different experimental conditions were investigated and no polymorphs were found. However, this does not exclude the possibility that multiple polymorphs exist.

For this reason, Monomorphic adjustment was introduced based on the literature precedence [50]. Structures with only one refcode were considered to be not sufficiently studied to determine whether these are indeed monomorphic or polymorphic with undiscovered polymorphs. The unfiltered CSD single component dataset has 1 % of polymorphic structures, which is significantly lower than other, more thoroughly studied datasets presented in Table 4.2. After elimination of structures with only one refcode, the dataset was reduced to 6,633 structures of which 25 % are polymorphic. The resulting dataset is referred to as the adjusted CSD single component dataset. The process reduced the dataset by 97 %, which reflects the prevalence of single-entry compounds. Concerns related to the large reduction in size are discussed in 4.3.2.2.

4.2.2 Molecular structure information

4.2.2.1 Matched Molecular Pairs

Matched Molecular Pairs were used to study the effects of small molecular transformations on the polymorph propensity of the molecule. The database method developed in Chapter 3 was used for the analysis. For details of the method, refer to the chapter. The maximum size of transformation used was 15 heavy atoms. Based on the analysis presented in 4.3.2.2, the data was filtered by limiting the ratio of the transformation to 0.3. The effects of limiting the ratio are discussed in 4.3.2.3.

4.2.2.2 Molecular flexibility and other molecular information

MMPs formed the basis of the study; however, additional information was also used to further study the effects of small molecular transformations. Molecular properties relevant to crystal lattice formation were selected.

Some molecules exhibit polymorphism due to the compound’s ability to crystallise in different conformational forms, such as the case of ritonavir [8]. This was captured by the molecular flexibility descriptor – nConf20 [212]. Other descriptors such as rotatable bond count were outperformed by nConf20 in crystallisability prediction study (86.1 % test set accuracy compared to 74.8 % for rotatable bond). The descriptor attempts to capture the accessible conformational space of the molecule by generating and optimising 50 random conformers. The optimisation is done using MMFF94 molecular mechanics forcefield [213]. The lowest energy conformer is selected as the reference structures. Any symmetrically similar conformers, based on root mean squared distance (RMSD) of less than 1 Å, to the reference structure was removed. The molecules were aligned prior to RMSD computation. The energy of each of the conformer was calculated. If the energy difference between a conformer and the reference structure was less than 20 kcal/mol, the value of the nConf20 descriptor was increased by 1 (initialised by nConf20 = 0). In essence, the descriptor is the number of conformers that fall within the 20 kcal/mol of the optimal structure. The parameters of the descriptor (number of random conformers and the energy cut-off) were selected based on the analysis carried out in the original publication [212].

As discussed in 2.2.2, intermolecular interactions such as hydrogen bonding and Van der Waals interactions play an important role in determining the crystal structure of the compound. The number of hydrogen bond donors and acceptors was used to approximate the molecules ability to form hydrogen bonds. Van der Waals interactions tend to increase as the size of the molecule increases, so the compound size was used [214]. Heavy atom count was used as a measure of the compound size.

4.2.3 Software

The work in this chapter was done using Python 2.7. All structures within the CSD were analysed using the CSD Python API (version 1.5.2) [215]. The database of MMPs was constructed using the workflow presented in Chapter 3. nConf20 descriptor calculations were done using the script from the original publications [212]. Data processing and visualisation was performed using pandas [216] and seaborn [217,218]. Scripts used in this chapter can be found in Appendix 1.

4.3 Results and Discussion

4.3.1 Polymorphism in the CSD

The fraction of polymorphic structures in the dataset derived from the CSD is significantly lower than for other literature sources presented in Table 4.2. The discrepancy is most likely due to the nature of the different data sources. As discussed in 4.2.1.2, the CSD reflects the research interests of a broader community that does not necessarily focus on polymorphism. Similarly, the microscopy studies were likely focused on crystal structure observation rather than a search for polymorphism. The European Pharmacopeia, SSCI (Southern Society for Clinical Investigations) polymorph screens, and the two pharmaceutical company database were more focused on finding polymorphs of the different compounds. As a result, these datasets contain a much higher fraction of polymorphic structures. This suggests the CSD single component dataset contain some structures that are polymorphic but for which the polymorphs remain undiscovered. The issues associated with this caveat are discussed in more details in 4.3.4.

Table 4.1: Most common transformations within the CSD single component dataset.

Statistics of the polymorph count change (mean and standard deviation) and the number of MMPs with that transformation are also included

Transformation	Mean	Std. dev.	Count
R-H → R-CH_3	0.037	0.312	6017
R-H → R-Cl	0.044	0.349	2333
R-H → R-OCH_3	0.036	0.308	1884
R-H → R-OH	0.018	0.410	1708
R-H → R-Ph	0.040	0.387	1653
R-H → R-Br	0.054	0.309	1396
R-H → R-NO_2	0.035	0.392	1391
R-CH_3 → R-Ph	-0.004	0.328	1310
R-H → R-F	0.043	0.356	1083
R-Ph(meta)-R’ → R-Ph(para)-R’	-0.020	0.337	1002

4.3.2 Effects of molecular transformations

4.3.2.1 CSD single component dataset

The MMPDB script identified 4,599,447 MMPs with 3,404,016 unique transformations. The ten most common transformations are shown in Table 4.1. The transformations represent a wide range of chemical changes such as the introduction of hydrogen bonding hydroxyl group, or π- π stacking phenyl ring. However, the mean change for all these transformations is approximately 0 with the biggest deviation from that being 0.054 (R-H → R-Br). Due to the fact that only 1 % of the structures are polymorphic (within the dataset), it is unlikely to find a transformation with MMPs that consistently contain polymorphic structures.

4.3.2.2 Adjusted CSD single component dataset

The analysis workflow was repeated for the adjusted CSD single component dataset. This dataset is 4 % of the original dataset, hence the number of transformations and MMPs is significantly reduced to 2,048 and 3,913 respectively. The monomorphic adjustment also had an impact on the distribution of the transformation effect. Figure 4.1 shows a comparison of the most common transformation (R-H → R-CH_3) for the two datasets.

The MMP count for this transformation decreased from 6,015 to 211, while the mean changed from 0.03 to -0.13. The tail of the distribution appears to be more prominent. This is because a large number of monomorphic entries were removed based on the adjustment. The change is not significant based on the paired t-test, without considering multiple hypothesis testing correction [187,219]. Multiple hypothesis testing is typically performed. This is unsurprising since a small transformation that does not significantly alter the potential intermolecular interactions was not expected to have an effect on polymorph propensity. However, it is important to note the reduction in the number of MMPs that occurs when the monomorphic adjustment is made as this is a consistent issue across all studied transformation.

Emphasis was placed on transformations that are likely to be statistically significant based on the paired t-test. No multiple hypothesis testing was performed at this point. 5 % significance level was chosen as the basis of selections of transformations (37 transformations were identified. Distributions for some of these transformations (selected based on statistical or chemical interest) are shown in Figure 4.2.

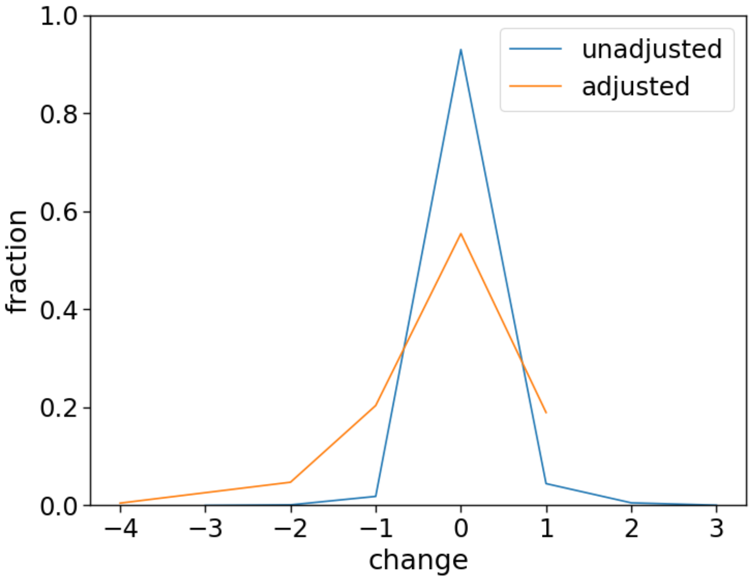


Figure 4.1: Comparison of the R-H → R-CH_3 transformation for adjusted and unadjusted CSD single component dataset.

The hydroxyl to phenyl transformation was focused upon. It contains transformations that alter the molecule significantly (Figure 4.3). The molecule doubles in size due to the transformation. The change was considered to be too dramatic for the MMPA to

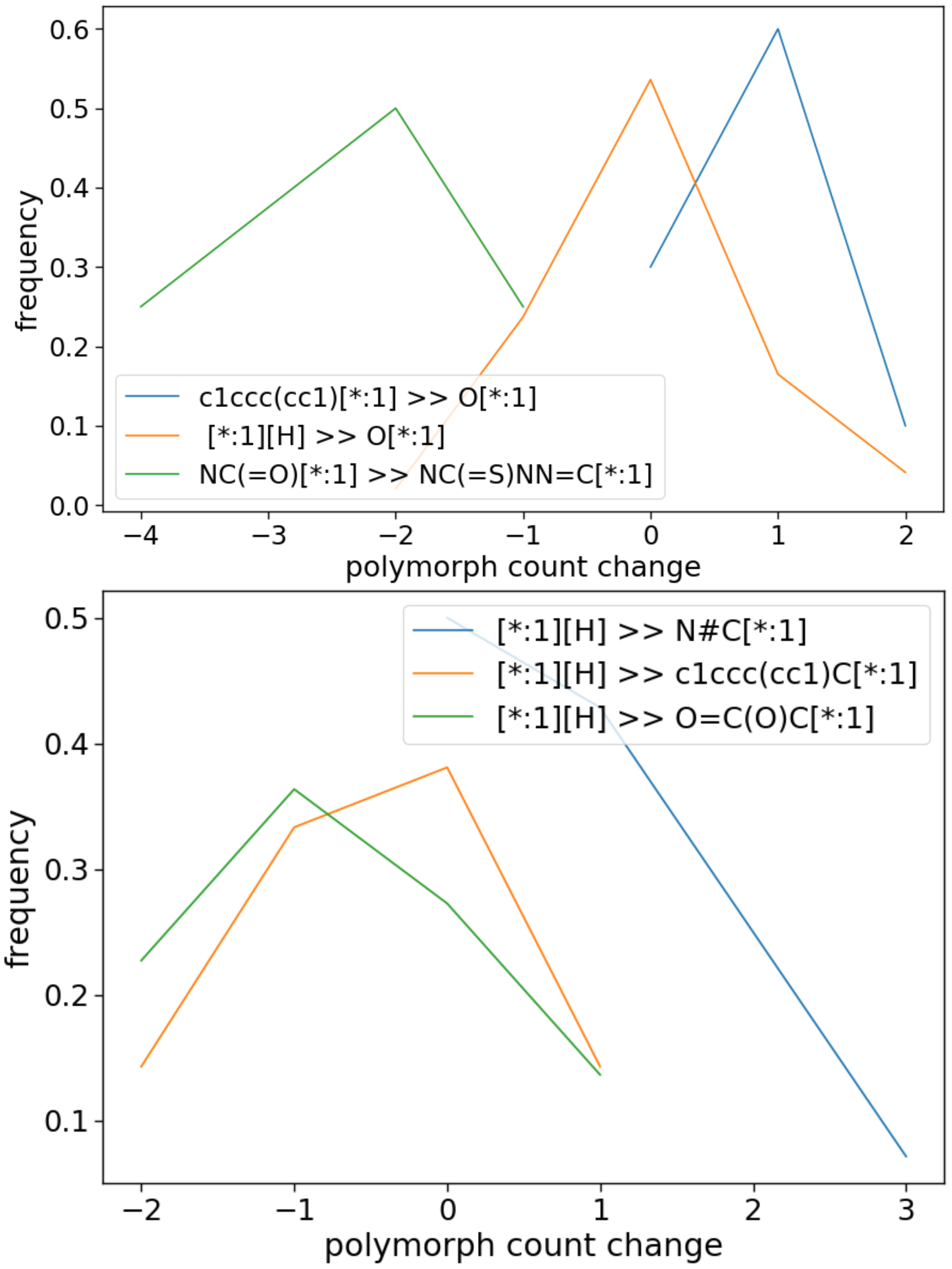


Figure 4.2: Distributions of the effects of the selected transformation on polymorph count.

Top: biggest change within 5 % significance interval (blue), most common transformation (grey), lowest p-value (dark blue). Bottom: highest count within 5 % significance interval (green), large fragment size change (dark blue), large change in reactivity (brown)

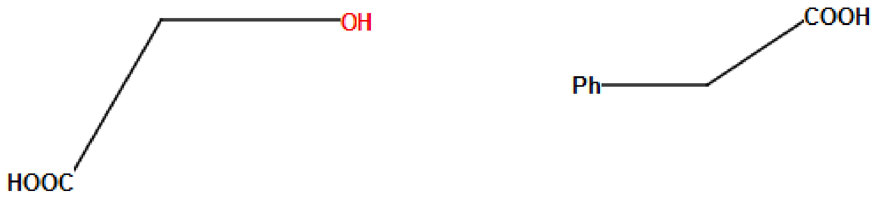


Figure 4.3: Example MMP of the hydroxyl to phenyl transformation. Refcodes: GLICAC (left), ZZZMLY (right).

be a useful assessment of the effects of transformation change. Similar issue persists with other MMPs, so the maximum change ratio of 0.3 was imposed (i.e. the heavy atom count of the change cannot exceed 30 % of the count of the whole molecule) [180].

4.3.2.3 MMPs limited by the ratio of the change

The imposition of the ratio restriction further reduced the number of MMPs (2,776 MMPs, reduced from 3,913). Transformations with the highest MMP count, largest mean change, and most likely to be statistically significant were selected for closer analysis. The hydrogen to phenyl group had the MMP count of 9 and the largest mean change of -0.667. The paired t-test p value of 0.156 suggests this is not a significant change. The low MMP count is the likely reason for the high p value. Furthermore, there is one data point with the change value of -4. This single datapoint shifts the average by 0.417 from -0.250 (when calculated omitting this point). With the increase in data quality (monomorphic adjustment) and the focus on chemically meaningful transformation (ratio limit), the number of MMPs are reduced to the point where a single data point may sway the overall average.

The hydrogen to chlorine transformation is most likely to be statistically significant based on p-value, with the paired t-test p-value of 0.085. However, the transformation is not significant at the 5 % level. Similarly, to the hydrogen to phenyl transformation, the number of MMPs is low (8).

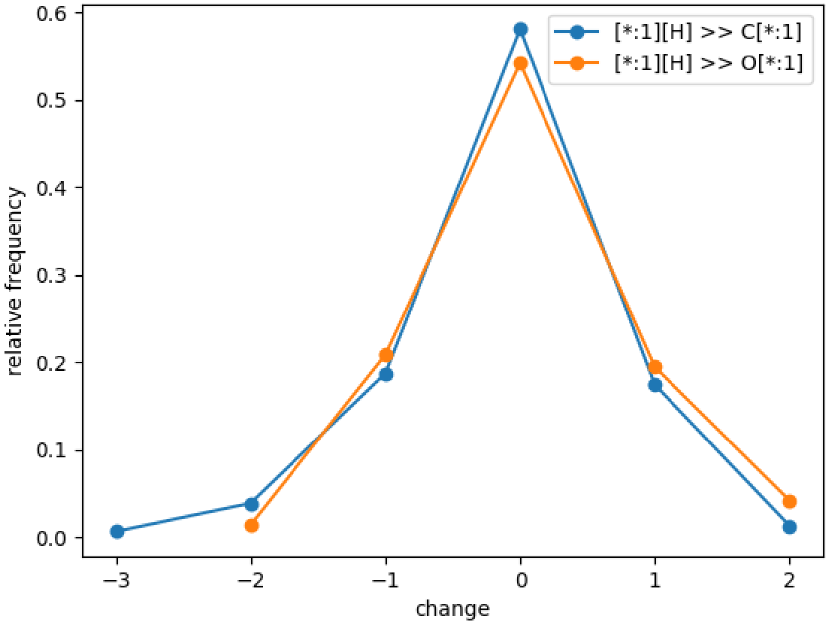
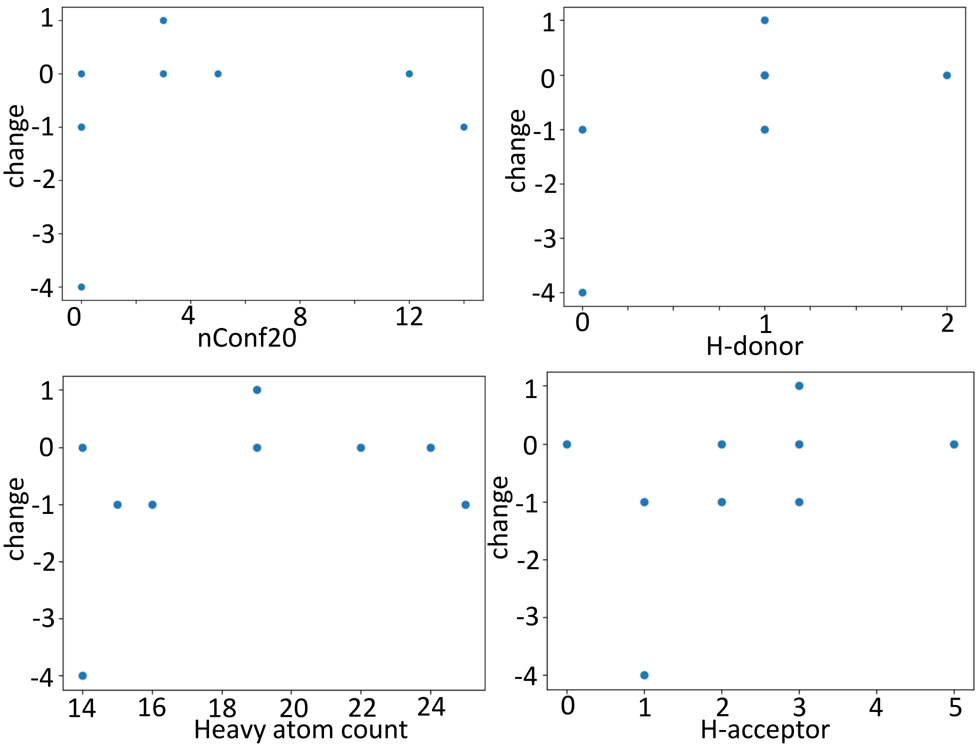


Figure 4.4: Comparison of hydrogen to methyl and hydrogen to hydroxyl transformation for adjusted CSD single component dataset with ratio limited MMPs.

On the other hand, the transformations with the highest counts tend to have a mean change of approximately 0. For example, hydrogen to hydroxyl transformation has 72 MMPs and the mean change of 0.041. The distribution of the effect approximately matched the distribution for hydrogen to methyl transformation (Figure 4.4). This is likely due to the limited proportion of polymorphic structures within the dataset, even after the monomorphic adjustment. A number of compounds may have unknown polymorphs which skew the mean transformation effect towards 0. This is further discussed in 4.3.4. Another possibility for the observed results is the importance of the context of the MMP (the part of the molecule that does not change across the pair).

The flexibility (nConf20), potential to form hydrogen bond (donor and acceptor count), and the Van der Waals interactions (heavy atom count) of the context (part of an MMP that is same across the pair) of each of the MMP was calculated. Only single-cut MMPs were used and a hydrogen was placed at the cut to make a valid molecule. The effects of each of the descriptor on the size of

change for the MMPs with the hydrogen to phenyl transformation are shown in



(

Figure 4.6). No correlations were found for this transformation. The process was repeated for hydrogen to hydroxyl transformation which had more MMPs (72) to see if any pattern emerges with larger data size (

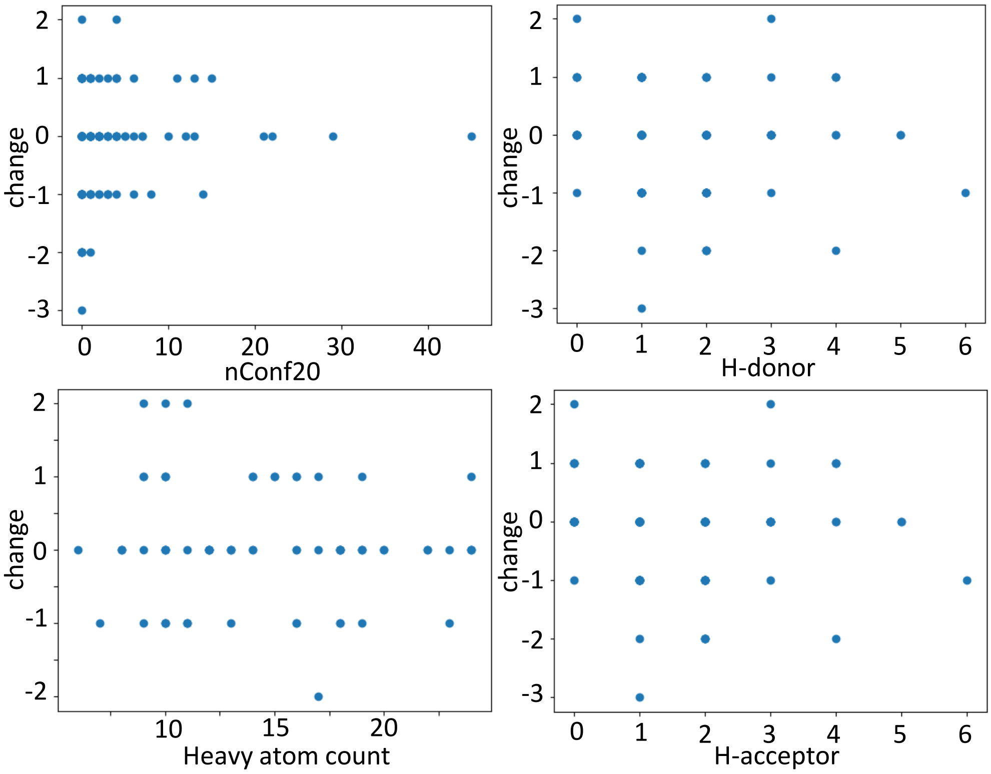


Figure 4.5). No correlation was found between any of the descriptors.

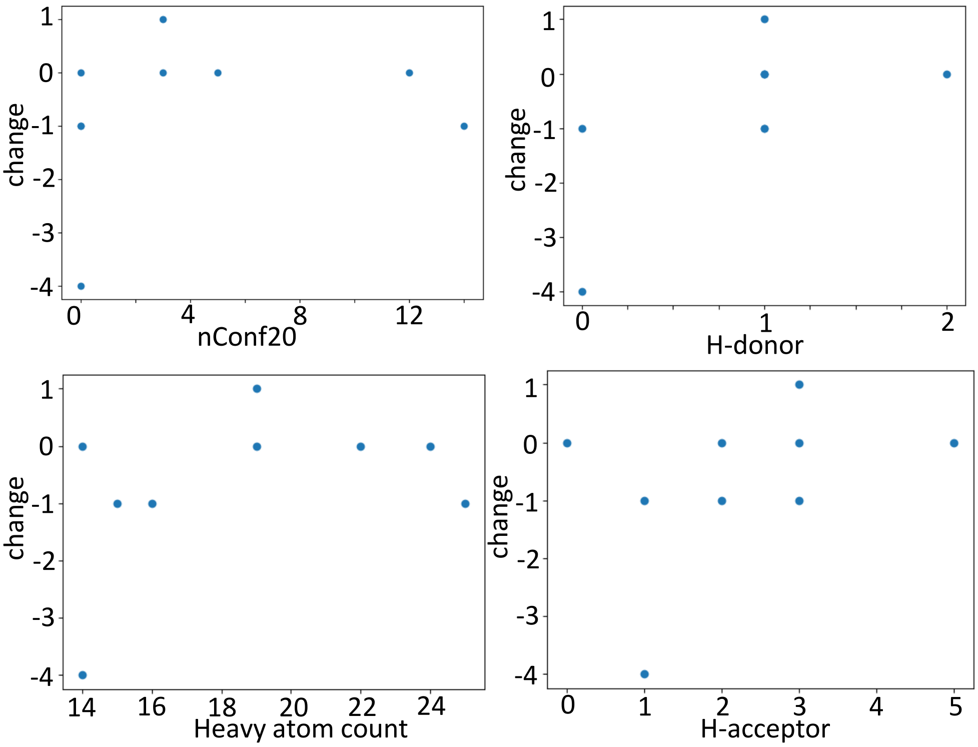


Figure 4.6: Effects of nConf20, H-bond donor / acceptor count, compound size on the change for MMPs with hydrogen to phenyl transformation.

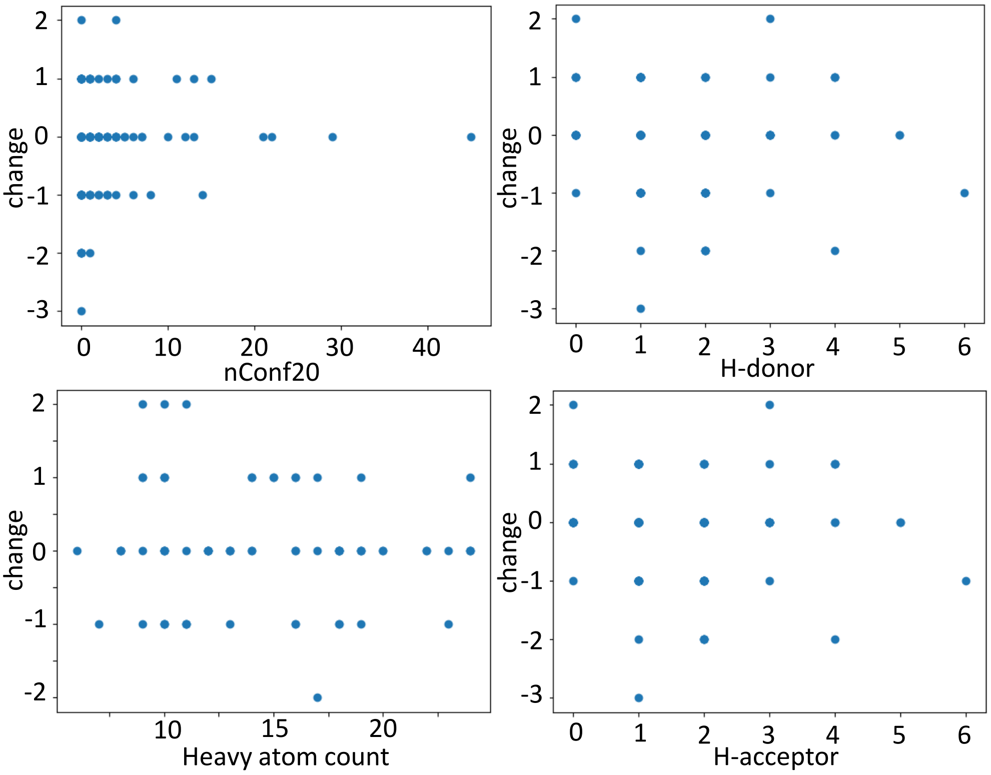


Figure 4.5: Effects of nConf20, H-bond donor / acceptor count, compound size on the change for MMPs with hydrogen to hydroxyl transformation

4.3.3 Effects of molecular flexibility

Flexibility was considered a potential factor that influences the polymorph propensity. As well as examining the influence on polymorph propensity due to molecular transformations, the effects of the flexibility itself on polymorph propensity were studied. A more flexible compound was expected to be able to form more distinct crystal packing. The distribution of the nConf20 descriptor for the different number of polymorphs is shown in Figure 4.7. Visually, there appears to be no difference between monomorphic and polymorphic compounds. The median for both, monomorphic and polymorphic structures is 5 with the means of 9.24 and 8.56 respectively. Mann Whitney U test was done to compare the two distributions and the result was not statistically significant (p value = 0.415). Interestingly, the flexibility of the polymorphic structures appears to be lower on average than that of the monomorphic structures. This can be explained by considering the intention behind the development of the nConf20 descriptor, which was to determine the crystallisability of a compound [212]. In essence, higher nConf20 value, the more difficult it is to crystallise a molecule. Therefore, the lower nConf20 value for polymorphic structures could be the artefact of the ease of crystallisation of these

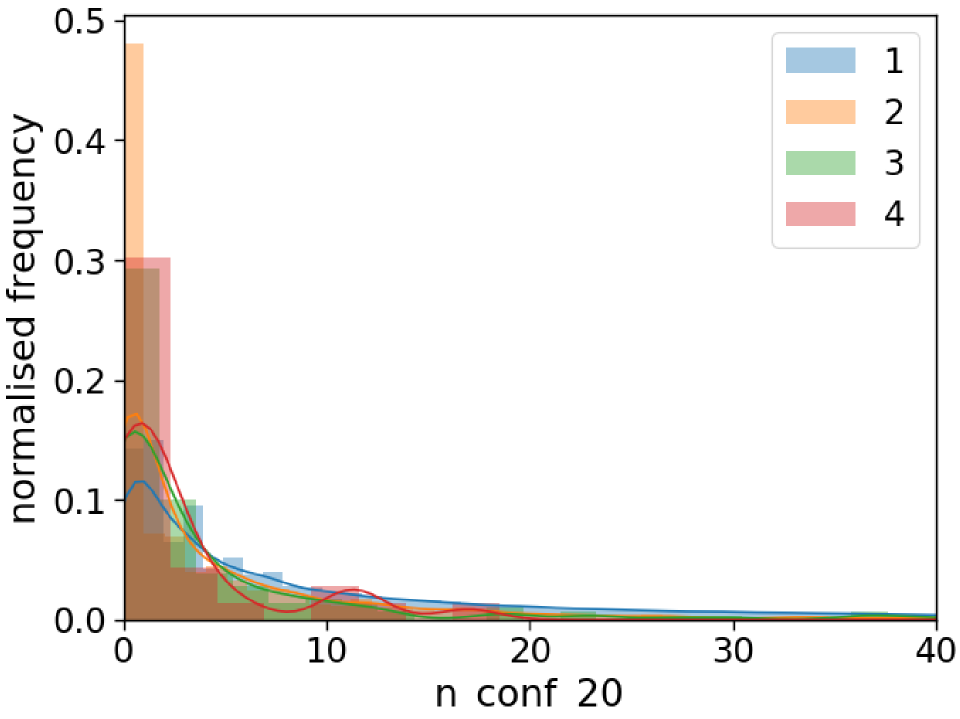


Figure 4.7: Distribution of nConf20 descriptor for compounds with different number of polymorphs on the CSD adjusted dataset.

structures. This may result in an increased likelihood that such structures were crystallised and added to the CSD. Hence, this trend is most likely an artefact of the data availability.

4.3.4 Issue of unknown polymorphs

The issue of unknown polymorphs has been mentioned in several sections of this chapter. In this section, the discussion is collated and further analysis of this is presented. The unique challenges associated with this issue are also discussed.

4.3.4.1 Exploration of the issues

The number of polymorphic structures is significantly lower within the CSD than other data sources that focus more on polymorph screening (Table 4.2). Two hypotheses for explaining this can be constructed: CSD represents a chemical space that is less polymorphic compared to heavily screened pharmaceutics chemical space, or a number of structures that are classed as monomorphic are actually polymorphic. The

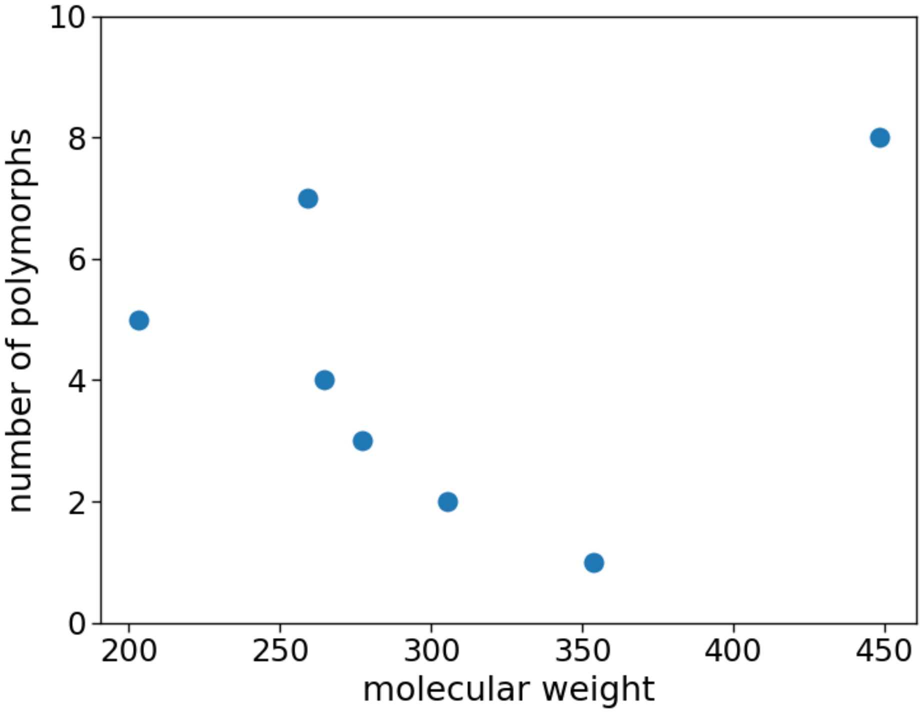


Figure 4.8: The number of polymorphs as a function of molecular weight. The mean weight of n-polymorphic compounds were taken. 8-polymorphic compounds consist of a single molecule (same for 7-polymorphic).

chemical space of the CSD has been compared to the chemical space of drugs and other molecules held in pharmaceutical company databases [85]. In the study, heavy atom count, flexibility, and hydrogen bond donors /acceptors were used to compare the chemical space of various datasets.

The CSD has a wider range of compound sizes with larger representation in the smaller range [85]. 13 % of molecules within CSD are smaller than 100g/mol, while the industrial datasets do not have many molecules in this range. The industrial datasets tend to have larger molecules than the CSD. The industrial datasets also tend to have more polymorphic structures (Table 4.2). This leads to the suggestion that larger molecules tend to have a higher propensity for polymorphism. However, this is not reflected in the CSD where smaller molecules appear to have a higher propensity for polymorphism (Figure 4.8), further emphasising the likelihood that the results are affected by the data artefacts.

A similar trend can be observed for molecular flexibility. In the cited study [85], the rotatable bond count was used as a descriptor of flexibility. The CSD has more molecules with no rotatable bonds than the industrial datasets. The industrial datasets have more molecules with 5 and 6 rotatable bonds. Again, the increase in flexibility does not correlate with an increase in polymorph propensity. For this reason, it is unlikely that the difference in polymorphism found in CSD compared to other sources is due to the difference in chemical space covered.

The lack of polymorphism in the CSD is likely due to the limited effort spend on finding polymorphs. As stated in 4.3.1, industrial datasets tend to contain results of polymorph screens. Therefore, polymorphism in the CSD is likely a reflection of the research interests of the scientific community rather than actual polymorphism. Smaller, more easily crystallisable compounds (low nConf20) are more likely to be studied. This then results in more polymorphs for that structure to be found. This was first noted by McCrone by his now-famous statement that “the number of forms known for a given compound is proportional to the time and money spent in research on that compound” [220]. The validity of the statement was tested on the CSD single component dataset. The time and money spent on research were approximated by the number of redeterminations a compound has in the CSD. A redetermination is often the same polymorph studied under different conditions (a refinement of the same structure being another reason for redeterminations). It represents a repeated study of

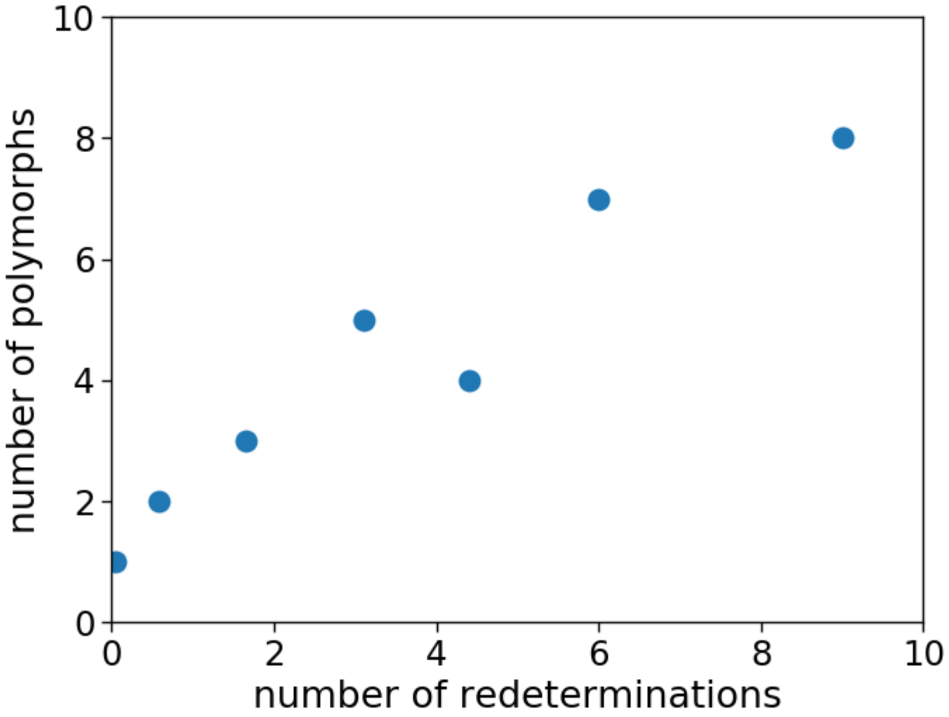


Figure 4.9: Number of polymorphs as a function of redeterminations.

The number of redeterminations is calculated by subtracting the number of polymorphs from the total number of structures for a given compound. Mean number of redeterminations were taken for each number of polymorphs. It is important to note that the high polymorph count is rare (single datapoints for 7 and 8 polymorphs compounds, 9 datapoints for 5 polymorphs compounds).

the same compound under different conditions. The relationship between the number of redeterminations and the number of polymorphs is shown in Figure 4.9.

4.3.4.2 Challenges

The issue of undiscovered polymorphs poses a significant challenge for the study of polymorph propensity. Firstly, unlike other properties such as melting points, it is difficult to assess the quality of the data. The number of redeterminations or number of publications on the compound of interest may be used as an indicator of the quality (i.e. the likelihood that all polymorphs have been found). However, it is not as rigorous as the assessment of error for other experimentally determined properties.

Rather than studying polymorph propensity through the number of polymorphs found, it could be studied by comparison of monomorphic and polymorphic structures to create a classifier for the task. However, the issues highlighted here are still likely to affect this analysis. This method also emphasizes the issue of the definition of monomorphism. There is a large set of conditions under which crystallisation may be attempted, making a definitive polymorph screen to find all physically possible polymorphs unrealistic. Similar to the polymorph screening done by the pharmaceutical industry, only the relevant conditions should be considered. At the time of writing, no such dataset is publicly available, so the use the number of redeterminations as a surrogate for the data quality remains the only suitable means of analysing data quality of polymorph counts for the study. This could be expanded by the inclusion of how wide of a range of conditions were investigated or by the number of experiments performed within a set range of conditions.

4.4 Conclusion

In this chapter, the factors affecting polymorph propensity were studied. MMPs were used to assess the effects of small molecular transformations on the propensity to form polymorphs. However, no statistically significant transformations were identified. This is partially due to the reduction in the dataset size due to the pursuit of quality in terms of polymorphism data (monomorphic adjustment) and MMPs (elimination of pairs where the change is larger than 30 % of either of the molecule based on heavy atom count). The issues related to the small number of MMPs identified within the dataset is explored in Chapter 6. The CSD single component dataset contains less polymorphic structure than other sources (1 % vs 66 % for Lily internal dataset). This is likely due to the active search for polymorphs within the pharmaceutical industry. The number of polymorphs in the CSD appears to correlate with the time spend researching that compound as approximated by the number of redeterminations. Due to the difficulty in assessing the quality of polymorphism data, high-quality dataset based on polymorph screened compounds is desirable for the propensity study. This was attempted using Pfizer internal database. However, the crystal structures found within the database are not grouped into polymorphs (redeterminations and polymorphs are not distinguished). This issue was taken as an opportunity to benchmark the existing automated methods of polymorph and redetermination classification as well as develop machine learning classifiers for the task. This work is discussed in the following chapter.

Chapter 5

Benchmarking of Automated Approaches for Differentiating Between Polymorphs and Redeterminations

5.1 Introduction

The ability to predict polymorph propensity is of crucial interest within the pharmaceutical research community (see 2.1.3 for more details). The research on the topic presented in the previous chapter identified several challenges. The data quality and quantity is one of such difficulties. It is difficult to ascertain the number of polymorphs a structure exhibits because the lack of polymorphs may be due to lack of emphasis on the determination of crystal structure, rather than lack of possible polymorphs. Further challenge arises from the differentiation between polymorphs and redeterminations. CCDC publishes a list of structures of each polymorph for all structures in the CSD, based on the spectra comparison method [211]. However, no similar list is available for in-house databases. This opportunity was taken to benchmark the existing spectra comparison method, and develop alternative methods of classification of pairs of structures into redeterminations and polymorphs.

In principle, the classification is best carried out "manually", i.e. by visual inspection and assessment by an expert in molecular crystallography. In practice, expert identification of the polymorphs reported in large datasets is challenging, as it is very labour intensive if the crystal structures were not annotated at the point of curation. Moreover, inconsistencies can arise during "manual" curation due to fatigue, insufficient expertise, or different experts assigning different labels to the same polymorphs. For example, a variety of labels have been reported in the literature for polymorphs of sulfathiazole [221]. While these inconsistencies could be avoided by an expert panel working according to a standard operating procedure; this would still require a considerable investment of time. Accurate, automated approaches to identifying polymorphs are hugely desirable, with a means of differentiating polymorphs and redeterminations of the same chemical being an important first step.

Automated approaches for identifying polymorphs are of value for both the CSD and industrial crystal databases. As many of the latter databases are structured similarly to the CSD, automated approaches which are applicable to the CSD should be widely applicable within industry. Currently, manual labelling of polymorphs is reported, if at all, at the point of deposition by individual researchers. This leads to incomplete and potentially inconsistent assignments. In industry, in-house databases may not have been annotated with the polymorph identity, even where this information may have been experimentally determined.

An automated method can provide a consistent classification of large datasets. One automatic method that was developed to classify polymorphs and redeterminations of structures within the CSD is based on the comparison of simulated powder diffraction spectra (hereafter referred to as spectra method) [211,222]. The refcode family is used to group crystal structures of the same molecular composition and then, within each family, a pairwise diffraction pattern comparison is undertaken. The peak positions of the spectra are determined by the unit cell parameters, while the intensities are calculated from the molecular structure and the space groups (i.e. the packing arrangement) [211,222]. A comparison of the peak locations and the intensities allows the similarity of the two crystal structures to be determined, hence the classification of the pair as polymorphs or redeterminations. The effect of the experimental conditions (temperature and pressure) are accounted for using unit cell volume normalisation and a peak shift tolerance factor. The peak shift tolerance factor was introduced to deal with the case of substantial differences in temperature or pressure, for which cell volume normalisation alone is not sufficient. The spectra method is used by the CCDC to generate the best_R_factor_list, which is a list that contains the refcode with the lowest R- factor, a measure of crystallographic data quality, for each polymorph within the database [211].

However, the spectra method is not as easily implemented for CSD-like databases with less curated information (such as no refcode family assignment), although chemical structure representation (InChI, SMILES) based grouping of entries retrieved using the CSD Python Application Programming Interface (API) [215], might address this issue. Moreover, the spectra method was only benchmarked on a small set of 386 structures (83 refcode families) [211,222]. This is a small dataset compared to the entirety of the CSD that the method is applied to. Version used in the study (5.39) contained, 950,516 crystal structures while currently there are over 1 million structures. In particular, when developing machine learning algorithms for this task, it is desirable to have larger datasets which can be used to benchmark the method.

This chapter presents a more thorough, robust evaluation of the spectra method than has previously been reported in the literature. The performance of the spectra method is compared to the performance of machine learning methods for classifying pairs of structures into polymorphs and redeterminations. A large dataset with manually assigned polymorph labels, filtered to remove inter-expert inconsistencies, was constructed to allow for the benchmarking of the classifiers. This analysis identified the most suitable automated approach for discriminating polymorphs from redeterminations to automatically identify polymorphs in the CSD or any CSD-like in-house database.

5.2 Methods and Data

5.2.1 Datasets

The Manual label, Best R, and Benchmark datasets were derived from version 5.39 of the CSD database. Prior to splitting, all crystal structure entries were filtered only to retain structures for which the latest implementation of the spectra method from the CCDC could be applied without raising any errors. All suitable refcodes were grouped by the refcode family, and all possible combinations of refcode pairs within each group were identified (81,401 pairs). Manual labels are not available for every structure, and similarly, the best R factor list cannot be used for every pair, so the putative label assignment was attempted by both methods for all pairs before they were split into datasets. For each pair, a classification of 1 was given if the labelling method determined that the pair are different polymorphs and 0 for pairs of redeterminations. No value was given in case the labelling method could not be used to give a classification (e.g. when one of the structures lacks a manual label). The way in which the classification was done for each of the labelling methods (best R and manual label) is described in the following sections (5.2.1.1 and 5.2.1.2 respectively). Based on the obtained classifications, the pairs were split into the three datasets and the respective subsets (section 5.2.1.3) as illustrated in Figure 5.1.

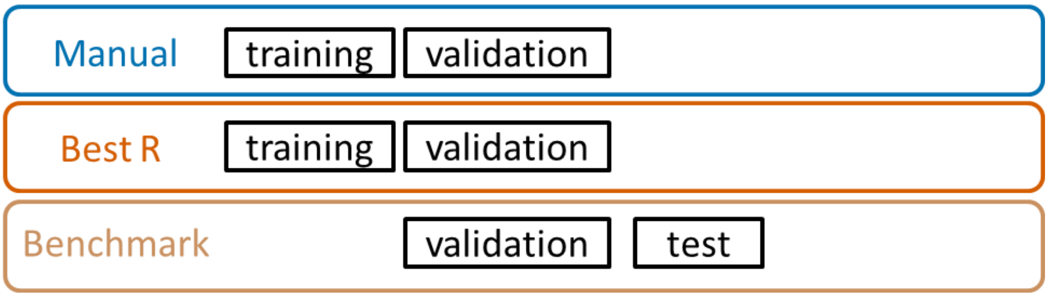


Figure 5.1: Datasets used in the polymorph redetermination

Manual, Best R, Benchmark, and In-house. These can be further divided into training, validation, testing, and application subsets.

5.2.1.1 Best R dataset

For each pair of refcodes, the classification was assigned based on the process described in Figure 5.2. If both refcodes are in the best R factor list, the two structures represent two different polymorphs. In any other case, if there is only one refcode in the best R factor list for that refcode family, the pair of refcodes is the redetermination of the same structure. In case there is more than one refcode in the best R factor list, it is not possible to determine whether the pair is a redetermination or different polymorphs and so no label is assigned. This limits the number of available pairs for the best R dataset. 51,649 pairs were given a classification based on Best R factor list (referred to as Best R classification hereafter).

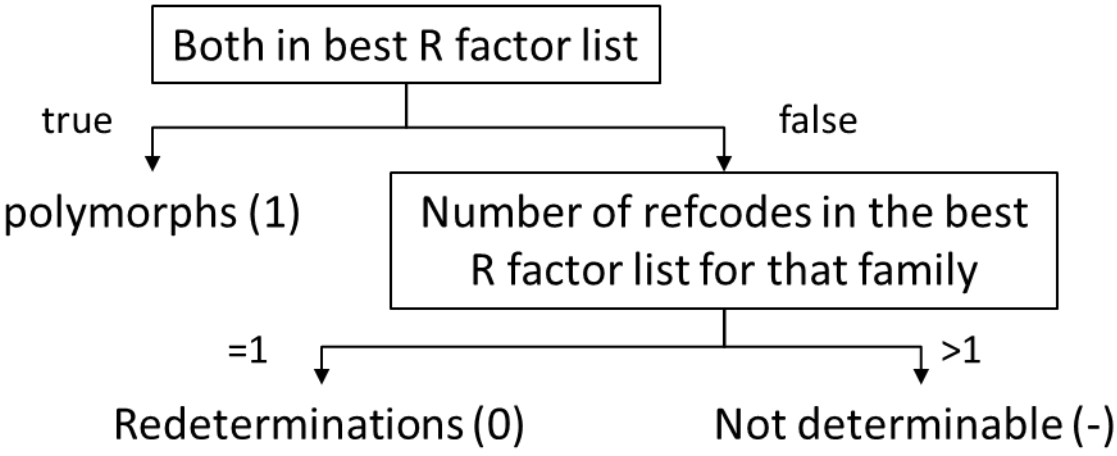

Both in best R factor list
true
false
polymorphs (1)
Number of refcodes in the best R factor list for that family
=1
>1
Redeterminations (0)
Not determinable (-)

Figure 5.2: Label assignment process flow chart for labels based on the best R factor list.

5.2.1.2 Manual label and benchmark datasets

The manual label and benchmark datasets are based on the polymorph label assigned by the authors submitting the structure to the CSD; this is available for 4 % of the structures studied. The distinction between polymorphs and redetermination was made by comparing the manual polymorph labels. If the labels matched, the pair was considered a set of redeterminations (classification = 0), and polymorphs (classification = 1) otherwise. In cases where there was no label for one or both of the structure, no label assignment was carried out (the label is referred to as Manual classification hereafter).

For each pair of structures, along with the Manual classification, information concerning whether the structures came from the same literature source was also noted. To remove inconsistency in labels due to different polymorph labels by different researcher [221], for the benchmark dataset, only structures that came from the same literature source were considered. Furthermore, an effort was made to eliminate any errors within the labels. However, some false polymorphs may have been identified due to mismatch of the label caused by spelling mistakes. For example, JIBCIG04 and JIBCIG06 are redeterminations of the same polymorph with labels: ‘othorhombic’ and ‘orthorhombic’ (the missing ‘r’ was subsequently corrected in the CSD). The curation workflow attempted to eliminate all such cases; however, the possibility of some noise in the data cannot be entirely eliminated due to the large number of pairs considered (17,364).

5.2.1.3 Dataset split

After each pair was labelled using the best R factor list and manual labels, along with checking if the literature source is the same, the datasets were split into six subsets according to Figure 5.1. Three intersecting sets were constructed based on the availability of labelling discussed in 5.2.1.1 and 5.2.1.2.

$$Best\ R = \{all\ pairs\ that\ have\ Best\ R\ classification\}$$ Equation 5.1

$$Manual = \{all\ pairs\ that\ have\ Manual\ classification\}$$ Equation 5.2

$$Literature = Manual \cap \{all\ pairs\ that\ come\ from\ same\ source\}$$ Equation 5.3

The total number of available pairs ($Best\ R\ \cup Manual$) for dataset creation is 76,309. The detailed breakdown of the sets is shown in Figure 5.3. The manual label training and best R training datasets were selected from pairs that had an only manual label and best R label respectively ($Manual/Best\ R, Best\ R/Manual$). This was to ensure that the effects of training machine learning models for distinguishing polymorphs from redeterminations using different means of assigning the training set labels were not confounded by differences in the crystal structures used for training the models. Due to the limited number of pairs available for Manual training dataset, the training set size was limited to 24,660 pairs.

All the validation and test sets come from pairs that have both manual and best R labels ($Valid\_\_and\_test = Manual \cap Best\_R$). This is done to allow for further comparison between the models trained on the two datasets. The benchmark validation and test sets were selected where each structure came from the same literature source ($Valid\_and\_test \cap Literature$). Best R and manual validation sets come from the remaining pairs that have both labels ($Valid\_and\_test/Literature$). To ensure the same size for all validation sets, the set size of 2,594 pairs was used. The Benchmark test set consists of 3,415 pairs. The dataset splits are shown in Figure 5.3.

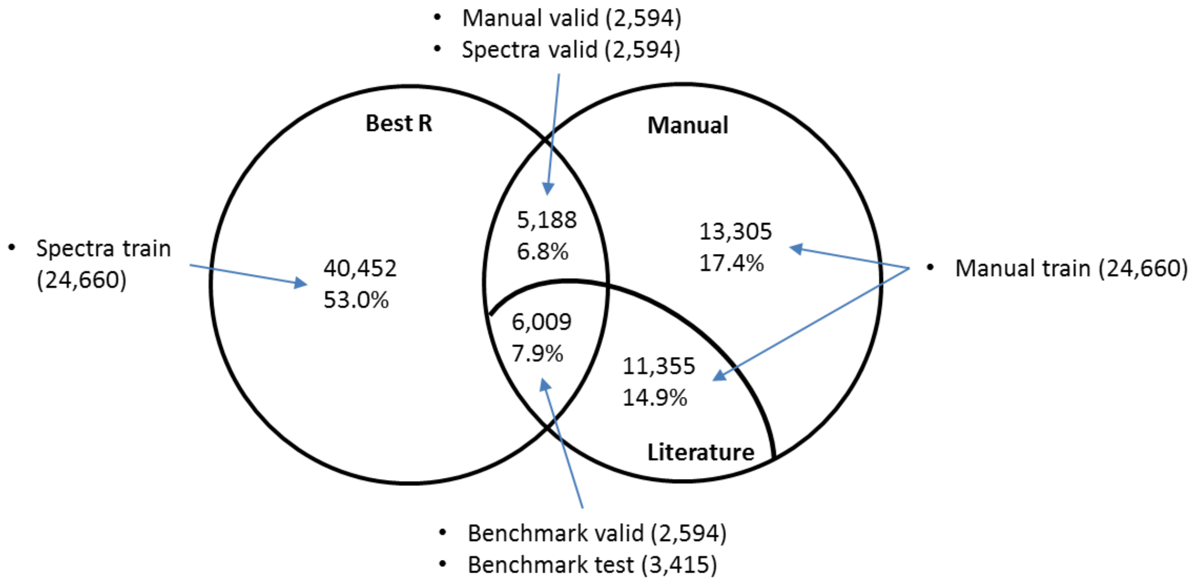


Figure 5.3: Availability of labels from the best R factor list and manual labels. For the pairs that have a manual label, whether both structures come from the same literature source was also noted.

