



The
University
Of
Sheffield.

PhD Thesis in Machine Learning for Personalized Medicine

**Bringing Models to the Domain: Deploying Gaussian
Processes in the Biological Sciences**

Max Zwieße

October 12, 2017

Supervisor

Neil D. Lawrence
University of Sheffield

Magnus Rattray
University of Manchester

Name, first name: Zwießele, Max

Registration No: 120261372

Department of Computer Science

Bringing Models to the Domain: Deploying Gaussian Processes in the Biological Sciences

PhD Thesis

University of Sheffield

Period: 01.04.2013-31.3.2017

Thesis Deposit Agreement

1. I, the author, confirm that the Thesis is my own work, and that where materials owned by a third party have been used, copyright clearance has been obtained. I am aware of the University's Guidance on the Use of Unfair Means (www.sheffield.ac.uk/ssid/exams/plagiarism).
2. I confirm that all copies of the Thesis submitted to the University, whether in print or electronic format, are identical in content and correspond with the version of the Thesis upon which the examiners based their recommendation for the award of the degree (unless edited as indicated above).
3. I agree to the named Thesis being made available in accordance with the conditions specified above.
4. I give permission to the University of Sheffield to reproduce the print Thesis (where applicable) in digital format, in whole or part, in order to supply single copies for the purpose of research or private study for a non-commercial purpose. I agree that a copy of the eThesis may be supplied to the British Library for inclusion on EThOS and WREO, if the thesis is not subject to an embargo, or if the embargo has been lifted or expired.
5. I agree that the University of Sheffield's eThesis repository (currently WREO) will make my eThesis (where applicable) available over the internet via an entirely non-exclusive agreement and that, without changing content, WREO and/or the British Library may convert my eThesis to any medium or format for the purpose of future preservation and accessibility.
6. I agree that the metadata relating to the eThesis (where applicable) will normally appear on both the University's eThesis server (WREO) and the British Library's EThOS service, even if the eThesis is subject to an embargo.

Signature _____ Date _____
Max Zwiesssele

University stamp

Abstract

Recent developments in single cell sequencing allow us to elucidate processes of individual cells in unprecedented detail. This detail provides new insights into the progress of cells during cell type differentiation. Cell type heterogeneity shows the complexity of cells working together to produce organ function on a macro level. The understanding of single cell transcriptomics promises to lead to the ultimate goal of understanding the function of individual cells and their contribution to higher level function in their environment.

Characterizing the transcriptome of single cells requires us to understand and be able to model the latent processes of cell functions that explain biological variance and richness of gene expression measurements. In this thesis, we describe ways of jointly modelling biological function and unwanted technical and biological confounding variation using Gaussian process latent variable models. In addition to mathematical modelling of latent processes, we provide insights into the understanding of research code and the significance of computer science in development of techniques for single cell experiments.

We will describe the process of understanding complex machine learning algorithms and translating them into usable software. We then proceed to applying these algorithms. We show how proper research software design underlying the implementation can lead to a large user base in other areas of expertise, such as single cell gene expression. To show the worth of properly designed software underlying a research project, we show other software packages built upon the software developed during this thesis and how they can be applied to single cell gene expression experiments.

Understanding the underlying function of cells seems within reach through these new techniques that allow us to unravel the transcriptome of single cells. We describe probabilistic techniques of identifying the latent functions of cells, while focusing on the software and ease-of-use aspects of supplying proper research code to be applied by other researchers.

Acknowledgements

First of all, I thank Neil Lawrence for helping me through the process of writing a PhD thesis and providing guidance along the way.

Special thanks go to Karsten Borgwardt to sending me on this path, Bertram Müller Myhsok and Volker Tresp for letting me second in their respective research labs.

I also thank the members of my work group in Sheffield - Michael Croucher, Zhenwen Dai, Andreas Damianou, Nicolo Fusi, Javier Gonzalez, James Hensman, Alan Saul and Michael Smith - for useful, inspiring discussion and proper proofreading. Additionally I thank Sarah Teichmann and Aleksandra Kolodziejczyk for support in biological questions and interpretation of results, as well as providing biological data for analysis.

Finally, I thank my parents Sibylle and Frieder for their patience and assistance in helping me finishing this thesis.

I am grateful for financial support from the European Union 7th Framework Programme through the Marie Curie Initial Training Network “Machine Learning for Personalized Medicine” MLPM2012, Grant No. 316861.



Contents

Abstract	v
Acknowledgements	vii
Contents	ix
List of Figures	xiii
List of Tables	xv
Nomenclature	xvi
1 Introduction	1
1.1 Contribution and Roadmap	3
1.2 Biological Background - From DNA to Protein	4
1.2.1 Discovery and Structure of DNA	4
1.2.2 Functional View on DNA – Genes, Expression and Proteins	6
1.3 Machine Learning	8
1.4 Research Code	10
1.4.1 Tutorials	11
1.4.2 Codebase & Knowledge of Algorithm	11
1.4.3 Making Algorithms Accessible	12
1.4.4 Github	13
1.4.5 Automating Code Correctness	13
1.4.6 Summary	14

2	Methods	15
2.1	Gaussian Process Regression	16
2.1.1	Gradients	18
2.1.2	Gaussian Process Prior (Covariance Function)	20
2.1.3	Prediction	24
2.1.4	Example and Sample	24
2.1.5	ARD: Automatic Relevance Determination	25
2.2	Principal Component Analysis	27
2.3	Gaussian Process Latent Variable Model (GPLVM)	30
2.3.1	Inferring Subspace Dimensionality	32
2.4	Sparse Gaussian Process Regression	33
2.4.1	Optimization and Complexity	35
2.4.2	Implementation	35
2.4.3	Prediction	35
2.4.4	Intuition	36
2.5	Variational Bayesian GPLVM	37
2.5.1	On ARD Parameterization in Bayesian GPLVM	38
2.5.2	Implementation	39
2.5.3	Factorization and Parallelization	39
2.5.4	Large Scale Bayesian GPLVM	41
2.6	MRD: Manifold Relevance Determination	42
2.6.1	Intuition and Simulation	43
3	Case Study GPy	49
3.1	Splitting the Algorithm into Parts	50
3.2	Likelihood & Prior	51
3.3	Inference	51
3.4	Numerical Stability	52
3.5	Posterior Prediction	54
3.6	Gradients	55
3.7	Optimization	56

3.8	Mean Function	57
3.8.1	Gradients	57
3.9	Bringing it all Together	58
3.10	Comparison to other implementations	59
3.10.1	GPFlow	59
3.10.2	scikit-learn	60
3.11	Plotting and Visualization	61
3.12	Parameterization with the Paramz Framework	62
3.13	Summary	65
4	Applications in Gene Expression Experiments	67
4.1	Topslam: Topographical Simultaneous Localization and Mapping . .	68
4.1.1	State-of-the-Art Pseudo Time Ordering Extraction	68
4.1.2	Topslam	70
4.1.3	Comparison of Dimensionality Reduction Techniques using Differentiation Profile Simulations	73
4.1.4	Comparison of Pseudo Time Extraction Techniques using Dif- ferentiation Profile Simulations	76
4.1.5	Probabilistic Waddington’s Landscape for other Dimension- ality Reduction Techniques	76
4.1.6	Runtime & Complexity	78
4.1.7	Bayesian GPLVM vs. Topslam	79
4.2	Mouse Embryonic Development Landscape	80
4.2.1	Model Optimization using GPy	80
4.2.2	Differential Expression in Time	82
4.3	T Helper Cell Differentiation	84
4.3.1	Data Description and Cleanup	85
4.3.2	MRD Application	86
4.3.3	Pseudo Time Ordering	87
4.3.4	Split Detection	87
4.3.5	Differential Gene Expression	88

4.3.6	Summary	89
5	Discussion and Conclusions	91
5.1	Packages Based on GPy	91
5.2	"Bridging the Gap"	92
5.3	Understanding and Intuition	93
5.3.1	Example: Initialization	94
5.3.2	Example: Optimization	94
5.4	Conclusions	95
A	Mathematical Details	96
A.1	GP Posterior Inference	96
A.2	Latent Function Posterior	96
A.3	Sparse GP: Conditional of Inducing Outputs given Observed Data	97
B	Software	99
B.1	Python	99
B.1.1	SciPy	99
B.1.2	Seaborn	100
	References	101

List of Figures

1.1	Double helix structure of DNA	5
1.2	From DNA to Protein	7
1.3	Waddington landscape surface.	8
2.1	Three different inverse lengthscales for a GP with an exponentiated quadratic covariance function.	23
2.2	Plotted are mean (thick line) and 95% confidence interval (shaded area) of a GP constraint to increasing amount of observed data.	25
2.3	Three two Dimensional Gaussian Process Samples for ARD	26
2.4	ARD parameter simulation for ARD selection intuition.	27
2.5	Principal component analysis on two dimensional sample phenotype.	28
2.6	Sparse GP plots. The inducing input locations differ to show the effect of their position on the underlying GP.	37
2.7	Simulated data for MRD	45
2.8	MRD simulation model plots and weights.	46
3.1	Illustration of GPy plotting capabilities.	66
4.1	Differentiation Process Simulations	73
4.2	Distribution of squared distance in the three different data sets	74
4.3	Extracted landscape and correlation to simulated time for one simulated dataset.	74
4.4	Comparison of correlation between pseudotimes extracted from other dimensionality reduction techniques landscapes.	77

4.5	Compare correlation between pseudotimes extracted from other dimensionality reduction techniques landscapes and learning of probabilistic Waddington's landscape from those landscapes.	78
4.6	Runtimes of Topslam on all five simulated differentiation profiles (Fig. 4.1). Runtimes include model learning, distance correction and pseudo time ordering.	79
4.7	Bare Bayesian GPLVM vs Topslam.	79
4.8	Plotting the magnification factor with GPy.	81
4.9	Comparison of dimensionality reduction techniques on Guo et al. [29] dataset.	83
4.10	Distance Correction Visualization.	84
4.11	Marker gene detection by differentiation path selection along time line	85
4.12	Relevance parameters for gene subsets of T cell differentiation.	86
4.13	Marker gene progression in extracted landscape for T helper progression.	87
4.14	Pseudo time ordering extraction for T helper differentiation progression.	88
4.15	Overlapping mixture of Gaussians for T helper cell differentiation assignment.	89
4.16	Differential expression between Th1 and Th2 cell states for each gene.	90

List of Tables

2.1	Some covariance functions with a plot for intuition.	21
2.2	MRD simulation structure for intuitional insight into the model. . . .	45
3.1	Comparison of GPy versus GPflow and scikit-learn.	60
4.1	BGPLVM vs. other dimensionality reduction.	75
4.2	Pearson correlation coefficients between simulated and extracted time.	76
4.3	Differential gene expression between terminal cell stages of mouse embryonic stem cells.	83

Nomenclature

$N \in \mathbb{N}$	Natural number
$x \in \mathbb{R}$	Real number
$\mathbf{x} \in \mathbb{R}^N$	Vector \mathbf{x} of size $N \times 1$
$\mathbf{X} = (\mathbf{x}_d)_{1 \leq d \leq D} \in \mathbb{R}^{N \times D}$	Matrix \mathbf{X} of size $N \times D$
$\mathbf{I}_N \in \mathbb{R}^{N \times N}$	Identity matrix of size $N \times N$ - Matrix of ones on diagonal
$\mathbf{I}_{N \times D} \in \mathbb{R}^{N \times D}$	Unit matrix of size $N \times D$, padded with zeros appropriately
\mathbf{X}^\top	Transpose of \mathbf{X} , such that $\mathbf{X}_{ij} = \mathbf{X}_{ji}^\top$ and $\mathbf{X}^\top \in \mathbb{R}^{D \times N}$
$ \mathbf{X} $	Determinant of \mathbf{X}
$\mathbf{X} \circ \mathbf{Y}$	Hadamard (element wise) product of \mathbf{X} & \mathbf{Y}
$\mathbb{E}_{p(\mathbf{X})}(p(\mathbf{Y})) = \langle p(\mathbf{X}) \rangle_{p(\mathbf{X})}$ $= \int p(\mathbf{Y})p(\mathbf{X}) d\mathbf{X}$	Expected value of $p(\mathbf{Y})$ under the distribution $p(\mathbf{X})$
$p(\mathbf{Y} \mathbf{X}) = \prod_{i=1}^D p(y_{\cdot i} \mathbf{X}) = \prod_{\mathbf{y} \in \mathbf{Y}} p(\mathbf{y} \mathbf{X})$	Factorisation of probability distribution if not stated otherwise. $\mathbf{y} \in \mathbf{Y}$ are the columns of \mathbf{Y} .
$\mathcal{N}(\mathbf{Y} \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_{\mathbf{y} \in \mathbf{Y}} \mathcal{N}(\mathbf{y} \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Iid. factorisation of a Gaussian if not stated otherwise.

Chapter 1

Introduction

The past decade stood under the banner of *data science*. Data science is the science of extracting information from unstructured and, commonly, heterogeneous data. Many branches of research are increasingly focusing on more unstructured data, and letting the model resolve its complexity to reveal the underlying signal. This has led to a significant increase in the number of implementations and software packages published in many areas of life sciences [14]. In this thesis, we will describe ways of ensuring implementation quality, reusability and extendability of software to analyse complex microbiological data and questions. We use probabilistic modelling to incorporate the quantitative measurements taken from biological systems. We perform pattern recognition to facilitate biologists to extract insights from the system and incorporate prior knowledge from the biologist to increase signal discovery.

In biology, the advances in revealing molecular biological quantitative traits has made data science a big driver of scientific discoveries in molecular biology. *Next generation sequencing* [76] technologies facilitate measuring the expression of all genes in whole tissues. Genes are the basic unit for heritable information in life [59]. The amount of gene expression in the cell machinery provides a quantitative measure for the amount of protein present (Sec. 1.2). Proteins are the basic actors in cell activity and are therefore crucial in identifying cell function. The ability to consider these quantitative measures allows for new and exciting ways of unraveling the underlying mechanisms and functional relations inside biology on a molecular basis.

Traditionally, descriptive analytics was used to describe data. Prior to drawing conclusions, data was carefully filtered and (hand) selected for analysis. Advances in *machine learning* techniques allowed data to be used in predictive analytics. That is, we supply the machine learning technique with ways to reveal which parts of the data are informative for the prediction of variables of interest. This initiated the

so-called “big data” trend. The more data an appropriate algorithm sees, the better it gets at estimating probable outcomes. Not only more data, but also different sources and heterogeneity of data improve prediction [38; 58; 72].

For example, if the big data comes from molecular measurements of single cells of a tissue [42], we can extract biological answers about the inter-relationship of cells within that tissue. The nature of machine learning and data science algorithms means more data amounts to more predictive power. Thus, the amount of certainty in biological insight increases with the amount of measurements of cells we extract. By measuring the gene expression of cells from different tissues, scientists are able to identify genes, which are active in only one of the tissue types. This gives hints towards the function of genes in tissues overall, by assigning the tissue function to the identified genes. In a tissue, cells work together to perform the function of this part of the body. However, recent studies show evidence for more heterogeneity between cells than previously assumed [42]. Even within previously assumed to be homogeneous cell types, new sub types arise [79]. Separating single cells and individually measuring their gene expression enables us to distinguish cell types on an unprecedented level of detail. And with this level of detail new challenges arise for the handling of new types of confounding variation in collected measurements. The low amounts of molecules within one cell require amplification. A failure of amplification results in a so called *dropout* event and the particular gene is wrongly assigned not to be expressed. This is not the only new source of variation: heterogeneity between individual cells, experimental circumstances and biological variation have a large impact on the function of a cells genes [79]. In the process of measuring this gene expression, many technical steps need to be performed [42]. This means there is a high probability for experimental influences to generate effects not associated to the variable of interest. Also, biological variability between subjects (or cells) can influence the outcome in a non expected manner. For that, *confounder correction* is required to reveal the real effect of the variable of interest [52].

The most commonly measured tissue is blood and the information contained in it. For many diseases blood is the carrier of nutrients and helper cells, which help at the attacked site. Clinical diagnostics still mostly rely on chemical blood measurements, not including gene expression as a diagnostic tool. Thus, improving the analysis and throughput of genetic measurements on a singular cell basis will ultimately improve diagnostics in a clinical setting, maybe even allowing the prevention of disease before onset.

1.1 Contribution and Roadmap

In this thesis, we describe the full picture of implementing complex (machine learning) algorithms from a research software design perspective. Additionally, we show how to apply these techniques and results from machine learning algorithms, when applied to single cell gene expression experiments. The thesis comprises the combination of three major parts of research to a fully functional application of machine learning to biological data: research software engineering, machine learning and gene expression measurements. The major contributions of this thesis are in the research software design for a proper foundation of code, proper understanding of the machine learning tools and the application to single cell gene expression experiments.

In the following sections, we will first introduce the biological background (Sec. 1.2) of gene expression experiments. Second, we will introduce the general idea of the main probabilistic machine learning algorithm used in this thesis (Sec. 1.3). Third, we will show general introductory insights into research code and the difficulties which arise during development (Sec. 1.4). During this, we will elucidate general ways of keeping a code base clean and having testing suites to be confident about the research code. This is well known in software design and has to be taken seriously by the machine learning community to produce reproducible results.

The following chapters of this thesis include the methodological chapter (Cha. 2), explaining Gaussian process (GP) based machine learning techniques. The first chapter explains basic GP regression and following steps to get to the main machine learning tool used in this thesis, the variational Bayesian Gaussian process latent variable model. This model is a way of learning the inputs \mathbf{X} for a mapping $f(\mathbf{X}) = \mathbf{Y}$ with a GP constraint on f . All of these models were implemented during the course of this thesis and are supplied in GPy. GPy is the software package alongside this thesis and a large portion of GPy represents a part of the contributions of this thesis. We will introduce the basic steps of software engineering and design in detail in the case study around GPy (Cha. 3), showing the implementation of Gaussian processes. This chapter also includes and explains the steps for software to supply a proper extendible code base to be built upon. In Chapter 4, we show applications of machine learning tools to single cell gene expression. Here, we also show Topslam (Sec. 4.1), which is a method to unravel the ordering of single cells. It makes use of the probabilistic nature of the machine learning technique and estimates a landscape on which distances may be distorted. Topslam corrects for these distortions to supply a more stable estimate of the ordering of cells. In the following sections we will first apply Topslam to a real world experiment of early

embryonic development in mice (Sec. 4.2), extracting the ordering of cells in early development only from snap shot gene expression measurements taken in series. We then (Sec. 4.3) conclude the results by applying machine learning tools which were developed in cooperation, or solely, by other researchers on top of GPy. This chapter shows the expanse of GPy and its usage outside of the scope of this thesis. It provides supplement to the main point of this thesis: bringing machine learning to the domain.

In summary, this thesis shows how to apply software engineering in a machine learning environment, supply the machine learning tools to other areas of expertise and problems which may arise during such a development. We show the solutions we found during the course of this thesis and tackle communication and translation hurdles when working across areas of expertise.

1.2 Biological Background - From DNA to Protein

Most heritable information in life is written in the DNA of an organism. Species specific information separates the different branches on the tree of life. A *gene* is one unit of heritable information. Complex combinations of temporal activation of sets of genes define the function and differentiation of cells along the life of an organism. Most prominently shown in the early stages of development, one can distinguish patterns of cells dividing into the different parts of the body. In this thesis we focus on revealing these functional features of cells and their intrinsic signals using machine learning techniques (Sec. 1.3) [98].

1.2.1 Discovery and Structure of DNA

Gregor Mendel first described heritability from breeding experiments with peas. In 1865 he discovered, that the different phenotypes of pea flowers can be explained by specific ratio laws [59]. To this date, these laws are called and used as the “Mendelian laws”. The first isolation of DNA was by Friedrich Miescher in Tübingen, but at the time he failed to realize the importance of his discovery. In 1909 Wilhelm Johannsen described one unit of heredity with the word *gene*. In 1953 the double-helix structure of DNA was revealed by Watson and Crick [94]. DNA consists of four bases, connected by a sugar-phosphate *backbone*. The atoms are connected from 5' C atom to 3' C of the base, which gives the DNA strand its direction (from 5' to 3'). These bases, called *nucleotides*, build up two complementary linearly composed strands, which form a *double helix*. Two nucleotides respectively form hydrogen bonds, which stabilize the double helix. The four nucleotides are Adenine, Thymine, Guanine and Cytosine. A and T can bind to each other by form-

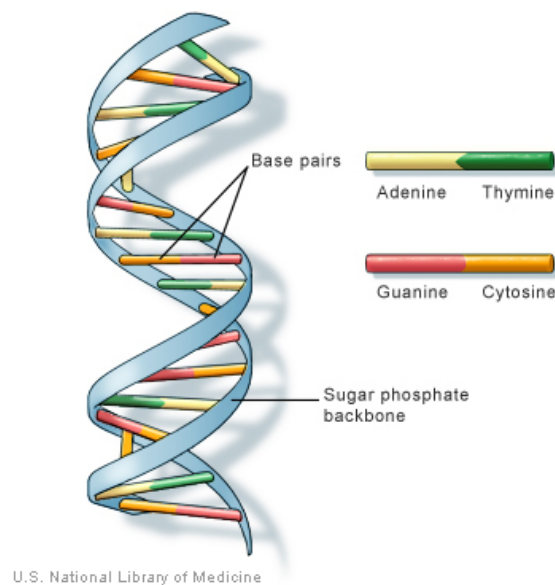


Figure 1.1: Double helix structure of the DNA. The double helix is built up by linearly linked nucleotides - connected by a sugar-phosphate backbone. Hydrogen-bonds form only between A and T and between G and C. U.S. National Library of Medicine [89]

ing two hydrogen bonds. Therefore, A and T are said to be *complementary*. G and C are also complementary: they form three hydrogen bonds. This complementary strand is called *complementary DNA (cDNA)*. When two fitting cDNA strands bind to each other they form a double helix, mentioned above. The binding of two complementary strands is called *hybridisation*, where the complementary strands are also directionally complementary [97]. Inside the cell, DNA is always present in double helix form. Figure 1.1 depicts the double helix form with its backbone and nucleotides [98].

Crick et al. [17] describe the *transcription* from DNA to RNA and the triplet coded translation from RNA to protein. Sanger et al. [73] described the first *DNA sequencing* techniques. This means reading the information written in the DNA molecules of organisms as the four nucleotides. The next years revealed many different properties of DNA. The first *microarray technologies* were developed in the 1980s, enabling gene expression quantification utilising RNA molecule counts. We use the RNA count as a proxy of protein activity, by assumption. The first full sequenced human genome was published in 2001 by Lander et al. [46].

In the years since 1865 many more discoveries about DNA and its properties were made, and describing all of them would go beyond the scope of this thesis. Here we will focus on the analysis of gene expression as a quantitative trait of cells

to elucidate function, relation and properties of the building blocks of life [98].

1.2.2 Functional View on DNA – Genes, Expression and Proteins

In all life on earth the DNA holds the heritable information from one generation to the next. We divide the DNA sequence of nucleotides into regions: *coding* and *non-coding* regions.

The coding regions hold the genes, which are transcribed and translated into proteins. Non-coding regions take part in secondary processes (see more for example in Ahnert et al. [2]; Mercer et al. [60]).

A coding region gets transcribed and translated into protein. It is flanked by specific start and stop codons (triplets of nucleotides encoding an amino acid). Flanking this, untranslated regions (UTR, “empty spaces”) can hold enhancers or structural features to influence the expression of the gene. The coding region consists of *introns* and *exons*. DNA is transcribed into pre-messenger ribonucleic acid (RNA) before splicing takes place. The introns are spliced (“cut”) out during transcription, so that only the exons are left for the protein. This messenger RNA gets chemically sealed by a 5' cap and 3' poly-A tail. Protected against degrading proteins, it now traverses the nucleus membrane into the intracellular space. Here ribosomes translate the mRNA into the corresponding protein. Figure 1.2 gives an overview over the expression process of proteins.

The number of RNA molecules measured in a cell is known as *gene expression*. We assume relative differences between gene expression of different genes as a proxy for relative amounts of protein of that gene in the cell. Proteins are the active component in a cell performing the function of the gene. The functionality of the gene is to provide the information for the protein it encodes. Interaction with other genes and non-coding regions or proteins changes the expression of a gene over time. This enables a cell to respond to differing environmental circumstances in and around it. Thus, if we measure a gene to be differentially expressed under certain (changes of) circumstances, we can, with high likelihood, assign this gene (and its transcript) to respond or take part in the handling of this circumstance.

Not only other genes and proteins can influence gene expression. The non coding regions of DNA contain so-called *biomarkers*, which are sequence elements, that can influence expression. One of which is a mutation at one singular point – a single nucleotide polymorphism (SNP) – in the genetic code. If the SNP is in and around a coding region, it can influence its expression. SNPs can occur inside a gene, so they can directly influence the protein sequence and therefore functionality. They can also occur far away from the gene they interact with, supplying structural changes to the far away region or binding affinities to translation factors,

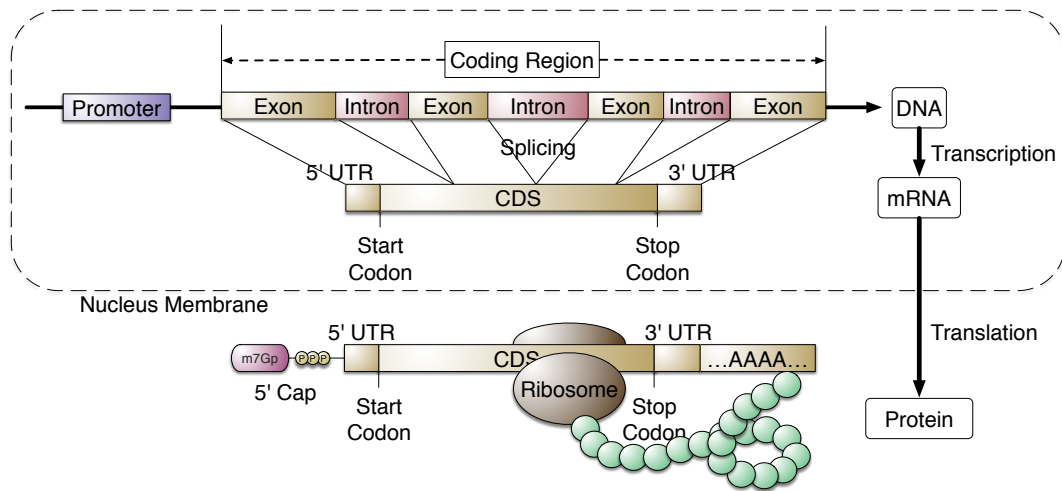


Figure 1.2: Prior to the coding region of a gene is its promoter on the DNA. The coding region is transcribed into mRNA, and introns are spliced away. The exons form the protein coding sequence (CDS), flanked by untranslated regions and the start and stop codon of the protein. The resulting mRNA gets chemically sealed by a 5' cap and a 3' poly-A tail. Protected from degrading proteins it passes the nucleus membrane and gets translated into a protein in the cell cytoplasm (depicted as green folding chain) by ribosomes running along the mRNA. Taken with author approval from [98]

which then ultimately alter the gene expression from a distance. A SNP can also influence proteins outside the nucleus through so called *microRNA*, which can pass the nucleus membrane and act upon proteins.

As already mentioned, gene expression changes over time. Waddington [92] describes the differentiation pattern of genes as a ball rolling down a landscape stochastically deciding at junctions which way to go. The ball rolling represents time and the landscape represents the environment around the gene, influencing its expression. The landscape Waddington describes is depicted in Figure 1.3.

In the original of Waddington's landscape the mechanisms forming the shape of the landscape (think of strings, pulling the surface down at the right places) are biomarkers, epigenetic modification and environmental influences, deciding which differentiation profiles are possible for the cells. This ensures reproducing cells in specific organs only differentiate into cells of that organ and not other body parts. To model the gene expression over time, we need to rely on flexible models. Flexible enough to describe the changes over time, caused by the influences described above. We observe the influences and functionality only indirectly through the expression value of genes. The model has to take relative expression patterns to other genes into account and model the underlying tasks arising from those. All of the above proves modelling gene expression for heterogeneous populations, such as single cell measurements, a complex task.

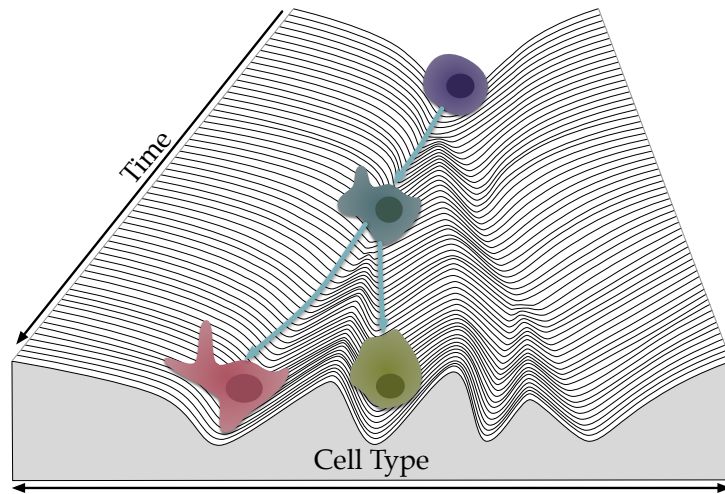


Figure 1.3: Waddington landscape surface. Depicted is the landscape along which cells differentiate, progressing in time. Time acts like gravity on a ball rolling down a hill. At junction points, cells decide which way to go. Differentiation mechanisms usually enforce a distribution on the cells, so that the right fractions of cells is accomplished at the differentiated stage. [Figure reproduced from Waddington [92]]

As an intuition for Waddington’s landscape we like to think of a skiing piste, on which it is easier to ski down the made paths, as opposed to the rough non made bits. We will use this idea in the Topslam (Sec. 4.1) algorithm. In Topslam we reinterpret the deterministic mechanical idea of the landscape to a probabilistic one. The probabilistic nature of the underlying model used (Sec. 2.5) allows us to extract a landscape for cells, giving them relative positions on the landscape. By following the topology of the landscape, we can deduce relative distances and correct for possible hills between cells. This reduces outliers and unwanted variation.

In this section, we have provided an overview of some aspects of genetics that are key to understanding this thesis. This is by no means a full description of molecular biology. For a more detailed view on genetics and epigenetics refer to the books of Hennig [30]; Seyffert and Balling [75]; Zien [97].

1.3 Machine Learning

In the past decade the machine learning method became increasingly popular. In this kind of algorithm, we try to give the machine as much freedom as necessary to be able to extract (‘learn’) patterns in data. In machine learning it is common to not give underlying physical or logical models, but learn patterns in form of parameters and pre-chosen functional forms. Then we provide enough examples for a particular task for the machine to learn the patterns of the data and memorize

general functional relationships between inputs and outputs for the seen data. For a more complete description of machine learning techniques we refer the interested reader for example to the books of Bishop [9]; MacKay [56]. One limitation of other approaches is, that they commonly only provide point estimates for predictions. That is, they only provide one answer, without giving a sense of the uncertainty of the model. This can lead to difficulties in interpretation, when faced with a counterintuitive result. For more insights see the books above.

In this thesis, we focus on a probabilistic approach, in which probabilistic interpretations of functions are integrated over to give a functional relation between data seen. These are called Bayesian *non-parametric* statistical models [9; 70], in which all possible latent functions are integrated over and the most likely moments a posteriori are reported. This gives the opportunity for results to be interpreted in a coherent way, as the uncertainty of the model can be assessed. This leads to Gaussian processes described in Section 2.1, which this thesis focuses on for machine learning techniques. In Gaussian processes, we require so called *hyper-parameters*, which are optimized using gradient based optimization.

1.3.0.1 Prediction Based Analysis in Gene Expression Experiments

In biology, mechanisms play an important role. As we saw in the genetics Section (1.2), biology has developed very complex strategies during evolution. As evolution builds upon old strategies [21] there is a lot of redundancy associated with that complexity. This complexity is not easily fully described and makes taking all mechanisms into account a very hard task. It is useful to model the real biology by more general, but complex enough machine learning techniques on a predictive bases. Machine learning techniques learn from seeing data, adjusting the (hyper-)parameters to fit a general view. This general view, usually does not include underlying biological mechanisms.

Prediction based analysis is essentially a “black box” based approach. The machine learning designer design the underlying functional relations inside this black box. They then supply intuitional insights into the function of the black box for other researchers (preferably crossing expertise), so that they can apply the algorithm to their problem. It is important to point out, that the design of the internals of the black box can have big impacts on results and should not be disregarded completely. However, researchers from other fields of expertise should be able to apply such complex algorithms to their own data without having to fully know the exact underlying code and technique. You should essentially be able to provide the training inputs and output pairs and let the machine learn the patterns in the data (by fitting parameters). After the learning period, we can ask the black box for

likely outcomes of newly seen inputs.

Most prediction based algorithms try to expose some of the mechanisms (parameters) to the user, so that more insight can be gained into the decision making of the algorithm. This comes in particularly handy when the underlying method assigns importance to newly learned features. This can help to decide which features the algorithm deems important to solve the prediction from inputs to outputs.

In this thesis, we show how to apply software design ideas to implement a Gaussian process in a prediction based way, to be able to directly apply it to gene expression experiments without having to know the exact internals of the underlying technique (Sec. 4.2.1).

In summary, machine learning is to restrain from direct mechanistic modelling of expected patterns and go closer to giving the machine enough resources to generalize over patterns itself and decide how to use the patterns in the inputs to extrapolate on the outputs. This can significantly improve results, as long as the provided mechanisms are general enough to generalize over patterns, while not being too general to overfit the training data, over explaining patterns directly.

1.4 Research Code

In this thesis there is a particular focus on implementation and handling of code produced in research. As described before, we will show details about implementation and reusability in machine learning research. This section will describe an overview for common reoccurring problems and solutions we encountered during the course of this thesis. This is by no means exhaustive and is only meant to give insights and material for thought when implementing a piece of software during the development of new machine learning models. Mainly, we present examples of solutions, which can help in producing reproducible and easy-to-use research code. “There is a culture that reinforces the idea that producing and publishing code has no perceived benefit to the researcher” [14]. We believe quite the opposite: the code published with a transcript (in computational research) is at least as important as, if not more important than the results presented. If results are not reproducible by provided code, it is unlikely that others will use the provided model.

Writing reusable research code is a difficult task. It requires rigorous discipline and planning of progression of the code. When prototyping ideas, a structure in code “pops out”, which can (and most likely will) introduce unforeseen problems. Additionally, many researchers do not have an understanding of the difficulties other researches encounter when trying to replicate results. We often think “if I can do it, so can others”. This is not always true, especially in terms of time spent to

apply and understand parameterizations and intuitions of methods. Time becomes an increasing issue in this result oriented society, where others might not have the time to set up, especially when crossing expertise.

1.4.1 Tutorials

Writing comprehensive tutorials can be difficult, but they are essential for understanding the underlying code. Most of the time it comes down to writing a simple example of the code used in the paper. It is important to point out the parameters and difficulties here, or supply some methodologies to handle these for the inexperienced user (default parameters). Most importantly, a platform for help, discussion and comments should be supplied [18]. Github provides a good platform for this in the issues and commenting section (Sec. 1.4.4). We, as researchers, should consider more carefully how to speed up the process of others using and prototyping methods that we develop. Being able to simply run a method of interest on an own dataset and seeing first results gives confidence at first glance and the will to further investigate. Here again, the discussion platform comes in handy, to further improve upon the simple results. It helps a lot having the ability to just download a package, which is easily applicable, but as complex as necessary. Fellow researchers do not have the time to re-implement complex algorithms. The devil often sits in the details of an implementation, such as gradients, numerical stability and the right handling of parameters.

1.4.2 Codebase & Knowledge of Algorithm

Keeping code clean and simple can be difficult, if only one set of eyes look at the codebase. Thus, it is important to ask ones group members to use the code and apply it to a simple dataset. Make them explain the application, without looking at the source code, just by documentation alone. Some groups might even think of creating their own group package, so group members need to include their code in a meaningful way inside this overarching package. Members need to ensure that newly added code is tested and can be run (Sec. 1.4.5). This can also be done with the community in mind. Make your code accessible right away and do not wait too long for a first version of your codebase [67]. The more people use your code and apply it to their data, the more feedback you can expect and improve practicability and ease-of-use.

