Multivariate volatility modelling with Gaussian and Student-t process latent variable models

Faculty of Physics
Goethe University Frankfurt

Thesis

as part of the requirements to fulfill the degree of
Master of Science in Physics

by

Jesse Jones
born 15.06.1995 in Hanau

April 2020

First Supervisor:	Prof. Dr. J. Triesch
Second Supervisor:	Prof. Dr. N. Bertschinger

Statutory Declarations

I declare that I have written this thesis independently and using just the cited sources and auxiliary means. Both, regarding content, as well as citations, have been identified as such. The thesis has not been submitted to an examination committee in this or any other comparable form before.

Datum: _____ Unterschrift: _____

Acknowledgements

First, I want to thank Prof. Dr. Nils Bertschinger for the opportunity to do the following thesis in his group. Many thanks to Prof. Dr. Jochen Triesch for being my supervisor.
Then, I want to thank Rajbir Nirwan, for fruitful discussions, explanations and advice. I want to thank my Mom and Dad for funding the roof over my head and the food on my plate, and of course for my upbringing which has always led me to think about stuff in more depth.

Abstract

Machine Learning models, influencing almost every aspect of your daily lives without us noticing, have become a cornerstone of modern day amenities. When trusting these models to draw a decision based on causality instead of recalling from memory, statistical inference through stochastic processes provides the solution. Since reality is not as simple as our usual models assume, extracting the useful information from an array of noisy data becomes more important by the day. A financial market is therefore an optimal object of interest to study Bayesian models, which have proven useful in determining factors to describe such a system thoroughly. In this thesis a list of models able to infer important information from noisy systems is presented in order to find a solution to the observed systematical impairment observed in the previous models. They employ Gaussian and Student-t distributions to base the stochastic processes on, showing that the Gaussian and Student-t Process Latent Variable Model provide excellent reconstruction of the covariance structure of stock market data. The analysis of the systematical impairment indicate, that the covariance structure is above all a matter of the size of observed quantities. The work in this thesis therefore provides an arrangement of models with which market analysis can be improved upon through better understanding systematical influences of different variables on the results and interpretability of statistical latent variable models.

Contents

List of Figures

1 Introduction

Linear regression problems can be solved analytically, but nonlinear regression problems are usually analytically intractable. Soon statistics provided a new view on regression, where dimensionality did not represent an intrinsic problem anymore, but just one of computability. Principal Component Analysis was expanded to Probabilistic Principal Component Analysis [1], and Stochastic Processes were introduced in the 1940s with Wieners [2] and Kolmogorovs [3] work, while the idea for stochastic processes dates back as far as the 1880s [4] [5]. The first applicable Gaussian Processes are attributed to the field of geostatistics, where a method called *kriging* [6] [7] was introduced, which is a Gaussian Process Regression for the prediction of spatial data, spanning mostly two and three dimensional input spaces. The field of spatial statistics picked up on Gaussian processes, with overviews given in [8], [9]. After the first computer experiments showed that through looking at noise free data a theory for optimising the hyper parameters can be constructed, the field started to be accepted more widespread. Williams and Rasmussen first described Gaussian Processes in a machine learning context [10], first connecting Gaussian processes to infinite nerual networks [11]. Gaussian processes also tie into regularisation theory, splines, support vector machines and relevance vector machines, other methods from the field of machine learning.

In the field of finance however, Gaussian processes have not been studied much. Prior to e.g. [12], stochastic process models were listed in the literature to have possible applications in finance, but such tests were not widespread. Coming from modern portfolio theory [13], financial problems usually were described through equations involving statistics. A problem unsolved to this day is the prediction of prices in financial contexts, e.g. stocks in a market. Prior research has shown that equally weighted portfolios outperform sample estimate portfolios [14], with high data space dimensions leading to the challenging task of estimating the equally high dimensional covariance matrices. This estimation becomes even more challenging, when the number of assets is large compared to the number of observations. In those cases, the sample covariance matrix is often unstable or even singular. Factor models were then introduced, like the single index model [15], or shrinkage estimators [16], developed and employed in portfolio optimisation.
Topical machine learning models have been used to try and optimise problems in finance. Reinforcement learning was used to try and optimally execute trades [17], asset prices were forecast using neural networks [18], and even with Gaussian processes [19]. Deep autoencoders were used to optimally allocate portfolios [20], and volatility models were developed using Gaussian processes too. At last, time varying covariance matrices were introduced to estimate time-varying dependencies [21]. Bayesian methods have become more popular in this domain, since they work with random variables instead of true values, giving rise

to model incorporated estimation uncertainties. This is also helpful, when estimating data with outliers. Hypotheses testing is also possible through choice of prior, where additional information can be incorporated into the model. A latent space variant of the Gaussian process models are Gaussian process latent variable models [22]. These models can be interpreted as a non-linear extension of standard factor models [23]. Parameters of the GPLVM can be treated readily interpretable, for example as market betas. Here, the sample covariance matrix is shrunk toward a more strucutred matrix given by the kernel. This matrix is supposed to maximize the likelihood function, while shrinking, and therefore systematically reduces sampling errors [12]. Since errors remain that are systematical, further investigation is provided in this work. All experiments were performed on subsets of S&P500 stocks from different time periods of about three years after 2010/01/01. Several novel models were developed, a GPLVM with a latent space time dynamic, a GPLVM with infered time dependent volatility, and a Student-t Latent Variable Model, which were in turn compared to each other and the GPLVM from [12].

2 Theory

2.1 Econometric Motivation

2.1.1 Portfolio Theory

When trying to calculate the optimal portfolio, to maximize returns while still minimizing risk, or in explaining effects of risk diversification, the need for a portfolio theory arises. This theory should explain how inverstors make sensible decisions in the optimisation problem of risk versus return, as well as explain how diversification in experiment achieves such a feature. While previous studies [13] of the models have shown that lesser correlations result in risk reduction, the effect can be used efficiently to more accurately predict stock returns in the future. The model proposed by Markowitz can achieve risk-minimisation, while still providing return maximisation. The goal of the model is to create action instructions enabling investors to build their optimal portfolio based on the combination of investment opportunities. To achieve this, the theory is based upon a set of assumptions.

- The investor is an agent only interested in amassing its own wealth. It is only interested in basing decisions on known financial information.
- The investor agent acts only rational and usage-maximizing, weighing only profits against risks.
- Risks should always be averted, so high-risk actions should only be taken if the expected return grows disporportionately higher.
- The capital market is complete.
- Systematic risk affects all assets, while specific risk only affects the respective specific assets.

If the above assumptions hold, an investor will always choose a portfolio over another, if the expected return μ is greater or equal, with a smaller variance σ, or if the return μ is greater while the variance σ is equal. A complete capital market is a market with negligible transaction costs and perfect information, and the existence of a price for every asset in every possible state of the world [24]. Due to the inequalities in these conditions, only unique portfolios will appear in the theory. A solution to the equations is called an efficient portfolio. Efficient portfolios avert unreasonably high overall risks while retaining a relatively high expected return.

When obtaining an asset with money x_0 and selling it at a later date for money x_1, the ratio between the two can be defined as the absolute return R.

$$R = \frac{x_1}{x_0} \tag{2.1}$$

From this, we can easily derive the rate of return, or relative return, which will in later chapters will be regarded as *return* r.

$$r = R - 1 = \frac{x_1 - x_0}{x_0} \tag{2.2}$$

These definitions hold for buy-sell actions, as well as short selling, where the investor first sells assets from the broker and then rebuys them at a later date, resulting in a sell-buy action. In the case of short selling, double negative signs in the fractions arise, but cancel out. Risks are, for now, expected to always be real positive values, never exactly zero. This leads to a normalized system of investments through weights w_i describing how much of the total monetary aggregate is invested into asset or portfolio i

$$\sum_{i=1}^{n} w_i x_0 = x_0. \tag{2.3}$$

The rate of return of the total investment is

$$r = R - 1 = \sum_{i=1}^{n} R_i w_i - \sum_{i=1}^{n} w_i = \sum_{i=1}^{n} r_i w_i. \tag{2.4}$$

The Markowitz Mean-Variance Portfolio Theory then models the rate of returns on assets as random variables. A global optimisaition is then applied to find the best weights for each part of the portfolio. The volatility of an asset is surrogated through the proportional variance. We find the Markowitz problem to be the optimisation of

$$\mathbf{min}\left(0.5\boldsymbol{w}^\top \Sigma \boldsymbol{w}\right) \tag{2.5a}$$

$$\text{subject to } \boldsymbol{m}^\top \boldsymbol{w} \geq \mu_b \text{ and } \boldsymbol{e}^\top \boldsymbol{w} = 1 \tag{2.5b}$$

Where w is the vector of weights, Σ the covariance matrix of the random random vector z containing the returns, $m^\top w$ the mean of the random variable, $w^\top \Sigma w$ is the variance and $\mu_b = \mathbb{E}[r_b]$ the acceptable baseline expected return. Also, e is the unit vector of the arbitrary dimension suiting the problem and n, the number of possible assets. To solve this nonlinear problem, Karush-Kuhn-Tucker (KKT) conditions [25] can be formulated, which allow the analytic and algorithmic solution of the problem. At first, showing that the problem is feasible guarantees the existence of a finite optimal value, as well as the existence of a solution to find the optimal value. If the acceptable baseline expected return is below the mean, $\mu_b > \boldsymbol{m}^\top \boldsymbol{w}$, the obtained solution is a least variance solution. The mean return associated with the least-variance solution μ_{lv} is

$$\mu_{lv} = \frac{\boldsymbol{m}^\top \Sigma^{-1} \boldsymbol{e}}{\boldsymbol{e}^\top \Sigma^{-1} \boldsymbol{e}} \tag{2.6}$$

The case $\mu_b = \boldsymbol{m}^\top \boldsymbol{w}$ is also possible, where we can identify that the optimal portfolio usually is two-fold.

$$w = (1-\alpha)\frac{\Sigma^{-1}\boldsymbol{e}}{\boldsymbol{e}^\top\Sigma^{-1}\boldsymbol{e}} + \alpha\frac{\Sigma^{-1}\boldsymbol{m}}{\boldsymbol{e}^\top\Sigma^{-1}\boldsymbol{m}} = (1-\alpha)w_{lv} + \alpha w_{mk} \tag{2.7}$$

Any solution to the Markowitz problem can be presented as a linear combination of these two sets of weights: the risk-free least-variance solution, and the market portfolio, which incorporates the rest of the knowledge about the market. The previous statement is also known as the *Two Fund Theorem*. We can obtain the solution space by finding all possible curves parametrised by

$$\left(\sqrt{\boldsymbol{w}_i^\top}, r_i\right) = \left(\sqrt{var(r_i)}, \mathbb{E}(r_i)\right) \tag{2.8a}$$

$$\text{where } r_i = \boldsymbol{w}_i^\top r, \text{ as } i, \text{ the number of assets, varies from 0 to } +\infty \tag{2.8b}$$

If a risk-free asset is introduced now, as is a good approximation for very low risk securities, e.g. treasury bonds, some properties of the described system change. First, the *One Fund Theorem* is introduced. It states, that for the case of available risk-free assets, the least variance portfolio is always comprised of only those risk-free assets. Also, every efficient portfolio can be created using a combination of these risk-free portfolios and another Fund F with risk > 0. This leads directly towards the Capital Asset Pricing Model (CAPM), which provides and solves for optimal portfolios, assuming risk-free assets exist.
This still leaves open the question of how to optimally calculate a covariance matrix, so that it is the closest approximation of the true covariance matrix. A simple theory of construction is given by the sample covariance matrix,

$$K = \frac{1}{D}(\boldsymbol{r} - \hat{\boldsymbol{\mu}})(\boldsymbol{r} - \hat{\boldsymbol{\mu}})^\top. \tag{2.9}$$

Where $\hat{\boldsymbol{\mu}}$ is again the mean of the asset samples respectively. If the number of days of observation D is large enough, this estimation is called the maximum likelihood estimator. It provides an unbiased estimation, but in practice, especially if the ratio N/D is not small, the matrices often become unstable or singular. Other methods include the Ledoit-Wolf estimation, where shrinkage is used to estimate the covariance matrix [16], or Gaussian Processes, which will be discussed in sections 2.3 and **??**.

2.1.2 Capital Asset Pricing Model

Within the Capital Asset Pricing Model (CAPM), a linear model including latent spaces, we can define the market portfolio r_M as

$$r_M = \begin{bmatrix} r_f \\ r \end{bmatrix}^\top \begin{bmatrix} 1 - \boldsymbol{e}^\top \Sigma^{-1}(\boldsymbol{m} - r_f \boldsymbol{e}) \\ \Sigma^{-1}(\boldsymbol{m} - r_f \boldsymbol{e}) \end{bmatrix}, \tag{2.10}$$

which leads to the expected return on any asset in CAPM

$$\mu_i = r_f \beta_i (\mu_M - r_f), \tag{2.11}$$

where β_i is the covariance relation of market portolio r_M and individual asset i. β is understood to be the risk,

$$\beta_i = \frac{\sigma_{iM}}{\sigma_M^2}. \tag{2.12}$$

Then, the expected return of the portfolio μ_r satisfies

$$\mu_r = r_f + \frac{\mu_M - r_f}{\sigma_M} \sigma_r, \tag{2.13}$$

where for a given portfolio r the mean portfolio return $\mu_r = \mathbb{E}(r)$ and $\sigma_r^2 = \mathrm{var}(r)$. The line of efficiency is then given by

$$\mu_M = \mathbb{E}(r_M) = r_f + (\boldsymbol{m} - r_f \boldsymbol{e})^\top \Sigma^{-1}(\boldsymbol{m} - r_f \boldsymbol{e}), \tag{2.14a}$$

$$\sigma_M^2 = \mathrm{var}(r_M) = (\boldsymbol{m} - r_f \boldsymbol{e})^\top \Sigma^{-1}(\boldsymbol{m} - r_f \boldsymbol{e}). \tag{2.14b}$$

Recalling the relation 2.2, where the return rate was defined, we can find the CAPM pricing formula, substituting the sell-price x_1 with P, and the buy-price x_0 with Q.

$$P = \frac{\mu_Q}{1 + r_f + \beta_r(\mu_M - r_f)} \tag{2.15}$$

The main connection between the expected value of the return of an asset and its risk is described by

$$\mathbb{E}[\boldsymbol{r}_n] = \boldsymbol{r}_{rf} + \beta_n \mathbb{E}[\boldsymbol{r}_m - \boldsymbol{r}_{rf}] = \boldsymbol{r}_f + \beta_n \mathbb{E}[\boldsymbol{r}_{me}]. \tag{2.16}$$

with the risk-free ($\beta = 0$) return rate $\boldsymbol{r}_{rf}$, which is the expected return rate for a zero-risk portfolio. For any portfolio with risk $\beta_n \neq 0$, the excess return of the market $\boldsymbol{r}_{me} = \boldsymbol{r}_m - \boldsymbol{r}_f$ applies, and the investor is compensated with a higher expected return. Note, that all $\boldsymbol{r}_{f/m/me} \in \mathbb{R}^D$ for D different days. So equation 2.16 can be rewritten in terms of the excess return $\boldsymbol{r}_e = \boldsymbol{r} - \boldsymbol{r}_f$ with excess return of an asset n: $\boldsymbol{r}_{e,n}$,

$$\mathbb{E}[\boldsymbol{r}_{e,n}] = \beta_n \mathbb{E}[\boldsymbol{r}_{me}]. \tag{2.17}$$

2.1.3 Arbitrage Pricing Theory

If the model is then expanded to include multiple additional risk factors $\boldsymbol{F}$, other than systematic risk factors of the market and specific risk factors of the assets, we arrive at Arbitrage Pricing Theory (APT) [26]. APT assumes, that every asset return follows a factor structure with an additional noise term ϵ_n, that is modeled as a Gaussian with zero mean and variance σ_n^2 [27]. The factor structure allows the Arbitrage pricing theory to incorporate other factors, which different studies have shown to be at least partly influential on the predictive power of CAPM. One of the aforementioned models is the Fama-French three factor model, consisting of the expected returns, the difference between returns in portfolios of small and big stocks, and the returns of portfolios containing high and low stocks [28] [29].

$$\mathbb{E}[\boldsymbol{r}_M] - \boldsymbol{r}_f = \boldsymbol{\beta}_M[E[\boldsymbol{r}_M] - \boldsymbol{r}_f] + \boldsymbol{\beta}_{SMB}[\mathbb{E}[R_{SMB}]] + \boldsymbol{\beta}_{HML}[\mathbb{E}[R_{HML}]]. \tag{2.18}$$

The expected returns are shown to behave like

$$\boldsymbol{r}_n = \alpha_n + \boldsymbol{F}\boldsymbol{\beta}_n + \epsilon_n \tag{2.19a}$$

$$\mathbb{E}[\boldsymbol{r}_{e,n}] = \mathbb{E}[\boldsymbol{F}]\boldsymbol{\beta}_n \tag{2.19b}$$

This equation for the asset return factor structure $\boldsymbol{r}_n$ can then be rewritten, so that it matches the form of a Gaussian Process Latent Variable Model, which will be explained in detail in section 2.3.

$$\boldsymbol{r}_{n,:} = f(\boldsymbol{B}_{n,:}) + \epsilon_n \tag{2.20}$$

where : stands for all entries in the specified dimension, and $\boldsymbol{B}$ is the factor matrix constructed from the $\boldsymbol{\beta}_n$. The three factor model also has shortcomings [30], which especially arise from the momentum effect, where due to inverstors beliefs in well performing stocks over a timespan of e.g. 3-12 months, these stocks continue to do well afterwards. The same effect also applies to stocks that do not do well over such a timespan, they will probably not do well for the next few months after.
In APT, the factors are unobserved quantities, that can be inferred using e.g. PCA. The drawback of PCA, assuming an error with fixed variance for each element of the return matrix r, highlights that alternatives are needed. Another approach was factor analysis [31], which allowed different noise variances for each asset contained in r. Since there is no analytic result to the optimization of the marginal distribution in this case, these can be solved iteratively using the expectation maximization algorithm [32].

2.1.4 ARCH and GARCH

ARCH

An econometrics model for time series data describing the variance of the current error through statistics is the autoregressive conditional heteroskedasticity (*ARCH(q)*) model (xxx). The ARCH model is based upon the assumption, that the current error term is a function of the previous time periods error terms. In terms of variance, often the square of the previous time steps error terms are used. In practice, the ARCH model is often used for modeling financial time series exhibiting time-dependent volatility, or so called volatility clustering, e.g. periods of higher volatility, followed by periods of comparably lower volatility, and the other way around. The model is specified through the time series terms.

$$\epsilon_t = \sigma_t z_t \tag{2.21}$$

Here, ϵ_t denotes the error terms that make up the ARCH process, z_t is a random variable defined by a strong white noise process, and the series σ_t^2 is educed as

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2, \tag{2.22}$$

where all free parameters $\alpha_0, \alpha_i > 0$. The ARCH(q) model was shortly after refined as GARCH(p,q) model, where an important generalization was implemented into the model.