5.2.2 Descriptors

To build machine learning models for classifying pairs of crystal structures, contained within CSD-like in-house databases, as polymorphs or redeterminations, suitable descriptors needed to be identified. Initial descriptor selection was carried out based on data available within the in-house database and understanding of polymorphism. Different combinations of descriptors were evaluated based upon analysis of CSD data, including the effect of removing certain descriptors on the performance of the machine learning methods, as evaluated using the validation sets.

The following experimental data (structural data and experimental metadata) were available for crystal structures in the in-house database and were considered relevant to classifying structures, with the same molecular structure, were polymorphs or redeterminations. Based upon the assessment described below, a subset of these parameters was chosen to be used as descriptors, alongside the calculated packing similarity (see below), for the machine learning models.

All entries in the in-house database and the CSD included information such as cell parameters (lengths and angles), crystal system, and density, while the experiment temperature and the R-factor are missing for some cases. Each numeric descriptor was calculated as the difference across pairs of structures, i.e. difference in temperatures was taken as the temperature descriptor. Otherwise, for qualitative variables, such as crystal system, pairs where the values of these variables matched or did not match were assigned a value of 0 or 1 respectively for the corresponding descriptor.

Cell parameters (angles and lengths) are expected to change across a pair of polymorphs as different packing arrangements are likely to affect the unit cell dimensions. In many cases, the crystal system (i.e. lattice type) differs between polymorphs. In principle, this should not be different for redeterminations, but the documented crystal system may occasionally differ for some redeterminations. (For example, a slight difference in apparent cell lengths may lead a cubic polymorph to be considered orthorhombic for some redeterminations.) However, differentiating between polymorphs and redeterminations is more difficult for structures with the same lattice type. If the lattice type is the same across the pair of structures, a value of 0 is assigned (1 is assigned if the system is different).

Regarding differences in symmetry, there are over 230 space groups possible with some degree of similarity [59]. A method of grouping similar space groups was not readily available, so differences in space group, i.e. symmetry differences, were not encoded as a descriptor. For a given molecule, only a combination of cell parameters and the space group can change the density, i.e. the ratio between the mass of all atoms in a unit cell and the volume of the cell (the mass of all atoms in the unit cell is determined by the number of molecules per unit cell – determined by the symmetry operators, and the volume is determined by the cell parameters.) Therefore, it is expected that any information captured by density is largely included within the cell parameters. Crystallisation temperature along with other experiement conditions can affect the cell parameters, and the same polymorph can have different apparent cell parameters if studied at a different temperature. The R-factor is an indicator of how well the structure calculated from a crystallographic model agrees with the experimental X-ray diffraction data. In some cases, a redetermination with improved R-factor can have different apparent cell parameters to the original structure [223]. To capture these phenomena, the changes in R factor and the temperature across the pairs of structures were used as descriptors.

Further to the descriptors available within the databases, a comparison of packing can be made. Crystal polymorphism can be defined as structures with different packing arrangements. COMPACK[224], as it is available through CSD API[189], can be used to quantify the packing similarity between pairs of crystal structures. A molecule is selected from the crystal structure and a 15 molecule packing shell is generated based on the crystal packing. The packing shells for the two crystal structures are superimposed and are aligned to minimise the distance between matched atoms of each molecule from each cluster. The number of molecules that fit within the predefined distance tolerance (0.2 Å [224]) is returned. A high number indicates that the packing is similar for the crystal structure pair. In this study, the number was divided by 15 to scale it within the range of 0 to 1 (1, meaning 15 out of 15 molecules were within the tolerance distance). The descriptor is referred to as packing similarity hereafter.

The differences in cell parameters (i.e. cell lengths a, b, c and angles alpha, beta, gamma), an indicator variable denoting differences in the crystal system as explained above, differences in R-factors, differences in the temperature of crystallisation, and packing similarity were initially selected as descriptors for the model development. All of these descriptors were available for the set of entries chosen from the CSD to form the datasets summarised in Figure 5.1. Analysis performed using the training and validation data was also used to refine the set of descriptors chosen for the final machine learning model.

5.2.3 Descriptor analysis

Prior to developing models using supervised machine learning, the descriptors were analysed. The purpose of this step was to (1) assess whether the distributions between the different CSD derived datasets used for training, validating and testing were comparable, and (2) develop descriptor sets suitable for the classifier development.

5.2.3.1 Correlation matrix

A correlation matrix was used to eliminate descriptors that are highly correlated based on the training sets. The Pearson correlation coefficient was used to quantify the correlation between descriptors [225]. The coefficient for two descriptors (X, Y) is given by equation (4), where *n* is the number of samples, i.e. pairs of polymorphs or redeterminations, x_i is the value of X for the *i* th sample and $\bar{x}$ is the arithmetic mean of X (analogous for Y). The correlation coefficient was calculated for all combinations of descriptors.

$$r_{XY} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\ \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \qquad \text{Equation 5.4}$$

5.2.3.2 Principal Component Analysis

Principal component analysis (PCA)[226] was performed for two reasons: (1) to develop an understanding of key descriptors that determine whether a pair of structures are polymorphs or redeterminations; (2) to assess whether the different training, validation and test sets were sufficiently similar in order for the validation and test sets to lie within the applicability domain of the machine learning models developed using the corresponding training sets. The descriptors selected have different magnitudes of scale. Hence, to ensure none of the features dominates the analysis, min-max scaling was used. This transforms each descriptor to fit within the range of 0 to 1 based on the minimum and maximum values.

The PCA was fitted to the Best R training set and Manual label training set, resulting in two sets of loadings for the original descriptors, referred to as PCA-s and PCA-m respectively. The two sets of loadings were then used to transform the manual label validation set, Best R validation set, and the benchmark dataset to compare the datasets based upon their distribution with respect to the principal components associated with the highest contribution to the variance in the scaled descriptors. The loadings associated with the principal components which best-separated pairs of polymorphs from redeterminations were also examined, providing insight into the key descriptors providing linear discrimination between polymorphs and redeterminations. However, the supervised machine learning methods used to build the classifiers were also able to take account of non-linear relationships. This was necessary as linear separation is not sufficient for classification. For example, unit cell length difference may arise from polymorphism, but also from temperature difference, hence the interdependent nature of some the features need to be taken in account.

5.2.4 Classifier development

5.2.4.1 Development process

The model development stage was divided into three steps; training, validation, and testing. The training subsets from the two datasets (Best R and Manual) were used to train classifiers, using various machine learning algorithms, to distinguish between polymorphic and redetermination pairs (Figure 5.4). The evaluated machine learning algorithms are described below. The best algorithm and algorithm parameters, i.e. hyperparameters, from each training set was selected for the validation step (selection criteria discussed in below). The performance of the selected models was assessed using the validation subsets of the Best R method, Manual label, and benchmark datasets. Validation using three datasets was used to study the generalisability of the models and to study the difference in the datasets. The process was repeated for various descriptor sets developed based on the dataset analysis.

Finally, the single best performing machine learning model based on the benchmark validation set, along with the spectra method, was applied to the benchmark test dataset to assess the performance on a high-quality external dataset.

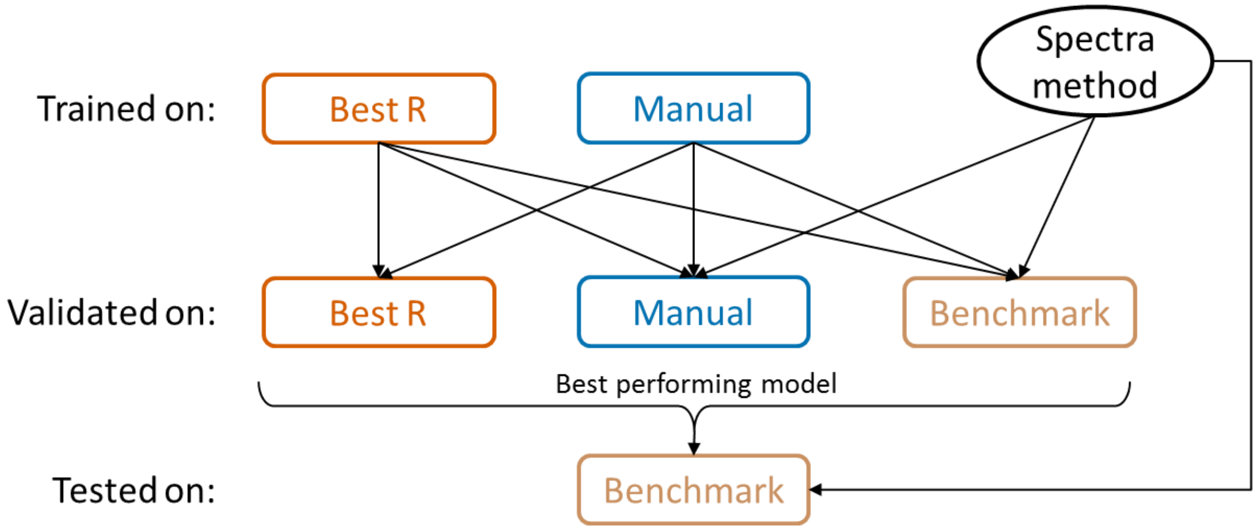


Figure 5.4: Overview of the model development process

In this work, the following Random Forest (2.4.4) hyperparameter values were investigated: number of trees 10 – 200, split criterion Gini or entropy, max depth 1 – 110. The following Support Vector Machine (2.4.5) hyperparameter values were investigated: Gamma (10^{-4} to 10^{3}) and C (10^{-4} to 10^{3}). Random search was used to probe the hyperparameter search space.

The *F1* score (2.4.3, Equation 2.21) is used as the primary performance metric for the models. It was used for selection of the single best machine learning model out of all models, based upon different combinations of descriptors and training sets, applied to the validation sets. It was also used for the selection of the most suitable algorithm, as well as for the optimisation of SVM and RF models, i.e. the selection of the best combinations of algorithms and hyperparameters was based upon the combination which led to the largest mean *F1* score obtained from cross-validation on the training set.

5.2.5 Computational details

All computational work in this paper was performed using Python 2.7 (64-bit, as installed using the Anaconda Distribution version 4.3.34). Any interaction with the crystal structure repositories was handled using the CSD Python API (1.5.2) [215].

The API was also used for calculation of packing similarity between crystal structures. The pandas (0.19.2) library was used for handling of data and for construction of the correlation matrix [216]. Principal Component Analysis, Random Forest, Support Vector Machine, and hyperparameter selection using cross-validation were performed using SciKit-Learn (0.18.1) [123]. See Appendix 3 for the scripts used.

5.3 Results and Discussion

5.3.1 Descriptor Selection

5.3.1.1 Correlation matrix

A correlation matrix was created to eliminate any highly correlated descriptors (Figure 5.5). There were no significant correlations between the features, so all selected descriptors were used.

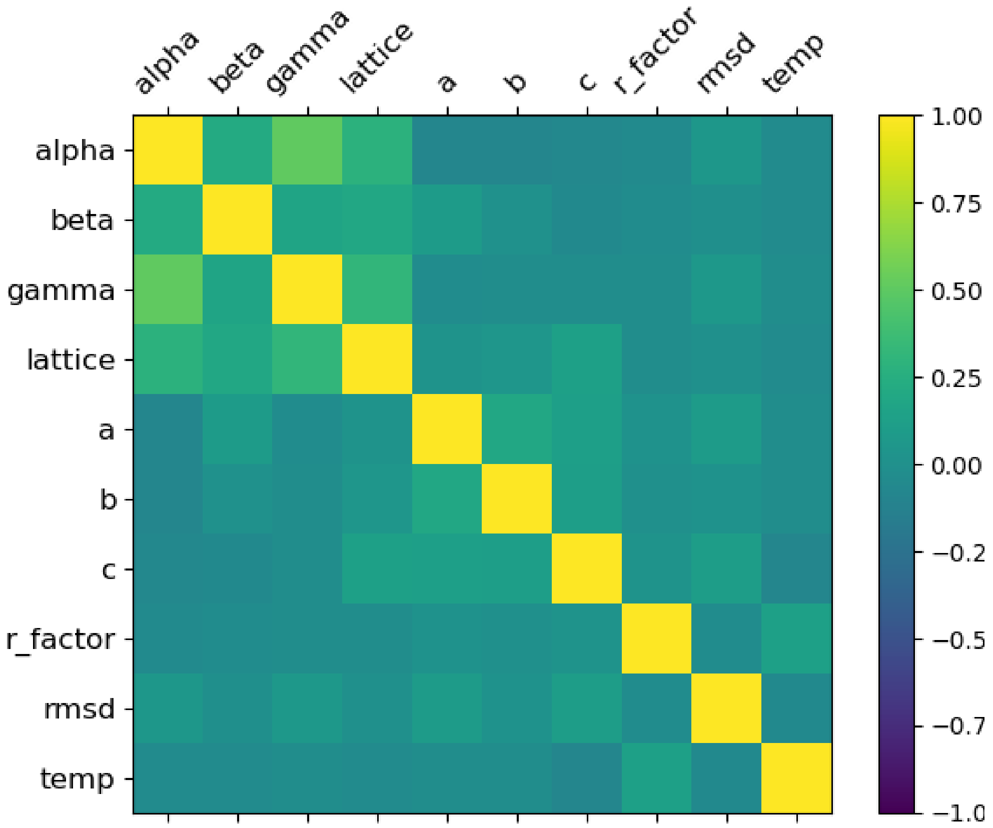


Figure 5.5: Pearson Correlation coefficient matrix of the selected descriptors within the Best R training set

5.3.1.2 Principal Component Analysis

The PCA was fitted to the manual label training set (PCA-m) and best R training set (PCA-r). The descriptor contributions to the first four principal components are summarised in Table 5.1. The explained variance of each of the principal component is shown in Figure 5.6. The first component (PC1) explains significantly more of the variance compared to the subsequent components. The largest contribution to the first

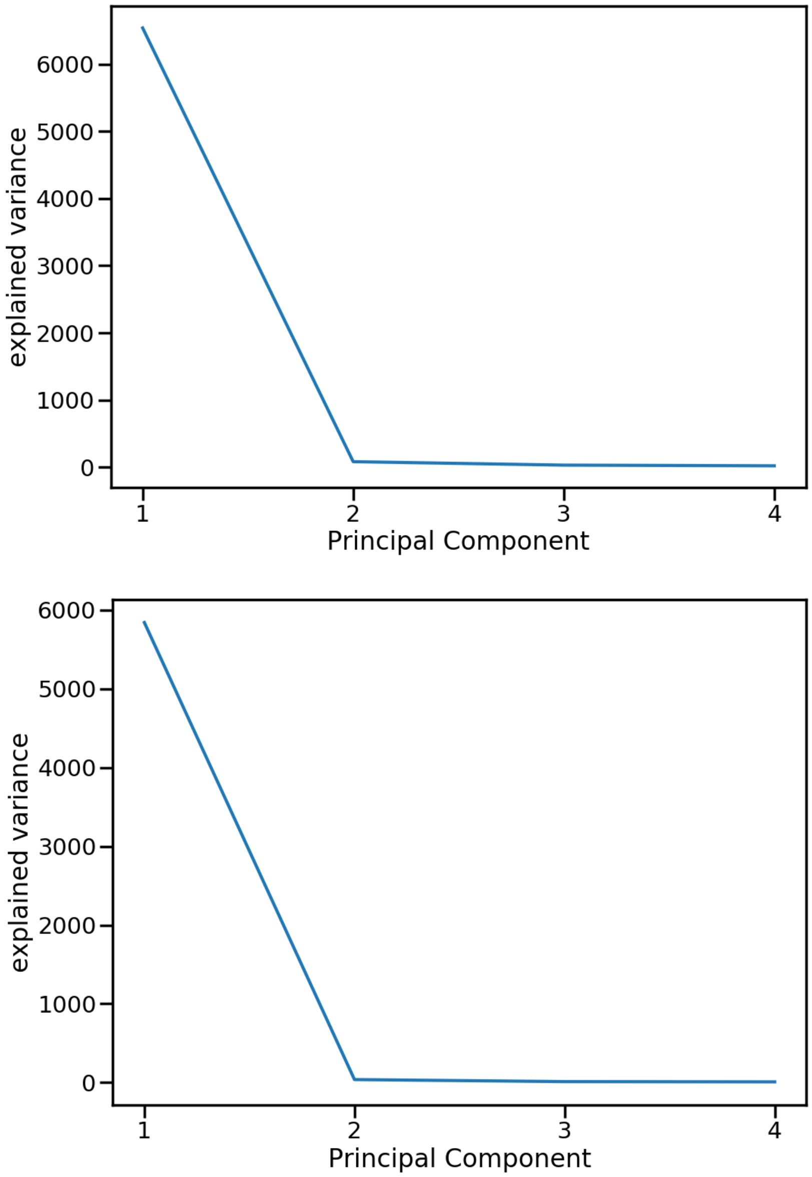


Figure 5.6: explained variance of PCA fit to manual train dataset (top) and Best R dataset (bottom).

principal component (PC 1) comes from packing similarity (-0.796 and -0.902 respectively) followed by crystal system (0.564 and 0.390 respectively). However, there are differences in contributions to PC 2 and PC 3. For manual label dataset, crystal system and packing similarity are the predominant contributors to the PC 2, while the temperature and R factor are the largest contributors to the PC 3. The contributions to PC 2 and PC 3 for best R dataset are similar to the contributions to PC 3 and PC 2 for the manual label dataset respectively.

Table 5.1: First four principal components for the Manual label and Best R training sets.

Descriptor	***Fitted to Manual Training***				***Fitted to Best R training***			
	PC 1	**PC 2**	**PC 3**	**PC 4**	**PC 1**	**PC 2**	**PC 3**	**PC 4**
Alpha	0.049	0.047	-0.028	-0.042	0.051	0.021	0.082	-0.019
Beta	0.150	0.053	0.029	0.019	0.087	0.033	0.096	-0.006
Gamma	0.095	0.119	-0.001	-0.034	0.071	0.022	0.117	-0.020
a	0.077	-0.007	-0.037	0.067	0.062	0.018	0.054	0.011
b	0.064	-0.020	0.001	0.053	0.080	0.021	0.075	0.017
c	0.044	-0.022	0.005	0.034	0.072	0.020	0.076	0.015
Crystal system	0.564	0.796	-0.050	-0.001	0.390	0.086	0.874	-0.075
Temperature	0.027	0.030	0.843	-0.531	0.005	0.882	-0.135	-0.450
Packing similarity	-0.796	0.588	0.022	0.036	-0.902	0.082	0.417	0.023
R factor	0.043	0.003	0.533	0.840	0.058	0.452	-0.004	0.889

Packing similarity and crystal system are the two factors that contribute the most to the principal components and were further analysed to develop suitable descriptor sets for the classifier development.

The Best R validation dataset transformed using PCA-m is shown in Figure 5.7. The figure shows that the polymorphs and redeterminations are not linearly separatable. Thus, the use of machine learning algorithms capable of capturing a more complex relationship between the descriptors and the target (polymorph or redetermination) is required.

5.3.1.3 Packing similarity

Polymorphism is the difference in the crystal packing. For this reason, it can be argued that packing similarity alone should be sufficient in developing a classifier for polymorph-redetermination. Based on the PCA, packing similarity is an important descriptor, and so it was investigated how well it can separate polymorphic and redeterminations pairs (Figure 5.8). As the figure shows, redeterminations tend to have high packing similarity. 97.6 % of redetermination pairs in the benchmark validation set have packing similarity of >0.8. For this reason, a single descriptor model was considered in the classifier development stage (5.3.2).

Although it is a useful indicator, there is an overlap between the polymorphs and redeterminations. For the benchmark validation set, 10.6 % of polymorphic pairs have packing similarity of >0.8. Calculation of the packing similarity is computationally expensive, so a descriptor set that does not use it was also used in the classifier development (5.3.2).

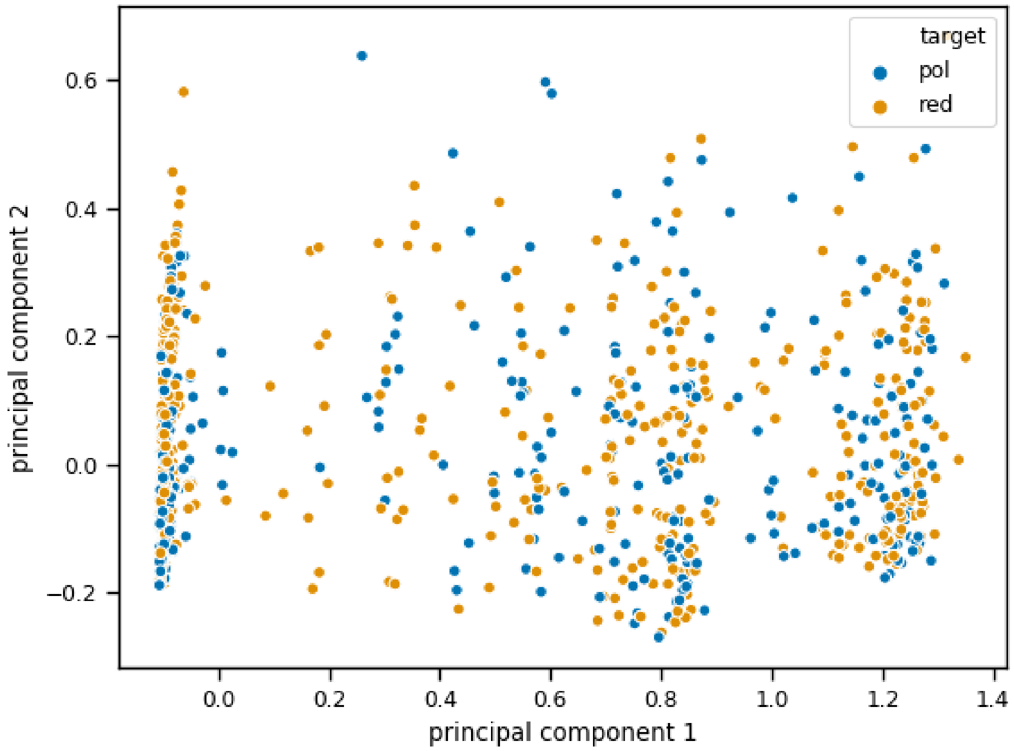


Figure 5.7: Best R validation dataset transformed using the PCA-m (PCA fitted to the manual label training dataset.

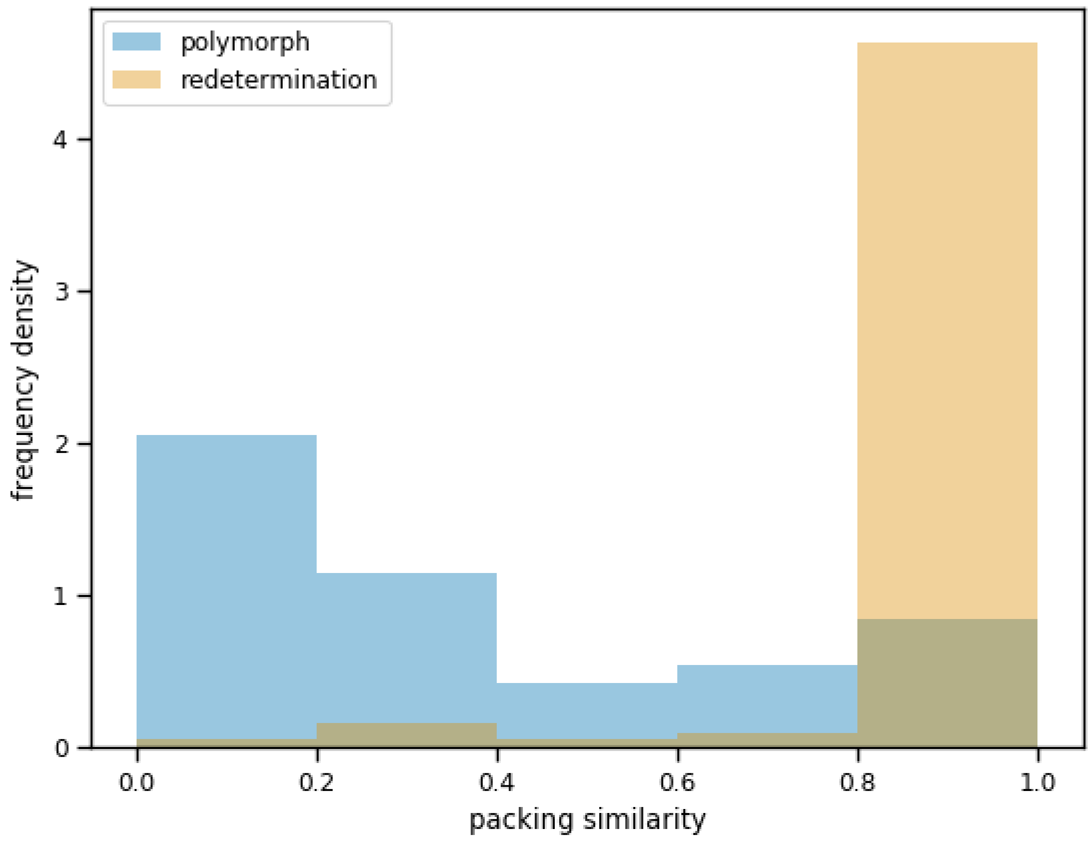


Figure 5.8: Comparison of packing similarity between pairs of polymorphs and redetermination for the benchmark validation set. The figure is normalised to the area under the graph = 1.

5.3.1.4 Lattice type

The lattice type is the second descriptor with the largest contribution to the first principal component. The relationship between the lattice type descriptor and the polymorph and redetermination classification using spectra method (left) and the manual label (right) is shown in Figure 5.9. There are no redeterminations based on the manual label that have different lattice types, whereas 18 % of redeterminations based on the spectra method have different lattice types. TMACNZ07 and TMACN09 are a pair of structures that are in the 18 %. The only difference in cell angles is a change of 0.42^{o} of β from 90^{o}. For this reason, TMACNZ07 is classed as a monoclinic lattice and TMACNZ09 is orthogonal. Comparing the two crystal structures further, the average percentage change between the cell lengths is 0.7 %, with the change in b from 15.309 Å to 15.105 Å being the largest difference. The packing similarity is 1.0, indicating that all molecules within the 15 molecule packing shell align within the tolerance of 0.2 Å. It is plausible that the two structures are the same polymorph, with the difference in the unit cell parameters being due to the difference in the R factor (6.4 and 2.6) and the temperature (200 K and 100 K). Another similar example is the pair of COQNUR and CONQNUR01, where different lattice type was assigned due to a difference of the angle β of 0.02°, while the difference in cell lengths is below 0.9% and the packing similarity is 1.0. It is possible that in some of the cases, a different polymorph label was assigned based on the different lattice type assignment without a thorough comparison of the crystal packing. For this reason, a descriptor set that excludes lattice type was used for the classifier development (5.2.4).

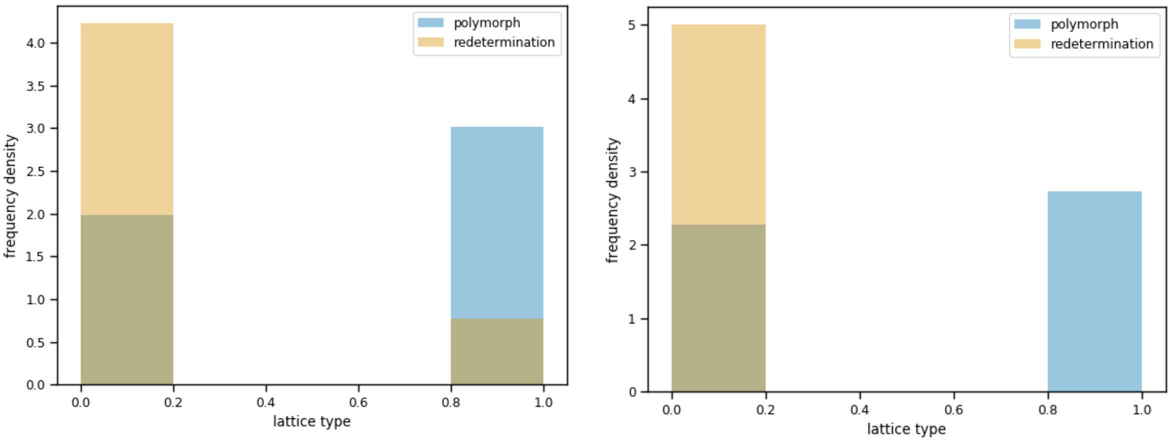


Figure 5.9: Comparison between polymorphs and redeterminations for the best R validation set. The classification based on spectra method (left) and manual label (right). The graph is normalised to the area under the graph = 1.

5.3.2 Classifier development

5.3.2.1 Training

Two training datasets, Manual training and Best R training sets were used to train classifiers. For every combination of the training dataset and descriptor set (All, no packing similarity, and no lattice), a RF and SVM classifiers were trained and optimised. The F1 score on the cross-validation set for the best performing models is summarised in Table 5.2. RF outperformed SVM for all descriptor sets, trained on the Best R training dataset. The performance of the two algorithms was much closer in case of models trained on the Manual training dataset, but the SVM models had the higher F1 score. For the single descriptor model that only uses packing similarity, only a random forest model was trained. It obtained a F1 score of 0.867.

The best performing algorithm for each combination of training set and descriptor set were selected for the validation step of the classifier development.

Table 5.2: F1 scores of the trained classifiers.

Descriptor set	Manual		Best R	
	RF	SVM	RF	SVM
All	0.897	0.899	0.882	0.794
No Packing	0.892	0.900	0.811	0.681
No Lattice	0.893	0.898	0.880	0.793

5.3.2.2 Validation

The models with the highest F1 scores from each combination of training dataset and descriptor set were applied to the three validation sets (Manual, Best R, and Benchmark). The results are summarised in Table 5.3.

The performance on the homogenous validation set (i.e. performance of model trained with manual label training set on manual label validation set and vice versa), heterogeneous validation set (i.e. performance of model trained with manual label training set on spectra method validation set and vice versa), and benchmark validation set was analysed.

All models performed better on the homogeneous validation sets compared to heterogenous validation sets with the exception of model 4, which had a similar F1 score for both (0.886 and 0.887). Out of the models trained on the Manual training dataset (model 1 - 4), the single descriptor model (4) had the highest F1 score, in contrast to model 2 which did not use packing similarity as a descriptor and had the lowest F1 score. This further strengthens the argument for the usefulness of packing similarity for the polymorph redetermination classification. Out of the models trained on the Best R training dataset (model 5 – 8), single descriptor model (8) had the lowest F1 score due to the very low recall (0.175) while the precision is high (0.933). Unlike in the case of model 3, in model 7, the exclusion of the lattice type as a descriptor does not improve the performance. As discussed in 5.3.1.4, in some cases, lattice type may be incorrectly used to classify polymorphs and redeterminations manually. This is not the base for Best R dataset, so no improvement was observed by dropping it as a descriptor. Omitting packing similarity, reduced the F1 score to 0.886 (from 0.938).

Table 5.3: Performance on the validation sets of classifiers trained on Manual and Best R training dataset, using different descriptor sets.

Trained on	***Model ID***	***Descriptors***	***Performance on***		
			Manual valid	**Best R valid**	**Benchmark valid**
Manual	1	**ALL**	F1 : 0.883 Precision: 0.861 Recall: 0.906	F1 : 0.801 Precision: 0.679 Recall: 0.977	F1 : 0.920 Precision: 0.966 Recall: 0.878
	2	**NO PACKING**	F1 : 0.879	F1 : 0.797	F1 : 0.911
	3	**NO LATTICE**	F1 : 0.881 Precision: 0.860 Recall: 0.903	F1 : 0.803 Precision: 0.679 Recall: 0.982	F1 : 0.918 Precision: 0.965 Recall: 0.875
	4	**PACKING ONLY**	F1 : 0.886 Precision: 0.956 Recall: 0.825	F1 : 0.887 Precision: 0.814 Recall: 0.974	F1 : 0.907 Precision: 0.952 Recall: 0.868
Best R	5	**ALL**	F1 : 0.852 Precision: 0.988 Recall: 0.749	F1 : 0.938 Precision: 0.925 Recall: 0.952	F1 : 0.816 Precision: 0.989 Recall: 0.694
	6	**NO PACKING**	F1 : 0.819	F1 : 0.886	F1 : 0.790
	7	**NO LATTICE**	F1 : 0.850 Precision: 0.988 Recall:0.746	F1 : 0.934 Precision: 0.923 Recall: 0.945	F1 : 0.813 Precision: 0.989 Recall: 0.875
	8	**PACKING ONLY**	F1 : 0.217 Precision: 0.982 Recall: 0.121	F1 : 0.295 Precision: 0.933 Recall: 0.175	F1 : 0.176 Precision: 0.896 Recall: 0.097

The performance on the heterogenous validation datasets is worse than on the homogeneous datasets. The only exception is model 4, which has a consistent performance across the two validation datasets. However, the similar F1 score is caused by a proportional drop in precision and an increase in recall.

The performance on the benchmark validation set was higher for the models trained on the Manual dataset (model 1 – 4) compared to the models trained on the Best R dataset (model 5 – 8). The PCA did not indicate any clear differences between the two datasets; however, the consistent difference in performances indicate that some difference exists.

Model 8 has the worst performance overall, caused by low recall values. However, the same descriptor set trained on the Manual dataset achieved F1 score of 0.907. Not using lattice type as a descriptor has a minimal effect on the performance of the models for the two groups (model 1 and 3, and model 5 and 7). The best performing models for each training set are ones that use all descriptors (model 1 and 5) with model 1 having the highest F1 core (0.920). This model was selected for the testing stage discussed below.

5.3.2.3 Test

The best performing model (trained on manual label dataset with all descriptor set) achieved a F1 score of 0.910 (recall = 0.864, precision = 0.962) on the benchmark test set (Table 5.4). The performance is similar to the one achieved on the benchmark validation set. The spectra method was also compared to the manual labels from the benchmark test set; the confusion matrix for which is presented in Table 5.4. The F1 score of the spectra method is 0.780 with recall of 0.645 and precision of 0.988. The spectra method had fewer false positives compared to the model 1, but a higher rate of false positives. Comparison of the misclassifications by the two methods are visualised in Figure 5.10. It was attempted to find differences in the descriptor distributions across the different subsets of the misclassified pairs. However, none of these were statistically significant (at 5 % confidence level).

Table 5.4: Confusion matrix of the trained machine learning model 1 and the spectra method on the test set

		Model 1		Spectra method	
		Red.	**Pol.**	**Red.**	**Pol.**
Manual label	**Redetermination**	943	67	994	26
	Polymorph	265	1676	689	1252

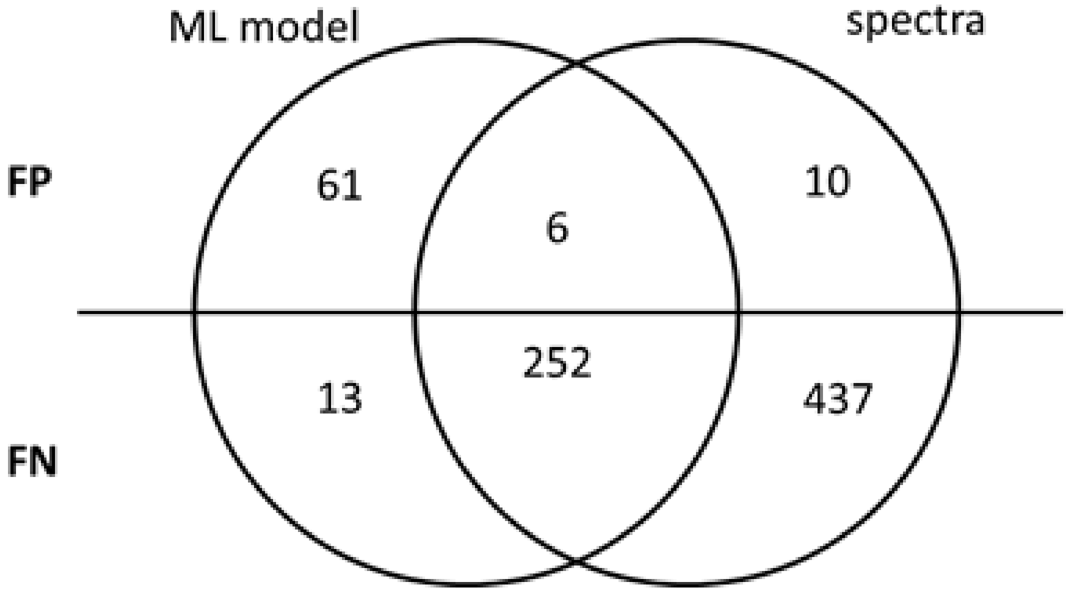


Figure 5.10: Comparison of false negatives and false positives of the trained machine learning model 1 and the spectra method.

5.4 Conclusion

A dataset for benchmarking the performance of automated methods of classifying polymorphic and redetermination pairs of crystal structures was developed. The dataset consists of pairs of structures that have been manually assigned a polymorph label and came from the same publication to ensure consistency of labels. 6,009 such pairs were identified in total, making this the largest available benchmarking dataset for assessing the polymorph redetermination classification.

A number of machine learning models were developed for the task of classifying structures into polymorphs and redeterminations. The model with the highest F1 score was selected and its performance was compared to the currently used method of based on spectra comparison. The best performing model achieved an F1 score of 0.910, while F1 score for the spectra method was 0.780. The machine learning approach appears to be a promising avenue for the development of automated methods for classification of polymorphs and redeterminations.

The work in this chapter shed some light on the data quality regarding polymorph propensity study presented in Chapter 4. The spectra comparison method was used for the polymorph count. This chapter showed that the polymorph count derived in this way, may not be accurate (F1 score of 0.780). This likely contributed to the lack of statistically significant trends observed on the polymorph propensity study. However, the correlation between research intensity and polymorph count along with challenging properties of the dataset itself are likely to have a more significant contribution to the lack of trends observed. In the following chapter, the property of the dataset itself, namely how suitable it is for MMPA is assessed.

Chapter 6

Matched Molecular Graphs

6.1 Introduction

In Chapter 4, several issues relating to MMPA of polymorph propensity were identified. In this chapter, the datasets are examined and compared to other literature data sources to assess the suitability for MMPA. Herein, the focus is upon issues which prevent MMPA yielding statistically significant results, even when those trends exist and would be statistically significant given adequate datasets.

The MMP approach was first utilised to analyse the most common substitutions found in drug-like substances [181]. Since then, it has been used for lead optimisation tasks within the Discovery stage of pharmaceutical product development [11,12,184,190,196,199]. A range of properties related to the early stage of the drug development was studied, such as molecular solubility (as typically taken as a molecular property rather than an equilibrium between the solid state and the continuous phase) [190,192], activity [12,227,228] and clearance [55]. With one notable exception where the effects of molecular transformation on crystal packing were studied [189], the applications of MMPA are typically limited to properties of interest during Discovery (i.e. where solid form may not be known or not focused upon). As such, the datasets that are used for the analysis are predominantly derived from Discovery datasets.

The datasets from different stages of pharmaceutical product development are compared to develop a better understanding of the potential reasons for the low MMP count for the CSD dataset used in Chapter 4. This is accomplished using Matched Molecular Graphs (MMG), details of which are presented in 6.2.2

6.2 Method and Data

6.2.1 Dataset

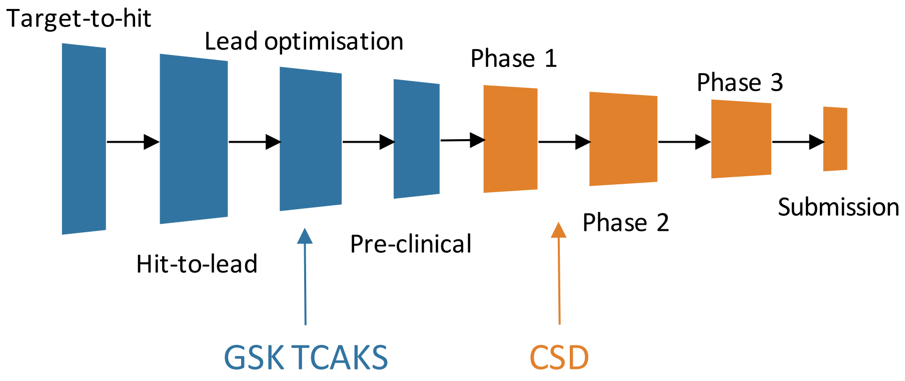


Figure 6.1: Datasets selected for MMG study across the pharmaceutical development process.

The process diagram adapted from [19]**. Discovery stages illustrated in blue. Development stages shown in orange.**

Datasets from a range of stages of the pharmaceutical product development were selected (Figure 6.1). ChEMBL-NTD set 14 (GSK TCAKS (Tres Cantos Anti-Kinetoplastids Set) dataset) [229] was taken as a representative dataset of the Discovery stage where MMPA is typically applied. The dataset was selected due to its size and the fact that it came from a single pharmaceutical company. The CSD monomorphic adjusted single component dataset (as defined in Chapter 4) corresponds to the Development stage. Patent melting point dataset (2.3.3) [83] was used to systematically study the effects of dataset size on the properties of the MMG.

6.2.2 Graph construction

The MMG method uses a graph constructed from the MMPs. The basic concepts of graphs were introduced in 2.4.1.2. In this chapter, some of the properties of graphs are examined. The degree of a vertex is defined as the number of edges that connect to the vertex. For graphs that do not have multiple edges (connecting the same pair of vertices) nor loops, the degree of a vertex is equivalent to the cardinality of the set of neighbours of the vertex (Equation 2.2).

$$\deg(v) = |NBR(v)| \qquad \text{Equation 6.1}$$

6.2.2.1 Matched Molecular Pair identification

The method described in Chapter 3 was used to identify MMPs within each of the datasets. If not explicitly specified, the maximum change size was limited to 10 heavy atoms, and the ratio of the change to the whole molecular was limited to 0.3.

6.2.2.2 Pairs to graph

Each MMP can be seen as an edge (small molecular change - transformation as defined in Chapter 4) that connects two vertices (molecules), as shown in Figure 6.2. All compounds within a dataset are initialised as vertices with some identifier (typically SMILES or refcode family) as a label. Additional labels such as the number of polymorphs can also be added. List of all MMPs is used to construct the edges. The molecules in the MMP are joined via an edge with the transformation being stored as an edge label. Additional labels such as the property change for that transformation may also be used. The created edge is a directed edge (e_{12} in Figure 6.2) and the reverse edge should be added (e_{21}). However, the script used to create the MMP Database orders the transformations consistently. For this reason, the edge e_{21} is redundant as all hydroxyl to methyl MMPs would be ordered in the same way. Hence, the e_{21} edge is not added to the graph.

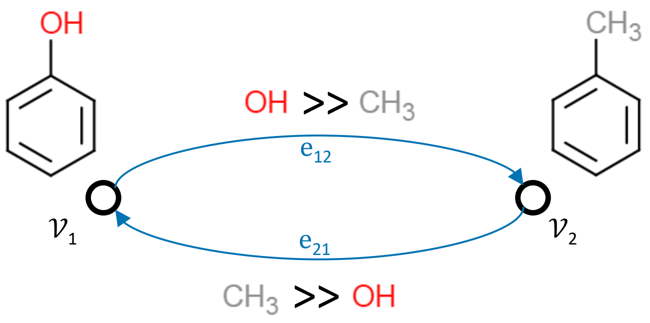


Figure 6.2: Visualisation of the Matched Molecular Graph contruction from a MMP.

Labels such as molecular structure (SMILES) can be assigned to each of the vertices. Labels such as the transformation (SMIRKS) can be assigned to edges.

6.2.2.3 Visualisation

Any graph operations can be performed in Python using NetworkX library [230]. However, it is often useful to visualise the graph and interact with it graphically. For this purpose, Gephi software package was used [231]. ForceAtlas2 algorithm was used for the vertex placement for visualisation [232]. The principle behind this algorithm is that vertices repel one another, but edges attract. The result is that clusters of interconnected vertices remain close, while vertices without many edges get repelled further away.

6.2.3 Software

Matched Molecular Pair were extracted from the database introduced in Chapter 3. Scripts were written to directly interact with the database. Graph visualisation was carried out using Gephi [231]. Python 2.7 was used for data manipulation. Figures that were not generated using Gephi were created using matplotlib [218] and seaborn [217]. The scripts used for the Gephi input generation is available in Appendix 1.

6.3 Results and Discussion

6.3.1 Monomorphic adjusted single component CSD dataset

A Matched Molecular Graph was constructed from the data used in the polymorph propensity study in Chapter 4. The dataset contained 6,633 entries with 2776 MMPs. The constructed graph is shown in Figure 6.3. Due to the nature of the ForceAtlas2 algorithm, molecules with no MMPs (no edges) are pushed to the outside, while molecules with MMPs tend to remain closer to the centre. This results in the graph to consist of three components. The outer ring of vertices (dark blue in the figure) consists of molecules with no MMPs. The middle ring (gold in the figure) contains

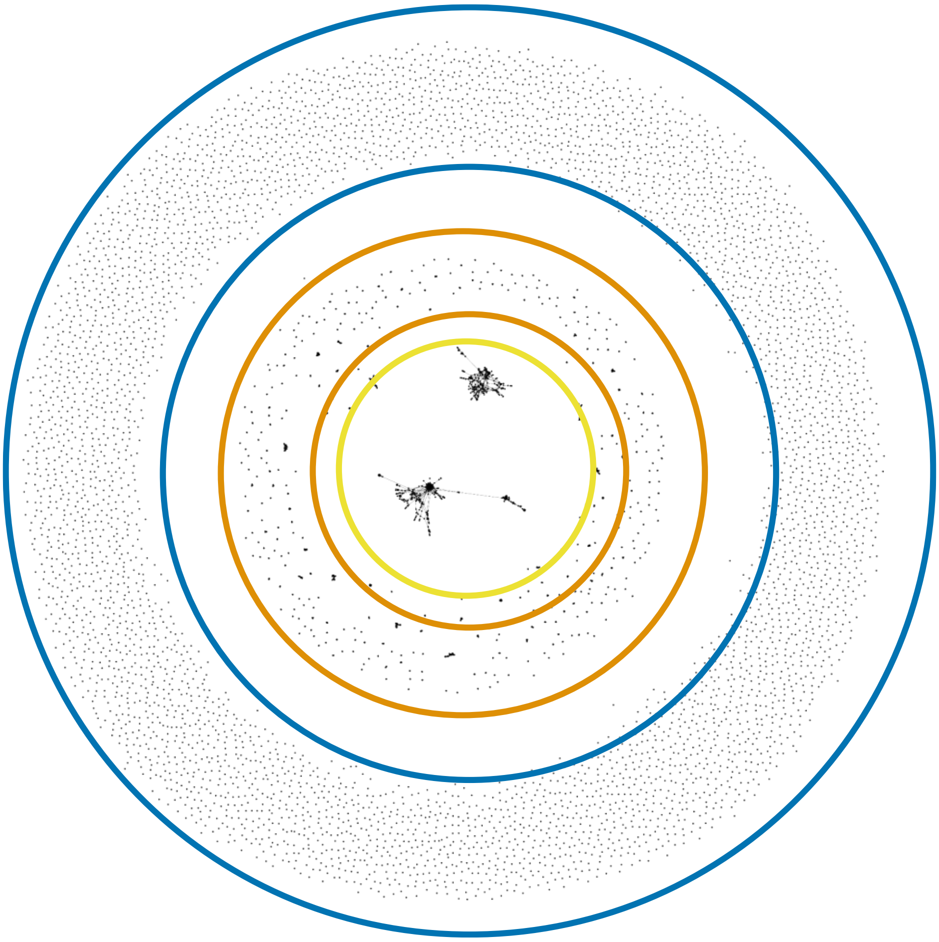


Figure 6.3: Matched Molecular Graph of monomorphic adjusted CSD single component dataset with max change size of 10 heavy atoms and max ratio of change of 0.3 for all MMPs.

The outer ring (dark blue) consists of molecules with no MMPs. The middle ring (gold) consists of molecules with few (typically one or two) MMPs. The inner circle (yellow) contains large clusters of molecules that share many MMPs.

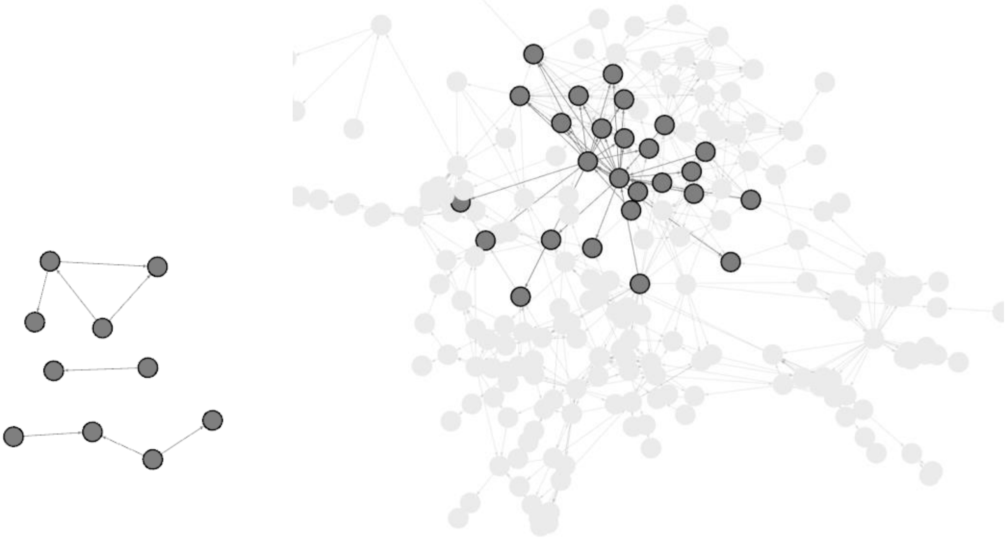


Figure 6.4: Example of clusters found in the Matched Molecular Graph.

On the left, typical clusters found in the mid-ring of the graph. On the right, a large cluster found in the central circle of the graph. Emphasised are all MMPs that contain the central molecule. Vertices were moved manually to avoid overlap for the sake of clarity.

molecules that have few MMPs. These typically consist of small clusters of molecules (left in Figure 6.4). In here, the term clusters are not used in accordance with the graph theory definition, but rather as a synonym for a disjointed subgraph. As can be seen from the graph, these clusters may be a pair of molecules (a single MMP) or a group of molecules that share multiple MMPs. Much larger clusters can be found within the inner circle of the graph in Figure 6.3. Molecules in these large clusters have up to 42 MMPs with other molecules from the cluster.

A closer examination of the relative size and properties of the three identified components in the MMG was performed to identify aspects that affect the performance of the MMPA procedure. The outer layer contains 74.9 % of the molecules (4,847) from the dataset, yet contain none of the MMPs (no edges). This means that any MMPA done on this dataset ignores almost three-quarters of the available data. This is particularly problematic for small datasets such as the one used in Chapter 4. The lack of MMPs for 74.9 % of the molecules suggests that these molecules are dissimilar from one another.

The middle ring contains 17.7 % of molecules (1,146) and 41.3 % of the MMPs. 482 molecules (7.4 %) are found in the inner circle. These molecules contribute to the rest of the MMPs (58.7 %). This means that over half of MMPA result comes from 7.4 %

of the molecules. In the case of polymorph propensity study where polymorphic structures are rare, this may result in polymorphic structures being underrepresented.

The analysis of the MMG suggests that MMPA carried out on this dataset may not be robust nor representative. To see which factors contribute to such MMG properties, two dataset characteristics were considered: dataset size and the origin within the pharmaceutical product development of the dataset. These are explored below.