When implementing an algorithm it is easy to fall into the trap of trying to solve everything at once. Most of the time the algorithm can be broken down into small interchangeable components. Good code shows itself in clean naming (across the

whole package), proper separation of sub modules and reuse of functionality to increase reliability [95]. One way of ensuring reliability is to test smaller components on their supposed logic (so-called *unit tests*). When tested appropriately, one can rely on these components to build higher level functions. This means we can include new features into the system, without having to rewrite the logic around those components. A highly regarded method of producing a piece of a reliable framework is to first write the (unit) test before the actual implementation. If this is not possible or code already exists, retrofitting tests to test the old codebase should be done before implementing new parts. This ensures the reliability of the old codebase and give confidence in the function of the newly added piece of software. The tests will test the supposed outcome from all expected inputs and therefore ensure the functionality. Usually, discovering the algorithm and writing pieces for it in code comes hand in hand and greatly improves the understanding of the algorithm from a new perspective. Usually the perspective from the programming point of view gives a more practical and intuitive view on the algorithm and helps to explain an algorithm more rigorously.

1.4.3 Making Algorithms Accessible

Algorithms and new ideas are often not accessible to researchers, due to programming language differences, lack of documentation, unclear design, or simply no download option. Sometimes, the codebase of a researcher is only accessible through direct contact and the code is not clean. One very simple solution to this, is to “get code out there”. That is, making the code public to a wider audience and have others consider the code within their own use case. In accordance with the institute of the research, upload the code to an open source storage, where people can download and install the code themselves. This will increase usage of the codebase and people will apply the algorithms involved to their data, increasing the knowledge about the algorithm. This spreads the knowledge of the algorithm and may give others ideas of how to use or improve upon the algorithm, or how to improve the code itself to make it easier to apply [95].

At times it is also preferable to have colleagues, maybe even of different specialties, apply and use the code of interest. This is either during development of the algorithm before publication, or to collaborate on extensions or applications of itself. Research software should also not go too far in simplifying things. The danger is, that the code might become “opinionated code” [63], offering solutions, which are specifically tuned to particular problems. This inhibits extendability and development for different tasks in the codebase.

In the following, we will describe ways of how to spread code, keeping a clean

codebase and how to split algorithms apart to allow further development for others.

1.4.4 Github

Github is a combination of storage and version control [53]. Storing code on an open source base is the best way to improve code and get feedback on applicability and usability. Other interested programmers can participate through cloning the codebase and putting pull requests for suggested changes onto Github. This means we can check the changes they made and approve or comment on issues. There are also tools to automatically check code correctness and coverage of tests (Sec. 1.4.5). On Github, one pushes and pulls from a repository in order to synchronize to the cloud. It is a commit based upload, each of which has a history and its own commit message to record the specific task the commit is to accomplish. These commit messages are an important tool to check history and fixes. Keeping commits small and focused on one particular task is the most straight forward way to keeping the codebase clean. Only if commits do not hold side effects not described in the commit message, we can revert a commit to undo history.

All communication is open and users can communicate enhancements, questions and suggestions easily through the Github issues section. This makes applicability of the codebase even more accessible, as users can ask for help at every step along the process [18].

1.4.5 Automating Code Correctness

There are a lot of interconnected tools to keeping a repository tidy and running besides version control. We need to make sure tests run on all platforms supported by the software and keep running, even when changes to the codebase are being done [95]. `Travis-ci.org` provides a testing suite for package unit tests to be run on different operating systems and under user specified conditions. This makes multi-platform code easier to handle and to keep track of problems on platforms, not easily at hand. Travis (and other so-called *continuous integration* services) install the package and dependencies on their server and run the testing suite provided by the user. Additionally to running tests, they supply the service to run other code based quality control tools, such as `codecov.io`. Codecov checks the coverage of the code, when running the tests. That is each line that has not been hit by the code, when running the tests is marked as missing. Additionally to running tests, a continuous integration service usually provides ways to publish (upload the package to a package provider) packages and make the new “tagged” version available

for everyone through the program language specific channels. A tag in Github is a way to mark versions/releases of a package in the history. They make automated releases of the package accessible, without having to setup the deployment manually for each new version.

Including all those things may sound a lot, but once set up, it is easy to maintain a clean codebase and keep the tests running. This way of sharing code also greatly increases the trust in the codebase and relieves doubts about code not working as intended.

1.4.6 Summary

As an exemplar of the concepts shown in this section, we show the implementation of a complex algorithm in section 3. Section 3 describes the process of how to split algorithms into parts in the GPy framework [27]. Additionally, we show that the extraction of basic tasks enhances the ability of looking at the important parts of an algorithm (Sec. 3.12, 3.9). GPy includes all ideas and concepts presented in this section. Contributions to GPy include continuous integration (including coverage reports), testing, automatic deployment, user interaction, development of methodology and handling of contributions from the community.

Chapter 2

Methods

In this chapter, we will introduce the methodologies which are used to model the biological systems discussed in the thesis. Describing the complexity of biological systems in a mechanistic way, without using machine learning techniques is challenging, if not impossible. That is, because many biological systems are highly redundant and introduce many ways of sustaining function. We use general predictive modelling to model the outcome of the system reacting to certain stimuli. These stimuli can be anything from chemical substrates, genetic changes, or physical alterations of the system. The central limit theorem of stochastic variables states that the sum of independent random variables converge towards a normal distribution. As *Gaussian processes* are multivariate normal distributions (Sec. 2.1), they are attractive models for explaining biological systems. Gaussian processes can be interpreted as stochastic functions with a changing mean and variance along the input. With these functions, we can model the complexity of biological systems in a generalized way. Apparently stochastic small changes may be explained by independent variance. These small variations are assumed to be technical variation or biological processes that are too low in variance to assign to biological meaning. Biological variation, however, is assumed to be changing more slowly, and is modelled by changes in the mean of the model. Section 2.1 describes the base version of Gaussian processes in detail.

With Gaussian processes we are not forced to supply an input to the functions. It may be that the inputs to a function themselves are unknown. We can model an observed (high dimensional) output of a system and find the most likely inputs, as well as the functions mapping from the input to the output. This is known as the *Gaussian process latent variable model* (GPLVM, Sec. 2.3). It is commonly known as dimensionality reduction, where we try to find lower dimensional inputs describing the observed outputs. This is possible as we know biology is redundant and can thusly be explained using more dense spaces, encompassing the complex-

ity in fewer dimensions. Dimensionality reduction is primarily used to make the complex processes visible for humans to comprehend in fewer dimensions, such as plotting the first two most explanatory dimensions, which will show the overall similarity, clustering or trajectory of inputs. Section 2.2 describes principal component analysis, a dimensionality reduction technique directly leading up to GPLVM. GPLVM is described in Section 2.3. In the following sections, we will look at a variant of Gaussian processes which releases the complexity requirements of Gaussian processes, called sparse Gaussian processes and the Bayesian GPLVM (Section 2.5). This makes Gaussian processes applicable on a larger scale.

As the complexity in biological systems comes from combining different systems in redundant ways, we need to deal with confounded systems, in which the effect we are looking for might be shadowed by other processes. In such systems, we need to deal with, and model, the confounding variation and try to focus on the system we are interested in. Literature calls this confounder correction, though we model the confounding variation alongside the effect we are looking for, instead of explaining it away in a multi step analysis. Multi step analyses usually first find confounding variation and report residuals for the next step, or require confounding variables to be known [24; 51; 66]. This can introduce biases, or explain away the variable of interest in the residuals without a chance of recovery. In this work, we try to model both effects jointly [24] and try to find independent dimensions of the different effects, which explain the overall variation in the data together. We employ the *manifold relevance determination* (MRD) method to model genetic subsets jointly. Section 2.6 explains manifold relevance determination.

Probabilistic modelling of biological systems allows us to describe the topology of the lower dimensional representation: the so called *manifold embedding* [87]. We use this information about the topology of the lower dimensional representation to model and adjust distances between samples of data, which can be stretched or condensed by the manifold embedding. Section 4.1 explains the landscape extraction of the manifold of Gaussian processes and how to use that information to correct for distances between samples.

2.1 Gaussian Process Regression

A *Gaussian process* (GP) [70] is a generalization of the Gaussian distribution. It is defined as the distribution over functions f , such that any finite subset $\mathbf{y} = f(\mathbf{x})$ is jointly Gaussian distributed. Loosely speaking, a GP describes the expansion of the Gaussian distribution to a distribution over functions. The Gaussian distribution is a distribution over scalars or vectors, whereas a Gaussian process extends this

finite object to an infinite one.

To make use of this infinite object, we first have to introduce the dataset. In regression, we observe a dataset of $\mathcal{D} = (\mathbf{x}_i, \mathbf{y}_i)_{1 \leq i \leq N}$ of N input output pairs. In this definition the data can be higher dimensional $\mathbf{x}_i \in \mathbb{R}^Q$ and $\mathbf{y}_i \in \mathbb{R}^D$, such that the full dataset can be given in matrix form (\mathbf{X}, \mathbf{Y}) : $\mathbf{X} \in \mathbb{R}^{N \times Q}$, $\mathbf{Y} \in \mathbb{R}^{N \times D}$. The challenge is to find a predictive function f , which can predict the most probable outputs $\mathbf{Y}^* = f(\mathbf{X}^*)$ at any new points \mathbf{X}^* . With GPs, we do not only find a predictive function for the most probable outputs, but a distribution over functions. This means the subset of the Gaussian process which corresponds to the prediction is a joint Gaussian distribution

$$p(\mathbf{Y}^* | \mathbf{X}^*) = \prod_{d=1}^D \mathcal{N}(\mathbf{y}_d^* | \boldsymbol{\mu}_d, \boldsymbol{\Sigma}_d) ,$$

with mean $\boldsymbol{\mu}_d$ and covariance $\boldsymbol{\Sigma}_d$. Thus, after having observed the dataset, we will always be talking about finite subsets of the infinite GP in terms of Gaussian distributions.

We need to assign expected properties to the functions f , as otherwise we are bound to overfit the dataset and a generalization is not possible. These properties are given through the so called *covariance function* $k(\mathbf{X}, \mathbf{X}')$. The covariance function defines the covariance between inputs \mathbf{X} and \mathbf{X}' . A requirement for any covariance function is that it generates a positive definite covariance matrix $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}'_j)$ for the prior belief of the GP (Sec. 2.1.2). With this, we can define any subset \mathbf{F} for a GP prior as

$$p(\mathbf{F} | \mathbf{X}) = \prod_{d=1}^D \mathcal{N}(\mathbf{f}_d | \mathbf{0}, \mathbf{K}).$$

Notice, that we assume independence between output dimensions D of the function values \mathbf{F} of the GP prior.

To learn a dataset \mathcal{D} , we constrain the GP to the observed data. Loosely speaking, we draw from samples from the prior at the inputs and discard all functions which generate outputs that do not conform to the observed data. In mathematical terms, we integrate over the function space, integrating over all possible functions generated by the GP prior. To consider the data, we define the likelihood of the data as a factorized Gaussian distribution of noisy observations \mathbf{Y} of the function values \mathbf{F} of the GP as

$$p(\mathbf{Y} | \mathbf{X}, \mathbf{F}) = \mathcal{N}(\mathbf{Y} | \mathbf{F}, \sigma^2 \mathbf{I})$$

This leads to the marginal likelihood of the GP, marginalizing out all possible val-

ues \mathbf{F} of the GP functions

$$p(\mathbf{Y}|\mathbf{X}, \boldsymbol{\theta}) = \int p(\mathbf{Y}|\mathbf{X}, \mathbf{F}, \boldsymbol{\theta}_\ell)p(\mathbf{F}|\mathbf{X}, \boldsymbol{\theta}_k) \, d\mathbf{F} \, ,$$

where $\boldsymbol{\theta} = \boldsymbol{\theta}_\ell \cup \boldsymbol{\theta}_k$ are possible hyper-parameterizations of the likelihood and covariance function: $\boldsymbol{\theta}_\ell$ are the hyper-parameters for the likelihood ℓ and $\boldsymbol{\theta}_k$ are the hyper-parameters for the covariance function k . In the following, we will omit the hyper-parameters from equations¹ to unclutter notation.

We can solve the integral for the marginal likelihood analytically by “completing the square” [70]

$$\begin{aligned} p(\mathbf{Y}|\mathbf{X}) &= \prod_{i=d}^D \int \mathcal{N}(\mathbf{Y}_{\cdot d}|\mathbf{F}_{\cdot d}, \sigma^2)\mathcal{N}(\mathbf{F}_{\cdot d}|\mathbf{0}, \mathbf{K}) \, d\mathbf{F}_{\cdot d} \\ &= \mathcal{N}(\mathbf{Y}|\mathbf{0}, \mathbf{K} + \sigma^2\mathbf{I}) \, . \end{aligned} \tag{2.1}$$

See Equation (A.1) for a detailed calculation of this marginal likelihood. We can use Bayes’ theorem to turn around the marginal likelihood to compute the density for the latent function values $p(\mathbf{F}|\mathbf{Y}, \mathbf{X})$

$$\begin{aligned} p(\mathbf{F}|\mathbf{Y}, \mathbf{X}) &= \frac{p(\mathbf{Y}|\mathbf{F})p(\mathbf{F}|\mathbf{X})}{p(\mathbf{Y}|\mathbf{X})} \\ &= \mathcal{N}(\mathbf{F} | (\mathbf{K}^{-1} + \beta\mathbf{I})^{-1}\beta\mathbf{I}\mathbf{Y}, (\mathbf{K}^{-1} + \beta\mathbf{I})^{-1}) \, , \end{aligned}$$

where we define $\beta = \sigma^{-2}$. See Equation (A.2) for a detailed calculation of the conditional distribution of the latent GP function values \mathbf{F} .

In the following we will have a closer look at the gradient computations for Gaussian processes, leading up to Chapter 3, in which we will describe how to make use of them in a real world implementation.

2.1.1 Gradients

In mathematical terms, the inference in the GP is integrating over the latent function \mathbf{f} as illustrated in equation (2.1). We can see, that it could be numerically problematic to optimize the marginal likelihood, as it could be small and it only ranges positive values. Computers can only perform numerical computations, and are limited by the precision of the system. The precision of a computer system is bound by the number of bits of information being able to be stored in one number. This number is finite for computers.

The most prominent problem is if there is multiple dimensions $D > 1$, as it then is a product of potentially small values. In order to relieve this problem, we

¹We let $p(\mathbf{Y}|\mathbf{X}) \equiv p(\mathbf{Y}|\mathbf{X}, \boldsymbol{\theta})$.

employ the natural logarithm (\log). The logarithm is a monotonic transformation, this means, all optima in the original space, will be optima in log space as well. The logarithm stretches the space from 0 to 1 into the negative values. Additionally, what used to be a product of values, becomes a sum in log-space. We sum up positive and negative values, instead of multiplying.

Additionally, the logarithm reduces complexity in implementation of gradient based optimization in two ways. First, the products in the original space become independent parts in the sum. We can compute the partial gradients for each part and sum the results together to get the overall gradient. Second, the exponential of the Gaussian distribution gets cancelled out and simplifies the gradient, as the exponential would remain for gradient computations.

The maximization is done over the hyper-parameters θ , maximizing the log marginal likelihood $\mathcal{L} := \log p(\mathbf{Y}|\mathbf{X}, \theta, \sigma^2)$:

$$\begin{aligned} \mathcal{L} &= \log \left(\left(D ((2\pi)^N |\mathbf{K} + \sigma^2 \mathbf{I}|)^{-\frac{1}{2}} \right) \exp \left\{ -\frac{1}{2} \text{tr} \mathbf{Y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y} \right\} \right) \\ &= -\frac{ND}{2} \log 2\pi - \frac{D}{2} \log |\mathbf{K} + \sigma^2 \mathbf{I}| - \text{tr} \mathbf{Y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y} \\ \hat{\theta} &= \arg \max_{\theta} \mathcal{L} . \end{aligned} \quad (2.2)$$

With that, we can compute the gradients of the GP in parts, observing the chain rule through the parts. Whenever the parameters show up, they show up in the term

$$(\mathbf{K} + \sigma^2 \mathbf{I}) := \mathbf{K}_{GP} , \quad (2.3)$$

which we will call the *GP covariance*. We will push the gradients through the GP covariance to compute the gradients in steps. The gradients for each of the parts (likelihood and prior) split, as they appear as a sum in \mathbf{K}_{GP} ². The GP covariance is treated as element-wise evaluation of the covariance function and thus is treated as a collection of scalars. This means, for each parameter $\theta \in \boldsymbol{\theta}$ we simply sum up all the gradients going through the GP covariance.

$$\frac{\partial \mathcal{L}}{\partial \theta_t} = \sum_{i=1}^N \sum_{j=1}^N \frac{\partial \mathcal{L}}{\partial [\mathbf{K}_{GP}]_{ij}} \frac{\partial [\mathbf{K}_{GP}]_{ij}}{\partial \theta_t} . \quad (2.4)$$

There is only scalars in this chain rule computation, and thus we can handle all gradient computations from here on as scalar gradients. We can write the gradients

²Keep in mind, that if there is multiple parameters, we compute the gradient for each parameter, respectively, so there is no need for tensor based derivations.

of the log marginal likelihood with respect to the GP covariance $\mathbf{K}_{\mathcal{GP}}$ as [64]:

$$\begin{aligned}
\frac{\partial \log p(\mathbf{Y}|\mathbf{X})}{\partial \mathbf{K}_{\mathcal{GP}}} &= \frac{\partial}{\partial \mathbf{K}_{\mathcal{GP}}} \left(-\frac{ND}{2} \log(2\pi) - \frac{D}{2} \log |\mathbf{K} + \sigma^2 \mathbf{I}| - \frac{1}{2} \text{tr}(\mathbf{Y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y}) \right) \\
&= -\frac{D}{2} \frac{\partial}{\partial \mathbf{K}_{\mathcal{GP}}} \log |\mathbf{K} + \sigma^2 \mathbf{I}| - \frac{1}{2} \frac{\partial}{\partial \mathbf{K}_{\mathcal{GP}}} \text{tr} \mathbf{Y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y} \quad (2.5) \\
&= -\frac{D}{2} \text{tr} (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} + \frac{1}{2} \text{tr} (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y} \mathbf{Y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \\
&= -\frac{D}{2} \text{tr} \mathbf{K}_{\mathcal{GP}}^{-1} + \frac{1}{2} \text{tr} \mathbf{K}_{\mathcal{GP}}^{-1} \mathbf{Y} \mathbf{Y}^\top \mathbf{K}_{\mathcal{GP}}^{-1} .
\end{aligned}$$

As discussed in the element-wise computation for the gradient, we can drop the trace from here and push it one level up into the element-wise sum from eq. (2.4).

The gradients for the GP covariance with respect to its hyper-parameters will be handled separately for each covariance function, as well as the likelihood.

2.1.2 Gaussian Process Prior (Covariance Function)

A Gaussian process prior consists of a mean and its covariance function. The mean is often assumed as zero, as empirically we can always zero mean the observed data. The covariance function $k(\mathbf{x}, \mathbf{x})$ is used to model the properties of functions described by the Gaussian process. It builds the covariance matrix $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, where \mathbf{x}_i is the i -th row of the input matrix $\mathbf{X} \in \mathbb{R}^{N \times Q}$, of N samples and Q dimensions. The covariance function of a GP determines the “shape” of the functions the GP can take.

Differentiability of the covariance function at zero for example, determines the “smoothness” of functions generated. The higher the order of differentiability of a covariance function, the smoother the generative function of the GP gets. See some examples of functional shapes in Table 2.1.

Stationary covariance functions are independent of shifts to the input locations, and thus behave the same, “invariant to translation in the input space” [70]. Non-stationary covariance functions are dependent of shifts to the input location and behave differently, depending on the location of the inputs.

If covariance functions generate a process, the variance of which collapses at one point, it is called *degenerate*. Conversely, covariance function that do not collapse are called *non-degenerate*.

Again, we have sampled and plotted some covariance functions alongside their respective expressions in Table 2.1 for more intuition. Here, we will discuss only a few covariance functions in detail, which will be used in this theses. A more complete picture for covariance functions can be found in Rasmussen and Williams [70, Chapter 4].

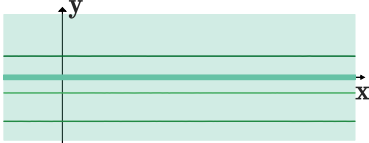
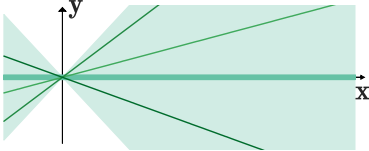
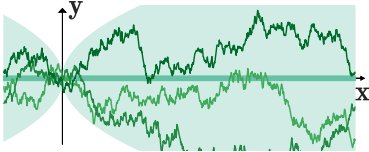
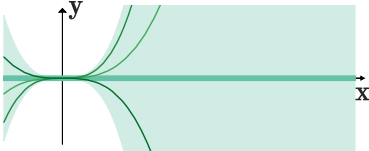
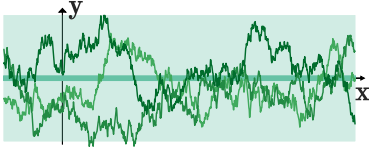
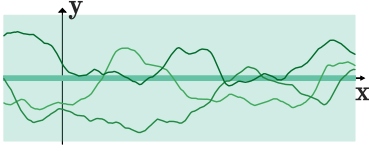
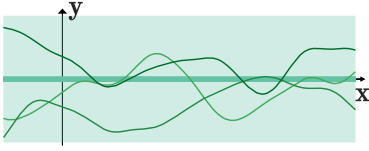
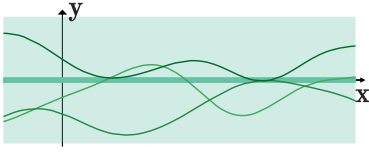
Function	Expression	Stationary Non-Degenerate	Figure
Bias	α	✓✓	
Linear	$\mathbf{x}_i \mathbf{A} \mathbf{x}_j^\top$		
Brownian	$\alpha \min(\mathbf{x}_i, \mathbf{x}_j)$		
Polynomial	$\alpha(\sigma_f^2 + \mathbf{x}_j \cdot \mathbf{x}_j^\top)^d$		
Exponential	$\alpha \exp(-\frac{r}{\ell})$	✓✓	
Matérn	$\frac{\alpha 2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}r}{\ell}\right)^\nu K_\nu\left(\frac{\sqrt{2\nu}r}{\ell}\right)$	✓✓	
Rational Quad.	$\alpha\left(1 + \frac{r^2}{2d\ell^2}\right)^{-d}$	✓✓	
Exp. Quad.	$\alpha \exp\left(-\frac{r^2}{2\ell^2}\right)$	✓✓	

Table 2.1: Some covariance functions with a plot for intuition. All expressions assume either $k(\mathbf{x}_i, \mathbf{x}_j)$ or $k(r)$, where $r = |\mathbf{x}_i - \mathbf{x}_j|$ as the function definition. $\mathbf{A} = \text{diag}(\alpha)$, α, ℓ, ν are positive parameters and K_ν is a modified Bessel function [1]. Mean (thick line) and 95% confidence intervals (shaded area) for different covariance functions. Three samples are plotted alongside the posterior to understand the intuition of the “quality of generative functions” that a GP produces, depending on its covariance function. (Quad.: Quadratic, Exp.: Exponentiated)

2.1.2.1 Linear

The linear covariance function is a non-stationary covariance function leading to a model of all linear functions:

$$k(\mathbf{X}, \mathbf{X}') = \mathbf{X} \mathbf{A} \mathbf{X}'^\top, \quad (2.6)$$

where \mathbf{A} is a diagonal matrix, and the diagonal $\text{diag } \mathbf{A} = \alpha$ contains one parameter α_q per dimension of the input. The smaller the parameter α_q the less relevant this dimension is for the GP prior. Here, relevant means that the function appears constant for the input range with a sufficiently small variance α . This behavior is called automatic relevance determination and is discussed in more detail in Section 2.1.5.

Gradients The only parameters for the linear covariance function are α_q . The gradients can be written as follows:

$$\begin{aligned} \frac{\partial \mathbf{K}_{ij}}{\partial \alpha_q} &= \frac{\partial}{\partial \alpha_q} \sum_{k=1}^Q \mathbf{X}_{ik} \alpha_k \mathbf{X}'_{jk} \\ &= \sum_{k=1}^Q \mathbf{X}_{ik} \frac{\partial \alpha_k}{\partial \alpha_q} \mathbf{X}'_{jk} \\ &= \mathbf{X}_{iq} \mathbf{X}'_{jq} \end{aligned}$$

For latent variable models in later sections (Secs. 2.3, 2.5), we will need the gradients of the covariance function w.r.t. \mathbf{X} , which can be written as:

$$\begin{aligned} \frac{\partial \mathbf{K}_{ij}}{\partial \mathbf{X}_{lq}} &= \frac{\partial}{\partial \mathbf{X}_{lq}} \sum_{k=1}^Q \mathbf{X}_{ik} \alpha_k \mathbf{X}_{jk} \\ &= \sum_{k=1}^Q \frac{\partial \mathbf{X}_{ik}}{\partial \mathbf{X}_{lq}} \alpha_k \mathbf{X}'_{jk} \\ &= \alpha_q \mathbf{X}'_{jq} . \end{aligned}$$

The inner partial gradient $\frac{\partial \mathbf{X}_{ik}}{\partial \mathbf{X}_{lq}}$ is only 1 if $k = q, i = l$.

Note: If $\mathbf{X} = \mathbf{X}'$, this gradient has to be multiplied by 2, as the summands will be squared inside, as then $j = l$.

2.1.2.2 Exponentiated Quadratic Covariance function

Also known as Radial Basis Function (RBF) or Gaussian covariance function (it has the form of a non normalized Gaussian distribution) the exponentiated quadratic

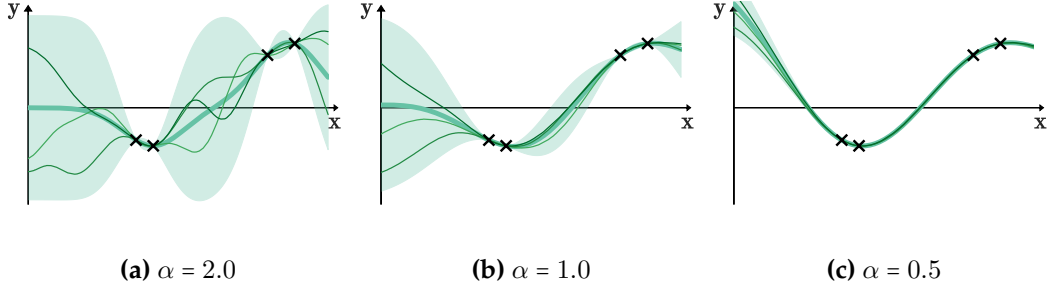


Figure 2.1: Three different inverse lengthscales for a GP with an exponentiated quadratic covariance function. Mean is indicated by thick line, 95% confidence interval is shaded and three samples are plotted alongside. The samples are restricted to the observed data, shown as black crosses.

covariance function is defined as

$$k(\mathbf{X}_i, \mathbf{X}'_j) = \sigma_f^2 \exp \left\{ -\frac{1}{2} \sum_{q=1}^Q \alpha_q (\mathbf{X}_{iq} - \mathbf{X}'_{jq})^2 \right\} . \quad (2.7)$$

This covariance function produces non linear, infinitely differentiable functions. In other words, smooth functions with any shape, where the “wiggleness” is defined by the inverse lengthscales α . See three different inverse-lengthscales GPs in Figure 2.1 for more intuition. As before, for detailed explanations please refer to Rasmussen and Williams [70, Section 4].

Gradients Here we want the gradients of the covariance matrix \mathbf{K} w.r.t. all parameters σ_f^2 and α .

$$\frac{\partial \mathbf{K}_{ij}}{\partial \sigma_f^2} = \exp \left\{ -\frac{1}{2} \sum_{q=1}^Q \alpha_q (\mathbf{X}_{iq} - \mathbf{X}'_{jq})^2 \right\} ,$$

and

$$\begin{aligned} \frac{\partial \mathbf{K}_{ij}}{\partial \alpha_k} &= \frac{\partial}{\partial \alpha_k} \sigma_f^2 \exp \left\{ -\frac{1}{2} \sum_{q=1}^Q \alpha_q (\mathbf{X}_{iq} - \mathbf{X}'_{jq})^2 \right\} \\ &= \mathbf{K}_{ij} \frac{\partial}{\partial \alpha_k} \left\{ -\frac{1}{2} \sum_{q=1}^Q \alpha_q (\mathbf{X}_{iq} - \mathbf{X}'_{jq})^2 \right\} \\ &= -\frac{1}{2} \mathbf{K}_{ij} (\mathbf{X}_{ik} - \mathbf{X}'_{jk})^2 . \end{aligned}$$

For latent variable models in later sections (Secs. 2.3, 2.5), we will also need the gradients w.r.t. the inputs \mathbf{X}

$$\begin{aligned}\frac{\partial \mathbf{K}_{ij}}{\partial \mathbf{X}_{ik}} &= \mathbf{K}_{ij} \frac{\partial}{\partial \mathbf{X}_{ik}} \left\{ -\frac{1}{2} \sum_{q=1}^Q \alpha_q (\mathbf{X}_{iq} - \mathbf{X}_{jq})^2 \right\} \\ &= -\alpha_k \mathbf{K}_{ij} (\mathbf{X}_{ik} - \mathbf{X}_{jk}) .\end{aligned}$$

2.1.3 Prediction

To make predictions, we need to calculate the probability density $p(\mathbf{F}^* | \mathbf{X}^*, \mathbf{Y}, \mathbf{X})$ of the GP function values at newly seen input locations \mathbf{X}^* . The function values $\mathbf{F}^* = f(\mathbf{X}^*)$ are the function values of the GP evaluated at the positions \mathbf{X}^* . We use the optimized hyper-parameters to describe the joint distribution over \mathbf{Y} and \mathbf{F}^* . This will be a Gaussian distribution as described in Section 2.1 and can be written as:

$$p\left(\begin{bmatrix} \mathbf{Y} \\ \mathbf{F}^* \end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix} \mathbf{Y} \\ \mathbf{F}^* \end{bmatrix} \middle| \mathbf{0}, \begin{bmatrix} \mathbf{K}_{\mathbf{F}\mathbf{F}} + \sigma^2 \mathbf{I} & \mathbf{K}_{\mathbf{F}\mathbf{F}^*} \\ \mathbf{K}_{\mathbf{F}^*\mathbf{F}} & \mathbf{K}_{\mathbf{F}^*\mathbf{F}^*} \end{bmatrix}\right) .$$

The covariance matrix describing the covariance between \mathbf{Y} and \mathbf{F}^* makes use of the marginalization property of GPs. Using the definition of GPs and the marginalization property in conjunction, we can directly read off the prediction density

$$p(\mathbf{F}^* | \mathbf{X}^*, \mathbf{X}, \mathbf{Y}) = \mathcal{N}(\mathbf{F}^* | \mathbf{M}, \mathbf{\Sigma}) \quad (2.8)$$

with mean prediction

$$\mathbf{M} = \mathbf{K}_{\mathbf{F}^*\mathbf{F}} (\mathbf{K}_{\mathbf{F}\mathbf{F}} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y}$$

and covariance of the prediction

$$\mathbf{\Sigma} = \mathbf{K}_{\mathbf{F}^*\mathbf{F}^*} - \mathbf{K}_{\mathbf{F}^*\mathbf{F}} (\mathbf{K}_{\mathbf{F}\mathbf{F}} + \sigma^2 \mathbf{I})^{-1} \mathbf{K}_{\mathbf{F}\mathbf{F}^*} .$$

See Petersen and Pedersen [64]; Rasmussen and Williams [70] for more details. Usually we are only interested in the variance of the prediction at the input points, which are the diagonal terms of $\mathbf{\Sigma}$.

2.1.4 Example and Sample

In Figure 2.2 some samples of a GP are drawn and plotted. From left to right, we add more and more observations to the GP, constraining possible outcomes of the samples drawn. You can see the mean as thick line and the 95% confidence interval as shaded area. Alongside the mean and variance of the posterior, we draw

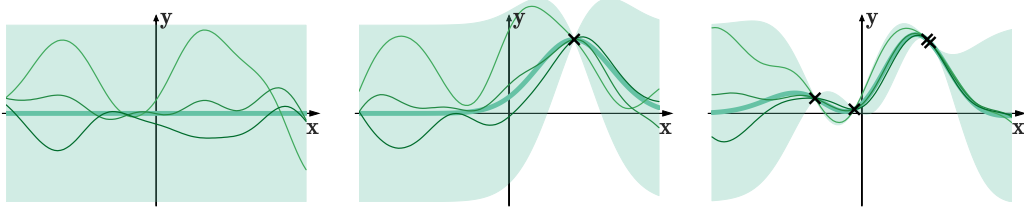


Figure 2.2: Plotted are mean (thick line) and 95% confidence interval (shaded area) of a GP. Thin lines are samples drawn from the posterior. From left to right, adding more and more observations restricting the posterior to assume observed values. Observe the uncertainty going down to (almost) zero, when we observe a value and all samples go through it. Here we assume a likelihood variance of $\sigma^2 = 1 \times 10^{-6}$, so all variation comes from the underlying latent functions f .

3 different samples of the GP in thin lines.

Sampling from a GP We want a sample \mathbf{y} to have a covariance of $\text{cov}(\mathbf{y}, \mathbf{y}) = \mathbf{K}$. Multiplying the Cholesky factorization $\mathbf{L}\mathbf{L}^\top = \mathbf{K}$ with a random draw $\mathbf{r} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ of a uniform Gaussian distribution achieves exactly this result. Let $\mathbf{y} = \mathbf{r}\mathbf{L}$, then

$$\begin{aligned} \text{cov}(\mathbf{r}\mathbf{L}, \mathbf{r}\mathbf{L}) &= \mathbf{L}\text{cov}(\mathbf{r}, \mathbf{r})\mathbf{L}^\top \\ &= \mathbf{L}(\mathbf{I})\mathbf{L}^\top \\ &= \mathbf{K} \end{aligned}$$

where we made use of expected value properties described in [64, Section 8.2]. Multiplying the lower Cholesky decomposition \mathbf{L} of a covariance matrix \mathbf{K} of any covariance function k by a random draw $\mathbf{r} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ of a univariate Gaussian distribution will give a sample of a GP.

2.1.5 ARD: Automatic Relevance Determination

Gaussian processes are multivariate distributions over observed data \mathbf{Y} employing the covariance matrix \mathbf{K} , where $\mathbf{K}_{ij} = k(\mathbf{X}_i, \mathbf{X}_j)$ for each pair of rows of \mathbf{X} . The form of the covariance function determines the nature of the generative process of the GP [70]. In this thesis, one goal is to select input features (columns of the latent input matrix \mathbf{X}) that correspond to pattern changes in the output data when learning a lower dimensional representation. This can be achieved using the so-called *automatic relevance determination* (ARD) kernels. Some kernels assign a scaling parameter per dimension, identifying the scaling of each dimension. This means, if the scaling is low, the output function appears constant over a change in the input space range. We call this notion “switching off” dimensions in the latent (input)

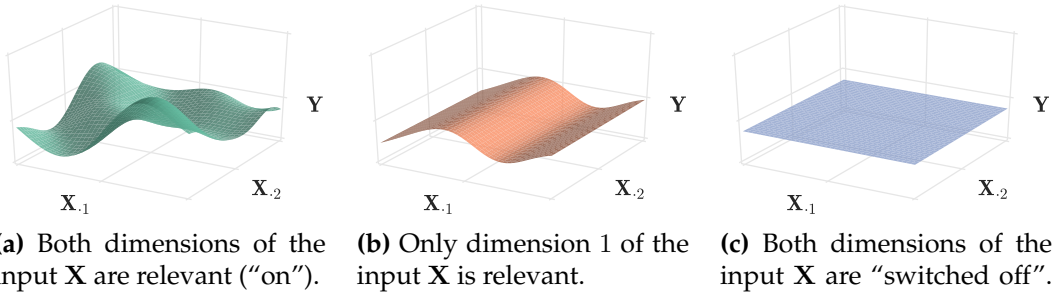


Figure 2.3: Three examples of a Gaussian process with ARD exponentiated quadratic covariance functions to show the effect of the scaling parameter on the input dimension. The scaling parameters were chosen as 1 for “on” and 1×10^{-10} for “off”.

space.