GARCH

The Generalized autoregressive conditional heteroskedasticity (*GARCH(p,q)*) model is the generalization of the ARCH(q) model, which directly includes lag. It is defined by

$$y_t = X - t'b + \epsilon_t, \tag{2.23a}$$

$$\epsilon_t | \psi_{t-1} \; \mathcal{N}(0, \sigma_t^2), \tag{2.23b}$$

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2. \tag{2.23c}$$

We then find the lag length through estimating the optimal ARCH process and estimating autocorrelations.

$$y_t = a_0 + \sum_{i=1}^{q} a_i y_{t-i} + \epsilon_t, \tag{2.24a}$$

$$\rho = \frac{\sum_{t=i+1}^{T} (\hat{\epsilon}_t^2 - \hat{\sigma}_t^2)(\hat{\epsilon}_{t-1}^2 - \hat{\sigma}_{t-1}^2)}{\sum_{t=1}^{T} (\hat{\epsilon}_t^2 - \hat{\sigma}_t^2)}, \tag{2.24b}$$

For large samples, GARCH errors are implicated if the standard deviation $\rho(i) = 1/\sqrt{T}$. Based on this, the Ljung-Box test can be applied to base a decision if to reject or accept it´s null hypothesis, that there are no ARCH or GARCH errors. In later chapters of this work, the GARCH process is sampled using a Hamilton Chain Monte Carlo algorithm (xxx pyflux), and then used to create datasets that are normed on the expected volatility.

2.2 Stochastic Process Models

2.2.1 Introduction

Assessing the chance of something happening in real life, one encounters more than just probabilities. It is directly connected to the model of our world, the probabilities of certain events happening inside the model, and how good the model actually resembles what we understand to be the truth. Models can be created using two trains of thought (xxx schools?) in statistics, about how to assign probabilities from observing events. Firstly, there is frequentist statistics, centered around the idea, that there is a fixed probability for an event that can, in theory, be derived from axioms[xxx]. Observing an event many times shall then directly converge towards the true value. Observe a coinflip as an example, the difference can be illustrated quite easily. If the coin is flipped often enough and the coin is fair, then we observe that every time the coin is flipped, the probabilities for observing "*heads*" and "*tails*" will converge another step towards 0.5. If we repeat the experiment with an unfair

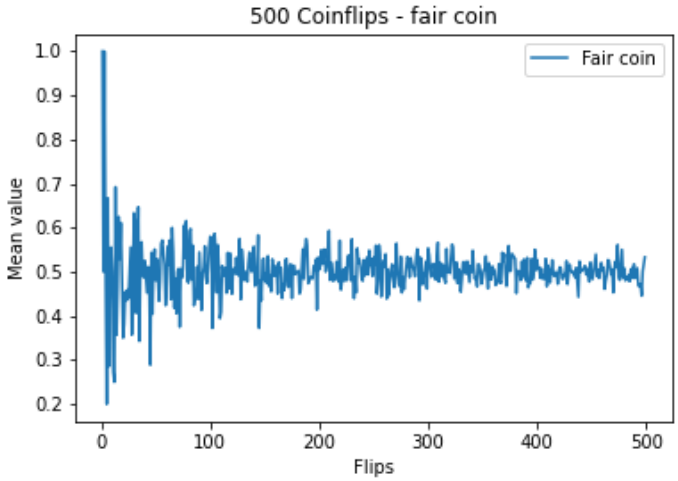


Figure 2.1: A simulated fair coin flip, with 500 iterations of either *heads* (1) or *tails* (0).

coin instead, our assumption of a fair coin will now still lead us to assume the probabilities for *heads* and *tails* were 0.5, just like in a perfect system. Here we are still assuming that if we observe enough events, the probabilities should be verified by the experiment. Statistical uncertainty in observing a fair coin could shift the observed probabilities for a while, but will always converge. Only after observing many events, we would start to question the assumptions we had at the beginning, e.g. of the coin being fair, because convergence is slow. So, from the frequentists perspective, if we were in the situation of playing coinflip with an unkown coin, it would take a long time, until we would accept the possibility that our model could be wrong. This inherent trust into the models is therefore used mostly in fields like physics, where the theory is assumed to reflect the ground truth no matter what, with the assumption that constants are always constant and the set of rules never changes, independent of where you are in the universe. This is called the Noether theorem, but since

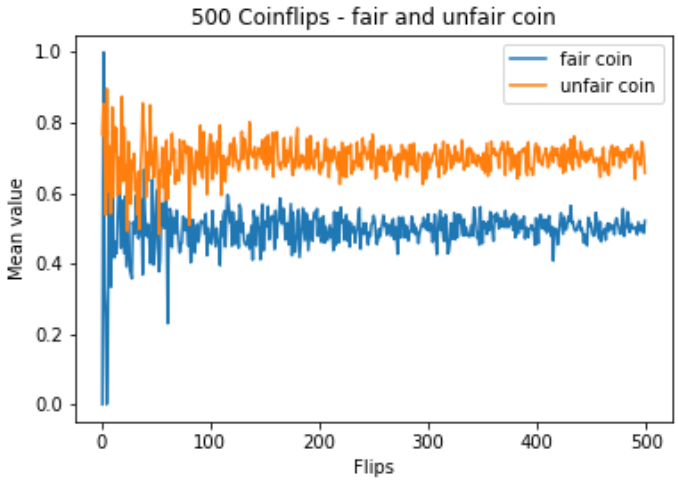


Figure 2.2: A simulated fair coin flip (blue) compared to the unfair coin flip (orange). Even though both tend towards different values for large sample sizes, in the first few flips, it is very hard to distinguish which coin is fair. Experimental conditions are the same as with the simulated fair coin flip.

it only applies to the set of rules in physics, where controlling variables can be enforced with great precision, other fields need a more flexible theory to determine the probability of an event. So, assuming there are variables influencing our system in ways we cannot foresee with absolute precision, we need to be able to update the expected probabilities as a function of our observations. Since the probabilities are a crucial part of the model we applied to reality in the first place, we need to find a model that can be updated each time an observation took place. Bayesian statistics allows us to update our beliefs along the way. It dates back to Thomas Bayes, who in his essay "*An essay towards solving a problem in the doctrine of chances*" [xxx] introduced the first version of this idea. Our percieved reality can always be flawed, not limited to a fair or unfair coin, but models for reality of all sorts can have intrinsic flaws, that can be quantified using this method of determining probabilities and updating models, along the way. Through this, probabilities become dependent on believability and credibility, confidence in decisions or environmental variables of the problem. Creating a model in bayesian statistics also allows for a causal bias introduced into the model, before observations even took place. So, if for example you would consider playing against someone with a coin you do not yet know to be fair, it is best to assume the coin is unfair at first. Noticing that a coin is unfair faster, and with higher probability, to update the model is valuable information. Previously we noticed, that the flexibility of bayesian models allow us to take into account environmental variables, but how many of those will actually lead to better predictions from the model? Occams Razor (xxx) is a good rule to follow, which would roughly translate to "*choose a model as complicated as necessary but no more than that*". Or, in other words, a more complex model with more hyperparameters will in the end lead to better explanation ("a better fit") of the observed data, but prediction power will intrinsically decrease along the way, due to more noise being learned as a feature.

2.2.2 Bayesian Statistics

Models using the bayesian framework of statistics rely on conditional probability. These probabilities are conditioned on previous assumptions about the general environment. This can be surroundings, game states, environmental variables, etc. Since the properties of complex systems are majorly influenced by these traits, incorporating them into the model is of utmost importance. The general case of Bayes rule combines prior knowledge of the model, called a prior probability, with the observed evidence and its likelihood to form the posterior probability. The posterior probability can be interpreted as the probability of observing a certain set of observations dependent on the likelihood of the observations, given the model.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.25}$$

Here, A is the proposition and B the evidence, and we try to get the posterior $P(A|B)$, the degree of belief in A if we have accounted B to be true. $P(A)$ is the prior, the initial belief in A, while $P(B)$ is the likelihood of the evidence being true. This rule currently applies only to probabilities, but it can just as well be scaled up to probability distributions.

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)} \tag{2.26}$$

Probability distributions, which are factors defining models, can be tested for reliability via Bayes rule too. Also using Bayes rule, we can test our model assumptions and compare models.

$$p(M|D, I) = \frac{p(D|M, I)p(M|I)}{p(D|I)} \tag{2.27}$$

Here, M is the model in question, while D is the observed Data, and I is the previously known information of the system. We can then formulate a hierarchical Bayesian workflow.

1. Use Bayes Rule to construct a naive model using only prior knowledge
2. Update knowledge inside naive model
3. Create more complex model using Bayes rule for models
4. Update knowledge about complex model
5. repeat steps 1-4 until KL-Divergence converges to 0.

Every bit of knowledge we have about the system is now encoded in probability distributions, but to calculate this workflow, we need some class of probability distributions, that is sensible to use in most models, and which can preferrably be solved analytically. In experiments, we will see that this workflow is not as easy as expected, since the evidence is often intractable (xxx look up from VB Wikipedia), which then can only be evaluated approximately. Also, we need to take into account, that the total solution space of Bayesian priors may not contain the solution we try to achieve, because the true model may be something very different,

from what we originally thought it was. Here Occams Razor is again the doctrine we use to make decisions about model choice: Our model shall be complicated enough to grasp the underlying dynamic, but needs to be naive enough to not learn noise as a feature. This leads to Gaussian processes, since they have a lot of convenient properties. The priors of Gaussian Processes ($\mathcal{GP}$) are appropriate and have infinite basis functions. These are flexible enough for nonlinear models, but still retain analytic solutions. Through this, they are especially computationally feasible. Also, Student-t-processes, as they are scaled Gaussian processes, seem to be a logical choice. Even though they are more restrained in the amount of computation tasks that can be done analytically, compared to the Gaussian Processes.
In applications of the Bayes rule there are posterior terms containing the information about the data, but sometimes they have combinatorially large search spaces. This leads to the intractability of these terms, represented by $\int_X P(Y,X)dX$.

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)} = \frac{P(Y|X)P(X)}{\int_Z P(Y,X)dX} \tag{2.28}$$

To obtain the approximation using Variational Bayes, we introduce a variational distribution.

$$Q(X) \approx P(X|Y) \tag{2.29}$$

This distribution $Q(X)$ is restricted to be of a family of simpler distributions than the true distribution, e.g. a posterior, $P(X|Y)$. The distribution family is selected with the intention of finding a $Q(X)$ similarly enough to $P(X|Y)$ so that $Q(X)$ can be used instead of $P(X|Y)$. We use KL-divergence to measure the dissimilarity of the distributions, but with a slight variation. Instead of using a KL-version $D_{KL}(p|q)$ that is applied in the expectation propagation algorithm, we use the reverse KL-divergence.

$$D_{KL}(Q|P) \triangleq \sum_X Q(X) \log \frac{Q(X)}{P(X|Y)} \tag{2.30}$$

given, that

$$P(X|Y) = \frac{P(Y,X)}{P(Y)}, \tag{2.31}$$

the KL Divergence here can be rewritten as

$$D_{KL}(Q|P) = \sum_X Q(X)\Big[\log \frac{Q(X)}{P(X,Y)} + \log P(Y)\Big] \tag{2.32a}$$

$$= \sum_X Q(X)[\log Q(X) - \log P(X,Y)] + \sum_X Q(X)[\log P(Y)]. \tag{2.32b}$$

After seeing that $P(Y)$ is constant with respect to X and $\sum_X Q(X) = 1$ because $Q(X)$ is a distribution, and rewritten with the definition of the expected value for a discrete random variable, we arrive at

$$\log P(Y) = D_{KL}(Q|P) - \mathbb{E}_X[\log Q(Y) - \log P(X,Y)] = D_{KL}(Q|P) + \mathcal{L}(Q) \tag{2.33}$$

Since $P(Y)$ is fixed under Q, maximizing $\mathcal{L}(Q)$ minimizes the KL Divergence from Q to P. $\mathcal{L}(Q)$ is also known as the Evidence Lower Bound, which will be important to the evaluation of models later on. The variational distribution is usually assumed to factorize over some partition of the latent Variables X.

$$Q(X) = \prod_{i=1}^{M} q_i(X_i) \tag{2.34}$$

Using Variational Calculus we find that the best distribution in a Bayesian workflow with Variational Bayes is

$$q_j^*(X_j|Y) = \frac{\exp(\mathbb{E}_{i\neq j}[\ln p(X,Y)])}{\int \exp(\mathbb{E}_{i\neq j}[\ln p(X,Y)])dZ_j}, \tag{2.35}$$

where $\mathbb{E}_{i\neq j}[\ln p(X,Y)]$ is the expectation value of the logarithm of the joint probability of data and latent variables not in the partition. Due to circular depencies between the parameters, the evaluation of these in a program offers to be solved iteratively. With this, due to the type of the distribution $Q(X)$, an analytical approximation for the posterior probability can be achieved, allowing the iterative calculation of parameters of the model. Variational Bayes can be used out of the box. Stan provides an implementation known as Automatic Differentiation Variational Inference (ADVI, Kucukelbir) that approximates the posterior with a Gaussian as the variational approximation. For more details, see section 2.4.

2.2.3 Gaussian Processes

Supervised Learning Machine Learning can be divided into two classes of problems, regression and classification. Where in Classification the machine learning algorithm should decide between discrete class labels, the output of a regression algorithm is usually a continuous quantity. Due to the nature of the Gaussian Processes suiting regression problems, a focus is put on these, even though Gaussian Processes can also solve classification problems [23]. There are several ways of interpreting a Gaussian process, the function space view, and the weight-space view. In the weigth space view, the Gaussian Process is pictured as a process, with infinity many basis functions with corresponding weights. In the function space view, the gaussian process is a distribution over functions, and inference takes place in the function space defined by the process. The Gaussian process, as mentioned in section 2.1.3 and 2.1.3, is mainly used to find the covariance matrices, that encode the covariance structure of the problem.

Weight Space View

A simple Gaussian process will be derived here. Starting with a standard linear regression model with Gaussian noise

$$f(x) = \boldsymbol{x}^\top \boldsymbol{w} \tag{2.36a}$$
$$y = f(\boldsymbol{x}) + \epsilon \tag{2.36b}$$
$$\epsilon \sim \mathcal{N}(0, \sigma_n^2) \tag{2.36c}$$

Here $\boldsymbol{x}$ is the input vector, $\boldsymbol{w}$ is the vector containing the weigths, f is the function value and y the observed target value. Also, ϵ is the independent noise, and it follows an independent, identically distributed Gaussian distribution with zero mean and variance σ_n^2. The likelihood, as probability density of observations given parameters factors over the cases of the training set.

$$p(\boldsymbol{y}|X, \boldsymbol{w}) = \prod_{i=1}^{n} p(y_i|\boldsymbol{x}_i, \boldsymbol{w}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_n} \exp\Big(- \frac{(y_i - \boldsymbol{x}_i^\top \boldsymbol{w})^2}{2\sigma_n^2}\Big) \tag{2.37a}$$
$$= \frac{1}{(2\pi\sigma_n^2)^{n/2}} \exp\Big(- \frac{1}{2\sigma_n^2}|\boldsymbol{y} - X^\top \boldsymbol{w}|^2\Big) = \mathcal{N}(X^\top \boldsymbol{w}, \sigma_n^2 I). \tag{2.37b}$$

X is all the data points from the input, and $|x|$ stands for the euclidian length of a vector x. Also, a prior must be specified, containing our beliefs about the parameters before observing the input. Therefore, we will distribute the weights using a zero mean Gaussian prior with covariance matrix Σ_p,

$$\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \Sigma_p). \tag{2.38}$$

To apply Bayes rule, we need a marginal likelihood, which acts as a normalizing constant, and is independent of the weights by integrating them out.

$$p(\boldsymbol{y}|X) = \int p(\boldsymbol{y}|X, \boldsymbol{w}) p(\boldsymbol{w}) d\boldsymbol{w} \tag{2.39}$$

The full Bayes rule in this case then becomes the following

$$p(\boldsymbol{w}|\boldsymbol{y}, X) = \frac{p(\boldsymbol{y}|X, \boldsymbol{w})p(\boldsymbol{w})}{p(\boldsymbol{y}|X)} \tag{2.40}$$

Using complete the square and ignoring the marginal likelihood, we find

$$p(\boldsymbol{w}|X, \boldsymbol{y}) \sim \mathcal{N}(\frac{1}{\sigma_n^2}(\frac{1}{\sigma_n^2}XX^\top + \Sigma_p^{-1}), (\frac{1}{\sigma_n^2}XX^\top + \Sigma_p^{-1})^{-1}) \tag{2.41}$$

With this, we can easily find the predictive distribution for $f_* \triangleq f(\boldsymbol{x}_*)$, by averaging the output of all possible linear models with respect to the Gaussian posterior,

$$p(f_*|\boldsymbol{x}_*, X, \boldsymbol{y}) = \int p(f_*|\boldsymbol{x}_*, \boldsymbol{w})p(\boldsymbol{w}|X, \boldsymbol{y})d\boldsymbol{w} \tag{2.42a}$$

$$= \mathcal{N}(\frac{1}{\sigma_n^2}\boldsymbol{x}_*^\top(\frac{1}{\sigma_n^2}XX^\top + \Sigma_p^{-1})^{-1}X\boldsymbol{y}, \boldsymbol{x}_*^\top(\frac{1}{\sigma_n^2}XX^\top + \Sigma_p^{-1})^{-1}\boldsymbol{x}_*). \tag{2.42b}$$

This model can easily be expanded by transforming the input into some high dimensional space using a set of basis functions and subsequently applying the linear model in this higher dimensional space. Applying the map $\phi(x) = (1, x, x^2, x^3, x^4, ...)^\top$ would for example implement polynomial regression, during which a D-dimensional input vector x is mapped into an N-dimensional feature space. This allows us to rewrite the model as

$$f(\boldsymbol{x}) = \phi(\boldsymbol{x})^\top \boldsymbol{w} \tag{2.43a}$$

$$f_*|\boldsymbol{x}_*, X, \boldsymbol{y} \sim \mathcal{N}\Big(\frac{1}{\sigma_n^2}\phi(\boldsymbol{x}_*)^\top(\sigma_n^{-2}\Phi\Phi^\top + \Sigma_p^{-1})^{-1}\Phi\boldsymbol{y}, \tag{2.43b}$$

$$\phi(\boldsymbol{x}^\top)(\sigma_n^{-2}\Phi\Phi^\top + \Sigma_p^{-1})^{-1}\phi(\boldsymbol{x}_*)\Big) \tag{2.43c}$$

where $\Phi(X)$ is the aggregation of all columns $\phi(\boldsymbol{x})$ for all cases in the data set in the transformed space. It presents a computational time benchmark when executing such a model, since the inversion of $N \times N$ matrices is of $\mathcal{O}(N^3)$ complexity. With high-dimensional feature spaces, reducing the argument of the complexity function is valuable. So, sometimes rewriting the equations as

$$f_*|\boldsymbol{x}_*, X, \boldsymbol{y} \sim \mathcal{N}\Big(\phi_*^\top\Sigma_p\Phi(K + \sigma_n^2 I)^{-1}\boldsymbol{y}, \phi_*^\top\Sigma_p\phi_* - \phi_*^\top\Sigma_p\phi_*(K + \sigma_n^2 I)^{-1}\phi_*^\top\Sigma_p\phi_*\Big) \tag{2.44}$$

is useful, since the complexity $\mathcal{O}(n^3)$ is the maximum complexity possible for n datapoints [23]. Then, the kernel function $k(\boldsymbol{x}, \boldsymbol{x'}) = k(\cdot, \cdot) = \phi_*^\top\Sigma_p\phi_*$ is easily identified. The kernel is often also called covariance function.