6.3.2 Dataset size

The dataset size was expected to have an impact on the number of MMPs within a dataset. The Patent Dataset was used for the study of the effects of the dataset size as it is a large dataset (289,379 entries). MMGs for randomly selected subsets of 1,000, 2,000, 5,000, 10,000, 20,000, 40,000, 60,000 and 80,000 molecules were constructed. The number of structures with at least one MMP (in the middle ring or inner circle in Figure 6.3) were tracked for the increasing dataset size (Figure 6.5). For the MMG made from 1,000 molecules, the fraction is only 2.2 % meaning that only 22 molecules have MMP and therefore can be involved in MMPA. For the Patent dataset, this

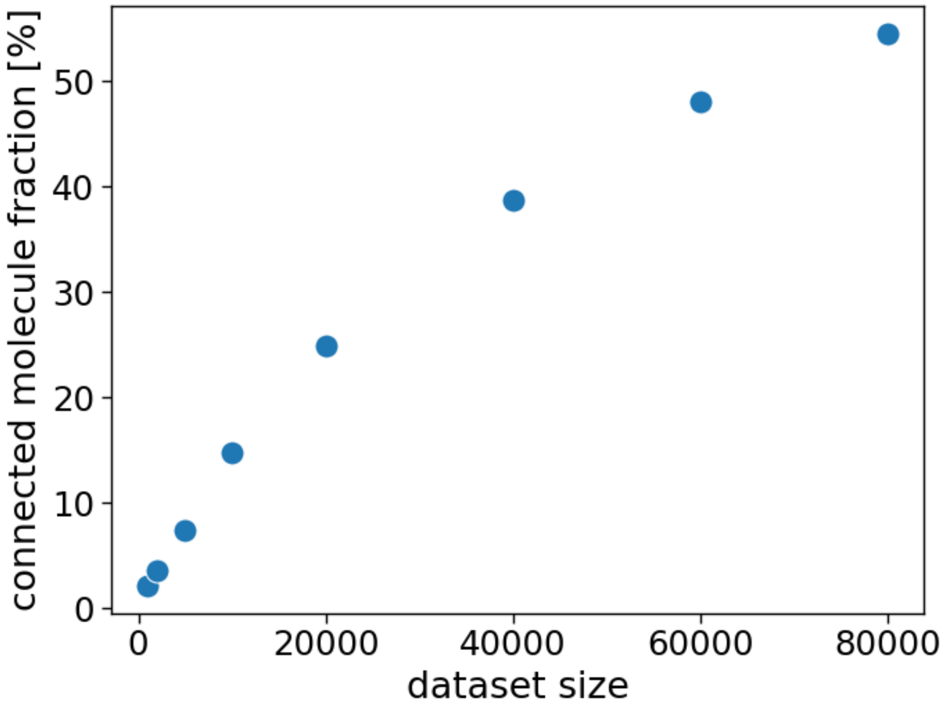


Figure 6.5: The fraction of molecules with at least one MMP as a function of the dataset size for the Patent dataset

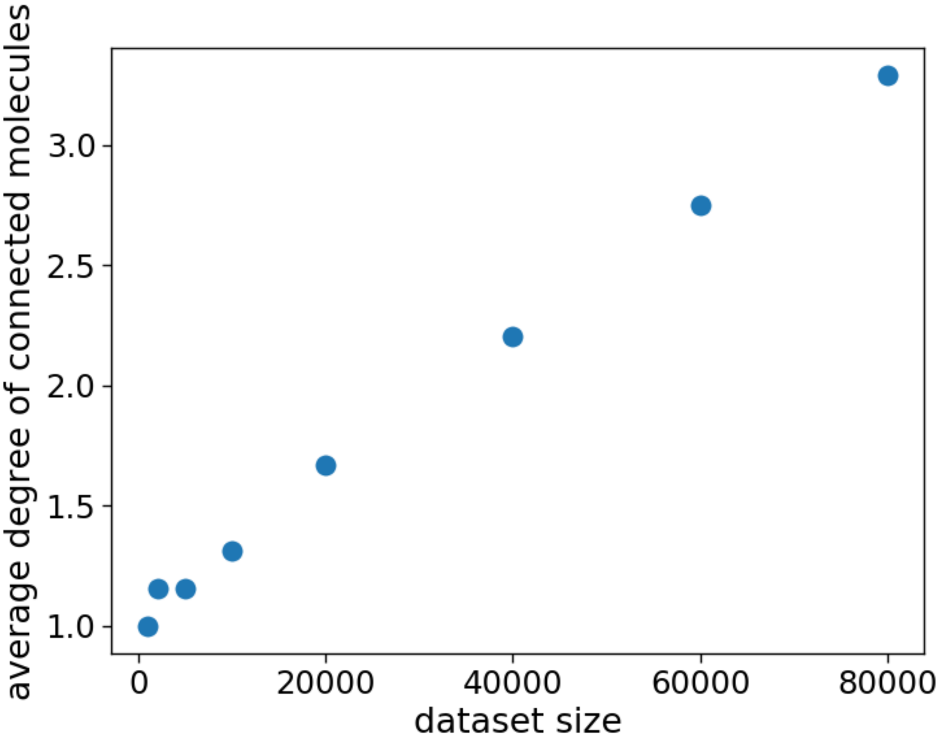


Figure 6.6: Average degree of molecules with at least one MMP as the dataset size increases for the Patent dataset.

fraction increases to 7.5 % for dataset size of 5,000. This is notably lower than the CSD dataset of the comparable size (6,633 molecules) in which 25.2 % of molecules had at least one MMP. The effects of the origin and purpose of the dataset are further discussed in the following section (6.3.3). The fraction of molecules with one or more MMPs continues to increase as the dataset size increases, making the MMPA more applicable.

Further to the increase in the fraction of molecules that can be used in MMPA (at least one MMP), the average number of MMPs (average degree of vertices) increases as well (Figure 6.6). The average degree was computed only for molecules that have at least one MMP in order to distinguish this measure from the fraction of molecules with at least one MMP discussed above. For the 1,000 dataset, the average degree is 1, indicating that there are only pairs and no clusters forming. This changes as the dataset size increases, with the average degree increasing to 3.3 for the 80,000 dataset.

Based on the analysis, the dataset size plays an important role two-fold. Firstly, the fraction of molecules that have at least one MMP increases with the dataset size. Secondly, the average number of MMPs for molecules that have at least one increases with size as well. As a result, analysis based on small datasets, such as the one done in Chapter 4, may not produce meaningful results due to the small number of MMPs and molecules that are included in the MMPA.

6.3.3 Datasets across the Pharmaceutical Product Development

The Patent dataset, which had a smaller fraction of molecules with at least one MMP compared to the CSD dataset, represents a more diverse range of molecules. The effects of the data source on the MMG were investigated further. The GSK TCAKS dataset was taken as a representative of the Discovery dataset as it comes from a single company against a specific target. In reality, pharmaceutical companies hold significantly larger datasets within Discovery stage compared to Development datasets, based on the attrition rate through the pharmaceutical development process. However, to study the effects of the data source, a subset of the same size as the

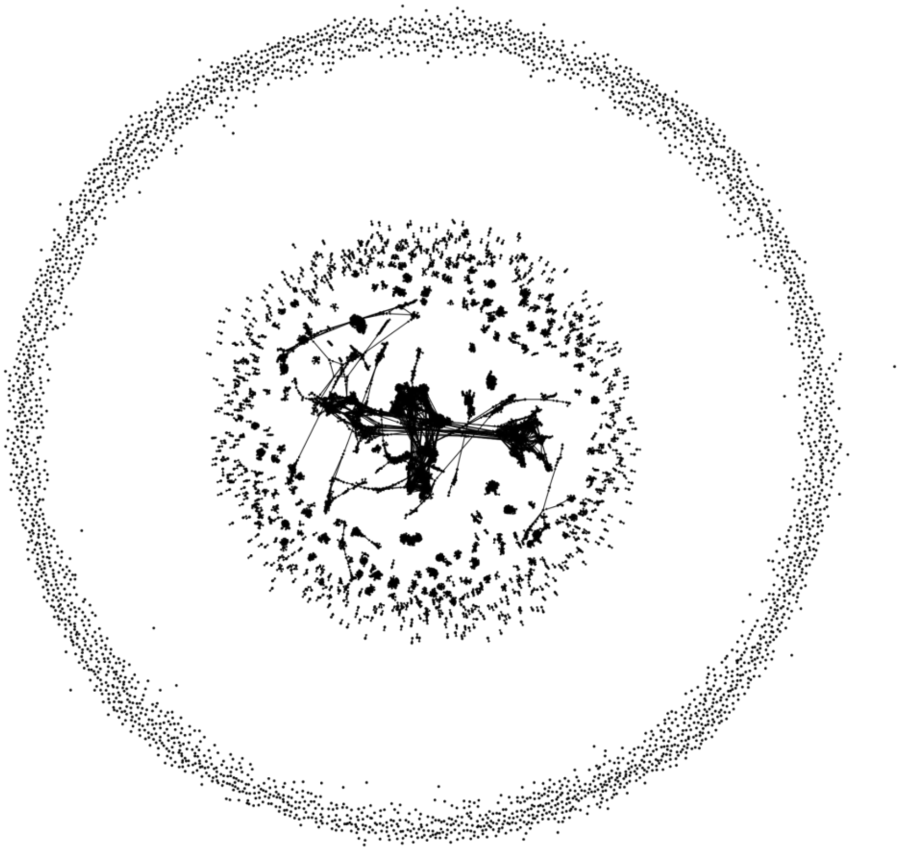


Figure 6.7: Matched Molecular Graph of GSK TCAKS dataset (Discovery dataset)

Development dataset was randomly selected. The CSD dataset was used as a surrogate for a Development dataset.

The MMG of the Discovery dataset is shown in Figure 6.7. The difference compared to the MMG of the Development dataset (Figure 6.3) is visually apparent. 58.2 % of molecules in the Discovery dataset have at least one MMP compared to 25.1 % for the Development dataset (Figure 6.8). This means that for the same dataset size, more than twice as many molecules can be involved in MMPA of Discovery datasets compared to a Development counterpart. Comparison of the MMGs can also reveal that the Discovery dataset has larger clusters. The average degree of molecules with at least one MMP is 5.34 for Discovery and 3.33 for Development. The total number of MMPs is 10,321 and 2,776, respectively, indicating the difference in suitability of MMPA for the two datasets.

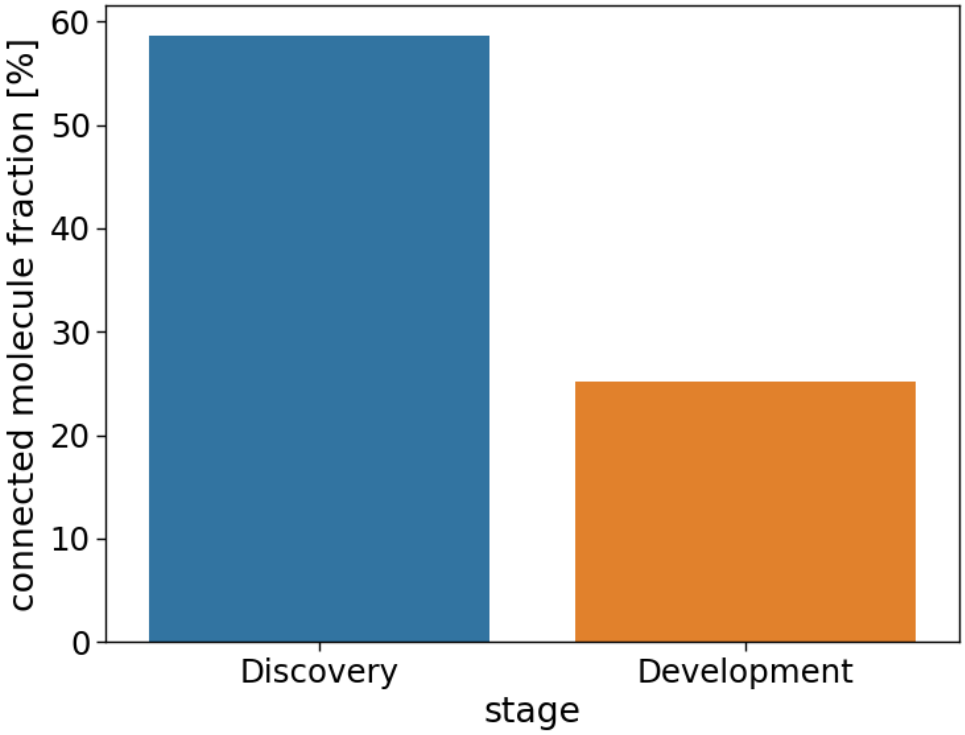


Figure 6.8: Comparison of the fraction of molecules with at least one MMP for datasets taken from different stages of the Pharmaceutical process development.

CSD monomorphic adjusted single component dataset used as an example of Development dataset. GSK TCAKS dataset is used as a representative example of a Discovery dataset.

6.4 Conclusion

The concept of Matched Molecular Graphs (MMGs) was developed to address the issue of the small number of MMPs found within the CSD dataset studied in Chapter 4. The analysis of the MMG constructed from the dataset revealed that 74.9 % of the structures do not contribute to MMPA as they do not have a single MMP. The majority of the MMPs (58.7 %) comes from a small fraction (7.4 %) of molecules that form dense clusters. The effects of dataset size and source (Discovery or Development) on the key MMG parameters were investigated. Unsurprisingly, the larger the dataset, the larger the fraction of molecules with at least one MMP and the more MMPs overall. The effect is particularly crucial for smaller dataset sizes (<10,000 molecules). The change in MMG parameters decreases as the dataset size increases. Datasets of the same size, taken from Discovery and Development, also show a difference in MMG parameters. The Discovery dataset contained approximately four times more MMPs (10,321 against 2,776). The number of molecules with at least one MMP was also considerably higher for the Discovery dataset (58.2 %) compared to the Development dataset (25.1 %). The analysis was performed only on a single dataset from each of the stages, so further analysis of more datasets is necessary to establish whether this trend is representative.

However, based on the analysis carried out here, it is clear that performing MMP-based analysis is likely to exclude some data. This is particularly significant when working with Development datasets which tend to be smaller and more diverse. Given the focus of the thesis on Development stage related property prediction, the emphasis shifted to QSPR approach. In the next chapter, work on a QSPR model for solid state-specific prediction of melting point is discussed with a particular focus on the ability to capture crystal information.

Chapter 7

Melting Point Prediction Using Message Passing Neural Networks Based on Molecular and Crystal Structures

7.1 Introduction

Thus far in the thesis, work on identifying polymorphs and predicting the propensity to form polymorphs was presented. This chapter focuses on the effects of polymorphism on the properties of the solid state. An attempt is made to develop ways to capture the solid state information to allow accurate solid state property prediction. Melting point is used as a case study in this chapter.

Melting point is the temperature at which a solid transitions into a liquid. The process consists of breaking of the intermolecular interactions that hold the molecules within the crystal lattice. The temperature at which this occurs is the ratio of enthalpy and entropy of melting [152]. The property, along with logP, can be used to estimate the solubility of a compound via the General Solubility Equation (GSE) [81]. The melting point can also be used as a descriptor of the strength of intermolecular interactions within the crystal [35,71]. This can be used to identify “brick dust” compounds, where the solubility is limited by the solid state interactions [35,71].

Melting point has been of interests to scientists for a long time. Earliest work on melting point prediction can be traced back to the 19th century [233]. Most of the work performed in the area focused on a narrow applicability domain such as alkenes or chlorobenzenes [234,235]. As the availability of data increased, predictive models that are applicable to a wider range of molecules were developed [172]. Less curated, larger datasets have also been shown to produce models with large applicability domain [83,84]. Reviews of the melting point prediction can be found elsewhere [45,83,236].

A number of common challenges were identified in the previous works on the melting point prediction. In particular, it has been suggested that the molecular descriptors are unable to capture the long range intermolecular interactions that need to be broken for the crystal to melt [94]. However, another significant contribution which limits model performance is the experimental error, which may be interpreted as the inconsistency between measurements reported in different studies. Analyses have suggested that experimental errors vary with melting point and may be of the order of 32-35 ^{o}C [83]. However, these estimates are based upon analysis of the variability of measurements for the same molecule and may be affected by polymorphism or impurity.

However, the value of incorporating solid state information can only be properly assessed by focusing on the prediction of melting points which directly correspond to that solid state information, i.e. polymorph specific melting point data are required. Hence, to assess the importance of the solid state contributions, a dataset for crystal structure-specific melting point was used in this study. The Cambridge Structural Database (CSD) is a curated repository of small organic and metal-organic crystal structures [74]. Some of the entries contain melting points measured for the specific crystal structure. These entries formed the bases on the dataset used in this study. Previous work done on incorporating solid state information into predictive models for temperature-dependent solubility did not yield expected improvements [70]. The inadequacy of the solid state descriptor was cited as one potential reason for the lack of significant improvement in performance, along with the lack of polymorph specific solubility data.

In this work, an attempt was made to capture the crystal information by a combination of different kind of descriptors, including a novel graph embedding representation of intermolecular interactions. The approach was presented at Computational Molecular Science conference in March of 2019 [237]. Since then, a similar approach to capturing crystal information [238] and ligand-protein intermolecular interaction [239] has been published.

Graph embedding techniques such as Message Passing Neural Networks (MPNN) can generate a fixed-length descriptor of a graph [104]. In MPNN, for each node (*v*), a message ($m_v^{(t+1)}$) is passed from each of the neighbouring nodes (NBR(*v*)) based on the edges type (l_e) where each edge is defined by the two nodes (*e* = *vw*). In the implementation used, this is achieved by the following function [105]

$$m_v^{(t+1)} = \sum_{w \in NBR(v)} W_{l_{vw}} h_w^{(t)} \qquad \text{Equation 7.1}$$

$W_{l_{vw}}$ is the weight matrix for the specific edge type (interaction between atoms) and $h_w^{(t)}$ is the state of the node (atom) at iteration *t*. The state of each node is updated at the end of the message passing stage using a Gated Recurrent Unit (GRU) [240] or a Recurrent Unit. (RU) [105]

$$h_v^{(t+1)} = f_{update}(h_v^{(t)}, m_v^{(t+1)}) \qquad \text{Equation 7.2}$$

The message passing is repeated *T* times after which all states are pooled to generate the fixed-length representation of the graph using gated regression layer [105].

$$p_{structure} = \sum_{v \in V} \sigma\left(i\left(h_v^{(T)}, h_v\right)\right) \odot \left(j\left(h_v^{(T)}\right)\right) \qquad \text{Equation 7.3}$$

In the implementation used in this work, *i* and *j* are Multi-Layer Perceptrons (MLP) with no hidden layers and ReLu (rectified linear unit) activation function. σ is the sigmoid activation function and $\odot$ indicates a Hadamard product (element-wise multiplication).

A detailed description of the principles of MPNN can be found elsewhere [241]. MPNN have been used to model many types of graph information, ranging from knowledge graphs to molecules [43,57,104,106,241,242]. Prediction of thermoelectric properties of materials using crystal graphs have also been studied [242].

In this work, the suitability of MPNN as a method of embedding crystal information for melting point prediction is assessed. This is achieved by comparing models that only had molecular information to ones that had access to molecular and crystal information, as well as through Matched Molecular Pair Analysis (MMPA as discussed in Chapter 3) and comparison of polymorph predictions.

7.2 Methods and data

7.2.1 Datasets

The CSD Melting point dataset (CSD MP set) was used in the study to develop models to predict the melting point. Approximately 17 % of single component structures in the CSD have melting point data reported along with the crystal structure. The melting point data was converted to consistent units and entries where the reported melting point range for a specific crystal structure was more than 5 °C were ignored. This was performed to minimise the effects of experimental errors on the dataset. Measurements of more than 5 °C range for a specific crystal structure were considered unreliable. Entries where instead of a melting point, a temperature of degradation or sublimation was reported, were ignored as well. This resulted in a dataset of 61,250 crystal structure specific melting points. The CSD MP set was split into training (75 %, 45,938), validation (15 %, 9,187), and tests (10 %, 6,125).

7.2.2 Model architecture

The model developed here consists of a graph model and prediction layers (Figure 7.1). Message Passing Neural Network is used as the graph model to generate a fixed-length representation of the graph inputs as per Equation 7.1 – Equation 7.3. The details of the graph model used and the respective hyperparameters are presented below (7.2.2.1). The graph representation along with additional descriptors are fed into the prediction layers which make the melting point prediction (7.2.2.2).

7.2.2.1 Graph model

The Graph Model (GM) uses a varied size graph as an input and outputs a fixed-length representation of it. The graphs were constructed using CSD Python API. The graphs contain nodes information (atom information) and edge information (intra- and inter-molecular interactions). Atom type (element identity) information was one hotkey encoded and padded with zeroes to the predefined size (node size). This was used as the initial node vector values. Edge types were categorised into: single, double, triple, or aromatic covalent bonds. Additionally, intramolecular Van der Waals (VdW), intermolecular VdW, and hydrogen bonds were used. Hydrogen bonds and Van der Waals interactions were identified using the distance and line of sight as defined by the default settings of the Python API.

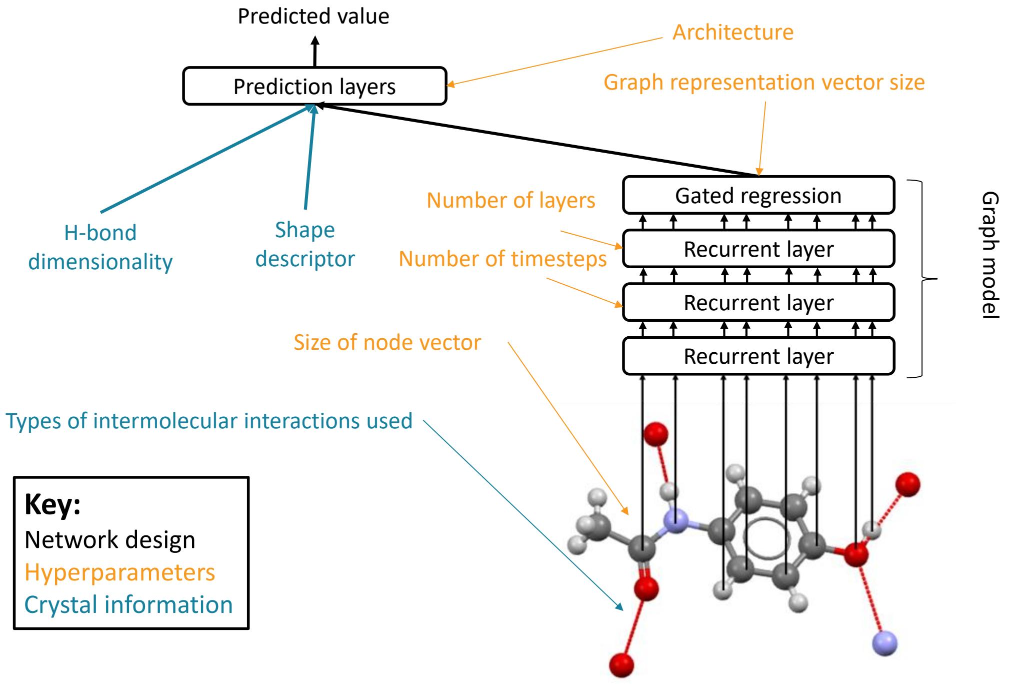


Figure 7.1: Overview of the model architecture.

A GM cell was assigned to each node (atom). These cells were either recurrent units (RU) or gated recurrent unit (GRU) [240]. A message is passed between each connected node based on the edge type (i.e. different weight matrix for each of the edge types) as per Equation 7.1 and Equation 7.2. The number of times the message is passed is divided into two hyperparameters: GM layers and GM timesteps. GM layers indicate how many different weight matrices are used for each of the edges. GM timesteps indicates how many times the message is passed using the same weights. For example, two GM layers with timesteps of two and one respectively indicate the passing of the messages using W_1 twice followed by message passing using W_2 once (where W_i is the weight matrix for a particular edge type for layer i). This is performed to allow the model to treat neighbouring atoms (e.g. within the same functional group) differently to ones further away. The total number of message passing steps (t) indicates how many neighbours is each of the nodes 'aware' of. After the message passing step is complete, all the messages are aggregated to produce a fixed-length representation of the graph (graph vector). All the nodes are summed and passed through a gated regression Equation 7.3. Similarly to GRU, the gated regression uses an update gate to select which information is passed. The generated fixed-length graph representation is then fed to the prediction layers. In the original implementation of the algorithm [57], the message passing step was repeated until all nodes vectors converged. However, later work showed that Gated Graph Neural Networks could use the gated pooling step to generate the fixed-length graph feature [105].

7.2.2.2 Prediction layers

The prediction layers (PL) are a multilayer perceptron (MLP). The graph representation along with the additional descriptors are the inputs to PL. The number of neurons per each of the two layer are two hyperparameters. ReLu was used as the activation function for all neurons except the outer layer, where a linear activation function was used to generate a single value prediction. The graph representations do not store any geometrical information; the additional descriptors focus on capturing this. The two additional descriptors attempt to capture the molecular and crystal interaction geometrical information respectively.

Shape change. The Root Mean Squared Deviation (RMSD) between the molecule in the crystal and the molecule in the gas phase is used as the molecular shape change descriptor. The descriptor intends to represent the energy required to distort the shape from its optimal conformation to the conformation found in the crystal.

A molecule is taken from the unit cell of the crystal, and 10 random conformers are generated. Each of these is then optimised in the gas phase using the Universal Forcefield [243] as implemented in RDkit [213]. The lowest energy one is taken to represent the global minimum conformation. RMSD is computed between the optimised molecule and each molecule in the unit cell of the crystal, following the maximum alignment of the structure. The average RMSD value is used as the descriptor.

Hydrogen bond dimensionality. Hydrogen bonds are captured by the graph model; however, the dimensionality of it may be lost. Hydrogen bond dimensionality was calculated using the method presented in this paper [244]. The possible outputs are: 0-D (point), 1-D (chain), 2-D (plane), or 3-D. One hotkey encoding was used to express this where a vector of zeroes was used for structures with no hydrogen bonds.

7.2.3 Model construction

Two types of models were developed to investigate the effect of incorporation of crystal information. Molecule model is constructed only from molecular information. This includes the atom type and the intramolecular bonds between them. Crystal model also has access to the additional crystal information. Usage of each type of crystal information is a hyperparameter where the model can learn to use or ignore it. The comparison of the information available and the information used is presented in Table 7.1.

Each model was trained using Adam optimiser [137]. The training was stopped after 300 epochs or after 25 consecutive epochs with no improvement in MSE on the validation set. Weight initiation was performed using Glorot initiation [133] (see 2.4.6.2). Hyperparameter optimisation was performed using Tree-structured Parzen Estimator (TPE) algorithm [147] as introduced in 2.4.7. Hyperopt [146] – Python implementation of the algorithm was used. Up to 1,000 steps of optimisation, with early stopping if no performance improvement was observed for 10 consecutive iterations, for each of the types of model were performed. The top 10 models of each type were analysed to determine the optimum combination of hyperparameters.

Table 7.1: Information used and made available to Molecule and Crystal models.

Y – available to the model, N – not available to the model, O – optionally available to the model.

Input to	Information	Molecule	Crystal
Graph model	Atom type	Y	Y
	Covalent bonds	Y	Y
	Hydrogen bonds	N	O
	Intra-molecular VdWs	N	O
	Inter-molecular VdWs	N	O
Prediction layers	Shape change	N	O
	Hydrogen bond dimensionality	N	O

Optional information for the Crystal model (Table 7.1) were used as optional hyperparameters (Use / not use). The graph model hyperparameters were: node size, node cell type (GRU or RU), graph representation size, number of timesteps and the number of layers. The prediction layers (MLP) hyperparameters were the two hidden layer sizes.

7.2.4 Performance analysis

The performance of the models was evaluated using Mean Absolute Error (MAE), and R^2 (mean coefficient of determination) value. Root Mean Squared Error (RMSE) was also reported. Further to this, Matched Molecular Pairs (MMPs) and polymorph pair comparison was used to de-convolute the relative importance of molecular and crystal information.

7.2.4.1 Matched molecular Pairs

Matched Molecular Pair database as introduced in Chapter 3 was used in this study. The MMPs were used to compare the actual change of melting point to the predicted change to see how well the model is able to predict small molecular changes. It was also used to estimate the typical effect a small molecular change has on melting point as a point of comparison against melting point differences of polymorphs.

7.2.4.2 Polymorph Pairs

Polymorph pairs were defined as pairs of crystal structures with the same molecular composition (based upon SMILES comparison) but different melting points. Structures with the same molecular composition and same melting point were considered to be redeterminations of the same crystal structure and were not used for this analysis. The change in actual melting point across polymorph pairs was used to estimate the average effect a polymorphic change has on the melting point. The change was compared to the predicted difference to approximate how well the model is able to predict the effects of solid state changes.

7.2.5 Software

The work in this paper was done using Python 2.7 and Python 3.6 environments due to the requirements of different libraries. The preparation of the CSD MP dataset was doing using the Python 2.7 environment. All interaction with the CSD was done using the CSD Python API (version 1.5.2) distributed by the CCDC with the database [215]. RDkit was used for molecule optimisation [202]. Script by Steven Kearnes from DeepChem library was used for the molecule optimisation workflow [245]. The Python 3.6 environment was used for the model development. Tensorflow [246] was used to construct the neural networks and was run on University of Leeds ARC facilities. Hyperopt [146] was used for hyperparameter optimisation. Pandas [216], scipy [247], matplotlib [218], and seaborn [217] were used for data processing and visualisation. The scripts used for the network construction was based on work available from GitHub [248]. The modified scripts along with ones developed specifically for this work are available in Appendix 4.

7.3 Results and Discussion

7.3.1 Model performance and architecture

The hyperparameter optimisation was continued for up to 1,000 iterations. The Crystal and Molecule models with the highest R^2 value (equivalent to MSE as calculated on the same dataset) are reported in Table 7.2. The optimisation converged on the best combination of hyperparameters; the top 10 best performing models have a similar set of hyperparameters (Table 7.2). The best performing Crystal model achieved R^2 value of 0.649 on the validation set and 0.550 on the test set (the optimisation curve shown

Table 7.2: The best Molecule and Crystal models' hyperparameters along with the average of the top 10 models for each category. Same treatment was applied to R^2. For categorical hyperparameters the most common value and the corresponding fraction is reported.

Hyperparameter	Molecule		Crystal	
	Best	Top 10	Best	Top 10
Graph vector	300	400	500	690
Node vector	110	117	90	94
PL layer 1	380	412	300	336
PL layer 2	160	200	230	252
GM cell type	RU	RU (1.00)	RU	RU (1.00)
GM timestep	2	2.00	2	2.00
GM layers	1	1.00	1	1.00
Use H-bond	-	-	False	False (1.00)
Use intra-VdW	-	-	True	True (1.00)
Use inter-VdW	-	-	False	False (1.00)
Use shape change	-	-	True	True (1.00)
Use H-bond dim	-	-	True	True (1.00)
R^2	0.628	0.621	0.649	0.631

in Figure 7.2). The corresponding best Molecule model achieved R^2 of 0.628 on the validation set and 0.500 on the test set.

Methods such as dropout [114] were utilised to reduce the risk of overfitting. However, the decrease in the R^2 value between the validation set and the test set (from 0.649 to 0.550 for the crystal model) may be indicative of overfitting. The Molecule model also underperformed on the test set (0.628 on the validation set and 0.500 on the test set). However, the mean absolute error remained relatively unchanged for the Molecule model (31.8 ^{o}C for both) and reduced from 30.8 ^{o}C to 29.5 ^{o}C on the test set for the Crystal model. The subsequent analysis needs to be considered with the caveat that some overfitting occurred. It was still considered valuable to compare the performances of the Molecule and Crystal models, as well as investigate how well the crystal model performed on pairs of polymorphs.

The major difference between the top 10 Crystal models comes from the size of the layers. In particular, the graph vector size ranges from 500 to 1,200. This suggests that the size of 500 is sufficient to capture graph information and any size above that does

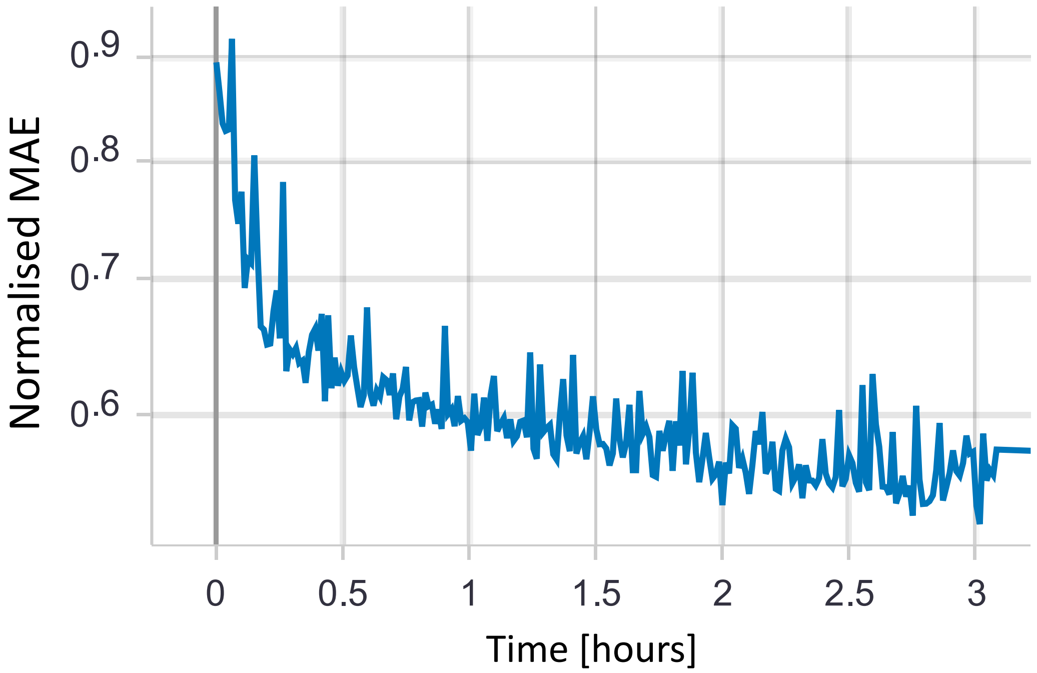


Figure 7.2: Optimisation curve for the training of the Crystal model.

The graph was extracted from Tensorboard as part of the used Tensorflow library. Each datapoint represents an iteration of training. The MAE is expressed in terms of the normalised data (standard deviation of 60.2).

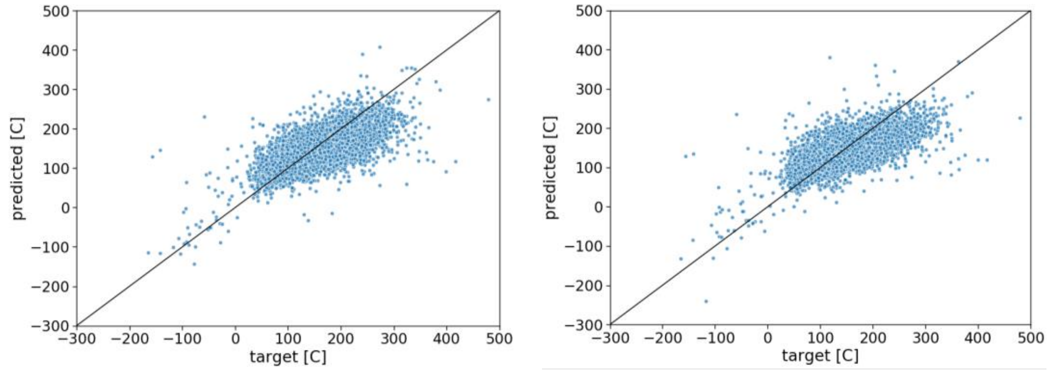


Figure 7.3: Target MP and predicted MP by Molecule (left) and Crystal models (right) on the validation set.

not contribute to the improvement of the model. This is reinforced by the fact that the best performing Crystal model had the graph size vector of 500.

The graph vector size was bigger for Crystal models compared to Molecule models. The Crystal models are able to store more information concerning the graph structure, since crystal graphs are more complex than molecular graphs. However, the optimal size of the node vector is smaller for the Crystal models compared to the Molecule models. Intuitively, this can be explained by the need of the Molecule models to implicitly capture information about the possible inter-molecular interaction that the Crystal models can capture explicitly. Therefore, the Molecule model needs a bigger vector to store the information. The optimal graph model set up, in terms of the GM layers and GM timesteps, was similar for the two types of models. Using only a single set of weights (one GM layer) seems to be sufficient. This is potentially due to the increase in number of trainable parameters associated with multiple GM layers which the model might not be able to fit adequately. Two GM timesteps, two degrees of separation, are sufficient for each node to learn about its neighbours. This appears consistent with the fact that a typical functional group can be identified by atom connection within two degrees of separation. For larger groups such as aromatic rings, a unique edge type is used, reducing the number of timesteps required for the model to learn the presence of this kind of a functional group.

Interestingly, hydrogen bonds are not as useful as part of the graph model component of the Crystal model. This is potentially because the possibility of forming hydrogen bonds is implicitly captured by the graph based on the functional groups present. Furthermore, the hydrogen bond dimensionality descriptor captures the complexity of the hydrogen bonding formed within the crystal structure, making the explicit hydrogen bonding encoding superfluous.

Based on the hyperparameter optimisation, the intermolecular Van der Waals interactions were not selected as useful. This is potentially due to the fact that VdW interactions are implicit in the molecular shape in the crystal structure, which is partially taken into account by both the information regarding molecular functional groups and intramolecular VdW interactions. This is encoded in the graph model via the intramolecular VdW, and the molecular shape descriptor representing how the molecular shape is distorted in the crystal from the gas phase preferred structure. This may also be the reason for the inclusion of intramolecular Van der Waals interactions. This may be complementing the shape descriptor in capturing the relative positions of the atoms within the crystal structure, since the VdW interactions were obtained purely based on distance.

To test these hypotheses regarding the reasons why the hydrogen bonding and VdW intermolecular interactions were not selected for the graph model component of the Crystal model, a new model which had access to crystal edges (H-bonds and VdWs) but not the H-bond dimensionality and the molecule shape descriptor, was trained and hyperparameter optimised. This model achieved R^2 of 0.630, worse than the best performing Crystal model, but comparable to the top 10 models and surpassing all the Molecule models. This suggests that in the absence of H-bond dimensionality, the H-bond and VdWs edges contribute to the performance of the model. The molecular shape change descriptor is a useful descriptor as the graphs do not store any geometrical information.

7.3.2 Does crystal information help?

Molecule and Crystal models were compared to see the relative importance of the additional crystal information. The best Crystal model achieved the R^2 of 0.649 and 0.550 on the validation and test sets, while the Molecule model achieved 0.628 and 0.500 (Figure 7.3 and Table 7.2). The top 20 best performing Crystal models are statistically different compared to the top 20 Molecule models based on the Mann-Whitney U test (p value = 2.937×10^{-6}). The addition of crystal information does improve the performance of the model. However, the improvement is relatively small, considering the amount of effort that is required to obtain a crystal structure. Three possible reasons for the limited improvement were investigated: (1) under-representation of polymorphic structures in the dataset, (2) inability to capture solid state-specific information, and (3) the lack of importance of solid state information.

7.3.2.1 Underrepresentation

The difference between two crystal structures can be separated into a molecular change and crystal change (Figure 7.4). Crystal change corresponds to the change in molecular shape and arrangement as is seen when comparing two polymorphs. A molecular change alone cannot be easily observed as a pair of different molecules also pack differently. Therefore, a molecular change is also associated with a crystal change.

The CSD MP dataset contains 672 molecules that are polymorphic (1.1 %). For the majority of training, different target values were presented along with different molecular and crystal changes. The model likely did not effectively learn the effects of crystal changes as these were often linked with molecular changes. The small number of occurrences of entries with the same molecular structure but different

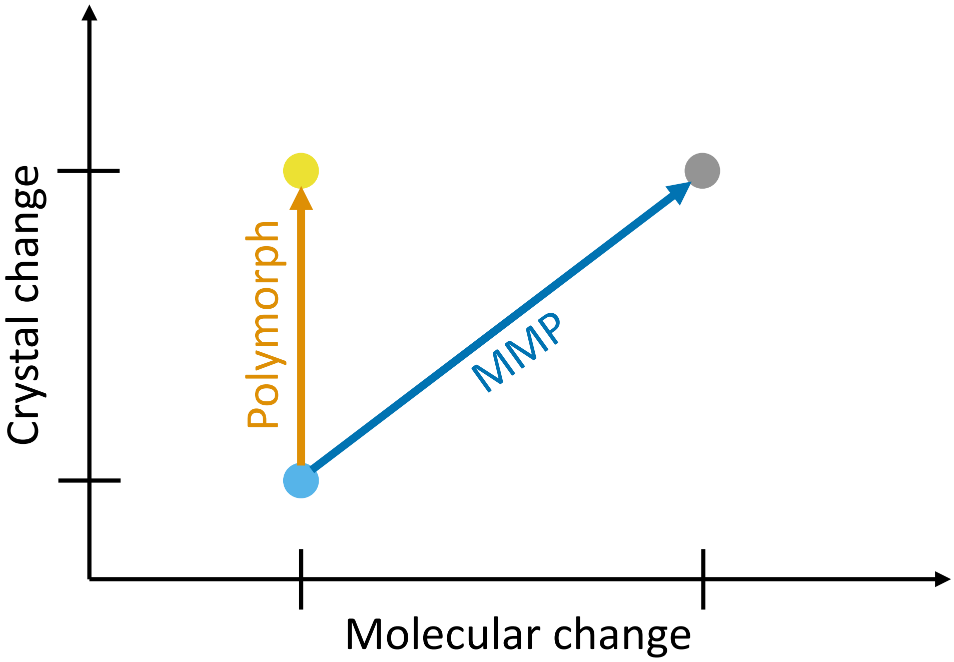


Figure 7.4: Illustration of molecular and crystal changes along with how these can be studied using polymorphs and Matched Molecular Pairs (MMPs)

Polymorphs is a crystal change with no associated molecular change. A small change in the molecular structure (MMP) results in a molecular and crystal change as the crystal packing is affected by the molecular structure.

crystal structure (polymorphs) may have led the model to incorporate crystal change as part of the molecular change.

7.3.2.2 Capturing crystal information

Despite being underrepresented, analysis on polymorph specific MP prediction can be performed. 21 pairs of polymorphs were identified within the validation set. Firstly, as a surrogate for relative polymorph stability, the order of melting points between polymorphs was investigated. 13 pairs of the 21 polymorphic pairs were predicted in the right order (62 %). A score of -1 was given where the order of melting points was incorrect and +1 if the order was correctly predicted. Random guessing is expected to result in a symmetric distribution around 0 (equal number of correct and incorrect guesses). Wilcoxon test was used to see if the model is statistically different from the random guessing. The p value calculated was 0.275, thus the null hypothesis that the distribution is symmetric around 0 (i.e. equivalent of random guessing) cannot be rejected. This is consistent with the hypothesis that the model failed to adequately capture the crystal change specific contributions to the melting point. However, it is important to note that the sample size is very small (21 pairs).

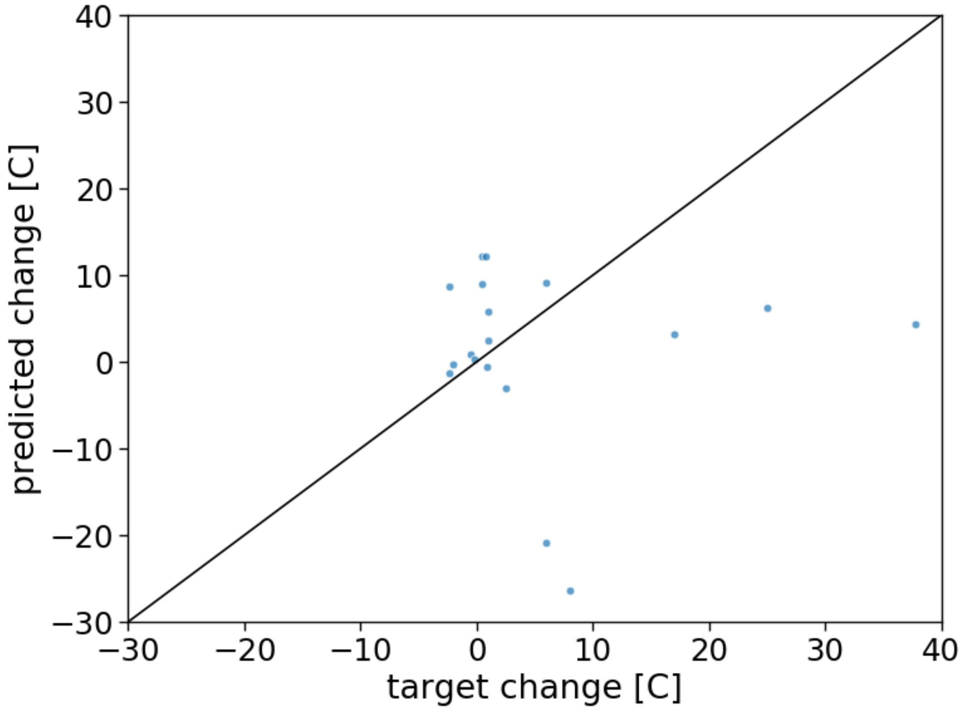
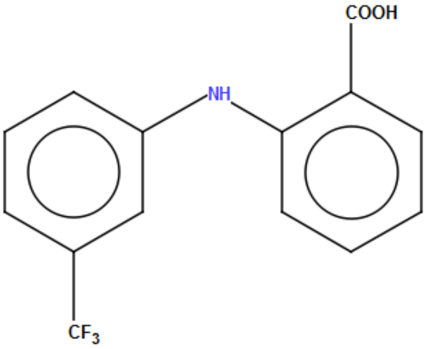
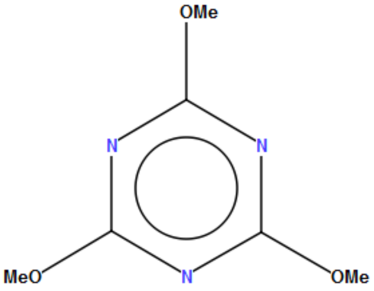
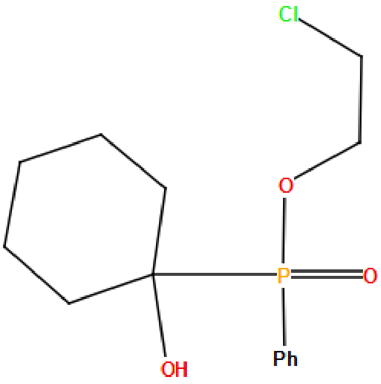
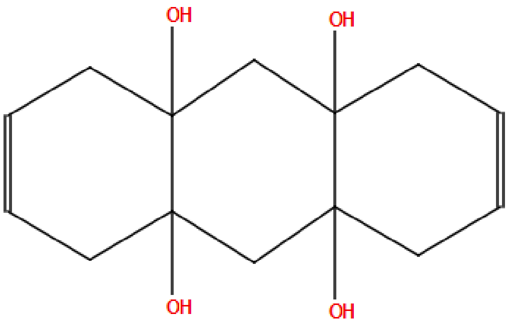


Figure 7.5: Actual and predicted change for polymorph pairs.

Table 7.3: List of molecules for which the Crystal model was not able to accurate predict the difference between polymorphs. Cases where the predicted value of a specific structure is incorrect by more than the MAE (30.8 °C) are highlighted.

Refcode	Target [C]	Predicted [C]	Molecule
FPAMCA12	120.7	151.7	
FPAMCA14	124.2	**70.0**	
JATFUF02	122.0	102.2	
JATFUF03	135.9	**208**	
CEPXHP	119.6	126.1	
CEPXHP01	127.6	99.8	
KARCOW	254.6	**142.7**	
KARCOW01	292.3	**147.1**	

investigated further by seeing how well the model is able to predict the difference between polymorphs. The comparison of the predicted and actual differences between polymorphs is presented in Figure 7.5. The magnitude of the actual change ranges from 0.5 ^{o}C to 37.8 ^{o}C while the predicted change is typical below 13.0 ^{o}C (two cases of the change predicted to be 21 and 26 ^{o}C shown in Figure 7.5 and two cases of over 80 ^{o}C not shown in the figure). Polymorphs where the discrepancy between actual difference and predicted difference was the largest were examined closer. A select few pairs of molecules are shown in Table 7.3.

A number of structures where the large difference in the actual melting points was due to errors in the data were left out of the analysis. XEHGOH and XEHGOH01 have target values of 138.9 and 163.9 ^{o}C respectively, but based on the comparison of packing [211], the two structures appear to be the same polymorph. Hence, these should have the same melting point. The model made a prediction that is 6.2 ^{o}C apart (141.2 and 147.4 ^{o}C respectively) which is consistent with the similarity of packing between the two crystal structures. In case of FPAMCA (12/14) and JATFUF(02/03), as shown in Table 7.3, one of the structures was predicted within the mean absolute error (30.8 ^{o}C) while the other was outside of that range. This suggests that in these cases, the solid state effect was not adequately captured. The pair of molecules of the refcode family KARCOW were both predicted inaccurately by over 100 ^{o}C. The order of polymorphs stability (melting point) was predicted correctly, but this is likely a coincidence considering the overall poor prediction for the structures.

Several shortcomings of the model setup may be the cause of the inability to accurately predict the solid state specific melting point. The intermolecular interactions are based purely on the geometric relationship between atoms rather than force-based consideration. This may result in the inaccurate assignment of the intermolecular interactions, especially in cases such as the π- π interaction between aromatic rings. In cases where hydrogen bonds do not form, the π- π stacking interaction may play an important role in contributing to the lattice energy such as in case of p-aminobenzoic acid [249].

7.3.2.3 Relative importance of solid state changes

The importance of the additional crystal information is useful only if crystal change (polymorphism) plays an important role in melting point. To assess this, the effects of polymorphs were compared to the transformations corresponding to matched molecular pairs (MMPs) identified within the CSD dataset. Representative transformations were selected and their impact on the melting point was compared.

The most common transformation was the substitution of hydrogen by a methyl group (1,302 occurrences). It was expected to be relatively insignificant due to the small size of the methyl group and the lack of the change of hydrogen bond acceptor/donor count. The transformation was expected to have limited impact on molecular packing and VdW interactions. As an example of a transformation that may affect the melting point more, a substitution of a hydrogen with a carboxyl group was selected as it changes the number of potential hydrogen bonds. Considering transformations with at least 10 MMPs within the used dataset, this transformation had the third-highest mean effect on the melting point. The top two are hydrogen to carbamoyl and methyl to carboxyl. However, hydrogen to carboxyl has much higher MMP count within the dataset (74 against 15 and 19 respectively) so it was selected, as its effects on melting point could be assessed more robustly.

The addition of the hydrogen bonding carboxyl group has the largest effect on the melting point, with the mean and median change of 95 ^{o}C and 107 ^{o}C. The hydrogen to methyl transformation, albeit small, has an average mean and median change of -7 and -4 ^{o}C. The differences between polymorph pairs were always taken as a positive number. The mean and median change for polymorphs are 11 and 4.5 ^{o}C, respectively. The absolute change for the two selected molecular transformations and polymorph pairs are presented in Figure 7.6. The absolute change was selected rather than change as the focus of the study is to compare the potential magnitude of effect rather than determining the effect itself. This is lower than the mean and median of the absolute change for the hydrogen to methyl transformation which is 30 and 23 °C, respectively. This analysis suggests that even small molecular changes tend to have a larger effect on the melting point than do crystal packing changes. This is in agreement with literature which suggests that the typical energy difference between polymorphs is small (less than 1 kcal/mol) [50].

7.3.2.4 When does crystal change matter?

The melting point difference between polymorphs is typically small (median of 4.5 °C). However, there are cases where the difference is more profound. An attempt was made to identify molecules where crystal change results in a significant change in the melting point. The focus was placed on intermolecular interactions and the molecular shape change.

The number of hydrogen bond donors and acceptors, and their ratio were compared to the change in melting point, but no statistically significant relationship was found. The potential importance of hydrogen bonding was further studied using the hydrogen dimensionality descriptor (Figure 7.7). The polymorph pairs were separated into two

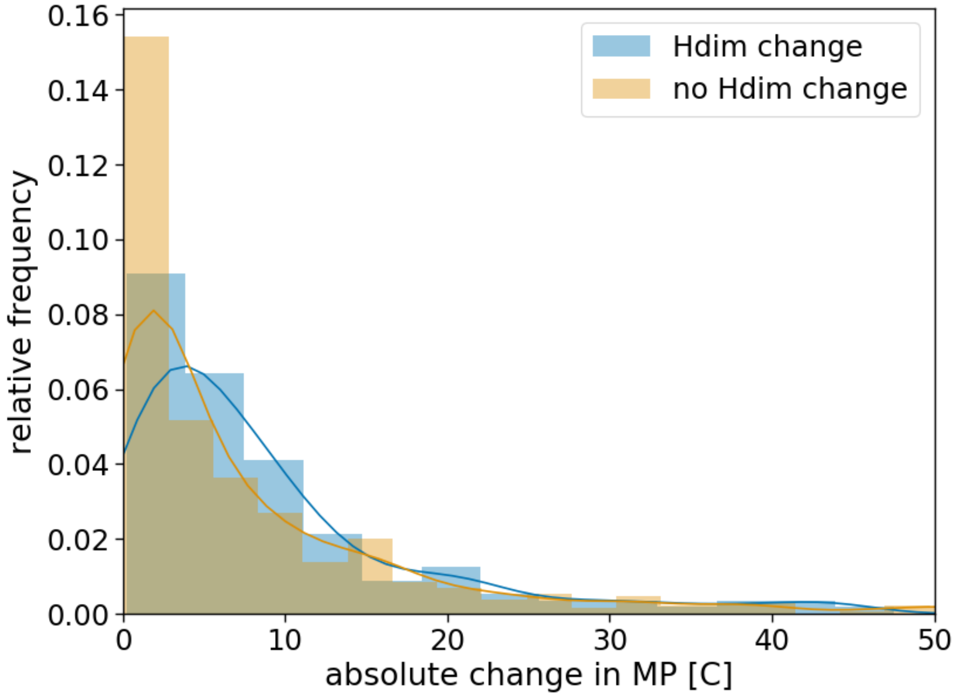


Figure 7.7: Comparison of absolute change of MP for pairs of polymorphs where hydrogen bond dimensionality changes or remains constant.

groups: no hydrogen dimensionality change occurs between the pair, and pairs where there is a change in dimensionality. For the purpose of the study, the degree of change was not considered (i.e. no differentiation between a change from 1D to 2D and a change from 1D to 3D). Pairs of polymorphs where the dimensionality of the hydrogen bonds change have higher median change (6.7 ^{o}C) compared to pairs where there is no change in the dimensionality (4.0 ^{o}C). Polymorphs where the dimensionality of the hydrogen bonding changes, typically have 68 % higher change in melting points. The two distributions are statistically different based on the Mann-Whitney U test (p value = 4.665×10^{-4}). This indicates the potential future avenue of research, focused on data-driven tools to study what molecular properties lead to larger changes in solid state properties across polymorphs. However, it is important to note that the median difference for the two types of polymorphs discussed here is smaller than the change caused by molecular transformations. This may be the reason why the performance difference between the Molecule and Crystal models is small, as even a model that perfectly captures the crystal difference can only be expected to improve the performance by the average of 4.5 ^{o}C.