In this section we will look at the ARD exponentiated quadratic covariance function (Eq. (2.7))

$$k(\mathbf{X}_i, \mathbf{X}_j) = \sigma_f^2 \exp \left\{ -\frac{1}{2} \sum_{q=1}^Q \alpha_q (\mathbf{X}_{iq} - \mathbf{X}_{jq})^2 \right\} .$$

The scaling parameters are $\alpha \in \mathbb{R}^D$, which create the relevance score for each dimension q of the inputs $\mathbf{X} \in \mathbb{R}^{N \times Q}$. A large α_q identifies that the dimension \mathbf{X}_q is relevant to the covariance. Comparatively, with decreasing α_q the relevance of dimension q goes down and the generative function gets flat (can be linear) in the generative function space.

Intuitively, we can think of the scaling as the frequency of the functions crossing the origin (moving up and down) within a range of the input \mathbf{x} . To gain more insight, we plot three two dimensional examples of a GP with different scaling parameters, to show the “switching off” effect in Figure 2.3. In the plot, we see three two dimensional samples of a GP with differing ARD parameters. It shows that the ARD parameter α_q for dimension q regulates the relevance for each dimension. With it, the model can decide whether a dimension is relevant by choosing the appropriate parameter. In a GP setting, each parameterisation of a covariance matrix introduces an implicit penalisation through the interaction between the normalization term $-\frac{D}{2} \log |\mathbf{K} + \sigma^2 \mathbf{I}|$ and the data fit term $-\text{tr} \mathbf{Y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y}$ of the log marginal likelihood (Eq. (2.1)). Intuitively speaking, the more ARD parameters are high valued, the higher the rank the GP covariance will be and thus, the higher the normalization term will be. On the other hand the connectivity of the precision (inverse GP covariance) needs to be high enough, to minimize the empirical covariance of the data as seen through the data term. We depict this interaction in a

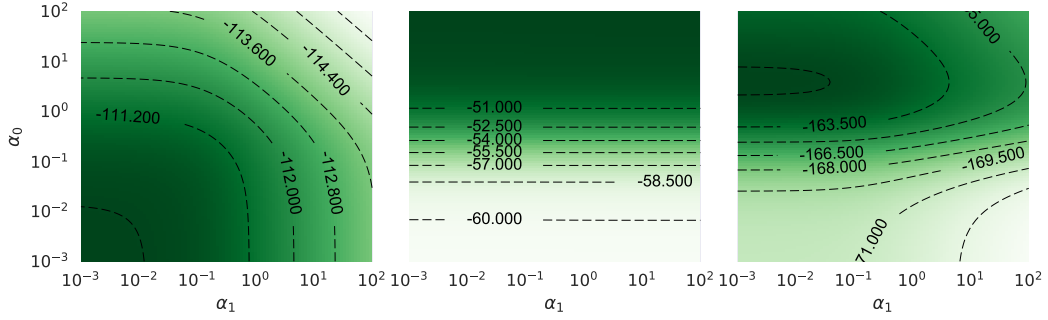


Figure 2.4: Simulation of the interaction between the normalization term $-\frac{D}{2} \log |\mathbf{K} + \sigma^2 \mathbf{I}|$ and the data fit term $-\text{tr} \mathbf{Y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y}$ of the log marginal likelihood when altering the ARD parameters α of a two dimensional linear kernel. The left hand plot shows the normalization term, the middle the data fit term and the right the sum of the two. The plots shown were generated from a simple two dimensional input set \mathbf{X} of which only the first dimension is used to create the output \mathbf{Y} .

simple simulation in Figure 2.4. This implicitly drives the model parameters to be chosen as complex as necessary and as simple as possible (see Occam’s razor e.g. Rasmussen and Ghahramani [69]). The linear covariance function (2.6)

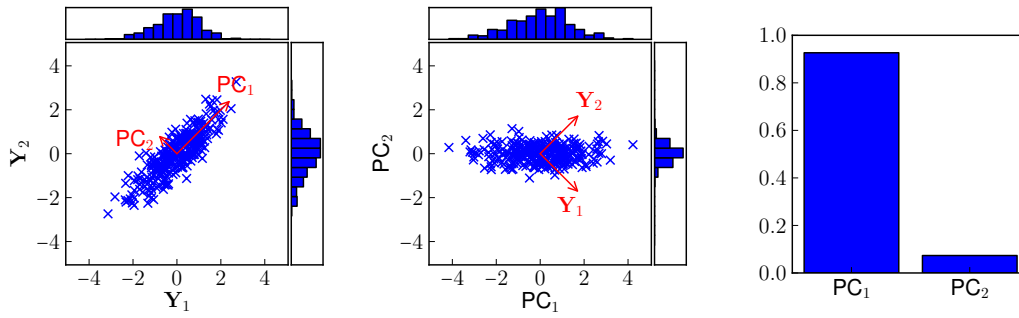
$$k(\mathbf{X}_i, \mathbf{X}_j) = \sum_{q=1}^Q \alpha_q X_{iq} X_{jq} ,$$

with ARD parameters α has also the property of ARD, similar to the above. The ARD parameters here describe the variance of the slope for the linear embedding of the generative function of the GP. This means, the smaller the ARD parameter in a linear covariance function model, the less the slope can vary and is close to the prior, which is defined as zero. In summary, each covariance function has its own interpretation of ARD parameters and have to be taken with care, when interpreting results.

In this thesis, we mostly rely on the exponentiated quadratic ARD covariance function. This means, the ARD parameters tell us about the amount of non-linearity in the respective dimension. Any linear relationships in the latent space are explained by smaller ARD parameters and act like a residual regression of PCA before doing non-linear analysis.

2.2 Principal Component Analysis

One main tool for analysis in this thesis is dimensionality reduction. We make use of the Gaussian process latent variable model described in Section 2.3 to identify patterns in high dimensional data. The GPLVM is a generalization of the well



(a) Scatter plot in original feature space. Red arrows show principal component directions. The histograms show the marginal distribution of features. (b) Principal component space of original features. Red arrows indicate original data space directions. Marginals are shown for each PC. (c) Fractions of explained variance by principal components.

Figure 2.5: Principal component analysis on example feature set \mathbf{Y} . The rows of \mathbf{Y} represent the samples and the columns are the two features measured. (a) shows a scatter plot of the features in the original data space. In principal component space (b) original space has been rotated, so that the variance of the PCs are ordered by their magnitude. (c) shows that the first principal component explains most of the variance of the data, and the second is almost negligible. We make use of that, by only using the first principal component as a lower dimensional approximation of the data. [98, p.13]

known method, Principal component analysis (PCA, [41]), which is why we will describe PCA briefly here.

PCA is a way of reducing the dimensionality of a dataset by basis rotation. The goal is to find a combination of a rotation and sorting of dimensions of the observed data \mathbf{Y} , so that the resulting lower dimensional view \mathbf{X} has the highest variance in the first dimension, second highest in the second dimension and so on. We then choose to ignore dimensions of the rotated view \mathbf{X} at a cutoff value, which has to be chosen. PCA is not really a dimensionality reduction technique itself, but by ignoring dimensions of low variance, we effectively reduce the dimensionality.

Suppose we have an observed dataset $\mathbf{Y} \in \mathbb{R}^{N \times D}$ of $N = 300$ samples over $D = 2$ features correlated as depicted in Figure 2.5a ([98, p.13]). PCA finds the new orthogonal basis to maximize the variance explained in order. For this observed dataset, the basis is shown in red, annotated as $\text{PC}_{\{1,2\}}$. This basis corresponds to the direction of the eigenvectors $\mathbf{V} \in \mathbb{R}^{D \times D} = (\mathbf{v}_d)_{1 \leq d \leq D}$ of the empirical covariance $\mathbf{Y}^\top \mathbf{Y}$ of the phenotype. Then, the fraction of variance explained is the fraction of eigenvalues of this covariance. We keep both matrices sorted descending according to the eigenvalue fractions. As we can clearly see, the second principal component does not explain much variance and can be cut off to reduce dimensionality. We can use PCA to find Q dimensional latent factors in the phenotype. As already

mentioned, the direction of highest variance of the phenotype corresponds to the direction of the eigenvector with the highest eigenvalue. To transform the observed dataset \mathbf{Y} into the full principal component space $\mathbf{X}^{\text{PCA}} = (\text{PC}_d)_{1 \leq d \leq D}$, we have to multiply it by the eigenvectors $\mathbf{V} \in \mathbb{R}^{D \times D}$ of the covariance matrix $\mathbf{Y}^\top \mathbf{Y}$

$$\mathbf{X}^{\text{PCA}} = \mathbf{Y}\mathbf{V} .$$

Here, we can see the connection to the generative model, which can be recovered by right-multiplying the inverse of the eigenvector matrix \mathbf{V} to the PCA solution

$$\mathbf{Y} = \mathbf{X}\mathbf{V}^{-1} .$$

Figure 2.5b ([98, p.13]) shows PC_1 against PC_2 , which corresponds to the columns of \mathbf{X}^{PCA} . To reduce the dimensionality to a value $Q < D$, we take only the first Q eigenvectors \mathbf{V}_Q , instead of the full eigenvector matrix. This corresponds to the Q highest eigenvalues of the covariance and we get a lower dimensional representation

$$\mathbf{X} = \mathbf{Y}\mathbf{V}_Q .$$

In this toy example, we can easily see how many dimensions to use for the downstream analysis (Fig. 2.5c [98, p.13]). Namely, the first principal component, as it describes the majority of variance for the given data. Choosing a lower dimensionality, however, can become a hard choice when dealing with millions of features and requires expert knowledge and repeated experiments. Also, some heuristics exist to find the number of dimensions. One way, is to plot the eigenvalue fractions in order and see if there are “kinks” in the connected lines between eigenvalue fractions. It is then the norm to keep only the dimensions directly before the most prominent kink. Another way is to only keep a chosen percentage of variance explained (say keep 95% of variance explained).

One way of tackling the problem of finding the right dimensionality is to build a Bayesian probabilistic model. The Bayesian probabilistic view on functions is to consider all possible functions and weighting by their probability of explaining data observed. Or in other words, we sample from the prior and reject all samples not in accordance with the observed data. By modelling the PCA in a Bayesian probabilistic setting

$$p(\mathbf{Y}) = \int p(\mathbf{Y}|\mathbf{X})p(\mathbf{X}) d\mathbf{X} , \quad (2.9)$$

we can determine the number of dimensions by making use of the probabilistic nature of the model and applying the ARD (Sec. 2.1.5) parameters to select relevant dimensions.

There have been many approaches on probabilistic modelling of gene expression data [16; 24; 37; 78; 96] and my master thesis [98]. In this thesis, we focus on the Gaussian process latent variable model [47] to analyse high dimensional biological data (i.e. gene expression data). In the next section, we will have a closer look at the Gaussian process latent variable model and some extensions to further facilitate dimensionality reduction and propagation of uncertainty.

2.3 Gaussian Process Latent Variable Model (GPLVM)

In this section, we will have a detailed look at the core dimensionality reduction technique of this thesis, the Gaussian process latent variable model (GPLVM, [98]) developed by Lawrence [47, 48].

In dimensionality reduction, we want to assign a lower dimensional input space $\mathbf{X} \in \mathbb{R}^{N \times Q}$ and latent function $f(\mathbf{X})$ generating the higher dimensional observed variables $f(\mathbf{X}) = \mathbf{Y}$. We will start with the linear mapping $\mathbf{Y} = \mathbf{X}\mathbf{V}$ with weights $\mathbf{V} \in \mathbb{R}^{Q \times D}$, weighting the individual influences of latent features and release that assumption later. As opposed to the method used in this thesis, there is another Bayesian probabilistic principal component analysis [84], which puts a prior on the latent inputs \mathbf{X} and integrates them out. This means, they learn the weights for a given \mathbf{X} to create the data observed and find an optimal \mathbf{X} for given weights in an alternating manner, see Tipping and Bishop [84] for a detailed description of probabilistic PCA. In the method of this thesis, we will integrate over the weights using a GP prior to eliminate the need for a parametric solution entirely.

In GPLVM - the method used in this thesis - we put the prior on the mapping from \mathbf{X} to \mathbf{Y} (which in the linear case corresponds to the weights \mathbf{V}), and integrate over the inputs. We learn the most probable input \mathbf{X} for a given output \mathbf{Y} explained by the generative model $\mathbf{Y} = \mathbf{X}\mathbf{V}$. (This turns out to be equivalent to principal component analysis solution when considering the linear mapping in GPLVM [8; 83], the GPLVM allows for a probabilistic non-linear extension of PCA).

We assume independence across dimensions of the weights and thus, the prior for the weights take the form

$$p(\mathbf{V}) = \prod_{d=1}^D \mathcal{N}(\mathbf{V}_{\cdot d} | \mathbf{0}, \mathbf{I}_Q) , \quad (2.10)$$

where $\mathbf{V}_{\cdot d}$ is the d th column (i.e. weighting dimension d) of \mathbf{V} . By integrating over \mathbf{V} the likelihood for the phenotype \mathbf{Y} , given an input \mathbf{X} can be written as (Eq. (2.1))

$$p(\mathbf{Y}|\mathbf{X}) = \prod_{d=1}^D p(\mathbf{Y}_{\cdot d}|\mathbf{X}) , \quad (2.11)$$

where

$$p(\mathbf{Y}_{\cdot d}|\mathbf{X}) = \mathcal{N}(\mathbf{Y}_{\cdot d}|\mathbf{0}, \mathbf{K} + \beta^{-1}\mathbf{I}_N) . \quad (2.12)$$

Here, \mathbf{K} is the covariance matrix created by the covariance function $k(\mathbf{X}_{\cdot d}, \mathbf{X}'_i)$, evaluated at every sample (i.e. row) \mathbf{X}_i and \mathbf{X}'_i of the inputs \mathbf{X} and $\beta = \sigma^{-2}$. In the linear case this corresponds to

$$\mathbf{K} = \mathbf{X}\mathbf{A}\mathbf{X}^\top , \quad (2.13)$$

where $\mathbf{A} \in \mathbb{R}^{Q \times Q}$ is a diagonal matrix with parameters $\boldsymbol{\theta} = \text{diag}(\mathbf{A})$. We can now estimate the distribution of the data (2.9) by maximum a posteriori (MAP) estimate of both the parameters of the model and the latent inputs jointly. This estimate can be found by maximising the log marginal likelihood of the model given the parameters and \mathbf{X} (Eq. (2.2))

$$\ln p(\mathbf{Y}|\mathbf{X}) = -\frac{DN}{2} \ln(2\pi) - \frac{D}{2} \ln |\mathbf{K}_{\mathcal{GP}}| - \frac{1}{2} \text{tr}(\mathbf{K}_{\mathcal{GP}}^{-1} \mathbf{Y}\mathbf{Y}^\top) , \quad (2.14)$$

where $\mathbf{K}_{\mathcal{GP}} = \mathbf{K} + \beta^{-1}\mathbf{I}_N$. The set of parameters

$$\{\hat{\mathbf{X}}, \hat{\boldsymbol{\theta}}, \hat{\beta}\}_{\text{MAP}} = \arg \max_{\{\mathbf{X}, \boldsymbol{\theta}, \beta\}} \ln p(\mathbf{Y}|\mathbf{X}, \boldsymbol{\theta}, \beta) , \quad (2.15)$$

which maximizes the log-likelihood (2.14) contains the (a posteriori) most probable inputs for the given phenotype \mathbf{Y} . Here, we intentionally added the parameters $\boldsymbol{\theta}$ of the covariance function and the precision parameter β of each Gaussian process (Eq. 2.12). This set also contains the most probable set of inputs \mathbf{X} , generating the phenotype under the generative model $\mathbf{Y} = \mathbf{X}\mathbf{V}$. This model is generated by the covariance function one chooses to use (which we assumed to be linear).

To find the most probable set of parameters (2.15) we apply gradient based optimisation, because in most cases there is no fixed point solution to maximize (2.14). However, in the linear case, the fixed point solution can be written as the well known PCA (Section 2.2). The gradient of (2.14) w. r. t. \mathbf{X} can be written as

$$\frac{\partial \ln p(\mathbf{Y}|\mathbf{X})}{\partial \mathbf{X}} = \mathbf{K}_{\mathcal{GP}}^{-1} \mathbf{Y}\mathbf{Y}^\top \mathbf{K}_{\mathcal{GP}}^{-1} \mathbf{X} - D\mathbf{K}_{\mathcal{GP}}^{-1} \mathbf{X} ,$$

where α is the noise parameter for the prior over \mathbf{V} (2.10). Finding the fixed point solution and solving for \mathbf{X} leads to

$$\frac{1}{D} \mathbf{Y}\mathbf{Y}^\top \mathbf{K}_{\mathcal{GP}}^{-1} \mathbf{X} = \mathbf{X} . \quad (2.16)$$

With some algebraic manipulation this solution can further be simplified (see de-

tails in [48]) to

$$\mathbf{X} = \mathbf{U}_Q \mathbf{L} \mathbf{\Sigma}^\top, \quad (2.17)$$

where \mathbf{U}_Q is a $N \times Q$ matrix, whose columns are eigenvectors of $\mathbf{Y}\mathbf{Y}^\top$, \mathbf{L} is a diagonal $Q \times Q$ matrix, whose diagonal entries $\ell = (l_q)_{1 \leq q \leq Q}$ are $l_q = \left(\frac{\lambda_q}{\alpha D} - \frac{1}{\beta \alpha} \right)^{-\frac{1}{2}}$, while λ_q is the q th eigenvalue of $\mathbf{Y}\mathbf{Y}^\top$, and $\mathbf{\Sigma}$ is a $Q \times Q$ orthogonal matrix. This is the PCA (Section 2.2) solution to the problem [48]. In Section 2.2, we used the eigenvectors \mathbf{V}_Q of the covariance matrix $\mathbf{Y}^\top \mathbf{Y}$. These can be transformed to the eigenvectors \mathbf{U} of the matrix $\mathbf{Y}\mathbf{Y}^\top$. (see e.g. Bishop [9]; Tipping [83]). This transformation and a corresponding normalisation leads to the solution above.

2.3.1 Inferring Subspace Dimensionality

PCA can be seen as a special case of the GPLVM model, assuming a linear mapping f between input and output. Now let us relax the linearity assumption in the GPLVM case. We can see that Eq. (2.11) is a product of D independent Gaussian processes with linear covariance function $k(\mathbf{X}, \mathbf{X}')$. It is natural to extend this model to non-linear mappings between latent inputs \mathbf{X} and measured outputs \mathbf{Y} by introducing any non-linear covariance function k , for example the ARD exponentiated quadratic (Sec. 2.1.5). With that, we are able to find non-linear input spaces, generating observed data. As described in the ARD section (Sec. 2.1.5), we can use the ARD parameters of the linear or exponentiated quadratic kernel to identify dimensions of interest. Thus, we only have to supply enough dimensions initially for the model to fit the data. The GPLVM can tell us how many dimensions of the initial ones it needed to find the optimal solution. This means we have reduced the problem of how many dimensions to pick to a problem of supplying enough dimensions initially. The number of dimensions of the latent space is not expensive in terms of runtime requirements, but the gradients need to be computed, so it is a tradeoff to be aware of, when running GPLVM.

Leading up to the next section, we still have a problem with applying this model to high dimensional and big data, as we have to compute the inverse \mathbf{K}^{-1} of the $N \times N$ covariance matrix \mathbf{K} . In the next section, we will see how to introduce $M \ll N$ inducing inputs \mathbf{Z} for \mathbf{K} , such that we can approximate \mathbf{K} with a rank M form. As we will discover, the underlying Gaussian process prior still is able to approximate the true non parametric posterior of the likelihood, and the complexity of the GP computation can be brought down from $\mathcal{O}(N^3)$ to $\mathcal{O}(NM^2)$.

2.4 Sparse Gaussian Process Regression

Gaussian processes are flexible tools for non-parametric regression. One disadvantage is the computational cost of $\mathcal{O}(N^3)$, where N is the number of samples to regress over. The cost comes from the inversion of the covariance matrix $\mathbf{K}_{\mathcal{GP}}^{-1}$ necessary to compute the log-marginal likelihood of the GP (Eq. 2.1). One proposed solution to this cost, is to approximate the covariance matrix $\mathbf{K}_{\mathcal{GP}}$ of the GP. In the following, we will separate the noise variance hyper parameter σ^2 from the GP covariance $\mathbf{K}_{\mathcal{GP}} = \mathbf{K} + \sigma^2\mathbf{I}$ and handle the likelihood hyper parameter separately from the covariance matrix.

The sparse GP approximation is done by introducing inducing inputs $\mathbf{Z} \in \mathbb{R}^{M \times Q}$ (surrogate variables, that live in the same space as the inputs \mathbf{X}), where $M \ll N$. They are used to compute a secondary covariance of lower rank than the original covariance of the GP

$$\begin{aligned} \mathbf{K} &\approx \mathbf{K}_{\mathbf{FU}} \mathbf{K}_{\mathbf{UU}}^{-1} \mathbf{K}_{\mathbf{UF}} \text{ ,} \\ [\mathbf{K}_{\mathbf{FU}}]_{ij} &= k(\mathbf{X}_i, \mathbf{Z}_j) \qquad \qquad \qquad [\mathbf{K}_{\mathbf{UU}}]_{ij} = k(\mathbf{Z}_i, \mathbf{Z}_j) \text{ .} \end{aligned}$$

To introduce the inducing inputs, we follow the approach of Titsias [85]. In Section 2.1, we have observed that the distribution over latent function values \mathbf{F} for the inputs \mathbf{X} can be written as $p(\mathbf{F}|\mathbf{X})$ (Eq. (2.1)). Parallel to this, we observe the latent values of the GP at the inducing inputs \mathbf{Z} to be the inducing outputs \mathbf{U} , with their prior distribution defined over the same covariance evaluated at the inducing inputs $\mathbf{K}_{\mathbf{UU}}$:

$$p(\mathbf{U}|\mathbf{Z}) = \mathcal{N}(\mathbf{U}|\mathbf{0}, \mathbf{K}_{\mathbf{UU}}) \text{ .}$$

The cross covariance between inducing inputs \mathbf{Z} and inputs \mathbf{X} is $\mathbf{K}_{\mathbf{FU}}$.

This gives us the opportunity to upper bound the original marginal likelihood of the GP, which only requires us to compute the inverse of the secondary covariance. Using Sparse GPs, we can speed up computation from $\mathcal{O}(N^3)$ to $\mathcal{O}(NM^2)$, as the cross covariance from inputs to inducing inputs has to be computed.

Following along the definition of GPs, we can observe the conditional distribution over the latent functions \mathbf{F} given the inducing outputs

$$\begin{aligned} p(\mathbf{F}|\mathbf{U}) &= \mathcal{N}(\mathbf{F}|\mathbf{K}_{\mathbf{FU}} \mathbf{K}_{\mathbf{UU}}^{-1} \mathbf{U}, \mathbf{\Lambda}) \\ \mathbf{\Lambda} &= \mathbf{K}_{\mathbf{FF}} - \mathbf{K}_{\mathbf{FU}} \mathbf{K}_{\mathbf{UU}}^{-1} \mathbf{K}_{\mathbf{UF}} \text{ .} \end{aligned}$$

Let $\beta = \sigma^{-2}$ in the following for uncluttered notation. With the above we can observe, that the log marginal likelihood of the GP can be written including the

inducing inputs

$$\log p(\mathbf{Y}|\mathbf{X}, \mathbf{Z}) = \int \int p(\mathbf{Y}|\mathbf{F})p(\mathbf{F}|\mathbf{U}, \mathbf{X}, \mathbf{Z})p(\mathbf{U}|\mathbf{Z}) \, d\mathbf{U} \, d\mathbf{F} .$$

Following Titsias [85], we perform variational integration of \mathbf{U} by introducing Jensen's bound on the log marginal likelihood. We will first integrate out the latent function \mathbf{F} as

$$\begin{aligned} \log p(\mathbf{Y}|\mathbf{U}, \mathbf{X}, \mathbf{Z}) &= \log \langle p(\mathbf{Y}|\mathbf{F}) \rangle_{p(\mathbf{F}|\mathbf{U}, \mathbf{X}, \mathbf{Z})} \\ &\geq \langle \log p(\mathbf{Y}|\mathbf{F}) \rangle_{p(\mathbf{F}|\mathbf{U}, \mathbf{X}, \mathbf{Z})} && \text{Jensen's bound} \\ &= D \log c - \frac{1}{2} \text{tr} \langle (\mathbf{Y} - \mathbf{F})\beta\mathbf{I}(\mathbf{Y} - \mathbf{F}) \rangle_{p(\mathbf{F}|\mathbf{U}, \mathbf{X}, \mathbf{Z})} && \text{Matr.cookbook p.42} \\ &= D \log c - \frac{1}{2} \text{tr} \left((\mathbf{Y} - \mathbf{K}_{\mathbf{F}\mathbf{U}}\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1}\mathbf{U})^\top \beta\mathbf{I}(\mathbf{Y} - \mathbf{K}_{\mathbf{F}\mathbf{U}}\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1}\mathbf{U}) + \beta\mathbf{\Lambda} \right) \\ &= \log \mathcal{N}(\mathbf{Y}|\mathbf{M}_{\mathcal{L}_1}, \beta^{-1}\mathbf{I}) - \frac{1}{2}\beta \text{tr} \mathbf{\Lambda} \\ &=: \mathcal{L}_1 , \text{ with} \\ \mathbf{M}_{\mathcal{L}_1} &:= \mathbf{K}_{\mathbf{F}\mathbf{U}}\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1}\mathbf{U} . \end{aligned}$$

Here, $c = (2\pi)^{-\frac{N}{2}}|\sigma^2\mathbf{I}|^{-\frac{1}{2}}$ is the normalizer for $p(\mathbf{Y}|\mathbf{F})$. This gives us the log marginal likelihood in terms of the inducing outputs \mathbf{U} . It comes to show, that the distribution $p(\mathbf{Y}|\mathbf{X})$ can be fully explained by the inducing inputs, if we were to actually apply the integral over \mathbf{F} . This would not lead to additional insight, as it would just be a switch of variables from \mathbf{F} to \mathbf{U} (and respective inputs \mathbf{X} to \mathbf{Z}). Hence, we perform the variational approximation using Jensen's bound.

From here, we can now integrate out the latent GP function values \mathbf{U} by taking the expectation under the inducing output distribution:

$$\begin{aligned} \log p(\mathbf{Y}|\mathbf{X}, \mathbf{Z}) &= \log \langle p(\mathbf{Y}|\mathbf{U}, \mathbf{X}, \mathbf{Z}) \rangle_{p(\mathbf{U})} \\ &\geq \log \langle \exp \mathcal{L}_1 \rangle_{p(\mathbf{U})} \\ &= \log \left(\exp \left\{ -\frac{1}{2}\beta \text{tr} \mathbf{\Lambda} \right\} \int \mathcal{N}(\mathbf{Y}|\mathbf{M}_{\mathcal{L}_1}, \beta^{-1}\mathbf{I})\mathcal{N}(\mathbf{U}|\mathbf{0}, \mathbf{K}_{\mathbf{U}\mathbf{U}}) \, d\mathbf{U} \right) \\ &= D \log c - \frac{1}{2} \text{tr} \left(\mathbf{Y}^\top (\beta\mathbf{I} - \mathbf{K}_{\mathbf{F}\mathbf{U}}\beta\mathbf{I}(\mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta\mathbf{K}_{\mathbf{U}\mathbf{F}}\mathbf{K}_{\mathbf{F}\mathbf{U}})^{-1}\beta\mathbf{I}\mathbf{K}_{\mathbf{U}\mathbf{F}})\mathbf{Y} \right) - \frac{1}{2}\beta \text{tr} \mathbf{\Lambda} \\ &= \log \mathcal{N}(\mathbf{Y}|\mathbf{0}, \mathbf{K}_{\mathbf{F}\mathbf{U}}\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1}\mathbf{K}_{\mathbf{U}\mathbf{F}} + \beta^{-1}\mathbf{I}) - \frac{1}{2}\beta \text{tr} \mathbf{\Lambda} \\ &=: \mathcal{L}_2(\mathbf{Y}) , \end{aligned} \tag{2.18}$$

with the normalizer factor $c = (2\pi)^{-\frac{N}{2}}|\mathbf{K}_{\mathbf{F}\mathbf{U}}\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1}\mathbf{K}_{\mathbf{U}\mathbf{F}} + \beta^{-1}\mathbf{I}|^{-\frac{1}{2}}$.

2.4.1 Optimization and Complexity

Compared to the original maximization of the log marginal likelihood of the GP (Eq. (2.2)), we now have to optimize the inducing input locations in addition to the hyper parameters of the covariance and likelihood:

$$\{\hat{\mathbf{Z}}, \hat{\boldsymbol{\theta}}\} = \arg \max_{\mathbf{Z}, \boldsymbol{\theta}} \mathcal{L}_2 \text{ .}$$

Thus, we can optimize the sparse GP bound \mathcal{L}_2 with respect to the positions of the inducing inputs $\mathbf{Z} \in \mathbb{R}^{M \times Q}$ in $\mathcal{O}(NM^2)$. This computation is dominated by the computation of $\mathbf{K}_{\text{FU}} \mathbf{K}_{\text{UU}}^{-1} \mathbf{K}_{\text{UF}}$, where the inversion of the inducing inputs covariance matrix $\mathbf{K}_{\text{UU}}^{-1}$ can be computed in $\mathcal{O}(M^3)$. This is dominated by the product between $\mathbf{K}_{\text{FU}} \mathbf{K}_{\text{UU}}^{-1}$. This results in the overall complexity of this variant of the sparse GP algorithm to be $\mathcal{O}(NM^2)$.

2.4.2 Implementation

The implementation of sparse GPs can be seen analogous to standard GPs. The inducing inputs \mathbf{Z} are variational parameters and are optimized jointly with the prior parameters $\boldsymbol{\theta}$. The gradients of the inducing inputs can be gotten by using the chain rule and computing the gradients of K w.r.t. \mathbf{X} (or more precisely \mathbf{Z}).

$$\frac{\partial \mathcal{L}_2}{\partial \mathbf{Z}} = \frac{\partial \mathcal{L}_2}{\partial \mathbf{K}_{\text{UU}}} \frac{\partial \mathbf{K}_{\text{UU}}}{\partial \mathbf{Z}} + \frac{\partial \mathcal{L}_2}{\partial \mathbf{K}_{\text{UF}}} \frac{\partial \mathbf{K}_{\text{UF}}}{\partial \mathbf{Z}} \text{ .}$$

If we want to handle uncertain inputs, all equations can be explained using the Bayesian GPLVM expressions. Please see expressions used in 2.5 for further details.

2.4.3 Prediction

To perform (approximate) predictions for new values \mathbf{Y}^* at new (given) input locations \mathbf{X}^* , we need to compute the conditional distribution over \mathbf{Y}^* given the new inputs \mathbf{X}^* and seen data

$$p(\mathbf{Y}^* | \mathbf{X}^*, \mathbf{X}, \mathbf{Y}) = \int p(\mathbf{Y}^* | \mathbf{U}, \mathbf{X}^*) p(\mathbf{U} | \mathbf{Y}, \mathbf{X}, \mathbf{Z}) \text{ d}\mathbf{U} \text{ .}$$

We first begin to find the conditional distribution of the inducing inputs when seen the data using Bayes' rule

$$p(\mathbf{U} | \mathbf{Y}, \mathbf{X}, \mathbf{Z}) = \frac{p(\mathbf{Y} | \mathbf{U}, \mathbf{X}, \mathbf{Z}) p(\mathbf{U} | \mathbf{Z})}{\langle p(\mathbf{Y} | \mathbf{U}, \mathbf{X}, \mathbf{Z}) \rangle_{p(\mathbf{U} | \mathbf{Z})}} \text{ .}$$

As we are using variational approximations for the computations of the bounds, we use the approximated probability to compute the posterior prediction as follows:

$$\begin{aligned}
\tilde{p}(\mathbf{U}|\mathbf{Y}, \mathbf{X}, \mathbf{Z}) &= \frac{\exp(\mathcal{L}_1)p(\mathbf{U}|\mathbf{Z})}{\langle \exp(\mathcal{L}_1) \rangle_{p(\mathbf{U}|\mathbf{Z})}} \\
&= \frac{\mathcal{N}(\mathbf{Y}|\mathbf{M}_{\mathcal{L}_1}, \beta^{-1}\mathbf{I})\mathcal{N}(\mathbf{U}|\mathbf{0}, \mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1}) \exp(\frac{1}{2}\beta \text{tr } \mathbf{\Lambda})}{\int \mathcal{N}(\mathbf{Y}|\mathbf{M}_{\mathcal{L}_1}, \beta^{-1}\mathbf{I})\mathcal{N}(\mathbf{U}|\mathbf{0}, \mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1}) d\mathbf{U} \exp(\frac{1}{2}\beta \text{tr } \mathbf{\Lambda})} \\
&= \mathcal{N}(\mathbf{U}|\mathbf{M}_{\mathbf{U}}, \mathbf{S}_{\mathbf{U}}) , \text{ where} \\
\mathbf{S}_{\mathbf{U}} &= (\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1}\mathbf{K}_{\mathbf{U}\mathbf{F}}\beta\mathbf{I}\mathbf{K}_{\mathbf{F}\mathbf{U}}\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1} + \mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1})^{-1} \\
&= \mathbf{K}_{\mathbf{U}\mathbf{U}}\mathbf{\Sigma}^{-1}\mathbf{K}_{\mathbf{U}\mathbf{U}} , \text{ where } \mathbf{\Sigma} = \mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta\mathbf{K}_{\mathbf{U}\mathbf{F}}\mathbf{K}_{\mathbf{F}\mathbf{U}} \\
\mathbf{M}_{\mathbf{U}} &= \mathbf{S}^{-1}\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1}\mathbf{K}_{\mathbf{U}\mathbf{F}}\beta\mathbf{I}\mathbf{Y} \\
&= \beta\mathbf{K}_{\mathbf{U}\mathbf{U}}\mathbf{\Sigma}^{-1}\mathbf{K}_{\mathbf{U}\mathbf{F}}\mathbf{Y} .
\end{aligned}$$

See A.3 for detailed derivation of the above. With that, we can now make predictions using

$$\begin{aligned}
p(\mathbf{Y}^*|\mathbf{X}^*, \mathbf{X}, \mathbf{Y}) &= \int p(\mathbf{Y}^*|\mathbf{U}, \mathbf{X}^*)p(\mathbf{U}|\mathbf{Y}, \mathbf{X}, \mathbf{Z}) d\mathbf{U} \\
&= \mathcal{N}(\mathbf{Y}^*|\beta\mathbf{K}_{\mathbf{F}^*\mathbf{U}}\mathbf{\Sigma}^{-1}\mathbf{K}_{\mathbf{U}\mathbf{F}}\mathbf{Y}, \beta^{-1}\mathbf{I} + \mathbf{K}_{\mathbf{F}^*\mathbf{U}}\mathbf{\Sigma}^{-1}\mathbf{K}_{\mathbf{U}\mathbf{F}^*}) , \\
\mathbf{\Sigma} &= \mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta\mathbf{K}_{\mathbf{U}\mathbf{F}}\mathbf{K}_{\mathbf{F}\mathbf{U}} .
\end{aligned}$$

Note the posterior prediction is non-parametric and therefore will not collapse if there are no inducing inputs. Only when fitting the model, the inducing inputs are the necessary statistics for the bound. As the computation of the bound, the computation of the prediction only includes the inversion of $\mathbf{K}_{\mathbf{U}\mathbf{U}}$ and benefits from the speedup itself.

2.4.4 Intuition

As the prediction of a sparse GP is non parametric, samples of the prior behave the same as in a full GP. The variance outside the defined input range expands to the prior variance and the mean goes back to the prior mean. The difference is when fitting to data (\mathbf{X}, \mathbf{Y}) (and that it is an approximate covariance structure $\mathbf{K}_{\mathbf{F}\mathbf{F}} \approx \mathbf{K}_{\mathbf{F}\mathbf{U}}\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1}\mathbf{K}_{\mathbf{U}\mathbf{F}} + \beta^{-1}\mathbf{I}$). Only the inducing outputs $\mathbf{U}|\mathbf{Z}$ will act as anchor points for the GP fit, replacing the original latent function values \mathbf{F} . The data can only be seen through the induced statistics by \mathbf{U} , and will have to be learnt to closely resemble the inputs most influential areas. To see this in a more intuitive way, we have plotted three different states of a sparse GP fit with differing states of inducing input locations in Figure 2.6.