Function Space View

This derivation can also be done in function space. Here, we can define a Gaussian Process as a collection of random variables, any finite number of which have a joint Gaussian distribution. Since a Gaussian process is solely defined by its mean and kernel function, it can be

written as

$$m(x) = \mathbb{E}[f(x)] \tag{2.45a}$$
$$k(\boldsymbol{x}, \boldsymbol{x'}) = \mathbb{E}[(f(\boldsymbol{x}) - m(\boldsymbol{x}))(f(\boldsymbol{x'}) - m(\boldsymbol{x'}))] \tag{2.45b}$$
$$f(x) \sim \mathcal{GP}(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'})). \tag{2.45c}$$

The Function space view notation will be the notation further on used in this thesis. Here, $\mathcal{GP}$ stands for a Gaussian Process, $m(\boldsymbol{x})$ is the mean function, and $k(\boldsymbol{x}, \boldsymbol{x'})$ is the kernel function. The consistency requirement, also marginalization property, states, that for a $\mathcal{N}$ $(y_a, y_b) \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ with general mean vector $\boldsymbol{\mu} = (m(\boldsymbol{x}_a), m(\boldsymbol{x}_b))^\top$ and covariance matrix Σ, the underlying $\mathcal{N}(y_a)$ is specified by $y_a \sim \mathcal{N}(\mu_a, \Sigma_{aa})$. Σ_{aa} then is the submatrix of Σ relevant to the distribution for y_a. Since the submatrixes are not intertwined in Σ, the examination of a larger set of variables does not change the distribution of a subset. Using the previously discussed linear regression model $f(x) = \phi(\boldsymbol{x})\boldsymbol{w}^\top$ with a gaussian prior on $\boldsymbol{w}$ we have mean and covariance

$$\mathbb{E}[f(\boldsymbol{x})] = \phi(\boldsymbol{x})^\top \mathbb{E}[\boldsymbol{w}] = 0 \tag{2.46a}$$
$$\mathbb{E}[f(\boldsymbol{x})f(\boldsymbol{x'})] = \phi(\boldsymbol{x})^\top \mathbb{E}[\boldsymbol{w}\boldsymbol{w}^\top]\phi(\boldsymbol{x}) = \phi(\boldsymbol{x})^\top \Sigma_p \phi(\boldsymbol{x'}). \tag{2.46b}$$

Given a kernel function, we can generate a random sample from a GP with Covariance matrix $K(X_*, X_*)$, where X_* is the matrix containing all input points.

$$\boldsymbol{f_*} \sim \mathcal{N}(\boldsymbol{0}, K(X_*, X_*)) \tag{2.47}$$

The most used kernel function is the squared exponential,

$$k_{sqe}(\boldsymbol{x_i}, \boldsymbol{x_j}) = exp(-\frac{1}{2l^2} d_{ij}^2). \tag{2.48}$$

Incorporating the knowledge of the training data for noise free predictions then leads us to

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{f_*} \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X, X) & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix}\right), \tag{2.49}$$

where now all variables without $_*$ are training variables, and all variables with $_*$ are the respective test variables. This gives

$$\boldsymbol{f_*}|X_*, X, \boldsymbol{f} \sim \mathcal{N}(K(X_*, X)K(X, X)\boldsymbol{f}, K(X_*, X_*) - K(X_*, X)K(X, X)^{-1}K(X, X_*)) \tag{2.50}$$

for the conditioned, joint Gaussian prior distribution on observations. Extending the model to incorporate noise on the data, we find

$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{f_*} \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X, X) + \sigma_n^2 I & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix}\right), \tag{2.51}$$

where the key predictive equations for Gaussian process regression are

$$\begin{aligned} \boldsymbol{f}_*|X, \boldsymbol{y}, X \sim \mathcal{N}(\overline{\boldsymbol{f}_*}, cov(\boldsymbol{f}_*)), \\ \overline{\boldsymbol{f}_*} \triangleq \mathbb{E}[\boldsymbol{f}_*|X, \boldsymbol{y}, X_*] = K(X_*, X)[K(X, X) + \sigma_n^2 I]^{-1}\boldsymbol{y}, \\ cov(\boldsymbol{f}_*) = K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma_n^2 I]^{-1}K(X, X_*). \end{aligned} \tag{2.52a}$$

Introducing the marginal likelihood $p(\boldsymbol{y}|X)$ again, we find

$$p(\boldsymbol{y}|X) = \int p(\boldsymbol{y}|\boldsymbol{f}, X)p(\boldsymbol{f}|X)d\boldsymbol{f}. \tag{2.53}$$

Solving this integral analytically it results to

$$log\Big(p(\boldsymbol{y}|X)\Big) = -\frac{1}{2}\boldsymbol{y}^\top(K + \sigma_n^2 I)^{-1}\boldsymbol{y} - \frac{1}{2}log|K + \sigma_n^2 I| - \frac{n}{2}log(2\pi). \tag{2.54}$$

Using this, we have derived the regular Gaussian process. It is easy to see, that iterative usage of the learning equations (previously: key predictive equations) is the same as learning from all data points at once due to the marginalization property. The following figures provide an example. At first, we introduce random prior samples in figure 2.3, specified by a squared exponential kernel centered around a mean of 0 and with a covariance of 2. Then,

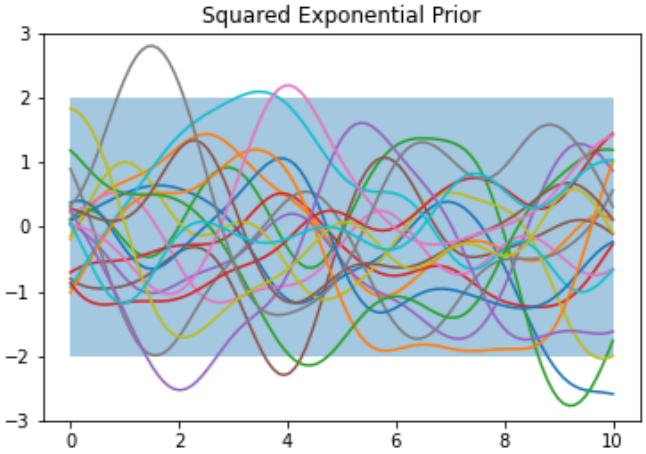


Figure 2.3: Samples from a Gaussian Process prior with a squared exponential kernel. To add to the draws, the 95% confidence interval is highlighted with a blue background.

we introduce data points, which were randomly generated using a modified sine function with added gaussian noise, see figure 2.4. Using the learning equations 2.52, we can train a Gaussian process on the data points and find the following posterior samples, see figure 2.5 and confidence intervals, having solved the Gaussian Process. The confidence intervals in figure 2.6 have 1, 2, and 3 σ. This depiction also highlights the interpretation of GP predictions as probability distributions.

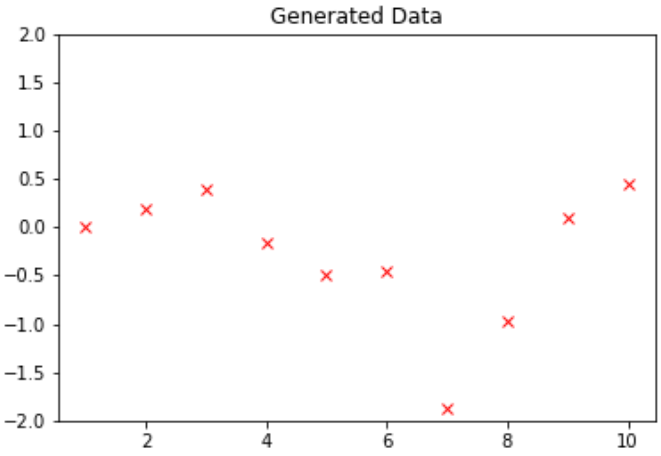


Figure 2.4: Data generated from a sine function with added Gaussian noise.

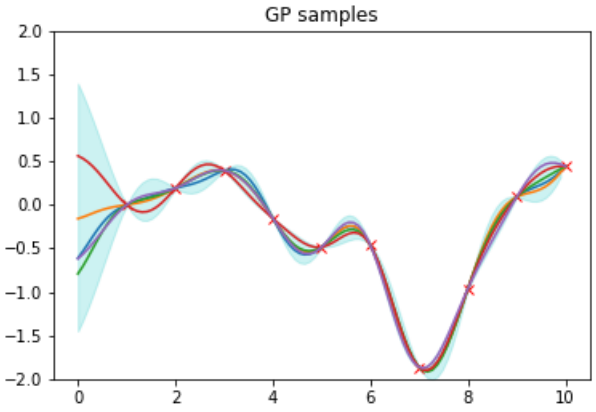


Figure 2.5: The GP conditioned on the generated dataset, with the 95% confidence interval and 5 samples drawn from the GP.

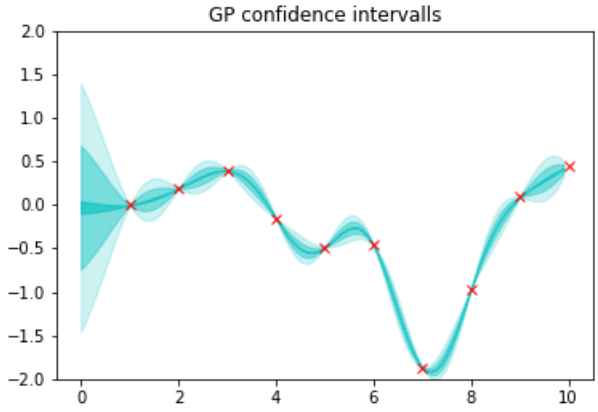


Figure 2.6: The GP conditioned on the generated dataset, with confidence intervals drawn into the graph as blue shading.

2.2.4 Gaussian Kernel Functions

A covariance function, or kernel function is a key part of the model used. Since not any function linking two inputs x and x' will in general be a viable kernel function, necessary, as well as useful properties of kernel functions need to be discussed. First, kernel functions must be positive semidefinite,

$$\int k(\boldsymbol{x}, \boldsymbol{x'}) f(\boldsymbol{x}) f(\boldsymbol{x'}) d\mu(\boldsymbol{x}) d\mu(\boldsymbol{x'}) \geq 0. \tag{2.55}$$

Here, μ denotes a respective measure. Kernel functions also can be stationary, when it is a function of $\boldsymbol{x} - \boldsymbol{x'}$, exhibiting translation invariance in input space. A kernel function can be isotropic, a function of $|\boldsymbol{x} - \boldsymbol{x'}|$, making it invariant to all rigid motions. Isotropic kernel functions are also known as radial basis functions (RBF). If the kernel function is a function of $\boldsymbol{x} \cdot \boldsymbol{x'}$, it is called a dot-product kernel function. These are invariant to a rotation of coordinates about the origin, but not translation. Kernel functions also are the underlying reason for some properties concerning the total process, e.g. mean-square continuity and differentiability.

Continuity: Continuity is a property of the Gaussian process, directly exhibited from the choice of kernel function. A stochastic process is continuous, if in a sequence of points $x_1, x_2, ..$ with another fixed point ${}_x* \text{ in } \mathbb{R}^D$ $|\boldsymbol{x}_k - \boldsymbol{x}_*| \to 0$ as $k \to \infty$. Then a process $f(x)$ is continuous in mean-square at x_* if $\mathbb{E}[|f(x_k) - f(x_*)|^2] \to 0$, as $k \to \infty$. If this holds for all $x_* \in A$ where A is a subset of $\mathbb{R}^D$, then $f(x)$ is said to be continuous in mean square over A. A random field is continuous in mean square if and only if its covariance function is continuous at $x = x_* = x'$. For stationary covariance funtions this reduces to checking continuity at $k(0)$.

Differentiability: Using the mean square derivative of $f(x)$ in ith direction

$$\frac{\partial f(x)}{\partial x_i} = \lim_{h \to 0} \frac{f(x + h\boldsymbol{e}_i) - f(x)}{h}, \tag{2.56}$$

and checking if the limit exists for order $2k$ and is finite at $x = 0$, then the kth order limit exists as a mean square limit. Here, the properties of the kernel around 0 determine the smoothness properties of a stationary process.

Following, some of the more prominent kernel functions are presented, with some practical properties. Almost all of the following are stationary and non-degenerate, meaning they are stationary and of infinite rank.

$$k_{linear}(\boldsymbol{x_i}, \boldsymbol{x_j}) = \sigma^2 \boldsymbol{x_i}^\top \cdot \boldsymbol{x_j} \tag{2.57}$$

is the linear kernel. As the most simple of kernels, it is as an exceptional case in this listing neither stationary, nor non-degenerate.

$$k_{exp}(\boldsymbol{x_i}, \boldsymbol{x_j}) = exp(-\frac{1}{2l} d_{ij}), \tag{2.58}$$

the exponential kernel, where $d_{ij} = ||\boldsymbol{x}_i - \boldsymbol{x}_j||_2$ is the Euclidian distance between the input arguments.

$$k_{sqe}(\boldsymbol{x_i}, \boldsymbol{x_j}) = exp(-\frac{1}{2l^2}d_{ij}^2) \tag{2.59}$$

is the squared exponential kernel. It is also known as radial basis function.

$$k_{mat32}(\boldsymbol{x_i}, \boldsymbol{x_j}) = \left(1 + \frac{\sqrt{3}d_{ij}}{l}\right)exp\left(-\frac{\sqrt{3}}{2l}d_{ij}\right) \tag{2.60}$$

is the Matern3/2 kernel. it is a special case like the Matern5/2 kernel

$$k_{mat52} = \left(1 + \frac{\sqrt{5}d_{ij}}{l} + \frac{5r^2}{3l^2}\right)exp\left(\frac{\sqrt{5}d_{ij}}{l}\right) \tag{2.61}$$

of the Matern class of kernels

$$k_{mat} = \frac{2^{1-\nu}}{\Gamma(\nu)}\Big(\frac{\sqrt{2\nu}d_{ij}}{l}\Big)^{\nu} K_{\nu}\Big(\frac{\sqrt{2\nu}d_{ij}}{l}\Big). \tag{2.62}$$

The matern kernels can be constructed with positive parameter ν, K_ν is a modified Bessel function. Also, as $\nu \to \infty$ the function becomes the smooth squared exponential kernel, for $\nu = 1/2$ the Matern class kernel becomes the Ornstein-Uhlenbeck kernel. This kernel gives rise to a continuous-time AR(p) Gaussian process. But since this kernel was not used in the further work, it will not be discussed more. Kernel functions are used to calculate the entries of the covariance matrices. Different kernel functions apply better or worse to different problems. Even though there sometimes are kernel function choices that seem more applicable to a problem from prior assumptions, different kernel functions should be used and compared to ensure better models.

On another note, novel kernels can be created through different methods, for example by combining different existing kernels. Since kernels are always independent, kernels can be added to create a new kernel. For the same reason the multiplication of two kernels also produces a new kernel. In addition, new kernels can be created using rescaling, where kernels are e.g. normalized, or through convolution, where the kernels are mapped onto other spaces. Another major part of the properties of kernel functions are hyperparameters. These influence for example the frequency of changes exhibited by a kernel function. This hyperparameter, usually denoted as l is often called kernel lengthscale. An example, of how this parameter influences the model is shown in figure **??**. The variances, often denoted by σ_n for the noise variance, and σ_f for the kernel function variance, influence the confidence interval in which we assume the functions to lie. σ_n stems from the noise we attribute to the special systemic noise of the problem at hand, while σ_f stems from just observing data that is noisy. A large noise variance and kernel variance will exhibit the kernel matrix incorporating a larger error tolerance. This can be seen in the enlargening of the confidence intervals on the draws of functions from the same process. Higher Variances lead to more flexible fitting, while lower variances will be more rigid. On the other hand, if variances are too high or too low, essential information about the structure of the problem can get lost.

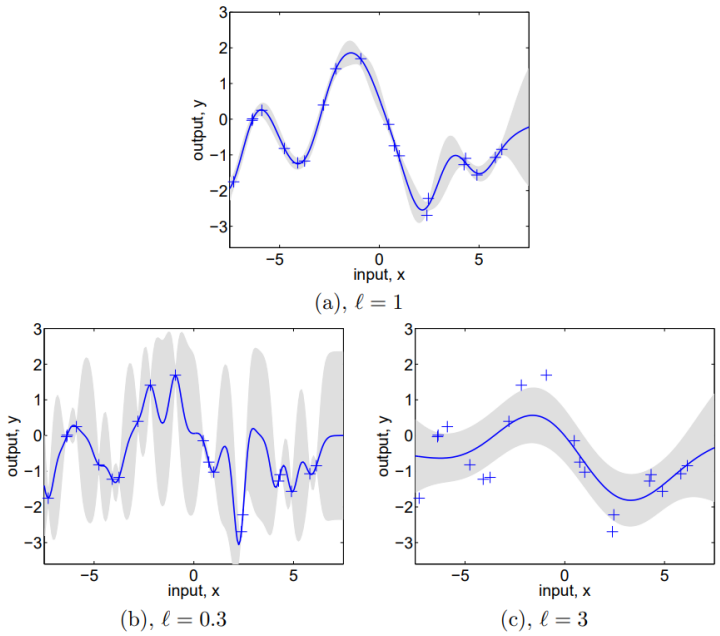


Figure 2.7: Comparison of hyperparameters for the squared exponential kernel function. A large lengthscale exhibits properties where the function does not react to changes quickly, while a small lengthscale gives the opportunity to react to changes almost immediately. While small lengthscales usually lead to better fits, the prediction power often decreases since more information from noise is incorporated into the covariance matrix. The data is generated from a GP with $l_a = 0.3$, $l_b = 1$, $l_c = 3$, $\sigma_f = 1$, and $\sigma_n = 0.1$ [23]. The panel a has lengthscale l_a, panel b has lengthscale l_b, and panel c l_c.

2.3 Gaussian Process Latent Variable Models

Latent variables are variables of an experiment or theory, that influence the outcome of an experiment without being directly measurable. The only way to obtain a latent variable is to infer it from the information that can be observed in the experiment. Intelligence, for example, is a latent variable. It is not directly measurable, but is influencing the outcome of some experiments in psychology. While tests for intelligence try to measure intelligence by inferring it, they can only give a general direction of the true value, assuming it exists and can be ascertained to a single value in a controlled environment. The test then tries to measure what we believe to be components of intelligence, e.g. skills in logic, pattern recognition and grammar. The measured scores are then used to calculate an overall score, which we believe to be a measure for intelligence. While latent variable models, such as the one used in psychology to infer intelligence from a test, enable scientists to more accurately describe phenomena we already have a general understanding about, the models also enable us to infer data in other fields. Since latent variable models are already directly connected to statistics, it is not a far stretch to apply some of them in fields that previously used almost exclusively statistical methods to describe phenomena. Also, latent variable models could reveal relations between different variables of the system, if different variables are dependent on the same set of latent variables, and therefore change with a behaviour that can be infered by knowing the state of the latent variables.
Assume that a data matrix $Y \in \mathbb{R}^{N \times D}$ is given and the goal is to find a lower dimensional representation $X \in \mathbb{R}^{N \times Q}$ of the matrix, without loosing a major share of the information encoded in the data matrix. Previous models have mostly used Principal Component Analysis (PCA) to achieve this dimensionality reduction. Principal Component Analysis has also been shown to be the maximum likelihood solution to a particular form of Gaussian Latent Variable Model [33][34]. PCA here embeds $\mathcal{Y}$, the data space, via linear mapping into a latent space $\mathcal{X}$. Later on, the Gaussian Process Latent Variable Model was introduced as a non-linear extension to probabilistic PCA [22].