Apart from the hydrogen bonds, another important intermolecular interaction is the π-π stacking. This interaction was not fully captured by the model as this interaction typically occurs between aromatic rings and not individual atoms which is what the graph is based on. A number of polymorphs with the largest melting point difference were structures with multiple aromatic rings and one or none hydrogen bonding sites (Figure 7.8). The dataset was sliced based on the number of aromatic rings present and whether there were any hydrogen bonds present, but no statistically significant difference was observed. A caveat to note here is the fact that only 21 pairs were studied here.

A number of molecular descriptors were used to study its effect on the difference in melting point across polymorphs. No statistically significant correlation was found between the melting point difference and molecular size (heavy atom count) nor molecular flexibility (nConf20 [212]). The flexibility descriptor was developed to predict the crystallisability of a molecule, so lack of correlation may be an artefact of the lack of polymorph data for difficult to crystallise molecules. The number of conformers that can coexist within a crystal structure was also considered as a descriptor. With the higher number of conformers, the number of possible intermolecular interactions can be expected to increase. The highest observed Z prime in all polymorphs of a given molecule was used as the conformer compatibility descriptor. Z prime is the number of molecules within the asymmetric unit of the crystal structure. For molecules with no self-symmetry, the value is one for structures with one conformer present. However, no statistically significant difference was observed for molecules with different conformer compatibility descriptor values.

7.4 Conclusion

Message Passing Neural Network was used to construct two QSPR models to predict the melting point. The Molecule model, one constructed from molecular graphs,

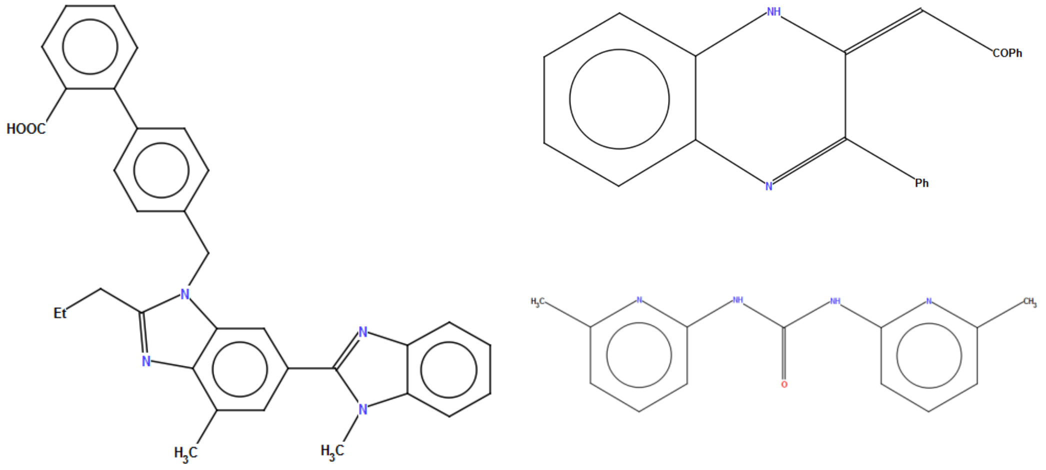


Figure 7.8: Example molecules with large MP difference between polymorphs. Left – XUYHOO (86.0 C), right top – QAPKOH (132.5 C), right bottom – ICAKAY (129.7 C)

achieved R^2 value of 0.628 on the validation set and 0.500 on the test set. The Crystal model, one constructed from molecular and intermolecular graphs along with additional crystal descriptors, achieved an R^2 value of 0.649 and 0.550 on the validation and test sets respectively. The graph-based approach has shown some promise in capturing the molecular and crystal properties. However, further work is necessary to the suitability of the approach for melting point prediction. Recent work in the use of molecular graph has shown promise in the QSPR field [238,239].

Some insights can be derived from the study regarding the ability to capture crystal information and its relative importance. A statistically significant (p value= 2.937×10^{-6}), albeit small improvement (average R^2 = 0.621 and 0.631 on the validation set) was observed between the Molecule and Crystal models. The small performance improvement between the two types of models is likely due to a combination of three reasons: (1) underrepresentation of polymorphs in the dataset, (2) inability of the model to capture solid state specific information, and (3) the typically small property difference between polymorphs.

Only 1.1 % of the molecules are polymorphic within the CSD MP dataset; hence the trained models had limited exposure to structures with only solid state differences. The model did not predict the relative stability of polymorphs (as approximated by comparison of melting points) in a way that is statistically significant. However, this may be due to the small number of polymorph pairs that were studied (21 pairs).

Only geometric considerations were undertaken when assessing the intermolecular interactions which potentially reduced the usefulness of the information captured by the models. The π- π interaction were not explicitly captured by the model which may have contributed to the inability to fully capture the various intermolecular interactions that affect the melting point. Geometry of the π- π interaction affects the strength of the interaction [250], so a way of capturing this beyond the graph method is required.

The typical melting point difference between polymorphs is 4.5 ^{o}C, much smaller than the effects of molecular changes such as the substitution of a hydrogen with a methyl group. Therefore, the potential performance improvement from capturing the crystal information is also small. However, in some cases, the difference between polymorphs is large (over 30 ^{o}C). Potential factors that contribute to the large difference were investigated. Hydrogen bond dimensionality is a potential indicator of this phenomenon; whereas molecular flexibility, the presence of aromatic rings appear not to be indicative, although this may be due to the issue of unknown polymorphs discussed in Chapter 4.

The study has shown that there is some benefit to including crystal information for solid state-specific melting point prediction. The graph-based approach to capturing molecular and crystal information also shows some promise, although further work is required. The ability to predict whether a molecule exhibits polymorphs with a wide range of melting points would be a useful tool to complement the molecule structure-based melting point prediction.

Chapter 8

Conclusion

8.1 Introduction

In this thesis, the extent to which techniques deployed during the Discovery stage can be applied to the Development stage datasets to address challenges encountered at this stage was investigated. Chapter 2 contextualised this challenge within the pharmaceutical product development process and the Material Science Tetrahedron (MST). Based on the survey of the challenges, two specific topics were identified as the focus of the thesis. The topics are: (1) the prediction of the propensity of molecules to form polymorphs and (2) the prediction of crystal structure-specific melting points as an indication of potential solid state-specific solubility. Chapter 2 also provided an overview of the techniques used in the thesis. Matched Molecular Pair Analysis (MMPA) and Quantitative Structure Property Relationship (QSPR) were selected as the two methods of addressing the topics identified. In Chapter 3, a novel database approach to the MMPA was introduced. The MMP database workflow was utilised to study the effects of small molecular transformations on the polymorph propensity (Chapter 4). Issues related to the quality of the polymorphism data is partially addressed in Chapter 5 by the development of a machine learning-based polymorph-redetermination classification method which was benchmarked against the existing methods. The issue related to the low number of MMPs present in the MMPA of polymorph propensity was explored within the context of the different stages of the drug development process in Chapter 6 by introducing Matched Molecular Graphs as a method of exploring the dataset suitability for MMPA. Work on the second topic identified in Chapter 2, crystal structure specific melting point prediction, was discussed in Chapter 7.

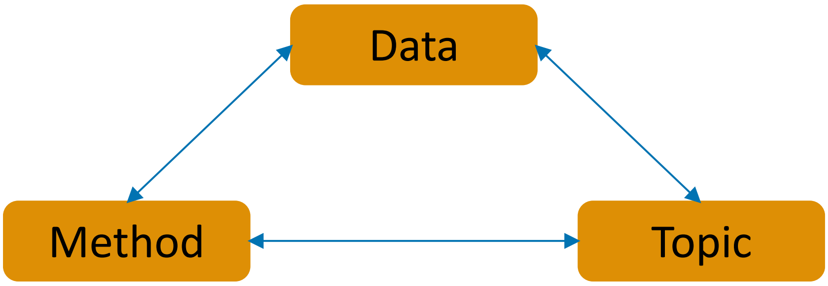


Figure 8.1: The three themes used to discuss the key findings of the thesis.

The key findings from the thesis are discussed within the framework of three interconnected aspects: Data, Empirical method, and research topic, as illustrated in Figure 8.1. Conclusions are drawn on the specific topics studied, as well as on the broader implications for the research in the area of solid state informatics for pharmaceutical development.

The framework was selected due to the unique nature of the data-driven approaches, where the available data and its quality plays a vital role on the suitability of methods and topics. For example, polymorph specific solubility prediction may not be a suitable research topic if the required data availability is limited [70,251]. Similarly, multi-layered neural networks may not be the most suitable tool for QSPR modelling if the dataset has only 10s or 100s of examples due to the high likelihood of overfitting. Any research project needs to consider the interdependence of these three aspects. These are discussed within the scope of the project and as learning outcomes that can be used to inform further research.

8.2 Data Management

The availability and quality of data is a crucial consideration when undertaking a data driven project. The importance of the quality of data is indicated by Wilf Hey's maxim – "Garbage in, Garbage out". Assuming consistent quality, the more data is available, the better empirical models can be developed. The two aspects are discussed within the scope of the project below. Based on the work, recommendations for future projects in a similar research area are presented.

8.2.1 Quality

In Chapter 4 and Chapter 5, several issues related to the polymorphism data were identified. Firstly, many of the molecules in the CSD have not been studied under different conditions to identify all polymorphs (within a reasonable set of conditions). This results in underestimation of polymorphism in the database. Monomorphic adjustment [50] (as described in Chapter 4) was implemented to mitigate the issue by eliminating molecules that are likely to have not been studied extensively. Further adjustments can be done by, for instance, eliminating all molecules that do not have at least 3 distinct structures (as opposed to two as done in the original publication [50] and the thesis where the dataset was reduced by 97 % to 6,633 structures). The process can be continued until the fraction of polymorphic structures matches that of the more heavily screened, smaller datasets. However, this approach also introduces a new bias – where the more commonly studied compounds are overrepresented. This may result in developing a model to predict popular compounds, rather than polymorphic compounds. The process also reduces the dataset size, making it difficult to identify any statistically significant trends.

The uniqueness of the challenges associated with polymorph propensity lies in the inability to assess the data quality and completeness on individual entry level. Namely, the lack of polymorphic data cannot be distinguished between the lack of polymorphism or the lack of research done on the molecule. In the case of melting point, it is easy to identify molecules that have no information on the property. A public database of polymorph screens with studied conditions and resulting crystal structure information, even if no distinct structures were identified, would be a useful tool to aid this challenge. The number of entries to this database would be an indication of how much the molecule has been studied, and the range of conditions of these studies would inform the completeness of polymorph search. Running polymorph screening is typically expensive, so it is unlikely that such a database could be purpose-built by an individual group. However, crystallisation is a commonly studied process, so there is a possibility for such a database to be successfully constructed based on inter-institutional collaboration. Similar databases may also be constructed in-house by pharmaceutical companies to better utilise the data already at their disposal.

The second issue related to the polymorphism data quality is the classification of structures into polymorphs and redeterminations. Majority of crystallographic research focuses on specific molecule or group of molecules. Systematic studies of polymorphism are relatively rare [50,209]. The systematic studies rely on automated methods for polymorph-redetermination classification. The method commonly used is the simulated spectra comparison method[211] (discussed in Chapter 5). However, the method was only tested on a small subset of 386 structures (83 molecular compositions). A more detailed benchmarking was performed in Chapter 5 where the existing and novel machine learning-based automated methods were compared against manually assigned labels on a dataset of 2,951 pairs filtered to eliminate inconsistent manual labels. The best machine learning model achieved a F1 score (defined in

Equation 2.21) of 0.910 with recall and precision of 0.864 and 0.962 respectively. The F1 score of the spectra comparison method was 0.780 with relatively lower recall value of 0.645 (precision was 0.987). The spectra comparison method had lower false positive rate but a higher false negative rate.

The benchmarking also shed light on some of the inconsistencies found with polymorph identification. A research community-wide effort to more vigorously label polymorphs and develop an appropriate automated classification method is needed. The lack of standard method has an impact on other areas of research such as the solid state-specific melting point prediction explored in Chapter 7. A community endorsed, vigorous method of crystal structure classification would clarify these situations and allow research in solid state-specific property prediction to accelerate. To this end, the benchmarking dataset was curated.

8.2.2 Availability

The availability and the dataset size is another consideration that needs to be made when undertaking empirical modelling. This was lightly touched upon above (8.2.1) with the issue of decreasing dataset size of polymorph counts. However, the issue of data availability is an important aspect of any data-driven project in its own right. The history of empirical modelling of molecular properties begun with work on 10s of compounds and a narrow applicability domain [87]. With time, the dataset sizes increased and so did the applicability domain of the empirical models based on the data [171,174]. A more general melting point models were developed using a dataset of over 280,000 molecules [83]. A similar trend can be observed in other areas of empirical modelling, in particular machine learning, where dataset sizes and the capability of the models increased over time [252].

Therefore, the acquisition of data should be one of the priorities when it comes to future development in the area. In most cases, the data is already present; however, in a difficult to use format. According to some estimates, up to 95 % of data is unstructured, meaning it is difficult to access and utilise [253]. In research-driven organisations such as universities, the data is typically arranged project-wise (Figure 8.2). This allows easy retrieval of data for a specific project.

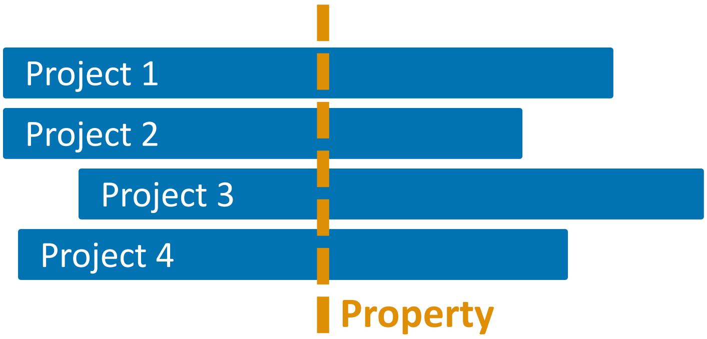


Figure 8.2: Typical data arrangement within research organisations.

The data is arranged per project bases, making it easy to locate all relevant data for a particular project, however it results in difficulty in accessing similar data across multiple projects.

However, search for specific information, such as property data, is difficult if not unfeasible to do on a large scale. There are typically no consistent naming conventions enforced at the organisation level and the data is structured appropriately for the needs of the specific project. Along with the difficulty in accessing the data itself, metadata extraction is also challenging. There is little incentive for individuals responsible for these projects to make the data more easily accessible across projects as this often requires additional work. Notable exceptions exist where data from multiple sources was curated [80,82,83]. These datasets can be utilised within the cheminformatics community for the study of the respective properties. Approaches such as the FAIR data management principles [254] and EPSRC expectation around data availability [255] attempt to address some of these issues. However, as long as the data producers (ones who hold responsibility for data generation) and the data consumers (ones that reap the benefits of data availability) are disjointed, the problems are likely to persist. Therefore, for a consistent solution to the problem, an organisational level and preferable inter-organisational strategy is required. Improving the data accessibility is likely to increase further research using empirical methods.

8.2.3 Suitability

The Matched Molecular Graph (MMG) was developed (Chapter 6) to assess the suitability of a dataset for MMPA. MMG allows for dataset comparison and visualisation of MMPs. The analysis of the constructed graph on the monomorphism-corrected CSD single component dataset revealed that 74.9 % of molecules did not have any MMPs, thus were not taken into account when performing MMPA. ChEMBL-NTD set 14 (GSK TCAKS (Tres Cantos Anti-Kinetoplastids Set) dataset) [229] was selected as a representative Discovery-like dataset. The graph property analysis revealed that the Discovery dataset had 58.2 % of molecules with at least one MMP (compared to 25.1 %) and the molecules that had at least one MMP, typically had 5.34 MMPs (compared to 3.33). These factors resulted in the large difference in the number of MMPs for the two datasets despite the same size (10,321 MMPs in Discovery dataset, 2,776 in the used CSD dataset).

The MMG analysis between the two datasets indicated that there are some differences in the datasets between Discovery and Development. Such differences should be considered when attempting to utilise other data-driven approaches, that have been successfully implemented in Discovery settings, to Development datasets. The analysis also showed the need to consider dataset suitability beyond the size and the target value quality. Although not quantitatively vigorous, MMG can be used to assess the suitability of datasets for MMPA.

8.3 Empirical Method

A number of empirical methods ranging from simple linear regression[256] to deep learning [257] have been used within cheminformatics. In this thesis Matched Molecular Pairs Analysis (Chapter 4), machine learning (Chapter 5) and deep learning (Chapter 7) were used. MMPA represent an easily interpretable empirical method while the Neural Network represents a more capable of capturing complex relationships but more difficult to interpret class of methods. The performance of the two methods within the project, as well as potential future implementations are discussed here.

8.3.1 Message Passing Neural Networks

The Message Passing Neural Networks (MPNN) were used to develop melting point QSPR models using molecular and crystal information. The model uses simple descriptors (element and covalent bond type) to obtain molecular information which is complemented by crystal information based on the potential intermolecular interaction, hydrogen bond dimensionality, and conformation change between structure in the solid state and gas phase (shape change descriptor). A set of Molecule models (only had access to molecular structure) and Crystal models (had access to molecular and crystal information) were trained. Some improvement of the Crystal model ($R^2 = 0.550$) was observed compared to the Molecule model ($R^2 = 0.500$). This suggest that the model is capable of capturing some intermolecular interactions to improve the performance. The inability to capture the strength of interactions as well as limited ability to capture π-π interactions likely contributed to the small improvement of the models, Recently, a similar model architecture was used to successfully embed drug-target interactions using a graph-based approach to classify molecules into active and inactive (Area Under the Curve = 0.968 and 0.935 on two test sets) [239]. Although the test set performance of the Crystal model was lower than on validation set (0.649), it has shown the ability of MPNNs to capture some intermolecular interactions.

8.3.2 Matched Molecular Pairs – Graphs and Analysis

The Matched Molecular Pair framework was used for typical MMPA as well as for construction of Matched Molecular Graph (MMG) and related analysis of the dataset suitability for MMPA. The first step in the process is the identification of the MMPs within a given dataset as discussed in Chapter 3. Several key issues relating to this step were identified and addressed. As identified in other works on MMPs identification using unspecified transformation methods (defined in 2.5.1.2), it is important to limit the maximum size of the change of a MMP [180]. This is done either by limiting the maximum size of the change or the ratio of the size of change to the molecule. However, even within the limited scope, many unnecessary MMPs are maintained. This includes a set of MMPs between the same pair of molecules. The same pair of molecules may be cut in different ways to yield several MMPs. The smallest change is most likely to be observed across multiple pairs of molecules, contributing to the analysis. The other MMPs of the same molecule only contribute to an increase in the number of MMPs. In the developed MMP Database, only one MMP is kept for a pair of molecules based on the largest context (i.e. the smallest change as these are more common). The adjustments made to the MMP identification procedure reduce the number of total MMPs identified within a dataset, but ensures that the pairs identified are likely to contain transformations that are more common (thus more useful).

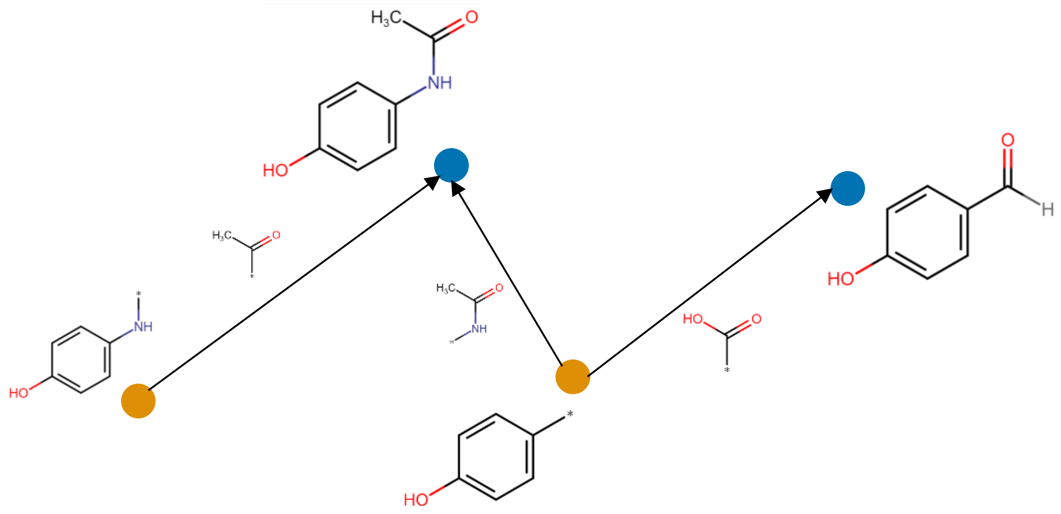


Figure 8.3: Graph based MMP identification.

Two types of nodes are used: contexts (gold) and molecules (blue). Edges contain information on the core associated with the connected context and molecule.

However, the reduction in the number of MMPs along with the reduction in the dataset (monomorphic correction) resulted in MMPA that did not identify any statistically significant transformations for polymorph propensity (Chapter 4). Lack of polymorph data (discussed in 8.2) and the small number of MMPs were identified as the most likely reason for the lack of statistically significant transformation.

The MMPA carried out in the thesis led to the development of the concept of MMG. MMG can be further expanded to extract more information from an MMP Database. Work can be performed to improve the process of database construction by abandoning relational databases for graph databases. These types of database put more emphasis on the relationships between elements (edges) [258]. The database structure allows for more computationally efficient parsing based in the edges. The MMP identification step could be replaced by simpler, more computationally efficient, graph-based operations. Graph databases typically allow the specification of different types of nodes and edges, as well as the addition of properties to these. Molecules can be added to the database as nodes, with attached properties such as the molecule ID, structure information (SMILES), properties value (dark blue nodes in Figure 8.3). Upon fragmentation, the context-core pairs can be screened based on the maximum size of the change and maximum ratio of change criteria (as discussed in Chapter 3 and Chapter 4). The filtered contexts can be added to the database as a different type of node (gold nodes in Figure 8.3). An edge can be added between the molecule and the added context, containing the core structure as its property (arrow in Figure 8.3). MMPs can be identified by finding all molecules that are connected via a context node. The search can be easily modified to find all MMPs with a given context, or all MMPs of a given molecule or a set of molecules. Another layer to the graph can be added to incorporate crystal information. A new type of node, a crystal node, can be added that contains the crystal ID (refcode) and any other relevant crystal property (e.g lattice type). These can be connected to molecule nodes based on the molecular composition. Solvates and co-crystals would be connected to multiple molecules, so an edge property used to indicate whether a molecule is a primary component or a solvent molecule may be added.

The MMG can also be potentially used for property prediction based on MMPA if a dense enough graph can be constructed. For a new molecule (m) with an unknown property value (p_m), all MMPs can be identified. A set of all molecules that are MMPs with the given molecule are given by the following equation where $\mathcal{E}$ represents all edges (MMPs) in an MMG.

$$MMP(m) = \{n\colon (m, n) \in \mathcal{E}\ \} \qquad \text{Equation 8.1}$$

A series of predictions can be made based on each molecule and the corresponding transformations, where *pi* is the property of the molecule *i* and t_i is the average effect of the transformation that links molecule *i* and *m*.

$$\widehat{p_{m_i}} = p_i + t_i \ where\ i\ \in MMP(m) \qquad \text{Equation 8.2}$$

A weight (w_i) may be assigned to each of the MMP based on how well the effects of the transformation are studied (number of occurrences of the given transformation within the MMG). Hence, the final property prediction of the new molecule (m) can be calculated based on the following.

$$\widehat{p_m} = \frac{\sum_{i \in MMP(m)} (p_i + t_i)}{\sum_{i \in MMP(m)} w_i} \qquad \text{Equation 8.3}$$

The method is akin to the k-Nearest Neighbours (kNN) method [259]. kNN method predicts the target value using *k* samples with the nearest feature values. In the proposed method, the neighbour identification is performed based on molecular transformations instead. A further distinction is the way in which the predicted value is calculated. The effects of the transformation are used (along with a weighting based on statistical confidence in the effect and its size) to compute the predicted property value. In this way, a molecule that differs only slightly from a known one may be predicted a vastly different property value if the transformation relating the two molecules is known to have a large effect on the studied property. In this way, knowledge of activity cliffs [41] may be included in the predictive model. Further work is required to assess the feasibility and usefulness of such an approach.

As stated above, for this method to work, a dense MMG is required, unlike the ones studied in the thesis (Chapter 6). However, the concept can be applied in tandem with a QSPR model on a more sparse MMG. Prediction-driven MMPA was previously used to generate new MMPs to enable better analysis of transformations with low MMP count [198] (introduced in 2.5.2).

Model explainability is an important aspect of regulatory approval of QSPR models [260] as well as an active area of research within the machine learning community [15,261]. Local Interpretable Model-agnostic Explanations (LIME) relies on sampling random values in the feature space surrounding the prediction of interests and constructs a local, easily interpretable model (decision tree, linear model etc.) [261]. In many QSPR cases, the feature space may be not continuous and the meaning behind each feature may be difficult to interpret by a researcher. Instead, a chemical space surrounding the prediction of interest may be sampled using MMPs (Figure 8.4). A set of new predictions (dark blue in the figure) can be made to generate MMPs with the prediction of interest (gold in the figure). Several strategies for the MMP generation are feasible. Pairs such that each functional group on the molecule of interest is replaced with an inert functional group is one possibility. This is illustrated in Figure 8.4 with the replacement of terminating functional groups with a methyl group and chain functional groups with a methylene bridge. This may result in the ability to obtain the group contribution to the predicted property value of the molecule of interest which can lead to increased model explainability.

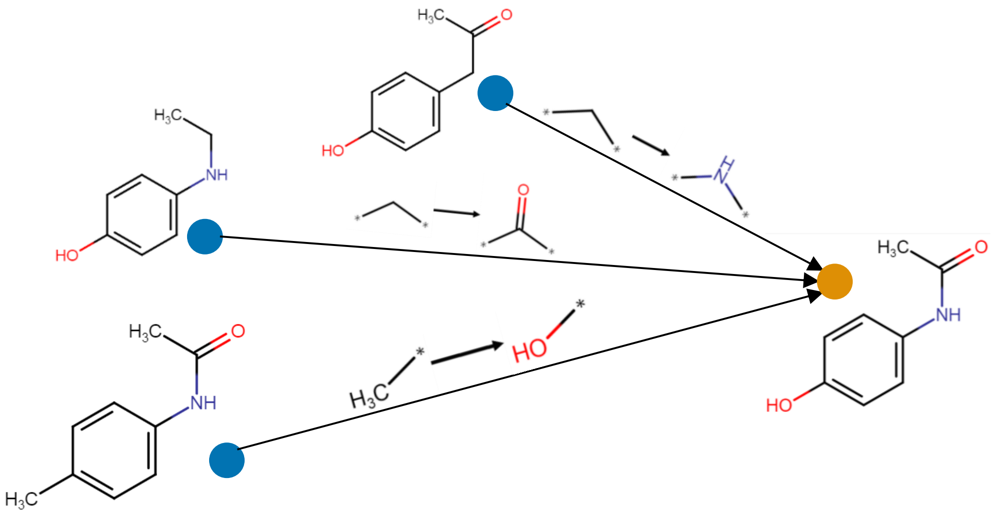


Figure 8.4: MMP based group contribution for QSPR model prediction explanation.

For a given prediction (gold) a set of MMPs can be generated (blue) to determine the group contributions to the prediction.

Alternatively, MMPs may be generated to match common transformations within the studied dataset. Rather than increasing the number of MMPs for better MMPA [198], it can be used to compare the predicted MMPs to the ones from the datasets. This allows for the assessment of the QSPR model's ability to adequately map the effect of these transformations. Similar analysis, without the generation of new molecules, was carried out in Chapter 7 to assess the melting point model performance. The method provided some means of more in-depth analysis of model performance which can be further expanded with future research in this area.

8.4 Research Topic

The research topics covered in the thesis were contextualised within the Material Science Tetrahedron presented in Chapter 2. The focus was placed on the multi-level structure property relationship for properties relevant to the pharmaceutical product development as visualised in Figure 8.5. The key findings of the for the polymorph propensity study and the solid state specific melting point prediction are summarised below. Based on the discussion of data (8.2) and the empirical method analysis (8.3), future research topics are suggested.

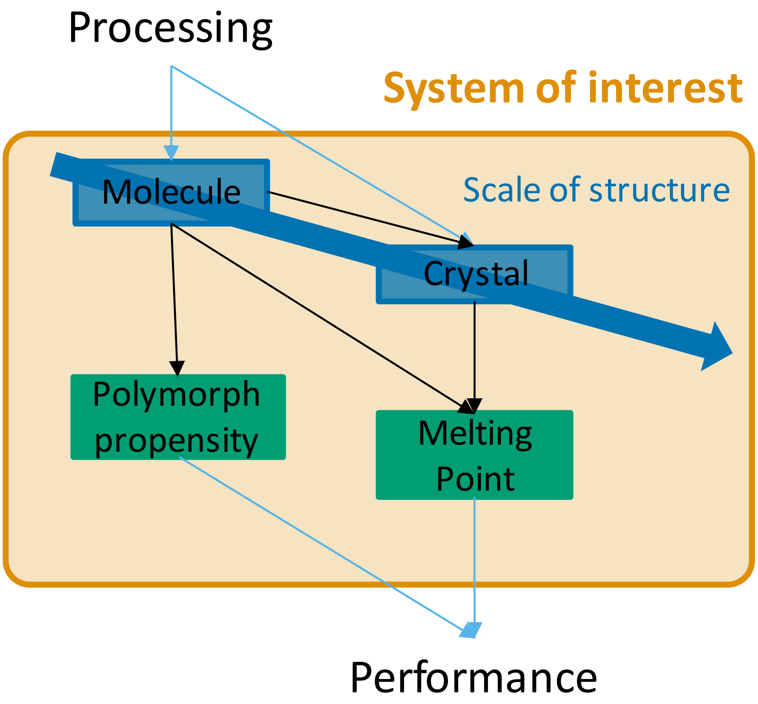


Figure 8.5: Structure Property Relationship studied in the thesis.

The figure is based on the adapted Material Science Tetrahedron introduced in Chapter 2.

8.4.1 Polymorph propensity

Polymorph propensity was studied using MMP approach in Chapter 4. No statistically significant transformations were identified. Beyond the data related limitations discussed earlier, the phenomenon of polymorphism may be too complex to capture using MMPA. The way in which a transformation affects the propensity may be highly dependent on the context of the pair (part of the MMP that does not change). However, even when some properties of the context (such as flexibility and number of hydrogen bond donors/acceptors) were taken into account, no meaningful correlations were found. Furthermore, previous research into the effects of molecular transformations on crystal packing found that some transformations had consistent effect [189]. Therefore, it is likely that the MMPA did not yield any statistically significant results due to the analysis technique selected and the available data as previously discussed (8.2.1 and 8.3.2). An empirical method that utilises the entirety of the dataset (such as QSPR modelling) are likely to be more suitable for the propensity study as it can utilise the entirety of the dataset. The challenge of the low data quality may be partially addressed by focusing on prediction of polymorphism (i.e. classification into monomorphic and polymorphic compounds). If at least two polymorphs are identified, the molecule would be considered to be polymorphic without the need to have found all possible polymorphs (as is the case for polymorph propensity study). However, this does not fully solve the issue of unknown polymorphs, as many false monomorphic molecules will skew the results. Industrial, polymorph screened datasets suggest that up to 66 % of compounds may be polymorphic [50], which is much higher than the observed rate of polymorphism in the CSD. Polymorphism and polymorph propensity are of great interest, however it may be difficult to construct empirical models without access to datasets with adequate quality.

8.4.2 Solid state specific melting point

Many properties relevant to the pharmaceutical product development are solid state-dependent (see 2.1.4.3). In this thesis, melting point was studied due to the availability of solid state specific data within the CSD. Previous work done on incorporating solid state information indicated the inability to appropriately capture the crystal structure as a reason for little improvement in model performance compared to molecule-based models [70,262]. Graph-based structure description complemented by crystal descriptors were used here in an attempt to capture solid state information.

The improvement between models that did not have any crystal information and ones that did was relatively small (0.628 to 0.649 on the validation set and 0.500 to 0.550 on the test set). This can be interpreted in two ways, either the crystal structure typically does not affect the property studied or the crystal structure information is not adequately captured. It is also possible that a combination of the two factors contributes to the small improvement. The improvement is particularly disappointing if the labour intensity for the acquisition of the additional crystal structure features are considered. These features require the crystal structure to be solved before computing the values. This is much more demanding than features computed based on molecular structure alone. Furthermore, the change in property across polymorphs is typically small, especially compared to the effects of small molecular changes (MMPs). Therefore, further attempts to capture crystal structure information for solid state-specific bulk property prediction may not yield significant improvements in predictive power in relation to the labour intensity in acquiring solid state specific data. This is not to diminish the importance of solid state informatics in developing better predictive models in areas where crystal structure plays an important role such as crystal surface properties [76,79].

8.4.3 Future research topics

Analysis of polymorph propensity was complicated due to the reduction in dataset and unreliability of data. However, the MMP approach has previously been used to see which transformation maintain isostructurality [189]. This can be expanded by considering lattice energy changes caused by the molecular transformation. It is possible to calculate group contribution to lattice energy [263]. The group contribution of the context of the MMP can be calculated. A large change indicates a disruption to the intermolecular interaction. By carrying out MMPA, it may be possible to determine what transformations are likely to disrupt the intermolecular interactions. This can potentially be linked to morphology modification. Some crystals grow as needles due to the preferential intermolecular interactions [249]. By disrupting its intermolecular interactions, it may be possible to avoid molecules with undesirable morphological tendencies. In contrast, it may also be useful to identify transformations that do not disrupt the crystal packing but influence some other property of interest or vice versa.

The work on solid state-specific melting point, as well as previous works [50] indicated that property difference between polymorphs is typically small, with few notable exceptions where the difference is significant as such as the case of the solubility of ritonavir [8]. A molecular structure based QSPR model to predict the size of the property difference between polymorphs would be of great interest. However, the issue of unknown polymorphs with unknown property value poses a similar challenge as the one encountered in Chapter 4.

8.5 Concluding Remarks

The thesis aimed to investigate the extent to which data-driven techniques, typically used during Discovery, could be used to address challenges commonly addressed during the Development stage of the pharmaceutical development process. The research was contextualised with the MST [5]. Emphasis was placed on the structure-property relationship, where the framework was expanded to explicitly include the concept of scale – for structures (molecular and crystal structure) as well as the properties. The difference in the relevant scales of structure presents a major difference between the two stages of drug development process where in Discovery, molecular structure is considered, while in Development larger structures such as crystals and particles are relevant.

Effects of molecular changes on the propensity of the molecule to exhibit polymorphism were studied using MMPA, a method commonly used for the analysis of Discovery-related properties [11]. To carry out the analysis, a database was developed independently of other researchers [201]. A database of single component crystal structures from the CSD was constructed and the analysis was carried out. However, no statistically significant transformations were observed. Several potential reasons were investigated for the lack of any noticeable trends. A Matched Molecular Graph was constructed to investigate the properties of the dataset itself that may have affected the analysis. Despite a large total number of MMPs identified, only 25.2 % of molecules had at least one MMP. A Discovery dataset of the same size (TCAKS dataset) [229] had 58.2 % of molecules with at least one MMP. This indicates that the Development datasets may be less suitable for MMPA analysis compared to the Discovery datasets.

The subsequent analysis focused on the data quality of the polymorph dataset. Unobserved polymorphs have likely skewed the results of the analysis. A general trend was observed that the more commonly studied structures (as approximated by the number of redeterminations within the CSD) had more polymorphs as well.

Another issue related to the study of polymorphism that was addressed is the need for automated methods for polymorph and redetermination classification of crystal structures. A benchmark dataset was derived from the CSD based on the availability of polymorph labelling provided by a single research group (per molecular composition). The commonly used spectra comparison method achieved an F1 score of 0.780 with relatively low recall value of 0.645 (precision was 0.987). Machine learning-based approach achieved F1 score of 0.910 with recall and precision of 0.864 and 0.962, respectively. The benchmark dataset may be used as a starting point for future work aimed at addressing the issue of polymorph and redetermination classification.

Another challenge that was addressed in the thesis is the ability to adequately capture crystal structure information for machine learning-based modelling. A graph was constructed based on identified intermolecular interactions, along with other crystal descriptors (shape change descriptor and hydrogen bond dimensionality) to investigate if this improves the performance of melting point models. The observed improvement (0.628 to 0.649 on the validation set and 0.500 to 0.550 on the test set) was relatively small. This may be due to the inability to capture solid state information. However, the typically small effect of solid state changes on the melting point is also likely to have contributed to the small performance improvement.

The research opened several avenues for further investigation. Although MMPA of polymorph propensity did not identify any statistically significant trends, the ease of interpretability remains an attractive feature of MMPA. The method could be applied to study the effects of molecular transformation on the disruption of intermolecular interactions via lattice energy analysis. This would build on existing MMPA of iso-structurality [189]. However, issues of systematic and accurate lattice energy calculation need to be addressed [70].

Polymorphs typically have similar properties with a small fraction of notable exceptions [50]. An interesting scope of future research is the prediction of the difference of property values between solid state structures based on the molecular structure. If successful, this would allow to anticipate the potential Development challenges during Discovery.

The research and development of treatments for illnesses has been a major challenge for civilisations for millennia [16,18]. In recent years, the efficiency of the pharmaceutical product development process has been decreasing [33,34]. The studies presented in this thesis provide partial solutions to problems addressed at the Development stage. A novel method of analysing datasets – Matched Molecular Graphs – showed that Development datasets tend to have less similar molecules, resulting in methods that are commonly used in the Discovery stage (such as MMPA) being less suitable. A graph-based approach to capturing crystal information showed some promising results. The work in the thesis can inform future research in the area of solid state informatics to address the challenges encountered when developing new pharmaceutical products.

References

[1] J.P. Bunker, The role of medical care in contributing to health improvements within societies, Int. J. Epidemiol. 30 (2001) 1260–1263. https://doi.org/10.1093/ije/30.6.1260.

[2] World Health Organization, Essential Medicines List, World Health Organization, 2015. http://www.who.int/selection_medicines/list/en/ (accessed February 8, 2017).

[3] M. Lindenberg, S. Kopp, J.B. Dressman, Classification of orally administered drugs on the World Health Organization Model list of Essential Medicines according to the biopharmaceutics classification system, Eur. J. Pharm. Biopharm. 58 (2004) 265–278. https://doi.org/10.1016/j.ejpb.2004.03.001.

[4] Science Engineering Committee on Materials, Solid State Sciences Committee, Board on Physics and Astronomy, and R. Commission on Physical Sciences, Mathematics, Board National Materials Advisory, Commission on Engineering and Technical Systems, National Research Council, MATERIALS SCIENCE AND ENGINEERING FOR THE 1990s, National Academy Press, Washington DC, 1989. https://www.nap.edu/read/758/chapter/1.

[5] C. Sun, Materials Science Tetrahedron—A Useful Tool for Pharmaceutical Research and Development, J. Pharm. Sci. 98 (2009) 1671–1687.

[6] N. Kawashita, H. Yamasaki, T. Miyao, K. Kawai, Y. Sakae, T. Ishikawa, K. Mori, S. Nakamura, H. Kaneko, <Review> A Mini-review on Chemoinformatics Approaches for Drug Discovery, J. Comput. Aided Chem. 16 (2015) 15–29. https://doi.org/10.2751/jcac.16.15.

[7] Y.-C. Lo, S.E. Rensi, W. Torng, R.B. Altman, Machine learning in chemoinformatics and drug discovery, Drug Discov. Today. 23 (2018) 1538–1546. https://doi.org/10.1016/J.DRUDIS.2018.05.010.

[8] J. Bauer, S. Spanton, R. Henry, J. Quick, W. Dziki, W. Porter, J. Morris, Ritonavir: An Extraordinary Example of Conformational Polymorphism, Pharm. Res. 18 (2001) 859–866. https://doi.org/10.1023/A:1011052932607.

[9] International Conference on Harmonisation of Technical Requirements for

Registration Of Pharmaceuticals for Human Use, Specifications: Test Procedures and Acceptance Criteria for New Drug Substances and New Drug Products: Chemical Substances Q6A, 1999. https://www.ich.org/fileadmin/Public_Web_Site/ICH_Products/Guidelines/Quality/Q6A/Step4/Q6Astep4.pdf (accessed July 4, 2019).

[10] C. Nantasenamat, C. Isarankura-Na-Ayudhya, T. Naenna, V. Prachayasittikul, A PRACTICAL OVERVIEW OF QUANTITATIVE STRUCTURE-ACTIVITY RELATIONSHIP, EXCLI J. 8 (2009) 74–88. https://www.excli.de/vol8/Prachayasittikul_04_2009/Prachayasittikul_050509_proof.pdf (accessed May 14, 2019).

[11] K. Müller, The Power of MMPA and a Teaching Lesson in Medicinal Chemistry, J. Med. Chem. 55 (2012) 1815–1816. http://pubs.acs.org/doi/abs/10.1021/jm300163y (accessed July 8, 2016).

[12] A. de la V. de León, J. Bajorath, Prediction of Compound Potency Changes in Matched Molecular Pairs Using Support Vector Regression, J. Chem. Inf. Model. 54 (2014) 2654–2663. http://pubs.acs.org/doi/abs/10.1021/ci5003944 (accessed July 8, 2016).

[13] A.S. Raw, M.S. Furness, D.S. Gill, R.C. Adams, F.O. Holcombe, L.X. Yu, Regulatory considerations of pharmaceutical solid polymorphism in Abbreviated New Drug Applications (ANDAs), Adv. Drug Deliv. Rev. 56 (2004) 397–414. https://doi.org/10.1016/J.ADDR.2003.10.011.

[14] P. Polishchuk, Interpretation of Quantitative Structure-Activity Relationship Models: Past, Present, and Future, J. Chem. Inf. Model. 57 (2017) 2618–2639. https://doi.org/10.1021/acs.jcim.7b00274.

[15] E. Štrumbelj, I. Kononenko, Explaining prediction models and individual predictions with feature contributions, Knowl. Inf. Syst. 41 (2014) 647–665. https://doi.org/10.1007/s10115-013-0679-x.

[16] A.W. Jones, Early drug discovery and the rise of pharmaceutical chemistry, Drug Test. Anal. 3 (2011) 337–344. https://doi.org/10.1002/dta.301.

[17] D.J. W, Pharmaceutical Industry, Britannica. (2018). https://www.britannica.com/technology/pharmaceutical-industry.

[18] E.K. Teall, Medicine and Doctoring in Ancient Mesopotamia, Gd. Val. J. Hist. 3 (2014) 1–8. http://scholarworks.gvsu.edu/gvjhhttp://scholarworks.gvsu.edu/gvjh/vol3/iss1/2 (accessed April 5, 2019).

[19] S.M. Paul, D.S. Mytelka, C.T. Dunwiddie, C.C. Persinger, B.H. Munos, S.R. Lindborg, A.L. Schacht, How to improve R&amp;D productivity: the pharmaceutical industry’s grand challenge, Nat. Rev. Drug Discov. 9 (2010) 203–214. https://doi.org/10.1038/nrd3078.

[20] J.W. Scannell, J. Bosley, When Quality Beats Quantity: Decision Theory, Drug Discovery, and the Reproducibility Crisis., PLoS One. 11 (2016) e0147215. https://doi.org/10.1371/journal.pone.0147215.

[21] J.G. Mahdi, A.J. Mahdi, A.J. Mahdi, I.D. Bowen, The historical analysis of aspirin discovery, its relation to the willow tree and antiproliferative and anticancer potential, Cell Prolif. 39 (2006) 147–155. https://doi.org/10.1111/j.1365-2184.2006.00377.x.

[22] W. Hamilton, The history of medicine, surgery and anatomy, Henry Colburn and Richard Bentley, London, 1831. https://archive.org/details/historyofmedicin02unse.

[23] C. Yapijakis, Hippocrates of Kos, the father of clinical medicine, and Asclepiades of Bithynia, the father of molecular medicine, In Vivo (Brooklyn). 23 (2009) 507–14. http://www.ncbi.nlm.nih.gov/pubmed/19567383 (accessed April 9, 2019).

[24] C. Krishnamurti, S.C. Rao, The isolation of morphine by Serturner., Indian J. Anaesth. 60 (2016) 861–862. https://doi.org/10.4103/0019-5049.193696.

[25] D. Taylor, The Pharmaceutical Industry and the Future of Drug Development, Royal Society of Chemistry, Online, 2015. https://doi.org/10.1039/9781782622345-00001.

[26] L.X. Yu, Pharmaceutical Quality by Design: Product and Process Development, Understanding, and Control, Pharm. Res. 25 (2008) 781–791. https://doi.org/10.1007/s11095-007-9511-1.

[27] M. Entzeroth, H. Flotow, P. Condron, Overview of high-throughput screening,

Curr. Protoc. Pharmacol. 44 (2009) 9.4.1-9.4.27. https://doi.org/10.1002/0471141755.ph0904s44.

[28] H. Gubler, U. Schopfer, E. Jacoby, Theoretical and experimental relationships between percent inhibition and IC50 data observed in high-throughput screening, J. Biomol. Screen. 18 (2013) 1–13. https://doi.org/10.1177/1087057112455219.

[29] M. Palucki, J.D. Higgins, E. Kwong, A.C. Templeton, Strategies at the Interface of Drug Discovery and Development: Early Optimization of the Solid State Phase and Preclinical Toxicology Formulation for Potential Drug Candidates, J. Med. Chem. 53 (2010) 5897–5905. https://doi.org/10.1021/jm1002638.

[30] M. Segall, A. Chadwick, The risks of subconscious biases in drug-discovery decision making, Futur. Med. Chem. 3 (2011) 771–4. https://doi.org/10.4155/FMC.11.33.

[31] T.J. DiFeo, Drug Product Development: A Technical Review of Chemistry, Manufacturing, and Controls Information for the Support of Pharmaceutical Compound Licensing Activities, Drug Dev. Ind. Pharm. 29 (2003) 939–958. https://doi.org/10.1081/DDC-120025452.

[32] K.M. Lee, Overview of Drug Product Development, Curr. Protoc. Pharmacol. . (2001) 7.3.1-7.3.10. https://doi.org/10.1002/0471141755.ph0703s15.

[33] T.J. Hwang, D. Carpenter, J.C. Lauffenburger, B. Wang, J.M. Franklin, A.S. Kesselheim, Failure of Investigational Drugs in Late-Stage Clinical Development and Publication of Trial Results, JAMA Intern. Med. 176 (2016) 1826. https://doi.org/10.1001/jamainternmed.2016.6008.

[34] J.W. Scannell, A. Blanckley, H. Boldon, B. Warrington, Diagnosing the decline in pharmaceutical R&amp;D efficiency, Nat. Rev. Drug Discov. 11 (2012) 191–200. https://doi.org/10.1038/nrd3681.

[35] C.A.S. Bergström, W.N. Charman, C.J.H. Porter, Computational prediction of formulation strategies for beyond-rule-of-5 compounds, Adv. Drug Deliv. Rev. 101 (2016) 6–21. https://doi.org/10.1016/J.ADDR.2016.02.005.

[36] K. Raza, P. Kumar, S. Ratan, R. Malik, S. Arora, Polymorphism: The Phenomenon Affecting the Performance of Drugs, SOJ Pharm. Pharm. Sci. 1

(2014) 10. https://doi.org/10.15226/2374-6866/1/2/00111.

[37] A.J. Blacker, M.T. Williams, Introduction, in: A.J. Blacker, M.T. Williams (Eds.), Pharm. Process Dev., Royal Society of Chemistry, Cambridge, 2011.

[38] U.S. Food and Drug Administration, Pharmaceutical Quality for the 21st Century A Risk-Based Approach Progress Report, 2007. https://www.fda.gov/about-fda/center-drug-evaluation-and-research/pharmaceutical-quality-21st-century-risk-based-approach-progress-report (accessed June 21, 2019).

[39] A.T. Chadwick, M.D. Segall, Overcoming psychological barriers to good discovery decisions, Drug Discov. Today. 15 (2010) 561–569. https://doi.org/10.1016/J.DRUDIS.2010.05.007.

[40] E. Gawehn, J.A. Hiss, G. Schneider, Deep Learning in Drug Discovery, Mol. Inform. 35 (2016) 3–14. https://doi.org/10.1002/minf.201501008.

[41] D. Stumpfe, H. Hu, J. Bajorath, Evolving Concept of Activity Cliffs, ACS Omega. 4 (2019) 14360. https://doi.org/10.1021/acsomega.9b02221.

[42] Y. Zhou, S. Cahya, S.A. Combs, C.A. Nicolaou, J. Wang, P. V. Desai, J. Shen, Exploring Tunable Hyperparameters for Deep Neural Networks with Industrial ADME Data Sets, J. Chem. Inf. Model. (2019) acs.jcim.8b00671. https://doi.org/10.1021/acs.jcim.8b00671.

[43] K. Liu, X. Sun, L. Jia, J. Ma, H. Xing, J. Wu, H. Gao, Y. Sun, F. Boulnois, J. Fan, Chemi-Net: A molecular graph convolutional network for accurate drug property prediction, Comput. Res. Repos. abs/1803.0 (2018). http://arxiv.org/abs/1803.06236 (accessed July 19, 2018).

[44] T. Loftsson, Physicochemical Properties and Pharmacokinetics, in: Essent. Pharmacokinet., Academic Press, online, 2015: pp. 85–104. https://doi.org/10.1016/B978-0-12-801411-0.00003-2.

[45] C.A.S. Bergström, P. Larsson, Computational prediction of drug solubility in water-based systems: Qualitative and quantitative approaches used in the current drug discovery and development setting, Int. J. Pharm. 540 (2018) 185–193. https://doi.org/10.1016/J.IJPHARM.2018.01.044.

[46] F. Meng, W. Xu, Drug permeability prediction using PMF method, J. Mol.

Model. 19 (2013) 991–997. https://doi.org/10.1007/s00894-012-1655-1.

[47] K.R. Chu, E. Lee, S.H. Jeong, E.-S. Park, Effect of particle size on the dissolution behaviors of poorly water-soluble drugs, Arch. Pharm. Res. 35 (2012) 1187–1195. https://doi.org/10.1007/s12272-012-0709-3.

[48] R.B. Hammond, K. Pencheva, K.J. Roberts, T. Auffret, Quantifying solubility enhancement due to particle size reduction and crystal habit modification: Case study of acetyl salicylic acid, J. Pharm. Sci. 96 (2007) 1967–1973. https://doi.org/10.1002/jps.20869.

[49] R. Censi, P. Di Martino, Polymorph Impact on the Bioavailability and Stability of Poorly Soluble Drugs, Molecules. 20 (2015) 18759–18776. https://doi.org/10.3390/molecules201018759.

[50] A.J. Cruz-Cabeza, S.M. Reutzel-Edens, J. Bernstein, Facts and fictions about polymorphism, Chem. Soc. Rev. 44 (2015) 8619–8635. https://doi.org/10.1039/c5cs00227c.

[51] H.P.G. Thompson, G.M. Day, Which conformations make stable crystal structures? Mapping crystalline molecular geometries to the conformational energy landscape, Chem. Sci. 5 (2014) 3173–3182. https://doi.org/10.1039/c4sc01132e.

[52] A.Y. Lee, D. Erdemir, A.S. Myerson, Crystal Polymorphism in Chemical Process Development, Annu. Rev. Chem. Biomol. Eng. 2 (2011) 259–280. https://doi.org/10.1146/annurev-chembioeng-061010-114224.

[53] J.B.O. Mitchell, Machine learning methods in chemoinformatics, Comput. Mol. Sci. 4 (2014) 468–481.

[54] S. Pirhadi, F. Shiri, J.B. Ghasemi, Multivariate statistical analysis methods in QSAR, RSC Adv. 5 (2015) 104635–104665. https://doi.org/10.1039/C5RA10729F.

[55] A.G. Dossetter, E.J. Griffen, A.G. Leach, Matched Molecular Pair Analysis in drug discovery, Drug Discov. Today. 18 (2013) 724–731. https://doi.org/10.1016/j.drudis.2013.03.003.

[56] D. Weininger, SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules, J. Chem. Inf. Model. 28

(1988) 31–36. https://doi.org/10.1021/ci00057a005.

[57] F. Scarselli, M. Gori, Ah Chung Tsoi, M. Hagenbuchner, G. Monfardini, The Graph Neural Network Model, IEEE Trans. Neural Networks. 20 (2009) 61–80. https://doi.org/10.1109/TNN.2008.2005605.

[58] M. Ladd, R. Palmer, Lattices and Space-Group Theory, in: Struct. Determ. by X-Ray Crystallogr., Springer US, Boston, MA, 2013: pp. 51–110. https://doi.org/10.1007/978-1-4614-3954-7_2.

[59] H. Arnold, M.I. Aroyo, E.F. Bertaut, H. Burzlaff, G. Chapuis, W. Fischer, H.D. Flack, A.M. Glazer, H. Grimmer, B. Gruber, T. Hahn, H. Klapper, E. Koch, P. Konstantinov, V. Kopský, D.B. Litvin, A. Looijenga-Vos, K. Momma, U. Müller, U. Shmueli, B. Souvignier, J.C.H. Spence, P.M. de Wolff, H. Wondratschek, H. Zimmermann, International Tables for Crystallography: Space-group symmetry, Int. Union Crystallogr. A (2016).

[60] J.W. Mullin, Crystallization, 4th ed., Butterworth-Heinemann, Oxford, 2001.

[61] J.D. Wright, Molecular Crystals, 2nd ed., Cambridge University Press, Cambridge, 1995.

[62] J.N. Israelachvili, Intermolecular and Surface Forces, 3rd ed., Academic Press, Online, 2011. https://www.sciencedirect.com/book/9780123751829/intermolecular-and-surface-forces#book-description.

[63] A. Shahi, E. Arunan, Why are Hydrogen Bonds Directional?, J. Chem. Sci. 128 (2016) 1571–1577. https://doi.org/10.1007/s12039-016-1156-3.