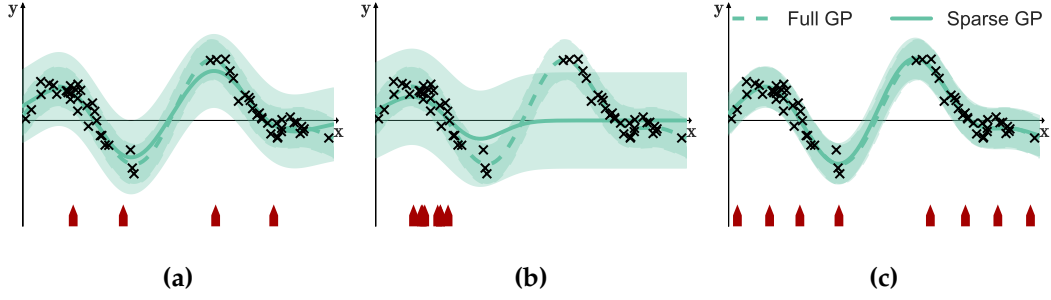


Figure 2.6: Sparse GP plots. The inducing input locations \mathbf{Z} (red arrows at the bottom) differ to show the effect of their position on the underlying GP. The full GP fit is shown as dashed lines, whereas the sparse GP is shown as solid line. **(a):** Too few inducing inputs, so they are not able to fully explain a full GP fit. **(b):** There are enough inducing inputs, but in sub optimal positions. **(c):** Shows an optimal fit of inducing inputs, showing there is almost no difference to a full GP.

2.5 Variational Bayesian GPLVM

In the next step, we want to integrate out the inputs \mathbf{X} , to provide a latent variable model, where the latent space \mathbf{X} is learnt as the solution to the mapping $f(\mathbf{X}) = \mathbf{Y}$ [19; 86]. $f(\mathbf{X})$ has a GP prior $p(\mathbf{X})$, which we approximate by a variational Gaussian prior $q(\mathbf{X}_d) = \mathcal{N}(\mathbf{M}_d, \mathbf{S}_d \mathbf{I}) =: \mathcal{N}(\mathbf{M}, \mathbf{S})$, where \mathbf{S} contains the diagonals \mathbf{S}_d for each dimension as columns, such that the corresponding variance for the mean dimension \mathbf{M}_d is $\mathbf{S}_d \cdot \mathbf{I}$. This implies that there is no cross covariance between latent dimensions, the latent points Gaussian variance is aligned to their respective dimensions. In the following, we will omit the inducing inputs \mathbf{Z} as much as possible, as it will clutter expressions.

We want to integrate over the latent space \mathbf{X} and from the standard GP bound (Eq. (2.1)). This is intractable and we need an approximation. Here, we make use of another variational approximation on top of the sparse GP approximation to have the speedup from the sparse GP bound as well as be able to learn the latent space \mathbf{X} :

$$\begin{aligned}
 \log p(\mathbf{Y}) &= \log \int p(\mathbf{Y}|\mathbf{X})p(\mathbf{X}) \, d\mathbf{X} \\
 &= \log \int p(\mathbf{Y}|\mathbf{X})p(\mathbf{X}) \frac{q(\mathbf{X})}{q(\mathbf{X})} \, d\mathbf{X} \\
 &= \log \left\langle \frac{p(\mathbf{Y}|\mathbf{X})p(\mathbf{X})}{q(\mathbf{X})} \right\rangle_{q(\mathbf{X})} \\
 &\geq \left\langle \log p(\mathbf{Y}|\mathbf{X}) - \log \frac{q(\mathbf{X})}{p(\mathbf{X})} \right\rangle_{q(\mathbf{X})} \\
 &\geq \langle \mathcal{L}_2(\mathbf{Y}) \rangle_{q(\mathbf{X})} - \text{KL}(q(\mathbf{X})||p(\mathbf{X})) \\
 &=: \mathcal{L}_3(\mathbf{Y}) .
 \end{aligned} \tag{2.19}$$

When maximizing the above bound \mathcal{L}_3 , we will minimize the (KL) divergence between p and q , while maximizing the sparse GP bound, which will try to find a suitable fit for the latent function \mathbf{F} to fit the seen data \mathbf{Y} , at the same time as finding the right input locations \mathbf{X} to provide the most likely latent space under the current function \mathbf{F} . This can be confusing in the beginning, we will see the different contributions for the latent space later on.

Let us focus on the left side of the expression. We will use (2.18) to extract the variational bound for Bayesian treatment of the latent space \mathbf{X} . Let, in the following, $\langle \cdot \rangle = \langle \cdot \rangle_{q(\mathbf{X})}$

$$\begin{aligned}
\langle \mathcal{L}_2 \rangle &= \left(D \log c - \frac{1}{2} \text{tr} \mathbf{Y}^\top (\beta \mathbf{I} - \mathbf{K}_{\mathbf{F}\mathbf{U}} \beta \mathbf{I} (\mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta \mathbf{K}_{\mathbf{U}\mathbf{F}} \mathbf{K}_{\mathbf{F}\mathbf{U}})^{-1} \beta \mathbf{I} \mathbf{K}_{\mathbf{U}\mathbf{F}}) \mathbf{Y} - \frac{1}{2} \beta \text{tr} \mathbf{\Lambda} \right) \\
&= -\frac{ND}{2} \log 2\pi + \frac{ND}{2} \log \beta + \frac{D}{2} \log |\mathbf{K}_{\mathbf{U}\mathbf{U}}| + \frac{1}{2} \log |\mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta \langle \mathbf{K}_{\mathbf{U}\mathbf{F}} \mathbf{K}_{\mathbf{F}\mathbf{U}} \rangle| \\
&\quad - \frac{1}{2} \text{tr} \mathbf{Y}^\top (\beta \mathbf{I} - \beta^2 \langle \mathbf{K}_{\mathbf{F}\mathbf{U}} \rangle (\mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta \langle \mathbf{K}_{\mathbf{U}\mathbf{F}} \mathbf{K}_{\mathbf{F}\mathbf{U}} \rangle)^{-1} \langle \mathbf{K}_{\mathbf{U}\mathbf{F}} \rangle) \mathbf{Y} \\
&\quad - \frac{\beta}{2} \langle \text{tr}(\mathbf{K}_{\mathbf{F}\mathbf{F}}) \rangle + \frac{\beta}{2} \text{tr}(\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1} \langle \mathbf{K}_{\mathbf{U}\mathbf{F}} \mathbf{K}_{\mathbf{F}\mathbf{U}} \rangle) \\
&= -\frac{ND}{2} \log 2\pi + \frac{ND}{2} \log \beta + \frac{D}{2} \log |\mathbf{K}_{\mathbf{U}\mathbf{U}}| + \frac{1}{2} \log |\mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta \mathbf{\Psi}_2| \\
&\quad - \frac{1}{2} \text{tr} \mathbf{Y}^\top (\beta \mathbf{I} - \beta^2 \mathbf{\Psi}_1 (\mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta \mathbf{\Psi}_2)^{-1} \mathbf{\Psi}_1^\top) \mathbf{Y} \\
&\quad - \frac{\beta}{2} \psi_0 + \frac{\beta}{2} \text{tr}(\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1} \mathbf{\Psi}_2) ,
\end{aligned}$$

with the normalizing factor $c = (2\pi)^{-\frac{N}{2}} |\mathbf{K}_{\mathbf{F}\mathbf{U}} \mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1} \mathbf{K}_{\mathbf{U}\mathbf{F}} + \beta^{-1} \mathbf{I}|^{-\frac{1}{2}}$. Here, we defined the ψ statistics, which have to be computed depending on the covariance function used:

$$\begin{aligned}
[\psi_0]_i &= \langle k(\mathbf{x}_i, \mathbf{x}_i) \rangle_{q(\mathbf{X})} && \in \mathbb{R}^N \\
[\mathbf{\Psi}_1]_{ij} &= \langle k(\mathbf{x}_i, \mathbf{z}_j) \rangle_{q(\mathbf{X})} && \in \mathbb{R}^{N \times M} \\
[\mathbf{\Psi}_2]_{ijk} &= \langle k(\mathbf{z}_j, \mathbf{x}_i,) k(\mathbf{x}_i, \mathbf{z}_k) \rangle_{q(\mathbf{X})} && \in \mathbb{R}^{N \times M \times M} .
\end{aligned}$$

With this, the bound (2.19) for the variational Bayesian GPLVM (called Bayesian GPLVM in the following) is fully defined.

2.5.1 On ARD Parameterization in Bayesian GPLVM

The ARD parameters of the covariance function, as described in Section 2.1.5 are used for feature selection in the input space, identifying relevant dimensions. Usually, feature selection is driven by penalisation (see e.g. lasso [55]) of the parameters α . Here, we do not have explicit penalisation as we have a Bayesian model. We introduce implicit penalisation by applying Bayesian treatment of the latent space \mathbf{X} , so that the latent space can be uncertain and therefore the ARD parameter can get very small. This is called Bayesian model selection [70].

2.5.2 Implementation

To implement this bound we need to implement the different gradients. We will use the concept of the chain rule in order to make the gradients generally applicable. For any kernel hyper parameter α the following holds for the Bayesian GPLVM bound:

$$\frac{\partial \mathcal{L}_3}{\partial \alpha} = \frac{\partial \mathcal{L}_3}{\partial \mathbf{K}_{\mathbf{U}\mathbf{U}}} \frac{\partial \mathbf{K}_{\mathbf{U}\mathbf{U}}}{\partial \alpha} + \frac{\partial \mathcal{L}_3}{\partial \Psi_0} \frac{\partial \Psi_0}{\partial \alpha} + \frac{\partial \mathcal{L}_3}{\partial \Psi_1^\top \mathbf{Y}} \frac{\partial \Psi_1^\top \mathbf{Y}}{\partial \alpha} + \frac{\partial \mathcal{L}_3}{\partial \Psi_2} \frac{\partial \Psi_2}{\partial \alpha} .$$

Thus, we need to find the partial derivatives of the bound w.r.t. $\mathbf{K}_{\mathbf{U}\mathbf{U}}$ and the Ψ statistics and then the partial derivatives of the kernel of $\mathbf{K}_{\mathbf{U}\mathbf{U}}$ and the Ψ statistics w.r.t. to specific parameters α .

This helps immensely to reduce complexity in the implementation, as the implementation of the kernel only has to implement their respective partial derivatives, and the model implementation can implement their respective partials for the Ψ statistics in general. Note, that we do also need the gradients for the inducing inputs \mathbf{Z} and variational parameters (\mathbf{M}, \mathbf{S}) for the variational distribution over the latent space $q(\mathbf{X}) = \mathcal{N}(\mathbf{M}, \mathbf{S})$. Remember, that \mathbf{S} contains the diagonals for each dimension as columns, such that the corresponding variance for $\mathbf{M}_{\cdot i}$ is $\mathbf{S}_{\cdot i} \mathbf{I}$, which simplifies computations.

For β the above is not useful, as β is the only hyper parameter not depending on the prior covariance function. We need to derive the direct derivative of the bound \mathcal{L}_3 w.r.t. β .

With that, we are able to implement the Bayesian GPLVM algorithm fully and the latent space \mathbf{X} for observed measurements \mathbf{Y} can be learnt in a complexity of $\mathcal{O}(NM^2)$. The Bayesian GPLVM algorithm was implemented and optimized during the course of this thesis and previous work [98].

2.5.3 Factorization and Parallelization

To speed up the computation of Bayesian GPLVM even further (on top of the sparse GP approximation), we can employ parallel computation of tasks in modern multi core computational units. In this section, we will have a look at how the Bayesian GPLVM bound can be factorized in a simple view. The factorization of the Bound across dimensions D has been implemented during the course of this thesis and is available through GPy (Sec. 3).

We can then take advantage of that and distribute the calculations of partial sums on different cores and collect the sums in a master node. The bound described in this section has useful factorization properties in both, N and D . Here, we will

describe in short how these factorizations come to be and can be used to compute in parallel to speed up computation.

2.5.3.1 Factorization in Dimensions D

To factorize the Bayesian GPLVM bound, we need to find a way of writing the bound as a sum across divisible parts of the data. In this section we look at the factorization across dimensions, so that each dimension can be handled individually by individual cores. When looking at the Bayesian GPLVM bound

$$\begin{aligned}\mathcal{L}_3 &= \langle \mathcal{L}_2(\mathbf{Y}) \rangle_{q(\mathbf{X})} - \text{KL}(q(\mathbf{X})||p(\mathbf{X})) \\ \langle \mathcal{L}_2(\mathbf{Y}) \rangle_{q(\mathbf{X})} &= -\frac{ND}{2} \log 2\pi + \frac{ND}{2} \log \beta + \frac{D}{2} \log |\mathbf{K}_{\mathbf{U}\mathbf{U}}| + \frac{1}{2} \log |\mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta \mathbf{\Psi}_2| \\ &\quad - \frac{1}{2} \text{tr} \mathbf{Y}^\top (\beta \mathbf{I} - \beta^2 \mathbf{\Psi}_1 (\mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta \mathbf{\Psi}_2)^{-1} \mathbf{\Psi}_1^\top) \mathbf{Y} \\ &\quad - \frac{\beta}{2} \psi_0 + \frac{\beta}{2} \text{tr}(\mathbf{K}_{\mathbf{U}\mathbf{U}}^{-1} \mathbf{\Psi}_2) \ ,\end{aligned}$$

we can see that only the second term of the expectation of \mathcal{L}_2 under $q_{\mathbf{X}}$ includes the data. Additionally, we can see, that the data is included inside a trace, and as such is a sum across dimensions

$$\begin{aligned}& - \frac{1}{2} \text{tr} \mathbf{Y}^\top (\beta \mathbf{I} - \beta^2 \mathbf{\Psi}_1 (\mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta \mathbf{\Psi}_2)^{-1} \mathbf{\Psi}_1^\top) \mathbf{Y} \\ &= - \frac{1}{2} \sum_{d=1}^D \mathbf{Y}_{\cdot d}^\top (\beta \mathbf{I} - \beta^2 \mathbf{\Psi}_1 (\mathbf{K}_{\mathbf{U}\mathbf{U}} + \beta \mathbf{\Psi}_2)^{-1} \mathbf{\Psi}_1^\top) \mathbf{Y}_{\cdot d} \ .\end{aligned}$$

This means, parallelization over the dimensions D of the bound can be done trivially, by computing these individual sums for each dimension individually across multiple cores. The master node for the parallel computation sums the parts together and computes the KL term as well as other non data dependent terms in each iteration. In fact, we can use this fact when computing the bound for the sparse GP itself.

In high dimensional problems, as for example in gene expression experiments, we can now perform stochastic gradient descent [9, p.240] along the genes to further simplify computational costs. Stochastic gradient descent is an approximation to full gradient descent, in which we update the gradients for all parameters data-point by datapoint. Let $\boldsymbol{\theta}$ be the hyper parameters for Bayesian GPLVM, then the update rule for stochastic gradient descent can be written as

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \frac{\partial}{\partial \boldsymbol{\theta}^{(i)}} \mathcal{L}_3(\mathbf{Y}_{\cdot d}) \ ,$$

with η being an optimization parameter, controlling the step size of the optimization and d being the current dimension of the optimization. We perform the opti-

mization either by cycling through dimensions D , selection d at random from all dimensions D or taking dimensions as batches of smaller size than D , summing gradients before updating the parameters. The advantage of this procedure is, that we can do distributed computations of the bound and do online learning. As the gradient is not computed exact, it is more likely to circumvent local optima during optimization, and parallelization can be done, distributing gradient updates across multiple workers and updating the global parameters online [49].

This factorization and parallelization is implemented in GPy, such that stochastic gradient descent and parallel computing can be taken advantage of in a real world setting.

2.5.3.2 Factorization in N

Factorization in N is a little harder to see, but we can make use of the trace operators to see a factorization in N in the bound [25]. We need to rewrite some terms in the bound in order to see the factorization in N :

$$\begin{aligned}\Psi_0 &= \text{tr}(\mathbf{K}_{\mathbf{F}\mathbf{F}}) = \sum_i \langle k(\mathbf{X}_i, \mathbf{X}_i) \rangle \\ \Psi_1^\top \mathbf{Y} &= \sum_{i=1}^N \langle k(\mathbf{X}_i, \mathbf{Z})^\top \rangle \mathbf{y}_i \quad \in \mathbb{R}^{M \times D} \\ \Psi_2 &= \sum_{i=1}^N \langle k(\mathbf{Z}, \mathbf{X}_i) k(\mathbf{X}_i, \mathbf{Z})^\top \rangle \quad \in \mathbb{R}^{M \times M} \\ \text{KL}(q(\mathbf{X}) \| p(\mathbf{X})) &= \sum_{i=1}^N \text{KL}(q(\mathbf{X}_i) \| p(\mathbf{X}_i)) \quad .(\text{if } q(\mathbf{X}_i) \text{ factorizes})\end{aligned}$$

This makes the four parts above that factorize in N . We can implement workers to work on sub parts of the data in N . The workers return partial sums of the four parts. The master will then sum those parts together and calculate the partial derivatives for $\frac{\partial \mathcal{L}_3}{\partial \Psi_1^\top \mathbf{Y}}$ and $\frac{\partial \mathcal{L}_3}{\partial \Psi_2}$. The workers can then calculate the full derivatives for the kernel parameters θ , likelihood precision β and inducing inputs \mathbf{Z} . Here, the partial gradients w.r.t. \mathbf{Z} factorize in N again and can be computed on the worker nodes. The partial gradients get sent back to the master and the master sums them up, updating the gradients of the global parameters. While the master sums the gradients for the global parameters, the workers can update the gradients for their local parameters \mathbf{M} and \mathbf{S} , if in a Bayesian GPLVM setting.

2.5.4 Large Scale Bayesian GPLVM

These two factorization properties can now be used to factorize both in D and in N . We will factorize first in D , distributing for each batch in D the worker load in

batches in N . All workers in D need a full copy of the global and local parameters, where the sub workers in N will only have the global and their local parameters. This makes Bayesian GPLVM widely applicable to datasets with millions of data points, assuming a big enough cluster to fit the model into memory. At the time of thesis writing, there is no implementations known (or public) that implement both factorizations at the same time. Only the factorization across dimensions D is implemented and accessible through GPY. Additionally, applying the idea of factorization across dimensions D GPY has the capability of handling missing data at random by the GP marginalization property, which allows us to ignore missing data points in a high dimensional setting. This, however, adds the number of dimensions to the complexity, raising from $\mathcal{O}(NM^2)$ to $\mathcal{O}(NM^2D)$ when allowing for missing data at random.

The implementation of Bayesian GPLVM, factorization across dimensions, stochastic gradient descent and missing data is part of this thesis contribution, supplied through GPY.

2.6 MRD: Manifold Relevance Determination

As already seen in Section 2.5, the Bayesian GPLVM bound factorizes across dimensions D of the observed data $\mathbf{Y} \in \mathbb{R}^{N \times D}$. Manifold Relevance Determination (MRD) [20] makes use of that fact to handle multiple subsets of dimensions or multiple datasets, which are taken from the same samples independently. Each dataset \mathbf{Y}^i (or subset of data across dimensions) gets its own functional relation $\mathbf{F}_i(\mathbf{X})$ assigned to it, while learning a shared latent representation \mathbf{X} for all observed datasets sets jointly. Based on biological knowledge about the analyzed data, we can split the extracted data into subsets of functional groups. For example, we can select the genes associated to cell cycle activity as confounding factor subset and find independent dimensions in the latent space \mathbf{X} for those. A non-Bayesian two-step approach for this idea of separating non-wanted variation from the signal of interest was proposed by Buettner et al. [12] in 2015.

To explain the MRD approach, we will define the MRD bound for C independent datasets $\mathbf{Y}^i, 1 \leq i \leq C$. Importantly, these datasets need to be sample aligned, that is, all datasets have to be taken from the same samples across N , but could be different measurements in the dimensions. As an example, the original authors in [20] relate two dimensional shadows of three dimensional positions of stick men. The idea is to be able to predict the position of a stick man in three dimensions given a two dimensional shadow of the same man. In gene expression measurements, we can think of functional groups of genes as different datasets, which we

want to model jointly. By doing the MRD approach we can find independent dimensions in the latent space \mathbf{X} for each subset of genes, as we assign independent GP priors to each dataset (datasets are independent given the latent space \mathbf{X}).

We assign each subset of genes a different covariance function to jointly model the datasets \mathbf{Y}^i with one latent space $q(\mathbf{X})$. The different datasets have each a function

$$f^i(\mathbf{X} + \varepsilon_{\mathbf{X}}) + \varepsilon_{\mathbf{Y}} = \mathbf{Y}^i$$

relating the input space $q(\mathbf{X})$ to the observed sets \mathbf{Y}^i .

To define the MRD bound, we will make the kernel a parameter of the Bayesian GPLVM, as the kernel is the only parameter defining the GP prior for each of the datasets. With this in mind, the Bayesian GPLVM bound can be written as follows

$$\mathcal{L}_3(\mathbf{Y}^i, \mathbf{K}^i) := \langle \mathcal{L}_2(\mathbf{Y}^i, \mathbf{K}^i) \rangle_{q(\mathbf{X})} - \text{KL}(q(\mathbf{X}) \| p(\mathbf{X})) \text{ ,}$$

so that the covariance function \mathbf{K}^i corresponds to the observed dataset \mathbf{Y}^i . With this, we can assign one kernel for each of the datasets and sum the bound together to make up the full MRD bound. The MRD bound sums across datasets inside the expectation of \mathcal{L}_2 and computes the KL divergence outside:

$$\mathcal{L}_4\left(\bigcup_{i=1}^C \mathbf{Y}^i\right) := \left(\sum_{i=1}^C \langle \mathcal{L}_2(\mathbf{Y}^i, \mathbf{K}^i) \rangle_{q(\mathbf{X})} \right) - \text{KL}(q(\mathbf{X}) \| p(\mathbf{X})) \text{ .}$$

Each covariance function k^i retains its hyper parameters θ^i , so that the functions explaining the observed subsets of data can differ.

Now each of the datasets \mathbf{Y}^i (or subsets of dimensions) have their own latent function f^i mapping each dataset \mathbf{Y}^i to a common latent space $q(\mathbf{X})$. To assign dimensions of the common latent space to datasets, we make use of the ARD parameters of the kernel. If two (or more) ARD parameters of kernels \mathbf{K}^i overlap (are “switched on”, compare Sec. 2.1.5), the corresponding dimensions of $q(\mathbf{X})$ are shared across the datasets \mathbf{Y}^i . Thus, we can unravel the dependance structure of the latent space on the subsets \mathbf{Y}^i of the observed data.

As the partition across datasets is inside the sum in the normal Bayesian GPLVM, we preserve all properties of the normal Bayesian GPLVM while introducing additional insight into the observed data.

2.6.1 Intuition and Simulation

To gain some intuitional insight into MRD, we will show a simulation to explain the model. We now want to simulate three different subsets of observed data $\{\mathbf{Y}^i : 1 \leq i \leq 3\}$, by mixing different latent functions linearly. This means, we take

only some of the input functions to simulate a dataset \mathbf{Y}^i . After mixing, we push the datasets to higher dimensions for each dataset set \mathbf{Y}^i to have different dimensionality from each other. We perform the increase in dimensionality by multiplying with a matrix $\mathbf{A}^i \in \mathbb{R}^{q^i \times D^i}$ of high dimensional noise $\mathbf{A}_{jk}^i = \mathcal{N}(0, \alpha_i)$. Each output data set \mathbf{Y}^i has its own noise variance, input dimensionality q^i and output dimensionality D^i assigned to it. This means, with this simulation, we will show several aspects of the strengths of MRD:

- Ability to handle differing signal to noise ratios for each dataset \mathbf{Y}^i by differing noise variances in \mathbf{A}^i .
- Learn differing mappings f^i for each dataset \mathbf{Y}^i by allowing different kernel structures \mathbf{K}^i .
- Identify independency (and dependency) structures from different input dimensions q of the input \mathbf{X}_q to the observed datasets by ARD parameterization of the covariance functions \mathbf{K}^i .

2.6.1.1 Simulation of Datasets

To simulate the different datasets, we first generate the different latent functions. We simulate three different latent functions on an ordering (underlying time) \mathbf{t} . Time is used for visualization purposes to give an order to the latent function. It is not needed to learn the latent representation. The first latent function is a cosine $\ell_1 = \cos(\mathbf{t})$ of the underlying time \mathbf{t} . The second latent input is a sine $\ell_2 = \sin(\mathbf{t})$ of the underlying time ordering \mathbf{t} . Finally, the last latent input is a negative exponential negative cosine $\ell_3 = -\exp\{-\cos(2\mathbf{t})\}$ with doubled frequency. Note, here ℓ are generating the input matrix $\mathbf{X} := [\ell_i]_{1 \leq i \leq 4}$ for the simulation.

We simulate three different datasets \mathbf{Y}^i from mixtures of the above defined functions ℓ :

- The first simulated dataset $\mathbf{Y}^1 = [\ell_1, \ell_2] \cdot \mathbf{A}^{\mathbf{s}^1}$, is generated using the first two input functions $[\ell_1, \ell_2]$. It has $\alpha_1 = 0.3$ noise variance added to it.
- The second simulated dataset \mathbf{Y}^2 contains only the first simulated latent input ℓ_1 with an added noise variance of $\alpha_2 = 0.2$.
- And \mathbf{Y}^3 is generated from all three latent inputs combined $[\ell_1, \ell_2, \ell_3]$ with $\alpha_{\mathbf{s}^3} = 0.25$.

See Table 2.2 for a clear representation of which combination of input functions ℓ_i generates which dataset \mathbf{Y}^i . Additionally, we show an illustration of the three simulated datasets in Figure 2.7.

Dataset	Input Function			α
	$\ell_1 = \cos(\mathbf{t})$	$\ell_2 = \sin(\mathbf{t})$	$\ell_3 = -\exp\{-\cos(2\mathbf{t})\}$	
\mathbf{Y}^1	✓	✓		0.3
\mathbf{Y}^2	✓			0.2
\mathbf{Y}^3	✓	✓	✓	0.25

Table 2.2: Simulation of three datasets \mathbf{Y}^i to explain MRD dependence structure deduction. The table shows which latent inputs ℓ_i where used to generate each dataset and how much noise variance α_i was added.

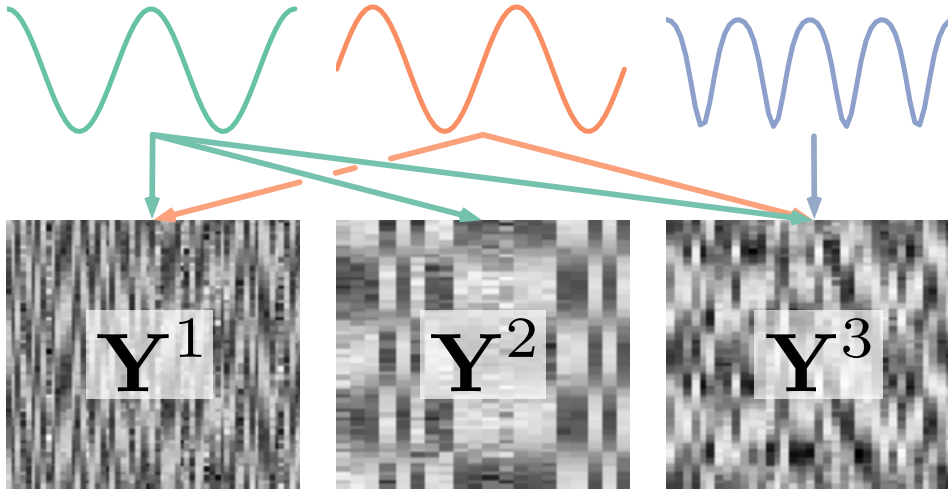


Figure 2.7: Simulation of three datasets to illustrate MRD. The three simulated functions ℓ_i are shown at the top as lines, the combinations are shown as arrows pointing to the 3 resulting datasets \mathbf{Y}^i , shown as grey scale images with samples in rows and dimensions in columns. Note the differing number of dimensions across simulated datasets. The rows for each of the datasets have to match, so that each row of all datasets \mathbf{Y}_j^i corresponds to the same time point \mathbf{t}_j .

2.6.1.2 MRD Model Learning and Results

The MRD only gets the three simulated datasets \mathbf{Y}^i as inputs. Everything else is being deduced by the model and making use of the model structure. We learn the model allowing for 4 latent dimensions $q(\mathbf{X}) \in \mathbb{R}^{N \times 4}$, to show that one latent dimension is being learnt as non informative. In Figure 2.8, we summarize the results from the MRD learning. We plot the mean \mathbf{M}_i and 95% confidence interval $2\sqrt{\mathbf{S}_i}$ of the learnt latent space $q(\mathbf{X}) = \mathcal{N}(\mathbf{M}, \mathbf{S})$ for each of the dimensions on the left side of the plot as lines and shaded areas. The confidence intervals for the three first learnt dimensions are too small to see the shaded areas. The samples are plotted from left to right for visibility. As you can see, the three input signals ℓ get learnt with high confidence. Additionally, the last learnt dimension has mean $\mathbf{M}_{.4} = 0$ and a variance of $\mathbf{S}_{.4} = 1$, which indicates a non informative dimension.

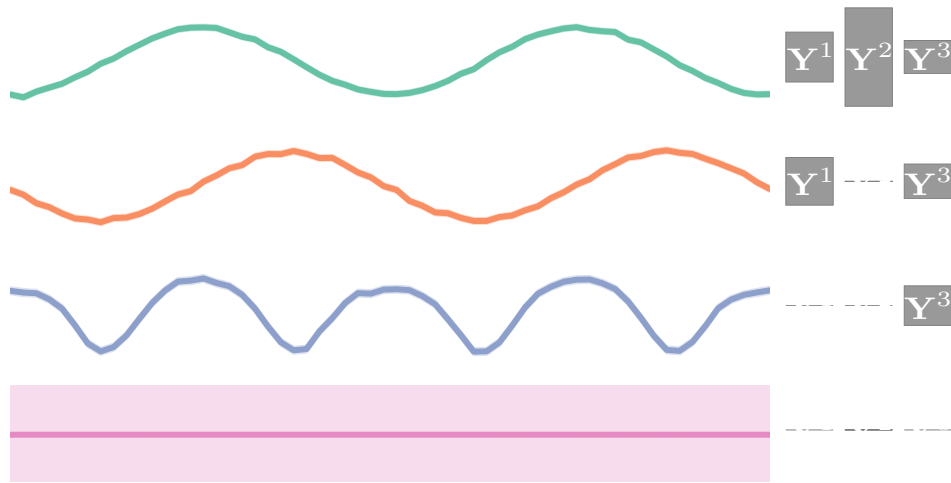


Figure 2.8: MRD simulation with three different subsets of observed data. We plot the mean M_i as thick line and 95% confidence interval $2\sqrt{S_i}$ as shaded area of the learnt latent space $q(\mathbf{X}) = \mathcal{N}(\mathbf{M}, \mathbf{S})$ for each of the dimensions i on the left. On the right, we plot the ARD parameters (height of the grey vertical bars) corresponding to the latent dimensions $\mathbf{X}_{\cdot q}$ (rows), respectively for each dataset \mathbf{Y}_{s_i} . We can see, that the MRD model conforms to the simulation as shown in Figure 2.7). Note also, the non-informative last dimension (last row).

This part can be learnt using a simple GPLVM model on the concatenation of all datasets across dimensions (or even PCA as we simulated a linear relationship in the data).

MRD, however, gives us additional insight on top of the input signals recovered. It tells us which dimensions come from which dataset provided. We recover this information through the ARD parameters of each covariance function k^i for each dataset \mathbf{Y}^i . Each covariance function supplies one ARD parameter per dimension of the latent space. Here, this was $q = 4$. So we have 3×4 ARD parameters. To show the dependence structure of the datasets on the learnt latent spaces, we plot these ARD parameters to the right hand side of the dimensions (Fig. 2.8). Thus, each dimension gets 3 ARD parameter bars, indicating which dataset includes the signals. The ARD parameters show either “switched on” or “switched off” signals, indicated by either visible bars, or a “switched off” state indicated by a line. For example, we can deduce that the input signal corresponding to the first latent dimension (the first row in the plot) is active in all three datasets. And we can see, that the last non informative dimension is not active in either of the datasets.

In summary, the MRD model is able to find the latent inputs for all of the subsets jointly, while selecting which of the latent input dimensions correspond to which subset. We make use of this to assign latent dimensions to different subsets of genes. For example, we can subset an observed gene expression matrix into cell cy-

cle related genes and other genes. Usually in gene expression experiments, we are not interested in cell cycle related activity. Cell cycle activity is basic functionality and usually not associated with tissue function. We can then find the shared latent space for those genes and assign latent dimension to either cell cycle related or not. This jointly corrects for cell cycle related variation, as in the private dimensions for the other genes the cell cycle related activity is not contained. This means, we can use the non cell cycle dimensions for further analyses, minimizing confounding cell cycle related activity.

The MRD model provides a powerful tool of modelling subsets of genes jointly, while learning a shared lower dimensional representation. The ARD parameters on the covariance function for each subset give insights into the composition of latent dimensions explaining the observed data.

The MRD model was implemented and added to GPy during the course of this thesis and presents another contribution of this thesis.

Chapter 3

Case Study GPy

In this chapter, we will discuss the implementation of a complex machine learning algorithm and what thoughts go into the process of undertaking a software implementation for research software design. This project is a collaboration of many people and researchers and part of the contributions where done as part of this thesis.

GPy is an open source Gaussian process framework implementing the algorithms described before in a single combined framework. We will start showing how to decompose a Gaussian process into its primary components and collecting and combining these components for a good flow and interchangeability. This particular implementation uses Python (Sec. B.1) as the primary programming language, but all thoughts in this manuscript can be transferred to other programming languages. In this chapter, we will show our chosen path of successfully implementing a general framework around an algorithm to allow extendability by encapsulation, whilst also considering numerical stability issues and separation of code. This means, we will show some parts of real code interlaced with mathematical and algorithm challenges and thoughts behind the implementation. To make this process as easy as possible, we will start with a simple implementation and extend it towards the numerically stable, fully interchangeable solution for a stable framework. This example is extendable to other algorithms and is to show how to think about an algorithm and the maths involved in order to translate it into a stable framework. This allows for easy extendability and new development of features, potentially not thought of previously.

Gaussian processes are non parametric tools for regression as described in Section 2.1. With modifications they can be extended to classification (e.g. Bishop [9]; Hensman et al. [34]) and dimensionality reduction (e.g. Lawrence [48]; Titsias and Lawrence [86]). All of these modifications have been implemented into GPy by fellow researchers and are accessible through the framework.

First, we will introduce the implementation for a Gaussian process and the parts which go into it. We will discuss how to split the algorithm and make sure extendability is preserved, and a general exchangeable codebase is maintained. This presents a part of this thesis contributions. We will start with a simple implementation of a GP and extend it piece by piece increasing generality and stability (Sec. 3.1ff).

Section 3.10 will discuss and compare GPy to existing implementations. The focus is on the foundational work, extendability, employability and limitations of all implementations.

In the latter part of this Chapter (Sec. 3.12ff), we will discuss the foundation of GPy, handling the front end for the user, as well as providing a sophisticated parameter handling framework for the developer called *paramz*. This underlying package handling parameters and other basic functionality has been a big focus during the course of this thesis and is part of the contributions of this thesis.

3.1 Splitting the Algorithm into Parts

As described in Section 2.1, Gaussian process regression is based on integrating out latent function values \mathbf{F} from the product between likelihood $p(\mathbf{Y}|\mathbf{F})$ and prior $p(\mathbf{F}|\mathbf{X})$. This is the first separation we can find. There is a likelihood and a prior. The likelihood (in the vanilla regression case) is a simple Gaussian distribution, which explains the independent variance from the observed variable \mathbf{Y} to the latent function \mathbf{F} . The prior $p(\mathbf{F}|\mathbf{X}, \boldsymbol{\theta})$ is a multivariate normal with covariance matrix \mathbf{K} , which is built up by the covariance function $\mathbf{K}_{ij} = k(\mathbf{X}_i, \mathbf{X}_j, \boldsymbol{\theta})$.