Probabilistic PCA (P-PCA) is a model derived by using the probabilistic framework. While P-PCA facilitates statistical testing and Bayesian workflows, it can also be applied to problem sets, where PCA previously could not be used efficiently. This is especially the case for missing data in the dataset. Also, P-PCA can be represented as a general Gaussian density model, which can efficiently be solved through computing maximum likelihood estimates for the parameters associated with the covariance matrix from principal components. This promises effective classification and novelty detection. Starting with factor analysis, a linear relation between the observation vector $\boldsymbol{t} \in \mathbb{R}^d$ and the vector of latent variables $\boldsymbol{x} \in \mathbb{R}^q$, $d > q$ can be established like

$$\boldsymbol{t} = W\boldsymbol{x} + \boldsymbol{\mu} + \boldsymbol{\epsilon} \tag{2.63}$$

with noise $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$ and mean vector $\boldsymbol{\mu}$ to permit the model with a non-zero mean. It promises a parsimonious explanation of dependencies between observations. If $\boldsymbol{x}$ is assumed

to be standard Gaussian, we can find the distribution for t

$$\boldsymbol{t} \sim \mathcal{N}(\boldsymbol{\mu}, WW^\top + \Sigma). \tag{2.64}$$

Now, when constraining the covariance matrix Σ to be a diagonal matrix, we will have only autocorrelation in observed variables (conditional independence), given the latent variables $\boldsymbol{x}$ and the mapping W. The latent variables are therefore used to explain correlations between observation variables.
With this, we can find P-PCA by considering the $\boldsymbol{x}$-conditional probability distribution over $\boldsymbol{t}$-space

$$\boldsymbol{t}|\boldsymbol{x} \sim \mathcal{N}(W\boldsymbol{x} + \boldsymbol{\mu}, \sigma^2 I) \tag{2.65}$$

Integrating out the latent variables, we arrive at the Gaussian marginal distribution for $\boldsymbol{t}$

$$\boldsymbol{t} \sim \mathcal{N}(\boldsymbol{\mu}, WW^\top + \sigma^2 I) \tag{2.66}$$

with corresponding log-likelihood

$$\log p(\boldsymbol{t}) = -\frac{N}{2}\Big(d\ln(2\pi) + \ln|WW^\top + \sigma^2 I| + \mathrm{tr}((WW^\top + \sigma^2 I)^{-1}\frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{t}_n - \boldsymbol{\mu})(\boldsymbol{t}_n - \boldsymbol{\mu})^\top)\Big). \tag{2.67}$$

Iterative maximisation of the log-likelihood now presents a fast and reliable way to optimise the P-PCA. From this model, standard PCA can be recovered when $\sigma^2 \to 0$ and $WW^\top + \sigma^2 I \to (W_{ML}^\top W_{ML})$.

Considering the GPLVM, we can find the generating procedure to be

$$Y_{n,:} = \boldsymbol{f}(X_{n,:}) + \boldsymbol{\epsilon}_n, \tag{2.68}$$

closely resembling the linear model of a GP, equation 2.63 again. Here, $\boldsymbol{f} = f(f_1, ..., f_D)$ is a group of D independent samples from a Gaussian process $f_d \sim \mathcal{GP}(0, k(\cdot,\cdot))$. X is the matrix containing the latent space ($\mathcal{X}$) positions. Due to the structure of the problem at hand, the rows of the data matrix Y are assumed to be jointly Gaussian distributed, while the columns are independent. This can be interpreted as every data point (return rate) of a single day being Gaussian distributed, while there is no time-dynamic. Each sample of $Y_{:,d} \sim \mathcal{N}(Y_{:,d}|\boldsymbol{0}, K)$ is assumed to incorporate noise inside the covariance matrix $K = k(X,X) + \sigma^2 I$, where σ^2 is the variance of the noise $\boldsymbol{\epsilon}$, which is assumed to be random. We find the marginal likelihood of Y to be

$$p(Y|X) = \prod_{d=1}^{D}\mathcal{N}(Y_{:,d}|\boldsymbol{0}, K) = \frac{1}{(2\pi)^{ND/2}|K|^{D/2}}\exp\left(-\frac{1}{2}tr(K^{-1}YY^\top)\right). \tag{2.69}$$

The covariance matrix K gives the dependency on the kernel hyperparameters as well as the latent space positions X. Lawrence (2005) suggested to optimize the log-marginal-likelihood $\log p(Y|X)$, with the hyperparameters and latent space positions as the parameters used for optimisation.

2.3.1 Student-t-Process

An alternative to the Gaussian processes provide the Student-t-processes [35]. They can be regarded as a generalized Gaussian process. Assuming an inverse Wishart prior for the kernel of a Gaussian process will result in a Student-t process. Due to this, we can also think of the Student-t process as a prior over functions, that is non-parametric. Also belonging to the family of elliptical processes, the Student-t process ($\mathcal{TP}$) offers more robustness to outliers inherent to the process, due to the broader flanks of the Student-t distribution. It is the most general elliptically symmetric process for which analytical marginal and predictive distributions exist. While some argue [35], that the predictive covariances of a Gaussian process do not depend on the training observations, the predictive covariances from a Student-t process do depend on training observations. Deriving the Student-t process, we will start with a Wishart process $W_n(\nu, K)$. The Wishart distribution is a distribution over the set of real-valued symmetric matrices, of size $n \times n$, that are positive definite $\Pi(n)$. The density function is

$$\Sigma \sim W_n(\nu, K), \text{ if} \tag{2.70a}$$

$$p(\Sigma) = \Big(|K|^{\nu/2}2^{\nu n/2}\Gamma_n(\nu/2)\Big)^{-1}|\Sigma|^{(\nu-n-1)/2}exp\Big(-\frac{1}{2}tr(K^{-1}\Sigma)\Big) \tag{2.70b}$$

with $\nu > n-1,\ \nu \in \mathbb{R}_+$. This definition exhibits the marginalisation property, just like a GP. But since for $\nu \to \infty$ almost surely $\nu^{-1}\Sigma \to K$, thereby loosing the usefulness of the process. This property is not exhibited in the inverse Wishart process ($Iw_n(\nu, K)$) [36].

$$\Sigma \sim IW_n(\nu, K) \text{ if} \tag{2.71a}$$

$$p(\Sigma) = \Big(\frac{|K|^{(\nu+n-1)/2}}{2^{(\nu+n-1)n/2}\Gamma_n((\nu+n-1)/2)}\Big)^{-1}|\Sigma|^{-(\nu+n-1)/2}exp\Big(-\frac{1}{2}tr(K\Sigma^{-1})\Big) \tag{2.71b}$$

This formulation requires $\nu > 2$ and $\mathbb{E}[\Sigma] = (\nu-2)^{-1}K$, for the mean and the covariance to exist. Both Wishart distributions place prior mass on every possible matrix Σ stemming from the set $\Pi(n)$. It was also previously shown, that the Inverse Wishart distribution is consistent under marginalization. Any submatrix Σ_{11} will be $Iw_{n1}(\nu_1 K - \Sigma_{11})$ distributed. With this, we can define a Inverse Wishart Process analogous to a Gaussian process with a base kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$

$$\sigma \sim \mathcal{IWP}(\nu, k(\cdot,\cdot)). \tag{2.72}$$

If the Inverse Wishart process is applied as prior on the kernel function of a $\mathcal{GP}$, a Student-t process

$$\sigma \sim \mathcal{IWP}(\nu, k_\theta) \tag{2.73a}$$

$$\boldsymbol{y}|\sigma \sim \mathcal{GP}(\mu, (\nu-2)\sigma) \tag{2.73b}$$

generative approach is achieved. We find the data to be Student-t distributed if the density is of the structure

$$\boldsymbol{y} \sim \mathcal{MVT}_n(\nu, \boldsymbol{\mu}, K) \tag{2.74a}$$

$$p(\boldsymbol{y}) = \frac{\Gamma(\frac{\nu+n}{2})}{((\nu-2)\pi)^{\pi/2}\Gamma(\nu/2)}|K|^{-1/2}\Big(1 + \frac{(\boldsymbol{y}-\boldsymbol{\mu})^\top K^{-1}(\boldsymbol{y}-\boldsymbol{\mu})}{\nu-2}\Big)^{-(\nu+n)/2}, \tag{2.74b}$$

for which the mean and covariance are given by

$$\mathbb{E}[\boldsymbol{y}] = \mathbb{E}[\mathbb{E}[\boldsymbol{y}|\Sigma]] = \boldsymbol{\mu} \tag{2.75a}$$

$$cov[\boldsymbol{y}] = \mathbb{E}[\mathbb{E}[(\boldsymbol{y} - \boldsymbol{\mu})(\boldsymbol{y} - \boldsymbol{\mu})^\top|\Sigma]] \tag{2.75b}$$

[35]. The Student-t process is consistent under marginalisation. As a shorthand, we write $f \sim \mathcal{TP}(\nu, \boldsymbol{\mu}, K)$ for a joint Student-t process over a finite collection of function values, as a similar counterpart to the Gaussian process over function values 2.45. The student-t process generalizes the Gaussian process through the introduction of the degree of freedom parameter ν. For $\nu \to \infty$ we arrive at a Gaussian process, while at $\nu = 1$ we have a Cauchy process. Analyzing this hyperparameter, we find that it controls how much probability mass lies in the tails of a distribution, directly influencing how large of a variance samples from a Student-t process have compared to a Gaussian process. In figures 2.8 and 2.9, the

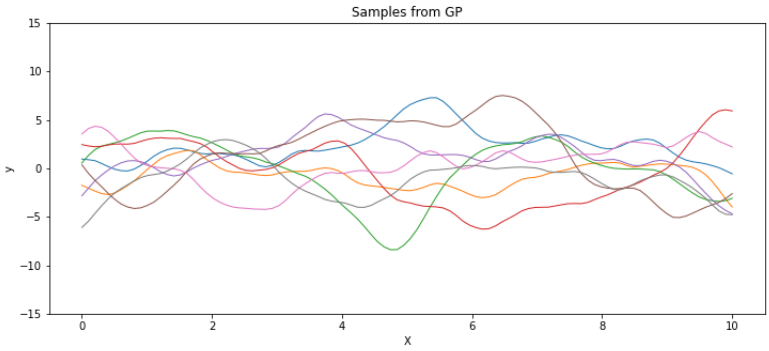


Figure 2.8: Samples from a GP.

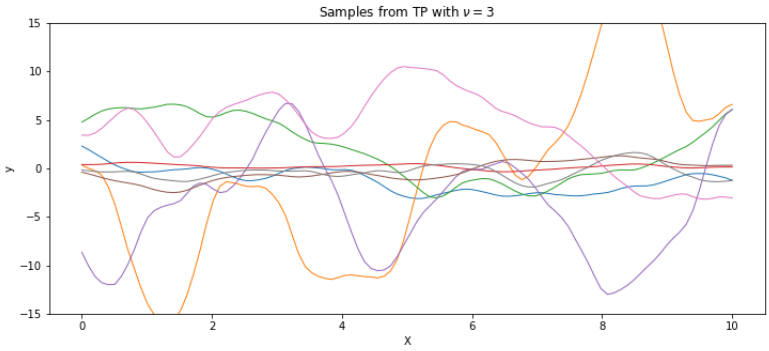


Figure 2.9: Samples from a TP.

comparison of Gaussian process prior samples (upper) and Student-t (lower) process prior samples are shown. In general the Student-t samples have a broader credibility predictive sample interval compared to the Gaussian process. This leads to a higher tolerance for outliers in the data sets, since the outliers do not impact the covariance as much as with a Gaussian process. A higher parameter ν leads to a more narrow credibility interval. After

analysing the properties of a Student-t process prior, the conditional distributions can be analyzed. Splitting up a data set into e.g. a test ($\boldsymbol{y}_*$) and training ($\boldsymbol{y}$) dataset, the respective multivariate Student-t process becomes

$$\begin{aligned} \boldsymbol{y}_*|\boldsymbol{y} \sim \mathcal{MVT}_{n_*}(\nu+n, K(X_*,X)K(X,X)^{-1}(\boldsymbol{y}-\boldsymbol{\mu})+\boldsymbol{\mu}_*, \\ \frac{\nu+(\boldsymbol{y}-\boldsymbol{\mu})^\top K(X,X)^{-1}(\boldsymbol{y}-\boldsymbol{\mu})-2}{\nu+n-2}(K(X_*,X_*)-K(X_*,X)K(X,X)^{-1}K(X,X_*))) \end{aligned} \tag{2.76a}$$

with number of datapoints n_* in $\boldsymbol{y}_*$ and n in $\boldsymbol{y}$ respectively.

2.4 Algorithms for stochastic problem computation

Markov Chain Monte Carlo (MCMC) methods [37] are a class of algorithms that are used to sample a probability distribution that is analytically intractable, or just very complex. A Markov Chain is constructed, that has the same distribution as the target distribution, and recording the chain, the target distribution is uncovered. Since MCMC methods only uncover the true target distribution for very large chain lengths, or even infinite length, a practical convergence criterion is needed to numerically approximate the target multidimensional distributions. We know, that target distributions are calculated via solving integrals of the form 2.28, so MCMC methods are a likely choice to base further evaluations on. An example of this is depicted in the following figure, where an algorithm is shown that lets the test distribution converge towards the target distribution using the metropolis hastings algorithm. A necessary precondition is that the probability density of the target random variable

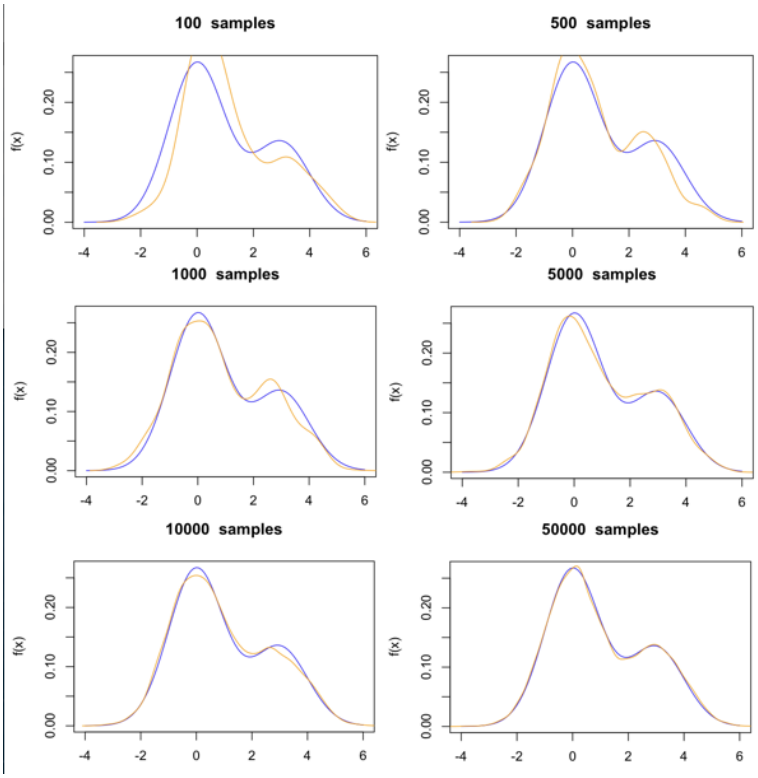


Figure 2.10: Convergence of the Metropolis Hastings algorithm as an example for Markov chain Monte Carlo methods approximating the target distribution of a bayesian problem. It is clear that more samples lead to a better approximation of the target distribution.

is given up to a constant. If the integral is intractable, MCMC methods can evaluate the integral statistically by generating an ensemble of chains starting at arbitrarily chosen points. A chain is a stochastic process, that moves around randomly on this high dimensional probability surface where moves are generated via Monte Carlo sampling, and then an evaluation

of the contribution of this place to the integral is carried out. If the posterior mass is low in some regions, the chain will move away faster from this location, while on the other hand it will spend more moves in regions with higher posterior mass. These Markov chains then have an equilibrium distribution proportional to the target distribution. MCMC methods, even though outperforming generic Monte Carlo algorithms, still suffer from the curse of dimensionality, where regions of higher probability tend to stretch due to a high number of dimensions. Regions with higher probability mass then do not get the proportion of moves necessary in the faster scaling volume of high dimensional space, and contribute less to the solution. This method also implies a problem of when to accept a chain to be converged to a stationary distribution with a reasonable error. Here, the term of rapid mixing becomes important, describing the phenomenon of stationary distributions being reached quickly while starting from arbitrary positions, highlighting the need to sample several chains for non-variational inference algorithms. An extension of the central limit theorem, the Markov chain central limit theorem [38], states that for a sequence of random elements forming a Markov chain with the Markov property, the mean of the Markov chain will tend towards a normal distribution. Formally, we have a sequence of random elements $X_1, X_2, X_3, ...$ of some set $\mathcal{X}$ forming the Markov chain with a stationary probability distribution. Also, the initial distribution of the process, the distribution leading to X_1 has to be stationary, which entails that $X_1, X_2, X_3, ...$ are identically distributed. Then, the Markov property is necessary, and we need a measurable real-valued function g with $var(g(X_1)) < +\infty$.

$$\mu = \mathbb{E}(g(X_1)), \tag{2.77a}$$

$$\sigma^2 = var(g(X_1)) + 2\sum_{k=1}^{\infty} cov(g(X_1), g(X_{1+k})), \tag{2.77b}$$

$$\hat{\mu}_n = \frac{1}{n}\sum_{k=1}^{n} g(X_k). \tag{2.77c}$$

For $n \to \infty$, we find:

$$\sqrt{n}(\hat{\mu_n} - \mu) \to \mathcal{N}(0, \sigma^2), \tag{2.78}$$

resulting in an important feature for convergence. Furthermore, a solution to the curse of dimensionality would be to accept higher autocorrelation and expensive computations by using smaller steps of the chain, which is impractical. More sophisticated methods, like Hamiltonian Monte Carlo reduce autocorrelation while keeping the chain in the desired regions of the probability surface.

Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) [39] is a MCMC method that applies derivatives of the density function to generate efficient transitions when approximating the posterior. The dynamics of the chain is given by Hamiltons equations of motion. The Hamiltonian dynamics simulation is carried out using numerical integration. To achieve this, HMC uses auxiliary momentum variables to form a joint density to draw from.

$$p(\rho, \theta) = p(\rho|\theta)p(\theta), \tag{2.79}$$

where ρ are the auxiliary momentan variables, and θ are the parameters of the model. By choosing the following distribution for ρ, it becomes independent of θ.

$$\rho \sim \mathcal{N}(0, M). \tag{2.80}$$

Here, M is the Euclidian metric, which is just a transformation of parameter space to enhance the efficiency of sampling. The joint density of auxiliary and regular parameters then leads to the combined Hamiltonian

$$H(\rho, \theta) = -log(p(\rho, \theta)) = -log(p(\rho|\theta)) - log(p(\theta)) = T(\rho|\theta) + V(\theta), \tag{2.81}$$

where we define $T(\rho|\theta)$ and $V(\theta)$ as kinetic energy and potential energy respectively, directly analogous to physics. in statistics we find that $T(\rho|\theta) = -log(p(\rho|\theta))$ and $V(\theta) = log(p(\theta))$. The potential energy is often in a statistical context regarded as a log density. From this on, we can use Hamilton´s equations to find find the momentum needed for state transitions. The momentum parameter values are independently drawn using 2.80, so that momentum is changing across iterations. We use

$$\frac{d\theta}{dt} = \frac{\partial H}{\partial \rho} = \frac{\partial T}{\partial \rho}, \tag{2.82a}$$

$$\frac{d\rho}{dt} = -\frac{\partial H}{\partial \theta} = \frac{\partial T}{\partial \theta} - \frac{\partial V}{\partial \theta} = -\frac{\partial V}{\partial \theta}, \tag{2.82b}$$

where we have recognized the momentum density $\partial T/\partial\theta$ to be zero, due to the independence of momentum density and target density. This two-state differential equation can then be solved using many integrators. We will discuss the Leapfrog integrator here, since it is most widely used in HMC implementations. In general the Leapfrog integrator uses half integer time steps to calculate velocities and integer time steps to calculate positions, which pans out to the following updating equations:

$$\rho \leftarrow \rho - \frac{\epsilon}{2}\frac{\partial V}{\partial \theta}, \tag{2.83a}$$

$$\theta \leftarrow \theta + \epsilon M^{-1}\rho. \tag{2.83b}$$

Here, the first update rule is carried out twice, once at the start of a timestep and once at the end. The symplectic Leapfrog integrator has a $\mathcal{O}(\epsilon^3)$ error per step, and globally an error of $\mathcal{O}(\epsilon^2)$. Following a calculated transition, the Metropolis acceptance step begins, where evaluation of the energy of the Hamiltonian leads to accepting the proposed step, or throwing this set of parameters away. The probability of keeping a step can be calculated via

$$min(1, exp(H(\rho, \theta)) - exp(H(\rho^*, \theta^*))), \tag{2.84}$$

where ρ^*, θ^* are the proposed values and ρ, θ the origin of the proposed transition. We do not have to implement HMC by ourself. We can use probabilistic programming languages such as stan. For that, we onl yghave to define the unnormalized posterior density and the approximation thereof is carried out by stan.

Stochastic Gradient Ascent

Stans variational inference algorithm, Automatic Differentiation Variational Inference [40], optimizes the ELBO in real-coordinate space. The stochastic gradient ascent obtains unbiased, yet noisy gradients through automatic differentiation and evaluates the ELBO using Monte Carlo integration. Along these gradients, the algorithm ascends using a sequence of adaptive stepsizes. Typically, in practice, the ELBO is sampled with around 100 samples, so that the true ELBO value can be approximated with reasonably high confidence. The ELBO is only evaluated every few iterations, to save computation time, but inside high dimensional manifolds, this seems not to be a problem in practice [41]. The gradients are also approximated using Monte Carlo Integration, but usually only one sample is drawn, as experience has shown that, while remaining high computational efficiency, stochastic gradient ascent is capable of following such gradients nonetheless. Adaptive step sizes are optimized during a warmup phase where a good value for the single exposed parameter is selected from values spanning multiple orders of magnitude [41]. This parameter is then used in combination with a finite memory version of adaGrad [42]. In the end, every calculation needs to asses convergence, but since there are no closed form analytical expressions available, the progression of the ELBO values has to be tracked. Using a rolling window, which is heuristically determined, in which the average and median change of the ELBO are computed, we assume convergence if either of those values has fallen below a certain threshold.

3 Methods

3.1 Mathematical Models

This work compares different models on metrics of how good a prediction is on the dataset. Different datasets have been used in this evaluation, to compare the results of different models, as well as different subsets of the S&P500 stocks. The GPLVM, described in section 2.3 is the basis for all the models. All explicit kernel functions will be provided with the example of a squared exponential kernel.

3.1.1 Gaussian Process Latent Variable Model in Finance

?? The GPLVM, as explained earlier in section 2.3, considers the same variance for each stock and also the same noise variance. Stocks, however, have different volatilities, which in turn are displayed in the model as variances. Therefore, we reparametrize the model. Given the data matrix $Y = (\boldsymbol{y}_1, ..., \boldsymbol{y}_N)^\top \in \mathbb{R}^{N \times D}$ and the latent spaces $X = (\boldsymbol{x}_1, ..., \boldsymbol{x}_N)^\top \in \mathbb{R}^{N \times Q}$, the model becomes

$$Y_n \sim \mathcal{N}(0, K), \tag{3.1}$$

where the mean is assumed to be 0, and the covariance matrix to be given by K. The covariance matrix is constructed using the Xs, the time dependent correlation matrix $K_{\text{corr}}(t)$, noise variances σ_i and the variance matrix $\Sigma = \text{diag}(\sigma_i)$.

$$K = \Sigma K_{corr}(d)\Sigma + \sigma_k \mathbb{1} \tag{3.2}$$

An entry of the resulting covariance matrix K looks like

$$k(d_i, d_j) = \alpha^2 exp\Big[-\frac{1}{2l^2}(d_i - d_j)\Big] + \delta_{ij}(\sigma^2 + g), \tag{3.3}$$

where $k(d_i, d_j)$ is the covariance function, α and l are the kernel hyperparameters, σ is the variance and g is a very small value (jitter) added to the diagonal of the matrix, to prohibit singular covariance matrices and thus ensure invertibility. This describes a nonlinear mapping from latent space positions to data space,

$$Y_{:,d} = f_d(X_{:,d}) + \epsilon_{:,d}. \tag{3.4}$$

Here, the function f_d, that is the map, is a $\mathcal{GP}(0, k)$, with a nonlinear kernel function k. The noise term is $\epsilon_{:,d} \sim \mathcal{N}(0, \text{diag}(\boldsymbol{\sigma_n^2}))$, which directly leads us towards accepting different

volatilities for different stocks. We can then decompose the covariance Matrix K into a correlation matrix and diagonal noise matrices.

$$K = \Sigma K_{\text{corr}} \Sigma, \tag{3.5}$$

which, for a stationary kernel becomes

$$K_{\text{stationary}} = \Sigma k_{\text{stationary}}(X, X)\Sigma + k_{\text{noise}}(X, X) \tag{3.6}$$

and the likelihood of the full model is given by

$$p(Y|B, \boldsymbol{\theta}) = \prod_{d=1}^{D} \mathcal{N}(Y_{:,d}|\mathbf{0}, K) = \frac{1}{(2\pi)^{0.5 \cdot ND} K^{0.5 \cdot D}} \exp(-0.5\text{tr}(K^{-1} Y Y^{\top})) \tag{3.7}$$

The generative process for this model then becomes

$$\begin{aligned} l &\sim \text{Inv-Gamma}(3, 1), \\ \alpha &\sim \text{Inv-Gamma}(3, 1), \\ \sigma_n &\sim \mathcal{N}(0, 0.5), \\ \sigma_k &\sim \mathcal{N}(0, 0.5), \\ X[n] &\sim \mathcal{N}(0, 1), \\ Y[d] &\sim \mathcal{N}(0, K). \end{aligned} \tag{3.8a}$$

(3.8b)

3.1.2 Time Dynamic Gaussian Process Latent Variable Model

The Time Dynamic Gaussian Process Latent Variable Model (TD-GPLVM) allows the latent space positions to vary over time. Instead of describing $Y \in \mathbb{R}^{N \times D}$ and $X \in \mathbb{R}^{N \times Q}$, which is fixed in latent space, we increase the dimension of X to $X \in \mathbb{R}^{N \times Q \times D}$, where $X_d \in \mathbb{R}^{N \times Q}$ and $X_{n,q,:} \in \mathbb{R}^{D}$. $X_{n,q,:}$ then follows a Gaussian Process.

$$X_{n,q,:} \sim \mathcal{N}(0, K_{DD}) \tag{3.9}$$

The computational complexity of this process is of order $\mathcal{O}(D^3)$, because of the inversion of $K_{DD} \in \mathbb{R}^{D \times D}$. Therefore, only a few days of observations can be modelled simultaneously. The idea of this model is to incorporate a time dependent correlation of latent variables, which may exist for certain datasets. In matrix notation, the covariance matrices are formed through the quadratic diagonal form of the time dependent correlation matrix $K_{corr}(t)$, noise variance σ_n and variance matrix $\Sigma = \text{diag}(\sigma_i)$.

$$K = \Sigma K_{corr}(d)\Sigma + \sigma_n \mathbb{1} \tag{3.10}$$

An entry of the resulting covariance matrix K_x looks like

$$K_{ij} = k(x_i, x_j) = \alpha_i \alpha_j exp\Big[-\frac{1}{2l^2}(d_i - d_j)\Big] + \delta_{ij}(\sigma_i^2 + g), \tag{3.11}$$

where $k(x_i, x_j)$ is the covariance function, $\alpha_i\alpha_j$ and l are the kernel hyperparameters, σ_i are the variances and g is a very small value (jitter) added to the diagonal of the matrix, to prohibit singular covariance matrices and thus ensure invertibility. The inputs to the covariance function are d_i and d_j, the ith or jth entry of a vector containing ordered timesteps $\boldsymbol{d} = (d_1, d_2, d_3, ..., d_D)^\top$. The covariance function of K_y then becomes

$$k_y(x_i[d], x_j[d]) = \alpha_i\alpha_j exp\Big[-\frac{1}{2l_y^2}(x_i[d] - x_j[d])\Big] + \delta_{ij}(\sigma_i^2 + g) \tag{3.12}$$

where again, $k_y(x_i[d], x_j[d])$ is the kernel function, $\alpha_i\alpha_j$ and l_y^2 are the kernel hyperparameters, and σ_y is the variance. Also, g continues to be a very small value added as jitter to the diagonal of the covariance matrix to prevent singularity and $x_i[d]$ and $x_j[d]$ are the positions of asset i and j at time point d in latent space $\mathcal{X}$. After conditioning the model was parametrised by

$$\begin{aligned} l &\sim \text{gamma}(10, 5) \\ l_y &\sim \text{inv-gamma}(3, 1) \\ \sigma_k &\sim \mathcal{N}(0, 0.5) \\ \sigma_n &\sim \mathcal{N}(0, 0.05) \\ X[n, q, :] &\sim \mathcal{N}(\mathbf{0}, K_x) \\ Y[:, d] &\sim \mathcal{N}(\mathbf{0}, K_y[d]) \end{aligned}$$

the stan code can be found in Appendix **??**, as *gplvm-time.stan*.

3.1.3 Volatility Gaussian Process Latent Variable Model

Another model that may solve problems of simpler models is the Volatility Gaussian Process Latent Variable Model (V-GPLVM). The V-GPLVM works similar to the regular GPLVM, but the diagonal matrix with the variances, which work as a placeholder for the proportional volatility, is time dependent.

$$K_d = \text{diag}\Big(\Sigma(:, d)\Big) K_{corr}\ \text{diag}\Big(\Sigma(:, d)\Big) + \text{diag}(\sigma_i) \tag{3.14}$$

Again, σ_n are the noise variances, and K_{corr} is the correlation matrix. $\text{diag}(\Sigma(:, d))$ is the time dependent diagonal variance matrix, which will be inferred throughout the process. While evaluating the programm, these matrices are constructed from columns of the total variance matrix $\Sigma \in \mathbb{R}^{N\times D}$, representing the inferred values for variances for stocks on a given time step. The latent space Gaussian Process prior is just a standard normal distribution $\mathcal{N}(0, 1)$, while the Gaussian Process over the data space if more complex.

$$K[d]_{ij} = \Sigma[d, i]\Sigma[d, j]\left(exp\Big(-\frac{1}{2l^2}(x_i - x_j)\Big)\right) + \delta_{ij}(\sigma_k^2 + g) \tag{3.15}$$

After conditioning the model priors were parametrised like

$$\begin{aligned} X[n] &\sim \mathcal{N}(0,1) \\ \log \Sigma[d] &\sim \mathcal{N}(0,1) \\ \sigma_k &\sim \mathcal{N}(0,1) \\ l &\sim \text{inv-gamma}(3,1) \\ Y[:,d] &\sim \mathcal{N}(\mathbf{0}, K[d]), \end{aligned}$$

where K is evaluated after Cholesky factorising the kernel matrix slices $K[d] \in \mathbb{R}^{N \times Q}$ through Cholesky decomposition $K = LL^*$, where the matrix is decomposed into a lower triangular matrix L. Using the Cholesky Decomposition reduces computing time, while maintaining numerical stability. The method is more stable and cost efficient for matrix inversion, which, as previously discussed, is the major bottleneck for computing time, since using the decomposition reduces the needed number of evaluations to half. The stan code can be found in Appendix **??** as *gplvm-vola.stan*.

3.2 Programs

3.2.1 Programm Sequence

As shown in the program sequence overview **??**, the process executed starts with a second programm, decoupled from the acutal calculations, downloading stock market data from yahoo finance [44] with previously specified tickers from the S&P500 index over an also specified time period, where the full time period is 2010/01/01-2013/01/01. In the calculation sequence a dataset is downloaded, and lists of values for latent dimensions (Qs), kernel functions (kernels), and different models are specified. With these, the models written in stan are compiled by the stan model compile function, and saved as a pickled model. This model is then executed for all combinations of Qs and kernels using the variational bayes algorithm. Variational Bayes outputs diagnostic information of the model, and samples for all parameters specified in either the parameter block or the generated quantities block of the stan model. These are both saved as files for later checking of the models diagnostic, through both the diagnostic information, most notably ELBO values from the variational inference, and the samples, from which e.g. R^2 values can be calculated. With this, different sanity checks are applied, like plotting the true measured values against the predictions calculated from the models learned structure, checking the values from different sampled variables agains the expectations. All of these will be discussed in detail in the results section 4.5.

3.2.2 The stan language

The stan language was referenced a few times before, in this section a little more light will be shed on how to apply it. To recap, stan [45] is a program that compiles an efficient $c++$ program from a textfile in which in the stan language a statistical model is specified. This text file is comprised of up to 9 blocks, in which different parts of the model are specified and whose compiled program parts are executed at different times during the execution of the compiled program. This will be henceforth referenced to as "evaluation of a block". At first, the data block is evaluated. It specifies the variables that are fixed, and are handed to the compiled program for execution. Typical entries are a number of which kernel function to use, the data matrix Y, as well as some initially fixed variables like a jitter or the number of dimensions of the latent space Q. This block can be preceded by a functions block, which is not needed for execution, but can be convenient when the model has some parts of code that need to be executed several times. After the data block, a block called transformed data can be introduced, in which constants and transforms of the content of the data block can be specified. This can be used to e.g. create null-vectors, which in a lot of models are used as starting parameters of a mean. Then follows the parameters block, which is filled with variables that are to be inferred, reckoning sampled or optimized, during the execution of the program. mPosterior distributions of variables specified as parameters of the model in the parameters block are approximated during variational inference. The parameters can be transformed in the transformed parameters block. No statements are allowed in the previous blocks, except for transformed data and transformed parameters

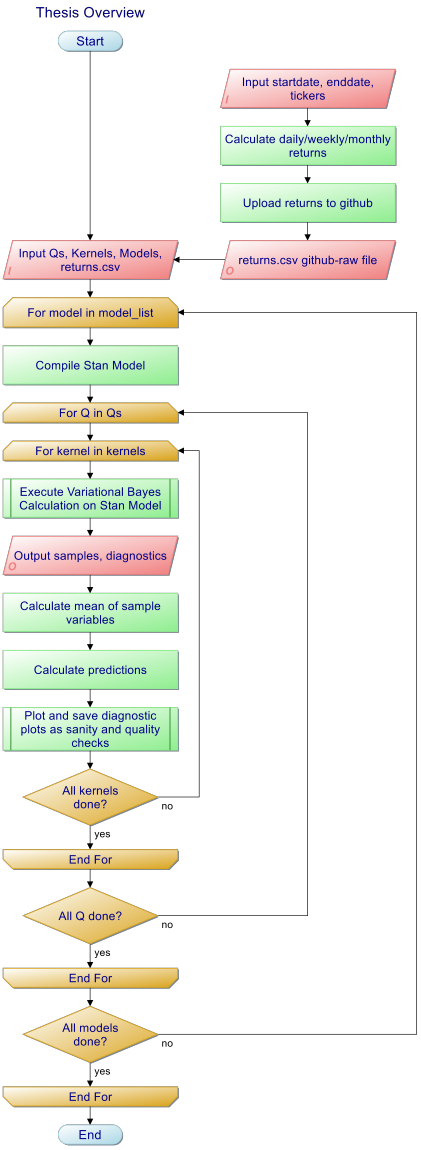


Figure 3.1: Program Sequence overview according to DIN 66001 [43].

block. In the model block, the log of the unnormalized posterior density is specified, and the sampling statements of parameters are selected. Finally, the generated quantities block allows for the creation of derived variables, containing the variables from the parameters block.