[64] P.A. Wood, T.S.G. Olsson, J.C. Cole, S.J. Cottrell, N. Feeder, P.T.A. Galek, C.R. Groom, E. Pidcock, Evaluation of molecular crystal structures using Full interaction maps, CrystEngComm. 15 (2013) 65–72.

[65] B.A. Kolesov, M.A. Mikhailenko, E. V. Boldyreva, Dynamics of the intermolecular hydrogen bonds in the polymorphs of paracetamol in relation to crystal packing and conformational transitions: A variable-temperature polarized Raman spectroscopy study, Phys. Chem. Chem. Phys. 13 (2011) 14243–14253. https://doi.org/10.1039/c1cp20139e.

[66] C. Hammond, Introduction to crystallography, revised, Oxford University

Press, New York, New York, USA, 1992.

[67] A.K. Tiwary, Modification of Crystal Habit and Its Role in Dosage Form Performance, Drug Dev. Ind. Pharm. 27 (2001) 699–709. https://doi.org/10.1081/DDC-100107327.

[68] Z.B. Kuvadia, M.F. Doherty, Effect of Structurally Similar Additives on Crystal Habit of Organic Molecular Crystals at Low Supersaturation, Cryst. Growth Des. 13 (2013) 1412–1428. https://doi.org/10.1021/cg3010618.

[69] N. Pudasaini, C.R. Parker, S.U. Hagen, A.D. Bond, J. Rantanen, Role of Solvent Selection on Crystal Habit of 5-Aminosalicylic Acid—Combined Experimental and Computational Approach, J. Pharm. Sci. 107 (2018) 1112–1121. https://doi.org/10.1016/J.XPHS.2017.12.005.

[70] R.L. Marchese Robinson, K.J. Roberts, E.B. Martin, The influence of solid state information and descriptor selection on statistical models of temperature dependent aqueous solubility, J. Cheminform. 10 (2018) 44. https://doi.org/10.1186/s13321-018-0298-3.

[71] R. Docherty, K. Pencheva, Y.A. Abramov, Low solubility in drug development: de-convoluting the relative importance of solvation and crystal packing., J. Pharm. Pharmacol. 67 (2015) 847–56. https://doi.org/10.1111/jphp.12393.

[72] T.W. Boyle, Glaxo Inc. v. Novopharm Ltd., 931 F. Supp. 1280 (E.D.N.C. 1996), Eastern District of North Carolina, 1996. https://law.justia.com/cases/federal/district-courts/FSupp/931/1280/2346630/ (accessed August 23, 2018).

[73] CCDC, (n.d.). https://www.ccdc.cam.ac.uk/.

[74] C.R. Groom, I.J. Bruno, M.P. Lightfoot, S.C. Ward, The Cambridge Structural Database, Acta Crystallogr. Sect. B Struct. Sci. Cryst. Eng. Mater. 72 (2016) 171–179. https://doi.org/10.1107/S2052520616003954.

[75] M. Ishikawa, Y. Hashimoto, Improving the Water-Solubility of Compounds By Molecular Modification to Disrupt Crystal Packing, in: C.G. Wermuth, D. Aldous, P. Raboisson, D. Rognan (Eds.), Pract. Med. Chem. Fourth Ed., 4th ed., Academic Press, Online, 2015. https://doi.org/10.1016/B978-0-12-417205-0.00031-6.

[76] S.N. Bhattachar, L.A. Deschenes, J.A. Wesley, Solubility: it's not just for physical chemists, Drug Discov. Today. 11 (2006) 1012–1018. https://doi.org/10.1016/J.DRUDIS.2006.09.002.

[77] J.L. McDonagh, N. Nath, L. De Ferrari, T. Van Mourik, J.B.O. Mitchell, Uniting cheminformatics and chemical theory to predict the intrinsic aqueous solubility of crystalline druglike molecules, J. Chem. Inf. Model. 54 (2014) 844–856. https://doi.org/10.1021/ci4005805.

[78] Y. Zhang, N. Wang, L. Zou, M. Zhang, R. Chi, Molecular dynamics simulation on the dissolution process of Kaempferol cluster, J. Mol. Liq. 304 (2020) 112779. https://doi.org/10.1016/j.molliq.2020.112779.

[79] N. Feeder, E. Pidcock, A.M. Reilly, G. Sadiq, C.L. Doherty, K.R. Back, P. Meenan, R. Docherty, The integration of solid-form informatics into solid-form selection, J. Pharm. Pharmacol. (2015). https://doi.org/10.1111/jphp.12394.

[80] S.H. Yalkowsky, Y. He, P. Jain, Handbook of Aqueous Solubility Data, 2nd ed., Taylor and Francis Group, Boca Raton, 2010.

[81] J.L. McDonagh, T. van Mourik, J.B.O. Mitchell, Predicting Melting Points of Organic Molecules: Applications to Aqueous Solubility Prediction Using the General Solubility Equation, Mol. Inform. 34 (2015) 715–724. https://doi.org/10.1002/minf.201500052.

[82] J.C. Bradley, A. Lang, A. Willaims, Open Melting Point Data, Online. (2011). http://lxsrv7.oru.edu/~alang/meltingpoints/download.php.

[83] I. V. Tetko, D. M. Lowe, A.J. Williams, The development of models to predict melting and pyrolysis point data associated with several hundred thousand compounds mined from PATENTS, J. Cheminform. 8 (2016) 2. https://doi.org/10.1186/s13321-016-0113-y.

[84] M. Salahinejad, T.C. Le, D.A. Winkler, Capturing the crystal: Prediction of enthalpy of sublimation, crystal lattice energy, and melting points of organic compounds, J. Chem. Inf. Model. (2013). https://doi.org/10.1021/ci3005012.

[85] M.J. Bryant, S.N. Black, H. Blade, R. Docherty, A.G.P. Maloney, S.C. Taylor, The CSD Drug Subset: The Changing Chemistry and Crystallography of Small Molecule Pharmaceuticals, J. Pharm. Sci. 108 (2019) 1655–1662.

https://doi.org/10.1016/J.XPHS.2018.12.011.

[86] H. Kopp, On the Relation between Boiling-Point and Composition in Organic Compounds, Philos. Trans. R. Soc. London. 150 (1860) 257–276. https://www.jstor.org/stable/108772?seq=1#metadata_info_tab_contents (accessed May 14, 2019).

[87] J.B. Austin, A Relationship between the molecular weights and melting points of organic compounds, J. Am. Chem. Soc. 52 (1930) 1049–1053. https://doi.org/10.1021/ja01366a032.

[88] S.M. Free, J.W. Wilson, A Mathematical Contribution to Structure-Activity Studies, J. Med. Chem. 7 (1964) 395–399. https://doi.org/10.1021/jm00334a001.

[89] R. Todeschini, V. Consonni, Handbook of Molecular Descriptors, Wiley-VCH Verlag GmbH, Weinheim, 2008.

[90] P. Smialowski, D. Frishman, S. Kramer, Pitfalls of supervised feature selection, Bioinformatics. 26 (2010) 440–443. https://doi.org/10.1093/bioinformatics/btp621.

[91] N.M. O'Boyle, D.S. Palmer, F. Nigsch, J.B. Mitchell, Simultaneous feature selection and parameter optimisation using an artificial ant colony: case study of melting point prediction., Chem. Cent. J. 2 (2008) 21. https://doi.org/10.1186/1752-153X-2-21.

[92] H. Moriwaki, Y.-S. Tian, N. Kawashita, T. Takagi, Mordred: a molecular descriptor calculator, J. Cheminform. 10 (2018) 4. https://doi.org/10.1186/s13321-018-0258-y.

[93] H. Hong, Q. Xie, W. Ge, F. Qian, H. Fang, L. Shi, Z. Su, R. Perkins, W. Tong, Mold2 , Molecular Descriptors from 2D Structures for Chemoinformatics and Toxicoinformatics, J. Chem. Inf. Model. 48 (2008) 1337–1344. https://doi.org/10.1021/ci800038f.

[94] Y.A. Abramov, Major Source of Error in QSPR Prediction of Intrinsic Thermodynamic Solubility of Drugs: Solid vs Nonsolid State Contributions?, Mol. Pharm. 12 (2015) 2126–2141. https://doi.org/10.1021/acs.molpharmaceut.5b00119.

[95] S. Emami, A. Jouyban, H. Valizadeh, A. Shayanfar, Are Crystallinity Parameters Critical for Drug Solubility Prediction?, J. Solution Chem. 44 (2015) 2297–2315. https://doi.org/10.1007/s10953-015-0410-5.

[96] O. Isayev, D. Fourches, E.N. Muratov, C. Oses, K. Rasch, A. Tropsha, S. Curtarolo, Materials Cartography: Representing and Mining Materials Space Using Structural and Electronic Fingerprints, Chem. Mater. 27 (2015) 735–743. https://doi.org/10.1021/cm503507h.

[97] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, R.P. Adams, Convolutional Networks on Graphs for Learning Molecular Fingerprints, Comput. Res. Repos. abs/1509.09292 (2015). http://arxiv.org/abs/1509.09292 (accessed July 19, 2018).

[98] T. Cheng, Q. Li, Y. Wang, S.H. Bryant, Binary Classification of Aqueous Solubility Using Support Vector Machines with Reduction and Recombination Feature Selection, J. Chem. Inf. Model. 51 (2011) 229–236. https://doi.org/10.1021/ci100364a.

[99] M. Yang, B. Tao, C. Chen, W. Jia, S. Sun, T. Zhang, X. Wang, Machine Learning Models Based on Molecular Fingerprints and an Extreme Gradient Boosting Method Lead to the Discovery of JAK2 Inhibitors, J. Chem. Inf. Model. 59 (2019) 5002–5012. https://doi.org/10.1021/acs.jcim.9b00798.

[100] H. Cai, V.W. Zheng, K.C.-C. Chang, A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications, IEEE Trans. Knowl. Data Eng. 30 (2018) 1616–1637. https://doi.org/10.1109/TKDE.2018.2807452.

[101] P. Zhang, J. Yellen, J.L. Gross, Handbook of graph theory, 2nd ed., Taylor & Francis Group, New York, New York, USA, 2015.

[102] J. Gilmer, S.S. Schoenholz, P.F. Riley, O. Vinyals, G.E. Dahl, Neural Message Passing for Quantum Chemistry, Comput. Res. Repos. abs/1704.0 (2017). http://arxiv.org/abs/1704.01212 (accessed July 20, 2018).

[103] T. Pham, T. Tran, S. Venkatesh, Graph Memory Networks for Molecular Activity Prediction, Comput. Res. Repos. abs/1801.02622 (2018). http://arxiv.org/abs/1801.02622 (accessed July 19, 2018).

[104] M. Gori, G. Monfardini, F. Scarselli, A new model for learning in graph

domains, in: Int. Jt. Conf. Neural Networks., IEEE, 2005: pp. 729–734. https://doi.org/10.1109/IJCNN.2005.1555942.

[105] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated Graph Sequence Neural Networks, Comput. Res. Repos. abs/1511.0 (2015). http://arxiv.org/abs/1511.05493 (accessed July 20, 2018).

[106] S. Kearnes, K. McCloskey, M. Berndl, V. Pande, P. Riley, Molecular graph convolutions: moving beyond fingerprints, J. Comput. Aided. Mol. Des. 30 (2016) 595–608. https://doi.org/10.1007/s10822-016-9938-8.

[107] P.W. Battaglia, R. Pascanu, M. Lai, D. Rezende, K. Kavukcuoglu, Interaction Networks for Learning about Objects, Relations and Physics, Artif. Intell. (2016). http://arxiv.org/abs/1612.00222 (accessed July 1, 2019).

[108] A. Dey, Machine Learning Algorithms: A Review, Int. J. Comput. Sci. Inf. Technol. 7 (2016) 1174–1179.

[109] Y. Li, Deep Reinforcement Learning, Eprint ArXiv:1810.06339. (2018). http://arxiv.org/abs/1810.06339 (accessed May 31, 2019).

[110] M. Khanum, T. Mahboob, W. Imtiaz, H.A. Ghafoor, R. Sehar, A Survey on Unsupervised Machine Learning Algorithms for Automation, Classification and Maintenance, Int. J. Comput. Appl. 113 (2015) 34–39. https://www.semanticscholar.org/paper/A-Survey-on-Unsupervised-Machine-Learning-for-and-Khanum-Mahboob/0c63ae912aa3264013b70c15d1c0c040d27219f7 (accessed May 11, 2019).

[111] G.C. Cawley, N.L.C. Talbot, On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation, J. Mach. Learn. Res. 11 (2010) 2079–2107. http://jmlr.org/papers/v11/cawley10a.html (accessed September 25, 2020).

[112] D. Baumann, K. Baumann, Reliable estimation of prediction errors for QSAR models under model uncertainty using double cross-validation, J. Cheminform. 6 (2014) 47. https://doi.org/10.1186/s13321-014-0047-1.

[113] J. Bergstra, Y. Bengio, Random Search for Hyper-Parameter Optimization, J. Mach. Learn. Res. 13 (2012) 281–305. http://scikit-learn.sourceforge.net.

(accessed April 15, 2019).

[114] N. Srivastava, G. Hinton, A. Krizhevsky, R. Salakhutdinov, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, J. Mach. Learn. Res. 15 (2014) 1929–1958. http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf (accessed June 4, 2019).

[115] P.J. Huber, Robust Estimation of a Location Parameter, Ann. Math. Stat. 35 (1964) 73–101. https://doi.org/10.1214/aoms/1177703732.

[116] S. Jadon, A survey of loss functions for semantic segmentation, Image Video Process. (2020). http://arxiv.org/abs/2006.14822 (accessed September 25, 2020).

[117] D.L.J. Alexander, A. Tropsha, D.A. Winkler, Beware of R2: Simple, Unambiguous Assessment of the Prediction Accuracy of QSAR and QSPR Models, J. Chem. Inf. Model. 55 (2015) 1316–1322. https://doi.org/10.1021/acs.jcim.5b00206.

[118] L. Rosasco, E. De Vito, A. Caponnetto, M. Piana, A. Verri, Are Loss Functions All the Same?, Neural Comput. 16 (2004) 1063–1076. https://doi.org/10.1162/089976604773135104.

[119] N. Cristianini, J. Shawe-Taylor, An Introduction to Support Vector Machines and Other Kernel-based Learning Methods, Cambridge University Press, 2000. https://doi.org/10.1017/cbo9780511801389.

[120] L.E. Raileanu, K. Stoffel, Theoretical Comparison between the Gini Index and Information Gain Criteria, Ann. Math. Artif. Intell. 41 (2004) 77–93. https://doi.org/10.1023/B:AMAI.0000018580.96245.c6.

[121] N. Chinchor, MUC-4 evaluation metrics, in: Proc. 4th Conf. Messag. Underst. - MUC4 ’92, Association for Computational Linguistics (ACL), Morristown, NJ, USA, 1992: p. 22. https://doi.org/10.3115/1072064.1072067.

[122] L. Breiman, Random Forest, Mach. Learn. 45 (2001) 5–32. https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf (accessed July 7, 2019).

[123] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M.

Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, É. Duchesnay, Scikit-learn: Machine Learning in Python, J. Mach. Learn. Res. 12 (2011) 2825–2830. http://jmlr.org/papers/v12/pedregosa11a.html (accessed August 24, 2018).

[124] B.H. Menze, B.M. Kelm, R. Masuch, U. Himmelreich, P. Bachert, W. Petrich, F.A. Hamprecht, A comparison of random forest and its Gini importance with standard chemometric methods for the feature selection and classification of spectral data, BMC Bioinformatics. 10 (2009) 213. https://doi.org/10.1186/1471-2105-10-213.

[125] W.S. Noble, What is a support vector machine?, Nat. Biotechnol. 24 (2006) 1565–1567. https://doi.org/10.1038/nbt1206-1565.

[126] C.J.C. Burges, A Tutorial on Support Vector Machines for Pattern Recognition, Data Min. Knowl. Discov. 2 (1998) 121–167. https://www.microsoft.com/en-us/research/publication/a-tutorial-on-support-vector-machines-for-pattern-recognition/ (accessed September 26, 2020).

[127] W.S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, Bull. Math. Biophys. 5 (1943) 115–133. https://doi.org/10.1007/BF02478259.

[128] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature. 521 (2015) 436–444. https://doi.org/10.1038/nature14539.

[129] S. Hochreiter, J.J. Urgen Schmidhuber, Long short -term memory, Neural Comput. 9 (1997) 1735–1780. http://www7.informatik.tu-muenchen.de/~hochreithttp://www.idsia.ch/~juergen (accessed July 7, 2019).

[130] M. Hagan, H.B. Demuth, M.H. Beale, O. De Jesus, Neural Network Design, 2nd ed., PWS Publishing Co, Boston, 2016. https://hagan.okstate.edu/nnd.html (accessed September 25, 2020).

[131] F. Siddique, S. Sakib, M.A.B. Siddique, Recognition of handwritten digit using convolutional neural network in python with tensorflow and comparison of performance for various hidden layers, in: 2019 5th Int. Conf. Adv. Electr. Eng. ICAEE 2019, Institute of Electrical and Electronics Engineers Inc., 2019: pp. 541–546. https://doi.org/10.1109/ICAEE48663.2019.8975496.

[132] S. Ruder, An overview of gradient descent optimization algorithms, Mach. Learn. (2016). http://arxiv.org/abs/1609.04747 (accessed May 2, 2019).

[133] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: 13th Int. Conf. Artif. Intell. Stat., PMLR, 2010: pp. 249–256. http://proceedings.mlr.press/v9/glorot10a.html (accessed May 11, 2019).

[134] Y.A. LeCun, L. Bottou, G.B. Orr, K.-R. Müller, Efficient BackProp, in: Montavon G, Orr G B, Müller KR (Eds.), Neural Networks: Tricks of the Trade, 2nd ed., Springer, Berlin, 2012: pp. 9–48. https://doi.org/10.1007/978-3-642-35289-8_3.

[135] B. Hanin, D. Rolnick, How to Start Training: The Effect of Initialization and Architecture, Mach. Learing. (2018). http://arxiv.org/abs/1803.01719 (accessed May 12, 2019).

[136] A.M. Saxe, J.L. McClelland, S. Ganguli, Exact solutions to the nonlinear dynamics of learning in deep linear neural networks, Neural Evol. Comput. (2013). http://arxiv.org/abs/1312.6120 (accessed May 12, 2019).

[137] D.P. Kingma, J. Ba, Adam: A Method for Stochastic Optimization, in: 3rd Int. Conf. Learn. Represent., San Diego, 2015. http://arxiv.org/abs/1412.6980 (accessed May 2, 2019).

[138] G. Huang, G.-B. Huang, S. Song, K. You, Trends in extreme learning machines: A review, Neural Networks. 61 (2015) 32–48. https://doi.org/10.1016/j.neunet.2014.10.001.

[139] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies, in: J.F. Kolen, S.C. Kremer (Eds.), A F. Guid. to Dyn. Recurr. Networks, IEEE, 2009. https://doi.org/10.1109/9780470544037.ch14.

[140] Duchi John, E. Hazan, Y. Singer, Adaptive Subgradient Methods for Online Learning and Stochastic Optimization * Elad Hazan, J. Mach. Learn. Res. 12 (2011) 2121–2159.

[141] M. Claesen, B. De Moor, Hyperparameter Search in Machine Learning, XI Metaheuristics Int. Conf. (2015). http://arxiv.org/abs/1502.02127 (accessed April 15, 2019).

[142] O. Obrezanova, G. Csányi, J.M.R. Gola, M.D. Segall, Gaussian processes: A method for automatic QSAR modeling of ADME properties, J. Chem. Inf. Model. 47 (2007) 1847–1857. https://doi.org/10.1021/ci7000633.

[143] S. Ding, H. Li, C. Su, J. Yu, F. Jin, Evolutionary artificial neural networks: a review, Artif. Intell. Rev. 39 (2013) 251–260. https://doi.org/10.1007/s10462-011-9270-6.

[144] J.T. Tsai, J.H. Chou, T.K. Liu, Tuning the Structure and Parameters of a Neural Network by Using Hybrid Taguchi-Genetic Algorithm, IEEE Trans. Neural Networks. 17 (2006) 69–80. https://doi.org/10.1109/TNN.2005.860885.

[145] J. Snoek, H. Larochelle, R.P. Adams, Practical Bayesian Optimization of Machine Learning Algorithms, ArXiv:1206.2944v2 . (2012). http://arxiv.org/abs/1206.2944 (accessed April 15, 2019).

[146] J. Bergstra, D. Yamins, D.D. Cox, Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms, in: PROC. 12th PYTHON Sci. CONF, 2013: pp. 13–20. http://www.youtube.com/watch?v=Mp1xnPfE4PY! (accessed April 15, 2019).

[147] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for Hyper-Parameter Optimization, Adv. Neural Inf. Process. Syst. (2011). https://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf (accessed April 15, 2019).

[148] D.R. Jones, A Taxonomy of Global Optimization Methods Based on Response Surfaces, J. Glob. Optim. 21 (2001) 345–383. https://doi.org/10.1023/A:1012771025575.

[149] J. Villemonteix, E. Vazquez, E. Walter, An informational approach to the global optimization of expensive-to-evaluate functions, J. Glob. Optim. 44 (2009) 509–534. https://doi.org/10.1007/s10898-008-9354-2.

[150] F. Hutter, H.H. Hoos, K. Leyton-Brown, Sequential Model-Based Optimization for General Algorithm Configuration, Learn. Intell. Optim. 6683 (2011) 507–523. https://doi.org/10.1007/978-3-642-25566-3_40.

[151] J. Francisco, M. Diaz, C. Maurice, F. Lerasle, F.L. Hyper, F. Madrigal, F. Lerasle, F. Madrigal, C. Maurice, F. Lerasle, Hyper-parameter optimization

tools comparison for Multiple Object Tracking applications, Mach. Vis. Appl. 30 (2018) 269–289. https://doi.org/10.1007/s00138-018-0984-1ï.

[152] P.D. Tsakanikas, S.H. Yalkowsky, Estimation of melting point of flexible molecules: Aliphatic hydrocarbons, Toxicol. Environ. Chem. 17 (1988) 19–33. https://doi.org/10.1080/02772248809357275.

[153] T. Hanser, C. Barber, J.F. Marchaland, S. Werner, Applicability domain: towards a more formal definition, SAR QSAR Environ. Res. 27 (2016) 893–909. https://doi.org/10.1080/1062936X.2016.1250229.

[154] F. Svensson, N. Aniceto, U. Norinder, I. Cortes-Ciriano, O. Spjuth, L. Carlsson, A. Bender, Conformal Regression for Quantitative Structure-Activity Relationship Modeling - Quantifying Prediction Uncertainty, J. Chem. Inf. Model. 58 (2018) 1132–1140. https://doi.org/10.1021/acs.jcim.8b00054.

[155] H. Kopp, On a great regularity in the physical properties of analogous organic compounds, Philos. Mag. 130 (1842) 187–197.

[156] L.P. Hammett, The Effect of Structure upon the Reactions of Organic Compounds. Benzene Derivatives, J. Am. Chem. Soc. 59 (1937) 96–103. https://doi.org/10.1021/ja01280a022.

[157] J. Taskinen, J. Yliruusi, Prediction of physicochemical properties based on neural network modelling, Adv. Drug Deliv. Rev. 55 (2003) 1163–1183. https://doi.org/10.1016/S0169-409X(03)00117-0.

[158] A.R. Katrizky, M. Kuanar, S. Slavov, C.D. Hall, M. Karelson, I. Kahn, D.A. Dobchev, Quantitative correlation of physical and chemical properties with chemical structure- Utility for prediction, Chem. Rev. 110 (2010) 5714–5789.

[159] W.L. Jorgensen, E.M. Duffy, Prediction of drug solubility from structure, Adv. Drug Deliv. Rev. 54 (2002) 355–366. https://doi.org/10.1016/S0169-409X(02)00008-X.

[160] S.R. Johnson, W. Zheng, Recent progress in the computational prediction of aqueous solubility and absorption, AAPS J. 8 (2006) E27–E40.

[161] B.F. Begam, J.S. Kumar, A Study on Cheminformatics and its Applications on Modern Drug Discovery, Procedia Eng. 38 (2012) 1264–1275. https://doi.org/10.1016/J.PROENG.2012.06.156.

[162] C. Hansch, P.P. Maloney, T. Fujita, R.M. Muir, Correlation of biological activity of phenoxyacetic acids with Hammett substituent constants and partition coefficients, Nature. 194 (1962) 178–180. https://doi.org/10.1038/194178b0.

[163] C. Hansch, T. Fujita, ρ-σ-π Analysis. A Method for the Correlation of Biological Activity and Chemical Structure, J. Am. Chem. Soc. 86 (1964) 1616–1626. https://doi.org/10.1021/ja01062a035.

[164] Y.C. Martin, Hansch analysis 50 years on, Wiley Interdiscip. Rev. Comput. Mol. Sci. 2 (2012) 435–442. https://doi.org/10.1002/wcms.1096.

[165] C. Hansch, R.M. Muir, T. Fujita, P.P. Maloney, F. Geiger, M. Streich, The Correlation of Biological Activity of Plant Growth Regulators and Chloromycetin Derivatives with Hammett Constants and Partition Coefficients, J. Am. Chem. Soc. 85 (1963) 2817–2824. https://doi.org/10.1021/ja00901a033.

[166] H. Briem, J. Günther, Classifying "Kinase Inhibitor-Likeness" by Using Machine-Learning Methods, ChemBioChem. 6 (2005) 558–566. https://doi.org/10.1002/cbic.200400109.

[167] S.-S. So, M. Karplus, Three-Dimensional Quantitative Structure−Activity Relationships from Molecular Similarity Matrices and Genetic Neural Networks. 1. Method and Validations, J. Med. Chem. 40 (1997) 4347–4359. https://doi.org/10.1021/JM970487V.

[168] D.-S. Cao, Q.-S. Xu, Y.-Z. Liang, X. Chen, H.-D. Li, Prediction of aqueous solubility of druglike organic compounds using partial least squares, back-propagation network and support vector machine, J. Chemom. 24 (2010) n/a-n/a. https://doi.org/10.1002/cem.1321.

[169] S. Enami, A. Jouyban, H. Valizadeh, A. Shayanfar, Are Crystallinity Parameters Critical for Drug Solubility Prediction, J. Solution Chem. 44 (2015) 2297–2315. http://link.springer.com/article/10.1007%2Fs10953-015-0410-5.

[170] I. V. Tetko, Y. Sushko, S. Novotarskyi, L. Patiny, I. Kondratov, A.E. Petrenko, L. Charochkina, A.M. Asiri, How Accurately Can We Predict the Melting Points of Drug-like Compounds?, J. Chem. Inf. Model. 54 (2014) 3320–3329. https://doi.org/10.1021/ci5005288.

[171] N. Jain, S.H. Yalkowsky, UPPER III: Unified physical property estimation relationships. Application to non-hydrogen bonding aromatic compounds, J. Pharm. Sci. 88 (1999) 852–860. https://doi.org/10.1021/JS990117P.

[172] M. Karthikeyan, R.C.G. And, A. Bender*, General Melting Point Prediction Based on a Diverse Compound Data Set and Artificial Neural Networks, J. Chem. Inf. Model. 45 (2005) 581–59. https://doi.org/10.1021/CI0500132.

[173] A.U. Bhat, S.S. Merchant, S.S. Bhagwat, Prediction of Melting Points of Organic Compounds Using Extreme Learning Machines, Ind. Eng. Chem. Res. 47 (2008) 920–925. https://doi.org/10.1021/IE0704647.

[174] C.A.S. Bergström, U. Norinder, K. Luthman, P. Artursson, Molecular Descriptors Influencing Melting Point and Their Role in Classification of Solid Drugs, J. Chem. Inf. Comput. Sci. 43 (2003) 1177–1185. https://doi.org/10.1021/CI020280X.

[175] L. Zhao, S.H. Yalkowsky, A Combined Group Contribution and Molecular Geometry Approach for Predicting Melting Points of Aliphatic Compounds, Ind. Eng. Chem. Res. 33 (1999) 1405–1409. https://doi.org/10.1021/IE990281N.

[176] M. Zhou, N. Duan, S. Liu, H.Y. Shum, Progress in Neural NLP: Modeling, Learning, and Reasoning, Engineering. 6 (2020) 275–290. https://doi.org/10.1016/j.eng.2019.12.014.

[177] N.C. Thompson, K. Greenewald, K. Lee, G.F. Manso, The Computational Limits of Deep Learning, Machine Learn. (2020). http://arxiv.org/abs/2007.05558 (accessed September 26, 2020).

[178] M. Salahinejad, T.C. Le, D.A. Winkler, Aqueous Solubility Prediction- Do Crystal Lattice Interactions Help, Mol. Pharm. 10 (2013) 2757–2766.

[179] C. Tyrchan, E. Evertsson, Matched Molecular Pair Analysis in Short: Algorithms, Applications and Limitations, Comput. Struct. Biotechnol. J. 15 (2017) 86–90. https://doi.org/10.1016/j.csbj.2016.12.003.

[180] J. Hussain, C. Rea, Computationally Efficient Algorithm to Identify Matched Molecular Pairs (MMPs) in Large Data Sets, J. Chem. Inf. Model. 50 (2010) 339–348. http://pubs.acs.org/doi/abs/10.1021/ci900450m (accessed July 8,

2016).

[181] R.P. Sheridan, The Most Common Chemical Replacements in Drug-Like Compounds, J. Chem. Inf. Comput. Sci. 42 (2001) 103–108. https://doi.org/10.1021/ci0100806.

[182] D.J. Warner, E.J. Griffen, S.A. St-Gallay, WizePairZ: A Novel Algorithm to Identify, Encode, and Exploit Matched Molecular Pairs with Unspecified Cores in Medicinal Chemistry, J. Chem. Inf. Model. 50 (2010) 1350–1357. http://pubs.acs.org/doi/abs/10.1021/ci100084s (accessed July 8, 2016).

[183] J. Weber, J. Achenbach, D. Moser, E. Proschak, VAMMPIRE: A Matched Molecular Pairs Database for Structure-Based Drug Design and Optimization, J. Med. Chem. 56 (2013) 5203–5207. http://pubs.acs.org/doi/abs/10.1021/jm400223y (accessed July 8, 2016).

[184] J. Weber, J. Achenbach, D. Moser, E. Proschak, VAMMPIRE-LORD: A Web Server for Straightforward Lead Optimization Using Matched Molecular Pairs, J. Chem. Inf. Model. 55 (2015) 207–213. http://pubs.acs.org/doi/abs/10.1021/ci5005256 (accessed July 8, 2016).

[185] C. Kramer, J.E. Fuchs, S. Whitebread, P. Gedeck, K.R. Liedl, Matched Molecular Pair Analysis- Significance and the Impact of Experimental Uncertainty, J. Med. Chem. 57 (2014) 3786–3802.

[186] A.G. Leach, H.D. Jones, D.A. Cosgrove, P.W. Kenny, L. Ruston, P. MacFaul, J.M. Wood, N. Colclough, B. Law, Matched molecular pairs as a guide in the optimization of pharmaceutical properties; a study of aqueous solubility, plasma protein binding and oral exposure, J. Med. Chem. (2006). https://doi.org/10.1021/jm0605233.

[187] S.-Y. Chen, Z. Feng, X. Yi, A general introduction to adjustment for multiple comparisons., J. Thorac. Dis. 9 (2017) 1725–1729. https://doi.org/10.21037/jtd.2017.05.34.

[188] P. Ranganathan, C.S. Pramesh, M. Buyse, Common pitfalls in statistical analysis: The perils of multiple testing., Perspect. Clin. Res. 7 (2016) 106–7. https://doi.org/10.4103/2229-3485.179436.

[189] I. Giangreco, J.C. Cole, E. Thomas, Mining the Cambridge Structural Database

for Matched Molecular Crystal Structures: A Systematic Exploration of Isostructurality, Cryst. Growth Des. 17 (2017) 3192–3203. https://doi.org/10.1021/acs.cgd.7b00155.

[190] G. Papadatos, M. Alkarouri, V.J. Gillet, P. Willett, V. Kadirkamanathan, C.N. Luscombe, G. Bravi, N.J. Richmond, S.D. Pickett, J. Hussain, J.M. Pritchard, A.W.J. Cooper, S.J.F. Macdonald, Lead Optimization Using Matched Molecular Pairs: Inclusion of Contextual Information for Enhanced Prediction of hERG Inhibition, Solubility, and Lipophilicity, J. Chem. Inf. Model. 50 (2010) 1872–1886. https://doi.org/10.1021/ci100258p.

[191] L. Zhang, H. Zhu, A. Mathiowetz, H. Gao, Deep understanding of structure–solubility relationship for a diverse set of organic compounds using matched molecular pairs, Bioorg. Med. Chem. 19 (2011) 5763–5770. https://doi.org/10.1016/j.bmc.2011.08.036.

[192] S. Schultes, C. de Graaf, H. Berger, M. Mayer, A. Steffen, E.E.J. Haaksma, I.J.P. de Esch, R. Leurs, O. Kramer, A medicinal chemistry perspective on melting point- matched molecular pair analysis of the effects of simple descriptors on the melting point of drug-like compounds, Med. Chem. Commun. 3 (2012) 584–591. https://doi.org/10.1039/c2md00313a.

[193] T. Geppert, B. Beck, Fuzzy Matched Pairs: A Means To Determine the Pharmacophore Impact on Molecular Interaction, J. Chem. Inf. Model. 54 (2014) 1093–1102. http://pubs.acs.org/doi/abs/10.1021/ci400694q (accessed July 8, 2016).

[194] A.M. Wassermann, J. Bajorath, A Data Mining Method To Facilitate SAR Transfer, J. Chem. Inf. Model. 51 (2011) 1857–1866. https://doi.org/10.1021/ci200254k.

[195] D. Dimova, Y. Hu, J. Bajorath, Matched Molecular Pair Analysis of Small Molecule Microarray Data Identifies Promiscuity Cliffs and Reveals Molecular Origins of Extreme Compound Promiscuity, J. Med. Chem. 55 (2012) 10220–10228.

[196] A.M. Wassermann, D. Dimova, P. Iyer, J. Bajorath, Advances in computational medicinal chemistry: Matched molecular pair analysis, Drug Dev. Res. 73 (2012) 518–527. https://doi.org/10.1002/ddr.21045.

[197] J.M. Beck, C. Springer, Quantitative Structure–Activity Relationship Models of Chemical Transformations from Matched Pairs Analyses, J. Chem. Inf. Model. 54 (2014) 1226–1234. https://doi.org/10.1021/ci500012n.

[198] Y. Sushko, S. Novotarskyi, R. Körner, J. Vogt, A. Abdelaziz, I. V Tetko, Prediction-driven matched molecular pairs to interpret QSARs and aid the molecular optimization process, J. Cheminform. 6 (2014) 48. https://doi.org/10.1186/s13321-014-0048-0.

[199] A. de la V. de León, J. Bajorath, Compound Optimization through Data Set-Dependent Chemical Transformations, J. Chem. Inf. Model. 53 (2013) 1263–1271. http://pubs.acs.org/doi/abs/10.1021/ci400165a (accessed July 8, 2016).

[200] J. Janowiak, E.B. Martin, K.J. Roberts, Marchese Robinson, Richard L, Maloney, Andrew, Giangreco, Ilenia, K. Pencheva, Adaptation of a Matched Molecular Pair Identification Algorithm for Solid State Informatics Analysis of the Cambridge Structural Database, in: UK QSAR, University of Cardiff, 2018.

[201] A. Dalke, J. Hert, C. Kramer, Mmpdb: An Open-Source Matched Molecular Pair Platform for Large Multiproperty Data Sets, J. Chem. Inf. Model. 58 (2018) 902–910. https://doi.org/10.1021/acs.jcim.8b00173.

[202] Open-source cheminformatics, RDKit, (n.d.). http://www.rdkit.org.

[203] G.R. Desiraju, Supramolecular Synthons in Crystal Engineering—A New Organic Synthesis, Angew. Chemie Int. Ed. English. 34 (1995) 2311–2327. https://doi.org/10.1002/anie.199523111.

[204] G.M. Day, A. V. Trask, W.D.S. Motherwell, W. Jones, Investigating the latent polymorphism of maleic acid, Chem. Commun. (2006) 54–56. https://doi.org/10.1039/B513442K.

[205] R.S. Payne, R.J. Roberts, R.C. Rowe, R. Docherty, Examples of successful crystal structure prediction: Polymorphs of primidone and progesterone, Int. J. Pharm. 177 (1999) 231–245. https://doi.org/10.1016/S0378-5173(98)00348-2.

[206] J. Kendrick, G.A. Stephenson, M.A. Neumann, F.J.J. Leusen, Crystal structure prediction of a flexible molecule of pharmaceutical interest with unusual polymorphic behavior, Cryst. Growth Des. 13 (2013) 581–589. https://doi.org/10.1021/cg301222m.

[207] L. Iuzzolino, P. McCabe, S.L. Price, J.G. Brandenburg, Crystal structure prediction of flexible pharmaceutical-like molecules: density functional tight-binding as an intermediate optimisation method and for free energy estimation, Faraday Discuss. 211 (2018) 275–296. https://doi.org/10.1039/c8fd00010g.

[208] A.M. Reilly, R.I. Cooper, C.S. Adjiman, S. Bhattacharya, A.D. Boese, J.G. Brandenburg, P.J. Bygrave, R. Bylsma, J.E. Campbell, R. Car, D.H. Case, R. Chadha, J.C. Cole, K. Cosburn, H.M. Cuppen, F. Curtis, G.M. Day, R.A. DiStasio, A. Dzyabchenko, B.P. Van Eijck, D.M. Elking, J.A. Van Den Ende, J.C. Facelli, M.B. Ferraro, L. Fusti-Molnar, C.A. Gatsiou, T.S. Gee, R. De Gelder, L.M. Ghiringhelli, H. Goto, S. Grimme, R. Guo, D.W.M. Hofmann, J. Hoja, R.K. Hylton, L. Iuzzolino, W. Jankiewicz, D.T. De Jong, J. Kendrick, N.J.J. De Klerk, H.Y. Ko, L.N. Kuleshova, X. Li, S. Lohani, F.J.J. Leusen, A.M. Lund, J. Lv, Y. Ma, N. Marom, A.E. Masunov, P. McCabe, D.P. McMahon, H. Meekes, M.P. Metz, A.J. Misquitta, S. Mohamed, B. Monserrat, R.J. Needs, M.A. Neumann, J. Nyman, S. Obata, H. Oberhofer, A.R. Oganov, A.M. Orendt, G.I. Pagola, C.C. Pantelides, C.J. Pickard, R. Podeszwa, L.S. Price, S.L. Price, A. Pulido, M.G. Read, K. Reuter, E. Schneider, C. Schober, G.P. Shields, P. Singh, I.J. Sugden, K. Szalewicz, C.R. Taylor, A. Tkatchenko, M.E. Tuckerman, F. Vacarro, M. Vasileiadis, A. Vazquez-Mayagoitia, L. Vogt, Y. Wang, R.E. Watson, G.A. De Wijs, J. Yang, Q. Zhu, C.R. Groom, Report on the sixth blind test of organic crystal structure prediction methods, Acta Crystallogr. Sect. B Struct. Sci. Cryst. Eng. Mater. 72 (2016) 439–459. https://doi.org/10.1107/S2052520616007447.

[209] K. Kersten, R. Kaur, A. Matzger, Survey and analysis of crystal polymorphism in organic structures, IUCrJ Chem. Cryst. Eng. 5 (2018) 124–129. https://doi.org/10.1107/S2052252518000660.

[210] P. Crafts, The Role of Solubility Modeling and Crystallization in the Design of Active Pharmaceutical Ingredients, in: Ka M. Ng, Rafiqul Gani, Kim Dam-Johansen (Eds.), Comput. Aided Chem. Eng., Elsevier, 2007: pp. 23–85. https://doi.org/10.1016/S1570-7946(07)80005-8.

[211] J. van de Streek, Searching the Cambridge Structural Database for the `best' representative of each unique polymorph, Acta Crystallogr. Sect. B Struct. Sci.

62 (2006) 567–579. https://doi.org/10.1107/S0108768106019677.

[212] J.G.P. Wicker, R.I. Cooper, Beyond Rotatable Bond Counts: Capturing 3D Conformational Flexibility in a Single Descriptor, J. Chem. Inf. Model. 56 (2016) 2347–2352. https://doi.org/10.1021/acs.jcim.6b00565.

[213] P. Tosco, N. Stiefl, G. Landrum, Bringing the MMFF force field to the RDKit: implementation and validation, J. Cheminform. 6 (2014) 37. https://doi.org/10.1186/s13321-014-0037-3.

[214] Y.L. Slovokhotov, I.S. Neretin, J.A.K. Howard, Symmetry of van der Waals molecular shape and melting points of organic compounds, New J. Chem. 28 (2004) 967–979. https://doi.org/10.1039/b310787f.

[215] CCDC, CSD Python API, (n.d.). https://downloads.ccdc.cam.ac.uk/documentation/API/.

[216] W. McKinney, Data Structures for Statistical Computing in Python, in: S. van der Walt, J. Millman (Eds.), Proc. 9th Python Sci. Conf., 2010: pp. 51–56. http://conference.scipy.org/proceedings/scipy2010/mckinney.html (accessed August 24, 2018).

[217] W. Michael, B. Olga, O. Drew, P. Hobson, J. Ostblom, L. Saulius, G. David C, T. Augspurger, H. Yaroslav, C. John B., W. Jordi, de R. Julian, P. Cameron, H. Stephan, V. Jake, V. Santi, Seaborn, (2018). https://doi.org/10.5281/zenodo.1313201.

[218] J.D. Hunter, Matplotlib: A 2D Graphics Environment, Comput. Sci. Eng. 9 (2007) 90–95. doi:10.1109/MCSE.2007.55.

[219] A. Farcomeni, A review of modern multiple hypothesis testing, with particular attention to the false discovery proportion, Stat. Methods Med. Res. 17 (2008) 347–388. https://doi.org/10.1177/0962280206079046.

[220] W.C. McCrone, Physics and Chemistry of the Organic Solid State, A. WeissbergerInterscience Publishers, New York, 1965.

[221] M.R. Abu Bakar, Z.K. Nagy, C.D. Rielly, S.E. Dann, Investigation of the riddle of sulfathiazole polymorphism, Int. J. Pharm. 414 (2011) 86–103. https://doi.org/10.1016/J.IJPHARM.2011.05.004.

[222] J. van de Streek, S. Motherwell, Searching the Cambridge Structural Database for polymorphs, Acta Crystallogr. Sect. B Struct. Sci. 61 (2005) 504–510. https://doi.org/10.1107/S0108768105020021.

[223] A.A. Moldovan, I. Rosbottom, V. Ramachandran, C.M. Pask, O. Olomukhoro, K.J. Roberts, Crystallographic Structure, Intermolecular Packing Energetics, Crystal Morphology and Surface Chemistry of Salmeterol Xinafoate (Form I), J. Pharm. Sci. 106 (2017) 882–891. https://doi.org/10.1016/J.XPHS.2016.11.016.

[224] J.A. Chisholm, S. Motherwell, COMPACK: a program for identifying crystal structure similarity using distances, J. Appl. Crystallogr. 38 (2005) 228–231. https://doi.org/10.1107/S0021889804027074.

[225] S. Boslaugh, Statistics in a Nutshell, 2nd ed., O'Reilly, online, 2012. https://www.oreilly.com/library/view/statistics-in-a/9781449361129/ (accessed September 13, 2020).

[226] J.E. Jackson, A Use's Guide to Principal Components, John Wiley & Sons, Inc., Hoboken, NJ, USA, 1991. https://doi.org/10.1002/0471725331.

[227] X. Hu, Y. Hu, M. Vogt, D. Stumpfe, J. Bajorath, MMP-Cliffs: Systematic Identification of Activity Cliffs on the Basis of Matched Molecular Pairs, J. Chem. Inf. Model. 52 (2012) 1138–1145. http://pubs.acs.org/doi/abs/10.1021/ci3001138 (accessed July 8, 2016).

[228] D. Dimova, J. Bajorath, Extraction of SAR information from activity cliff clusters via matching molecular series, Eur. J. Med. Chem. 87 (2014) 454–460. https://doi.org/10.1016/j.ejmech.2014.09.087.

[229] I. Peña, M. Pilar Manzano, J. Cantizani, A. Kessler, J. Alonso-Padilla, A.I. Bardera, E. Alvarez, G. Colmenarejo, I. Cotillo, I. Roquero, F. de Dios-Anton, V. Barroso, A. Rodriguez, D.W. Gray, M. Navarro, V. Kumar, A. Sherstnev, D.H. Drewry, J.R. Brown, J.M. Fiandor, J. Julio Martin, New Compound Sets Identified from High Throughput Phenotypic Screening Against Three Kinetoplastid Parasites: An Open Resource, Sci. Rep. 5 (2015) 8771. https://doi.org/10.1038/srep08771.

[230] A.A. Hagberg, D.A. Schult, P.J. Swart, Exploring Network Structure,

Dynamics, and Function using NetworkX, in: G. Varoquaux, T. Vaught (Eds.), Proc. Python Sci. Conf., Pasadena, 2008: pp. 11–15. http://conference.scipy.org/proceedings/SciPy2008/paper_2/ (accessed August 28, 2019).

[231] M. Bastian, S. Heymann, M. Jacomy, Gephi: an open source software for exploring and manipulating networks, in: Int. AAAI Conf. Weblogs Soc. Media, San Jose, 2009. https://gephi.org/.

[232] M. Jacomy, T. Venturini, S. Heymann, M. Bastian, ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software, PLoS One. 9 (2014) e98679. https://doi.org/10.1371/journal.pone.0098679.

[233] E.J. Mills, D.F.R.S. Sc, On melting-point and boiling-point as related to chemical composition, London, Edinburgh, Dublin Philos. Mag. J. Sci. 17 (1884) 173–187. https://doi.org/10.1080/14786448408627502.

[234] R. Todeschini, P. Gramatica, SD-modelling and Prediction by WHIM Descriptors. Part 5. Theory Development and Chemical Meaning of WHIM Descriptors, Quant. Struct. Relationships. 16 (1997) 113–119. https://doi.org/10.1002/qsar.19970160203.

[235] D. Cherqaoui, D. Villemin, V. Kvasnička, Application of neural network approach for prediction of some thermochemical properties of alkanes, Chemom. Intell. Lab. Syst. 24 (1994) 117–128. https://doi.org/10.1016/0169-7439(94)00012-3.

[236] J.C. Dearden, Quantitative structure-property relationships for prediction of boiling point, vapor pressure, and melting point, Environ. Toxicol. Chem. 22 (2003) 1696. https://doi.org/10.1897/01-363.

[237] J. Janowiak, E.B. Martin, R.L. Marchese Robinson, I. Giangreco, Melting Point Prediction using Message Passing Neural Networks based on Molecular and Crystal Structures, (2019).

[238] V. Korolev, A. Mitrofanov, A. Korotcov, V. Tkachenko, Graph Convolutional Neural Networks as “general-Purpose” Property Predictors: The Universality and Limits of Applicability, J. Chem. Inf. Model. 60 (2020) 22–28.

https://doi.org/10.1021/acs.jcim.9b00587.

[239] J. Lim, S. Ryu, K. Park, Y.J. Choe, J. Ham, W.Y. Kim, Predicting Drug-Target Interaction Using a Novel Graph Neural Network with 3D Structure-Embedded Graph Representation, J. Chem. Inf. Model. 59 (2019) 3981–3988. https://doi.org/10.1021/acs.jcim.9b00387.

[240] K. Cho, B. van Merrienboer, D. Bahdanau, Y. Bengio, On the Properties of Neural Machine Translation: Encoder-Decoder Approaches, Comput. Lang. (2014). http://arxiv.org/abs/1409.1259 (accessed August 15, 2019).

[241] P.C. St. John, C. Phillips, T.W. Kemper, A.N. Wilson, M.F. Crowley, M.R. Nimlos, R.E. Larsen, Message-passing neural networks for high-throughput polymer screening, ARXIV. 1807 (2018). http://arxiv.org/abs/1807.10363 (accessed October 12, 2018).

[242] L. Laugier, D. Bash, J. Recatala, H.K. Ng, S. Ramasamy, C.-S. Foo, V.R. Chandrasekhar, K. Hippalgaonkar, Predicting thermoelectric properties from crystal graphs and material descriptors - first application for functional materials, ArXiv:1811.06219. (2018). http://arxiv.org/abs/1811.06219 (accessed March 20, 2019).

[243] A.K. Rappe, C.J. Casewit, K.S. Colwell, W.A. Goddard III, W.M. Skiff, UFF, a full periodic table force field for molecular mechanics and molecular dynamics simulations, J. Am. Chem. Soc. 25 (1992) 10024–10035. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.208.7677&rep=rep1&type=pdf (accessed September 19, 2020).

[244] M.J. Bryant, A.G.P. Maloney, R.A. Sykes, Predicting mechanical properties of crystalline materials through topological analysis, CrystEngComm. 20 (2018) 2698–2704. https://doi.org/10.1039/c8ce00454d.

[245] Pande group, DeepChem, (2014). https://deepchem.io/docs/index.html.

[246] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O.

Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, (2015).

[247] E. Jones, T. Oliphant, P. Peterson, Others, SciPy: Open Source Scientific Tools for Python, (2001).

[248] Microsoft, Gated Graph Neural Networks, (2018). https://github.com/microsoft/gated-graph-neural-network-samples.

[249] I. Rosbottom, K.J. Roberts, R. Docherty, The solid state, surface and morphological properties of p-aminobenzoic acid in terms of the strength and directionality of its intermolecular synthons, CrystEngComm. 17 (2015) 5768–5788. https://doi.org/10.1039/C5CE00302D.

[250] A. Vriza, A.B. Canaj, R. Vismara, L.J. Kershaw Cook, T.D. Manning, M.W. Gaultois, P.A. Wood, V. Kurlin, N. Berry, M.S. Dyer, M.J. Rosseinsky, One class classification as a practical approach for accelerating π-π co-crystal discovery, Chem. Sci. 12 (2021) 1702–1719. https://doi.org/10.1039/d0sc04263c.

[251] R.L. Marchese Robinson, I. Lynch, W. Peijnenburg, J. Rumble, F. Klaessig, C. Marquardt, H. Rauscher, T. Puzyn, R. Purian, C. Åberg, S. Karcher, H. Vriens, P. Hoet, M.D. Hoover, C.O. Hendren, S.L. Harper, How should the completeness and quality of curated nanomaterial data be evaluated?, Nanoscale. 8 (2016) 9919–9943. https://doi.org/10.1039/C5NR08944A.

[252] S. Kang, K. Cho, Conditional Molecular Design with Deep Generative Models, Comput. Res. Repos. abs/1805.00108 (2018). http://arxiv.org/abs/1805.00108 (accessed July 19, 2018).

[253] A. Gandomi, M. Haider, Beyond the hype: Big data concepts, methods, and analytics, Int. J. Inf. Manage. 35 (2015) 137–144. https://doi.org/10.1016/j.ijinfomgt.2014.10.007.

[254] M.D. Wilkinson, M. Dumontier, Ij.J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.W. Boiten, L.B. da Silva Santos, P.E. Bourne, J. Bouwman, A.J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C.T. Evelo, R. Finkers, A. Gonzalez-Beltran, A.J.G. Gray, P. Groth, C. Goble, J.S. Grethe, J. Heringa, P.A.C. t Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S.J.

Lusher, M.E. Martone, A. Mons, A.L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M.A. Swertz, M. Thompson, J. Van Der Lei, E. Van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, B. Mons, Comment: The FAIR Guiding Principles for scientific data management and stewardship, Sci. Data. 3 (2016) 1–9. https://doi.org/10.1038/sdata.2016.18.

[255] EPSRC, Expectations - EPSRC website, (n.d.). https://epsrc.ukri.org/about/standards/researchdata/expectations/ (accessed September 9, 2020).

[256] B. Louis, V.K. Agrawal, P. V. Khadikar, Prediction of intrinsic solubility of generic drugs using MLR, ANN and SVM analyses, Eur. J. Med. Chem. 45 (2010) 4018–4025. https://doi.org/10.1016/j.ejmech.2010.05.059.

[257] A. Lusci, G. Pollastri, P. Baldi, Deep Architectures and Deep Learning in Chemoinformatics: The Prediction of Aqueous Solubility for Drug-Like Molecules, J. Chem. Inf. Model. 53 (2013) 1563–1575. https://doi.org/10.1021/ci400187y.

[258] I. Robinson, J. Webber, E. Eifrem, Graph Databases, 2nd ed., O'Reilly, Sebastopol, 2015.

[259] W. Zheng, A. Tropsha, Novel Variable Selection Quantitative Structure−Property Relationship Approach Based on the k -Nearest-Neighbor Principle, J. Chem. Inf. Comput. Sci. 40 (2000) 185–194. https://doi.org/10.1021/ci980033m.

[260] Organisation for Economic Co-operation and Development, OECD principles for the validation, for regulatory purposes, of (quantitative) structure-activity relationship models, 2004. hhttp://www.oecd.org/env/ehs/risk-assessment/validationofqsarmodels.htm (accessed September 6, 2019).

[261] M.T. Ribeiro, S. Singh, C. Guestrin, Why Should I Trust You? Explaining the Predictions of Any Classifier, ARXIV Mach. Learn. (2016). https://doi.org/10.1145/2939672.2939778.

[262] M. Salahinejad, T.C. Le, D.A. Winkler, Aqueous Solubility Prediction: Do Crystal Lattice Interactions Help?, Mol. Pharm. 10 (2013) 2757–2766.

https://doi.org/10.1021/mp4001958.

[263] G. Clydesdale, K.J. Roberts, R. Docherty, HABIT95 — a program for predicting the morphology of molecular crystals as a function of the growth environment, J. Cryst. Growth. 166 (1996) 78–83.

[264] K. He, X. Zhang, S. Ren, J. Sun, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, Comput. Vis. Pattern Recognit. (2015). http://arxiv.org/abs/1502.01852 (accessed May 12, 2019).