We use this split to make it explicit in the implementation. We have three parts composing the end result of the Gaussian process: likelihood, prior and the inference around it. The inference computes the marginal likelihood $p(\mathbf{Y}|\mathbf{X})$. It is called the marginal likelihood because it marginalizes over the latent function values \mathbf{F} :

$$p(\mathbf{Y}|\mathbf{X}) = \prod_{i=1}^D \int \overbrace{p(\mathbf{Y}_{\cdot i}|\mathbf{F}_{\cdot i})p(\mathbf{F}_{\cdot i}|\mathbf{X})}^{\text{inference}} d\mathbf{F}_{\cdot i} . \quad (3.1)$$

likelihood
prior

The implementation in GPy follows this structure to make each part interchangeable. In the regression case, the likelihood is usually Gaussian (for non-Gaussian likelihoods in regression the Laplace approximation can be used [9; 70]). The prior is a multivariate normal, solely based on the covariance matrix \mathbf{K} . We make use of this fact to implement different kinds of kernels in GPy.

3.2 Likelihood & Prior

We can extract the kernel from the likelihood, by separating the sum of the covariance of the likelihood $\mathbf{K} + \sigma^2\mathbf{I}$ and treating both separately.

To begin with, the likelihood will be assumed to be Gaussian. For now it is enough to assume the likelihood has a variable called variance, which holds the likelihood variance σ^2 .

```
1 class Gaussian(Likelihood):
2     def __init__(self, variance):
3         self.variance = 1.
4     def Gaussian_variance():
5         return self.variance
```

In the inference, we can access this variance independently from the specific likelihood implementation:

```
1 variance = likelihood.Gaussian_variance()
```

This enables different likelihoods (such as non-Gaussian likelihood approximations) to handle the approximations internally necessary for inference approximations, such as expectation propagation or the Laplace approximation [9].

Second, we assume the kernel class to implement a function to retrieve the kernel matrix \mathbf{K} for a given input \mathbf{X} . In particular, the kernel takes an arbitrary \mathbf{X} and translates it into the covariance matrix $\mathbf{K}_{ij} = k(\mathbf{X}_i, \mathbf{X}_j)$ of pairwise covariances in \mathbf{X} . Optionally, it can also compute pairwise covariances between \mathbf{X} and another \mathbf{X}' .

```
1 class Linear(Covariance):
2     def __init__(self, variance):
3         self.variance = 1.
4     def K(X, X2):
5         return self.variance * X.dot(X2.T)
```

For later use in the inference step, we can retrieve the covariance matrix for the kernel by calling the kernel matrix function:

```
1 K = kernel.K(X, X2)
```

3.3 Inference

As discussed in Section 2.1.1, we optimize the negative log marginal likelihood. For the implementation this means we need a likelihood, a prior and data to perform the inference.

Remember that the covariance function is evaluated at all pairs of rows \mathbf{X}_i of \mathbf{X} to create the covariance matrix $\mathbf{K}_{ij} = k(\mathbf{X}_i, \mathbf{X}_j, \boldsymbol{\theta})$. The log marginal likelihood

of the inference can be written as (Eq. (2.1))

$$\begin{aligned} \log p(\mathbf{Y}|\mathbf{X}, \boldsymbol{\theta}, \sigma^2) &= \log \left(\left(D \left((2\pi)^N |\mathbf{K} + \sigma^2 \mathbf{I}| \right)^{-\frac{1}{2}} \right) \exp \left\{ -\frac{1}{2} \text{tr} \mathbf{Y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y} \right\} \right) \\ &= -\frac{1}{2} \left(ND \log 2\pi + D \log |\mathbf{K} + \sigma^2 \mathbf{I}| + \text{tr} \mathbf{Y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y} \right) \\ &= -\frac{1}{2} \left(ND \log 2\pi + D \log |\mathbf{K}_{\mathcal{GP}}| + \text{tr} \mathbf{Y}^\top \mathbf{K}_{\mathcal{GP}}^{-1} \mathbf{Y} \right) . \end{aligned}$$

This equation at first glance looks cluttered and difficult to cut into exchangeable parts, but at a closer look we can do just that. We want to perform inference over the hyper-parameters $\boldsymbol{\theta}$ of the kernel function (see 2.1.2) and the likelihood variance σ^2 by maximizing the log marginal likelihood

$$\{\hat{\boldsymbol{\theta}}, \hat{\sigma}^2\} = \arg \max_{\boldsymbol{\theta}, \sigma^2} \log p(\mathbf{Y}|\mathbf{X}, \boldsymbol{\theta}, \sigma^2) .$$

Now we can start the implementation of the exact Gaussian process inference procedure. The data is given as \mathbf{X} and \mathbf{Y} , so that data point pairs match up as $(\mathbf{X}_i, \mathbf{Y}_i)_{i=1}^N \in (\mathbf{X}, \mathbf{Y})$.

```

1 def inference(kernel, X, likelihood, Y):
2     K = kernel.K(X) # Kernel matrix for pairs in X
3     sigma2 = likelihood.Gaussian_variance() # Gaussian variance of
4         likelihood
5     Sigma = K + np.eye(K.shape[0])*sigma2 # Covariance of GP
6     Sigma_i = np.linalg.inv(Sigma) # Inverse of Sigma
7     alpha = Sigma_i.dot(Y)
8     data_fit = np.trace(Y.T.dot(alpha)) # trace over dimensions in Y
9     const = np.log(2*np.pi) # normalisation constant
10    complexity = np.log(np.linalg.det(Sigma)) # normalisation for
11        covariance
12
13    log_marginal_likelihood = -.5 * (const + complexity + data_fit)
14
15    return log_marginal_likelihood

```

3.4 Numerical Stability

In this section, we will use insights into numerical stability to improve the stability of the inference for GPs. Numerical stability is an important issue and has to be accounted for when implementing a framework to be used by a wide range of users. There is often a trade-off between generality of implementation and the numerical stability of the application itself. In GPpy, we chose to split the algorithm at a convenient point, where implementation of new algorithms and ideas is promoted, while keeping numerical improvements and implementation freedom as clear cut as possible.

The inference of GPs relies on the inverse of the covariance matrix $\mathbf{K}_{ij} = k(\mathbf{X}_i, \mathbf{X}_j)$ of the covariance function k of the prior $p(\mathbf{F}|\mathbf{X})$. In a general framework implementation of GPs, we need to make sure, the inference is numerically stable for a wide range of covariance matrices. Numerical stability in algebra is a wide field and we will only explain specific parts necessary for the explanation of the implementation shown in this thesis. See Higham [36] for a detailed description of numerical stability and possible solutions. One aspect of the implementation in which the numerical stability can provide erroneous results is the inverse of the covariance matrix, and determinant of a covariance matrix. For both of these tasks, numerical stability can be improved by using the Cholesky decomposition

$$\mathbf{A} = \mathbf{L}\mathbf{L}^\top ,$$

where \mathbf{L} is a lower triangular matrix with positive diagonal entries. The Cholesky decomposition requires matrix \mathbf{A} to be positive semi-definite [36, Chapter 10], which is also the requirement for kernel functions [70].

We take the log-likelihood of a GP and turn it into a numerically stable equation for use in the actual implementation in a programming language. Now we can have a closer look at the summands. The first summand $\log 2\pi$ can be precomputed as a constant. The second summand can be stabilized numerically by the Cholesky factorization $\mathbf{L}\mathbf{L}^\top = \mathbf{K}$.

$$\begin{aligned} \log |\mathbf{K}| &= \log |\mathbf{L}\mathbf{L}^\top| \\ &= \log(|\mathbf{L}| \cdot |\mathbf{L}|) = 2 \log(|\mathbf{L}|) \\ &= 2 \log \left(\prod_{i=1}^N \mathbf{L}_{ii} \right) \quad \mathbf{L} \text{ is triangular} \\ &= 2 \sum_{i=1}^N \log \mathbf{L}_{ii} . \end{aligned}$$

The last summand $\text{tr } \mathbf{Y}^\top \mathbf{K}_{\mathcal{G}P}^{-1} \mathbf{Y}$ involves the inverse of the GP covariance matrix, which is being stabilized by making use of a special matrix solve – `dpotrs` – solving the system $\mathbf{S}\mathbf{x} = \mathbf{B}$ using the lower triangular Cholesky decomposition \mathbf{L}_S of \mathbf{S} . It first solves $\mathbf{L}_S \mathbf{y} = \mathbf{S}$ and using that solution solves $\mathbf{L}_S^\top \mathbf{x} = \mathbf{y}$. Both these operations can be done in quadratic time, as the cholesky decomposition \mathbf{L}_S is already triangular.

```

1 def inference(self, kern, X, likelihood, Y):
2     K = kern.K(X).copy()
3     Sigma = K + np.eye(K.shape[0])*(likelihood.variance+1e-8)
4     # add constant jitter for numerical stability
5     # Get Cholesky decomposition and computations
6     Si, LS, LSi, S_logdet = GPY.util.linalg.pdinv(Sigma)

```

```

7 # Si: Sigma^{-1}
8 # LS: Lower traingular Cholesky decomposition
9 # LSi: Inverse of the above
10 # S_logdet: \log\det(Sigma)
11 alpha, _ = GPy.util.linalg.dpotrs(LS, Y, lower=1)
12 # dpotrs solves Sx = Y using lower triangular
13 # Cholesky decomposition LS of S
14 # log marginal likelihood parts
15 data_fit = np.trace(alpha.T.dot(Y))
16 # Trace is for more than one dimension in Y
17 const = Y.size*np.log(2*np.pi)
18 complexity = Y.shape[1]*S_logdet
19 # marginal likelihood
20 log_marginal_likelihood = -.5 * (const + complexity + data_fit)
21 return log_marginal_likelihood

```

3.5 Posterior Prediction

The posterior of the GP is the fitted predictive distribution given the data and parameters of the likelihood and kernel. To predict at arbitrary (new) points \mathbf{X}^* we need to evaluate the posterior distribution $p(\mathbf{F}^*|\mathbf{X}^*, \mathbf{X}, \mathbf{Y}, \boldsymbol{\theta})$ (Eq. 2.8). For this, we employ another close look at the mathematical expression. We can see, that we can divide the prediction into two parts: training data part \mathcal{D} and prediction covariance part \mathbf{K}^* . The following equation shows the two parts for mean and covariance prediction at new points \mathbf{X}^* by over setting the specific parts:

$$\begin{aligned}
p(\mathbf{F}^*|\mathbf{X}^*, \mathbf{X}, \mathbf{Y}) &= \mathcal{N}(\mathbf{F}^*|\mathbf{M}, \boldsymbol{\Sigma}) \\
\mathbf{M} &= \mathbf{K}_{\mathbf{F}^*\mathbf{F}}^* \overbrace{(\mathbf{K}_{\mathbf{F}\mathbf{F}} + \sigma^2\mathbf{I})^{-1}\mathbf{Y}}^{\mathcal{D}} \\
\boldsymbol{\Sigma} &= \mathbf{K}_{\mathbf{F}^*\mathbf{F}^*}^* - \mathbf{K}_{\mathbf{F}^*\mathbf{F}}^* \overbrace{(\mathbf{K}_{\mathbf{F}\mathbf{F}} + \sigma^2\mathbf{I})^{-1}}^{\mathcal{D}} \mathbf{K}_{\mathbf{F}\mathbf{F}^*}^* .
\end{aligned}$$

The training data part is the only part we need to store to predict at new data points on demand (by the user). We will store these matrices for later usage.

$$\begin{aligned}
\mathbf{M} &= \mathbf{K}_{\mathbf{F}^*\mathbf{F}}^* \overbrace{(\mathbf{K}_{\mathbf{F}\mathbf{F}} + \sigma^2\mathbf{I})^{-1}\mathbf{Y}}^{\text{alpha}} \\
\boldsymbol{\Sigma} &= \mathbf{K}_{\mathbf{F}^*\mathbf{F}^*}^* - \mathbf{K}_{\mathbf{F}^*\mathbf{F}}^* \overbrace{(\mathbf{K}_{\mathbf{F}\mathbf{F}} + \sigma^2\mathbf{I})^{-1}}^{\text{Si}} \mathbf{K}_{\mathbf{F}\mathbf{F}^*}^* .
\end{aligned}$$

In the code, `alpha` can be computed using the Cholesky decomposition of the GP covariance. Using the insights into numerical stability from above (Sec. 3.4), we can additionally employ the Cholesky factorization lower triangular matrix `LS` to

solve the linear system with respect to \mathbf{Y} .

This allows us to only store the two data parts in a posterior object and then predict on demand by the user at new points \mathbf{X}^* by evaluating the respective prediction covariance parts. We also define the inference method to be a class, so it is exchangeable with a different inference method. This is being used for the different methods of inference, for example sparse GP, Bayesian GPLVM, Expectation propagation and others available in GPy.

The exact Gaussian process posterior class and inference class are defined as follows:

```

1 class Posterior(object):
2     def __init__(self, alpha, LS):
3         self.alpha = alpha
4         self.LS = LS
5     def predict(self, kern, Xstar, X):
6         Kstar = kern.K(Xstar, X)
7         mu = Kstar.dot(self.alpha)
8         Kss = kern.K(Xstar, Xstar)
9         tmp = GPy.util.linalg.dtrtrs(self.LS, Kx)[0]
10        # dtrtrs: solve system S x = B,
11        # using lower triangular Cholesky LS of S
12        var = Kss - GPy.util.linalg.tdot(tmp.T)
13        # tdot(X) = X.T.dot(X)
14        return mu, var
15
16 class ExactGaussianInference(Inference):
17     def inference(self, kernel, X, likelihood, Y):
18         ...
19         posterior = Posterior(K, alpha, LS)
20         ...
21     return posterior, log_marginal_likelihood

```

3.6 Gradients

As described in Section 2.1.1, we use the concept of the chain rule to aid computation of gradients and maintain separability between components. Specifically, the gradient of the above w.r.t. an arbitrary parameter α can be seen as scalar gradients (remember $\mathcal{L} := \log p(\mathbf{Y}|\mathbf{X})$)

$$\frac{\partial \mathcal{L}}{\partial \theta_t} = \sum_{i=1}^N \sum_{j=1}^N \frac{\partial \mathcal{L}}{\partial [\mathbf{K}_{GP}]_{ij}} \frac{\partial [\mathbf{K}_{GP}]_{ij}}{\partial \theta_t},$$

as each value in the matrix is a singular call to the covariance function k .

To implement a Gaussian process in a general setting, we just need to implement the general gradients of the posterior $p(\mathbf{Y}|\mathbf{X})$ with respect to to the GP covariance matrix \mathbf{K}_{GP} and then chain rule it with the implementation of the gradients

of the specific kernels we want to use. This allows any type of kernel to be implemented without the need for the model to know specifics about the kernel and vice versa. One caveat of this separation can be if the specific kernel in combination with the specific inference could allow for faster computations of the gradients or marginal likelihood by harnessing the specific mathematics.

As we can see from Equation 2.5, the gradient of the log marginal likelihood with respect to the GP covariance can be written as

$$\mathbf{K}_{\mathcal{GP}}^{-1} \mathbf{Y} \mathbf{Y}^{\top} \mathbf{K}_{\mathcal{GP}}^{-1} - \frac{D}{2} \mathbf{K}_{\mathcal{GP}}^{-1} .$$

In code this looks like the following:

```

1 def inference(self, kern, X, likelihood, Y):
2     ...
3     alpha, _ = GPy.util.linalg.dpotrs(LS, Y, lower=1)
4     ...
5     dL_dK = .5 * (alpha.dot(alpha.T) - Y.shape[1]*Sigma_i)
6     ...
7     return posterior, log_marginal_likelihood, dict(dL_dK=dL_dK)

```

3.7 Optimization

To learn an optimal model for the data seen, we need to maximize the log marginal likelihood with respect to the hyper-parameters θ , which in the case of linear is α and in the case of exponentiated quadratic $\{\sigma_f^2, \alpha\}$. For most optimization problems minimizing is well described, so we will minimize the *negative* log marginal likelihood in order to achieve a best fit for the data seen.

There are several different optimization algorithms and we recommend Limited-Broyden-Fletcher-Goldfarb-Shanno (L-BFGS-B) from personal experience. L-BFGS-B is a limited memory version of the original BFGS algorithm, which saves an approximation to the Hessian of the optimization problem to compute the directions of subsequent steps more accurately. The limited memory version requires only linear machine space [13]. In the years of working with Gaussian Processes L-BFGS-B has proven to be an invaluable algorithm to solve large scale optimization problems (with thousands of parameters).

The ability for the GP to explain the data using only the likelihood variance (i.e. has the option to explain all data as noise) makes initialization a crucial step in optimizing Gaussian process models. We want to initialize the prior hyper-parameters, such that the GP sees signal, and will try to explain the variance in the data using the prior \mathbf{K} , as opposed to the likelihood variance σ^2 . This is due to the variance parameter having a higher influence on the GP fit, than the covariance function

parameters, as the variance is directly added as independent noise to the diagonal of the GP. Thus, the GP fit of explaining everything by noise variance is a local optimum, which when reached can be hard to get out of.

3.8 Mean Function

The separation of the log marginal likelihood into parts enables us to also add in a mean function into the equations. The mean $\boldsymbol{\mu}$ is subtracted from the observed outputs \mathbf{Y} in the `data_fit`. As we have the separation of components, we can easily introduce a mean function of the inputs by evaluating it at \mathbf{X} and subtracting it from the observed data before doing the inference.

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}, \sigma^2) = \log \left(\left(2\pi |\mathbf{K} + \sigma^2 \mathbf{I}|^{-\frac{1}{2}} \right) \exp \left\{ (\mathbf{y} - \boldsymbol{\mu})^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} (\mathbf{y} - \boldsymbol{\mu}) \right\} \right) \quad (3.2)$$

$$= -\frac{1}{2} \left(\log 2\pi + \log |\mathbf{K} + \sigma^2 \mathbf{I}| + (\mathbf{y} - \boldsymbol{\mu})^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} (\mathbf{y} - \boldsymbol{\mu}) \right) . \quad (3.3)$$

The implementation can be modified by adjusting the observed values \mathbf{Y} by subtracting the mean function $\boldsymbol{\mu}$ evaluated at the inputs \mathbf{X} .

```

1 def inference(self, kern, X, likelihood, Y, mean_function=None):
2     m = mean_function.m(X)
3     Y_residual = Y-m
4     # replace Y with Y_residual from now:
5     ...
6     alpha, _ = GPy.util.linalg.dpotrs(LS, Y_residual, lower=1)
7     ...

```

3.8.1 Gradients

The gradients for the mean function can be pushed through the evaluated mean $\boldsymbol{\mu}$. As we handle multidimensional \mathbf{Y} , we fill a matrix \mathbf{M} with one column mean $\mathbf{M}_{.i} = \boldsymbol{\mu}_i$ per dimension of observed values. With that, we can compute the gradients of the log marginal likelihood with respect to the evaluated mean matrix \mathbf{M} .

$$\begin{aligned} \frac{\partial \log p(\mathbf{Y}|\mathbf{X})}{\partial \mathbf{M}} &= \frac{\partial}{\partial \mathbf{M}} \left(-\frac{1}{2} \text{tr}((\mathbf{Y} - \mathbf{M})^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} (\mathbf{Y} - \mathbf{M})) \right) \\ &= -\frac{1}{2} \frac{\partial}{\partial \mathbf{M}} \mathbf{M}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{M} \quad + \frac{1}{2} \frac{\partial}{\partial \mathbf{M}} 2\mathbf{M}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y} \\ &= -\frac{1}{2} 2 (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{M} \quad + \frac{1}{2} 2 (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y} \\ &= (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} (\mathbf{Y} - \mathbf{M}) , \end{aligned}$$

which is available and computed already in the form of `alpha`.

```

1 def inference(self, kern, X, likelihood, Y, mean_function=None):
2     m = mean_function.m(X)

```

```

3 Y_residual = Y-m # replace Y with Y_residual from now:
4 ...
5 alpha, _ = GPy.util.linalg.dpotrs(LS, Y_residual, lower=1)
6 ...
7 return posterior, log_marginal_likelihood, dict(..., dL_dm=alpha)

```

3.9 Bringing it all Together

For now, we have not described how to actually pull all the parts together to allow for the optimization. Most off-the-shelf optimizers (minimizers) accept an objective function $f(x)$ and gradient evaluation $df(x)$. Thus, we need to transform the parameters, which are currently simple variables, into known parameters for the optimization and have an effective and simple way to pulling them together.

In GPy, we use the package called `paramz` [99], which handles parameterized optimization in an efficient manner. `Paramz` is part of the contributions of this thesis and will be discussed in detail in Section 3.12. It provides routines for parameter constraining, printing, getting and setting and automatic model updates. One optimization step (iteration) is a cycle through all parameters, updating each part individually and calling the inference on the updated parameters. It keeps parameters in memory in one place (collated together into one array), so that getting and setting parameters individually, automatically updates the variable x of the objective function. This reduces overload in getting and setting parameters, as well as minimizes memory requirements of the model.

The `paramz` package also provides that gradients of variables are stored directly on the parameters themselves (also memory controlled), so that each of the parts can control their respective gradients, allowing for separation of code. One optimization cycle (iteration) goes as follows:

1. Update parameters x from optimizer.
2. Call inference with new parameters.
3. Update all gradients for the parameters.
4. Return to optimizer for next step (1).

In `paramz`, this is being done by the dependency injection pattern, linking the optimization cycle to a function called `parameters_changed()`. In GPy, the GP implementation class holds the likelihood, kernel and inference method in local variables `self.likelihood`, `self.kern` and `self.inference_method`. In the implementation of a GP itself, the `parameters_changed` method could look like this:

```

1 def parameters_changed(self):
2     # do inference:
3     tmp = self.inference_method.inference(
4         self.kern, self.X, self.likelihood, self.Y, self.mean_function
5     )
6     # extract inference
7     self.posterior, self._log_marginal_likelihood, self.grad_dict = tmp
8     # Update the gradients for parameters:
9     self.likelihood.update_gradients(self.grad_dict['dL_dK'])
10    self.kern.update_gradients(self.grad_dict['dL_dK'], self.X)
11    if self.mean_function is not None:
12        self.mean_function.update_gradients(self.grad_dict['dL_dm'], self.X)

```

We carefully understood the maths and identified parts, which might be interchanged in future. This allowed us to implement a Gaussian process efficiently and still introduce all necessary stability to the algorithm. The process of unraveling the structure of the Gaussian process also helped understanding Gaussian process regression in more detail. We made sure, that all parts can be independently implemented and changed for future changes in covariance functions, likelihoods or newly added mean functions. This gives a full overview of a way of implementing a full Gaussian process inference method using the open source paramz framework that forms one of the contributions of this thesis. The implementation of Gaussian processes can be found in the GPpy package [27]. Some parts may differ, as the GPpy implementation has more modular parts and more algorithmic considerations. The implementation presented here is fully functioning and can be used to do Gaussian process regression.

3.10 Comparison to other implementations

GPpy is a package to make it easy to run Gaussian process based models. We focus on ease-of-use for the user, while allowing freedom to implement new models into the existing framework. We welcome any contribution from researchers, who develop GP based applications. Separation of parts in implementation, as shown above, allows for researchers to focus on their part, when contributing. Any optimization of runtime and memory efficiency can be achieved through the structure given in the framework.

In this section we compare GPpy to two other frameworks providing Gaussian process based applications, to show the focus and difference in philosophies.

3.10.1 GPFlow

GPFlow is a similar implementation to Gaussian process regression as explained above. The main difference of GPFlow to GPpy is, that GPFlow makes use of the

Dataset	Model	GPy [s]	GPflow [s]	sklearn [s]
Oil	BayesianGPLVM	5.41 ±0.17	54.11±1.54	N/A
Oil	GPLVM	64.85 ±10.00	161.87±1.68	N/A
Simulated 1D	GPR	0.13 ±0.05	13.35±10.14	0.20±0.09

Table 3.1: Comparison of GPy versus GPflow and scikit-learn (sklearn). Two different datasets were generated: first, "Oil" is taken from Bishop [9, p. 678] with 1000 data points, ran for 200 iterations, respectively. Second, "Simulated 1D" is a one dimensional dataset with 50 training data points and 100 test data points. It was generated from a multivariate normal distribution with an exponentiated quadratic covariance function. As the implementations differ significantly, we compare all steps of fitting a GP: Model generation, optimizing hyper-parameters until convergence and predicting with the model. As mentioned in the text, scikit-learn does not provide Gaussian process latent variable models.

Google TensorFlow auto-differentiation and distributed computing package to compute gradients and distribute the computational workload. This greatly reduces the amount of code necessary when prototyping new ideas of algorithms. One downside of automated gradient computation could be, when partial gradients, substantial to the model of interest are missing. TensorFlow is an open source project and partial gradients can be submitted as pull requests (Sec. 1.4.4). This has been done before for automatic gradients of the Cholesky decomposition (personal correspondence, <https://github.com/tensorflow/tensorflow/pull/1465>). This is being mentioned to show how open source research software engineering can make research quicker and easier for all participants, one of the main points of this thesis.

Another downside can be speed of computation. Though, recent publications have shown that these are being addressed by the developers in fast turnarounds [68]. Manual computation of gradients provides an opportunity to optimize speed of and efficiency of computation in both memory and time. To show the difference in computation, we compare GPy against GPflow on different models and datasets in Table 3.1. As can be seen, GPflow is in general slower than GPy. Tensorflow provides an interface to GPU optimized computation, which can alleviate the problem by making use of the speed of the GPU in modern computers. This, however can not replace manual mathematical optimization for production code.

3.10.2 scikit-learn

Scikit-learn is a machine learning toolbox for "simple and efficient tools for data mining and data analysis" [62] implemented in Python. The toolbox is designed around a simple principle of creating an instance of an algorithm (creating the class with specific parameter settings), fitting it to data ($\text{fit}(X, Y)$) and then predict-

ing on new data (`predict(X)`). The implementation in scikit-learn follows similar steps as taken in the implementation here for the gradient computation of covariance function (kernel) gradients. The main difference is, that scikit-learn does not take different likelihoods into account, that means the kernel has to define independent noise for regression with variance on the outputs \mathbf{Y} . Additionally, scikit-learn does not allow for parametric mean functions to be added into the regression.

The implementation of algorithms in scikit-learn is focused on usability and implementing a wide spectrum of established algorithms. In GPy, we focus on the in-depth implementation of GP specific algorithms and explore the whole spectrum of Gaussian process based applications. Scikit-learn is a good tool of “trying out” different machine learning algorithms on a dataset. In particular, comparing a new development against standard algorithms of the literature. GPy is meant for optimized development and efficient application of GP based data analysis.

Scikit-learn provides only the basic GP regression for use, but focuses on providing a variety of different machine learning algorithms, not based on GPs. The overlap between GPy and scikit-learn is minimal. The philosophy of scikit-learn is to provide a wide range of well established data analysis algorithms, while GPy provides a more focused framework for “cutting-edge” algorithms involving GPs.

3.11 Plotting and Visualization

GPy provides automated convenience functions for plotting with different plotting frameworks for Python. As of the submission of this thesis, the supported plotting libraries are matplotlib [39] and plotly [65]. Plotting is an important part in understanding a Gaussian process model. In GPy, we focus on providing a simple and intuitive way of plotting different aspects of the GP models to understand the involved processes. For example, the kernels provide functions to plot the covariance function along the inputs. The `GPRRegression` models provide functions to plot the fit and prediction of the model, allowing for sub-selection of which kernels to plot, independently. To illustrate the plotting capabilities, we fitted a model to the Mauna-Loa data [43] with a covariance function as described by Rasmussen and Williams [70, p. 118]. The data contains atmospheric CO₂ readings in Mauna-Loa, Hawaii. We can plot the prediction in the future with all components of the covariance function, as well as the parts, individually. The full process of creating the GP regression model including the covariance function is shown in the following:

```
1 import GPy
2 from sklearn.datasets import fetch_mldata
3
4 data = fetch_mldata('mauna-loa-atmospheric-co2').data
```

```

5 X = data[:, [1]]
6 y = data[:, [0]]
7
8 k1 = GPy.kern.RBF(1, variance=1, lengthscale=100, name='Long term')
9 k2 = GPy.kern.RBF(1, 1, 90) * GPy.kern.StdPeriodic(1, variance=1,
    lengthscale=1.3, period=1)
10 k2.std_periodic.period.fix()
11 k2.name = 'Seasonal'
12 k3 = GPy.kern.RatQuad(1, variance=1, lengthscale=1.2, power=.78, name='
    Medium term irregularities')
13 k4 = GPy.kern.RBF(1, variance=1, lengthscale=1, name='Noise')
14 kernel = GPy.kern.Add([k1, k2, k3, k4])
15
16 m = GPy.models.GPRegression(X, y, kernel=kernel, normalizer=True)
17 m.optimize(messages=1)

```

With that, we show the plotting capabilities of GPy in Figure 3.1, showing the different plots with the respective calls to the GPy framework.

3.12 Parameterization with the Paramz Framework

As already shown above, the paramz framework [99] handles parameterization of the model and relieves the requirement for the developer to implement model optimization. Paramz is part of this thesis' contributions. It provides not only optimization routines, but also provides parameter constraints, parameter fixing, pretty printing, parameter getting and setting with automatic updates to the model, and caching of function calls in a memento pattern. In this section, we will elucidate these features.

Parameter Constraints Sometimes parameters in a model are only allowed to realize certain ranges of values. For example, a parameter α specifying the variance of a normal distribution $\mathcal{N}(0, \alpha)$ is bound to be positive. In paramz, constraining parameters is as simple as calling the `constrain_<how>` function on the parameter itself. For example, if we have the normal distribution as a class `Normal` defined as follows:

```

1 import paramz
2 class Normal(paramz.Parameterized):
3     def __init__(self, variance=1, name='Normal'):
4         super(Normal, self).__init__(name=name)
5         self.variance = paramz.Param("variance", variance)
6         self.link_parameter(self.variance)
7 n = Normal(.5)
8 print(n)
9 # Normal. | value | constraints
10 # variance | 0.5 |

```

we can make sure, that the parameter variance only allows positive values, constraining it positive:

```
1 n.variance.constrain_positive()
2 print(n)
3 # Normal. | value | constraints
4 # variance | 0.5 | +ve
```

Parameter Fixing Parameter fixing is just a special case of constraining parameters. A parameter can be fixed by calling the `fix()` method on the parameter. One can also constrain complete (parts of) models to constrain the whole model (or part): `n.fix()` will cause all children (and their children) of `n` to be fixed. Unfixing is just as simple, by just calling the `unfix()` method on the part to unfix. Importantly, fixing and unfixing will not alter the constraint state of a parameter, so after unfixing, it will still be constrained positive:

```
1 n.variance.fix() # direct selection of parameter
2 print(n)
3 # Normal. | value | constraints
4 # variance | 0.5 | +ve fixed
5 n.unfix() # indirect unfixing of the whole parameterized Normal
6 print(n)
7 # Normal. | value | constraints
8 # variance | 0.5 | +ve
```

Pretty Printing As already shown at a glance above, the `paramz` framework provides all parameterized objects (models and parameters) with a nice printing behaviour. Each `parameterized` object has a name associated with it, given at creation time. The printing will include the name of each object and create a hierarchy showing the connection of the parameters inside the model. To print the values of parameters of bigger models, `paramz` tries to keep the output clean and simple. If a parameter has multiple entries, it will only display the dimensions of the vector or matrix. If it is only one element, it will display the value of the element itself (see above). We extend the definition of the `Normal` to be a multivariate normal distribution with mean `mu` and covariance matrix `cov`:

```
1 import paramz, numpy as np
2 class MultivariateNormal(paramz.Parameterized):
3     def __init__(self, mu, cov, name='MultivariateNormal'):
4         super(MultivariateNormal, self).__init__(name=name)
5         self.mu = paramz.Param("mean", mu)
6         self.cov = paramz.Param("covariance", cov,
7                                 default_constraint=paramz.constraints.Logexp()
8                                 # Set the constraint directly on the parameter at creation time
9                                 )
10        self.link_parameters(self.mu, self.cov)
11 n = MultivariateNormal(np.random.normal(0,1,(3,1)),
```

```

12         np.cov(np.random.normal(0,1,(3,3)))
13 n.mean[2].constrain_bounded(-1,2)
14 print(n)
15 # MultivariateNormal. | value | constraints
16 # mean                | (3, 1) | {-1.0,2.0}
17 # covariance          | (3, 3) | +ve

```

Note, that we put the positive constraint programmatically on the `covariance` parameter at creation time. Additionally, we show the curly brackets around the bound constraint of the third entry of the `mean` to signify that only parts of the full mean are constrained. You can see that the two parameters, `mean` and `covariance`, are displayed with their dimensions as value. If we want to show all values and details of a parameter, we can print them by printing the specific parameter (leaf) of the model directly:

```

1 print(n.mean)
2 # index | MultivariateNormal.mean | constraints
3 # [0 0] | -0.52004708 |
4 # [1 0] | -0.11452660 |
5 # [2 0] | 0.10037635 | -1.0,2.0

```

Parameter Getting and Setting Getting and setting parameters is the same as getting and setting the underlying `numpy` (Sec. B.1.1) array. The advantage of `paramz` is, that it will register the changes and call the `parameters_changed()` method of the changed part of the model. It will also traverse the hierarchy upwards to update the higher part of the model. This makes sure, that no changes to the model will leave it in an inconsistent state, and update routines do not have to be done manually. To show the behaviour of the `paramz` framework, we will override the `parameters_changed()` method to sum the `mean` parameter into a `self.mean_sum` variable:

```

1 class MultivariateNormal(paramz.Parameterized):
2     ...
3     def parameters_changed(self):
4         self.mean_sum = self.mean.sum()
5 np.random.seed(1234)
6 n = MultivariateNormal(np.random.normal(0,1,(3,1)),
7                        np.cov(np.random.normal(0,1,(3,3))))
8 print(n.mean_sum)
9 # 0.713166437452
10 print(n.mean.sum())
11 # 0.713166437452
12 n[:] = 2 # change all parameters under n to be equal to 2
13 print(n.mean_sum) # observe the update from parameters_changed:
14 # 6.0

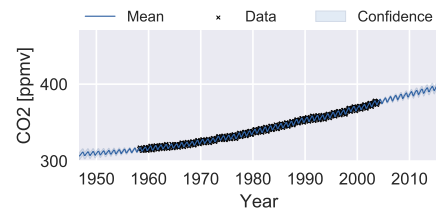
```


3.13 Summary

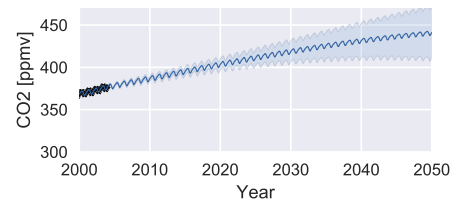
GPy is a collaboration of many researchers and a community effort. This thesis adds to the success of GPy by providing the foundational package `paramz`. Additionally, coordination and correction of additions and the oversight of directions for the GPy package are contributed during the course of this thesis.

We have shown, that by careful consideration of mathematical components of the algorithm, we can implement a complex algorithm in an extendable and stable manner. Breaking the algorithm down into its components allows developers to develop new parts easily, while users can make use of an ever growing feature set of the framework.

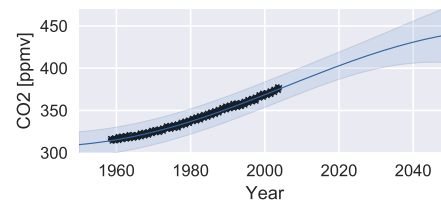
```
1 m.plot(legend='ontop', resolution=500)
```



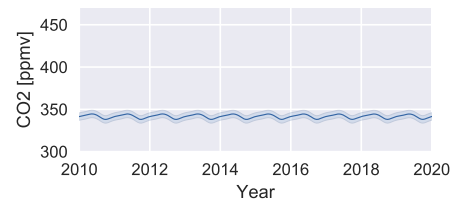
```
1 m.plot(plot_limits=[2000, 2050],
2       legend=False, resolution=500)
```



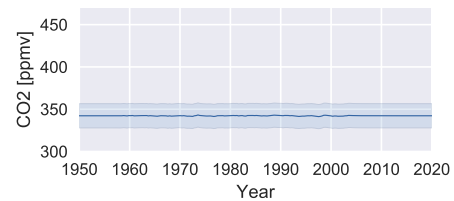
```
1 m.plot(plot_limits=[1950, 2050],
2       predict_kw=dict(kern=m.kern.Long_term),
3       plot_data=True, legend=False)
```



```
1 m.plot(plot_limits=[2010, 2020],
2       predict_kw=dict(kern=m.kern.Seasonal),
3       plot_data=False, legend=False)
```



```
1 m.plot(plot_limits=[1950, 2030],
2       predict_kw=dict(kern=m.kern.
3       Medium_term_irregularities),
3       plot_data=False, legend=False)
```



```
1 m.plot(plot_limits=[2010, 2020],
2       predict_kw=dict(kern=m.kern.Noise),
3       plot_data=False, legend=False)
```

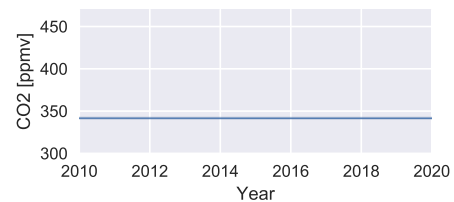


Figure 3.1: Illustration of GPy plotting capabilities. Let m be a GPy GPRegression model fitted to the Mauna-Loa [43] data.