Formally, all blocks are optional, but a typical sensible program will have at least a data block, parameters, and a model block. This ordering of the blocks is necessary, and disobeying it will break the program. In the following listing, a simple Gaussian process is depicted [41].

```
data {
    int<lower=1> N;
    real x[N];
}
transformed data {
    matrix[N, N] K = cov_exp_quad(x, 1.0, 1.0);
    vector[N] mu = rep_vector(0, N);
    for (n in 1:N)
    K[n, n] = K[n, n] + 0.1;
}
parameters {
    vector[N] y;
}
model {
    y ~ multi_normal(mu, K);
}
```

Variables declared in a program block will have scope over all blocks, in a way be global, and functions may be used in all appropriate blocks. Exceptions to this rule are, that variables from the model block are local, and variables declared in a later block are not yet known to an earlier block. Functions that generate random numbers are limited to the transformed data and generated quantities block, and functions that modify log probability may only be used in the transfomred parameters and model block. This stan language, with a syntax close to that of other high-level programming languages like *python* or *c++*, is comparably easy to write and learn, while compiling very efficient programs. Writing down the model in mathematical terms almost directly translates to the stan model, which makes application of stan very practical, especially for users without intensive training in computer science [41]. A compiled stan model can be used different ways. One can optimize the defined log posterior density, approximate their posterior using variational inference or sample from the posterior using Hamiltonian Monte Carlo. In the following figure, a more complex model is shown, which is referred to as GPLVM throughout this work.

```
functions {
    matrix cov_linear(vector[] X1, vector[] X2, real sigma){
        int N = size(X1);
        int M = size(X2);
        int Q = num_elements(X1[1]);
        matrix[N,M] K;
        {
            matrix[N,Q] x1;
            matrix[M,Q] x2;
            for (n in 1:N)
                x1[n,] = X1[n]';
            for (m in 1:M)
                x2[m,] = X2[m]';
            K = x1*x2';
        }
        return square(sigma)*K;
    }

    matrix cov_matern32(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        for (n in 1:N)
            for (m in 1:M){
                dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
```

```
27                  K[n,m] = square(sigma)*(1+sqrt(3)*dist/l)*exp(-sqrt(3)*dist/l);
28          }
29          return K;
30      }
31
32      matrix cov_matern52(vector[] X1, vector[] X2, real sigma, real l, real jitter){
33          int N = size(X1);
34          int M = size(X2);
35          matrix[N,M] K;
36          real dist;
37          for (n in 1:N)
38              for (m in 1:N){
39                  dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
40                  K[n,m] = square(sigma)*(1+sqrt(5)*dist/l+5*square(dist)/(3*square(l)))*exp(-sqrt(5)*dist/l);
41          }
42          return K;
43      }
44
45      matrix cov_exp_l2(vector[] X1, vector[] X2, real sigma, real l, real jitter){
46          int N = size(X1);
47          int M = size(X2);
48          matrix[N,M] K;
49          real dist;
50          for (n in 1:N)
51              for (m in 1:M){
52                  dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
53                  K[n,m] = square(sigma) * exp(-0.5/l * dist);
54          }
55          return K;
56      }
57
58      matrix cov_exp(vector[] X1, vector[] X2, real sigma, real l, real jitter){
59          int N = size(X1);
60          int M = size(X2);
61          matrix[N,M] K;
62          real dist;
63          int Q = rows(X1[1]);
64          for (n in 1:N)
65              for (m in 1:M){
66                  dist = 0;
67                  for (i in 1:Q)
68                      dist = dist + fabs(X1[n,i] - X2[m,i]);
69                  K[n,m] = square(sigma) * exp(-0.5/l * dist);
70          }
71          return K;
72      }
73
74      matrix kernel_f(vector[] X1, vector[] X2, real sigma, real l,
75      real a, int kernel, vector diag_stds, real jitter){
76          int N = size(X1);
77          int M = size(X2);
78          matrix[N,M] K;
79          if (kernel==1)
80          K = cov_linear(X1, X2, a);
81          else if (kernel==2){
82              K = cov_exp_quad(X1, X2, sigma, l);
83              for (n in 1:N)
84              K[n,n] = K[n,n] + jitter;
85              K = quad_form_diag(K, diag_stds);
86          }
87          else if (kernel==3){
88              K = cov_exp(X1, X2, sigma, l, jitter);
89              K = quad_form_diag(K, diag_stds);
90          }
91          else if (kernel==4){
92              K = cov_matern32(X1, X2, sigma, l, jitter);
93              K = quad_form_diag(K, diag_stds);
94          }
95          else if (kernel==5){
96              K = cov_matern52(X1, X2, sigma, l, jitter);
97              K = quad_form_diag(K, diag_stds);
98          }
99          return K;
100     }
101 }
102 data {
103     int<lower=1> N;
104     int<lower=1> D;
105     int<lower=1> Q;
106     matrix[N,D] Y;
107     int<lower=1,upper=5> kernel;
108     real<lower=0> jitter;
109 }
110 transformed data {
111     vector[N] mu = rep_vector(0, N);
112 }
113 parameters {
114     vector[Q] X[N];
115     real<lower=0> kernel_lengthscale;
```

```
116     vector<lower=0>[N] diag_stds;
117     vector<lower=0>[N] noise_std;
118     real<lower=0> alpha;
119 }
120 transformed parameters {
121     matrix[N,N] L;
122     real R2 = 0;
123     {
124         matrix[N,N] K = kernel_f(X, X, 1., kernel_lengthscale,
125         alpha, kernel, diag_stds, jitter);
126 
127         for (n in 1:N)
128         K[n,n] = K[n,n] + pow(noise_std[n], 2) + jitter;
129         L = cholesky_decompose(K);
130 
131         R2 = sum(1 - square(noise_std) ./diagonal(K) )/N;
132     }
133 }
134 model {
135     for (n in 1:N)
136     X[n] ~ normal(0, 1);
137 
138     diag_stds ~ normal(0, .5);
139     noise_std ~ normal(0, .5);
140     kernel_lengthscale ~ inv_gamma(3.0,1.0);
141     alpha ~ inv_gamma(3.0, 1.0);
142 
143     for (d in 1:D)
144     col(Y,d) ~ multi_normal_cholesky(mu, L);
145 }
146 generated quantities {
147     matrix[N,N] K = kernel_f(X, X, 1., kernel_lengthscale,
148     alpha, kernel, diag_stds, jitter);
149 }
```

3.2.3 Conditioning

Stan´s algorithms approximate the density, or penalized maximum likelihood, step-, and gradient based. This leads to the statistical efficiency of the stan programs relying on posterior curvature, which is not captured by these algorithms. To solve this problem, one has first to look at the Hessian, which is defined in a statistical context by

$$H(\theta) = \nabla\nabla log(p(\theta|y)) \tag{3.17}$$

where again, θ is the parameter vector, and y is the data. Here, the Hessian is a second order approximation to curvature, which can be expressed in matrix form as

$$H_{i,j}(\theta) = \frac{\partial^2 log(p(\theta|y))}{\partial\theta_i \partial\theta_j}. \tag{3.18}$$

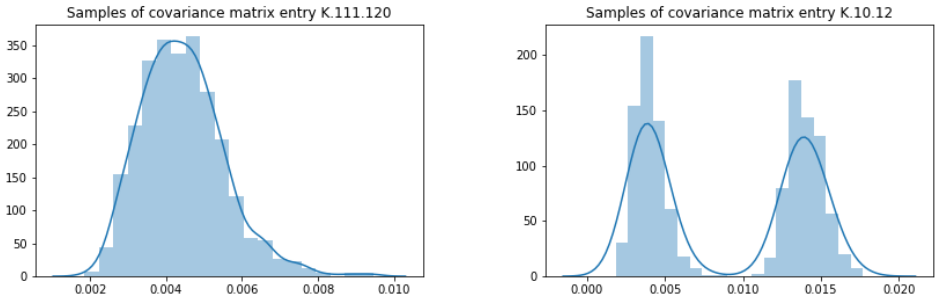
Using this, we find that comparing the largest and smallest eigenvalue of H as a ratio gives a good estimation of how much of a problem the aforementioned curvature presents. That ratio represents the largest and smallest curvature values, where typically the step size of a gradient based algorithm is bounded by the highest curvature value. Optimizing this highest curvature leads to better conditioning of the model, which in turn enhances computation cost and time. In all models, except the most simple ones (e.g. multivariate normal distributions), the Hessian will vary as the set of parameters θ varies. And for the assumption of fixed adaptation parameters, which is true in all algorithms except Riemannian Hamiltonian Monte Carlo, which in turn is not implemented in stan and computationally very expensive, a high variance of curvature will lead to worse adaptations covering the entire density sufficiently. Variable transformations are often proposed with the aim of improving

the conditioning of the Hessian. This usually entails, or is transferable with making the Hessian more consistent across relevant parts of the density. For variational inference the iterative algorithm then finds a single point as curvature of the path from the initial values of the parameters to the solution. Taking this into account, another way of improving statistical efficiency is reparameterising the model in such a fashion, that the same results may be calculated, using a better conditioned density or maximum likelihood. Diving into this would entail redefining bayesian posteriors as to what they technically are, probability measures, so this is left for further work.

4 Results

4.1 GPLVM

The Gaussian process latent variable model (GPLVM) is the basis of all following models. This stochastic process tries to learn a covariance structure from the data. All stocks are assigned a position in latent space, which influences the covariance structure the process can reproduce. This influences the covariance matrix, which is the key element of the learning equation of the GPLVM, representing the covariance structure learned. Data is fed into the model as a matrix of size $\mathbb{R}^{NxD}$, with N stocks' returns observed over a timespan of D days. If no data was available, the spot in the matrix was assumed to be 0, which is a reasonable assumption, since returns over small timespans are close to zero with only a small drift towards positive values. Then, the calculation specified by the model, as explained in more detail in the previous chapter 3.1.3, is carried out. Samples from the Variational Bayes algorithm are generated and used to calculate the mean prediction for a specific entry of a matrix, vector, or simply a scalar. With these, the interpretation is carried out. The



(a) The entry 111/120 of the covariance matrix of the GPLVM model trained on a dataset of $N = 120$, $D = 754$. For this covariance entry, the algorithm has converged sufficiently and sufficiently fast.

(b) The entry 10/12 of the covariance matrix of the GPLVM model trained on a dataset of $N = 120$, $D = 754$. For this covariance entry, the algorithm has not converged sufficiently or sufficiently fast.

Figure 4.1

plots in 4.1 show the distribution sampled by the variational inference algorithm as kde and histogram plot of two entries of the covariance matrix. The ratio of sampled values in the main peak compared to the other peaks, as well as the position of the global maximum of this distribution compared to the effectively used value, which is the overall mean and

further on used quantity, give a good estimate of the error that is effecting the further results for a calculation where the ELBO has converged. Entry K.10.12, figure 4.1b, shows critical behavior where the algorithm probably has not converged sufficiently, or where different values seem to provide stable solutions. This is especially interesting, since time-dependent solutions to the covariance between two stocks seem like a viable alternative. The entry K.111.120, figure 4.1a instead provides a clearer image of what was expected, where the mean is closer to the maximum of probability mass. The learning equation,

$$\hat{Y}_{GPLVM} = K[(K + \delta_{ij}\sigma_n)Y], \tag{4.1}$$

is used to calculate predictions from the model. Testing the prediction power of models is usually done in several steps, where the first step is to test it against predicting the initial data. With the predictions, we can calculate errors based on the model reproducing the dataset it was trained on, resulting in $\hat{Y}$, a matrix of equal size as Y. Since the model includes uncertainty and noise, we do not expect a perfect reconstruction of the data fed into the model. Comparing the data with the predictions yields interesting insight, since several possible errors can be directly visible from these results. But first, it is necessary to discuss various ways of quantifying the quality of a model. The Evidence Lower Bound, which was introduced in chapter 2.2.2, is a unit-less measure of the quality of a model. The higher the ELBO, the better the results. Comparing the ELBO values gives insight into how good a model was able to reproduce the true posterior with the variational inference distribution approximation of the true posterior, using KL-Divergence. The initial expectation is that models with more dimensions taken into account in the latent space (higher values of Q) perform better, and how more sophisticated kernels outperform less sophisticated kernels like the linear kernel [12]. As another measure of performance, the R^2 values of the models are calculated and plotted in the same fashion as the ELBO values. These are calculated of the data and predictions using the *LinearRegression* model from *arviz* [46], which uses the regular definition of the coefficient of determination (R^2).

$$R^2 = 1 - \frac{\sum(Y - \hat{Y})^2}{\sum(Y - \bar{Y})^2} \tag{4.2}$$

The coefficient of determination is a measure of how well the model has reconstructed the predictions as a function of the data, but it still has missing information. If we plot the data points against the corresponding predictions, we would expect a single straight line with slope 1 and intercept 0 for a perfect result. Since the model incorporates noise, we still would expect the model to have a slight deviation from this line, but it should be centered around the line. If the deviation is centered around the line, a Huber regression fit through the data-prediction pairs would also have a slope of 1 and an intercept of 0. Huber regression was chosen over linear regression, due to the outliers in predictions only accounting linearly instead of quadratic when doing linear least squares optimization during regression, and therefore provided much more stable and reliable information about the intercept and slope values. For more information, see Appendix 6.1. While reconstructing the model initially tested against these kind of data in [12], the plots showed an abnormality. The

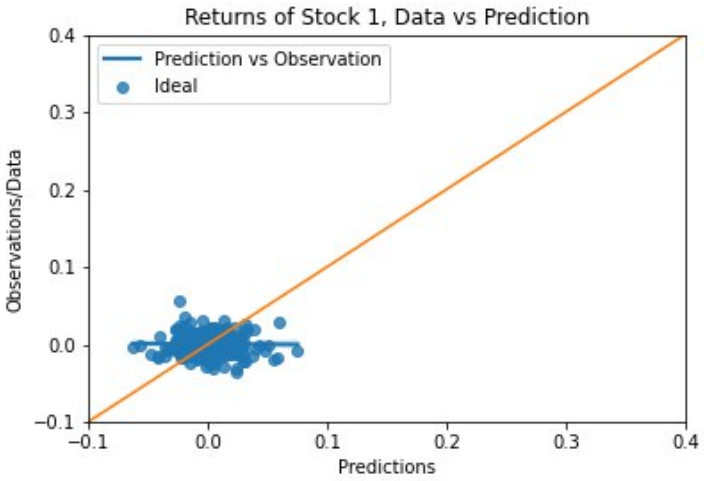


Figure 4.2: An example of a Y-$\hat{Y}$-pair plot of a stock fit with a Stochastic Process Latent Variable Model. A linear least squares fit is added, highlighting the observed effect of the predictions not accurately reflecting the data. Predictions on a regular basis seem to underestimate comparably high returns, and overestimate comparably low returns, resulting in a slope of the linear fit that is way off of the expectation.

model at hand overestimates lower returns, and underestimate higher returns, resulting in non-optimal ELBO and R^2 values. Higher ELBO values represent better estimation of the true posterior, but are not able to be interpreted with a higher bound due to the nature of the KL-divergence. R^2 values, the coefficients of determination, are a measure of the prediction quality, as a numerical value of how similar data and predictions are. A value relatively close to one is optimal, but due to the nature of how noisy stock market data is, a value of 1 would not be desirable. Because then the model would have learned a lot of noise as feature. Models that show considerably lower values in ELBO compared to the respective other models sharing characteristics probably have not converged towards the optimal point on the energy landscape, for example $Q = 3$ and $Q = 6$ with the exponential kernel in figure 4.3. We can find another metric of model quality in the distribution and mean values of intercepts, as well as slope values. These distributions show how the model has predicted the slopes, figure 4.5, and intercepts, figure 4.4, of all stocks entailed in the data matrix. Experiments show good agreement with expectations for the slopes, since they are close to the expected value of 1. Still they remain mostly to the left side of the bar at slope= 1, coinciding with the expectation that a model can never learn everything about a naturally noisy system. The distributions of the intercepts should be centered around the vertical line at intercept= 0, which would imply that almost the same number of stocks are over predicted as are under predicted.

Several sanity checks for the model are displayed, where the distributions of the samples, or just the distributions of the values of the entries of the vector-like or matrix-like objects

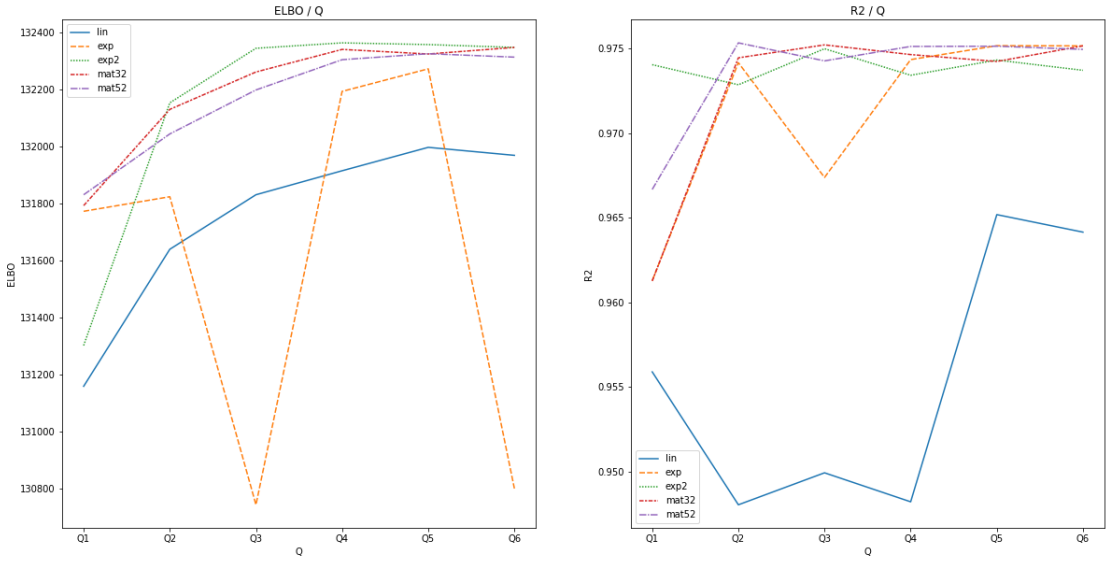


Figure 4.3: ELBO and R^2 values for the GPLVM model and $N = 60$, $D = 754$.

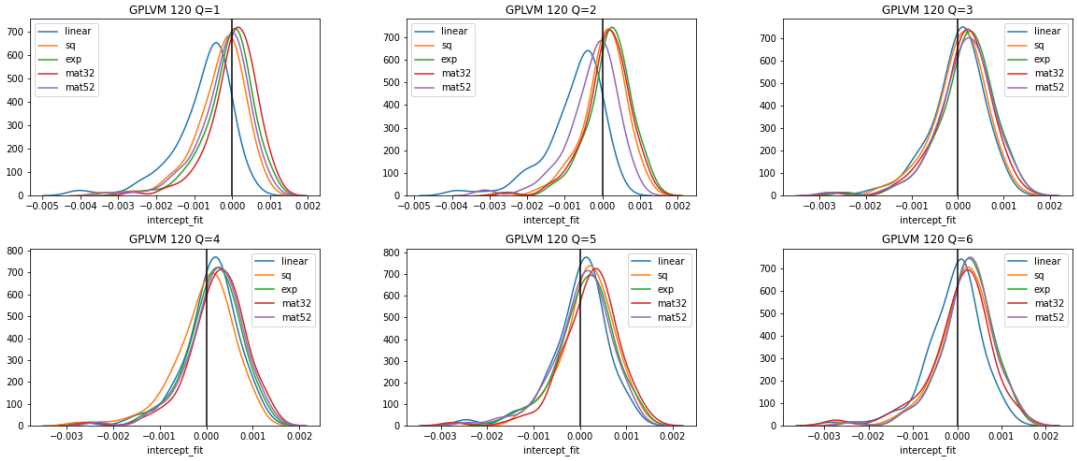


Figure 4.4: Comparing the distributions of stock slope values from the GPLVM model with different kernel functions over the scope of all latent dimensions.