Appendix 1
Matched Molecular Pairs Database scripts

Purpose:

This appendix contains all scripts used to generate a Matched Molecular Pair Database (Chapter 3) and carry out the analysis (Chapter 4). Scripts that generate required files for Matched Molecular Graph generation (Chapter 6) is also included here.

Folder structure and uses:

- mmpdb
    - __init__.py
    - tables.py
    - get_smiles.py
    - frag.py
    - indexfrag.py
    - mmp_identification
- csd_addon.py: prepares input for database.py if using CSD as source
- database.py: creates MMP database
- analysis.py: performs MMPA on database, optionally prepares data for MMG

__init__.py

```
import tables
import frag
import indexfrag
import mmp_identification
```

tables.py

```
def solids_table(c):
    c.execute(' CREATE TABLE IF NOT EXISTS '
              'Solid_properties '
              '(solid_id VARCHAR(10) PRIMARY KEY, structure_family 
VARCHAR(6),'
              ' polymorph_count INTEGER, structure_count INTEGER)')


def smiles_table(c):
    c.execute('CREATE TABLE IF NOT EXISTS '
              'all_smiles '
              '(line_id INTEGER PRIMARY KEY, solid_id VARCHAR(10), 
mol_id INTEGER '
              ' )')


def frag_table(c):
    c.execute('CREATE TABLE IF NOT EXISTS '
              'rfrag '
              '(rfrag_id INTEGER PRIMARY KEY, mol_id INTEGER, core 
TEXT, core_ni TEXT, core_id INTEGER, chain TEXT, '
              'chain_size INTEGER, chain_id, single_cut INTEGER 
DEFAULT 0, indexed INTEGER DEFAULT 0)')


def trans_table(c):
    c.execute('CREATE TABLE IF NOT EXISTS Transformation ( '
              ' trans_id    INTEGER NOT NULL, '
              ' R1_id   INTEGER NOT NULL, '
              ' R2_id   INTEGER NOT NULL, '
              ' SMIRKS TEXT NOT NULL,'
              ' PRIMARY KEY(trans_id) '
              ');')


def mmp_table(c):
    c.execute('CREATE TABLE IF NOT EXISTS MMP '
              '(mmp_id INTEGER PRIMARY KEY, trans_id INTEGER NOT 
NULL, '
              'mol1_id INTEGER NOT NULL, mol2_id INTEGER NOT NULL, 
context INTEGER, '
              'FOREIGN KEY (trans_id) REFERENCES 
Transformation(trans_id))')


def context_table(c):
    c.execute("CREATE TABLE IF NOT EXISTS context_table "
              "                  ( "
              "                    context_id INTEGER PRIMARY KEY, "
              "                    context_smi VARCHAR(1000) NOT 
NULL UNIQUE, "
              "                    context_size INTEGER, "
              "                    single_cut INTEGER DEFAULT 0 "
              "    )")


def core_table(c):
    c.execute("CREATE TABLE IF NOT EXISTS fragments"
              "                   ("
              "                     context_id INTEGER NOT NULL,"
              "                     cmpd_id INTEGER NOT NULL,"
              "                     core_id INTEGER, "
              "                     core_size INTEGER,"
              "                     ratio REAL,"
              "                     single_cut INTEGER,"
              "                     significant INTEGER NOT NULL
DEFAULT 0"
              "                     "
              "    )")


def unique_core_table(c):
    c.execute("CREATE TABLE IF NOT EXISTS `core_table` ( "
              " `core_id`   INTEGER, "
              " `core_smi`  TEXT UNIQUE, "
              " `core_smi_ni`   TEXT, "
              " PRIMARY KEY(`core_id`) "
              "); ")


def mol_descriptor_table(c):
    c.execute("CREATE TABLE IF NOT EXISTS mol_properties  "
              "( mol_id INTEGER PRIMARY KEY, "
              "smiles VARCHAR(1000),cmpd_size INTEGER, fragmented
INTEGER DEFAULT 0, MMP_identified INTEGER DEFAULT 0, "
              "n_conf_20 INTEGER "
              ")")


def all_tables(c):
    solids_table(c)
    smiles_table(c)
    # frag_table(c)
    trans_table(c)
    mmp_table(c)
    context_table(c)
    core_table(c)
    unique_core_table(c)
    mol_descriptor_table(c)
```

get_smiles.py

```
#
# This script can be used for any purpose without limitation subject
to the
# conditions at
http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution
must be
# included in all copies or substantial portions of this script.
#
# 2017-02-07: created by the Cambridge Crystallographic Data Centre

from ccdc import search
from rdkit import Chem


def heavy_atom_count(smi):
    m = Chem.MolFromSmiles(smi)
    return m.GetNumAtoms()


def get_rdkit_mol(ccdc_mol):
    """Return RDKit molecule, with 2D coordinates, from a CCDC
molecule."""
    mol_block = Chem.MolFromMolBlock(ccdc_mol)
    rdkit_mol_smiles = Chem.MolToSmiles(mol_block,
isomericSmiles=True)
    return rdkit_mol_smiles


def generate_smiles_from_mol_block(mol):
    """Return an RDKit SMILES from a sdf mol block of a CCDC
molecule."""
    mol_block = mol.to_string('sdf')
    return get_rdkit_mol(mol_block)


def generate_smiles_from_kekulized_mol_block(mol):
    """Return an RDKit SMILES from a sdf mol block of a kekulized
CCDC molecule."""
    mol.kekulize()
    kekulized_mol = mol.to_string('sdf')
    return get_rdkit_mol(kekulized_mol)


def generate_smiles_from_csd(mol):
    """Return an CSD SMILES from a CCDC molecule."""
    csd_smiles = mol.smiles
    return Chem.MolToSmiles(Chem.MolToSmiles(csd_smiles),
isomericSmiles=True)


class RDKitChargeConventionSetter:
    def __init__(self):
        self.editors = []
        # You can add in any other edits you need here
        # The pairs (1,-1) mean 'transform the atom labeled '1' in
the SMARTS to have a charge '-1', etc.'
        self._add_editor('[OX1:1]-[nX3:2]',[ (1,-1),(2,1) ])
        self._add_editor('[!#1]=[N:1]=[N:2]',[ (1,1),(2,-1) ])

    def _add_editor(self, smarts_pattern, charge_transformation):
        searcher = search.SubstructureSearch()
        sub = search.SMARTSSubstructure(smarts_pattern)
        searcher.add_substructure(sub)
        self.editors.append( (searcher, sub,
charge_transformation ) )

    def _charge_balance_molecule_with_editor(self, mol, editor):
        hits = editor[0].search(mol)
        for hit in hits:
            hit_atom_indexes = hit.match_atoms(indices=True)
            substructure = editor[1]
            for pair in editor[2]:
                sub_atom_index =
substructure.label_to_atom_index(pair[0])

mol.atoms[ hit_atom_indexes[sub_atom_index] ].partial_charge =
float(pair[1])

mol.atoms[ hit_atom_indexes[sub_atom_index] ].formal_charge  =
int(pair[1])

    def charge_balance_molecule(self, mol):
        for editor in self.editors:
            self._charge_balance_molecule_with_editor(mol,editor)


def generate_smiles(entry):
    mol = entry.molecule
    try:
        smiles = generate_smiles_from_mol_block(mol)
        method = 'mol block'
    except:
        try:
            smiles = generate_smiles_from_kekulized_mol_block(mol)
            method = 'kekulized mol block'
        except:
            try:
                setter = RDKitChargeConventionSetter()
                setter.charge_balance_molecule(mol)
                smiles = generate_smiles_from_mol_block(mol)
                method = "mol block with charges for N-oxide"
            except:
                try:
                    smiles = generate_smiles_from_csd(mol)
                    method = "CSD and canonicalised with RDKit"
                except:
                    smiles = ''
                    method = 'unable'
    return smiles, method
```

frag.py

```
# Copyright (c) 2013, GlaxoSmithKline Research & Development Ltd.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
are
# met:
#
#     * Redistributions of source code must retain the above
copyright
#       notice, this list of conditions and the following
disclaimer.
#     * Redistributions in binary form must reproduce the above
#       copyright notice, this list of conditions and the following
#       disclaimer in the documentation and/or other materials
provided
#       with the distribution.
#     * Neither the name of GlaxoSmithKline Research & Development
Ltd.
#       nor the names of its contributors may be used to endorse or
promote
#       products derived from this software without specific prior
written
#       permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT
# OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
#
# Created by Jameed Hussain, July 2013
#
# Modifications and optimizations by Greg Landrum, July 2015
#

import re
from rdkit import Chem
from rdkit.Chem import rdMMPA


def find_correct(f_array):

  core = ""
  side_chains = ""

  for f in f_array:
    attachments = f.count("*")
    if (attachments == 1):
      side_chains = "%s.%s" % (side_chains, f)
    else:
      core = f

  side_chains = side_chains.lstrip('.')

  #cansmi the side chains
  temp = Chem.MolFromSmiles(side_chains)
  side_chains = Chem.MolToSmiles(temp, isomericSmiles=True)

  #and cansmi the core
  temp = Chem.MolFromSmiles(core)
  core = Chem.MolToSmiles(temp, isomericSmiles=True)

  return core, side_chains


def delete_bonds(smi, id, mol, bonds, out):

  #use the same parent mol object and create editable mol
  em = Chem.EditableMol(mol)

  #loop through the bonds to delete
  isotope = 0
  isotope_track = {}
  for i in bonds:
    isotope += 1
    #remove the bond
    em.RemoveBond(i[0], i[1])

    #now add attachement points
    newAtomA = em.AddAtom(Chem.Atom(0))
    em.AddBond(i[0], newAtomA, Chem.BondType.SINGLE)

    newAtomB = em.AddAtom(Chem.Atom(0))
    em.AddBond(i[1], newAtomB, Chem.BondType.SINGLE)

    #keep track of where to put isotopes
    isotope_track[newAtomA] = isotope
    isotope_track[newAtomB] = isotope

  #should be able to get away without sanitising mol
  #as the existing valencies/atoms not changed
  modifiedMol = em.GetMol()

  #canonical smiles can be different with and without the isotopes
  #hence to keep track of duplicates use fragmented_smi_noIsotopes
  fragmented_smi_noIsotopes = Chem.MolToSmiles(modifiedMol,
isomericSmiles=True)

  valid = True
  fragments = fragmented_smi_noIsotopes.split(".")

  #check if its a valid triple cut
  if (isotope == 3):
    valid = False
    for f in fragments:
      matchObj = re.search('\*.*\*.*\*', f)
      if matchObj:
        valid = True
        break

  if valid:
    if (isotope == 1):
      fragmented_smi_noIsotopes = re.sub('\[\*\]', '[*:1]',
fragmented_smi_noIsotopes)

      fragments = fragmented_smi_noIsotopes.split(".")

      #print fragmented_smi_noIsotopes
      s1 = Chem.MolFromSmiles(fragments[0])
      s2 = Chem.MolFromSmiles(fragments[1])

      #need to cansmi again as smiles can be different
      output = '%s,%s,,%s.%s' % (smi, id, Chem.MolToSmiles(s1,
isomericSmiles=True),
                                 Chem.MolToSmiles(s2,
isomericSmiles=True))
      if ((output in out) == False):
        out.add(output)

    elif (isotope >= 2):
      #add the isotope labels
      for key in isotope_track:
        #to add isotope lables

modifiedMol.GetAtomWithIdx(key).SetIsotope(isotope_track[key])
      fragmented_smi = Chem.MolToSmiles(modifiedMol,
isomericSmiles=True)

      #change the isotopes into labels - currently can't add SMARTS
or labels to mol
      fragmented_smi = re.sub('\[1\*\]', '[*:1]', fragmented_smi)
      fragmented_smi = re.sub('\[2\*\]', '[*:2]', fragmented_smi)
      fragmented_smi = re.sub('\[3\*\]', '[*:3]', fragmented_smi)

      fragments = fragmented_smi.split(".")

      #identify core/side chains and cansmi them
      core, side_chains = find_correct(fragments)

      #now change the labels on sidechains and core
      #to get the new labels, cansmi the dot-disconnected side
chains
      #the first fragment in the side chains has attachment label 1,
2nd: 2, 3rd: 3
      #then change the labels accordingly in the core

      #this is required by the indexing script, as the side-chains
are "keys" in the index
      #this ensures the side-chains always have the same numbering

      isotope_track = {}
      side_chain_fragments = side_chains.split(".")

      for s in range(len(side_chain_fragments)):
        matchObj = re.search('\[\*\:([123])\]',
side_chain_fragments[s])
        if matchObj:
          #add to isotope_track with key: old_isotope, value:
          isotope_track[matchObj.group(1)] = str(s + 1)

      #change the labels if required
      if (isotope_track['1'] != '1'):
        core = re.sub('\[\*\:1\]', '[*:XX' + isotope_track['1'] +
'XX]', core)
        side_chains = re.sub('\[\*\:1\]', '[*:XX' +
isotope_track['1'] + 'XX]', side_chains)
      if (isotope_track['2'] != '2'):
        core = re.sub('\[\*\:2\]', '[*:XX' + isotope_track['2'] +
'XX]', core)
        side_chains = re.sub('\[\*\:2\]', '[*:XX' +
isotope_track['2'] + 'XX]', side_chains)

      if (isotope == 3):
        if (isotope_track['3'] != '3'):
          core = re.sub('\[\*\:3\]', '[*:XX' + isotope_track['3'] +
'XX]', core)
          side_chains = re.sub('\[\*\:3\]', '[*:XX' +
isotope_track['3'] + 'XX]', side_chains)

      #now remove the XX
      core = re.sub('XX', '', core)
      side_chains = re.sub('XX', '', side_chains)
      output = '%s,%s,%s,%s' % (smi, id, core, side_chains)
      if ((output in out) == False):
        out.add(output)


def fragment_mol(smi, id):

  mol = Chem.MolFromSmiles(smi)

  #different cuts can give the same fragments
  #to use outlines to remove them
  outlines = set()

  if (mol == None):
    print("Can't generate mol for: %s\n" % (smi))
    return
  else:
    frags = rdMMPA.FragmentMol(mol,
pattern="[#6+0;!$(*=,#[!#6])]!@!=!#[*]", resultsAsMols=False)
    for core, chains in frags:
      output = (str(smi),str(id),str(core),str(chains))
      if (not (output in outlines)):
        outlines.add(output)
  return outlines
```

indexfrag.py

```
# Copyright (c) 2013, GlaxoSmithKline Research & Development Ltd.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
are
# met:
#
#     * Redistributions of source code must retain the above
copyright
#       notice, this list of conditions and the following
disclaimer.
#     * Redistributions in binary form must reproduce the above
#       copyright notice, this list of conditions and the following
#       disclaimer in the documentation and/or other materials
provided
#       with the distribution.
#     * Neither the name of GlaxoSmithKline Research & Development
Ltd.
#       nor the names of its contributors may be used to endorse or
promote
#       products derived from this software without specific prior
written
#       permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT
# OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
#
# Created by Jameed Hussain, July 2013
#
# modifications by Jakub Janowiak, 2018

import pandas as pd
import re
from rdkit import Chem



# \\\\\\\\\\\\\\\\\\\\\core side/////////////////////
def core_db(conn):
    """Return pandas.DataFrame with cores from database."""
    sql = '''SELECT core_id, core_smi FROM core_table'''
    cores_old = pd.read_sql_query(sql, conn)
    cores_old['old'] = True
    # they are loaded in random order for some reason
    cores_old =
cores_old.sort_values(by=['core_id']).reset_index().drop('index',
axis=1)

    return cores_old


def get_cores(frag_df):
    """Return pandas.Dataframe with cores from frag_df."""
    cores_new = frag_df.drop_duplicates(subset='core_smi')
    cores_new.drop(['cmpd_size', 'context_size', 'core_size',
'ratio', 'context_smi', 'mol_id', 'single_cut'], axis=1,
inplace=True)
    cores_new['old'] = False
    return cores_new


def core_all(cores_new, cores_old):
    """Return a combined pandas.DataFrame with old and new
contexts"""
    if cores_old.empty is True:
        line = {'core_smi': ['[*:1][H]'], 'core_id': [0], 'old':
[False]}
        cores = pd.DataFrame(line, columns=line.keys())
        cores = pd.concat([cores, cores_new])
        cores = cores.reset_index().drop('index', axis=1)

        assert cores.loc[0, 'core_id'] == 0, 'something went wrong
with generating core_table; index issue'

    else:
        cores = pd.concat([cores_old, cores_new]).reset_index()
        cores.rename(columns={'core_id': 'core_old'}, inplace=True)
        # drops the contexts that were already in the database
        cores.drop_duplicates(subset='core_smi', keep='first',
inplace=True)
        cores.drop('index', axis=1, inplace=True)

        cores = cores.reset_index()
        cores.drop('index', axis=1, inplace=True)
        # check the old index didnt change
        row_index = len(cores[(cores['old'] == True) & (cores.index
== cores['core_old'])])
        row_old = len(cores[cores['old'] == True])
        assert row_old == row_index, 'duplicate cores found in
database.'

        cores.drop('core_old', axis=1, inplace=True)

    cores['core_id'] = cores.index

    return cores


def get_smi_ni(smi):
    """Return fragment string without numbered cuts """
    smi = re.sub(r'\[\*\:1\]', '[*]', smi)
    smi = re.sub(r'\[\*\:2\]', '[*]', smi)
    smi = re.sub(r'\[\*\:3\]', '[*]', smi)
    return  smi


def core_to_db(cores, conn):
    """Insert contexts to database and return the corresponding
pandas.Dataframe"""
    new_cores = cores[cores['old'] == False]
    # drop 'old' column
    new_cores = new_cores.drop('old', axis=1)
    if not new_cores.empty:
        new_cores['core_smi_ni'] = new_cores.apply(lambda row:
get_smi_ni(row['core_smi']), axis=1)
        new_cores.to_sql('core_table', conn, if_exists='append',
index=False) # TODO check this, hasnt been done yet


# \\\\\\\\\\\\\\\\\\\\\context side//////////////////////

def context_db(conn):
    """Return pandas.DataFrame with contexts from database."""
    sql = '''SELECT context_smi, context_id, context_size,
single_cut FROM context_table'''
    contexts_old = pd.read_sql_query(sql, conn)
    contexts_old['old'] = True
    contexts_old =
contexts_old.sort_values(by=['context_id']).reset_index().drop('inde
x', axis=1)

    return contexts_old


def get_contexts(frag_df):
    """Return pandas.Dataframe with contexts from frag_df."""
    contexts_new = frag_df.drop_duplicates(subset='context_smi')
    contexts_new = contexts_new.reset_index()
    contexts_new = contexts_new.drop(['cmpd_size', 'core_size',
'ratio', 'core_smi', 'mol_id', 'index'], axis=1)
    contexts_new['old'] = False

    return contexts_new


def context_all(contexts_new, contexts_old):
    """Return a combined pandas.DataFrame with old and new
contexts"""
    if contexts_old.empty is True:
        contexts = contexts_new
    else:
        contexts = pd.concat([contexts_old,
contexts_new]).reset_index()
        contexts.rename(columns={'context_id': 'context_old'},
inplace=True)
        # drops the contexts that were already in the database
        contexts.drop_duplicates(subset='context_smi', keep='first',
inplace=True)
        contexts.drop('index', axis=1, inplace=True)

        contexts = contexts.reset_index()
        contexts.drop('index', axis=1, inplace=True)
        # check the old index didnt change
        row_index = len(contexts[(contexts['old'] == True) &
(contexts.index == contexts['context_old'])])
        row_old = len(contexts[contexts['old'] == True])
        if row_index != row_old:
            raise Exception('duplicate contexts found in database.')
        contexts.drop('context_old', axis=1, inplace=True)

    contexts['context_id'] = contexts.index

    return contexts


def context_to_db(contexts, conn):
    """Insert contexts to database and return the corresponding
pandas.Dataframe"""
    new_contexts = contexts[contexts['old'] == False]
    # drop 'old' column
    new_contexts = new_contexts.drop('old', axis=1)
    if not new_contexts.empty:
        new_contexts.to_sql('context_table', conn,
if_exists='append', index=False) # TODO check this, hasnt been done
yet

    return new_contexts


def h_change(smi):
    """ """
    # replace [1] with H
    smi = re.sub(r'\[\*\:1\]', '[H]', smi)
    # construct a mol
    temp = Chem.MolFromSmiles(smi)
    if temp is None:
        print('failed to generate Chem.Mol for  {}'.format(smi))
        mol = None
    else:
        mol = Chem.MolToSmiles(temp, isomericSmiles=True)

    # return smiles
    return mol


def index_h_change(new_contexts, conn):
    new_contexts['mol'] = new_contexts.apply(lambda row:
h_change(row['context_smi']) if (row['single_cut'] == 1) else None,
axis=1)
    # drop na
    new_contexts.drop('single_cut',axis=1, inplace=True)
    new_contexts.dropna(axis=0, how='any', inplace=True)
    # new_contexts['context_id'] = new_contexts.index # loses index
on merge
    # merge with all_smiles (inner)
    sql = 'SELECT smiles, mol_id FROM mol_properties'
    all_smiles = pd.read_sql_query(sql,conn)
    new_contexts = new_contexts.merge(all_smiles, how='inner',
left_on='mol', right_on='smiles')
    new_contexts.drop(['smiles', 'mol'], axis=1, inplace=True)
    new_contexts['ratio'] = 0
    new_contexts['core_size'] = 0
    new_contexts['core_smi'] = '[*:1][H]'
    new_contexts['core_id'] = 0
    new_contexts['single_cut'] = 1
    new_contexts.rename(columns={'unique_smiles_id': 'smiles_id'},
inplace=True)

    return new_contexts


# \\\\\\\\\\\\\\\\\\\\\fragments//////////////////////
def get_id(frag_df, contexts, cores):
    """Return pandas.DataFrame with context_id and core_id"""
    contexts.drop(['context_size', 'old', 'single_cut'], axis=1,
inplace=True)
    cores.drop('old', axis=1, inplace=True)
    frag_df.drop('cmpd_size', axis=1, inplace=True)
    frag_df = frag_df.merge(contexts, how='left', on='context_smi')
    frag_df = frag_df.merge(cores, how='left', on='core_smi')

    return frag_df


# \\\\\\\\\\\\\\\\\\\\\main//////////////////////


def index_core(frag_df, conn):
    """ """
    cores_old = core_db(conn)
    cores_new = get_cores(frag_df)

    cores = core_all(cores_new, cores_old)

    core_to_db(cores, conn)

    return cores


def index_context(frag_df, conn):
    """ """
    contexts_old = context_db(conn)
    contexts_new = get_contexts(frag_df)

    contexts = context_all(contexts_new, contexts_old)

    if not contexts[contexts['old'] == False].empty:
        h_frag = context_to_db(contexts, conn)
        h_frag = index_h_change(h_frag, conn)
    else:
        h_frag = None
    return contexts, h_frag


def index_main(frag_df, conn):
    cores = index_core(frag_df, conn)
    contexts, h_frag = index_context(frag_df, conn)
    frag_df = get_id(frag_df,contexts,cores)

    if h_frag is not None:
        frag_df = pd.concat([frag_df,
h_frag]).reset_index(drop=True)
```

```
    frag_df = frag_df.drop(['context_smi', 'core_smi',
'context_size'], axis=1)
    frag_df.rename(columns={'mol_id': 'cmpd_id'}, inplace=True)
    frag_df.to_sql('fragments', conn, if_exists='append',
index=False)
```

mmp_identification.py

```python
import pandas as pd
import numpy as np
# from joblib import Parallel, delayed
import time

#from __main__ import conn
#from __main__ import c


context = {}


class DatabaseException(Exception):
    pass



"""returns compound list (cmpd_id, cmpd_size) and DataFrame with
fragments"""


def load_data(ratio, max_size, conn):
    sql_cmpd = (''
                'SELECT distinct(cmpd_id), cmpd_size '
                'FROM fragments, mol_properties '
                'WHERE cmpd_id = mol_id  AND MMP_identified = 0 '
                '')

    sql_frag = (''
                'SELECT fragments.cmpd_id, fragments.core_id,
fragments.context_id, context_size '
                'FROM fragments, context_table '
                'WHERE significant = 1 AND context_table.context_id
= fragments.context_id '
                'AND ratio <= {ratio} AND core_size <= {size}'
                '')
    var = {'ratio': ratio, 'size': max_size}
    sql_frag = sql_frag.format(**var)
    df_cmpd = pd.read_sql_query(sql_cmpd, conn)
    cmpd_list = df_cmpd.values.tolist()
    global fragmts
    fragmts = pd.read_sql_query(sql_frag, conn)

    return cmpd_list


def get_contexts(cmpd):
    """creates a dictionary with cmpd_id: [context_id,...]..."""
    c_id = cmpd[0]
    contexts = fragmts.context_id[fragmts.cmpd_id == c_id]
    context_list = contexts.values.tolist()
    context[c_id] = context_list


def find_mmp(cmpd):
    """finds MMPs of a molecule. Uses context dict and frag
DataFrame"""
    c_id, c_size = cmpd
    contexts = context[c_id]
    all_mmp = fragmts[(fragmts.context_id.isin(contexts)) &
(fragmts.cmpd_id != c_id)].groupby(fragmts['cmpd_id']) # TODO change
the conditions removed:  & (c_size - fragmts.context_id < 15)
    mmp_list = []
    for mmp in list(all_mmp):
        df = mmp[1]
        pair = df.ix[df['context_size'].idxmax()]
        mmp_list.append(pair)
    if mmp_list:
        mmps = pd.concat(mmp_list, axis=1).T.reset_index()
    else:
        mmps = None
    return c_id, mmps


def sort_mmp(mmp_set):
    """combines the generated mmp tables of each molecule into
one"""
    full_mmp_set = []
    NoneType = type(None)
    for i in mmp_set:
        if type(i[1]) is NoneType:
            continue
        else:
            table = i[1]
            table = table.rename(columns={'cmpd_id': 'mol2',
'core_id': 'R2', 'context_id': 'context'})
            table = table.assign(mol1=i[0])
            table = table.astype(int)

            full_mmp_set.append(table)
    # TODO sort it out for the case for no MMPs
    mmp_table = pd.concat(full_mmp_set)
    mmp_table = mmp_table.reset_index()
    mmp_table.drop(['level_0', 'index'], axis=1, inplace=True)
    mmp_table = mmp_table[mmp_table['mol1'] !=
mmp_table['mol2']].reset_index()
    mmp_table.drop('index', axis=1, inplace=True)
    return mmp_table


def find_r1(mmp):
    """adds symmetrical transformation to mmp dataframe"""
    mmp = mmp.merge(fragmts[['cmpd_id', 'core_id', 'context_id']],
left_on=['mol1', 'context'], right_on=['cmpd_id', 'context_id'],
how='left')
    mmp['R1'] = mmp['R1'].fillna(mmp['core_id'])
    mmp.drop(['context_id', 'core_id', 'cmpd_id'], axis=1,
inplace=True)
    mmp['R1'] = mmp['R1'].astype(int)
    return mmp


def reorganise(mmp):
    """reorganises mmps and transformation to avoid counting
symmetrical pairs as separate"""
    # reorders mol1, mol2 and R1, R2 so all are in order of R2 > R1
    idx = (mmp['R1'] > mmp['R2'])
    mmp.loc[idx, ['mol1', 'mol2']] = mmp.loc[idx, ['mol2',
'mol1']].values
    mmp.loc[idx, ['R1', 'R2']] = mmp.loc[idx, ['R2', 'R1']].values
    # drops duplicate
    mmp = mmp.drop_duplicates(subset=['mol1', 'mol2']).reset_index()
    mmp.drop('index', axis=1, inplace=True)
    return mmp


def get_trans(mmp):
    """selects all unique R1-R2 combinations from identified mmps"""
    trans = mmp[['R1', 'R2']].drop_duplicates(subset=['R1',
'R2']).reset_index()
    trans.drop('index', axis=1, inplace=True)
    trans['old'] = False
    trans.index.rename('trans_id', inplace=True)
    return trans


def trans_db(conn):
    """loads all transformations from the database"""
    sql = '''SELECT trans_id, R1_id, R2_id FROM Transformation'''
    trans_old = pd.read_sql_query(sql, conn, index_col='trans_id')
    trans_old['old'] = True
    trans_old = trans_old.rename(columns={'R1_id': 'R1', 'R2_id':
'R2'})
    return trans_old


def trans_all(trans_old, trans_new):
    """combines newly identified transformations with ones from the
database"""
    if trans_old.empty is True:
        trans = trans_new
    else:
        trans = pd.concat([trans_old, trans_new]).reset_index()
        trans = trans.rename(columns={'trans_id': 'transid_old'})
        # drops the transformations that were already in the
database
        trans.drop_duplicates(subset=['R1', 'R2'], keep='first',
inplace=True)
        # TODO see context and core workflows, see if index needs to
be reset
        # check the old index didnt change by accident
        row_index = len(trans[(trans['old'] == True) & (trans.index
== trans['transid_old'])])
        row_old = len(trans[trans['old'] == True])
        if row_index != row_old:
            raise DatabaseException('duplicate transformations found
in database')
        trans.drop('transid_old', axis=1, inplace=True)
    trans['trans_id'] = trans.index
    return trans


def get_smirks(trans, conn):
    sql = 'SELECT core_id, core_smi FROM core_table'
    cores = pd.read_sql_query(sql, conn)
    trans = trans.merge(cores, how='left', left_on='R1_id',
right_on='core_id')
    trans.rename(columns={'core_smi': 'R1'}, inplace=True)
    trans.drop('core_id', axis=1, inplace=True)
    trans = trans.merge(cores, how='left', left_on='R2_id',
right_on='core_id')
```

```
    trans.rename(columns={'core_smi': 'R2'}, inplace=True)
    trans.drop('core_id', axis=1, inplace=True)
    trans['SMIRKS'] = trans['R1'] + '>>' + trans['R2']
    trans.drop(['R1', 'R2'], axis=1, inplace=True)
    return trans


def trans_to_db(trans, conn):
    """appends the new unique transformations to the database"""
    # remove the old ones that are already in the database
    trans = trans[trans['old'] == False]
    # adjust the table so it matches the database table
    trans = trans.rename(columns={'R1': 'R1_id', 'R2': 'R2_id'})
    trans.drop('old', axis=1, inplace=True)
    trans = get_smirks(trans, conn)
    trans.to_sql('Transformation', conn, if_exists='append',
index=False)


def mmp_to_db(trans, mmp, conn):
    """appends the newly identified mmps to the database"""
    # merge transformations and mmps on R1 and R2 to get transid for
mmps
    mmp = pd.merge(mmp, trans, how='left', left_on=['R1', 'R2'],
right_on=['R1', 'R2'])
    # adjust the DataFrame to match the database table
    mmp.drop(['R1', 'R2', 'context_size'], axis=1, inplace=True)
    mmp = mmp.rename(columns={'mol1': 'mol1_id', 'mol2': 'mol2_id'})
    mmp.to_sql('MMP', conn, if_exists='append', index=False)


def identify_mmps(ratio, max_size, conn, c):
    cmpd_list = load_data(ratio, max_size, conn)
    for i in cmpd_list:
        get_contexts(i)
    # potentially combine the two so contexts are searched on the
fly. might save memory & time
    ti = time.time()
    # mmps = Parallel(n_jobs=2)(delayed(find_mmp)(i) for i in
cmpd_list)
    mmp_list = []
    for i in cmpd_list:
        mmp = find_mmp(i)
        mmp_list.append(mmp)
    tf = time.time()
    print(tf-ti)
    ti = time.time()
    mmps = sort_mmp(mmp_list)
    tf = time.time()
    print(tf-ti)
    mmps['R1'] = np.nan # add r1 column
    ti = time.time()
    mmps = find_r1(mmps)
    tf = time.time()
    print(tf-ti)
    ti = time.time()
    mmps = reorganise(mmps)
    tf = time.time()
    print(tf-ti)
    ti = time.time()
    trans_new = get_trans(mmps)
    trans_old = trans_db(conn)
    trans = trans_all(trans_old, trans_new) #TODO check if works
    tf = time.time()
    print(tf - ti)
    ti = time.time()
    trans_to_db(trans, conn)
    tf = time.time()
    print(tf - ti)
    ti = time.time()
    trans.drop('old', axis=1, inplace=True)
    mmp_to_db(trans, mmps, conn)
    tf = time.time()
    print(tf - ti)
    ti = time.time()
    c.execute('UPDATE mol_properties '
              'SET MMP_identified = 1 '
              'WHERE MMP_identified = 0 AND '
              'mol_id IN (SELECT distinct(cmpd_id) FROM fragments)
')
    tf = time.time()
    print(tf - ti)
    conn.commit()
```

database.py

```
import sqlite3
import time
import argparse
import sys
import re
import os
import glob
import csv
import pandas as pd
from rdkit import Chem

try:
    import mmpdb # TODO check if it imports from the folder or
outside of it
except ImportError:
    print('could not import necessary files. Ensure mmpdb folder is
in a directory python can access')
    sys.exit(1)

# -------------------- Input smiles --------------------
def heavy_atom_count(smi):
    try:
        mol = Chem.MolFromSmiles(smi)
        return mol.GetNumAtoms()
    except AttributeError:
        return None

def solid_in_db(input_smiles, conn):
    # check if solid_id in database already
    # remove once that already in
    # return the df
    sql = 'SELECT solid_id FROM solid_properties'
    in_db = pd.read_sql_query(sql, conn)
    if not in_db.empty:
        input_smiles = input_smiles.merge(in_db, how='outer',
indicator=True, on='solid_id')
        input_smiles =
input_smiles[input_smiles['_merge']=='left_only']
        input_smiles = input_smiles.drop('_merge',axis=1)
    return input_smiles

def solid_to_db(input_smiles, conn):
    # select distinct solid_id
    # append to db
    # no returns
    solids = input_smiles.drop_duplicates(subset='solid_id')
    solids.drop('smiles', axis=1, inplace=True)
    solids.to_sql('Solid_properties', conn, if_exists='append',
index=False)

def all_smiles_to_db(input_smiles, conn):
    # add all smiles to perserve stoich
    to_db = input_smiles.drop(['smiles', 'old'], axis=1)
    to_db.to_sql('all_smiles', conn, if_exists='append',
index=False)

def get_smiles(input_smiles):
    smiles_new = input_smiles.drop_duplicates(subset='smiles')
    smiles_new = smiles_new.reset_index(drop=True)

    smiles_new['old'] = False
    return smiles_new

def smiles_db(conn):
    sql = 'SELECT mol_id, smiles FROM mol_properties' # check names
    smiles_old = pd.read_sql_query(sql, conn)
    smiles_old['old'] = True
    smiles_old =
smiles_old.sort_values(by=['mol_id']).reset_index(drop=True)
    return smiles_old

def smiles_all(smiles_new, smiles_old):
    if smiles_old.empty is True:
        smiles = smiles_new
    else:
        smiles =
pd.concat([smiles_old,smiles_new]).reset_index(drop=True)
        smiles.rename(columns={'mol_id':'smiles_old'}, inplace=True)
        smiles.drop_duplicates(subset='smiles', keep='first',
inplace=True)
        smiles.reset_index(drop=True)
        row_index = len(smiles[(smiles['old'] == True) &
(smiles.index == smiles['smiles_old'])])
        row_old = len(smiles[smiles['old'] == True])
        if row_index != row_old:
            raise Exception('duplicate {} found in
database.'.format('smiles'))
        smiles.drop('smiles_old', axis=1, inplace=True)
    smiles['mol_id'] = smiles.index
    return smiles

def smiles_to_db(smiles, conn):
    smiles['cmpd_size'] = smiles.apply(lambda row:
heavy_atom_count(row['smiles']),axis=1)
    smiles['fragmented'] = 0
    smiles['MMP_identified'] = 0
    smiles.to_sql('mol_properties', conn, if_exists='append',
index=False)

def input_smiles(input_file, conn):
    input_smiles = pd.read_csv(input_file, header=None)

    col_count = len(input_smiles.columns)
    if col_count == 1:
        solid_state = False
        input_smiles.columns = ['smiles']
    elif col_count == 2:
        solid_state = True

        input_smiles.columns = ['smiles','solid_id']
    else:
        print('something wrong with input')
        sys.exit(1)
    # in some cases there might be empty smiles (failure on the
input side)
    input_smiles.dropna(axis=0, how='any', inplace=True)

    if solid_state:
        input_smiles = solid_in_db(input_smiles, conn)
        solid_to_db(input_smiles, conn)
        smiles_new =
get_smiles(input_smiles.drop('solid_id',axis=1))
        # create a all_smiles df without adding to db
    else:
        smiles_new = get_smiles(input_smiles)
    # assign smiles_ids to all new molecules

    smiles_old = smiles_db(conn)
    smiles = smiles_all(smiles_new,smiles_old)
    if solid_state:
        # merge with all_smiles on smiles to get the new mol_id
        # add to db from all_smiles df
        input_smiles = input_smiles.merge(smiles, how='left',
on='smiles')
        all_smiles_to_db(input_smiles, conn)
    # drop 'old' column
    smiles = smiles[smiles['old']==False]
    smiles.drop('old',axis=1, inplace=True)
    smiles_to_db(smiles, conn)


# need to change unique_smiles_id = smiles_id bit everywhere
 # -------------------- Fragmentation --------------------
def get_context_size(context, attachments):
    mol = Chem.MolFromSmiles(context)
    size = mol.GetNumAtoms() - attachments
    return size


def fragmnt(smi, id, cmpd_size):
    """ """
    o = mmpdb.frag.fragment_mol(smi, id)
    frags = []
    if o:
        for l in o:
            core = l[2]
            chains = l[3]
            # no fragments
            if core == '' and chains == '':
                continue

            #single cut
            elif core == '':
                single_cut = 1
                side_chains = chains.split('.')

                # frag1-frag2 -> context: frag1, change: frag2
                context, change = side_chains
                context_size = get_context_size(context, 1)
                f = {'mol_id': id, 'context_smi': context,
'core_smi': change, 'single_cut': single_cut,
                     'cmpd_size': cmpd_size, 'context_size':
context_size}
                frags.append(f)

                # frag1-frag2 -> context: frag2, change: frag1
                change, context = side_chains
                context_size = get_context_size(context, 1)
                f = {'mol_id': id, 'context_smi': context,
'core_smi': change, 'single_cut': single_cut,
                     'cmpd_size': cmpd_size, 'context_size':
context_size}
                frags.append(f)

            # double / triple cut
            else:
                single_cut = 0
                context = chains
                change = core
                attachments = context.count('*')
                context_size = get_context_size(context,
attachments)
                f = {'mol_id': id, 'context_smi': context,
'core_smi': change, 'single_cut': single_cut,
                     'cmpd_size': cmpd_size, 'context_size':
context_size}
                frags.append(f)

    return frags


def fragment(c):
    c.execute('SELECT smiles, mol_id, cmpd_size '
              'FROM mol_properties '
              'WHERE fragmented = 0 ')
    results = c.fetchall()
    all_frags = []
    t1 = time.time()
    for line in results:
        frags = fragmnt(line[0], line[1], line[2])
        all_frags = all_frags + frags
    t2 = time.time()
    print(t2-t1)
    frag_df = pd.DataFrame(all_frags,columns=['mol_id',
'context_smi', 'core_smi', 'single_cut', 'cmpd_size',
'context_size'])

    frag_df['core_size'] = frag_df['cmpd_size'] -
frag_df['context_size']
    frag_df['ratio'] = frag_df['core_size'] / frag_df['cmpd_size']
    t3 = time.time()
    print(t3 - t2)
    c.execute('UPDATE mol_properties '
              'SET fragmented = 1') # there should be no need to add
this, make sure it aint done anywhere else
    return frag_df


 # -------------------- Indexing --------------------

# done in main()

def screen_cores(min_core_count, conn, c):
    if min_core_count != 0:
        c.execute('UPDATE fragments SET significant = 0')
        c.execute('UPDATE fragments SET significant = 1 WHERE
core_id IN '
                  '(SELECT core_id FROM fragments '
                  'GROUP BY core_id HAVING count(core_id) > ?)',
(min_core_count,))
    else:
        c.execute('UPDATE fragmnets SET significant = 1')
    conn.commit()
 # --------------------- MMP identification --------------------
# done in main()
 # --------------------- Main --------------------
def main():
    parser = argparse.ArgumentParser('''Generates SMILES from CSD
entries based on
the refcodes in the input file or CSD search.
Unique SMILES are fragmented and indexed using rdkit/MMPA.
Identified MMPs are added to database.''')

    parser.add_argument('input', help='input text file with refcodes
or type "CSD"')
    parser.add_argument('-o', '--output', default='MMP.db',
help='database name (default = MMP.db)')
    parser.add_argument('-r', '--ratio', default=0.3, help='max
ratio of change allowed. ratio = size of change / cmpd.'
                                                           ' Set to
1 to ignore ratios. (default = 0.3)')
    parser.add_argument('-c', '--change', default=10, help='max size
of change allowed. Set to a high number (eg 100) to'
                                                           ' ignore
max size of change. (default = 10)'
                                                           '')
    parser.add_argument('-s', '--screen', default=1, help='min count
of core to be considered for MMP identification.'
                                                          ' Higher
value reduces processing time and eliminates the '

'likelihood of identifying transformations with low MMPs count'
                                                          '(default
= 1)')

    args = parser.parse_args()


    # MMP identification settings
    max_size = int(args.change)
    ratio = float(args.ratio)
    min_core_num = int(args.screen)

    # connect to database
    db = re.search('\.db', args.output)
    if db is None:
        print('database name must end with .db')
        sys.exit(1)

    dbname = args.output
    conn = sqlite3.connect(dbname)
    c = conn.cursor()

    input_file = args.input # sample10000.csv

    try:
        mmpdb.tables.all_tables(c)
        print('tables created')
        input_smiles(input_file, conn)
        print('SMILES added')

        frag_df = fragment(c)
        print('SMILES fragmented')

        mmpdb.indexfrag.index_main(frag_df, conn)
        screen_cores(min_core_num, conn, c)
        print('indexing done')

        mmpdb.mmp_identification.identify_mmps(ratio, max_size,
conn, c)
        print('MMPs identified')
    finally:
        conn.commit()
        conn.close()

if __name__ == '__main__':
    main()
```

analysis.py

```python
import argparse
import sqlite3
import pandas as pd
from scipy import stats


# --------------------load data--------------------
def load_data(properties, conn):
    if properties == 'all':
        sql = 'SELECT * FROM mol_properties'
        data = pd.read_sql_query(sql, conn)
        data.drop(['smiles', 'fragmented', 'MMP_identified',
'cmpd_size'], axis=1, inplace=True)
    else:
        sql = 'SELECT mol_id'
        for prop in properties:
            sql = sql + ', ' + prop
        sql = sql + ' FROM mol_properties'  # add table selection
        data = pd.read_sql_query(sql, conn)
    return data

def load_mmp(conn):
    """returns a DF with all MMPs"""
    sql = 'SELECT trans_id, mol1_id, mol2_id FROM MMP'
    mmp = pd.read_sql_query(sql, conn)
    return mmp

def load_smirks(conn):
    sql = 'SELECT trans_id, SMIRKS FROM Transformation'
    smirks = pd.read_sql_query(sql, conn, index_col='trans_id')
    return smirks


# --------------------MMPA--------------------
def mmp_data(mmp, data):
    # rename property values
    prop_names = data.columns.tolist()
    prop_names.remove('mol_id')
    data_1 = data.copy()
    data_2 = data.copy()
    names_1 = {}
    names_2 = {}
    for name in prop_names:
        names_1[name] = name + '_1'
        names_2[name] = name + '_2'
    data_1.rename(columns=names_1, inplace=True)
    data_2.rename(columns=names_2, inplace=True)

    # do the merges
    mmp = mmp.merge(data_1, left_on='mol1_id', right_on='mol_id',
how='inner')
    mmp.drop('mol_id', axis=1, inplace=True)
    mmp = mmp.merge(data_2, left_on='mol2_id', right_on='mol_id',
how='inner')
    mmp.drop('mol_id', axis=1, inplace=True)

    # drop rows with missing values due to dataset limitations
    mmp.dropna(axis=1, how='any', inplace=True)
    return mmp, prop_names

def change(mmpa, prop_names):
    for name in prop_names:
        change = name +'_change'
        data_1 = name + '_1'
        data_2 = name + '_2'
        mmpa[change] = mmpa[data_2] - mmpa[data_1]

    return mmpa

def t_test(m1, s1, n1, m2, s2, n2):
    score = stats.ttest_ind_from_stats(m1, s1, n1, m2, s2, n2)
    return score.pvalue

def do_stats(mmpa, prop_names, min_count):
    all_stats = []
    for name in prop_names:
        change = name +'_change'
        data_1 = name + '_1'
        data_2 = name + '_2'
        headings = [change, data_1, data_2]

        # change data
        data_all = mmpa.groupby('trans_id')[headings]
        data_change = mmpa.groupby('trans_id')[change]
        mean_headings = {}
        std_dev_headings = {}
        for heading in headings:
            mean_headings[heading] = heading +'_mean'
            std_dev_headings[heading] = heading +'_std_dev'

        av = data_all.mean()
        av.rename(columns=mean_headings, inplace=True)

        std_dev = data_all.std()
        std_dev.rename(columns=std_dev_headings, inplace=True)

        med = data_change.median()
        med = med.to_frame(name + '_median')

        cnt = data_change.count()
        cnt = cnt.to_frame(name + '_count')

        std_err = data_change.sem()
        std_err = std_err.to_frame(name + '_sem')

        stats_data = pd.concat([av,std_dev, med, std_err, cnt],
axis=1) # see best way to combine these
        # either concat, or join or merge
        # concat should be fine since trans_id is the index

        # remove based on minimum count
        stats_data = stats_data[stats_data[name +
'_count']>min_count]

        # paired t test
        # ttest
        stats_data[name + '_ttest_pvalue'] = stats_data.apply(
            lambda x: t_test(x[data_1 + '_mean'], x[data_1 +
'_std_dev'], x[name + '_count'], x[data_2 +'_mean'], x[data_2 +
'_std_dev'], x[name + '_count']), axis=1)

        # drop for failed ones
        stats_data.dropna(axis=0, how='any', inplace=True)
        # remove useless columns now
        stats_data.drop([data_1 + '_mean', data_1 + '_std_dev',
data_2 + '_mean', data_2 + '_std_dev'], axis=1, inplace=True)
        stats_data[name + '_abs_mean'] = stats_data[change +
'_mean'].abs()
        all_stats.append(stats_data)
    mmp_stats = pd.concat(all_stats, axis=1)
    return mmp_stats

def drop_insignificant(mmp_data, prop_names, p_crit, drop_any):
    # so messy because the number/ names of columns not known
    drop = mmp_data[mmp_data[[name + '_ttest_pvalue' for name in
prop_names]] <=p_crit][[name + '_ttest_pvalue' for name in
prop_names]]

    if drop_any:
        drop.dropna(axis=0, how='any')
    else:
        drop.dropna(axis=0, how='all')

    #drop.drop(columns=[name + '_ttest_pvalue' for name in
prop_name])
    to_keep = drop.index.tolist()
    mmp_data = mmp_data[mmp_data.index.isin(to_keep)]
    return mmp_data


def prepare_mmn(mmpa, stats_data, prop_names, data):
    edges = mmpa.rename(columns={'mol1_id': 'Source', 'mol2_id':
'Target'})

    drop_col = [name +'_1' for name in prop_names] + [name +'_2' for
name in prop_names]
    edges.drop(columns=drop_col, inplace=True)
    edges = edges.merge(stats_data, how='inner', left_on='trans_id',
right_index=True)

    nodes = data.rename(columns={'mol_id': 'Id', 'smiles': 'Label'})

    edges.to_csv('edges.csv', index=False)
    nodes.to_csv('nodes.csv', index=False)

# main
def main():
    parser = argparse.ArgumentParser('''Carries out MMPA of desired
subset of the MMP database.
    Transformations with statistically significant results are
identified''')
    parser.add_argument('database', help='database name')
    parser.add_argument('-c', '--count', help='min number of MMPs
for a transformation to be considered, integer',
                        default=1, type=int)
    parser.add_argument('-t', '--t_crit', help=' statistical
significance level for a transformation to be considered using
paired t-test, '
```

```
                                               'default=no limit',
default=1)
    parser.add_argument('-n', '--network', help='create an output
file for MMN',
                            action='store_true')
    parser.add_argument('-p', '--property', nargs='*',help='list of
property/s to focus on. default=all)',
                       default='all')
    parser.add_argument('-s','--subset', help='perform analysis only
on subset. input txt file with SQL queries needed')
    parser.add_argument('-d', '--drop_any', help='a transformation
will be dropped where t-test p value for any of the properties is
below p_crit. No effect if MMPA of single variable.',
action='store_true')

    args = parser.parse_args()

    min_mmp = args.count
    p_crit = float(args.t_crit)
    dbname = args.database
    properties = args.property
    drop_any = args.drop_any
    subset = args.subset
    # !!!!!!!!!!!!!!!!!!!! move to  input file
    #sql_trans = '''SELECT trans_id FROM Transformation '''
    #sql_data = '''SELECT smiles_id, polymorph_count FROM
all_smiles, Solid_properties WHERE Solid_properties.refcode =
all_smiles.refcode GROUP BY all_smiles.refcode '''
    #sql_mol = '''SELECT smiles_id FROM all_smiles GROUP BY refcode
'''
    #sql_smirks = 'SELECT trans_id, SMIRKS FROM Transformation'


    # connect to database
    try:
        conn = sqlite3.connect(dbname)
        data = load_data(properties, conn)
        smirks = load_smirks(conn)
        mmp = load_mmp(conn)
        if subset:
            # trans of interest and mol of interest
            # add this at a later date
            pass
    finally:
        conn.close()

    # do analysis
    mmpa, prop_names = mmp_data(mmp,data) #property names are not
extracted from args.property in case of 'all' case, this way,
property data is extracted consistently
    mmpa = change(mmpa, prop_names)
    mmp_stats = do_stats(mmpa, prop_names, min_mmp)
    mmp_stats = drop_insignificant(mmp_stats, prop_names, p_crit,
drop_any)
    mmp_stats = mmp_stats.merge(smirks, how='left',
right_index=True, left_index=True)

    # generate output
    mmp_stats.to_csv('trans_data.csv')
    mmpa.to_csv('mmp_data.csv', index=False)
    if args.network:
        prepare_mmn(mmpa,mmp_stats,prop_names, data)

if __name__=='__main__':
    main()
```

csd_addon.py

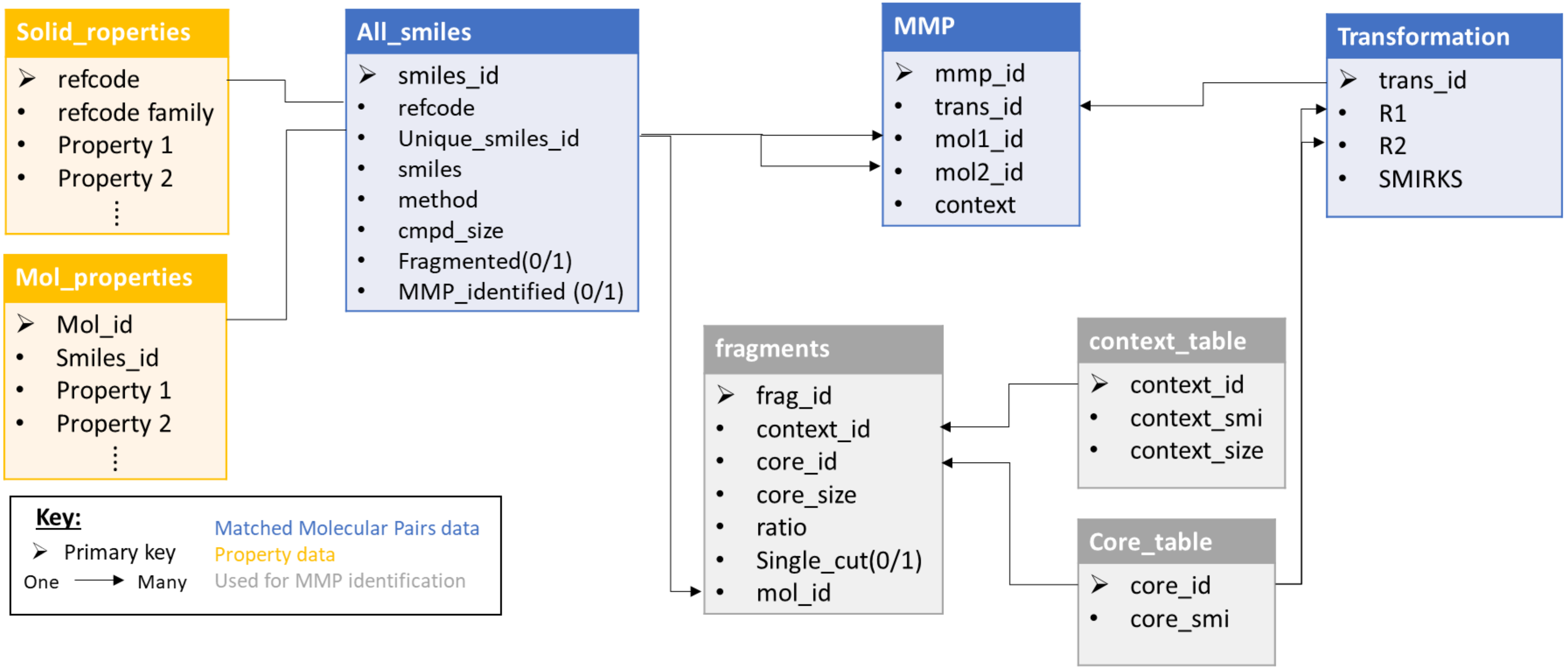
```python
import argparse
import os
import glob
import csv
import sys
from ccdc import search
from ccdc import io

try:
    from mmpdb import get_smiles
except ImportError:
    print('import of smiles_gen from mmpdb failed')
    sys.exit(1)

def csd_entry_to_smiles(refcode, csd_reader):
    try:
        entry = csd_reader.entry(refcode)
        crystal = csd_reader.crystal(refcode)
    except RuntimeError:
        return
    if crystal.has_disorder:
        return
    if entry.has_disorder:
        return

    smiles_all, method = get_smiles.generate_smiles(entry)
    smiles_list = smiles_all.split('.')
    rows = []
    for smi in smiles_list:
        rows.append([smi,refcode])
    return rows


def smiles_from_csd(csd_reader, settings):
    all_rows = []
    for e in csd_reader:
        if settings.test(e):
            try:
                ref = e.identifier
                rows = csd_entry_to_smiles(ref, csd_reader)
                all_rows = all_rows + rows
            except RuntimeError:
                continue
    return all_rows

def smiles_from_refcode(sourcefile, csd_reader):
    all_rows = []
    with open(sourcefile, 'r') as source:
        for line in source:
            ref_code = line.rstrip()
            rows = csd_entry_to_smiles(ref_code, csd_reader)
            all_rows = all_rows + rows
    return all_rows


def write_output(smiles, output_file):
    with open(output_file, 'wb') as csv_file:
        writer = csv.writer(csv_file,delimiter=',')
        writer.writerows(smiles)
```

```
def main():
    parser = argparse.ArgumentParser('''Generates SMILES from CSD
entries based on
    the refcodes in the input file or CSD search. Outputs file with
smiles for MMP analysis''')

    parser.add_argument('input', help='input text file with refcodes
or type "CSD" to do a search')
    parser.add_argument('-o', '--output', default='smiles.csv',
help='output name (default = smiles.csv)')
    parser.add_argument('-d', '--directory',
default=io.csd_directory(), help='directory of the CSD-like database
(default=CSD)')
    args = parser.parse_args()

    # setup input source
    useCSD = False
    if args.input == 'CSD':
        useCSD = True
    else:
        input_file = args.input

    #setup output
    output_file = args.output

    # CSD search settings
    csd_dir = args.directory
    csd_location = glob.glob(os.path.join(csd_dir, '*.inf'))
    csd_reader = io.EntryReader(csd_location)

    if useCSD:
        settings = search.Search.Settings()
        settings.has_3d_coordinates = True
        settings.only_organic = True
        settings.not_polymeric = True
        settings.no_powder = True
        settings.no_disorder = True
        settings.max_r_factor = 7.5
        settings.no_metals = True
        settings.must_not_have_elements = ['As', 'Te', 'At', 'He',
'Ne', 'Ar', 'Kr', 'Xe', 'Rn', 'B', 'Al', 'Ga', 'In',
                                           'Tl', 'Si', 'Ge', 'Sn',
'Pb', 'Sb', 'Po']
    if useCSD:
        smiles = smiles_from_csd(csd_reader, settings)
    else:
        smiles = smiles_from_refcode(input_file, csd_reader)

    write_output(smiles, output_file)

if __name__ == '__main__':
    main()
```

Appendix 2
Matched Molecular Database Schema

Purpose:

A4 size version of the Matched Molecular Pairs Database


Solid_roperties
refcode
refcode family
Property 1
Property 2
⋮
Mol_properties
Mol_id
Smiles_id
Property 1
Property 2
⋮
All_smiles
smiles_id
refcode
Unique_smiles_id
smiles
method
cmpd_size
Fragmented(0/1)
MMP_identified (0/1)
MMP
mmp_id
trans_id
mol1_id
mol2_id
context
Transformation
trans_id
R1
R2
SMIRKS
fragments
frag_id
context_id
core_id
core_size
ratio
Single_cut(0/1)
mol_id
context_table
context_id
context_smi
context_size
Core_table
core_id
core_smi
Key:
Primary key
One
Many
Matched Molecular Pairs data
Property data
Used for MMP identification

Appendix 3
Polymorph and Redetermination Classification

Purpose:

This appendix contains scripts used prepare datasets for the benchmark study (Chapter 5) and the training of machine learning based models.