Chapter 4

Applications in Gene Expression Experiments

In this chapter, we will look at applications of the methods described in Section 2 to real world single cell gene expression experiments. Hereby, we want to not only focus on the results in the classical sense, but also elucidate the ease of use of GPy. As mentioned in Section 3, GPy is the underlying package holding a large proportion of work during this thesis. It provides a package to apply Gaussian process (Sec. 2.1) based machine learning to datasets. We will first present the new developed method of Topslam, a method of extracting ordering information from high dimensional datasets. In the next section, we will show the application of Topslam to a single cell gene expression experiment. To show the ease of use of GPy, we will show the optimization process of Bayesian GPLVM using GPy first and show the results of the novel Topslam on this experiment.

Secondly, we will show the impact of GPy to biology research community. Packages have been built on GPy, because of its ease of use and attention to extendability. We will apply software packages provided by us and others to another single cell gene expression experiment. The second dataset is more difficult because of confounding variation and a label mixup at data creation. The time labels for the experiment were mixed up when the dataset was created, so we can only make use of marker genes to show the performance of extracted patterns applying machine learning. This section is to show how packages based on GPy supplied by their respective researchers can work together to provide a sophisticated toolbox for single cell gene expression analyses.

4.1 Topslam: Topographical Simultaneous Localization and Mapping

In this section, we present a new way of extracting pseudo ordering information for high dimensional datasets. This method is part of the contribution for this thesis and is in pre-print Zwiessele and Lawrence [100]. It has been submitted to the BMC Bioinformatics journal for publication.

Extracting pseudo time ordering is a way of ordering samples of a high dimensional experiment by supplying pseudo time stamps to each sample according to their intrinsic ordering. In single cell gene experiments, we usually have a rough estimate of time for each extracted cell, as we can keep track of the days at which the cells were fixed (stopped from their respective differentiation or development). Each cell, however, might have their own intrinsic time at which it divides or develops, and thus the rough estimates of extraction time might be misleading. Additionally, in many downstream analyses, such as Gaussian processes, it is preferable to have a more high resolution of time. This led to the methodology of extracting pseudo time ordering for single cell experiments. Note, that pseudo time ordering can be applied to any high dimensional extraction of information of samples.

In this section, we will first describe the predominant way of extracting pseudo time orderings as presented in the literature. Then we will show the newly developed method of Topslam as part of this thesis and last apply it to a real world dataset to show its performance of extracting the ordering information for a complex differentiation pattern. Third, we will show that we can overlay Topslam on top of other dimensionality reduction techniques and correct distances using the manifold embedding metric (Sec. 4.1.2.1) for those. We will show, that correcting for distances using Topslam is not detrimental to pseudo time ordering extraction and can lead to significant improvements in performance. And in contrast, we will compare extracting the pseudo time ordering with correction using the manifold embedding metric against extracting the pseudo time ordering without correcting for distorted distances.

4.1.1 State-of-the-Art Pseudo Time Ordering Extraction

In pseudo time ordering, we make one assumption from which everything follows. The assumption is, that similar looking patterns by a defined metric are close in time. This is a smoothness assumption for high dimensionality. Thus, when two patterns for a sample match up, their respective time ordering is assumed to be equal. From this, we can describe the predominant way of extracting pseudo time orderings from datasets in the literature:

- First, we extract a lower dimensional representation $\mathbf{X} \in \mathbb{R}^{N \times Q}$ of the high dimensional dataset $\mathbf{Y} \in \mathbb{R}^{N \times D}$, where in general $Q \ll D$. Usually Q is chosen to be $Q = 2$, to be able to visualize the newly found representation as a scatter plot. As mentioned above, we assume the data points which are close in this representation to be close in pseudo time ordering.
- Second, usually the experimenter supplies a known starting cell, from which we want the pseudo time ordering to originate. This cell can be chosen from the lower dimensional representation. One way, is to overlay the extraction time as colour labelling to the scatter plot and choose a starting cell by visual inspection.
- Third, we assign the ordering of the cells by following a smallest distances graph extracted from the lower dimensional representation. Usually this is done by using either a minimal spanning tree or k-nearest-neighbour graph. A minimal spanning tree is a graph, where each node has maximally three edges connected to it, spanning all nodes. The minimal spanning tree minimizes distances of edges across all nodes. A k-nearest-neighbour (KNN) graph is a graph in which all nodes are connected to the k nearest nodes to it. This can lead to more smooth connection of dense graphs and prevents unwanted structural effects, which can occur using a minimal spanning tree. To extract the pseudo time, we create the cumulative sum from the starting node across edges. We then assign this cumulative sum to each node, and thus, acquire an ordering according to the lower dimensional representation following the graph structure.
- Last, in many pseudo time ordering methods, there is a post processing step of cleaning up possible mistakes during pseudo time assignment. This could include smoothing of trajectories, branch detection, shortcut prevention and more.

In this thesis, we will compare to two methods: Monocle [88] and Wishbone [74]. Here, we will describe the way these methods extract the pseudo time in light of the above described general method of pseudo time extraction.

Monocle Monocle uses ICA (Sec. 4.1.3.1) as underlying dimensionality reduction technique. The ordering is extracted using a minimal spanning tree, after which some identification steps are undertaken to find the backbone (“main stem” for the differentiation) and branches of differentiation by a majority vote. In this thesis, we

will implement Monocle by applying the MST extraction technique directly on top of ICA.

Wishbone Wishbone uses diffusion maps [15] as underlying dimensionality reduction technique. It then extracts cell distances using multiple k-nearest-neighbour graphs. The ultimate ordering is a weighted average over those extracted graphs starting from so called waypoint cells, sampled across the trajectory. This decreases short-circuits, introduced by noise. After extraction of the ordering branch identification is done. Wishbone is meant for identifying only one branching point, i.e. is not meant to be applied for cell type differentiations into more than two cell types. In this thesis, we use the python implementation of wishbone [74] provided by the authors.

Topslam For comparison, we will summarize topslam here. The details hinted at here, will be elucidated in the following. Topslam uses Bayesian GPLVM 2.5 or MRD 2.6 as underlying dimensionality reduction technique. It then corrects distances along the extracted lower dimensional representation using the manifold embedding metric extracted from the probabilistic nature of the embedding. This decreases shortcuts and prevents erroneous branching along the trajectory. The non specificity of the dimensionality reduction technique and distance correction allows for any type of branching as shown by the simulations.

In the following, we will go into the details of Topslam and its intrinsic correction techniques, allowing for a more principled extraction of pseudo time according to probabilistic modelling. This will show the distance correction done by the manifold embedding metric. The general way of extracting pseudo time orderings opens itself to test different dimensionality reduction techniques and see how the preserve time orderings on simulated differentiation profiles. We will apply and compare different dimensionality reduction techniques to simulations and show their resemblance of extracted pseudo time to simulated time. We will then compare Monocle, Whishbone and Topslam using the same simulated datasets to assess their corrections and post processing.

4.1.2 Topslam

As explained in Section 2.5 and Section 2.6, we can extract a probabilistic lower dimensional representation for a high dimensional dataset. We call this lower dimensional representation the latent space, and the function mapping the latent space to the high dimensional space is called the manifold embedding. In this section, we describe a new algorithm combining Bayesian GPLVM with manifold distance

corrections to extract ordering information for data samples. In particular, we are interested in cell orderings by gene expression profiles.

The probabilistic nature of GPLVM, and by extension Bayesian GPLVM, allows us to extract a metric tensor for the latent space [87]. A metric tensor in this context is a high dimensional object, describing the topology (steepness, direction of slope) of the latent space at every point. We use this metric tensor to correct locally for pairwise distances in the latent space in a linear manner. In addition, the metric tensor can be visualized as landscape and gives useful insight about the latent embedding for cell orderings.

Correcting the distances for distortions in the latent embedding allows the extraction of ordering information more robust to outliers and noisy observations (additional to noise correction of the Bayesian GPLVM itself). To extract the ordering from the distances we employ a minimal spanning tree [44]. A minimal spanning tree is a graph spanning all nodes of a graph, minimizing the cumulative sum of edge weights. The nodes of the tree represent cells in the latent space, and edges represent corrected distances along the manifold. The cumulative sum of distances along the tree from a given starting node is then reported as ordering of the cells and can be used for downstream analysis as pseudo time ordering.

The method in this section will aid the extraction of manifold information, the reporting of underlying ordering and show simulation results, comparing GPLVM to other dimensionality reduction techniques. Additionally, we will compare this full extraction technique to others from the literature.

4.1.2.1 Extraction of Ordering Information using Topslam

Let $\mathbf{Y} \in \mathbb{R}^{N \times D}$ be the extracted gene expression matrix and $\mathbf{X} \in \mathbb{R}^{N \times Q}$ the learnt latent space by GPLVM, Bayesian GPLVM or MRD (Secs. 2.6, 2.5, 2.3). To extract ordering information from the samples, we need to make an assumption about the representation of cells in the latent space. We assume two cells to be in a similar differentiation state, if their gene expression pattern is similar. If two cells are close in the latent space, we can assign them close in the extracted ordering. One important assumption here is, that the dimensionality reduction technique preserves similarities, so that the ordering in original space will be represented in the latent space.

To extract an ordering, we make use of the distance matrix between all pairs (i, j) of cell latent positions \mathbf{X}_i and \mathbf{X}_j .

$$D_{ij}^{\text{latent}} = \sqrt{(\mathbf{X}_i - \mathbf{X}_j)^\top (\mathbf{X}_i - \mathbf{X}_j)} . \quad (4.1)$$

These distances are corrected using the Wishart embedding metric tensor as described by Tosi et al. [87]. By defining a Wishart distribution $\mathcal{W}(p, \Sigma, \boldsymbol{\mu}^\top \boldsymbol{\mu})$ on the Jacobian $\frac{\partial p(\mathbf{Y}|\mathbf{X})}{\partial \mathbf{X}} = \mathbf{J}$ of posterior distribution $p(\mathbf{Y}|\mathbf{X})$ of the GP w.r.t. \mathbf{X} we compute the expected manifold metric tensor

$$\begin{aligned} \mathbf{M} &:= \langle \mathbf{J}^\top \mathbf{J} \rangle = \langle \mathbf{J} \rangle^\top \langle \mathbf{J} \rangle + D\text{cov}(\mathbf{J}, \mathbf{J}) \\ &= \boldsymbol{\mu}^\top \boldsymbol{\mu} + D\Sigma \quad , \end{aligned}$$

of the latent embedding. Here, $\boldsymbol{\mu}$ is the mean of the gradient of the posterior distribution of the GP, and Σ is its covariance. This metric tensor describes distortions of the latent space functional embedding. Consider a sphere, where we can describe every point on its surface with two coordinates, but itself lives in a three dimensional space. To correct for the latent space distortions (curvature of the sphere), we would need to follow the geodesics of the latent space embedding (lines with shortest distance between points along the manifold/sphere), which is expensive to compute in the Bayesian GPLVM embedding. Therefore, we use a heuristic approach to correct for the embedding distortions. We correct locally for distortions of pairs of latent embeddings of cells (Fig. 4.10), such that local distances are corrected linearly

$$D_{ij}^{\text{corrected}} = \sqrt{(\mathbf{X}_i - \mathbf{X}_j)^\top \frac{\mathbf{M}_i + \mathbf{M}_j}{2} (\mathbf{X}_i - \mathbf{X}_j)} \quad .$$

We now use these corrected distances to construct a minimal spanning tree and compute distances along this tree, using for example Dijkstras [23] algorithm for shortest paths along a graph. With that, the distances computed follow approximately the geodesics and can be used to extract pseudo time orderings for the cells (Fig. 4.3a). We show the process in a simulation application to a manifold embedding in Section 4.1.3.

4.1.2.2 Waddington's Landscape

As described in Section 1.2.2, Waddington envisioned a (epigenetic) landscape (Fig. 1.3) that considers cell differentiation as a surface. Cells would follow the surfaces defined gravity wells to decide on the cell fate at each step of differentiation. We propose a *probabilistic representation of Waddington's landscape*, which is defined by the underlying manifold density, described by the magnification factor [10]

$$\mathbf{W}_i := \sqrt{\det \mathbf{M}_i} \quad \forall 1 \leq i \leq N \quad (4.2)$$

of the expected manifold metric tensor. The magnification factor describes the density of the space given by the manifold metric tensor at every point in the latent

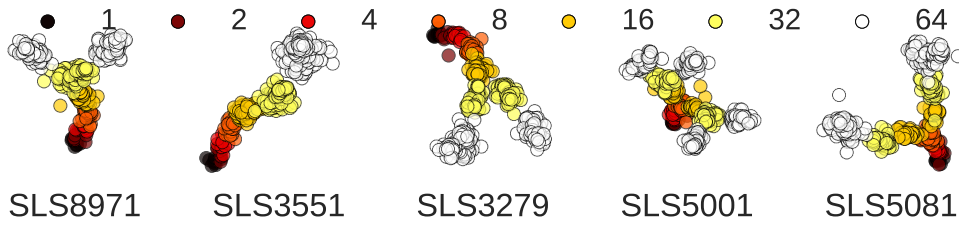


Figure 4.1: Differentiation simulation for cell progression. Each simulation has its unique seed. (Figure taken from the pre-print [100])

space. This representation assigns each position \mathbf{X}_i one density value, describing the density of the manifold at that position. This density is similar to Waddington’s original idea of a landscape, but has clear distinctions. Here, the landscape is a representation of the underlying manifold for the dimensionality reduction technique. It quantitates the accessibility of areas in the latent space. We do not acquire a mechanistic representation of the underlying processes of cells, but a probabilistic lower dimensional generative mapping to the gene expression patterns observed.

4.1.3 Comparison of Dimensionality Reduction Techniques using Differentiation Profile Simulations

To show the advantage of Topslam over other methodologies, we will compare other dimensionality reduction techniques to Topslam as an underlying algorithm for the pseudo time ordering extraction. The comparison is done on simulated latent spaces. We simulate cell differentiation profiles from 1 to 64 cell stages, shown in Figure 4.1. The simulation is done by creating a two dimensional tree splitting structure, of which the nodes are cell stages and the edges are differentiation fates. By simulating technical variance – progressively increasing noise variance of Gaussian distributed noise – we obtain a differentiation profile for cells in two dimensional space. We then use these simulated latent spaces \mathbf{X} to generate high dimensional gene expression measurements, by applying non-linear functions $\mathbf{Y} = f(\mathbf{X})$, where f are non-parametric Gaussian process samples, generated with an exponentiated quadratic covariance (Sec. 2.1.2.2).

Figure 4.2 visualizes the distribution of pairwise distances in gene expression space. This is to ensure, that the generated distribution over distances is representative for real world gene expression measurements. We will look at one of the experiments more closely in Section 4.2.

The code for the simulation can be found in the Python package developed

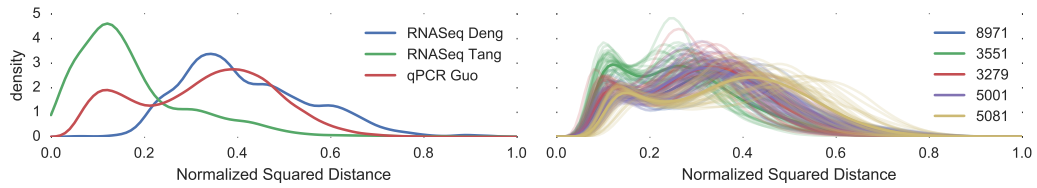
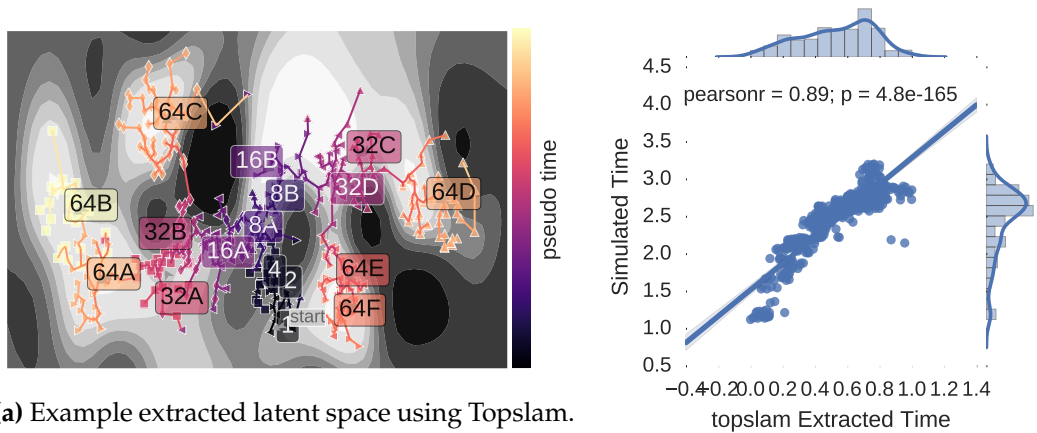


Figure 4.2: Left side shows the distance distribution for three real world data sets. Two RNASeq [22; 80] experiments and one qPCR Guo et al. [29] experiment. The right plot shows traces for all repetitions of data generation, as well as thick mean estimates overall. Only upper triangle distances are used for the kernel density estimates. All distances are normalized for comparability. (Taken with author approval from [100])



(a) Example extracted latent space using Topslam. The background shading shows the magnification factor for the manifold embedding. The lighter the shade, the easier it is to move around in it (i.e. distances get closer) and vice versa.

(b) Linear fit and Pearson correlation fit between extracted and simulated time.

Figure 4.3: Extracted landscape and correlation to simulated time for one simulated dataset.

as part of this thesis provided on Github <https://github.com/mzwiessle/topslam>.

4.1.3.1 Extraction and Comparison of Landscapes

To make the comparison, we first extract a latent space from 30 different generations of gene expression profiles of all 5 simulated differentiation profiles. We then apply different dimensionality reduction techniques to the gene expression profiles to extract latent spaces, from which we can extract pseudo time orderings using the MST approach (Sec. 4.1.2.1). An example extraction of the simulated differentiation profile “SLS5001” using Topslam is shown in Figure 4.3a, as well as the direct comparison of simulated time vs. the extracted pseudo time ordering (Fig. 4.3b).

We extract the ordering information using the MST approach, building a MST

	SLS8971	SLS3551	SLS3279	SLS5001	SLS5081
topslam	0.88±0.11	0.96±0.01	0.87±0.04	0.85±0.05	0.94±0.02
BGPLVM	0.83±0.19	0.83±0.06	0.84±0.07	0.79±0.08	0.92±0.05
Isomap	0.93±0.02	0.94±0.08	0.86±0.09	0.77±0.11	0.93±0.03
t-SNE	0.82±0.13	0.76±0.26	0.63±0.21	0.79±0.07	0.82±0.13
PCA	0.88±0.05	0.94±0.02	0.81±0.08	0.76±0.11	0.91±0.04
ICA	0.88±0.05	0.92±0.05	0.82±0.07	0.75±0.09	0.92±0.02
Spectral	0.68±0.09	0.77±0.09	0.71±0.11	0.57±0.09	0.81±0.06

Table 4.1: Comparison of topslam and Bayesian GPLVM (BGPLVM) to other standard dimensionality reduction techniques. Shown are mean and standard deviation of Pearson correlation coefficients between simulated and extracted times.

and follow its shortest paths to order the cells seen along it. Results for all dimensionality reduction techniques are shown in table 4.2.

To show the performance of Bayesian GPLVM from another point of view, we will compare to the extracted timelines of other dimensionality reduction techniques (Tab. 4.1). The compared dimensionality reduction techniques are

- Isomap: Extract quasi-isometric low dimensional embedding [82]. This method has shown to be non-unique and has stability issues, which requires careful normalization of the original data [5].
- t-SNE: extract t-distributed Stochastic Neighbourhood Embedding from data [90] for visualization. This method is meant for visualization and clustering and does not retain global distances of samples. This means it is not suitable for pseudo-time ordering of clusters, as the ordering could be distorted.
- Principal Component Analysis (PCA): Extract rotated linear bases to maximize variance in orthogonal components. By extracting components explaining a given amount of variance, we can decrease dimensionality [41].
- Independent Component Analysis (ICA): Extract the maximally non-Gaussian components under a linear factorization of the data [40].
- Spectral Embedding (Spectral): Extract lower dimensional representation using laplacian eigenmaps of connectivity graph of the samples [6].

This comparison shows the advantage of correcting for the latent embedding and non-linear dimensionality reduction in topslam. Notice the high performance of Isomap. This suggests a good point of study to use as fast dimensionality reduction technique underlying a pseudo-time extraction method.

4.1.4 Comparison of Pseudo Time Extraction Techniques using Differentiation Profile Simulations

We compare the three methods described in Section 4.1.1 to each other by using the simulated experiments from the previous Section 4.1.3. Table 4.2 shows the results from the comparison. Shown are the Pearson correlation coefficients between extracted and simulated timelines, averaged over the 30 simulation runs. To show the advantage of distance corrections using the manifold metric tensor of Topslam, we also show the pure Bayesian GPLVM extracted pseudo time orderings.

	SLS8971	SLS3551	SLS3279	SLS5001	SLS5081
Monocle	0.88 \pm 0.05	0.92 \pm 0.05	0.82 \pm 0.07	0.75 \pm 0.09	0.92 \pm 0.02
Wishbone	0.60 \pm 0.11	0.13 \pm 0.16	0.78 \pm 0.21	0.66 \pm 0.08	0.64 \pm 0.11
BGPLVM	0.83 \pm 0.19	0.83 \pm 0.06	0.84 \pm 0.07	0.79 \pm 0.08	0.92 \pm 0.05
Topslam	0.88 \pm 0.11	0.96 \pm 0.01	0.87 \pm 0.04	0.85 \pm 0.05	0.94 \pm 0.02

Table 4.2: Comparison Table for the tree extracted pseudo time simulations, showing results as mean and standard deviations for Pearson correlation coefficients ρ .

4.1.5 Probabilistic Waddington’s Landscape for other Dimensionality Reduction Techniques

The correction and extraction of the landscape can be seen as overlaying the probabilistic Waddington landscape on top of (Bayesian) GPLVM. This means, we take the lower dimensional representation of the underlying dimensionality reduction technique and overlay the landscape on top of it using the methodology described before.

Our method is not restricted to overlaying the landscape over (Bayesian) GPLVM. It can use other dimensionality reduction techniques as bases and overlay a probabilistic Waddington’s landscape over those. We first show that overlaying probabilistic Waddington’s landscapes over existing methods does not decrease performance. Second, we will show how to improve existing methods by learning a representation using the Bayesian GPLVM optimization on top of existing methods.

4.1.5.1 Overlaying a Landscape on top of other Techniques

To extract a probabilistic Waddington’s landscape for other methods, we will overlay the landscape on top of other dimensionality reduction techniques. This means, we take the probabilistic corrections to extend other non probabilistic techniques. In technical terms, we extract the landscape from a Bayesian GPLVM (Sec. 2.5) model, initialized and fixed to the underlying dimensionality reduction technique.

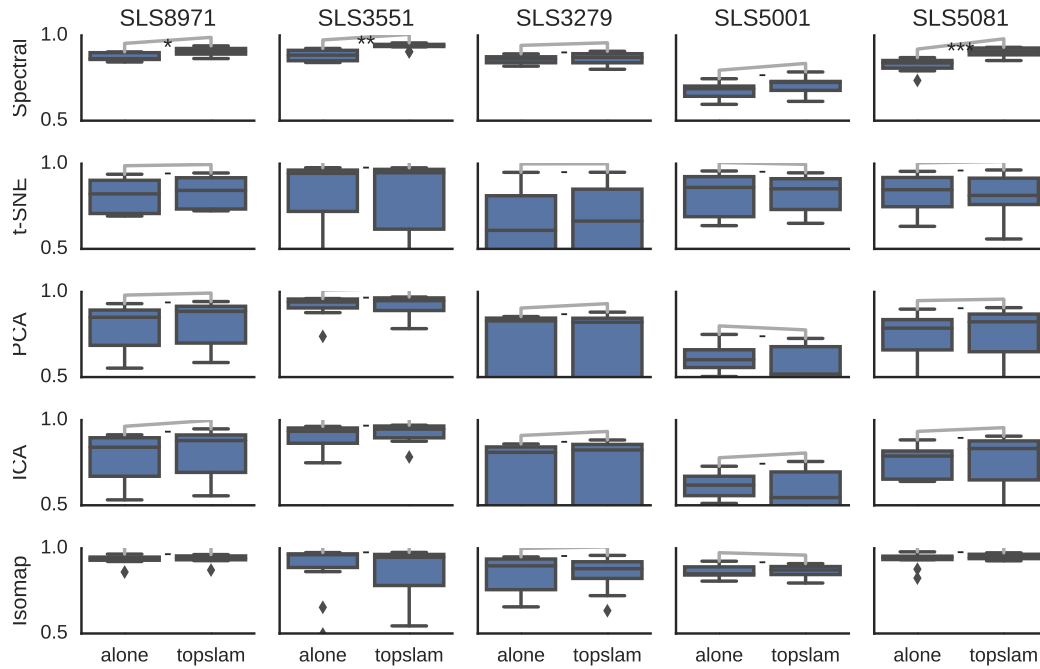


Figure 4.4: Comparison of correlation between pseudotimes extracted from other dimensionality reduction techniques landscapes and a correction of those techniques with our metric derived from the probabilistic Waddington's landscape. To the left of each subplot we see the pure landscape (*alone*) and to the right is our corrected landscape. (Taken with author approval from [100])

We compare the pseudo time orderings extracted using the overlay landscapes to the original landscapes extracted from other methods. In Figure 4.4 we show the correlation comparison. Note, that overlaying the probabilistic landscape never decreases performance and in some cases significantly increases performance of the respective methods.

4.1.5.2 Re-learn Bayesian GPLVM Landscape on top of other Techniques

Additionally to just fixing the underlying dimensionality reduction, we can also optimize the Bayesian GPLVM using the other method as initialization. The Bayesian GPLVM bound has local optima and a non global optimization technique, such as gradient descent can get stuck in those. Here, we want to show, that optimizing the bound will only increase performance and does not end up in worse places than the original method.

In Figure 4.5, we show that this approach results in significant improvements over just using the original method. Performance is increased significantly in terms of the mean and consistency is increased, as the variance over all results goes down.

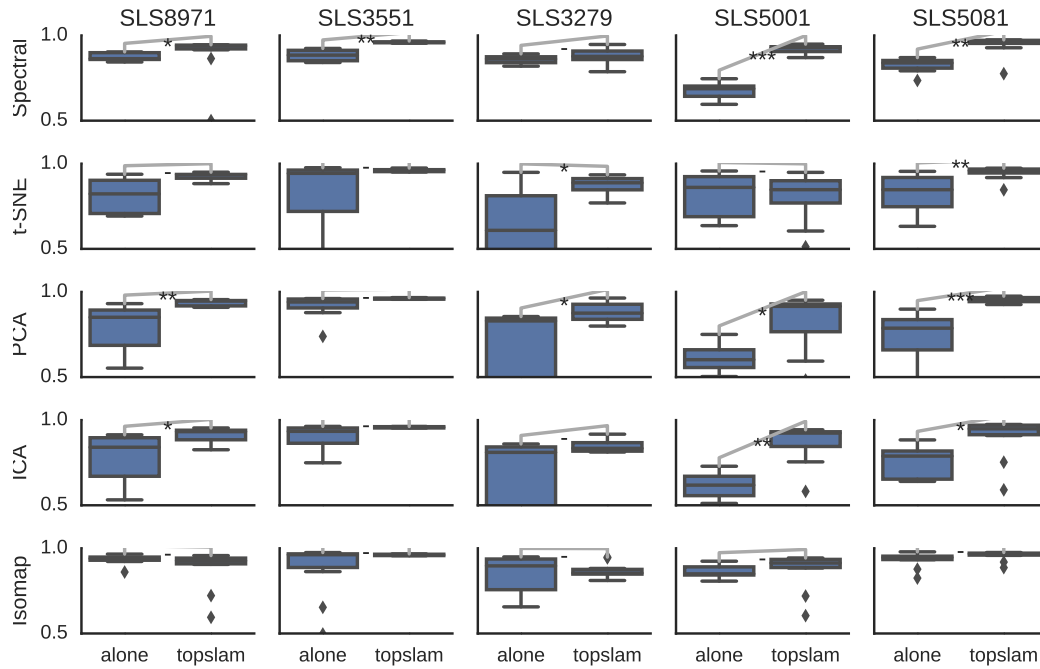


Figure 4.5: Compare correlation between pseudotimes extracted from other dimensionality reduction techniques landscapes and learning of probabilistic Waddington's landscape from those landscapes. This corrects the landscapes of the other techniques and uses them as a starting point (initialization). (Taken with author approval from [100])

Note that difficult latent spaces such as SLS5001 (compare Fig. 4.1) get improved for many techniques. And the variance of correlations for simple simulated latent spaces such as SLS3551 goes down.

Overall, using other methods as initialization for Bayesian GPLVM is viable in general and performance is improved overall using topslam.

4.1.6 Runtime & Complexity

Topslam is meant for complex differentiation profiles and noise disturbed datasets with confounding components of single cell gene expression experiments. In the application Chapter 4, we will show how to apply these techniques in practice. It can be used for simpler trajectories just as well, but the runtime requirements will be more expensive than for other pseudo time ordering extraction techniques found in the literature.

The runtime is a compound between Bayesian GPLVM and shortest distance extraction from the MST. Bayesian GPLVM has a runtime of $\mathcal{O}(MN^2)$ (Sec. 2.4), where N is the number of cells and M is an arbitrary approximation number, which is usually chosen to be in the tens. This dominates the upper bound for the com-

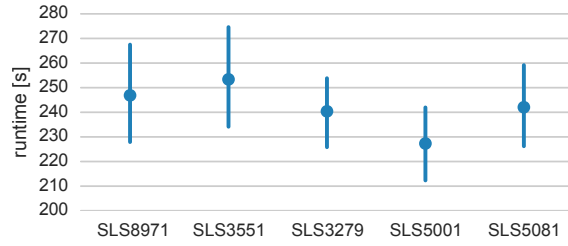


Figure 4.6: Runtimes of Topslam on all five simulated differentiation profiles (Fig. 4.1). Runtimes include model learning, distance correction and pseudo time ordering.

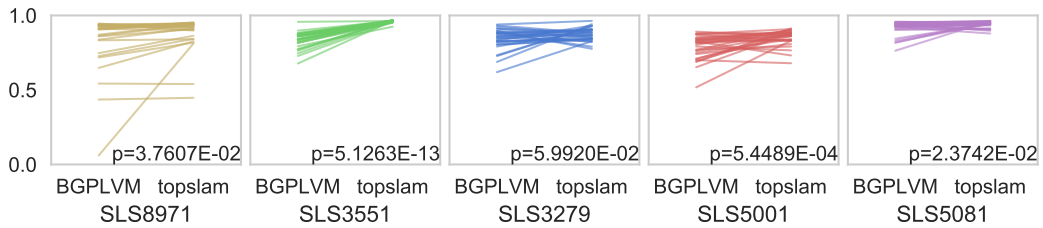


Figure 4.7: Bare Bayesian GPLVM vs Topslam. Visualization of the effect of taking the manifold embedding distortion into account. For each simulated latent space and repetition of simulation, we plot one line. The line connects the Pearson correlation coefficient between simulated and extracted time of bare Bayesian GPLVM on the left and Topslam on the right. p -values shown are for significance level between means differing.

plexity requirement of Dijkstra’s algorithm of $\mathcal{O}(N^2)$. In summary, the overall runtime for the algorithm is the complexity of Bayesian GPLVM.

We show the runtimes on the simulated datasets of approximately 500 cells and 48 genes in Figure 4.6. This includes all steps, from optimizing the Bayesian GPLVM, over distance corrections using the manifold metric tensor and extraction of pseudo time ordering along the extracted MST.

4.1.7 Bayesian GPLVM vs. Topslam

Topslam is an overlay on top of Bayesian GPLVM. To show the effect of taking the manifold topography into account, we plot the results of not correcting alongside the same result, with correction in Figure 4.7. The figure shows, that taking the manifold topography into account increases accuracy of pseudo time extraction significantly.

Using our method significantly improves the performance for recovery of simulated pseudotimes over existing techniques. This comes from the probabilistic modelling of the landscape and correction for topographical distortions. In summary, we are able to conclude that taking the landscapes topography into account is not detrimental to pseudo time ordering and at times improves recovery of in-

trinsic signals significantly.

4.2 Mouse Embryonic Development Landscape

Parts of this section’s methodology is in pre-print Zwiessele and Lawrence [100] and submitted to the BMC BioInformatics journal for publication.

Guo et al. [29] conducted a qPCR [28; 42] gene expression experiment over 437 cells and 48 genes of mouse embryonic stem cells. The experiment comprises up to 64 cell stage of early development in mice. It showed the differentiation of early stem cells, by means of manual application of linear dimensionality reduction. That is, the analysis is a series of PCA applications on sub selections of cells to elucidate their respective relation to each other.

Using Topslam (Sec. 4.1), we learn a landscape for the cells progression along time, capturing the differentiation process as a whole. The landscape provides the progression of time by following the valleys of the topography, depicted in Figure 4.9. The combination of non-linear dimensionality reduction using Bayesian GPLVM (Sec. 2.5) with the pseudo time ordering from Topslam, we are able to extract a more fine grained view on the differentiation of early embryonic stem cells.

4.2.1 Model Optimization using GPy

GPy is meant to be an easy to use tool to optimize Gaussian process based models. Additionally to optimizing and handling the model, we provide the data in the <https://github.com/sods/ods> repository under `pods.datasets.singlecell()`. With this, we can optimize the model in a few simple steps, including initialization of the model. Initially, we want the model to learn the latent space and not explain everything by noise. That is, the noise variance of the model is $\beta^{-1} = 1$ (Sec. 2.5). We set the noise variance to be equal to $\frac{1}{100}$ of the data variance. As we standardize the data gene wise, the noise variance will be set to 0.01. Additionally, GPy takes care of the initialization of the latent space $m.X$. It will set the mean of the latent space to PCA space and the variance equal to the fractions of eigenvalues for each dimension (compare Sec. 2.2). As a last step, GPy will set the lengthscale ℓ to the inverse of the of the eigenvalue fractions of PCA, to ensure the relevance of the dimensions is visible to the model initially.

```
1 import pods, GPy
2 data = pods.datasets.singlecell()
3 expr = data['Y'] # Extract expression matrix
4 labels = data['labels'].values # Extract cell stage labels (Guo et al.)
5
6 # Y is already standardized:
7 expr.mean(0)
```

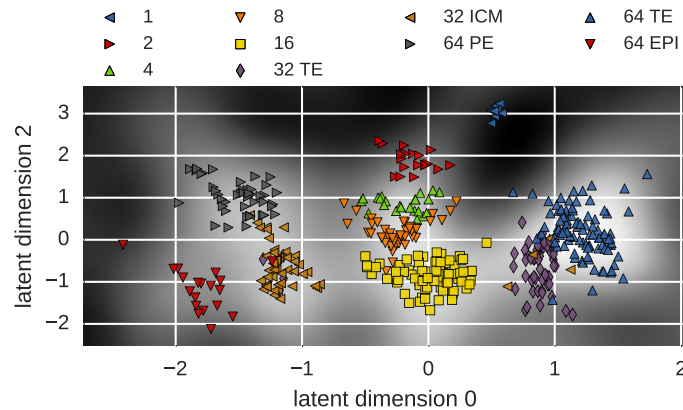



Figure 4.8: Plotting the magnification factor with GPy. Shown are the latent positions of cells as scatter plot and the magnification factor as grayscale from white (distances appear farther) and black (distances appear closer). This plot is generated by the simple call `m.plot_magnification(labels=labels)` to the GPy framework.

```

8 # Actb      2.089245e-08
9 # ...
10 # Tspan8   2.606407e-08
11 expr.var(0)
12 # Actb      1.002294
13 # ...
14 # Tspan8   1.002294
15
16 # Create BayesianGPLVM model:
17 from GPy.models import BayesianGPLVM
18 m = BayesianGPLVM(expr.values, input_dim=5, num_inducing=40)
19 m.likelihood.variance = 0.01 # Set the likelihood variance to 0.01
20 m.optimize(messages=1, max_iters=3000)

```

This gives us the optimized model in `m`. We can now plot the magnification factor on the latent space directly, using a colour gradient from black (distances are bigger than shown) to white (distances are closer than shown) by calling `m.plot_magnification(labels=labels)`. The resulting plot is shown in Figure 4.8.