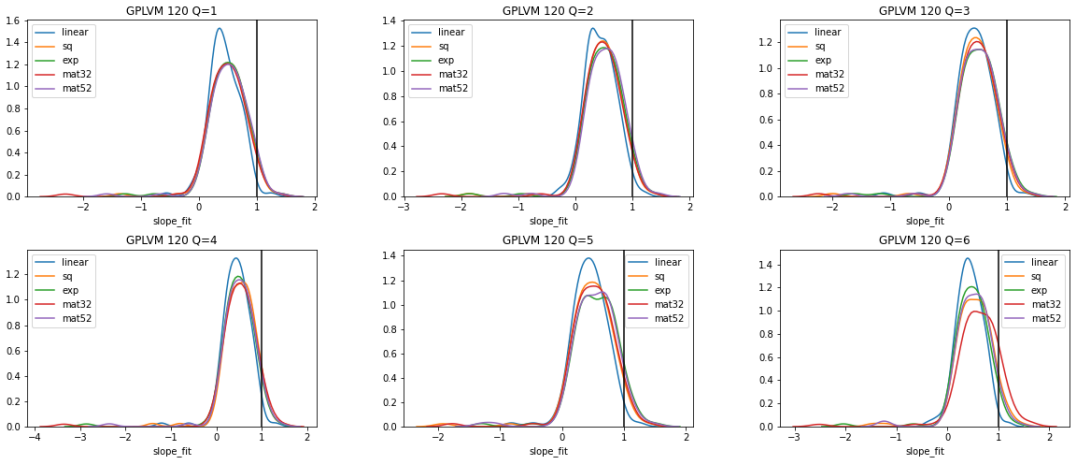


Figure 4.5: Comparing the intercept values of the GPLVM model with different kernel functions over the scope of all latent dimensions.

are plotted. Within these checks, larger samples of outliers can be detected, that may stem from local minima in the energy landscape in the variational inference solver. Figure 4.6, together with the learning equation, gives a direct interpretation of the kernel noise as the observation noise, since it is added to the diagonal of the covariance matrix. These entries resemble the self-correlation of the stocks. Also, it is necessary to point out, that these kernel density estimate plots of the variables can be misleading, since e.g. the noise is restricted to $\mathbb{R}_0^+$, since noise can not improve the measurement accuracy to be better than perfect. In figure 4.7 the variances are shown. These are estimated once for each stock, and then kept constant over time. The linear kernel is observed to more often need a higher volatility to function, even though it was outperformed by the other kernels.
Similarly, the entries of the inferred covariance matrix entries' distribution can be plotted, like in figure 4.8. They resemble the statistical connection between two stocks, and should also be $\in \mathbb{R}_0^+$. This is due to covariance matrices, by definition, being positive semi-definite. Also having many entries > 1 seems improbable, since that would imply a covariance of very great degree, in turn implying a correlation of an unusually high degree. A very large part of the values is close to 0, implying a sane model. After considering the sanity of the model compared to the expectations that can be deduced from observations of reality, different hypotheses can be formulated to explain the deviance from the expectation in the data-prediction-pair plot 4.2. At first, the effects of the size of the data matrices used to train the model were considered. For this, the size of the data matrix was scaled to include data matrices with different numbers of stocks, and different time periods over which the observations have taken place. We therefore look at different types of flaunty behaviour in the reconstruction, comparing figures 4.9, 4.10, 4.11 and 4.12, which show typical behaviour of data-prediction pair plots for $N = 20$ 4.9, $N = 60$ 4.10, $N = 100$ 4.11, and $N = 120$ 4.12. From this we can conclude, that the model has an intrinsic error reconstructing

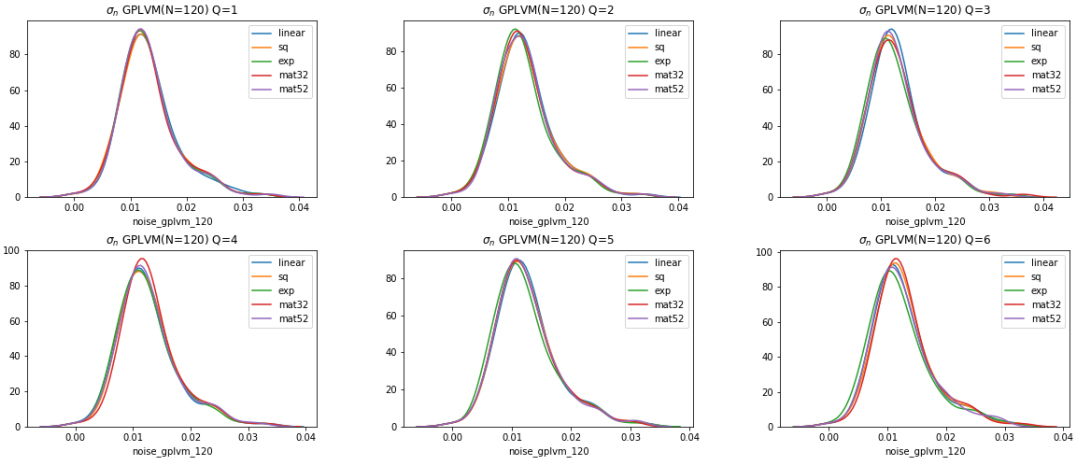


Figure 4.6: Distributions of vector elements of σ_n. The closer values are to 0, the more accurate the measurements are according to the model. The observation noise is non-isotropic.

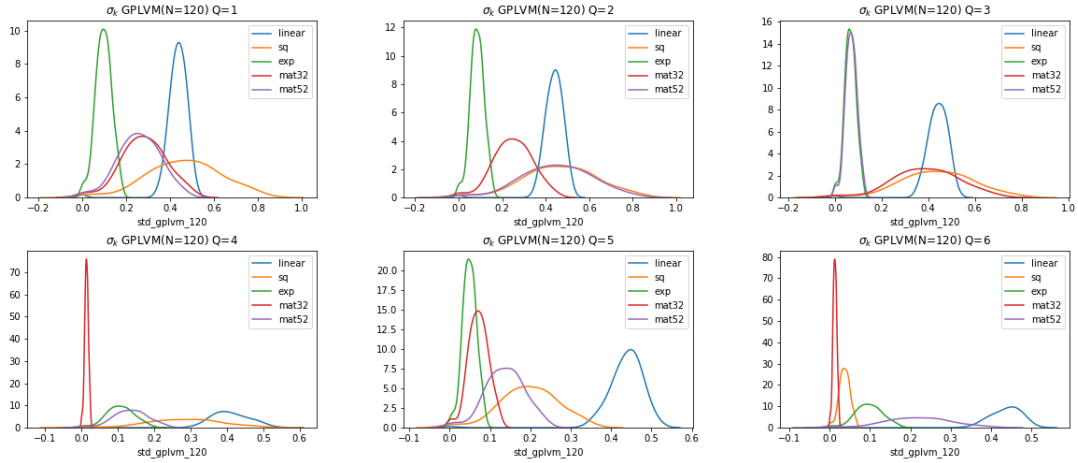


Figure 4.7: Distributions of vector elements of σ_k. These values represent volatility in a financial context.

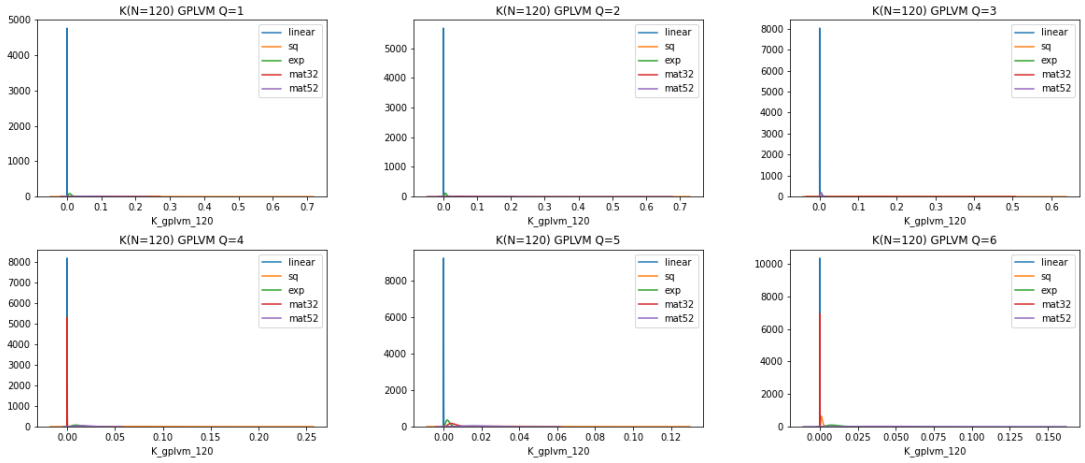


Figure 4.8: Distributions of the values of the entries of the covariance matrices.

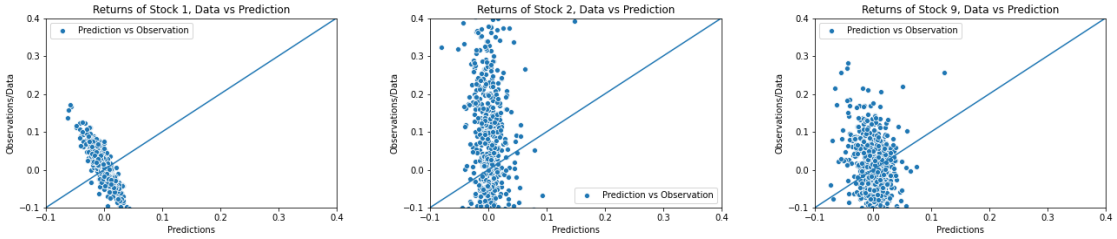


Figure 4.9: Plots from the GPLVM reconstruction with the $N = 20$, $D = 754$ dataset. The reconstruction failed, but the model had a fast convergence in distributions.

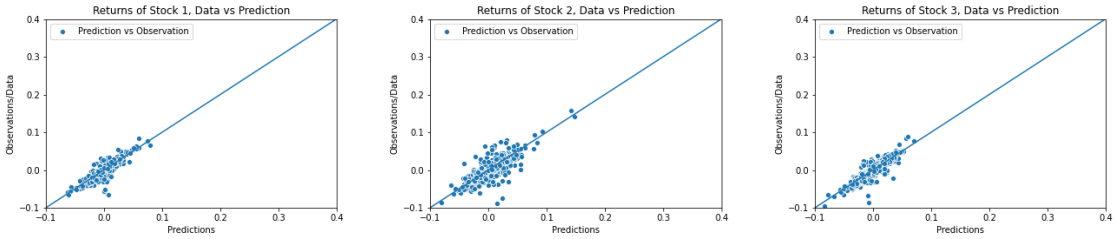


Figure 4.10: Plots from the GPLVM reconstruction with the $N = 60$, $D = 754$ dataset. The reconstruction worked properly.

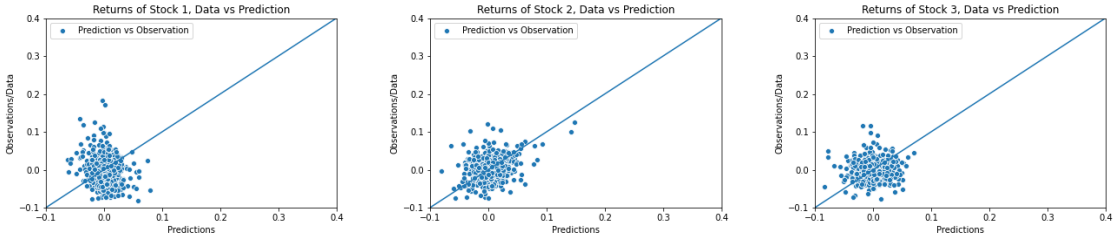


Figure 4.11: Plots from the GPLVM reconstruction with the $N = 60$, $D = 754$ dataset. The circular shape of the cloud indicates that the distributions of Y and $\hat{Y}$ are well fit, but subpar reconstruction.

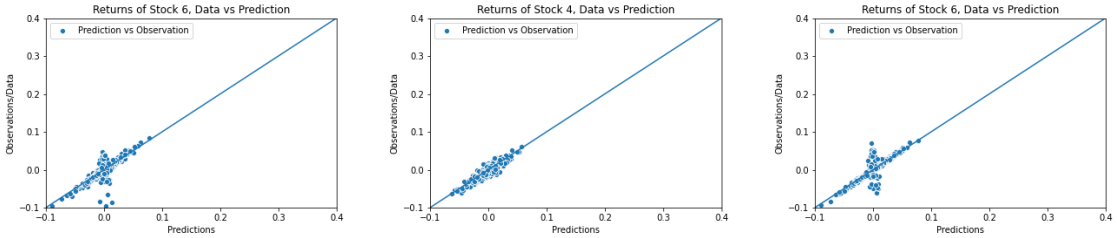


Figure 4.12: Plots from the GPLVM reconstruction with the $N = 60$, $D = 754$ dataset. We can find a good estimation of the true posterior, and a good reconstruction, but also find that in some cases, to match the data distribution some predictions are very close to 0, not representing the data at all.

the covariance structure while still matching the target 1D-distributions of the data. In the following part of this chapter, different possible solutions to this problem are discussed.

4.1.1 Volatility normalized datasets

Since the GPLVM model does not have time dependent volatility, different possible scenarios can not be incorporated into the model. For example, stocks with comparably high, but also highly varying volatility will not be represented very well, since the model will probably infer the mean volatility for this value. Therefore, the model was also tested against datasets where the stocks were normalized (VN - Volatility normalized) one by one, using the inferred volatility of a HCMC optimized GARCH process, compare sections 2.1.4 and 2.4, as well as [47]. The results are again depicted through example pictures of the systematic deviation observed in the $Y - \hat{Y}$-pair plots. Also, sanity checks were performed, e.g. plotting noises and variances. Here, since the expected volatility was used to normalize the returns on their respective predicted volatilities, the variance in figure 4.14 should be close to zero. This aspect was recognized correctly by the model. The inherent noise should be small, and is shown in 4.13. These figures also show, that the calculation does not resemble correctly our expectations of the model. Also, we acknowledge that another prior distribution would have been a better suited choice, since the kernel density estimate plots show values smaller than 0, which are forbidden through the generating process chosen. Since we found a dependency

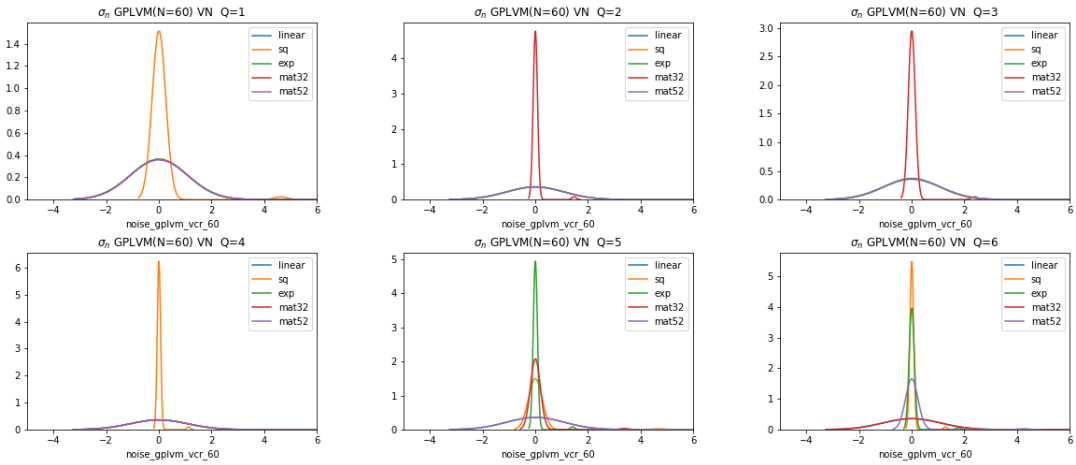


Figure 4.13: Distributions of σ_n, which were trained on a VN dataset with $N = 60$ and $D = 125$.

on the size of the data matrix Y, we test the model with different sizes of datasets, shown in figure 4.15. We recorded which sizes of data sets, within the confinement of the largest data set of $N = 120$ and $D = 754$, would successfully run without stan rejecting the initial values due to bad conditioning. Stan uses several hundreds of initial values for initialization. An orange dot is used if the model did not initialize. If the model did initialize, a blue dot is

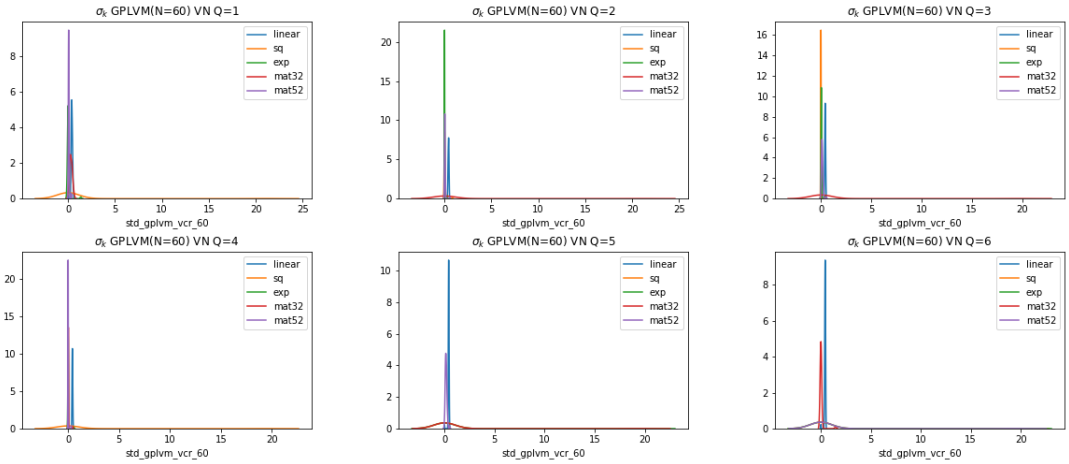


Figure 4.14: Distributions of σ_n, which were trained on a VN dataset with $N = 60$ and $D = 125$.

used. All experiments were carried out using a squared exponential kernel and using $Q = 3$ latent dimensions. We observe that the data sets that initialized successfully were either

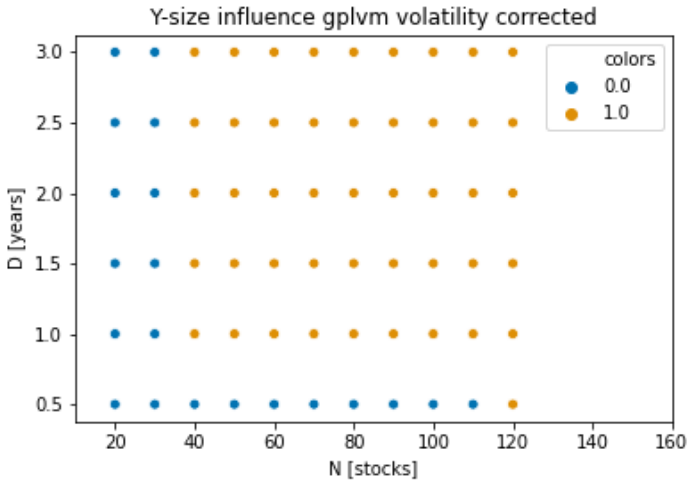


Figure 4.15: Shown here, in the same style as with the regular datasets and the GPLVM-model is the influence of dataset sizes on initialization. Again, blue points represent input matrix sizes, that were successfully initialized, while orange points represent matrix sizes, where initialization failed.

over very short time frames, or with very little stocks. We conclude therefore, that the used GARCH process breaks down for longer time frames due to errors propagating, as well as for more observed stocks, which reflect a more complicated part of the true covariance structure. In comparison, experiments showed that the regular, unnormalized, data set always

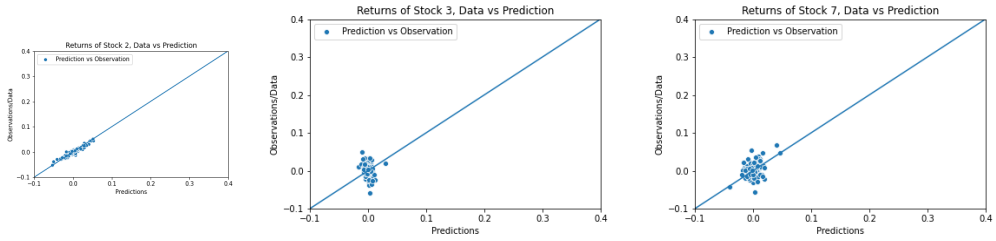


Figure 4.16: Plots from the GPLVM reconstruction with the $N = 60$, $D = 754$ dataset. We can find a good estimation of the true posterior, and a good reconstruction, but also find that in some cases, to match the data distribution some predictions are very close to 0, not representing the data at all.

initialized properly.