Folder structure and uses:

- **pre_process.py**: prepares datasets for the study
- **train.py**: carries out the training process on the datasets

pre_process.py

```
import pandas as pd
import numpy as np
import glob
import os
import re
from collections import defaultdict
import itertools
import json
import pickle
from sklearn.model_selection import train_test_split

from ccdc import io
from ccdc import crystal


class SpectraMethod:
    def __init__(self, cluster_f_name='CSDplus_clusters_s.txt'):
        with open(cluster_f_name, 'r') as f:
            lines = f.readlines()

        self.clusters = defaultdict(list)
        self.all_refs = []
        self.ref_groups = defaultdict(list)
        for line in lines:
            cluster = line.split()
            self.all_refs.extend(cluster)
            fam = re.sub('[0-9]+', '', cluster[0])
            self.clusters[fam].append(cluster)
            self.ref_groups[fam].extend(cluster)

    @staticmethod
    def get_polymorph_id(ref, cluster):
        id = None
        for i, pol in enumerate(cluster):
            if ref in pol:
                id = i
        return id

    def check_polymorphism(self, pair, clusters=None):
        ref1, ref2 = pair
        if clusters is None:
            clusters = self.clusters
        pol = None
        fam1 = re.sub('[0-9]+', '', ref1)

        cluster = clusters.get(fam1)
        if cluster:
            id_1 = self.get_polymorph_id(ref1, cluster)
            id_2 = self.get_polymorph_id(ref2, cluster)
            if id_1 is None or id_2 is None:
                print('refoces not in cluster {} {}'.format(ref1,
ref2))
            else:
                if id_1 == id_2:
                    pol = 0
                else:
                    pol = 1
        else:
            print('refcode fam {} not in clusters'.format(fam1))
        return pol


class ManualMethod:
    def __init__(self, all_refs, info=None, csd_reader=None):
        self.all_refs = all_refs

        if info is None:
            assert csd_reader is not None, 'if info is not provided,
csd_reader needed'
            info = self.get_info(csd_reader)
        self.info = info

    def get_info(self, csd_reader):
        info = {}
        for ref in self.all_refs:
            try:
                e = csd_reader.entry(ref)
                lit = e.publication
                pol = e.polymorph
                info[ref] = {'lit': lit, 'polymorph': pol}
            except RuntimeError:
                continue
        return info

    def check_polymorphism(self, pair):
        ref1, ref2 = pair
        info1, info2 = self.info.get(ref1), self.info.get(ref2)
        if info1 is not None and info2 is not None:
            pol1, pol2 = info1['polymorph'], info2['polymorph']
            if pol1 is None or pol2 is None:
                pol = None
            elif pol1 == pol2:
                pol = 0
            else:
                pol = 1

        return pol

    def check_lit_source(self, pair):
        ref1, ref2 = pair
        same = False
        if self.info.get(ref1) is not None and self.info.get(ref2)
is not None:
            if self.info[ref1]['lit'] == self.info[ref2]['lit']:
                same = True
        return same


class BestRMethod:
    def __init__(self, all_refs, best_r_file):
        with open(best_r_file, 'r') as f:
            lines = f.readlines()
        self.in_best_r = defaultdict(list)
        for ref in lines:
            ref = ref.rstrip()
            if ref in all_refs:
                fam = re.sub('[0-9]+', '', ref)
                self.in_best_r[fam].append(ref)

    def check_polymorphism(self, pair):
        ref1, ref2 = pair
        pol = None
        fam = re.sub('[0-9]+', '', ref1)
        cluster = self.in_best_r[fam]
        if ref1 in cluster and ref2 in cluster:
            pol = 1
        else:
            if len(cluster) == 1:
                pol = 0

        return pol


class Datasets:
    def __init__(self, ref_groups, all_refs=None):
        self.ref_groups = ref_groups
        if all_refs is None:
            all_refs = []
            for _, refs in ref_groups.iteritems():
                all_refs.extend(refs)
        self.refs = all_refs

        # get stuff for packing similirity
        self.packing_sim = crystal.PackingSimilarity()
        self.packing_shell_size =
self.packing_sim.settings.packing_shell_size

        self.combs = self.get_combinations()

        self.crystal_data = None

    def get_combinations(self):
        all_combs = []
        for _, refs in self.ref_groups.iteritems():
            comb = list(itertools.combinations(refs, 2))
            all_combs.append(comb)
        return all_combs

    def get_structure_data(self, reader):
        id_list = self.refs
        crystal_data = {}
        total = len(id_list)
        step = int(round(float(total) / 50))
        for i, ref in enumerate(id_list):
            if i % step == 0:
                print('data obtained for {} % of
structures'.format(round(float(i + 1) / total, 2) * 100))
            try:
                cryst = reader.crystal(ref)
                entry = reader.entry(ref)
            except RuntimeError:
                print('failed to access data for {}'.format(ref))
                continue
            # temperature
            T = entry.temperature
            if T:
                T_re = re.search('(?P<temp>-?[0-9]+(\.[0-
9])?)\s?(?P<units>\S+)', T)
                if T_re:
                    if T_re.group('units') != 'K':  # when deg.C
                        T_num = float(T_re.group('temp')) + 273.2
                    else:
                        T_num = float(T_re.group('temp'))
                else:
                    print(T)
            else:
                T_num = np.nan

            crystal_data[ref] = {'length_a': cryst.cell_lengths[0],
'length_b': cryst.cell_lengths[1],
                                 'length_c': cryst.cell_lengths[2],
'angle_a': cryst.cell_angles[0],
                                 'angle_b': cryst.cell_angles[1],
'angle_g': cryst.cell_angles[2],
                                 'r_factor': entry.r_factor,
'crystal_system': cryst.crystal_system,
                                 'temperature': T_num}

        self.crystal_data = crystal_data

    def get_descriptors(self, csd_reader, classify_func,
do_rmsd=True, check_lit=None):
        """classify_func is a dict with name (string) as key and
classifying function as value"""
        crystal_data = self.crystal_data
        packing_comp = self.packing_sim
        packing_shell_size = self.packing_shell_size

        def check_fam(pair):
            assert pair is not None, 'empty pair found'
            assert len(pair)==2, 'missing refcode with
{}'.format(pair)
            ref1, ref2 = pair
            fam1, fam2 = re.sub('[0-9]+', '', ref1), re.sub('[0-
9]+', '', ref2)
            assert fam1 == fam2, 'refcode family dont match. {},
{}'.format(ref1,ref2)

        def compare_structures(refs):
            # handle missing data
            if not all(_ in crystal_data.keys() for _ in refs):
                return None

            ref1 = refs[0]
            ref2 = refs[1]

            data1 = crystal_data[ref1]
            data2 = crystal_data[ref2]
            cryst1 = {'x': (data1['length_a'], data1['angle_a']),
'y': (data1['length_b'], data1['angle_b']),
                      'z': (data1['length_c'], data1['angle_g'])}
            cryst2 = {'x': (data2['length_a'], data2['angle_a']),
'y': (data2['length_b'], data2['angle_b']),
                      'z': (data2['length_c'], data2['angle_g'])}

            change = {}
            # cell parameters
            for axis in ['x', 'y', 'z']:
                dl = abs(cryst1[axis][0] - cryst2[axis][0])  #
change in length
                da = abs(cryst1[axis][1] - cryst2[axis][1])  #
change in angle
```

```
                change['length_{}'.format(axis)] = dl
                change['angle_{}'.format(axis)] = da

            for prop_name in ['r_factor', 'temperature']:
                if data1[prop_name] is not None and data2[prop_name]
is not None:
                    prop = abs(data1[prop_name] - data2[prop_name])
                else:
                    prop = np.nan
                change[prop_name] = prop

            # RMSD
            if do_rmsd:

                try:
                    packing_sim = packing_comp  # had insane RAM
usage for some reason, this seems to solve it
                    similarity =
packing_sim.compare(csd_reader.crystal(ref1),
csd_reader.crystal(ref2))
                    if similarity:
                        #rmsd = similarity.rmsd
                        # its not actually rmsd, its the number of
molecules that match within the tolerance
                        rmsd = similarity.nmatched_molecules
                        rmsd = float(rmsd) / packing_shell_size

                    else:
                        rmsd = np.nan
                except RuntimeError:
                    rmsd = np.nan
                change['rmsd'] = rmsd

            # crystal system
            if data1['crystal_system'] == data2['crystal_system']:
                cryst_sys = 0
            else:
                cryst_sys = 1
            change['crystal_system'] = cryst_sys

            change['refs'] = (ref1, ref2)
            return change

        data = []
        total = len(self.combs)
        step = int(round(float(total) / 20))
        for i, comb in enumerate(self.combs):
            if i % step == 0:
                print('comparison done for {} % of the
data'.format(int(round(float(i + 1) / total, 2) * 100)))
            for pair in comb:
                try:
                    check_fam(pair)
                except AssertionError, e:
                    print(e)
                    continue
                change = compare_structures(pair)
                if change:
                    for name, classifier in
classify_func.iteritems():
                        change[name] = classifier(pair)
                    if check_lit:
                        change['lit'] = check_lit(pair)
                    data.append(change)
                    # append to data
                    # create dataframe

        data_df = pd.DataFrame(data)
        return data_df



csd_dir = io.csd_directory()
csd_location = glob.glob(os.path.join(csd_dir, '*.inf'))
csd_reader = io.EntryReader(csd_location)

spectra = SpectraMethod()
all_refs = spectra.all_refs
ref_groups = spectra.ref_groups

manual = ManualMethod(all_refs, csd_reader=csd_reader)

best_r = BestRMethod(all_refs, r'C:\Program Files
(x86)\CCDC\CSD_2018\CSD_539\subsets\best_R_factor_list.gcd')

classify_func = {'spectra': spectra.check_polymorphism, 'manual':
manual.check_polymorphism,
                 'best_R': best_r.check_polymorphism}

datasets = Datasets(ref_groups, all_refs)
datasets.get_structure_data(csd_reader)
data = datasets.get_descriptors(csd_reader, classify_func,
check_lit=manual.check_lit_source, do_rmsd=True)

data.to_csv('new_data_backup.csv', index=False)


# split the datasets

len(data[~data['manual'].isnull()])

len(data[(~data['manual'].isnull())&(~data['best_R'].isnull())])

train_size = 24660
valid_size = 2594
test_size = 3415

spectra_train_pol = data[(~data['best_R'].isnull()) &
(data['manual'].isnull()) & (data['best_R']==1)]
spectra_train_red = data[(~data['best_R'].isnull()) &
(data['manual'].isnull()) & (data['best_R']==0)].sample(n=
train_size - len(spectra_train_pol))
spectra_train = pd.concat([spectra_train_pol, spectra_train_red])


manual_train = data[(data['best_R'].isnull()) &
(~data['manual'].isnull())]

manual_valid = data[(~data['best_R'].isnull()) &
(~data['manual'].isnull()) & (data['lit'] ==
False)].sample(n=valid_size)
```

```
spectra_valid = data[(~data['best_R'].isnull()) &
(~data['manual'].isnull()) & (data['lit'] == False) &
(~data.index.isin(manual_valid.index))]
assert len(spectra_valid) == valid_size

benchmark_valid = data[(~data['best_R'].isnull()) &
(~data['manual'].isnull()) & (data['lit'] ==
True)].sample(n=valid_size)

benchmark_test = data[(~data['best_R'].isnull()) &
(~data['manual'].isnull()) & (data['lit'] == True) &
(~data.index.isin(benchmark_valid.index))]
assert len(benchmark_test) == test_size

sets = {
    'manual_train': manual_train,
    'manual_valid': manual_valid,
    'spectra_train': spectra_train,
    'spectra_valid': spectra_valid,
    'benchmark_valid': benchmark_valid,
    'benchmark_test': benchmark_test
}


for set_name in sets.keys():
    sets[set_name]['ref1'], sets[set_name]['ref2'] =
zip(*sets[set_name]['refs'])


for set_name, table in sets.iteritems():
    f_name = set_name + '.csv'
    table.to_csv(f_name, index=False)
```

train.py

```
import argparse
import itertools
import sys
import glob
import os
import re
import numpy as np
import pandas as pd
import json
import pickle
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics
#from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import RandomizedSearchCV
#from sklearn.model_selection import StratifiedShuffleSplit
#from sklearn.model_selection import GridSearchCV

from ccdc import io
from ccdc import crystal

train_settings = {
    'retrain': False,
    'all_refs': 'single-red-powder.txt',
    'best_r': 'single-powder.txt',
    'fill_na': 'drop',
    'verbose': False,
    'csd_dir': None,
    'do_rmsd': False,
    'valid_frac': 0.1,
    'test_frac': 0.1
    }


def fix_datasets(df):
    df['refs'] = df[['ref1', 'ref2']].apply(lambda row:
(row['ref1'], row['ref2']), axis=1)
    cols = ['angle_x', 'angle_y', 'angle_z', 'crystal_system',
'length_x',
'length_y','length_z','r_factor','refs','rmsd','target','temperature
']
    return df[cols].copy()


class PolymorphClassifier:
    def __init__(self, train, valid, test, nan_method='drop',
use_rmsd=True):

        cols = list(train.columns)
        cols.remove('target')
        if not use_rmsd:
            if 'rmsd' in cols:

                cols.remove('rmsd')
        else:
            if 'rmsd' not in cols:
                print('rmsd not calculated')
        if 'refs' in list(train.columns):
            cols.remove('refs')
            # set indexes as ref1-ref2 so it stays with the rows
        datasets = {'train': train, 'valid': valid, 'test': test}
        datasets = self.handle_nans(datasets, method=nan_method)

        self.train_with_refs = datasets['train']
        self.valid_with_refs = datasets['valid']
        self.test_with_refs = datasets['test']
        for name, dataset in datasets.iteritems():
            dataset.index = dataset['refs'].apply(lambda row: '{}-
{}'.format(row[0], row[1]))
        # split into X, Y
        Xs = {}
        Ys = {}
        for name, dataset in datasets.iteritems():
            Xs[name] = dataset[cols].copy()
            Ys[name] = dataset['target']

        # Y = train['target']
        # X = train.loc[:, train.columns != 'target']
        # X = X[cols].copy()

        # Y_valid = valid['target']
        # X_valid = valid.loc[:, valid.columns != 'target']
        # X_valid = X_valid[cols].copy()

        self.X = Xs['train']
        self.Y = Ys['train']
        self.Y_valid = Ys['valid']
        self.X_valid = Xs['valid']
        self.Y_test = Ys['test']
        self.X_test = Xs['test']

        self.classifiers = {'RF': RandomForestClassifier,
'logistic_regression': LogisticRegression,
                            'KNN': KNeighborsClassifier, 'bayes':
GaussianNB, 'SVM': SVC}


    @staticmethod
    def handle_nans(datasets, method='drop'):
        # dict with inputs {'train':train, 'valid':valid}
        processed = {}
        for name, dataset in datasets.iteritems():
            if method == 'drop':
                processed[name] = dataset.dropna(axis=0, how='any')

            elif method == 'mean':
                processed[name] =
dataset.fillna(datasets['train'].mean())

            elif method == 'median':
                processed[name] =
dataset.fillna(datasets['train'].median())

            else:
                print('choose one of: drop, mean, median')
        return processed
    @staticmethod
    def ref_as_index(datasets):
        pass

    def fit_models(self):
        models = {}
        X, Y = self.X, self.Y

        for name, model in self.classifiers.iteritems():
            print('fitting {}'.format(name))
            models[name] = model().fit(X ,Y)

        rows = []
        for name, model in models.iteritems():
            print('testing {}'.format(name))

            pred = pd.DataFrame(model.predict(self.X_valid))
            pred.index = self.Y_valid.index
            C = metrics.confusion_matrix(self.Y_valid, pred)

            true_pos_rate = float(C[1][1]) / (C[1][1] + C[0][1])  # 
sensitivity, recall = TP/(TP+NF)
            false_pos_rate = float(C[1][0]) / (C[1][0] + C[0][0])  # 
FP/(FP+TN)
            # specificity = 1 - false_pos_rate
            pos_pred_value = float(C[1][1]) / (C[1][1] + C[1][0])  # 
precision = TP/(TP+FP)

            row = {'model': name, 'precision': pos_pred_value, 
'recall': true_pos_rate,
                   'specificity': 1 - false_pos_rate,
                   'F1_score': 2 * (pos_pred_value * true_pos_rate) 
/ (pos_pred_value + true_pos_rate)
                   }
            rows.append(row)
        models_summary = pd.DataFrame(rows)
        # self.models_summary = models_summary
        return models_summary

    def optimise_random_forest(self, n_iter=100, scoring='f1'):
        n_estimators = [int(x) for x in np.linspace(start=200, 
stop=2000, num=10)]
        criterions = ['gini', 'entropy']
        max_features = ['sqrt', 'log2']
        max_depth = [int(x) for x in np.linspace(10, 110, num=11)]
        max_depth.append(None)

        min_samples_split = [2, 5, 10]
        min_samples_leaf = [1, 2, 4]
        bootstrap = [True, False]
        random_grid = {'n_estimators': n_estimators,
                       'max_features': max_features,
                       'max_depth': max_depth,
                       'min_samples_split': min_samples_split,
                       'min_samples_leaf': min_samples_leaf,
                       'bootstrap': bootstrap,
                       'criterion': criterions}
        rf = RandomForestClassifier()
        rf_random = RandomizedSearchCV(estimator=rf, 
param_distributions=random_grid, n_iter=n_iter, cv=3,
                                       random_state=42, n_jobs=-1,
scoring=scoring)
        # Fit the random search model
        rf_random.fit(self.X, self.Y)
        return rf_random

    def optimise_svm(self, n_iter=100, scoring='f1'):
        C_range = np.logspace(-4, 3, 15)
        gamma_range = np.logspace(-4, 3, 15)
        random_grid = {'gamma': gamma_range, 'C': C_range}
        svm = SVC()
        # cv = StratifiedShuffleSplit(n_splits=5, test_size=0.2,
random_state=42)
        # svm_random = GridSearchCV(SVC(), param_grid=param_grid,
cv=cv)
        svm_random = RandomizedSearchCV(estimator=svm,
param_distributions=random_grid, n_iter=n_iter, cv=3,
                                        random_state=42, n_jobs=2,
scoring=scoring)
        svm_random.fit(self.X, self.Y)

        return svm_random

    def compare_performance(self, params_dicts):
        rows = []
        for name, param in params_dicts.iteritems():
            model =
self.classifiers[name](**param).fit(self.X_valid, self.Y_valid)
            pred = pd.DataFrame(model.predict(self.X_valid))
            C = metrics.confusion_matrix(self.Y_valid, pred)
            if train_settings['verbose']:

pd.DataFrame(C).to_csv('{}_confusion_valid.csv'.format(name))

            true_pos_rate = float(C[1][1]) / (C[1][1] + C[0][1])  #
sensitivity, recall = TP/(TP+NF)
            false_pos_rate = float(C[1][0]) / (C[1][0] + C[0][0])  #
FP/(FP+TN)
            # specificity = 1 - false_pos_rate
            pos_pred_value = float(C[1][1]) / (C[1][1] + C[1][0])  #
precision = TP/(TP+FP)

            row = {'model': name, 'precision': pos_pred_value,
'recall': true_pos_rate,
                   'specificity': 1 - false_pos_rate,
                   'F1_score': 2 * (pos_pred_value * true_pos_rate)
/ (pos_pred_value + true_pos_rate)
                   }
            rows.append(row)
        models_summary = pd.DataFrame(rows)
        if train_settings['verbose']:
            models_summary.to_csv('comparison_valid.csv',
index=False)
        # select best F1 score model
        best_model = models_summary[models_summary['F1_score'] ==
models_summary['F1_score'].max()]
        print('best performing model:
{}'.format(best_model['model'].tolist()[0]))
        print('F1 score:
{}'.format(best_model['F1_score'].tolist()[0]))
        return best_model['model'].tolist()[0]
    def retrain(self, name, params, test=True):
        """retrains the model on train + valid or train + valid +
train (for application)"""
        if test:
            X = pd.concat([self.X, self.X_valid])
            Y = pd.concat([self.Y, self.Y_valid])
        else:
            X = pd.concat([self.X, self.X_valid, self.X_test])
            Y = pd.concat([self.Y, self.Y_valid, self.Y_test])

        model = self.classifiers[name](**params).fit(X, Y)

        if test:
            Y_test = self.Y_test
            X_test = self.X_test
            pred = pd.DataFrame(model.predict(X_test),
columns=['predicted'])
            pred.index = Y_test.index
            C = metrics.confusion_matrix(Y_test, pred)
            if train_settings['verbose']:

pd.DataFrame(C).to_csv('{}_confusion_test.csv'.format(name))
                Y_test = Y_test.to_frame('actual')
                compare = Y_test.merge(pred, right_index=True,
left_index=True)
                compare['refs'] = compare.index
                compare['ref1'], compare['ref2'] =
zip(*compare['refs'].str.split('-'))
                compare = compare.merge(X_test, right_index=True,
left_index=True)
                cols = list(X_test.columns)
                cols += ['ref1', 'ref2', 'predicted', 'actual']
                compare[cols].to_csv('comparison.csv', index=False)

            true_pos_rate = float(C[1][1]) / (C[1][1] + C[0][1])  #
sensitivity, recall = TP/(TP+NF)
            false_pos_rate = float(C[1][0]) / (C[1][0] + C[0][0])  #
FP/(FP+TN)
            # specificity = 1 - false_pos_rate
            pos_pred_value = float(C[1][1]) / (C[1][1] + C[1][0])  #
precision = TP/(TP+FP)

            performance = {'model': name, 'precision':
pos_pred_value, 'recall': true_pos_rate,
                           'specificity': 1 - false_pos_rate,
                           'F1_score': 2 * (pos_pred_value *
true_pos_rate) / (pos_pred_value + true_pos_rate)
                           }

            return performance
        else:
            return model


def train():
    if not all(dataset in train_settings.keys() for dataset in
['train', 'valid', 'test']):
        # get datasets
        preprocess =
Preprocess(train_settings['all_refs'],train_settings['best_r'])
```

```
        # connect to CSD
        if train_settings['csd_dir'] is None:
            csd_dir = io.csd_directory()
        else:
            csd_dir = train_settings['csd_dir']
        csd_location = glob.glob(os.path.join(csd_dir, '*.inf'))
        csd_reader = io.EntryReader(csd_location)
        preprocess.get_structure_data(csd_reader)

preprocess.get_datasets(csd_reader,train_settings['do_rmsd'])
        train, valid, test =
preprocess.train_valid_test_split(train_settings['valid_frac'],
train_settings['test_frac'])

        train.to_csv('train.csv')
        valid.to_csv('valid.csv')
        test.to_csv('test.csv')

    else:
        #train = json.load(open(train_settings['train'], 'r'))
        train = pd.read_csv(train_settings['train'])
        train = fix_datasets(train)
        #valid = json.load(open(train_settings['valid'], 'r'))
        valid = pd.read_csv(train_settings['valid'])
        valid = fix_datasets(valid)
        #test = json.load(open(train_settings['test'], 'r'))
        test = pd.read_csv(train_settings['test'])
        test = fix_datasets(test)
    classifier = PolymorphClassifier(train,valid, test,
nan_method=train_settings['fill_na'],
use_rmsd=train_settings['do_rmsd'])
    model_summary = classifier.fit_models()
    model_summary = model_summary.sort_values(by='F1_score',
ascending=False).reset_index(drop=True)
    if train_settings['verbose']:
        model_summary.to_csv('model_summary.csv', index=False)
    best_models =
(model_summary['model'][0],model_summary['model'][1])
    opti_params = {}
    for model in best_models:
        if model == 'SVM':
            svm_random = classifier.optimise_svm()
            svm_summary = pd.DataFrame(svm_random.cv_results_)
            if train_settings['verbose']:
                svm_summary.to_csv('SVM_optimisation.csv',
index=False)
            svm_params =
svm_summary.sort_values(by='mean_test_score',ascending=False).reset_
index(drop=True)['params'][0]
            print('SVM classifier optimised with params:')
            for k,v in svm_params.iteritems():
                print('{}: {}'.format(k,v))
            opti_params['SVM'] = svm_params

        elif model == 'RF':
            rf_random = classifier.optimise_random_forest()
            rf_summary = pd.DataFrame(rf_random.cv_results_)
            if train_settings['verbose']:
                rf_summary.to_csv('RF_optimisation.csv',
index=False)
```

```
            rf_params = 
rf_summary.sort_values(by='mean_test_score',ascending=False).reset_i
ndex(drop=True)['params'][0]
            print('RF classifier optimised with params:')
            for k,v in rf_params.iteritems():
                print('{}: {}'.format(k,v))
            opti_params['RF'] = rf_params


        else:
            print('cant optimise {} algorithm at the 
moment'.format(model))
            sys.exit(1)
    best = classifier.compare_performance(opti_params)

    performance = classifier.retrain(best, opti_params[best])

    model = classifier.retrain(best, opti_params[best], test=False)

    pickle.dump(model, open('polymorph_classifier.p','w'))

json.dump(opti_params[best],open('{}_hyper_parameters.json'.format(b
est),'w'))

    return model


def main():
    # global hydrates
    # TODO: sort these out
    parser = argparse.ArgumentParser('''Trains polymorph 
redetermination classifier on CSD dataset''')
    parser.add_argument('--all_refs', help='file with all refcodes')
    parser.add_argument('--best_r', help='file with best R factor 
structures')
    parser.add_argument('--train', help='training dataset')
    parser.add_argument('--valid', help='validation dataset')
    parser.add_argument('--test', help='test dataset')
    parser.add_argument('--retrain', action='store_true', help='To 
retrain best performing algorithm before saving to a pickle') 
#remove this, always retrained
    parser.add_argument('--fill_na', help='strategy for handling 
missing values. [mean, median, drop] default=drop')
    parser.add_argument('--verbose', action='store_true')
    parser.add_argument('--csd_dir', help='CSD directory')
    parser.add_argument('--do_rmsd', action='store_true', help='do 
RMSD comparison as one of the descriptors')
    parser.add_argument('--valid_frac', type=float, help='validation 
set fraction')
    parser.add_argument('--test_frac', type=float, help='validation 
set fraction')
    args = parser.parse_args()

    input_settings = vars(args)
    for key, setting in input_settings.iteritems():
        if setting is not None:
            if key == 'fill_na':
                if setting not in ['mean', 'median', 'drop']:
                    print('WARNING: invalid fill_na strategy 
selected. drop used')
                    continue
            train_settings[key] = setting
    model = train()


if __name__ == '__main__':
    main()
```

Appendix 4 Message Passing Neural Network scripts

Purpose:

This appendix contains scripts used in Chapter 7 to train and hyperparameter optimise Message Passing Neural Networks for melting point prediciton. The pre-processing scripts which generate graph input and calculate crystal features are shown under Pre-processing. The modified tensorflow models are included. The hyperparameter optimisation script is available as well.

Files and uses:

Pre-processing (local):

- **crystal_graph.py:** takes csv file (Refcode, melting point) and generates a graph input
- **crystal_rmsd.py:** calculates the shape change descriptor

Training (HPC)

- Tensorflow models
  The following scripts were adapted from GGNN. (See Chapter 7 for details) The original scripts by Microsoft available under MIT license (relevant text on the next page).
    - **util.py**: utility functions
    - **chem_tensorflow.py**: base tensorflow model
    - **chem_tensorflow_sparse.py**: specific tensorflow model for training
    - **apply_chem_tensorflow.py**: tensorflow model, for testing
- Training scripts and hyperparamter optimisation
    - **optimiser.py**: hyperparameter optimisation script

The following license terms apply to **util.py , chem_tensorflow.py, chem_tensorflow_sparse.py,** and **apply_chem_tensorflow.py.**

crystal_graph.py

```
# Python 2
import pandas as pd
import glob
import os
import sys
import json
import csv
import argparse
import random
import numpy as np
from collections import defaultdict

from ccdc import io

from crystal_rmsd import CrystalRMSD
import HBond_Dimensionality as HBond


def read_csv():
    # use pandas df.to_dict('record')
    pass


def from_csv(file_name):
    raw = []
    with open(file_name, 'r') as csvfile:
        reader = csv.reader(csvfile, delimiter=',')
        for row in reader:
            if row[1] == '':
                continue
            raw.append({'refcode':row[0], 'y':float(row[1])})
    return raw


def split_data(raw, valid_frac, normalise=True):
    raw_data = {'train':[], 'valid':[]}

    size = len(raw)

    #-> n random numbers within range(len(raw_data))
    valid = random.sample(range(size), int(round(size*valid_frac)))
    only_y = []
    print('splitting data')
    for i, data in enumerate(raw):
        if i % 1000 == 0:
            done = (float(i)/size)*100
            print('done: {} % '.format(round(done, 2)))
        if normalise:
            only_y.append(data['y'])
        #process
        if i not in valid:
            raw_data['train'].append(data)
        else:
            raw_data['valid'].append(data)
        # get std_dev and mean of y (data[1])
    if normalise:
        mean = np.mean(only_y)
        std = np.std(only_y)
    else:
        mean, std = None, None
    return raw_data, mean, std


def onehot(feature, feature_vector):
    z = [0 for _ in range(len(feature_vector))]
    z[feature_vector.index(feature)] = 1
    return z


class CrystalGraph:
    def __init__(self, csd_reader, atom_list=None, mean=None,
std=None, crystal_rmsd=None, h_dims=False, vwd=True):
        if atom_list:
            self.atom_list = atom_list
            self.get_elements = False
        else:
            self.atom_list = []
            self.get_elements = True

        self.csd_reader = csd_reader

        self.mean = mean
        self.std = std
        if std is None and mean is None:
            self.to_normalise = False
        else:
            self.to_normalise = True
        self.graphs = defaultdict(list)
        self.bond_dict = {'SINGLE': 1, 'DOUBLE': 2, 'TRIPLE': 3,
"AROMATIC": 4, "HBOND":5, "VDW_INTER":6, "VDW_INTRA": 7}
        self.h_dim_features = ['Ring/enclosed', 'Chain (1D)', 'Sheet
(2D)', 'Lattice (3D)']
        self.skip_refcode = [] # list of refcodes that were trouble
        self.rmsd = crystal_rmsd
        self.do_h_dims = h_dims
        self.do_vwd = vwd
        self.atom_counts = {}

    def normalise(self, y):
        return (y - self.mean) / self.std

    def index_atoms(self, mol):
        label_to_index = {}
        index_to_label = {}
        nodes = []
        elements_list = []
        for i, atom in enumerate(mol.atoms):
            label_to_index[atom.label] = i
            index_to_label[i] = atom.label

            if self.get_elements:
                nodes.append(atom.atomic_symbol)
                elements_list.append(atom.atomic_symbol)
                self.atom_list = list(set(self.atom_list +
elements_list)) # add unique atoms to the list
            else:
                nodes.append(onehot(atom.atomic_symbol,
self.atom_list)) # do one hot already if atom list available
        return nodes, label_to_index, index_to_label

    def get_bonds(self, mol, label_to_index):
        edges = []
        for bond in mol.bonds:
            atom1, atom2 = bond.atoms
            edge_type = str(bond.bond_type).upper()

            edge = [label_to_index[atom1.label],
self.bond_dict[edge_type], label_to_index[atom2.label]]

            edges.append(edge)
        return edges

    def get_h_bonds(self, cryst, label_to_index):
        edges = []
        hbonded = {}  # so they can be eliminated from VDW
        for hbond in cryst.hbonds():
            donor = hbond.atoms[0].label
            acceptor = hbond.atoms[2].label
            edge = [label_to_index[donor],
self.bond_dict['hbond'.upper()], label_to_index[acceptor]]
            edges.append(edge)

            for atom in hbond.atoms:
                all_atoms = list(hbond.atoms)
                all_atoms.remove(atom)
                hbonded[atom.label] = [a.label for a in all_atoms]
        return edges, hbonded

    def get_vdw(self, cryst, label_to_index, bonded):  # only
intermolecular interactions
        edges = []

        for contact in cryst.contacts():

            if contact.atoms[0].label in bonded.keys():
                if contact.atoms[1].label in
bonded[contact.atoms[0].label]:  # already covered in H-bond
                    continue

            # could add a bit that would eliminate some based on
contact.strength

            if contact.intermolecular:
                edge_type = 'VDW_INTER'
            else:
                edge_type = 'VDW_INTRA'

            edge = [label_to_index[contact.atoms[0].label],
self.bond_dict[edge_type],
                    label_to_index[contact.atoms[1].label]]
            edges.append(edge)


        return edges

    def get_graphs(self, refcode):
        mol = self.csd_reader.molecule(refcode)
        cryst = self.csd_reader.crystal(refcode)
        nodes, label_to_index, index_to_label =
self.index_atoms(mol)
        try:
            bonds = self.get_bonds(mol, label_to_index)
            hbonds, interaction = self.get_h_bonds(cryst,
label_to_index)
            if self.do_vwd:
                vdw = self.get_vdw(cryst, label_to_index,
interaction)
                edges = bonds + hbonds + vdw
            else:
                edges = bonds + hbonds
        except KeyError:
            print('something wrong with labels with:
{}'.format(refcode))
            self.skip_refcode.append(refcode)
            return None, None
        return nodes, edges

    def get_crystal_properties(self, refcode):
        graph_features = {}
        if self.rmsd:
            try:
                print('calculating RMSD for {}'.format(refcode))
                graph_features['RMSD'] =
self.rmsd.calculate(refcode)
            except RuntimeError:
                print('RMSD failed with {}'.format(refcode))
                self.skip_refcode.append(refcode)
                return None

        if self.do_h_dims:
            h_dim_text =
HBond.dimensionality(self.csd_reader.crystal(refcode))
            if h_dim_text == 'No Hydrogen bonds':
                h_dim = [0, 0, 0, 0]
            else:
                h_dim = onehot(h_dim_text, self.h_dim_features)
            graph_features['H_dims'] = h_dim
        if len(graph_features) != 0:
            return graph_features
        else:
            return None

    def update_nodes(self):
        """now that all mols are processed, the atom_list is
complete and ready for one hot"""
        for section, data in self.graphs.iteritems():
            for i, mol in enumerate(data):
                for j, atom in enumerate(mol['node_features']):
                    self.graphs[section][i]['node_features'][j] =
onehot(atom, self.atom_list) # could use map() maybe, will have to
look into it

    def process(self, raw_data):
        for section, data in raw_data.iteritems():
            total = len(data)
            for i, mol in enumerate(data):
                refcode, y = mol['refcode'], mol['y']
                if self.to_normalise:
                    y = self.normalise(y)
                nodes, edges = self.get_graphs(refcode)
                if refcode in self.skip_refcode:
                    continue
                else:
                    atom_count = len(nodes)
                    self.atom_counts[refcode] = atom_count
                    row = {'targets': [[y]], 'graph': edges,
'node_features': nodes, 'id': refcode}
                    graph_features =
self.get_crystal_properties(refcode)
                    if graph_features is not None:
                        row['graph_features'] = graph_features
                    self.graphs[section].append(row)
                if i % 1000 == 0:
                    print('{} graphs processed: {}%'.format(section,
round(float(i)/total,3)*100))
        if self.get_elements:
            self.update_nodes()
        return self.graphs


def output_data(processed_data, header=None):
    if header is None:
        header='data'
    for section in processed_data.keys():
        with open('{}_{}.json'.format(header, section), 'w') as f:
            json.dump(processed_data[section], f)


def main():
    parser = argparse.ArgumentParser('''REFCODE, target to GNN
input''')
    parser.add_argument('input', help='input csv with REFCODE,
target per line')
    parser.add_argument('--atoms', help='file with list of atoms')
    parser.add_argument('-n', '--normalise', action='store_true',
help='normalise the target values')
    parser.add_argument('-s', '--split_frac', default=0.1,
help='valid set fraction')
    parser.add_argument('--rmsd', action='store_true', help='Do
crystal rmsd as a graph level descriptor')
    parser.add_argument('--hdim', action='store_true', help='Do H-
bond dimensionality as a graph level descriptor')
    parser.add_argument('--vdw', action='store_true', help='include
VdW interaction in crystal graph')
    parser.add_argument('--atom_count', action='store_true',
help='output atom count file')
    args = parser.parse_args()

    csd_dir = io.csd_directory()
    csd_location = glob.glob(os.path.join(csd_dir, '*.inf'))
    csd_reader = io.EntryReader(csd_location)

    in_f = args.input
    valid_frac = float(args.split_frac)
    to_normalise = args.normalise
    if args.atoms:
        with open(args.atoms, 'r') as csvfile:
            reader = csv.reader(csvfile, delimiter=',')
            for row in reader:
                atom_list = row
                # atom_list = reader.next() # didnt work for some
reason
    else:
        atom_list = None
    if args.rmsd:
        crystal_rmsd = CrystalRMSD(csd_reader)
    else:
        crystal_rmsd = None

    raw = from_csv(in_f)
    out_name = in_f.split('.')[0]
    raw_data, mean, std = split_data(raw, valid_frac, to_normalise)

    crystal_graphs = CrystalGraph(csd_reader, atom_list, mean, std, 
crystal_rmsd, args.hdim)

    graphs = crystal_graphs.process(raw_data)

    output_data(graphs, out_name)
    print('failed to process the following refcodes: ')
    for ref in crystal_graphs.skip_refcode:
        print(ref)

    if to_normalise:
        with open('{}_statistics.json'.format(out_name), 'w') as f:
            json.dump({'mean': mean, 'std': std}, f)
    if crystal_graphs.get_elements:
        with open('{}_atoms.csv'.format(out_name), 'w') as f:
            writer = csv.writer(f, delimiter=',')
            writer.writerow(crystal_graphs.atom_list)

    if args.atom_count:
        with open('{}_atom_counts.json'.format(out_name), 'w') as f:
            json.dump(crystal_graphs.atom_counts, f)


if __name__ == '__main__':
    main()
```

crystal_rmsd.py

```
from rdkit import Chem
from rdkit.Chem import rdMolAlign

from conf_gen import ConformerGenerator


class CrystalRMSD:
    def __init__(self, csd_reader, force_field='mmff'):
        self.csd_reader = csd_reader
        self.conformer_generator =
ConformerGenerator(force_field=force_field)

    def calculate(self, refcode):
        csd_mol = self.csd_reader.molecule(refcode)

        csd_mols = csd_mol.components # get a list of all molecules
in the crystal

        csd_mol1 = csd_mols[0] # get the first mol to get the
conformer
        mol1 = Chem.MolFromMolBlock(csd_mol1.to_string('sdf'))
        if mol1 is None:
            print('cant construct the molecule')
            raise RuntimeError
        try:
            conf = self.conformer_generator(mol1)
            if Chem.AllChem.EmbedMolecule(conf) == -1: # = -1 if
failed, id assigned otherwise 0,1,..
                print('molecule too large to generate conformer')
                raise RuntimeError
            rmsds = []
            for mol in csd_mols:
                mol = Chem.MolFromMolBlock(mol.to_string('sdf'))
                rmsd = rdMolAlign.GetBestRMS(mol, conf)
                rmsds.append(rmsd)
        except AttributeError:
            print('something went wrong with minimisation of
conformer')
            raise RuntimeError
        rmsd = sum(rmsds) / len(rmsds)

        return rmsd
```

util.py

```
#!/usr/bin/env/python

import numpy as np
import tensorflow as tf
import queue
import threading

SMALL_NUMBER = 1e-7


def glorot_init(shape):
    initialization_range = np.sqrt(6.0 / (shape[-2] + shape[-1]))
    return np.random.uniform(low=-initialization_range,
high=initialization_range, size=shape).astype(np.float32)


class ThreadedIterator:
    """An iterator object that computes its elements in a parallel
thread to be ready to be consumed.
    The iterator should *not* return None"""

    def __init__(self, original_iterator, max_queue_size: int=2):
        self.__queue = queue.Queue(maxsize=max_queue_size)
        self.__thread = threading.Thread(target=lambda:
self.worker(original_iterator))
        self.__thread.start()

    def worker(self, original_iterator):
        for element in original_iterator:
            assert element is not None, 'By convention, iterator
elements much not be None'
            self.__queue.put(element, block=True)
        self.__queue.put(None, block=True)

    def __iter__(self):
        next_element = self.__queue.get(block=True)
        while next_element is not None:
            yield next_element
            next_element = self.__queue.get(block=True)
        self.__thread.join()


class MLP(object):
    def __init__(self, in_size, out_size, hid_sizes,
dropout_keep_prob, family='MLP_layer'):
        self.in_size = in_size
        self.out_size = out_size
        self.hid_sizes = hid_sizes
        self.family = family
        self.dropout_keep_prob = dropout_keep_prob
        self.params = self.make_network_params()

    def make_network_params(self):
        dims = [self.in_size] + self.hid_sizes + [self.out_size]
        weight_sizes = list(zip(dims[:-1], dims[1:]))
        weights = [tf.Variable(self.init_weights(s),
name='MLP_W_layer%i' % i)
                   for (i, s) in enumerate(weight_sizes)]
        biases = [tf.Variable(np.zeros(s[-1]).astype(np.float32),
name='MLP_b_layer%i' % i)
```

```
                  for (i, s) in enumerate(weight_sizes)]

        network_params = {
            "weights": weights,
            "biases": biases,
        }

        return network_params

    def init_weights(self, shape):
        return np.sqrt(6.0 / (shape[-2] + shape[-1])) * (2 *
np.random.rand(*shape).astype(np.float32) - 1)

    def __call__(self, inputs):
        acts = inputs
        for W, b in zip(self.params["weights"],
self.params["biases"]):
            tf.summary.histogram('MLP_weights', W,
family=self.family)
            tf.summary.histogram('MLP_biases', b,
family=self.family)
            hid = tf.matmul(acts, tf.nn.dropout(W,
self.dropout_keep_prob)) + b
            acts = tf.nn.relu(hid)
        last_hidden = hid
        return last_hidden
```