Here, we have shown the optimization of a complex model such as Bayesian GPLVM using GPy. Note the simplicity of the optimization routine is a bit misleading. Here, the provided dataset is already cleaned and standardized. The initialization of the noise variance and the number of inducing inputs is already known and does not have to be found out anymore.

In the following section, we will extract the differential gene expression information along the time line and extract marker genes for the different cell stages.

4.2.2 Differential Expression in Time

The main advantage of a joint modelling of all cells, is that we have a continuous time line along all cells. This means, we can infer differing progression of gene expression along the time line. Here, the cells differentiate into three cell stages at the 64 cell stage:

- Trophoctoderm (TE).
- Epiblast (EPI).
- Primitive Endoderm (PE).

We use the same labelling of Guo et al. [29], which introduces some systematic errors. This labelling is produced by doing a staggered PCA approach. First extracting a global categorization and then fine tune in the categories. See Guo et al. [29] for a detailed description. With Topslam, we extract the latent positions of the cells using a non-linear embedding, so that differences between cells can be elucidated in more detail by the algorithm. In Figure 4.9, we show different dimensionality reduction techniques applied globally to the data. It is clearly visible, that the separation in other dimensionality reduction misses at least one essential component, respectively. t-SNE separates the components, but does not retain the ordering information, Isomap does not separate the components clearly and PCA does not separate either of the two on the full data (without the staggered approach). The figure also shows the probabilistic manifold embedding (magnification factor of the expected manifold metric tensor, Sec. 4.1.2.2). Here, we can see, that the distances get corrected, so that no shortcuts will be possible between say for example the 8 cell stage and the 64 PE cell stage. See the correction of the distances elucidated in Figure 4.10.

The pseudo time line allows us now to do inference on the differentiation process over time. As described in Section 1.2.2, we are interested in what makes the cells differentiate. The genes involved in differentiation along time will be differentially expressed between states. First, we need to identify the different differentiation branches along time to the terminal cell stages. This is done by making use of the tree structure underlying the pseudo time extraction. We can extract all edges and cells along the tree leading directly to a randomly selected terminal cell for each cell type (Fig. 4.11a). We can now perform differential gene expression detection in time series experiments [77]. This is a Gaussian process (Sec. 2.1) based differential gene expression algorithm. It compares two hypotheses by a Bayes factor to each other. The first hypothesis is that the compared timelines are shared, that is, they

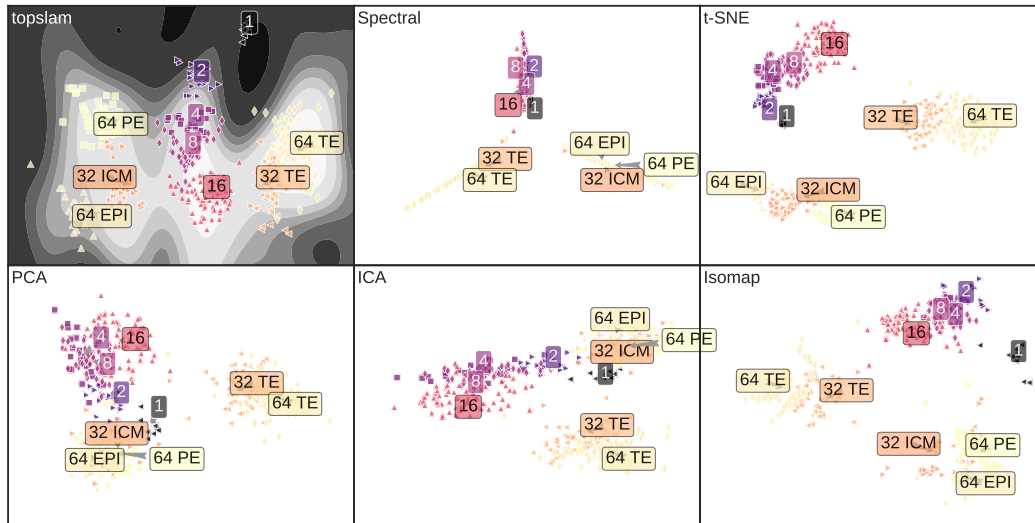


Figure 4.9: Comparison of dimensionality reduction techniques on Guo et al. [29] dataset. Topslam includes the extracted probabilistic Waddington landscape representation as contour shapes. The lighter the background colour, the easier it is to move (the closer the points to each other).

	0	1	2	3	4	5	6	7	8
TE EPI	Id2	Fgf4	Bmp4	Pecam1	Sox2	Dppa1	Fn1	Klf4	Fgfr2
PE EPI	Fgf4	Runx1	Fgfr2	Gata6	Pdgfra	Klf2	Bmp4	Gata4	Nanog
TE PE	Pdgfra	Id2	Gata4	Dppa1	Tspan8	Atp12a	Pecam1	Fn1	Creb312

Table 4.3: Differential gene expression between terminal cell stages of mouse embryonic stem cells. Shown are the top 8 differentially expressed genes between stages. If one stage is differentially expressed to the others, it is highly likely to be a marker for this particular stage. See e.g. *Id2* for trophectoderm (TE), which is a known marker [29].

come from one shared function. The second hypothesis is, that the compared timelines are independent and come from two independent functions. The Bayes factor is used as a scoring directly proportional to significance of differential expression.

Taking the differential expression scores between each stage and sorting them descending by their Bayes score will show marker genes for the differentiation between stages (Tab. 4.3). If a gene is differentially expressed from one stage to both of the others, it is a marker for this particular stage (see e.g. for TE *Id2*, *Tspan8*). The differentiation takes place over the timeline and there may be differing times at which differential expression kicks in. Having the time series as differential expressed marker genes, we can plot the exact time line of when genes get differentially expressed along pseudo time (Fig. 4.11b).

Using the probabilistic interpretation of Waddington’s landscape as a correction for the embedding and extraction techniques, we can extract pseudo time in-

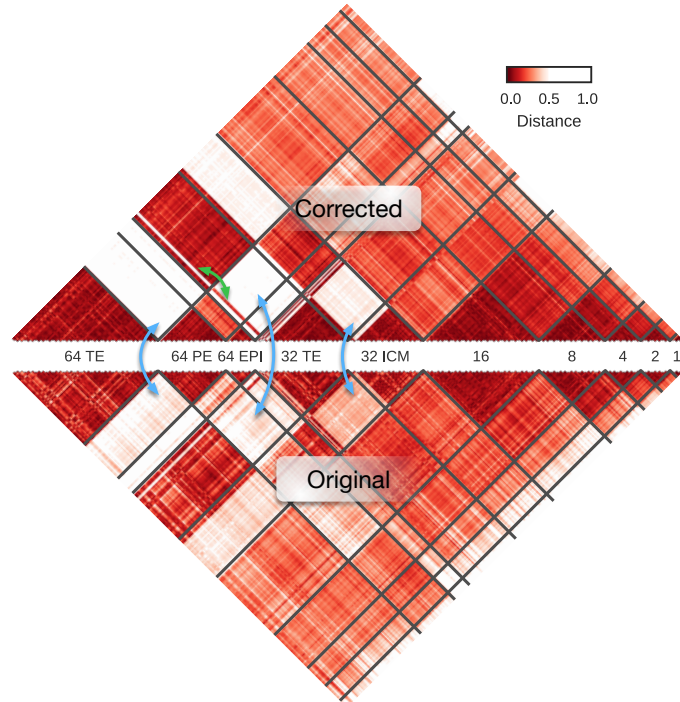


Figure 4.10: Distance corrections from manifold embedding. Some regions are highlighted with red arrows to see the correction of distances. Note the clearly visible miss classifications of cells from the label extraction method by Guo et al. [29] (green arrow).

formation more clearly and without additional information to ensure the time line extracted corresponds to the cell stages as seen in Guo et al. [29].

4.3 T Helper Cell Differentiation

This section will show, that the combination and application of tools based on a framework gives rise to complex analyses in single cell gene expression experiments. We will show the principled cooperation of MRD (Sec. 2.6) with Topslam. Importantly, we show the incorporation of prior knowledge into single cell gene expression experiments. In single cell experiments the element of unwanted variation in measurements becomes highly significant and has to be taken into account [12; 24].

We have a single cell gene expression measurement of differentiation of naive T cells to Th1 and Th2 differentiation cell states¹. T cells are the precursor cells to the immune system T helper cells. When the T cell receptor get presented antigen

¹The dataset was kindly provided by Aleksandra Kolodziejczyk and Sarah Teichmann for use in this thesis. The corresponding publication is pending.

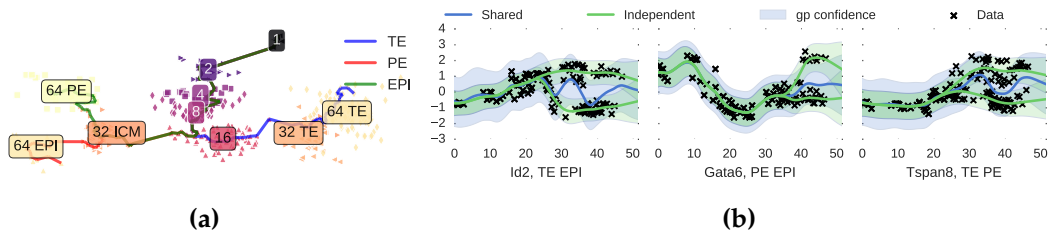


Figure 4.11: (a): Differentiation process along the time graph (endpoints randomly chosen within respective cell type). These differentiation paths are used for differential gene expression for marker gene detection. **(b):** Example time progression of marker genes. In green individual fits of two GPs, sharing one prior. In blue the shared fit of one GP to both stages. Differential expression is decided on which of those two models (green or blue) fits the data more likely. E.g. *Gata6* is a known marker for TE, compare Table 4.3.

from macrophages, they start to differentiate toward the T helper stages [7]. In this experiment we focus on Th1 and Th2 differentiation.

First, we show the data cleanup and subset selection. Second, we apply MRD for jointly modelling signal and confounding factors. Third, we will show the application of Topslam to the data and how to extract a pseudo time ordering. Finally, we will apply a mixture of Gaussian processes to find the split in the T helper cells and assign differentially expressed genes between T helper end stages.

4.3.1 Data Description and Cleanup

The dataset for this section was generated by Aleksandra Kolodziejczyk and a collaboration publication of partial results of the presented results is under consideration. The results shown in this section are shown with agreement by Aleksandra Kolodziejczyk by personal correspondence.

T cells were collected from two mouse strains, B6CAST and CASTB6. We collected 783 cells with 38,561 measured transcripts (genes). Whole transcriptome analysis was conducted by RNAseq [28; 42] single cell gene expression measurement.

The experiment was conducted *in vitro*. T cell differentiation was induced by T cell receptor activation and differentiation was guided by addition of Th1 and Th2 cells, respectively. The differentiation was tracked by 5 extractions of cells over 72 hours, though the time labels were lost during gene expression identification in the lab. This is one reason why we need to employ pseudo time extraction, but also a pseudo time assignment can further enhance the time for each cell, as cells might develop at different paces.

A filtering step was done before analysis with MRD (Sec. 2.6). We filtered for *cd4+* cells, marking for T cells [7].

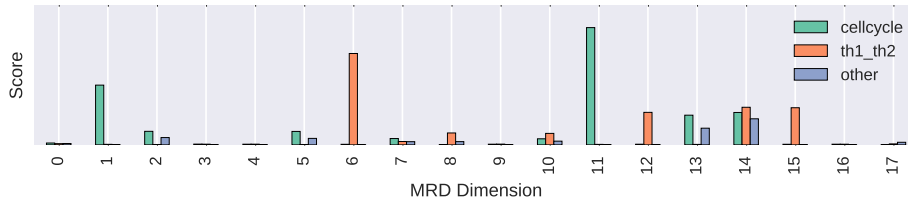


Figure 4.12: Relevance parameters for gene subsets of T cell differentiation.

Data was filtered before usage to limit noise variance in the following way. Only cells with at least 1,000 genes detected and only genes with at least 3 cells detected are kept.

At hour 72, batch gene expression measurements were conducted for subset selection. We computed differential gene expression between Th1 and Th2 cells in batch. This differential gene expression did not elude significant results after correction for multiple hypothesis testing. Therefore, we use the p-values of this experiment as guidance for subset selection for the MRD application (Sec. 4.3.2).

Additionally, a gene set associated with cell cycle activity was collected from Macosko et al. [57] for mouse.

4.3.2 MRD Application

We apply MRD (Sec. 2.6) to the data after normalization. The different subsets for MRD were selected to include cell cycle activity associated genes (subset taken from Macosko et al. [57]), genes differentially expressed between Th1 and Th2 stages in bulk data, and all other genes. This allows us to clearly identify each dimension of the lower dimensional representation. We can see the learnt ARD (Sec. 2.1.5) parameters in Figure 4.12. Each dimension of the learnt landscape (x -axis) has one score assigned, for each subset of genes. We can now deduce what information the dimensions hold for further analysis. First, we can see, that several dimensions are completely “switched off”, such as 0, 3 or 17. Those dimensions can be disregarded. We can also see, that the cell cycle genes subset has the lone influence on dimensions 1 and 11. This makes those two dimensions confounding variation, as we are not interested in cell cycle activity. We are interested in the differentiation between Th1 and Th2 cell stages, which is dominantly in dimension 6. So, dimension 6 is most likely the splitting dimension. Finally, dimension 14 is shared across all gene sets and is therefore most likely the progression in time. All cells are differentiating in the experiment and have to have a natural progression, which should be reflected in all gene patterns.

Having identified the relevant dimensions for further analysis, we can see if

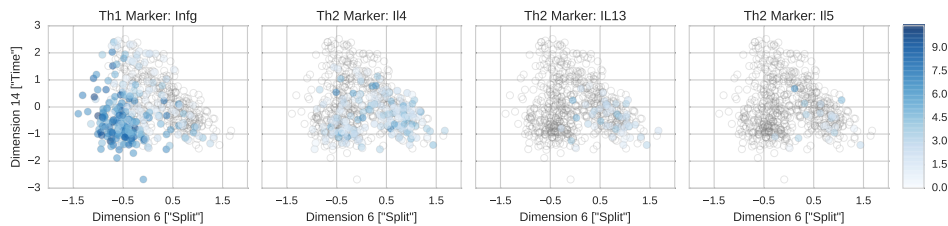


Figure 4.13: Marker gene progression in extracted landscape for T helper progression. Empty black circles are genes with a gene expression of 0 and likely missing.

they actually represent what we expect. We plot marker gene progression for known Th1 and Th2 markers [28; 42] in Figure 4.13. Note the low signal in Th2 marker genes. In the figure we can conclude, that the splitting dimension correlates with T helper differentiation in the following way. The left hand side (towards negative values) of the dimension encodes the Th1 cell state and the right hand side (towards positive values) represents the Th2 cell state. In combination with the time dimension, we can see the progression of cells.

4.3.3 Pseudo Time Ordering

We have identified the potential T helper splitting and time progression dimension. We can now extract the Topslam (Sec. 4.1) pseudo time ordering to extract a more accurate picture of the progression of cells. The amount of cells in this dataset gives a nice twist to the extraction. If we would use a minimal spanning tree to extract the ordering, we would add artificial branches into the picture, as the progression is not as clear cut as for example in the Guo et al. [29] case (Sec. 4.2). We will use a k-nearest-neighbour graph approach to extract the time progression information, replacing the minimal spanning tree, as described in 4.1.5. A k-nearest-neighbour graph is a graph constructed by connecting the k nearest nodes of a graph with edges, for all nodes. We choose a $k = 5$ nearest neighbour graph to extract the pseudo time ordering. See Figure 4.14 for a comparison between the two extraction techniques. The k nearest neighbour graph technique yields a continuous progression over time, whereas the minimal spanning tree introduces artificial structure into the timeline.

4.3.4 Split Detection

To unravel the split between Th1 and Th2, we employ a similar approach as shown in Lönnberg et al. [54]. They extract a split using a mixture model of Gaussian processes called overlapping mixture of Gaussian processes (OMGP, [31; 50]) to assign cells to the respective differentiation time lines. It is a variational approximation to

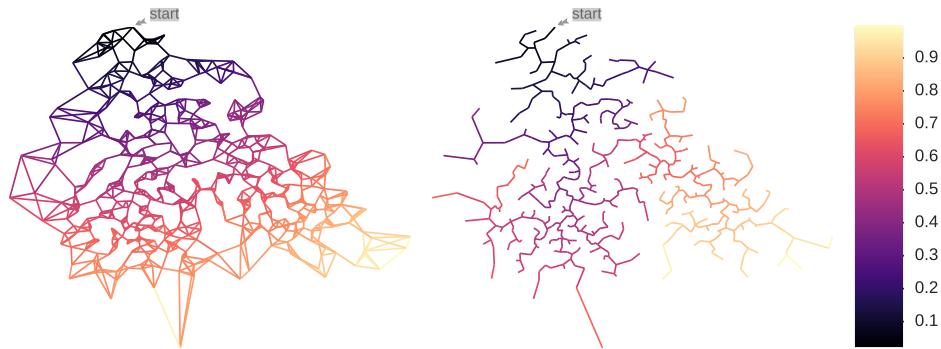


Figure 4.14: Pseudo time ordering extraction for T helper differentiation progression. On the left, we see the extraction using the $k = 5$ nearest neighbour graph and on the right using a minimal spanning tree. The colouring represents the pseudo time ordering assigned to the graph.

the weighted sum of Gaussian processes. A Dirichlet process prior on the weights ensures that the weights correspond to an assignment probability for each of the overlapping Gaussian processes [11; 45; 81].

We use an implementation of OMGP in the python package GPclust [32]. The python package is based on GPy (Sec. 3) and extends the functionality of GPy to mixtures of Gaussians, OMGPs and mixtures of hierarchical Gaussian processes [33; 35]. Thus, this section shows the usage of GPy beyond the scope of this thesis and how an implementation focused on extendability and ease of use can help improve the collaboration and usage of a package. The nature of GPy allowed the respective researchers used in this section to focus on their implementation of their part, while paramz supplied the backend for the ease of use of the end user.

The OMGP model uses the inferred pseudo time ordering of cells as input and the extracted time and split dimension as two dimensional output. Two Gaussian processes encode either of the two differentiation states. The priors over the weights of the OMGP model assign each cell a probability to come from one or the other overlapping Gaussian processes [54]. See Figure 4.15 for a plot of the split and time dimensions identified earlier, showing the assignment of each cell to the respective Th1 and Th2 differentiation states. It also shows the two underlying Gaussian processes, which explain the differentiation along time.

4.3.5 Differential Gene Expression

OMGPs are variational approximations to Bayesian models, supplying a lower bound to the marginal likelihood of the underlying Dirichlet mixture model. We make use of that to identify differentially expressed genes between the two states,

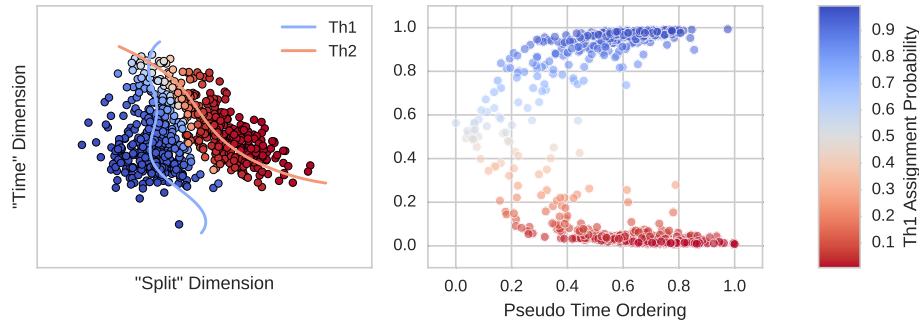


Figure 4.15: Overlapping mixture of Gaussians for T helper cell differentiation assignment. The left shows an overlay of the assignment probability and Gaussian processes for the split and time dimension identified using MRD. The right shows the extracted pseudo time ordering against the assignment probability of each cell to belong to Th1.

by comparing a null model of ambiguous assignments of cells (i.e. the assignment probability for each cell to either Th1 or Th2 is $p = 0.5$) to the observed model by a Bayes factor [9]. The Bayes factor provides us with a scoring for differential expression between Th1 and Th2 cell states for each gene. We make use of the GPfates package [54] based on GPy (Sec. 3) to extract the differential expression Bayes factor. To identify marker genes for T helper differentiation to Th1 and Th2 cell states, we plot the correlation of the (log) gene expression with the assignment probability for Th1 against the correlation of gene expression with pseudo time ordering. As significance score, we will use the log Bayes factor for differential expression. We plot the scatter plot for correlations in figure 4.16, highlighting the top ten marker genes and showing the potential of differential expression for all other genes. We can see, that known marker genes are identified as differentially expressed, such as *Infg*, *Gata3* and *Tanc* [7].

4.3.6 Summary

We were able to show, that the combination of multiple Gaussian process based techniques can alleviate the identification of marker genes in a single cell gene expression experiment. MRD helped us identify a lower dimensional representation most likely to contain time and split information in the data. Topslam extracted the according pseudo time ordering of the cells according to the underlying topography of the MRD model. OMGP was then able to assign each cell a probability to be Th1 or Th2 differentiated, or still in the progress of differentiation. As a last step, we showed, that a Bayes factor between an ambiguous and the observed OMGP model can identify differentially expressed genes in the data, helping to identify marker genes in T helper cell differentiation.

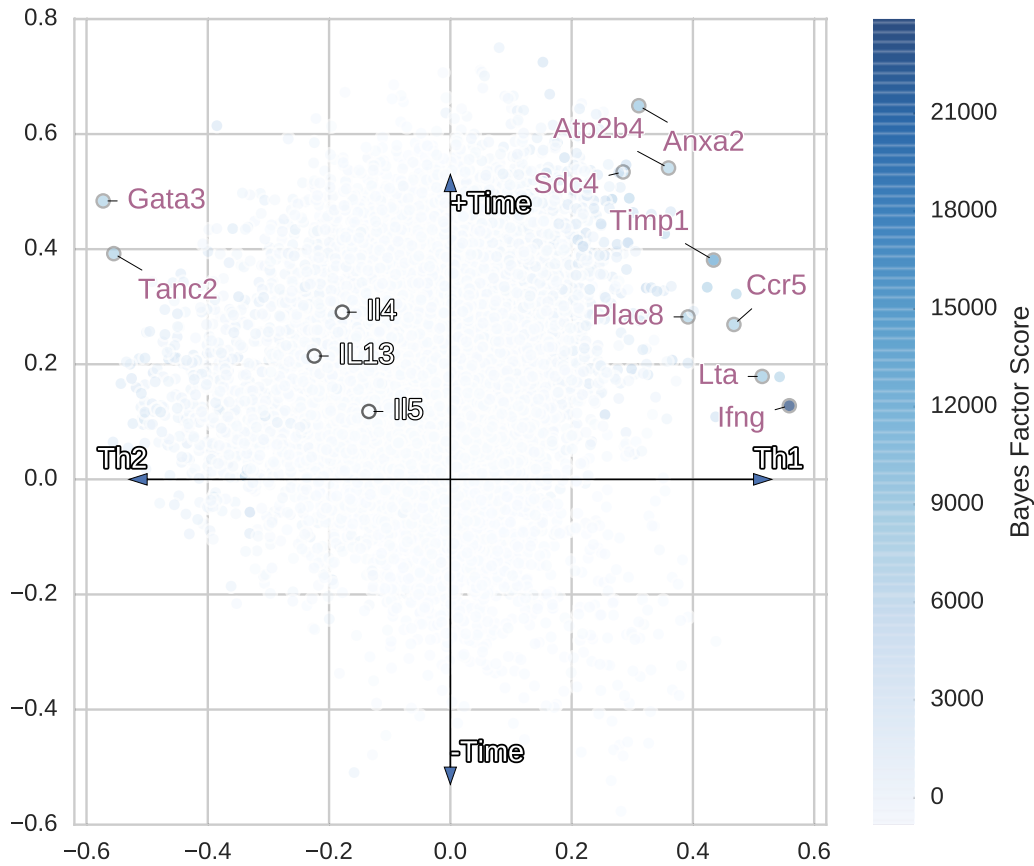


Figure 4.16: Differential expression between Th1 and Th2 cell states for each gene. X-axis is correlation with Th1 assignment probability and y-axis is correlation with pseudo time ordering. Colouring is log Bayes factor for differential detection as identified by OMGP.

In summary, we were able to develop a detailed picture of the T helper transcriptome along differentiation from a single cell gene expression experiment, without having to know the extraction times of the cells. This was done by employing packages based on GPy as a framework for Gaussian process based applications.

Chapter 5

Discussion and Conclusions

In this thesis, we have looked at the current issues in machine learning research involving Gaussian processes. In particular, we revealed new problems arising for the distribution of supplementary code. Most solutions to the problems are already known, and have to be taken into consideration by the research community around machine learning. We use code distribution platforms to make code publicly available and automated testing to make sure the codebase is in a working state at all times (Sec. 1.4). We have introduced and explained the philosophy behind GPy in making complex machine learning algorithms available to domain experts of different fields of expertise. While making algorithms available, we also want the codebase to be expandable by fellow machine learners, so that the framework can be constantly developed and extended (Sec. 3). At the time of writing, there is an active community of 38 contributors (<https://github.com/SheffieldMML/GPy/graphs/contributors>) on Github, contributing through 130 closed pull requests to GPy. GPy has been starred by over 500 data scientists, and forked 187 times, indicating an active community contributing to this framework for making complex algorithms accessible.

5.1 Packages Based on GPy

The problem of bringing machine learning to other domains is by no means solved, but we have made a big leap towards the ultimate goal of popularizing machine learning and statistics usage in the non-machine learning community. There are several packages relying on GPy as an underlying framework to perform other tasks relying on Gaussian processes. Packages include

- GPclust (Sec. 4.3.4), a toolbox for mixtures of Gaussian processes.
- GPflow (Sec. 3.10.1), a toolbox based on automated differentiation implementing Gaussian process based tools.

- GPyOpt [4], a Bayesian optimization toolbox for global optimization of parameters.
- GPy and GPyOpt are used in the NASA for their “Statistical Emulator for Expensive Classification Simulators” [71].
- GPfates [54], a toolbox for computing Bayes factors between ambiguous and observed OMGP models, as described in Section 4.3.5.

This shows that GPy is being actively used by fellow machine learners and other domains alike.

First, we thought of trying to fix the problems of another domain through machine learning. We soon came to grasp, that what we really should be doing is to bring machine learning to the domain. We think GPy is a good step towards this goal and shows that others have faced similar issues and challenges. GPy is also in the 97th impact percentile of research software packages, listed on `depsy.org` [14]. This is based on relative downloads, software re-use, and citation of the software.

This shows, that laying the foundation with an extendable and reusable piece of research software, we have paved the path for fellow researchers to focus on their research. It not only helps the fellow researchers to implement stable versions of their work, but also helps the user by consistency as they recognise the usage of the package. Paramz is to GPy what GPy is to the above packages.

5.2 "Bridging the Gap"

A challenge in machine learning is bridging the gap between methodologies and application domains. Machine learning algorithms are flexible tools capable of fitting all kinds of data. This means that machine learners work together with researchers of other fields to apply and analyse their data. This is the challenge of data science. In many cases, data to be analyzed is bound to the institute of question and cannot be shared with the machine learner off campus. Machine learning research needs to enable the institutes’ researchers to apply the machine learning techniques in an easy to use manner. Fully understanding the algorithm (with all its details) might be out of the scope of the data collecting colleague.

The ideas in this thesis are driven by personal experience. During secondments at the MPI for Psychology and Siemens (both) in Munich, I was mainly implementing algorithms for use of their internal researchers. This involved explaining how to use and what to expect from the resulting plots and analysis. I was, however, not able to use their data in the thesis, as it is bound to the institutes. This means

a lot of my research was about learning of how to implement complex algorithms to be used in an easy to use way. The implementation then, is to provide analysis tools for the domain experts to understand the results. The premise of this thesis is that I am not alone in having faced those challenges.

In the course of learning how to implement complex algorithms, we implemented the GPy (Sec. 3) package in the context of ease-of-use. The paramz package (Sec. 3.12) is a result of this progress, enabling parameterized model implementation. It aims at enabling both developers as well as users to design and apply complex models with parameters for gradient based optimization.

These experiences led to a more thorough understanding of the algorithms (Bayesian GPLVM, MRD, see methods 2 section) and how to apply them in different fields to gain insights into data. For example, applying MRD as a confounder correction technique, without having to make use of residual data sets. In single cell experiments the latent space dimensions play the important role for identifying the biological processes of interest. Incorporating subsets of genes of interest identifies the right latent space dimensions to look at and helps to identify the meaning that latent space dimensions carry (Sec. 4.3).

5.3 Understanding and Intuition

Applying Bayesian methods in machine learning has many advantages over other deterministic approaches, though predictive performance of, for example, neural networks is not to be undermined. Deterministic approaches lack the estimation of uncertainty in prediction, which Bayesian methods supply. In this thesis, we rely on the estimation of uncertainty, not only for the output predictions, but also for the input latent space. Additionally, we use the probabilistic nature of the mapping, that maps the inputs to the outputs to find the manifold density of the latent space (Sec. 4.1). Using this knowledge, we can correct for distortions in the latent space and improve upon ordering techniques by providing the correct distances to the extraction methods. Naturally, this part is to be done by the machine learning researcher.

It is difficult to communicate an understanding of an algorithm for practical use. In most cases, the practical use of machine learning comes down to knowing how to optimize (“fit”) the algorithm. This can be alleviated in providing the algorithm as a framework, such as GPy. There is still, however, the intuition missing. The algorithm can be implemented nicely and coherently, and still in practical use, problems arise.

5.3.1 Example: Initialization

It is difficult to communicate ideas such as initialization, parameter choice, prior knowledge and latent variables to fellow researchers who are not machine learners by training. It helps if colleagues are trained mathematicians or physicists, but intuitions for distinct machine learning methods can differ substantially. The intuitions about initialization are usually not acquired easily. GPy and paramz are great tools that make initialization and optimization strategy accessible, but still require the researcher to apply the intuition manually.

Some possible solutions to initialization of models could be either Bayesian global optimization [4; 61] or Markov Chain Monte–Carlo sampling [3; 26] for a few iterations. This could lead the parameters into the right direction for the gradient based optimization to take over from there. Bayesian optimization is not well adjusted to large numbers of parameters as of yet, sampling requires a lot of function executions to find good estimates for parameters. Heuristics can be applied to initialize parameters as well, but this can lead to difficulties explaining why model learning fails for some data. Current implementations usually require the intuition of the user to do the initialization and only do the rare heuristic choice for parameters. In GPy, we provide normalization of the data for the inference hidden from the user. The inference sees the normalized data, whereas all output functionality (such as plotting, prediction etc.) de-normalizes before reporting (`GPy.util.normalizer`).

It is unclear what the ultimate solution to this communication problem might be, automatization or training, but it is an open issue remaining to be solved.

5.3.2 Example: Optimization

For Bayesian methods, it often comes down to initialization of hyper parameters and correct fixing of parameters during optimization to restrain the dimensionality of the optimization problem and overcome local optima of the optimization landscape. A big advantage of Bayesian methods is the interpretation of hyper parameters and prior choice. Bayesian methods usually integrate over direct parametric (linear, polynomial, sigmoids etc.) solutions to find higher level prior choices. This leads to parameters and priors to encompass terms like “smoothness” versus “roughness”, “periodic” versus “chaotic”, “stationary” versus “shifting” and more. Here, we need to think about which prior choice to make and how initial parameters reflect our belief about the specific data at hand.

In machine learning fields there needs to be experts trained to understand the terms of the underlying technique. First comes the ability to apply the algorithm (such as the framework GPy), then there is the ability to understand outcomes and

difficulties (data scientist). In this thesis, we provided a framework for giving the data scientist the ability to apply the algorithm to their data. In parts, we also solved the issue of understanding, by running an actively used email list (<https://lists.shef.ac.uk/sympa/subscribe/gpy-users>) for questions regarding GPy applications and having an active community in the issues section of Github. It is still not fully solved, though, as in many cases, data scientists are not at liberty to discuss their data specific problems with the public and need training in understanding the algorithm from a different source. This is out of the scope of this thesis and must be addressed in future.

We have seen in both examples, that we can identify the problem in the communication of intuition. One big step towards this communication is to bring both sides on the same level, by providing the tools necessary to apply the machine learning tools. We have shown one way of solving this first step. Next steps involve communication of intuition through machine learners to domain expert data scientists, as well as automatization. We have showed the computer how to learn, now we have to show the human how to ask the right questions and interpret the results.

5.4 Conclusions

Overall, we are able to show, that supplying a codebase to other domains is a difficult task. Automation tools for testing and deploying code substantially decrease deployment turnarounds, but require a careful setup. Writing tests (Sec. 1.4) and splitting the algorithm for extendability and understandability greatly improve the ease-of-use for developers and users alike.

Keeping an active community through the issues section and email lists helps not only the domain experts to apply the algorithm to their data, but helps to improve the codebase itself. The open-source spirit helps to distribute tasks to other machine learning experts, who are willing to help to improve the codebase itself. We have had over 100 pull requests to GPy, suggesting and incorporating fixes, enhancements and bug fixes.

We are able to apply complex machine learning algorithms based on Gaussian processes in other domains by careful implementation and modularization of the codebase (Sec. 1.4). GPy provides the necessary underlying framework for several packages in machine learning, biology and astrology (Sec. 5.1). It is a step towards bringing Gaussian process models to the domain. Machine learning is not only to make the computer learn tasks, it is also to provide the (easy-to-use) tool for the domain expert to apply to their data.

Appendix A

Mathematical Details

A.1 GP Posterior Inference

In Section 2.1 we explain the Gaussian process inference method. Here we show the detailed derivation of the GP posterior. We marginalize out the latent function observations \mathbf{F} to create the posterior distribution $p(\mathbf{Y}|\mathbf{X})$ of the outputs \mathbf{X} given the inputs \mathbf{Y} :

$$\begin{aligned}
p(\mathbf{Y}|\mathbf{X}) &= \int (2\pi)^{-\frac{ND}{2}} |\sigma^2 \mathbf{I}|^{-\frac{D}{2}} \exp\left[-\frac{1}{2} \text{tr}((\mathbf{Y} - \mathbf{F})^\top \sigma^{-2} \mathbf{I} (\mathbf{Y} - \mathbf{F}))\right] \\
&\quad (2\pi)^{-\frac{ND}{2}} |\mathbf{K}|^{-\frac{D}{2}} \exp\left[-\frac{1}{2} \text{tr}(\mathbf{F}^\top \mathbf{K}^{-1} \mathbf{F})\right] d\mathbf{F} \\
&= (2\pi)^{ND} |\mathbf{K} \sigma^2 \mathbf{I}|^{-\frac{N}{2}} \exp\left[-\frac{1}{2} \text{tr}(\mathbf{Y}^\top \sigma^{-2} \mathbf{I} \mathbf{Y})\right] \\
&\quad \int \exp\left[\text{tr}(\mathbf{Y}^\top \sigma^{-2} \mathbf{I} \mathbf{F}) - \frac{1}{2} \text{tr}(\mathbf{F}^\top (\mathbf{K}^{-1} + \sigma^{-2})^{-1} \mathbf{F})\right] d\mathbf{F} \\
&= (2\pi)^{-ND} |\mathbf{K} \sigma^2 \mathbf{I}|^{-\frac{D}{2}} \exp\left[-\frac{1}{2} \text{tr}(\mathbf{Y}^\top \sigma^{-2} \mathbf{I} \mathbf{Y})\right] \\
&\quad (2\pi)^{\frac{ND}{2}} |\mathbf{K}^{-1} + \sigma^{-2} \mathbf{I}|^{-\frac{D}{2}} \exp\left[\frac{1}{2} \text{tr}(\mathbf{Y}^\top \sigma^{-2} \mathbf{I} (\mathbf{K}^{-1} + \sigma^{-2} \mathbf{I})^{-1} \sigma^{-2} \mathbf{Y})\right] \\
&= (2\pi)^{-\frac{ND}{2}} |\mathbf{K} \sigma^2 \mathbf{I} (\mathbf{K}^{-1} + \sigma^{-2} \mathbf{I})|^{-\frac{D}{2}} \\
&\quad \exp\left[-\frac{1}{2} \text{tr}(\mathbf{Y}^\top (\sigma^{-2} \mathbf{I} - \sigma^{-2} \mathbf{I} (\mathbf{K}^{-1} + \sigma^{-2} \mathbf{I})^{-1} \sigma^{-2} \mathbf{I}) \mathbf{Y})\right] \\
&= (2\pi)^{-\frac{ND}{2}} |\mathbf{K} + \sigma^2 \mathbf{I}|^{-\frac{D}{2}} \exp\left[-\frac{1}{2} \text{tr}(\mathbf{Y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{Y})\right] \\
&= \mathcal{N}(\mathbf{Y}|\mathbf{0}, \mathbf{K} + \sigma^2 \mathbf{I}) .
\end{aligned} \tag{A.1}$$

A.2 Latent Function Posterior

In Section 2.1.3, we show the posterior distribution $p(\mathbf{F}|\mathbf{Y}, \mathbf{X})$ of the latent functions \mathbf{F} given the observed data \mathbf{X}, \mathbf{Y} . The detailed derivation for this computation

is shown here:

$$\begin{aligned}
p(\mathbf{F}|\mathbf{Y}, \mathbf{X}) &= \frac{p(\mathbf{Y}|\mathbf{F})p(\mathbf{F}|\mathbf{X})}{p(\mathbf{Y}|\mathbf{X})} \\
&= \frac{\ell_1 \ell_2}{\ell_3} \exp\left(-\frac{1}{2} \text{tr}\left[\mathbf{F}^\top (\mathbf{K}^{-1} + \beta \mathbf{I}) \mathbf{F} - 2\mathbf{F}^\top \beta \mathbf{I} \mathbf{Y} + \mathbf{Y}^\top (\beta \mathbf{I} - (\mathbf{K} + \beta^{-1} \mathbf{I})^{-1}) \mathbf{Y}\right]\right) \\
&= \frac{\ell_1 \ell_2}{\ell_3} \exp\left(-\frac{1}{2} \text{tr}\left[\mathbf{F} - (\mathbf{K}^{-1} + \beta \mathbf{I})^{-1} \beta \mathbf{I} \mathbf{Y}\right]^\top (\mathbf{K}^{-1} + \beta \mathbf{I}) (\mathbf{F} - (\mathbf{K}^{-1} + \beta \mathbf{I})^{-1} \beta \mathbf{I} \mathbf{Y})\right) \\
&= \mathcal{N}(\mathbf{F} | (\mathbf{K}^{-1} + \beta \mathbf{I})^{-1} \beta \mathbf{I} \mathbf{Y}, (\mathbf{K}^{-1} + \beta \mathbf{I})^{-1}) \\
&= \mathcal{N}(\mathbf{F} | \mathbf{K}(\mathbf{K} + \beta^{-1} \mathbf{I})^{-1} \mathbf{Y}, \mathbf{K}(\mathbf{K} + \beta^{-1} \mathbf{I})^{-1} \beta^{-1} \mathbf{I}) ,
\end{aligned} \tag{A.2}$$

where ℓ_x are the respective constants and log determinants which resolve in a similar way as can be seen in (A.1).

A.3 Sparse GP: Conditional of Inducing Outputs given Observed Data

The derivation of sparse GPs (Sec. 2.4) in this thesis relies on the conditional distribution $\tilde{p}(\mathbf{U}|\mathbf{Y})$ of the inducing outputs \mathbf{U} given the observed outputs \mathbf{Y} . The

detailed derivation can be done as follows:

$$\begin{aligned}
\tilde{p}(\mathbf{U}|\mathbf{Y}) &= \frac{\exp(\mathcal{L}_1)p(\mathbf{U})}{\langle \exp(\mathcal{L}_1) \rangle_{p(\mathbf{U})}} \\
&= \frac{\mathcal{N}(\mathbf{Y}|\mathbf{M}_{\mathcal{L}_1}, \beta^{-1}\mathbf{I})\mathcal{N}(\mathbf{U}|\mathbf{0}, \mathbf{K}_{\mathbf{UU}}^{-1})}{\int \mathcal{N}(\mathbf{Y}|\mathbf{M}_{\mathcal{L}_1}, \beta^{-1}\mathbf{I})\mathcal{N}(\mathbf{U}|\mathbf{0}, \mathbf{K}_{\mathbf{UU}}^{-1}) d\mathbf{U}} \frac{\exp(\frac{1}{2}\beta \text{tr } \Lambda)}{\exp(\frac{1}{2}\beta \text{tr } \Lambda)} \\
&= \frac{c_1 c_2}{c_3} \exp\left\{-\frac{1}{2} \text{tr}\left((\mathbf{Y} - \mathbf{K}_{\mathbf{FU}}\mathbf{K}_{\mathbf{UU}}^{-1}\mathbf{U})^\top \beta \mathbf{I} (\mathbf{Y} - \mathbf{K}_{\mathbf{FU}}\mathbf{K}_{\mathbf{UU}}^{-1}\mathbf{U}) + \mathbf{U}^\top \mathbf{K}_{\mathbf{UU}}^{-1} \mathbf{U} \right. \right. \\
&\quad \left. \left. - \mathbf{Y}^\top (\mathbf{K}_{\mathbf{FU}}\mathbf{K}_{\mathbf{UU}}^{-1}\mathbf{K}_{\mathbf{UF}} + \beta^{-1}\mathbf{I})^{-1} \mathbf{Y}\right)\right\} \\
&= \frac{c_1 c_2}{c_3} \exp\left\{-\frac{1}{2} \text{tr}\left(\overbrace{(\mathbf{U} - (\mathbf{K}_{\mathbf{UU}}^{-1}\mathbf{K}_{\mathbf{UF}}\beta\mathbf{I}\mathbf{K}_{\mathbf{FU}}\mathbf{K}_{\mathbf{UU}}^{-1} + \mathbf{K}_{\mathbf{UU}}^{-1})^{-1}\mathbf{K}_{\mathbf{UU}}^{-1}\mathbf{K}_{\mathbf{UF}}\beta\mathbf{I}\mathbf{Y})}^{\mathbf{M}}\right)^\top \right. \\
&\quad \left. \overbrace{\mathbf{K}_{\mathbf{UU}}^{-1}\mathbf{K}_{\mathbf{UF}}\beta\mathbf{I}\mathbf{K}_{\mathbf{FU}}\mathbf{K}_{\mathbf{UU}}^{-1} + \mathbf{K}_{\mathbf{UU}}^{-1}}^{\mathbf{S}} (\mathbf{U} - \mathbf{M})\right) \\
&\quad + \frac{1}{2} \text{tr } \mathbf{Y}^\top \beta \mathbf{I} \mathbf{K}_{\mathbf{FU}} \mathbf{K}_{\mathbf{UU}}^{-1} (\mathbf{K}_{\mathbf{UU}}^{-1} \mathbf{K}_{\mathbf{FF}} \beta \mathbf{I} \mathbf{K}_{\mathbf{FU}} \mathbf{K}_{\mathbf{UU}}^{-1} + \mathbf{K}_{\mathbf{UU}}^{-1})^{-1} \mathbf{K}_{\mathbf{UU}}^{-1} \mathbf{K}_{\mathbf{UF}} \beta \mathbf{I} \mathbf{Y} \\
&\quad \left. - \frac{1}{2} \text{tr } \mathbf{Y} (\beta \mathbf{I} + \mathbf{K}_{\mathbf{FU}} \mathbf{K}_{\mathbf{UU}}^{-1} \mathbf{K}_{\mathbf{UF}})^{-1} \mathbf{Y}\right\} \\
&= \frac{c_1 c_2}{c_3} \exp\left\{\text{tr}(\mathbf{U} - \mathbf{M})^\top \mathbf{S} (\mathbf{U} - \mathbf{M}) \right. \\
&\quad \left. - \frac{1}{2} \text{tr } \mathbf{Y}^\top (\beta^{-1}\mathbf{I} + \mathbf{K}_{\mathbf{FU}}\mathbf{K}_{\mathbf{UU}}^{-1}\mathbf{K}_{\mathbf{UF}})^{-1} \mathbf{Y} + \frac{1}{2} \text{tr } \mathbf{Y}^\top (\beta^{-1}\mathbf{I} + \mathbf{K}_{\mathbf{FU}}\mathbf{K}_{\mathbf{UU}}^{-1}\mathbf{K}_{\mathbf{UF}})^{-1} \mathbf{Y}\right\} \\
&= \mathcal{N}(\mathbf{U}|\mathbf{M}_{\mathbf{U}}, \mathbf{S}_{\mathbf{U}}) , \text{ where} \\
\mathbf{S}_{\mathbf{U}} &= (\mathbf{K}_{\mathbf{UU}}^{-1}\mathbf{K}_{\mathbf{UF}}\beta\mathbf{I}\mathbf{K}_{\mathbf{FU}}\mathbf{K}_{\mathbf{UU}}^{-1} + \mathbf{K}_{\mathbf{UU}}^{-1})^{-1} \\
&= \mathbf{K}_{\mathbf{UU}}\mathbf{\Sigma}^{-1}\mathbf{K}_{\mathbf{UU}} , \text{ where } \mathbf{\Sigma} = \mathbf{K}_{\mathbf{UU}} + \beta\mathbf{K}_{\mathbf{UF}}\mathbf{K}_{\mathbf{FU}} \\
\mathbf{M}_{\mathbf{U}} &= \mathbf{S}_{\mathbf{U}}^{-1}\mathbf{K}_{\mathbf{UU}}^{-1}\mathbf{K}_{\mathbf{UF}}\beta\mathbf{I}\mathbf{Y} \\
&= \beta\mathbf{K}_{\mathbf{UU}}\mathbf{\Sigma}^{-1}\mathbf{K}_{\mathbf{UF}}\mathbf{Y} .
\end{aligned}$$

Appendix B

Software

B.1 Python

Python is an object oriented interpreted programming language [<http://python.org/>]. All scripts and packages mentioned in this thesis are written with Python and packages provided for it. All mathematics in this thesis make use of the NumPy package of SciPy (see below) for fast array-based computation. Plotting is done using Matplotlib and Seaborn (see below). Models are implemented using the Paramz [99] and GPy [27] packages.

B.1.1 SciPy

For scientific computations in this thesis, we make use of the SciPy package for Python. SciPy is a “open-source software [package] for mathematics, science, and engineering” [91]. It has underlying sub packages providing tools for different tasks in scientific computing. The packages are best described by their words, which are as follows (taken from [91]):

- *Python, a general purpose programming language. It is interpreted and dynamically typed and is very suited for interactive work and quick prototyping, while being powerful enough to write large applications in.*
- *NumPy, the fundamental package for numerical computation. It defines the numerical array and matrix types and basic operations on them.*
- *The SciPy library, a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics and much more.*
- *Matplotlib, a mature and popular plotting package, that provides publication-quality 2D plotting as well as rudimentary 3D plotting.*
- *pandas, providing high-performance, easy to use data structures.*

- *SymPy*, for symbolic mathematics and computer algebra. *IPython*, a rich interactive interface, letting you quickly process data and test ideas. The *IPython* notebook works in your web browser, allowing you to document your computation in an easily reproducible form.
- *nose*, a framework for testing Python code.

B.1.2 Seaborn

Seaborn [93] is an extension to Matplotlib (see above), providing an interface for drawing statistical graphs. It provides additional functionality for colour palettes and simple layout options (e.g. Figs. 4.4, 4.5, 4.7) when working with pandas (see above).

References

- [1] M. Abramowitz and I. A. Stegun. *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*, volume 55. Courier Corporation, 1964.
Cited on page 21.
- [2] S. Ahnert, T. Fink, and A. Zinovyev. How much non-coding DNA do eukaryotes require? *Journal of Theoretical Biology*, 252(4):587–592, 2008. Cited on page 6.
- [3] C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine learning*, 50(1):5–43, 2003. Cited on page 94.
- [4] T. G. authors. GPyOpt: A Bayesian Optimization framework in python. <http://github.com/SheffieldML/GPyOpt>, 2016. Cited on pages 92 and 94.
- [5] M. Balasubramanian and E. L. Schwartz. The Isomap Algorithm and Topological Stability. *Science*, 295(5552):pp. 7, 2002. doi: 10.1126/science.295.5552.7a. URL <http://science.sciencemag.org/content/295/5552/7>.
Cited on page 75.
- [6] M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003. Cited on page 75.
- [7] J. J. Bird, D. R. Brown, A. C. Mullen, N. H. Moskowitz, M. A. Mahowald, J. R. Sider, T. F. Gajewski, C.-R. Wang, and S. L. Reiner. Helper T cell differentiation is controlled by the cell cycle. *Immunity*, 9(2):229–237, 1998. Cited on pages 85 and 89.
- [8] C. M. Bishop. Bayesian PCA. In M. J. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems*, volume 11, pages 382–388. MIT Press, 1999. URL <http://papers.nips.cc/paper/1549-bayesian-pca.pdf>. Cited on page 30.
- [9] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
Cited on pages 9, 32, 40, 49, 50, 51, 60, and 89.

- [10] C. M. Bishop, M. Svensén, and C. K. Williams. GTM: The generative topographic mapping. *Neural computation*, 10(1):215–234, 1998. Cited on page 72.
- [11] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003. Cited on page 88.
- [12] F. Buettner, K. N. Natarajan, F. P. Casale, V. Proserpio, A. Scialdone, F. J. Theis, S. A. Teichmann, J. C. Marioni, and O. Stegle. Computational analysis of cell-to-cell heterogeneity in single-cell RNA-sequencing data reveals hidden subpopulations of cells. *Nature biotechnology*, 33(2):155–160, 2015. Cited on pages 42 and 84.
- [13] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995. Cited on page 56.
- [14] D. S. Chawla. The unsung heroes of scientific software. *Nature*, 529(7584):115–116, 2016. Cited on pages 1, 10, and 92.
- [15] R. R. Coifman, S. Lafon, A. B. Lee, M. Maggioni, B. Nadler, F. Warner, and S. W. Zucker. Geometric diffusions as a tool for harmonic analysis and structure definition of data: Diffusion maps. *Proceedings of the National Academy of Sciences of the United States of America*, 102(21):7426–7431, 2005. Cited on page 70.
- [16] A. Corvin, N. Craddock, and P. F. Sullivan. Genome-wide association studies: a primer. *Psychological Medicine*, 40(07):1063–1077, July 2010. URL <https://doi.org/10.1017/S0033291709991723>. Cited on page 30.
- [17] F. H. C. Crick, J. S. Griffith, and L. E. Orgel. Codes without commas. *Proceedings of the National Academy of Sciences*, 43(5):416–421, 1957. Cited on page 5.
- [18] G. M. Dall’Olio, J. Marino, M. Schubert, K. L. Keys, M. I. Stefan, C. S. Gillespie, P. Poulain, K. Shameer, R. Sugar, B. M. Invergo, L. J. Jensen, J. Bertranpetit, and H. Laayouni. Ten Simple Rules for Getting Help from Online Scientific Communities. *PLoS Comput Biol*, 7(9):1–3, 09 2011. URL <http://dx.doi.org/10.1371/journal.pcbi.1002202>. Cited on pages 11 and 13.
- [19] A. Damianou. *Deep Gaussian processes and variational propagation of uncertainty*. PhD thesis, University of Sheffield, 2015. Cited on page 37.
- [20] A. C. Damianou, C. H. Ek, M. K. Titsias, and N. D. Lawrence. Manifold Relevance Determination. In *Proceedings of the 29th International Conference on*

- Machine Learning*, volume 12, pages 145–152, Edinburgh, Scotland, GB, July 2012. URL <http://www.icml.cc/2012/papers/87.pdf>. Cited on page 42.
- [21] C. Darwin and A. R. Wallace. *Evolution by Natural Selection*. Cambridge University Press, 1958. Cited on page 9.
- [22] Q. Deng, D. Ramsköld, B. Reinius, and R. Sandberg. Single-cell RNA-seq reveals dynamic, random monoallelic gene expression in mammalian cells. *Science*, 343(6167):193–196, 2014. doi: 10.1126/science.1245316. URL <http://www.sciencemag.org/content/343/6167/193.abstract>. Cited on page 74.
- [23] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. Cited on page 72.
- [24] N. Fusi, O. Stegle, and N. D. Lawrence. Joint Modelling of Confounding Factors and Prominent Genetic Regulators Provides Increased Accuracy in Genetical Genomics Studies. *PLOS Computational Biology*, 8(1):1–9, Jan 2012. URL <http://dx.doi.org/10.1371/journal.pcbi.1002330>. Cited on pages 16, 30, and 84.
- [25] Y. Gal, M. van der Wilk, and C. E. Rasmussen. Distributed variational inference in sparse Gaussian process regression and latent variable models. In *Advances in Neural Information Processing Systems*, volume 27, pages 3257–3265. Curran Associates, Inc., 2014. URL <http://papers.nips.cc/paper/5593-distributed-variational-inference-in-sparse-gaussian-process-regression-and-latent-variable-models.pdf>. Cited on page 41.
- [26] W. R. Gilks, S. Richardson, and D. Spiegelhalter. *Markov chain Monte Carlo in practice*. CRC press, 1995. Cited on page 94.
- [27] GPy. GPy: A Gaussian process framework in python. <http://github.com/SheffieldML/GPy>, since 2012. Cited on pages 14, 59, and 99.
- [28] D. Grün and A. van Oudenaarden. Design and analysis of single-cell sequencing experiments. *Cell*, 163(4):799–810, 2015. Cited on pages 80, 85, and 87.
- [29] G. Guo, M. Huss, G. Q. Tong, C. Wang, L. Li Sun, N. D. Clarke, and P. Robson. Resolution of Cell Fate Decisions Revealed by Single-Cell Gene Expression Analysis from Zygote to Blastocyst. *Developmental cell*, 18(4):675–685, 2010. URL <http://dx.doi.org/10.1016/j.devcel.2010.02.012>. Cited on pages xiv, 74, 80, 82, 83, 84, and 87.

- [30] W. Hennig. *Genetik*. Springer Berlin, 1998. Cited on page 8.
- [31] J. Hensman, M. Rattray, and N. D. Lawrence. Fast Variational Inference in the Conjugate Exponential Family. In *Advances in Neural Information Processing Systems*, pages 2888–2896, 2012. URL <http://papers.nips.cc/paper/4766-fast-variational-inference-in-the-conjugate-exponential-family.pdf>.
Cited on page 87.
- [32] J. Hensman, V. Svensson, and M. Zwiessele. GPclust. <https://github.com/SheffieldML/GPclust/>, Dec 2012. Cited on page 88.
- [33] J. Hensman, N. D. Lawrence, and M. Rattray. Hierarchical Bayesian modelling of gene expression time series across irregularly sampled replicates and clusters. *BMC Bioinformatics*, 14(252):1–12, 2013. URL <http://dx.doi.org/10.1186/1471-2105-14-252>. Cited on page 88.
- [34] J. Hensman, M. Zwießeale, and N. D. Lawrence. Tilted Variational Bayes. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, volume 33, pages 356–364. Journal of Machine Learning Research, 2014. Cited on page 49.
- [35] J. Hensman, M. Rattray, and N. D. Lawrence. Fast Nonparametric Clustering of Structured Time-Series. *IEEE transactions on pattern analysis and machine intelligence*, 37(2):383–393, 2015. URL <https://doi.org/10.1109/TPAMI.2014.2318711>. Cited on page 88.
- [36] N. J. Higham. *Accuracy and stability of numerical algorithms*. Siam, 2002. Cited on page 53.
- [37] J. N. Hirschhorn and M. J. Daly. Genome-wide association studies for common diseases and complex traits. *Nature reviews. Genetics*, 6(2):95–108, Feb 2005. URL <http://dx.doi.org/10.1038/nrg1521>. Cited on page 30.
- [38] D. Howe, M. Costanzo, P. Fey, T. Gojobori, L. Hannick, W. Hide, D. P. Hill, R. Kania, M. Schaeffer, S. St Pierre, et al. Big data: The future of biocuration. *Nature*, 455(7209):47–50, 2008. Cited on page 2.
- [39] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007. Cited on page 61.
- [40] A. Hyvärinen, J. Karhunen, and E. Oja. *Independent Component Analysis*. John Wiley & Sons, 2004. Cited on page 75.

- [41] I. T. Jolliffe. *Principal Component Analysis*. Wiley Online Library, 2002. Cited on pages 28 and 75.
- [42] T. Kalisky and S. R. Quake. Single-cell genomics. *Nature methods*, 8(4): 311–314, 2011. URL <http://dx.doi.org/10.1038/nmeth0411-311>. Cited on pages 2, 80, 85, and 87.
- [43] R. Keeling, S. Piper, A. Bollenbacher, and J. Walker. Atmospheric Carbon Dioxide Record from Mauna Loa. *Trends: A Compendium of Data on Global Change. Carbon Dioxide Information Analysis Center, Oak Ridge National Laboratory, Oak Ridge, Tenn., U.S.A.*, 2009. doi: 10.3334/CDIAC/atg.035. Cited on pages 61 and 66.
- [44] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956. ISSN 00029939, 10886826. URL <http://www.jstor.org/stable/2033241>. Cited on page 71.
- [45] K. Kurihara, M. Welling, and Y. W. Teh. Collapsed Variational Dirichlet Process Mixture Models. In *Proceedings of the 20th international joint conference on Artificial intelligence*, volume 7, pages 2796–2801, 2007. Cited on page 88.
- [46] E. Lander, L. Linton, B. Birren, C. Nusbaum, M. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 02 2001. URL <http://dx.doi.org/10.1038/35057062>. Cited on page 5.
- [47] N. D. Lawrence. Gaussian Process Latent Variable Models for Visualisation of High Dimensional Data. In *Advances in Neural Information Processing Systems*, volume 16, pages 329–336. 2004. URL <http://papers.nips.cc/paper/2540-gaussian-process-latent-variable-models-for-visualisation-of-high-dimensional-data.pdf>. Cited on page 30.
- [48] N. D. Lawrence. Probabilistic Non-linear Principal Component Analysis with Gaussian Process Latent Variable Models. *Journal of Machine Learning Research*, 6(Nov):1783–1816, November 2005. Cited on pages 30, 32, and 49.
- [49] N. D. Lawrence and R. Urtasun. Non-linear matrix factorization with Gaussian processes. In *Proceedings of the 26th Annual International Conference on Machine Learning*, volume 26, pages 601–608, Montreal, June

2009. Omnipress. URL <http://www.machinelearning.org/archive/icml2009/papers/384.pdf>. Cited on page 41.
- [50] M. Lázaro-Gredilla, S. Van Vaerenbergh, and N. D. Lawrence. Overlapping Mixtures of Gaussian Processes for the data association problem. *Pattern Recognition*, 45(4):1386–1395, 2012. URL <http://dx.doi.org/10.1016/j.patcog.2011.10.004>. Cited on page 87.
- [51] J. T. Leek and J. D. Storey. Capturing Heterogeneity in Gene Expression Studies by Surrogate Variable Analysis. *PLOS Genetics*, 3(9):1–12, 09 2007. URL <http://dx.doi.org/10.1371/journal.pgen.0030161>. Cited on page 16.
- [52] J. T. Leek, R. B. Scharpf, H. C. Bravo, D. Simcha, B. Langmead, W. E. Johnson, D. Geman, K. Baggerly, and R. A. Irizarry. Tackling the widespread and critical impact of batch effects in high-throughput data. *Nature Reviews Genetics*, 11(10):733–739, 2010. Cited on page 2.
- [53] N. Loman and M. Watson. So you want to be a computational biologist? *Nature biotechnology*, 31(11):996–998, 11 2013. URL <http://dx.doi.org/10.1038/nbt.2740>. Cited on page 13.
- [54] T. Lönnberg, V. Svensson, K. R. James, D. Fernandez-Ruiz, I. Sebina, R. Montandon, M. S. F. Soon, L. G. Fogg, A. S. Nair, U. N. Liligeto, et al. Single-cell RNA-seq and computational analysis using temporal mixture modeling resolves TH1/TFH fate bifurcation in malaria. *Science Immunology*, 2(9):eaal2192, 2017. Cited on pages 87, 88, 89, and 92.
- [55] Y. Lu, Y. Zhou, W. Qu, M. Deng, and C. Zhang. A Lasso regression model for the construction of microRNA-target regulatory networks. *Bioinformatics*, 27(17):2406–2413, Sep 2011. URL <https://doi.org/10.1093/bioinformatics/btr410>. Cited on page 38.
- [56] D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003. Cited on page 9.
- [57] E. Z. Macosko, A. Basu, R. Satija, J. Nemeshe, K. Shekhar, M. Goldman, I. Tirosh, A. R. Bialas, N. Kamitaki, E. M. Martersteck, J. J. Trombetta, D. A. Weitz, J. R. Sanes, A. K. Shalek, A. Regev, and S. A. McCarroll. Highly Parallel Genome-wide Expression Profiling of Individual Cells Using Nanoliter Droplets. *Cell*, 161(5):1202–1214, 2015. URL <http://dx.doi.org/10.1016/j.cell.2015.05.002>. Cited on page 86.

- [58] V. Marx. Biology: The big challenges of big data. *Nature*, 498(7453):255–260, 06 2013. URL <http://dx.doi.org/10.1038/498255a>. Cited on page 2.
- [59] G. Mendel. Versuche über Pflanzen-Hybriden. In *Verhandlungen des naturforschenden Vereines in Brünn*, Abhandlungen, pages 3–182. Brünn : Im Verlage des Vereines, 1865. URL <http://www.mendelweb.org/Mendel.html>. Cited on pages 1 and 4.
- [60] T. Mercer, M. Dinger, and J. Mattick. Long non-coding RNAs: insights into functions. *Nature Reviews Genetics*, 10(3):155–159, 2009. Cited on page 6.
- [61] M. A. Osborne, R. Garnett, and S. J. Roberts. Gaussian processes for global optimization. In *3rd international conference on learning and intelligent optimization (LION3)*, 2009. URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.352.9682>. Cited on page 94.
- [62] 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, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. Cited on page 60.
- [63] J. M. Perkel. How bioinformatics tools are bringing genetic analysis to the masses. *Nature*, 543(7643):137–138, 2017. URL <http://www.nature.com/news/how-bioinformatics-tools-are-bringing-genetic-analysis-to-the-masses-1.21545>. Cited on page 12.
- [64] K. B. Petersen and M. S. Pedersen. The matrix cookbook. version: November 15, 2012, November 2012. URL http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/imm3274.pdf. Cited on pages 20, 24, and 25.
- [65] Plotly Technologies Inc. Collaborative data science, 2015. URL <https://plot.ly>. Cited on page 61.
- [66] A. L. Price, N. J. Patterson, R. M. Plenge, M. E. Weinblatt, N. A. Shadick, and D. Reich. Principal components analysis corrects for stratification in genome-wide association studies. *Nature Genetics*, 38(8):904–909, 08 2006. URL <http://dx.doi.org/10.1038/ng1847>. Cited on page 16.
- [67] A. Prlić and J. B. Procter. Ten Simple Rules for the Open Development of Scientific Software. *PLOS Computational Biology*, 8(12):e1002802, 12 2012. URL <http://dx.doi.org/10.1371/journal.pcbi.1002802>. Cited on page 11.

- [68] L. Rampasek and A. Goldenberg. TensorFlow: Biology’s Gateway to Deep Learning? *Cell Systems*, 2(1):12–14, 2016. ISSN 2405-4712. URL <http://dx.doi.org/10.1016/j.cels.2016.01.009>. Cited on page 60.
- [69] C. E. Rasmussen and Z. Ghahramani. Occam’s razor. In *Advances in Neural Information Processing Systems*, volume 13, pages 294–300. MIT Press, 2001. URL <http://mlg.eng.cam.ac.uk/zoubin/papers/occam.pdf>. Cited on page 27.
- [70] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006. URL <http://www.gaussianprocess.org/gpml/>. Cited on pages 9, 16, 18, 20, 23, 24, 25, 38, 50, 53, and 61.
- [71] J. Ross and J. A. Samareh. Statistical emulator for expensive classification simulators. <http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20160006302.pdf>, 2016. Cited on page 92.
- [72] S. Sagiroglu and D. Sinanc. Big data: A review. In *International Conference on Collaboration Technologies and Systems (CTS)*, pages 42–47. IEEE, May 2013. URL <https://doi.org/10.1109/CTS.2013.6567202>. Cited on page 2.
- [73] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977. URL <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC431765/>. Cited on page 5.
- [74] M. Setty, M. D. Tadmor, S. Reich-Zeliger, O. Angel, T. M. Salame, P. Kathail, K. Choi, S. Bendall, N. Friedman, and D. Pe’er. Wishbone identifies bifurcating developmental trajectories from single-cell data. *Nature biotechnology*, 34(6):637–645, 2016. URL <http://dx.doi.org/10.1038/nbt.3569>. Cited on pages 69 and 70.
- [75] W. Seyffert and R. Balling. *Lehrbuch der Genetik*. Fischer, 1998. Cited on page 8.
- [76] J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature Biotechnology*, 26(10):1135–1145, 10 2008. URL <http://dx.doi.org/10.1038/nbt1486>. Cited on page 1.
- [77] O. Stegle, K. J. Denby, E. J. Cooke, D. L. Wild, Z. Ghahramani, and K. M. Borgwardt. A robust Bayesian two-sample test for detecting intervals of differential gene expression in microarray time series. *Journal of Computational Biology*, 17(3):355–367, 2010. Cited on page 82.

- [78] O. Stegle, L. Parts, R. Durbin, and J. Winn. A Bayesian Framework to Account for Complex Non-Genetic Factors in Gene Expression Levels Greatly Increases Power in eQTL Studies. *PLOS Computational Biology*, 6(5):e1000770, May 2010. URL <http://dx.doi.org/10.1371/journal.pcbi.1000770>. Cited on page 30.
- [79] O. Stegle, S. A. Teichmann, and J. C. Marioni. Computational and analytical challenges in single-cell transcriptomics. *Nature Reviews Genetics*, 16(3):133–145, 03 2015. URL <http://dx.doi.org/10.1038/nrg3833>. Cited on page 2.
- [80] F. Tang, C. Barbacioru, Y. Wang, E. Nordman, C. Lee, N. Xu, X. Wang, J. Bodeau, B. B. Tuch, A. Siddiqui, et al. mRNA-Seq whole-transcriptome analysis of a single cell. *Nature Methods*, 6(5):377–382, 2009. URL <http://dx.doi.org/10.1038/nmeth.1315>. Cited on page 74.
- [81] Y. W. Teh, D. Newman, and M. Welling. A Collapsed Variational Bayesian Inference Algorithm for Latent Dirichlet Allocation. In *Advances in Neural Information Processing Systems*, volume 19, pages 1353–1360. MIT Press, 2007. URL <http://papers.nips.cc/paper/3113-a-collapsed-variational-bayesian-inference-algorithm-for-latent-dirichlet-allocation.pdf>. Cited on page 88.
- [82] J. B. Tenenbaum, V. De Silva, and J. C. Langford. A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, 290(5500):2319–2323, 2000. URL <http://science.sciencemag.org/content/290/5500/2319>. Cited on page 75.
- [83] M. E. Tipping. Sparse Kernel Principal Component Analysis. In *Advances in Neural Information Processing Systems*, volume 13, pages 633–639. MIT Press, 2001. URL <http://papers.nips.cc/paper/1791-sparse-kernel-principal-component-analysis.pdf>. Cited on pages 30 and 32.
- [84] M. E. Tipping and C. M. Bishop. Probabilistic Principal Component Analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622, 1999. URL <http://dx.doi.org/10.1111/1467-9868.00196>. Cited on page 30.
- [85] M. K. Titsias. Variational Learning of Inducing Variables in Sparse Gaussian Processes. *Artificial Intelligence and Statistics*, 5:567–574, April 2009. Cited on pages 33 and 34.

- [86] M. K. Titsias and N. D. Lawrence. Bayesian Gaussian Process Latent Variable Model. *Artificial Intelligence and Statistics*, 9:844–851, 2010. Cited on pages 37 and 49.
- [87] A. Tosi, S. Hauberg, A. Vellido, and N. D. Lawrence. Metrics for probabilistic geometries. In *Uncertainty in Artificial Intelligence*, volume 30, pages 800–808. AUAI Press (Association for Uncertainty in Artificial Intelligence), 2014. URL <http://auai.org/uai2014/proceedings/individuals/171.pdf>. Cited on pages 16, 71, and 72.
- [88] C. Trapnell, D. Cacchiarelli, J. Grimsby, P. Pokharel, S. Li, M. Morse, N. J. Lennon, K. J. Livak, T. S. Mikkelsen, and J. L. Rinn. The dynamics and regulators of cell fate decisions are revealed by pseudotemporal ordering of single cells. *Nature Biotechnology*, 32(4):381–386, 04 2014. URL <http://dx.doi.org/10.1038/nbt.2859>. Cited on page 69.
- [89] U.S. National Library of Medicine. What is DNA? <http://ghr.nlm.nih.gov/handbook/basics/dna>, September 2012. Cited on page 5.
- [90] L. van der Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008. URL <http://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf>. Cited on page 75.
- [91] P. Virtanen, R. Gommers, T. E. Oliphant, D. Cournapeau, E. Burovski, W. Weckesser, et al. SciPy: Open source scientific tools for Python, Sept. 2016. URL <https://doi.org/10.5281/zenodo.154391>. Cited on page 99.
- [92] C. H. Waddington. Principles of Development and Differentiation. *BioScience*, 16(11):821–822, November 1966. URL <https://doi.org/10.2307/1293653>. Cited on pages 7 and 8.
- [93] M. Waskom, O. Botvinnik, drewokane, P. Hobson, David, Y. Halchenko, S. Lukauskas, J. B. Cole, J. Warmenhoven, J. de Ruiter, S. Hoyer, J. Vanderplas, S. Villalba, G. Kunter, E. Quintero, M. Martin, A. Miles, K. Meyer, T. Augspurger, T. Yarkoni, P. Bachant, M. Williams, C. Evans, C. Fitzgerald, Brian, D. Wehner, G. Hitz, E. Ziegler, A. Qalieh, and A. Lee. seaborn: v0.7.1 (June 2016), June 2016. URL <https://doi.org/10.5281/zenodo.54844>. Cited on page 100.
- [94] J. D. Watson and F. H. C. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, 04 1953. URL <http://dx.doi.org/10.1038/171737a0>. Cited on page 4.

- [95] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, et al. Best Practices for Scientific Computing. *PLOS Biology*, 12(1):e1001745, 2014. URL <http://dx.doi.org/10.1371/journal.pbio.1001745>. Cited on pages 12 and 13.
- [96] L. Wodicka, H. Dong, M. Mittmann, M.-H. Ho, and D. J. Lockhart. Genome-wide expression monitoring in *Saccharomyces cerevisiae*. *Nature Biotechnology*, 15(13):1359–1367, 1997. URL <http://dx.doi.org/10.1038/nbt1297-1359>. Cited on page 30.
- [97] A. Zien. A primer on molecular biology. In B. Schölkopf, K. Tsuda, and J.-P. Vert, editors, *Kernel methods in computational biology*, chapter 1, pages 3–34. MIT press, 2004. Cited on pages 5 and 8.
- [98] M. Zwiessele. Probabilistic Modelling of Expression Variation in Modern eQTL Studies. Master Thesis Bioinformatics, Eberhard Karls Universität Tübingen, <https://github.com/mzwiessele/theses/raw/master/Master/Probabilistic%20Modelling%20of%20Expression%20Variation%20in%20Modern%20eQTL%20Studies.pdf>, October 2012. Cited on pages 4, 5, 6, 7, 28, 29, 30, and 39.
- [99] M. Zwiessele. Paramz. <https://github.com/sods/paramz/>, 2016. Cited on pages 58, 62, and 99.
- [100] M. Zwiessele and N. D. Lawrence. Topslam: Waddington landscape recovery for single cell experiments. *bioRxiv*, 2017. URL <http://biorxiv.org/content/early/2017/02/13/057778>. Cited on pages 68, 73, 74, 77, 78, and 80.