chem_tensorflow_.py

```
#!/usr/bin/env/python

from typing import Tuple, List, Any, Sequence

import tensorflow as tf
import time
import os
import json
import numpy as np
import pickle
import random

from utils import MLP, ThreadedIterator, SMALL_NUMBER


class ChemModel(object):
    @classmethod
    def default_params(cls):
        return {
            'num_epochs': 3000,
            'patience': 25,
            'learning_rate': 0.001,
            'clamp_gradient_norm': 1.0,
            'out_layer_dropout_keep_prob': 1.0,
            'gated_regression_keep_prob': 1.0,
            'graph_representation_size': 100,
            'prediction_layers_architecture': [50, 20],

            'hidden_size': 100,
            'num_timesteps': 4,
            'use_graph': True,

            'tie_fwd_bkwd': True,
            'task_ids': [0],

            'random_seed': 0,

            'train_file': 'molecules_train.json',
            'valid_file': 'molecules_valid.json'
        }

    def __init__(self, args):
        self.args = args
        self.edge_dict = {1: 'SINGLE', 2: 'DOUBLE', 3: 'TRIPLE', 4:
'AROMATIC', 5: 'HBOND', 6: 'VDW_INTER', 7: 'VDW_INTRA'}
        self.best_r = float('-inf') # best R^2 will be stored here
for the hyper optimiser to access
        # Collect argument things:
        data_dir = ''
        if '--data_dir' in args and args['--data_dir'] is not None:
            data_dir = args['--data_dir']
        self.data_dir = data_dir

        run_id = str(args.get('--run_id')) or str(os.getpid())
        self.run_id = "_".join([time.strftime("%Y-%m-%d-%H-%M-%S"),
run_id])
        log_dir = args.get('--run_dir') or '.'
        self.log_dir = log_dir
        self.log_file = os.path.join(log_dir, "%s_log.json" %
self.run_id)
```

```
        self.best_model_file = os.path.join(log_dir,
"%s_model_best.pickle" % self.run_id)

        # Collect parameters:
        params = self.default_params()
        config_file = args.get('--config-file')
        if config_file is not None:
            with open(config_file, 'r') as f:
                params.update(json.load(f))
        config = args.get('--config')
        if config is not None:
            params.update(json.loads(config))
        conf = args.get('conf')
        if conf is not None: # only for hyperopt
            params.update(conf)
        self.params = params
        with open(os.path.join(log_dir, "%s_params.json" %
self.run_id), "w") as f:
            json.dump(params, f)
        print("Run %s starting with following parameters:\n%s" %
(self.run_id, json.dumps(self.params)))
        random.seed(params['random_seed'])
        np.random.seed(params['random_seed'])


        # Load data:
        self.max_num_vertices = 0
        self.num_edge_types = 0
        self.annotation_size = 0
        self.num_graph_features = 0
        self.graph_features_list = []
        self.graph_feature_lengths = {}
        # modify usable data
        edge_types = params.get('edge_types')
        graph_descriptors = params.get('graph_descriptors')

        self.train_data = self.load_data(params['train_file'],
edge_types, graph_descriptors, is_training_data=True)
        self.valid_data = self.load_data(params['valid_file'],
edge_types, graph_descriptors, is_training_data=False)

        # Build the actual model
        config = tf.ConfigProto()
        config.gpu_options.allow_growth = True
        self.graph = tf.Graph()
        self.sess = tf.Session(graph=self.graph, config=config)
        self.writer = tf.summary.FileWriter(self.log_dir)
        with self.graph.as_default():
            tf.set_random_seed(params['random_seed'])
            self.placeholders = {}
            self.weights = {}
            self.ops = {}
            self.make_model()
            self.make_train_step()

            # Restore/initialize variables:
            restore_file = args.get('--restore')
            if restore_file is not None:
                self.restore_model(restore_file)
            else:
                self.initialize_model()
```

```
    def load_data(self, file_name, edge_types, graph_descriptors,
is_training_data: bool):
        full_path = os.path.join(self.data_dir, file_name)

        print("Loading data from %s" % full_path)
        with open(full_path, 'r') as f:
            data = json.load(f)

        restrict = self.args.get("--restrict_data")
        if restrict is not None and restrict > 0:
            data = data[:restrict]

        # block out edge types and graph features that are not to be
used
        if edge_types is not None or graph_descriptors is not None:
            for i, g in enumerate(data):
                if edge_types:
                    edges = []
                    for edge in g['graph']:
                        if self.edge_dict[edge[1]] in edge_types:
                            edges.append(edge)
                    data[i]['graph'] = edges

                if graph_descriptors:
                    features = {}
                    for feature, value in
g['graph_features'].items():
                        if feature in graph_descriptors:
                            features[feature] = value
                    data[i]['graph_features'] = features

        # Get some common data out:
        num_fwd_edge_types = 0
        for g in data:
            self.max_num_vertices = max(self.max_num_vertices,
max([v for e in g['graph'] for v in [e[0], e[2]]]))
            num_fwd_edge_types = max(num_fwd_edge_types, max([e[1]
for e in g['graph']]))
        self.num_edge_types = max(self.num_edge_types,
num_fwd_edge_types * (1 if self.params['tie_fwd_bkwd'] else 2))
        self.annotation_size = max(self.annotation_size,
len(data[0]["node_features"][0]))



        # get the number of graph features
        if is_training_data:
            if 'graph_features' in data[0].keys():
                self.num_graph_features =
len(data[0]['graph_features'])
                self.graph_features_list =
list(data[0]['graph_features'].keys())
                for feature in self.graph_features_list:
                    try:
                        dims =
len(data[0]['graph_features'][feature])
                    except TypeError:
                        dims = 1
                    self.graph_feature_lengths[feature] = dims
        return self.process_raw_graphs(data, is_training_data)
    @staticmethod
    def graph_string_to_array(graph_string: str) -> List[List[int]]:
        return [[int(v) for v in s.split(' ')]
                for s in graph_string.split('\n')]

    def process_raw_graphs(self, raw_data: Sequence[Any],
is_training_data: bool) -> Any:
        raise Exception("Models have to implement
process_raw_graphs!")

    def make_model(self):
        self.placeholders['target_values'] =
tf.placeholder(tf.float32, [len(self.params['task_ids']), None],

name='target_values')
        self.placeholders['target_mask'] =
tf.placeholder(tf.float32, [len(self.params['task_ids']), None],

name='target_mask')
        self.placeholders['num_graphs'] = tf.placeholder(tf.int64,
[], name='num_graphs')
        self.placeholders['gated_regression_keep_prob'] =
tf.placeholder(tf.float32, [], name='gated_regression_keep_prob')
        self.placeholders['out_layer_dropout_keep_prob'] =
tf.placeholder(tf.float32, [], name='out_layer_dropout_keep_prob')
        # get placeholder for each graph feature
        if self.num_graph_features > 0:
            for graph_feature in self.graph_features_list:
                dims = self.graph_feature_lengths[graph_feature]
                self.placeholders[graph_feature] =
tf.placeholder(tf.float32, [dims, None], name=graph_feature)

        with tf.variable_scope("graph_model"):
            self.prepare_specific_graph_model()
            # This does the actual graph work:
            if self.params['use_graph']:
                self.ops['final_node_representations'] =
self.compute_final_node_representations()
            else:
                self.ops['final_node_representations'] =
tf.zeros_like(self.placeholders['initial_node_representation'])

            with tf.variable_scope('gated_regression'):
                with tf.variable_scope("regression_gate"):
                    self.weights['regression_gate'] = MLP(2 *
self.params['hidden_size'],
self.params['graph_representation_size'], [],

self.placeholders[

'gated_regression_keep_prob'], 'gated_regression')
                with tf.variable_scope("regression"):
                    self.weights['regression_transform'] =
MLP(self.params['hidden_size'],
self.params['graph_representation_size'], [],

self.placeholders[

'gated_regression_keep_prob'], 'gated_regression')
```

```
                graph_representation =
self.gated_regression(self.ops['final_node_representations'],

self.weights['regression_gate'],

self.weights['regression_transform'])
                print('graph representation shape: ')
                print(graph_representation.get_shape())

        out_size = len(self.params['task_ids'])
        # get the total length of all graph level features
        graph_features_length = 0
        for feature in self.graph_features_list:
            graph_features_length +=
self.graph_feature_lengths[feature]

        in_size = self.params['graph_representation_size'] +
graph_features_length
        print('the in_size is {}'.format(in_size))
        with tf.variable_scope('prediction_layers'):
            input_tensors = [self.placeholders[feature] for feature
in self.graph_features_list]
            input_tensors.append(graph_representation)
            prediction_input = tf.concat(input_tensors, 0,
name='prediction_input')

            print('prediction_input_size: ')
            print(prediction_input.get_shape())
            print('this will be transposed')
            prediction_layer = MLP(in_size, out_size,
self.params['prediction_layers_architecture'],

self.params['out_layer_dropout_keep_prob'], 'prediction_MLP')
            computed_values =
prediction_layer(tf.transpose(prediction_input))
            computed_values = tf.transpose(computed_values)
            print('computed value shape: ')
            print(computed_values.get_shape())
            self.ops['predicted'] = computed_values
        with tf.variable_scope('performance_measure'):
            diff = tf.subtract(computed_values,
self.placeholders['target_values'], name='diff')
            diff = tf.multiply(diff,
self.placeholders['target_mask'])
            print('diff shape')
            print(diff.get_shape())
            task_target_num =
tf.reduce_sum(self.placeholders['target_mask'], axis=1,
name='batch_size') + SMALL_NUMBER

            #batch_mean = tf.div(tf.reduce_sum(computed_values,
1),task_target_num, name='batch_mean')

            with tf.variable_scope('mean_squared_error'):
                squared_diff = tf.reduce_sum(tf.square(diff),
name='squared_diff')
                loss = tf.div(squared_diff,task_target_num, 'MSE')
                self.ops['loss'] = tf.reduce_sum(loss) # total loss
across all tasks
                tf.summary.scalar('loss', self.ops['loss'],
family='overall_performance')
```

```
            with tf.variable_scope('mean_abs_error'):
                mae = tf.div(tf.reduce_sum(tf.abs(diff), 1),
task_target_num)
                print('MAE shape: ')
                print(mae.get_shape())
                self.ops['MAE'] = tf.reduce_sum(mae) # Mean Absolute
Error

            with tf.variable_scope('R2'):
                #tss = tf.subtract(tf.multiply(computed_values,
self.placeholders['target_mask']), tf.expand_dims(batch_mean,1)) # y
- y_mean
                tss =
tf.subtract(tf.multiply(self.placeholders['target_values'],
self.placeholders['target_mask']), tf.constant(0, dtype=tf.float32))
# y_mean = 0,bcuz standarised in preprocessing
                tss = tf.reduce_sum(tf.square(tss), axis=1,
name='TSS')  # sum((y-y_mean)^2)
                print('TSS shape: ')
                print(tss.get_shape())
                r_squared = tf.subtract(tf.constant(1,
dtype=tf.float32), tf.div(squared_diff, tss))
                print('R^2 shape: ')
                print(r_squared.get_shape())
                self.ops['R2'] = tf.reduce_sum(r_squared)


            for internal_id, task_id in
enumerate(self.params['task_ids']):
                tf_id = tf.constant([internal_id],
name='task_{}_id'.format(task_id))
                task_diff = tf.nn.embedding_lookup(diff, tf_id)
                print('task diff: ')
                print(task_diff.get_shape())
                tf.summary.histogram('diff', task_diff,
family='task_{}_performance'.format(task_id))
                task_mae = tf.nn.embedding_lookup(mae, tf_id)
                print('task MAE')
                print(task_mae.get_shape())
                tf.summary.scalar('task_MAE', tf.squeeze(task_mae),
family='task_{}_performance'.format(task_id))
                task_loss = tf.nn.embedding_lookup(loss, tf_id)
                tf.summary.scalar('task_loss',
tf.squeeze(task_loss), family='task_{}_performance'.format(task_id))
                task_r_squared = tf.nn.embedding_lookup(r_squared,
tf_id)
                tf.summary.scalar('task_R2',
tf.squeeze(task_r_squared),
family='task_{}_performance'.format(task_id))

                #task_tss = tf.nn.embedding_lookup(tss, tf_id)
                #tf.summary.scalar('task_tss', tf.squeeze(task_tss),
family='task_{}_performance'.format(task_id))
            # Currently not done
            # Normalise loss to account for fewer task-specific
examples in batch:
            # task_loss = task_loss * (1.0 /
(self.params['task_sample_ratios'].get(task_id) or 1.0))

        self.merged_summary = tf.summary.merge_all()
```

```
    def make_train_step(self):
        trainable_vars =
self.sess.graph.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES)
        if self.args.get('--freeze-graph-model'):
            graph_vars =
set(self.sess.graph.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
scope="graph_model"))
            filtered_vars = []
            for var in trainable_vars:
                if var not in graph_vars:
                    filtered_vars.append(var)
                else:
                    print("Freezing weights of variable %s." %
var.name)
            trainable_vars = filtered_vars
        optimizer =
tf.train.AdamOptimizer(self.params['learning_rate'])
        grads_and_vars =
optimizer.compute_gradients(self.ops['loss'],
var_list=trainable_vars)
        clipped_grads = []
        with tf.variable_scope('clib_by_norm'):
            for grad, var in grads_and_vars:
                if grad is not None:
                    clipped_grads.append((tf.clip_by_norm(grad,
self.params['clamp_gradient_norm']), var))
                else:
                    clipped_grads.append((grad, var))
        self.ops['train_step'] =
optimizer.apply_gradients(clipped_grads)
        # Initialize newly-introduced variables:
        self.sess.run(tf.local_variables_initializer())

    def gated_regression(self, last_h, regression_gate,
regression_transform):
        raise Exception("Models have to implement
gated_regression!")

    def prepare_specific_graph_model(self) -> None:
        raise Exception("Models have to implement
prepare_specific_graph_model!")

    def compute_final_node_representations(self) -> tf.Tensor:
        raise Exception("Models have to implement
compute_final_node_representations!")

    def make_minibatch_iterator(self, data: Any, is_training: bool):
        raise Exception("Models have to implement
make_minibatch_iterator!")

    def run_epoch(self, epoch_name: str, data, is_training: bool):


        loss = 0
        maes = []
        start_time = time.time()
        processed_graphs = 0
        batch_iterator =
ThreadedIterator(self.make_minibatch_iterator(data, is_training),
max_queue_size=5)
```

```
        for step, batch_data in enumerate(batch_iterator):
            num_graphs = batch_data[self.placeholders['num_graphs']]
            processed_graphs += num_graphs
            if is_training:

batch_data[self.placeholders['out_layer_dropout_keep_prob']] =
self.params['out_layer_dropout_keep_prob']

batch_data[self.placeholders['gated_regression_keep_prob']] =
self.params[
                    'gated_regression_keep_prob']

                fetch_list = [self.ops['loss'], self.ops['MAE'],
self.ops['train_step'], self.ops['R2']]
            else:

batch_data[self.placeholders['out_layer_dropout_keep_prob']] = 1.0

batch_data[self.placeholders['gated_regression_keep_prob']] = 1.0
                fetch_list = [self.ops['loss'], self.ops['MAE'],
self.merged_summary, self.ops['R2']]
            result = self.sess.run(fetch_list, feed_dict=batch_data)
            (batch_loss, batch_mae) = (result[0], result[1])
            loss += batch_loss * num_graphs
            maes.append(np.array(batch_mae) * num_graphs)
            if not is_training:
                summary = result[2]
            else:
                summary = None
            print("Running %s, batch %i (has %i graphs). Loss so
far: %.4f" % (epoch_name,

step,

num_graphs,

loss / processed_graphs),
                  end='\r')

        MAE = np.sum(maes, axis=0) / processed_graphs
        loss = loss / processed_graphs
        instance_per_sec = processed_graphs / (time.time() -
start_time)
        return loss, MAE, instance_per_sec, summary, result[3] # R^2

    def train(self):
        log_to_save = []
        total_time_start = time.time()
        with self.graph.as_default():
            if self.args.get('--restore') is not None:
                _, _, _, _, r = self.run_epoch("Resumed
(validation)", self.valid_data, False)
                best_r = np.sum(r)
                best_val_r_epoch = 0
                best_val_loss = None
                print("\r\x1b[KResumed operation, initial
R^2: %.5f" % best_r)
            else:
                (best_r, best_val_r_epoch) = (float("-inf"), 0)
                best_val_loss, best_val_mae = None, None
            for epoch in range(1, self.params['num_epochs'] + 1):
                print("== Epoch %i" % epoch)
                train_loss, train_mae, train_speed, _, train_r =
self.run_epoch("epoch %i (training)" % epoch,

self.train_data, True)

                print('\r[Train: epoch: {}, MSE: {}, MAE: {}, R^2:
{} instances/sec: {}'.format(epoch, train_loss, train_mae, train_r,
train_speed))

                valid_loss, valid_mae, valid_speed, summary, valid_r
= self.run_epoch("epoch %i (validation)" % epoch,

self.valid_data, False)
                self.writer.add_summary(summary, epoch)

                print('\r[Valid: epoch: {}, MSE: {}, MAE: {}, R^2:
{}, instances/sec: {}'.format(epoch, valid_loss, valid_mae, valid_r,

valid_speed))

                epoch_time = time.time() - total_time_start



                log_entry = {
                    'epoch': epoch,
                    'time': epoch_time,
                    'train_results': (float(train_loss),
float(train_mae), float(train_r), train_speed),
                    'valid_results': (valid_loss, valid_mae,
float(valid_r), valid_speed),
                }
                log_to_save.append(log_entry)
                with open(self.log_file, 'w') as f:
                    json.dump(log_to_save, f, indent=4)

                # type: float
                if valid_r > best_r:
                    self.save_model(self.best_model_file)
                    print("  (Best epoch so far, R^2 decreased
to %.5f from %.5f. Saving to '%s')" % (valid_r, best_r,
self.best_model_file))
                    best_r = valid_r
                    best_val_mae = valid_mae
                    best_val_loss = valid_loss
                    best_val_r_epoch = epoch
                elif epoch - best_val_r_epoch >=
self.params['patience']:
                    print("Stopping training after %i epochs without
improvement on validation accuracy." % self.params['patience'])
                    # print the performance summary
                    full_path = os.path.join(self.log_dir,
'performance.json')
                    date = self.run_id.split('_')[0]
                    run_id = self.run_id.split('_')[1]
                    self.best_r = best_r
                    with open(full_path, 'w') as f:
                        json.dump({'date': date, 'run_id': run_id,
'MAE': float(best_val_mae),
                                   'MSE': float(best_val_loss),
'R2': float(best_r), 'epochs': best_val_r_epoch}, f)
                    break

    def save_model(self, path: str) -> None:
        weights_to_save = {}
        for variable in
self.sess.graph.get_collection(tf.GraphKeys.GLOBAL_VARIABLES):
            assert variable.name not in weights_to_save
            weights_to_save[variable.name] = self.sess.run(variable)

        data_to_save = {
                         "params": self.params,
                         "weights": weights_to_save
                       }

        with open(path, 'wb') as out_file:
            pickle.dump(data_to_save, out_file,
pickle.HIGHEST_PROTOCOL)

    def initialize_model(self) -> None:
        init_op = tf.group(tf.global_variables_initializer(),
                           tf.local_variables_initializer())
        self.sess.run(init_op)
        self.writer.add_graph(self.sess.graph)
        self.merged_summary = tf.summary.merge_all()

    def restore_model(self, path: str) -> None:
        print("Restoring weights from file %s." % path)
        with open(path, 'rb') as in_file:
            data_to_load = pickle.load(in_file)

        # Assert that we got the same model configuration
        assert len(self.params) == len(data_to_load['params'])
        for (par, par_value) in self.params.items():
            # Fine to have different task_ids:
            if par not in ['task_ids', 'num_epochs']:
                assert par_value == data_to_load['params'][par]

        variables_to_initialize = []
        with tf.name_scope("restore"):
            restore_ops = []
            used_vars = set()
            for variable in
self.sess.graph.get_collection(tf.GraphKeys.GLOBAL_VARIABLES):
                used_vars.add(variable.name)
                if variable.name in data_to_load['weights']:

restore_ops.append(variable.assign(data_to_load['weights'][variable.
name]))
                else:
                    print('Freshly initializing %s since no saved
value was found.' % variable.name)
                    variables_to_initialize.append(variable)
            for var_name in data_to_load['weights']:
                if var_name not in used_vars:
                    print('Saved weights for %s not used by
model.' % var_name)

restore_ops.append(tf.variables_initializer(variables_to_initialize)
)
```

```
            self.sess.run(restore_ops)
```

chem_tensorflow_sparse.py

```
#!/usr/bin/env/python
"""
Usage:
    chem_tensorflow_sparse.py [options]

Options:
    -h --help                Show this screen.
    --config-file FILE       Hyperparameter configuration file path
(in JSON format).
    --config CONFIG          Hyperparameter configuration dictionary
(in JSON format).
    --run_dir DIR            Run dir name.
    --data_dir DIR           Data dir name.
    --restore FILE           File to restore weights from.
    --freeze-graph-model     Freeze weights of graph model
components.
    --run_id ID              Run_id.
"""
from typing import List, Tuple, Dict, Sequence, Any

from docopt import docopt
from collections import defaultdict, namedtuple
import numpy as np
import tensorflow as tf
import sys, traceback
import pdb

from chem_tensorflow import ChemModel
from utils import glorot_init, SMALL_NUMBER


GGNNWeights = namedtuple('GGNNWeights', ['edge_weights',
                                         'edge_biases',

'edge_type_attention_weights',
                                         'rnn_cells',])


class SparseGGNNChemModel(ChemModel):
    def __init__(self, args):
        super().__init__(args)

    @classmethod
    def default_params(cls):
        params = dict(super().default_params())
        params.update({
            'batch_size': 100000,
            'use_edge_bias': False,
            'use_propagation_attention': False,
            'use_edge_msg_avg_aggregation': True,
            'residual_connections': {  # For layer i, specify list
of layers whose output is added as an input
                                     "2": [0],
                                     "4": [0, 2]
                                    },

            'layer_timesteps': [2, 2, 1, 2, 1],  # number of layers
& propagation steps per layer

            'graph_rnn_cell': 'GRU',  # GRU or RNN
            'graph_rnn_activation': 'tanh',  # tanh, ReLU
            'graph_state_dropout_keep_prob': 1.,
            'task_sample_ratios': {},
        })
        return params

    def prepare_specific_graph_model(self) -> None:
        h_dim = self.params['hidden_size']
        self.placeholders['initial_node_representation'] =
tf.placeholder(tf.float32, [None, h_dim],

name='node_features')
        self.placeholders['adjacency_lists'] =
[tf.placeholder(tf.int32, [None, 2], name='adjacency_e%s' % e)
                                                for e in
range(self.num_edge_types)]
        self.placeholders['num_incoming_edges_per_type'] =
tf.placeholder(tf.float32, [None, self.num_edge_types],

name='num_incoming_edges_per_type')
        self.placeholders['graph_nodes_list'] =
tf.placeholder(tf.int64, [None, 2], name='graph_nodes_list')
        self.placeholders['graph_state_keep_prob'] =
tf.placeholder(tf.float32, None, name='graph_state_keep_prob')

        activation_name =
self.params['graph_rnn_activation'].lower()
        if activation_name == 'tanh':
            activation_fun = tf.nn.tanh
        elif activation_name == 'relu':
            activation_fun = tf.nn.relu
        else:
            raise Exception("Unknown activation function type
'%s'." % activation_name)

        # Generate per-layer values for edge weights, biases and
gated units:
        self.weights = {}  # Used by super-class to place generic
things
        self.gnn_weights = GGNNWeights([], [], [], [])
        for layer_idx in range(len(self.params['layer_timesteps'])):
            with tf.variable_scope('gnn_layer_%i' % layer_idx):
                edge_weights =
tf.Variable(glorot_init([self.num_edge_types * h_dim, h_dim]),

name='gnn_edge_weights_%i' % layer_idx)
                edge_weights = tf.reshape(edge_weights,
[self.num_edge_types, h_dim, h_dim])
                self.gnn_weights.edge_weights.append(edge_weights)

                if self.params['use_propagation_attention']:

self.gnn_weights.edge_type_attention_weights.append(tf.Variable(np.o
nes([self.num_edge_types], dtype=np.float32),

name='edge_type_attention_weights_%i' % layer_idx))

                if self.params['use_edge_bias']:

self.gnn_weights.edge_biases.append(tf.Variable(np.zeros([self.num_e
dge_types, h_dim], dtype=np.float32),
name='gnn_edge_biases_%i' % layer_idx))

                cell_type = self.params['graph_rnn_cell'].lower()
                if cell_type == 'gru':
                    cell = tf.nn.rnn_cell.GRUCell(h_dim,
activation=activation_fun)
                elif cell_type == 'rnn':
                    cell = tf.nn.rnn_cell.BasicRNNCell(h_dim,
activation=activation_fun)
                else:
                    raise Exception("Unknown RNN cell type '%s'." %
cell_type)
                cell = tf.nn.rnn_cell.DropoutWrapper(cell,

state_keep_prob=self.placeholders['graph_state_keep_prob'])
                self.gnn_weights.rnn_cells.append(cell)

    def compute_final_node_representations(self) -> tf.Tensor:
        node_states_per_layer = []  # one entry per layer (final
state of that layer), shape: number of nodes in batch v x D

node_states_per_layer.append(self.placeholders['initial_node_represe
ntation'])
        num_nodes =
tf.shape(self.placeholders['initial_node_representation'],
out_type=tf.int32)[0]

        message_targets = []  # list of tensors of message targets
of shape [E]
        message_edge_types = []  # list of tensors of edge type of
shape [E]
        for edge_type_idx, adjacency_list_for_edge_type in
enumerate(self.placeholders['adjacency_lists']):
            edge_targets = adjacency_list_for_edge_type[:, 1]
            message_targets.append(edge_targets)
            message_edge_types.append(tf.ones_like(edge_targets,
dtype=tf.int32) * edge_type_idx)
        message_targets = tf.concat(message_targets, axis=0)  #
Shape [M]
        message_edge_types = tf.concat(message_edge_types, axis=0)
# Shape [M]

        for (layer_idx, num_timesteps) in
enumerate(self.params['layer_timesteps']):
            with tf.variable_scope('gnn_layer_%i' % layer_idx):
                # Used shape abbreviations:
                #   V ~ number of nodes
                #   D ~ state dimension
                #   E ~ number of edges of current type
                #   M ~ number of messages (sum of all E)

                # Extract residual messages, if any:
                layer_residual_connections =
self.params['residual_connections'].get(str(layer_idx))
                if layer_residual_connections is None:
                    layer_residual_states = []
                else:
                    layer_residual_states =
[node_states_per_layer[residual_layer_idx]
```

```
                                             for residual_layer_idx
in layer_residual_connections]

                if self.params['use_propagation_attention']:
                    message_edge_type_factors =
tf.nn.embedding_lookup(params=self.gnn_weights.edge_type_attention_w
eights[layer_idx],

ids=message_edge_types)  # Shape [M]
                for edge_type_idx in
range(len(self.placeholders['adjacency_lists'])):

tf.summary.histogram('GNN_layer_{}_edge_type_{}'.format(layer_idx,
edge_type_idx),

self.gnn_weights.edge_weights[layer_idx][edge_type_idx])

                # Record new states for this layer. Initialised to
last state, but will be updated below:
                node_states_per_layer.append(node_states_per_layer[-
1])
                for step in range(num_timesteps):
                    with tf.variable_scope('timestep_%i' % step):
                        messages = []  # list of tensors of messages
of shape [E, D]
                        message_source_states = []  # list of
tensors of edge source states of shape [E, D]

                        # Collect incoming messages per edge type
                        for edge_type_idx,
adjacency_list_for_edge_type in
enumerate(self.placeholders['adjacency_lists']):
                            edge_sources =
adjacency_list_for_edge_type[:, 0]
                            edge_source_states =
tf.nn.embedding_lookup(params=node_states_per_layer[-1],

ids=edge_sources)  # Shape [E, D]
                            all_messages_for_edge_type =
tf.matmul(edge_source_states,

self.gnn_weights.edge_weights[layer_idx][edge_type_idx])  # Shape
[E, D]


messages.append(all_messages_for_edge_type)

message_source_states.append(edge_source_states)

                        messages = tf.concat(messages, axis=0)  #
Shape [M, D]

                        if self.params['use_propagation_attention']:
                            message_source_states =
tf.concat(message_source_states, axis=0)  # Shape [M, D]
                            message_target_states =
tf.nn.embedding_lookup(params=node_states_per_layer[-1],

ids=message_targets)  # Shape [M, D]
```

```
                            message_attention_scores =
tf.einsum('mi,mi->m', message_source_states, message_target_states)
# Shape [M]
                            message_attention_scores =
message_attention_scores * message_edge_type_factors

                            # The following is softmax-ing over the
incoming messages per node.
                            # As the number of incoming varies, we
can't just use tf.softmax. Reimplement with logsumexp trick:
                            # Step (1): Obtain shift constant as max
of messages going into a node
                            message_attention_score_max_per_target =
tf.unsorted_segment_max(data=message_attention_scores,

segment_ids=message_targets,

num_segments=num_nodes)  # Shape [V]
                            # Step (2): Distribute max out to the
corresponding messages again, and shift scores:
                            message_attention_score_max_per_message
= tf.gather(params=message_attention_score_max_per_target,

indices=message_targets)  # Shape [M]
                            message_attention_scores -=
message_attention_score_max_per_message
                            # Step (3): Exp, sum up per target,
compute exp(score) / exp(sum) as attention prob:
                            message_attention_scores_exped =
tf.exp(message_attention_scores)  # Shape [M]
                            message_attention_score_sum_per_target =
tf.unsorted_segment_sum(data=message_attention_scores_exped,

segment_ids=message_targets,

num_segments=num_nodes)  # Shape [V]

message_attention_normalisation_sum_per_message =
tf.gather(params=message_attention_score_sum_per_target,

indices=message_targets)  # Shape [M]
                            message_attention =
message_attention_scores_exped /
(message_attention_normalisation_sum_per_message + SMALL_NUMBER)  #
Shape [M]
                            # Step (4): Weigh messages using the
attention prob:
                            messages = messages *
tf.expand_dims(message_attention, -1)

                        incoming_messages =
tf.unsorted_segment_sum(data=messages,

segment_ids=message_targets,

num_segments=num_nodes)  # Shape [V, D]

                        if self.params['use_edge_bias']:
                            incoming_messages +=
tf.matmul(self.placeholders['num_incoming_edges_per_type'],
self.gnn_weights.edge_biases[layer_idx])  # Shape [V, D]

                        if
self.params['use_edge_msg_avg_aggregation']:
                            num_incoming_edges =
tf.reduce_sum(self.placeholders['num_incoming_edges_per_type'],

keep_dims=True, axis=-1)  # Shape [V, 1]
                            incoming_messages /= num_incoming_edges
+ SMALL_NUMBER

                        incoming_information =
tf.concat(layer_residual_states + [incoming_messages],
                                                          axis=-1)  #
Shape [V, D*(1 + num of residual connections)]

                        # pass updated vertex features into RNN cell
                        node_states_per_layer[-1] =
self.gnn_weights.rnn_cells[layer_idx](incoming_information,

node_states_per_layer[-1])[1]  # Shape [V, D]

        return node_states_per_layer[-1]

    def gated_regression(self, last_h, regression_gate,
regression_transform):
        # last_h: [v x h]

        gate_input = tf.concat([last_h,
self.placeholders['initial_node_representation']], axis=-1)  # [v x
2h]
        gated_outputs = tf.nn.sigmoid(regression_gate(gate_input)) *
regression_transform(last_h)  # [v x 1]

        # Sum up all nodes per-graph
        num_nodes = tf.shape(gate_input, out_type=tf.int64)[0]
        graph_nodes =
tf.SparseTensor(indices=self.placeholders['graph_nodes_list'],

values=tf.ones_like(self.placeholders['graph_nodes_list'][:, 0],

dtype=tf.float32),

dense_shape=[self.placeholders['num_graphs'], num_nodes])  # [g x v]
        return
tf.transpose(tf.sparse_tensor_dense_matmul(graph_nodes,
gated_outputs)) # [g]

    # ----- Data preprocessing and chunking into minibatches:
    def process_raw_graphs(self, raw_data: Sequence[Any],
is_training_data: bool) -> Any:
        processed_graphs = []
        for d in raw_data:
            (adjacency_lists, num_incoming_edge_per_type) =
self.__graph_to_adjacency_lists(d['graph'])
            if self.num_graph_features > 0:
                processed_graphs.append({"adjacency_lists":
adjacency_lists,

"num_incoming_edge_per_type": num_incoming_edge_per_type,
                                         "init": d["node_features"],
                                         "labels":
[d["targets"][task_id][0] for task_id in self.params['task_ids']],
                                         'graph_features':
d['graph_features']
                                         })
            else:
                processed_graphs.append({"adjacency_lists":
adjacency_lists,

"num_incoming_edge_per_type": num_incoming_edge_per_type,
                                         "init": d["node_features"],
                                         "labels":
[d["targets"][task_id][0] for task_id in self.params['task_ids']]
                                         })

        if is_training_data:
            np.random.shuffle(processed_graphs)
            for task_id in self.params['task_ids']:
                task_sample_ratio =
self.params['task_sample_ratios'].get(str(task_id))
                if task_sample_ratio is not None:
                    ex_to_sample = int(len(processed_graphs) *
task_sample_ratio)
                    for ex_id in range(ex_to_sample,
len(processed_graphs)):
                        processed_graphs[ex_id]['labels'][task_id] =
None

        return processed_graphs

    def __graph_to_adjacency_lists(self, graph) -> Tuple[Dict[int,
np.ndarray], Dict[int, Dict[int, int]]]:
        adj_lists = defaultdict(list)
        num_incoming_edges_dicts_per_type = defaultdict(lambda:
defaultdict(lambda: 0))
        for src, e, dest in graph:
            fwd_edge_type = e - 1  # Make edges start from 0
            adj_lists[fwd_edge_type].append((src, dest))
            num_incoming_edges_dicts_per_type[fwd_edge_type][dest]
+= 1
            if self.params['tie_fwd_bkwd']:
                adj_lists[fwd_edge_type].append((dest, src))

num_incoming_edges_dicts_per_type[fwd_edge_type][src] += 1

        final_adj_lists = {e: np.array(sorted(lm), dtype=np.int32)
                           for e, lm in adj_lists.items()}

        # Add backward edges as an additional edge type that goes
backwards:
        if not (self.params['tie_fwd_bkwd']):
            for (edge_type, edges) in adj_lists.items():
                bwd_edge_type = self.num_edge_types + edge_type
                final_adj_lists[bwd_edge_type] = np.array(sorted((y,
x) for (x, y) in edges), dtype=np.int32)
                for (x, y) in edges:

num_incoming_edges_dicts_per_type[bwd_edge_type][y] += 1

        return final_adj_lists, num_incoming_edges_dicts_per_type
    def make_minibatch_iterator(self, data: Any, is_training: bool):
        """Create minibatches by flattening adjacency matrices into
a single adjacency matrix with
        multiple disconnected components."""
        if is_training:
            np.random.shuffle(data)
        # Pack until we cannot fit more graphs in the batch
        dropout_keep_prob =
self.params['graph_state_dropout_keep_prob'] if is_training else 1.
        num_graphs = 0
        while num_graphs < len(data):
            num_graphs_in_batch = 0
            batch_node_features = []
            batch_target_task_values = []
            batch_target_task_mask = []
            batch_adjacency_lists = [[] for _ in
range(self.num_edge_types)]
            batch_num_incoming_edges_per_type = []
            batch_graph_nodes_list = []
            node_offset = 0
            batch_graph_features = defaultdict(list)

            while num_graphs < len(data) and node_offset +
len(data[num_graphs]['init']) < self.params['batch_size']:
                cur_graph = data[num_graphs]
                num_nodes_in_graph = len(cur_graph['init'])
                padded_features = np.pad(cur_graph['init'],
                                         ((0, 0), (0,
self.params['hidden_size'] - self.annotation_size)),
                                         'constant')
                batch_node_features.extend(padded_features)
                batch_graph_nodes_list.extend(
                    (num_graphs_in_batch, node_offset + i) for i in
range(num_nodes_in_graph))
                for i in range(self.num_edge_types):
                    if i in cur_graph['adjacency_lists']:

batch_adjacency_lists[i].append(cur_graph['adjacency_lists'][i] +
node_offset)

                # Turn counters for incoming edges into np array:
                num_incoming_edges_per_type =
np.zeros((num_nodes_in_graph, self.num_edge_types))
                for (e_type, num_incoming_edges_per_type_dict) in
cur_graph['num_incoming_edge_per_type'].items():
                    for (node_id, edge_count) in
num_incoming_edges_per_type_dict.items():
                        num_incoming_edges_per_type[node_id, e_type]
= edge_count

batch_num_incoming_edges_per_type.append(num_incoming_edges_per_type
)

                target_task_values = []
                target_task_mask = []
                for target_val in cur_graph['labels']:
                    if target_val is None:  # This is one of the
examples we didn't sample...
                        target_task_values.append(0.)
                        target_task_mask.append(0.)
                    else:
                        target_task_values.append(target_val)
                        target_task_mask.append(1.)
                batch_target_task_values.append(target_task_values)
                batch_target_task_mask.append(target_task_mask)
                for feature in self.graph_features_list:

batch_graph_features[feature].append(cur_graph['graph_features'][fea
ture])
                num_graphs += 1
                num_graphs_in_batch += 1
                node_offset += num_nodes_in_graph

            batch_feed_dict = {
                self.placeholders['initial_node_representation']:
np.array(batch_node_features),
                self.placeholders['num_incoming_edges_per_type']:
np.concatenate(batch_num_incoming_edges_per_type, axis=0),
                self.placeholders['graph_nodes_list']:
np.array(batch_graph_nodes_list, dtype=np.int32),
                self.placeholders['target_values']:
np.transpose(batch_target_task_values, axes=[1,0]),
                self.placeholders['target_mask']:
np.transpose(batch_target_task_mask, axes=[1, 0]),
                self.placeholders['num_graphs']:
num_graphs_in_batch,
                self.placeholders['graph_state_keep_prob']:
dropout_keep_prob,
            }
            # add graph features
            for feature in self.graph_features_list:
                # have to adjust the shape to match the desired
input shape
                if self.graph_feature_lengths[feature] == 1:
                    feed = np.array([batch_graph_features[feature]],
dtype=np.float32)
                else:
                    feed =
np.transpose(batch_graph_features[feature], axes=[1, 0])

                batch_feed_dict[self.placeholders[feature]] = feed


            # Merge adjacency lists and information about incoming
nodes:
            for i in range(self.num_edge_types):
                if len(batch_adjacency_lists[i]) > 0:
                    adj_list =
np.concatenate(batch_adjacency_lists[i])
                else:
                    adj_list = np.zeros((0, 2), dtype=np.int32)

batch_feed_dict[self.placeholders['adjacency_lists'][i]] = adj_list

            yield batch_feed_dict


def main():
    args = docopt(__doc__)
    try:
        model = SparseGGNNChemModel(args)
        model.train()
    except:
        typ, value, tb = sys.exc_info()
        traceback.print_exc()
        pdb.post_mortem(tb)


if __name__ == "__main__":
    main()
```

apply_chem_tensorflow.py

```
"""
Usage:
    chem_tensorflow_sparse.py [options]

Options:
    -h --help                Show this screen.
    --config-file FILE       Hyperparameter configuration file path
(in JSON format).
    --config CONFIG          Hyperparameter configuration dictionary
(in JSON format).
    --run_dir DIR            Run dir name.
    --data_dir DIR           Data dir name.
    --restore FILE           File to restore weights from.
    --freeze-graph-model     Freeze weights of graph model
components.
    --run_id ID              Run_id.
"""

from typing import List, Tuple, Dict, Sequence, Any

from docopt import docopt
import os
from collections import defaultdict, namedtuple
import numpy as np
import tensorflow as tf
import sys, traceback
import pdb
import time
import json
import pickle

from chem_tensorflow import ChemModel, ThreadedIterator
from chem_tensorflow_sparse import GGNNWeights, SparseGGNNChemModel
from utils import glorot_init, SMALL_NUMBER


class ApplyGGNNChemModel(SparseGGNNChemModel):
    def __init__(self, args):
        super().__init__(args)

    def process_raw_graphs(self, raw_data: Sequence[Any],
is_training_data: bool) -> Any:
        processed_graphs = []
        for d in raw_data:
            (adjacency_lists, num_incoming_edge_per_type) =
self.__graph_to_adjacency_lists(d['graph'])
            if self.num_graph_features > 0:
                processed_graphs.append({"adjacency_lists":
adjacency_lists,

"num_incoming_edge_per_type": num_incoming_edge_per_type,
                                         "init": d["node_features"],
                                         "labels":
[d["targets"][task_id][0] for task_id in self.params['task_ids']],
                                         'graph_features':
d['graph_features'],
                                         "id": d["id"]
                                         })
            else:
                processed_graphs.append({"adjacency_lists": 
adjacency_lists,

"num_incoming_edge_per_type": num_incoming_edge_per_type,
                                         "init": d["node_features"],
                                         "labels": 
[d["targets"][task_id][0] for task_id in self.params['task_ids']],
                                         "id": d["id"]
                                         })

        if is_training_data:
            np.random.shuffle(processed_graphs)
            for task_id in self.params['task_ids']:
                task_sample_ratio = 
self.params['task_sample_ratios'].get(str(task_id))
                if task_sample_ratio is not None:
                    ex_to_sample = int(len(processed_graphs) * 
task_sample_ratio)
                    for ex_id in range(ex_to_sample, 
len(processed_graphs)):
                        processed_graphs[ex_id]['labels'][task_id] = 
None

        return processed_graphs

    def __graph_to_adjacency_lists(self, graph) -> Tuple[Dict[int, 
np.ndarray], Dict[int, Dict[int, int]]]:
        adj_lists = defaultdict(list)
        num_incoming_edges_dicts_per_type = defaultdict(lambda: 
defaultdict(lambda: 0))
        for src, e, dest in graph:
            fwd_edge_type = e - 1  # Make edges start from 0
            adj_lists[fwd_edge_type].append((src, dest))
            num_incoming_edges_dicts_per_type[fwd_edge_type][dest] 
+= 1
            if self.params['tie_fwd_bkwd']:
                adj_lists[fwd_edge_type].append((dest, src))

num_incoming_edges_dicts_per_type[fwd_edge_type][src] += 1

        final_adj_lists = {e: np.array(sorted(lm), dtype=np.int32)
                           for e, lm in adj_lists.items()}

        # Add backward edges as an additional edge type that goes 
backwards:
        if not (self.params['tie_fwd_bkwd']):
            for (edge_type, edges) in adj_lists.items():
                bwd_edge_type = self.num_edge_types + edge_type
                final_adj_lists[bwd_edge_type] = np.array(sorted((y, 
x) for (x, y) in edges), dtype=np.int32)
                for (x, y) in edges:

num_incoming_edges_dicts_per_type[bwd_edge_type][y] += 1

        return final_adj_lists, num_incoming_edges_dicts_per_type

    def restore_model(self, path: str) -> None:
        print("Restoring weights from file %s." % path)
        with open(path, 'rb') as in_file:
            data_to_load = pickle.load(in_file)
        # Assert that we got the same model configuration
        assert len(self.params) == len(data_to_load['params'])
        for (par, par_value) in self.params.items():
            # Fine to have different task_ids:
            if par not in ['task_ids', 'num_epochs']:
                try:
                    assert par_value == data_to_load['params'][par]
                except AssertionError:
                    print('WARNING: params dont match')
                    print('expected:
{}'.format(data_to_load['params'][par]))
                    print('got: {}'.format(par_value))

        variables_to_initialize = []
        with tf.name_scope("restore"):
            restore_ops = []
            used_vars = set()
            for variable in
self.sess.graph.get_collection(tf.GraphKeys.GLOBAL_VARIABLES):
                used_vars.add(variable.name)
                if variable.name in data_to_load['weights']:

restore_ops.append(variable.assign(data_to_load['weights'][variable.
name]))
                else:
                    print('Freshly initializing %s since no saved
value was found.' % variable.name)
                    variables_to_initialize.append(variable)
            for var_name in data_to_load['weights']:
                if var_name not in used_vars:
                    print('Saved weights for %s not used by
model.' % var_name)

restore_ops.append(tf.variables_initializer(variables_to_initialize)
)
            self.sess.run(restore_ops)
        self.writer.add_graph(self.sess.graph)
        self.merged_summary = tf.summary.merge_all()

    def make_minibatch_iterator(self, data: Any, is_training: bool):
        """Create minibatches by flattening adjacency matrices into
a single adjacency matrix with
        multiple disconnected components."""
        if is_training:
            np.random.shuffle(data)
        # Pack until we cannot fit more graphs in the batch
        dropout_keep_prob =
self.params['graph_state_dropout_keep_prob'] if is_training else 1.
        num_graphs = 0
        while num_graphs < len(data):
            num_graphs_in_batch = 0
            batch_node_features = []
            batch_target_task_values = []
            batch_target_task_mask = []
            batch_info = []
            batch_adjacency_lists = [[] for _ in
range(self.num_edge_types)]
            batch_num_incoming_edges_per_type = []
            batch_graph_nodes_list = []
            node_offset = 0
            batch_graph_features = defaultdict(list)
            while num_graphs < len(data) and node_offset +
len(data[num_graphs]['init']) < self.params['batch_size']:
                cur_graph = data[num_graphs]
                num_nodes_in_graph = len(cur_graph['init'])
                padded_features = np.pad(cur_graph['init'],
                                         ((0, 0), (0,
self.params['hidden_size'] - self.annotation_size)),
                                         'constant')
                batch_node_features.extend(padded_features)
                batch_graph_nodes_list.extend(
                    (num_graphs_in_batch, node_offset + i) for i in
range(num_nodes_in_graph))
                for i in range(self.num_edge_types):
                    if i in cur_graph['adjacency_lists']:

batch_adjacency_lists[i].append(cur_graph['adjacency_lists'][i] +
node_offset)

                # Turn counters for incoming edges into np array:
                num_incoming_edges_per_type =
np.zeros((num_nodes_in_graph, self.num_edge_types))
                for (e_type, num_incoming_edges_per_type_dict) in
cur_graph['num_incoming_edge_per_type'].items():
                    for (node_id, edge_count) in
num_incoming_edges_per_type_dict.items():
                        num_incoming_edges_per_type[node_id, e_type]
= edge_count

batch_num_incoming_edges_per_type.append(num_incoming_edges_per_type
)

                target_task_values = []
                target_task_mask = []

                for target_val in cur_graph['labels']:
                    if target_val is None:  # This is one of the
examples we didn't sample...
                        target_task_values.append(0.)
                        target_task_mask.append(0.)
                    else:
                        target_task_values.append(target_val)
                        target_task_mask.append(1.)
                batch_target_task_values.append(target_task_values)
                batch_target_task_mask.append(target_task_mask)
                batch_info.append({'id': cur_graph['id'], 'target':
target_task_values})

                for feature in self.graph_features_list:

batch_graph_features[feature].append(cur_graph['graph_features'][fea
ture])
                num_graphs += 1
                num_graphs_in_batch += 1
                node_offset += num_nodes_in_graph

            batch_feed_dict = {
                self.placeholders['initial_node_representation']:
np.array(batch_node_features),
                self.placeholders['num_incoming_edges_per_type']:
np.concatenate(batch_num_incoming_edges_per_type, axis=0),
                self.placeholders['graph_nodes_list']:
np.array(batch_graph_nodes_list, dtype=np.int32),
                self.placeholders['target_values']:
np.transpose(batch_target_task_values, axes=[1,0]),
                self.placeholders['target_mask']:
np.transpose(batch_target_task_mask, axes=[1, 0]),
                self.placeholders['num_graphs']:
num_graphs_in_batch,
                self.placeholders['graph_state_keep_prob']:
dropout_keep_prob,
            }
            # add graph features
            for feature in self.graph_features_list:
                # have to adjust the shape to match the desired
input shape
                if self.graph_feature_lengths[feature] == 1:
                    feed = np.array([batch_graph_features[feature]],
dtype=np.float32)
                else:
                    feed =
np.transpose(batch_graph_features[feature], axes=[1, 0])

                batch_feed_dict[self.placeholders[feature]] = feed


            # Merge adjacency lists and information about incoming
nodes:
            for i in range(self.num_edge_types):
                if len(batch_adjacency_lists[i]) > 0:
                    adj_list =
np.concatenate(batch_adjacency_lists[i])
                else:
                    adj_list = np.zeros((0, 2), dtype=np.int32)

batch_feed_dict[self.placeholders['adjacency_lists'][i]] = adj_list

            yield batch_feed_dict, batch_info

    def run_epoch(self, epoch_name: str, data):
        loss = 0
        maes = []
        performance = []
        start_time = time.time()
        processed_graphs = 0
        batch_iterator =
ThreadedIterator(self.make_minibatch_iterator(data, False),
max_queue_size=5)
        for step, (batch_data, batch_info) in
enumerate(batch_iterator):
            num_graphs = batch_data[self.placeholders['num_graphs']]
            processed_graphs += num_graphs


batch_data[self.placeholders['out_layer_dropout_keep_prob']] = 1.0

batch_data[self.placeholders['gated_regression_keep_prob']] = 1.0
            fetch_list = [self.ops['loss'], self.ops['MAE'],
self.merged_summary, self.ops['R2'], self.ops['predicted']]
            result = self.sess.run(fetch_list, feed_dict=batch_data)
            (batch_loss, batch_mae) = (result[0], result[1])
            loss += batch_loss * num_graphs
            maes.append(np.array(batch_mae) * num_graphs)

            summary = result[2]
            predicted = result[4].T
            try:
                assert len(batch_info)==len(predicted)
            except AssertionError:
                print('#'*100)
                print('batch info:')
                print(batch_info)
                print('#'*100)
                print('predicted:')
                print(predicted)
            for i in range(len(batch_info)):
                json_ready = []
                for j in list(predicted[i]):
                    #assumes only one target property
                    json_ready.append(float(j))
                batch_info[i]['predicted'] = json_ready
            performance.extend(batch_info)
            print("Running %s, batch %i (has %i graphs). Loss so
far: %.4f" % (epoch_name,

step,

num_graphs,

loss / processed_graphs),
                  end='\r')

        MAE = np.sum(maes, axis=0) / processed_graphs
        loss = loss / processed_graphs
        instance_per_sec = processed_graphs / (time.time() -
start_time)
        return loss, MAE, instance_per_sec, summary, result[3],
performance

    def apply(self):
        with self.graph.as_default():
            loss, MAE, instance_per_sec, summary, r, perf =
self.run_epoch("Application run", self.valid_data)
            self.writer.add_summary(summary, 0)
            full_path = os.path.join(self.log_dir,
'validation_performance.json')
            date = self.run_id.split('_')[0]
            run_id = self.run_id.split('_')[1]
            with open(full_path, 'w') as f:
                json.dump({'date': date, 'run_id': run_id, 'MAE':
float(MAE),
                           'MSE': float(loss), 'R2': float(r),
'epochs': 0}, f)

            predicted_path = os.path.join(self.log_dir,
'{}_predicted.json'.format(run_id))
            print('saving to {}'.format(predicted_path))
            with open(predicted_path, 'w') as f:
                json.dump(perf, f)


def main():
    args = docopt(__doc__)
    try:
        model = ApplyGGNNChemModel(args)
        model.apply()
    except:
        typ, value, tb = sys.exc_info()
        traceback.print_exc()
        pdb.post_mortem(tb)


if __name__ == "__main__":
    main()
```

optimiser.py

```
import pickle
import argparse
import os
import json

from hyperopt import hp # hyperparameters space
from hyperopt import tpe # the optimisation algorithm
from hyperopt import Trials # history
from hyperopt import fmin # minimalisation
from hyperopt import STATUS_OK

from chem_tensorflow_sparse import SparseGGNNChemModel


def get_max_run_id(base_dir):
    sub_dirs = glob.glob(os.path.join(base_dir, 'run_*/'))
    max_run_id = -1
    if sub_dirs:
        for d in sub_dirs:
            m = re.search('[0-9]+', d)
            run_id = int(m.group(0))
            if run_id > max_run_id:
                max_run_id = run_id
    return max_run_id


def basic_param():
    return {
        'patience': 25,
        'learning_rate': 0.001,
        'clamp_gradient_norm': 1.0,
        'out_layer_dropout_keep_prob': 0.9,
        'gated_regression_keep_prob': 0.9,
        'use_graph': True,
        'tie_fwd_bkwd': True,
        'task_ids': [0],
        'random_seed': 0,
        'use_edge_bias': False,
        'use_propagation_attention': False,
        'use_edge_msg_avg_aggregation': True,
        'task_sample_ratios': {},
        "num_epochs": 300,
        "residual_connections": {},
        "graph_rnn_activation": "ReLU"

    }


def convert_param(params):
    conf = basic_param()
    # get each parameter
    graph_descriptors = []
    if params['use_rmsd']:
        graph_descriptors.append('RMSD')
    if params['use_h-dim']:
        graph_descriptors.append('H_dims')

    edge_types = ['SINGLE', 'DOUBLE', 'TRIPLE', 'AROMATIC']
    if params['use_h-bond']:
        edge_types.append('HBOND')
    if params['use_vdw-intra']:
        edge_types.append('VDW_INTRA')
    if params['use_vdw-inter']:
        edge_types.append('VWD_INTER')

    graph_representation_size = int(params['graph_vector'])

    hidden_size = int(params['node_vector'])

    prediction_layers_architecture = [int(params['p_layer_1']),
int(params['p_layer_2'])]

    layer_timesteps = [int(params['rnn_timestep']['rnn_timestep'])
for _ in range(int(params['rnn_timestep']['rnn_layers']))]

    graph_rnn_cell = params['rnn_cell']

    # put it in the conf
    conf['graph_descriptors'] = graph_descriptors
    conf['edge_types'] = edge_types
    conf['graph_representation_size'] = graph_representation_size
    conf['hidden_size'] = hidden_size
    conf['prediction_layers_architecture'] =
prediction_layers_architecture
    conf['layer_timesteps'] = layer_timesteps
    conf['graph_rnn_cell'] = graph_rnn_cell

    # from globals
    conf['train_file'] = train
    conf['valid_file'] = valid
    return conf


def set_up_dir():
    run_id = get_max_run_id(base_dir)
    run_id += 1
    while True:
        try:
            full_dir = os.path.join(base_dir,
'run_{}'.format(run_id), '')
            assert os.path.isdir(full_dir) is False
            break
        except AssertionError:
            print('failed at creating run directory')
            print('run_id is: {}'.format(run_id))
            print('will try next number up')
            run_id += 1

    os.makedirs(full_dir)

    return full_dir, run_id


def objective(params):
    conf = convert_param(params)
    full_dir, run_id = set_up_dir()

    args = {'conf': conf, '--run_id': run_id, '--run_dir': full_dir}

    model = SparseGGNNChemModel(args)
    model.train()
    r = model.best_r
    loss = 1 - r
    return {'loss': loss, 'status': STATUS_OK, 'run_id': run_id,
'param': params, 'config': conf}


def get_trials(trial_name=None):
    if trial_name:
        trial_dir = os.path.join(base_dir, trial_name)
        if os.path.isfile(trial_dir):
            trials = pickle.load(open(trial_dir, 'rb'))
    else:
        trials = Trials()

    return trials


def define_domain_space():
    space = {
        'use_rmsd': hp.choice('use_rmsd', [False, True]),
        'use_h-dim': hp.choice('use_h-dim', [False, True]),
        'use_h-bond': hp.choice('use_h-bond', [False, True]),
        'use_vdw-intra': hp.choice('use_vdw-intra', [False, True]),
        'use_vdw-inter': hp.choice('use_vdw-inter', [False, True]),
        'graph_vector': hp.quniform('graph_vector',100,1500,100),
        'node_vector': hp.quniform('node_vector', 30, 150, 10),
        'p_layer_1': hp.quniform('p_layer_1', 40, 600, 20),
        'p_layer_2': hp.quniform('p_layer_2', 10, 300, 10),
        'rnn_timestep': hp.choice('rnn_timestep', [{'rnn_timestep':
1, 'rnn_layers': hp.quniform('rnn_layers_1', 1, 5, 1)},
                                                   {'rnn_timestep':
2, 'rnn_layers': hp.quniform('rnn_layers_2', 1, 2, 1)},
                                                   {'rnn_timestep':
3, 'rnn_layers': 1}
                                                   ]),
        'rnn_cell': hp.choice('rnn_cell', ['GRU', 'RNN']),

    }

    return space


def define_mol_only_domain_space():
    space = {
        'use_rmsd': hp.choice('use_rmsd', [False]),
        'use_h-dim': hp.choice('use_h-dim', [False]),
        'use_h-bond': hp.choice('use_h-bond', [False]),
        'use_vdw-intra': hp.choice('use_vdw-intra', [False]),
        'use_vdw-inter': hp.choice('use_vdw-inter', [False]),
        'graph_vector': hp.quniform('graph_vector', 300, 1200, 100),
        'node_vector': hp.quniform('node_vector', 70, 120, 10),
        'p_layer_1': hp.quniform('p_layer_1', 200, 500, 20),
        'p_layer_2': hp.quniform('p_layer_2', 80, 300, 20),
        'rnn_timestep': hp.choice('rnn_timestep',
                                  [{'rnn_timestep': 1, 'rnn_layers':
hp.quniform('rnn_layers_1', 1, 8, 1)},
                                   {'rnn_timestep': 2, 'rnn_layers':
hp.quniform('rnn_layers_2', 1, 4, 1)},
                                   {'rnn_timestep': 3, 'rnn_layers':
hp.quniform('rnn_layers_3', 1, 3, 1)},
                                   {'rnn_timestep': 4, 'rnn_layers':
hp.quniform('rnn_layers_4', 1, 2, 1)},
                                   {'rnn_timestep': 5, 'rnn_layers':
1}
                                   ]),
        'rnn_cell': hp.choice('rnn_cell', ['RNN']),
    }

    return space


def define_intermol_interact_search_space():
    space = {
        'use_rmsd': hp.choice('use_rmsd', [False]),
        'use_h-dim': hp.choice('use_h-dim', [False]),
        'use_h-bond': hp.choice('use_h-bond', [False, True]),
        'use_vdw-intra': hp.choice('use_vdw-intra', [False]),
        'use_vdw-inter': hp.choice('use_vdw-inter', [False, True]),
        'graph_vector': hp.quniform('graph_vector', 300, 1200, 100),
        'node_vector': hp.quniform('node_vector', 60, 140, 10),
        'p_layer_1': hp.quniform('p_layer_1', 200, 500, 20),
        'p_layer_2': hp.quniform('p_layer_2', 80, 300, 20),
        'rnn_timestep': hp.choice('rnn_timestep',
                                  [{'rnn_timestep': 1, 'rnn_layers':
hp.quniform('rnn_layers_1', 1, 8, 1)},
                                   {'rnn_timestep': 2, 'rnn_layers':
hp.quniform('rnn_layers_2', 1, 4, 1)},
                                   {'rnn_timestep': 3, 'rnn_layers':
hp.quniform('rnn_layers_3', 1, 3, 1)},
                                   {'rnn_timestep': 4, 'rnn_layers':
hp.quniform('rnn_layers_4', 1, 2, 1)},
                                   {'rnn_timestep': 5, 'rnn_layers':
1}
                                   ]),
        'rnn_cell': hp.choice('rnn_cell', ['RNN', 'GRU']),
    }

    return space


def run_optimiser(batch_size, max_evals, space, trials, algorithm):
    current_eval = len(trials)
    pickle_f = True

    while current_eval < max_evals:
        if max_evals - current_eval < batch_size:
            current_eval = max_evals
        else:
            current_eval += batch_size

        _ = fmin(objective, space=space, algo=algorithm,
trials=trials, max_evals=current_eval)
        # best not taken as trials includes everything + can get run
id

        if pickle_f:
            pickle_f = False
            pickle_name = 'trials_dump_1.pickle'
        else:
            pickle_f = True
            pickle_name = 'trials_dump_2.pickle'
        pickle.dump(trials, open(os.path.join(base_dir,
pickle_name), 'wb'))

    return trials


def get_best_param(trials):
    min_loss = float('+inf')
    best_id = None
    best_param = {}
    conf = {}
    for i, run in enumerate(trials.results):
        if run['loss'] < min_loss:
            min_loss = run['loss']
            best_id = run['run_id']
            best_param = run['param']
            conf = run['config']
    print('best run: {}'.format(best_id))
    best_param['id'] = best_id
    json.dump(best_param, open(os.path.join(base_dir,
'best_hyperparam.json'), 'w'))
    json.dump(conf, open(os.path.join(base_dir, 'best_conf.json'),
'w'))


def main():
    parser = argparse.ArgumentParser("""Hyperparameter
optimisation""")
    parser.add_argument("--step_size", default=2, type=int,
help='number of iteration before backing up')
    parser.add_argument("--max_eval", default=100, type=int,
help='max number of iterations')
    parser.add_argument("--restore", type=str, help='restore from
trails pickle')
    parser.add_argument('-d', '--dir', type=str,
default=os.getcwd(), help='project directory with datasets.')
    parser.add_argument('--mol', action='store_true', help='use
molecular information only')
    parser.add_argument('--int_mol', action='store_true', help='use
mol info + intermol interaction')
    parser.add_argument('train', help='training set')
    parser.add_argument('valid', help='validation set')
    args = parser.parse_args()
    global base_dir, train, valid
    base_dir = args.dir
    train = args.train
    valid = args.valid

    batch_size = args.step_size
    max_evals = args.max_eval
    if args.mol:
        space = define_mol_only_domain_space()
    elif args.int_mol:
        space = define_intermol_interact_search_space()
    else:
        space = define_domain_space()
    trials = get_trials(args.restore)
    algorithm = tpe.suggest

    try:
        trials = run_optimiser(batch_size, max_evals, space, trials,
algorithm)
    finally:
        pickle.dump(trials, open('trials_dump.pickle', 'wb'))
        get_best_param(trials)


if __name__ == '__main__':
    main()
```