The example data set evaluated was $N = 60$, $D = 125$. It showed a range of behaviors when reproducing the data. Examples of these behaviors are shown in figures 4.16, where the model does a very good fit (*left*), reproduces a very bad fit (*center*), and shows the property to reproduce a correct target distribution but not with the right data points (*right*). These pitfalls have also been seen before, in the GPLVM model, especially figures 4.9, 4.11, and 4.12. The suspicion therefore arises, that normalizing the dataset either went wrong, or destroyed the overall covariance structure in such a way, that the model could not reconstruct it.

4.2 Student-t Process Latent Variable Model

The Student-t Latent Variable Model (SPLVM) is based upon the notion that a Student-t process can more accurately fit data with a higher percentage of outliers due to broader flanks. This, in theory resembles a higher volatility, but it is not directly implemented into the model. Overall, expectations are, that the higher robustness towards outliers will improve upon the results of the GPLVM.

First, we implement the model, similar to the GPLVM model, but using a function that solves the model using multivariate Student-t distributions. Compare the model code in Appendix **??**. Analyzing the closed form solutions described in 2.3.1, we find that the Student-t process will behave like the GPLVM, but with an additional multiplicative term giving more flexibility to the covariance matrix. Also, we observe that a Student-t process will become a Gaussian Process for the degree of freedom parameter $\nu \to \infty$, the evaluation of ν will be particularly interesting for interpretation. It can solve the question of whether a Gaussian process ($\nu \to \infty$), a Student-t Process ($\nu \in [2, \infty)$) solve the problem best, and even show tendencies since the interval is defined over all possible values for $\nu \in \mathbb{R}_0^+$. A rule of thumb in practice is, that from a value of $\nu \geq 30$ on, the Student-t distribution is sufficiently well approximated by a Gaussian distribution. The optimal inferred values of ν are shown in table **??**, for the calculations the same input sizes are used as in the main GPLVM run, $N = 120$, $D = 754$. Therefore, we can conclude temporarily that A more Student-t like stochastic process is probably better suited. We find the results of the Student-t model as figure 4.17, further exemplifying the notion that the Student-t process is better suited, if not by much. Again, the same holds as with the GPLVM, where the linear kernel underperforms, and higher numbers of latent dimensions perform better. The distributions of intercept values and slope values of the 120 stocks are shown in figures 4.18 and 4.19. Here, we see that the SPLVM model again performs slightly better than the GPLVM. We attribute this performance increase to the broader flanks of the Student-t distribution. Therefore, the outliers seem to be a problem. This does not explain which influences result in those outliers.
Just as the GPLVM model, the SPLVM model initializes with any regular data set size, the noticeable difference here, is that it performs so much worse with the VN data sets, that it did not initialize at all. This leads to the assumption, that the volatility does contribute into the broader flanks of the Student-t distribution. Also, we conclude the same sanity checks as before in section 4.1.1, looking at the noise and the variance entries. Figures 4.20 and 4.21 show that for all models, the noise and variance entries show no erroneous behavior.
The dependence of the SPLVM model on the size of data set was also determined by looking

ν	Q1	Q2	Q3	Q4	Q5	Q6
lin	8.847574	2.006339	2.001937	8.651835	2.007558	2.002983
exp	2.008647	2.003451	2.009492	2.003444	2.003168	2.001988
exp2	2.002464	2.007803	2.002753	8.199555	2.001955	2.009912
mat32	8.351861	2.002681	2.002718	2.003437	2.006565	2.009912
mat52	2.002612	2.011853	2.004670	2.002371	2.003071	2.001988

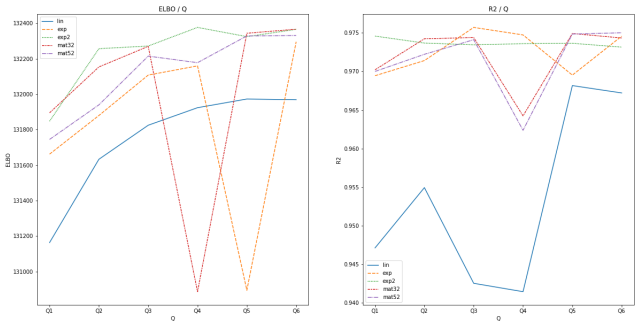


Figure 4.17: ELBO and R2 values of the Student-t Process Latent Variable Model (SPLVM).

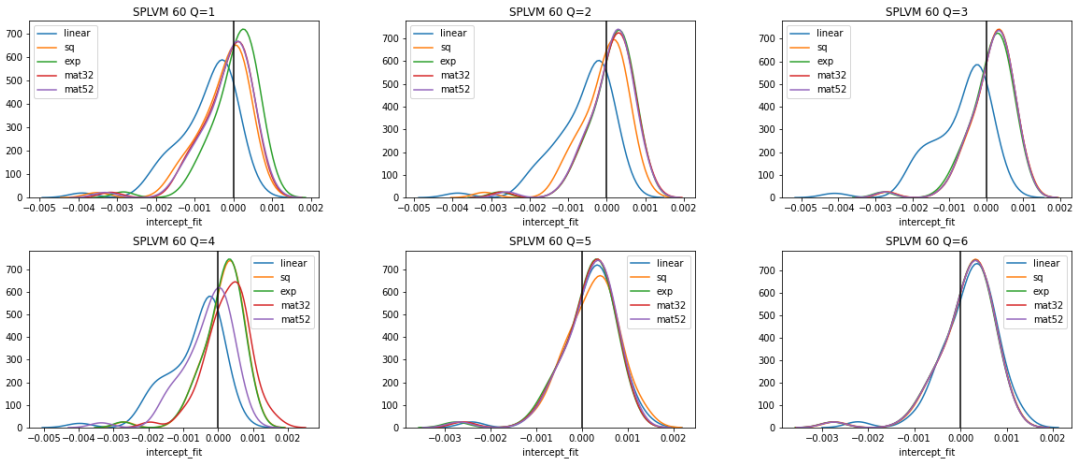


Figure 4.18: The distributions of the intercept values, which should be centered around the vertical line at $x = 0$.

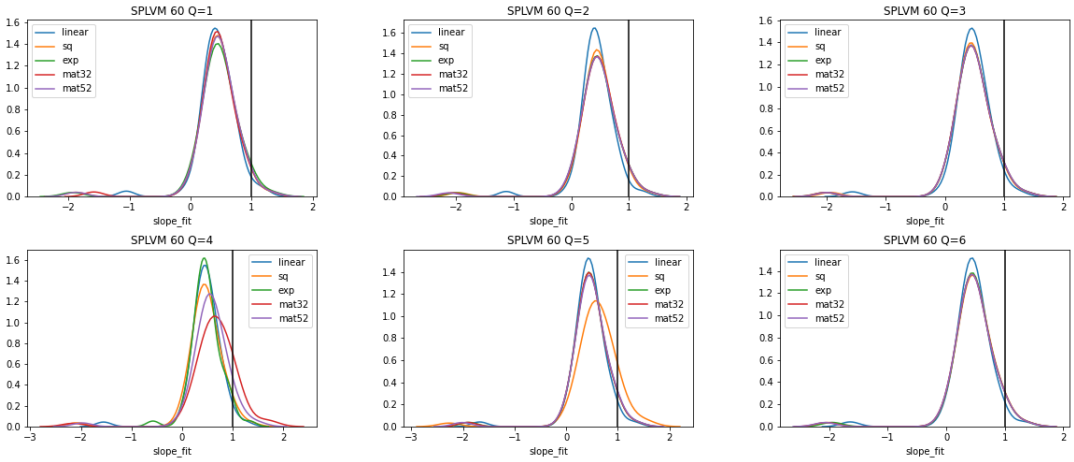


Figure 4.19: The distributions of the slope values, which should be centered around the left of the vertical line at $x = 1$.

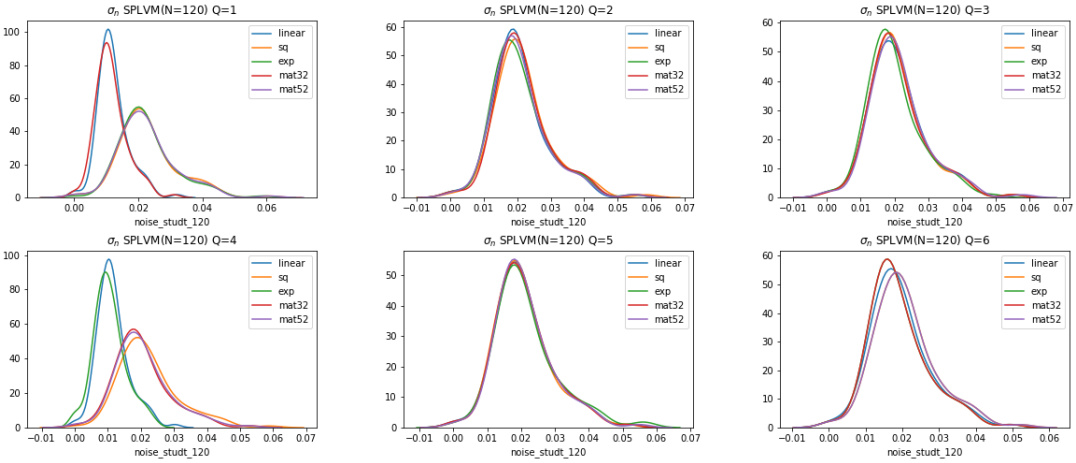


Figure 4.20: The distributions of values of the vector σ_n.

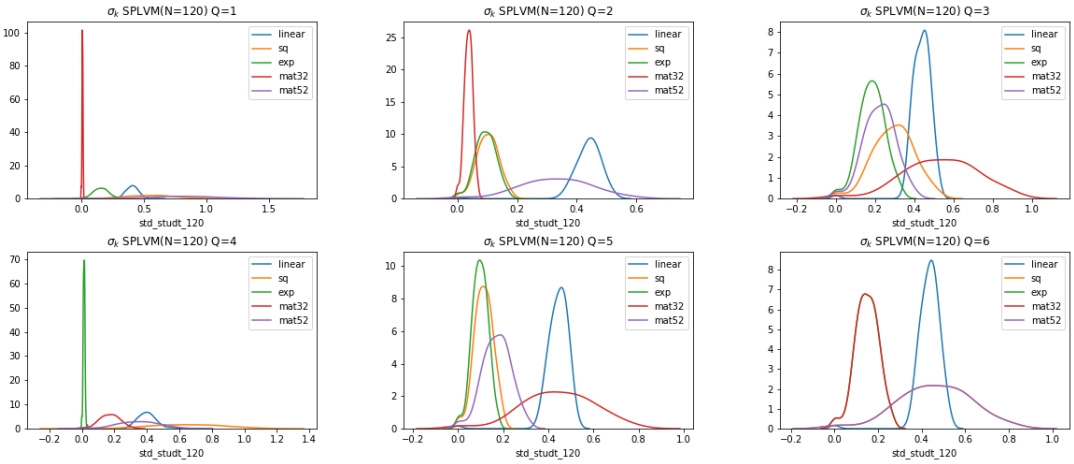


Figure 4.21: The distributions of values of the vector containing the variances.

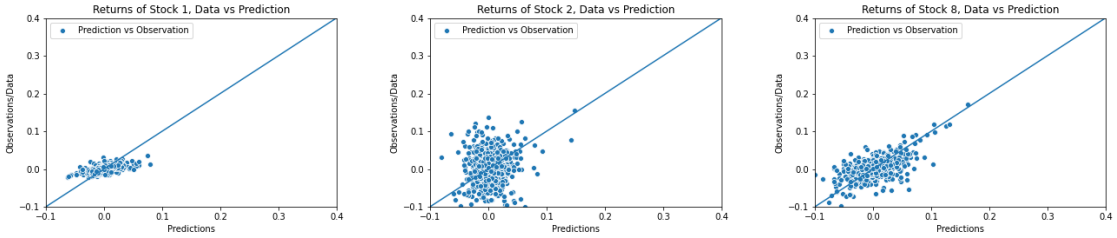


Figure 4.22: Plots from the SPLVM reconstruction with the $N = 20$, $D = 754$ dataset. The reconstruction failed showing the circular correct reconstruction of a 1D representation of the distribution of data points.

at different erroneous behaviors of reconstruction of the data, in figures 4.22, 4.23, 4.24 and 4.25. Overall we observe that the systematic error observed in the GPLVM model results is lessened by applying the Student-t model to the same problem.

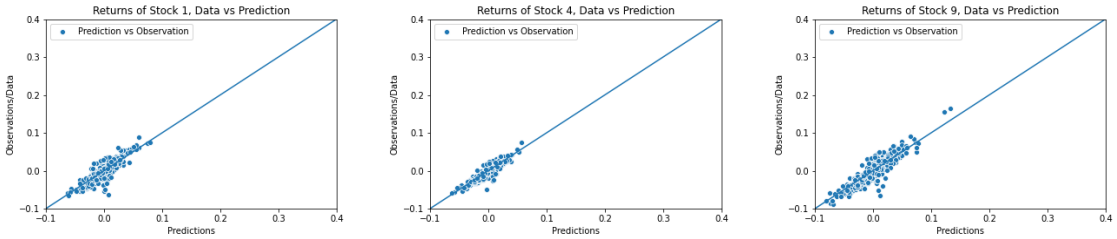


Figure 4.23: Plots from the SPLVM reconstruction with the $N = 60$, $D = 754$ dataset. The reconstruction worked properly.

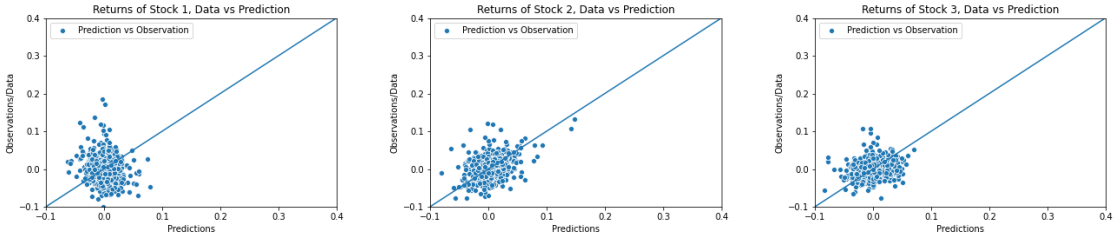


Figure 4.24: Plots from the SPLVM reconstruction with the $N = 60$, $D = 754$ dataset. The circular shape of the cloud indicates that the distributions of Y and $\hat{Y}$ are well fit, but subpar reconstruction of the data has taken place.

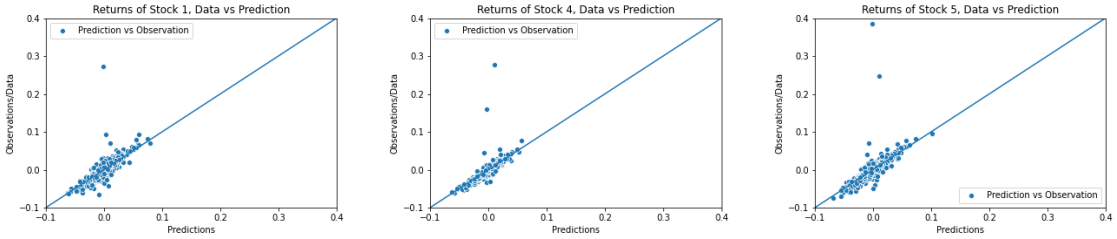


Figure 4.25: Plots from the SPLVM reconstruction with the $N = 60$, $D = 754$ dataset. We can find a good estimation of the true posterior, and a good reconstruction, while the off-diagonal that was observed with the GPLVM model is sufficiently smaller.

4.3 Volatility GPLVM

The Volatility GPLVM (V-GPLVM) model is a generalization of the GPLVM model, as discussed in section 3.1.3, where the volatility is reinterpreted as a time dependent quantity. Using the equations and generating process from section 3.1.3, we arrive at the model code found in **??**. Due to the large number of parameters that need to be stored in RAM at a time, the size of the data matrix Y had to be sufficiently smaller, $Y \in \mathbb{R}^{N,D} = \mathbb{R}^{10,250}$. We find that the model does not apply well to the data, which can be seen from the kernel density estimates of the distributions of noise (σ_n) and variance (Σ) values in figures 4.26 and 4.27. These estimates should not be having large parts of the estimation reach be outside $\mathbb{R}_0^+$, as this is forbidden by design. In addition, the values should not center around 1, again solidifying the evidence of a corrupted model. Also note, that

$$\Sigma \propto \log(\Sigma_V), \tag{4.3}$$

the variance Σ is proportional to the logarithm of the volatility $\log(\Sigma_V)$. The values of the covariance entries, which mainly lie at value 1 suggest, that the model finds and stays in a steep local minima on the energy surface, which seems unshirkable. It does not seem to be the global minimum though, since covariance entry values, ELBO values and R^2 values are far off from where we would expect them to be 4.28. We again see that the linear kernel under performs significantly, especially when looking at the volatility values, which need to be way higher, or even unreasonably high for the under performing model. More sophisticated kernels perform better in all tests, and again higher numbers of latent dimensions perform better than lower numbers. If we look at the results, we again can conclude, that not even all model specifications have landed securely in the suspected local minimum due to major differences in ELBO values and R^2 values. Also, the R^2 values shown in 4.28 lead to the wrong conclusion, since they only can compare lower dimensional representations of the distributions. Due to the differences in size in Y, these results can not be compared to the GPLVM results directly, but indirectly through looking at the behavior of the model reconstructing the input data in figures 4.29. These plots again show the same erroneous behavior as the GPLVM model, with a lot of stocks being represented with circle-like shape (*left*) that reproduces the distribution well, but does not reconstruct the higher dimensional representation of the pair plot correctly. This behavior can e.g. inflate R^2 values, misleading the reader. The *center* frame shows the erroneous behavior found in the GPLVM model, where to match the distributions, some predictions are falsely set to 0. Also, we can observe the behavior of the model simply underestimating high returns and overestimating low returns, so that the intercept is far off the desired value of 1. This leads directly to the slope, figure 4.30, and intercept, figure 4.31, distributions of the V-GPLVM model. Again, the model did not initialize sufficiently well with the volatility normalized data sets, as to provide an interpretation. Compared to the other models previously discussed, the slope and intercept distributions giving a quick overview of all the different possible Y-$\hat{Y}$-pair plots, are far off from the optimum. Therefore, we conclude that the model, even though not explicitly comparable, does not measure up with the previous models.

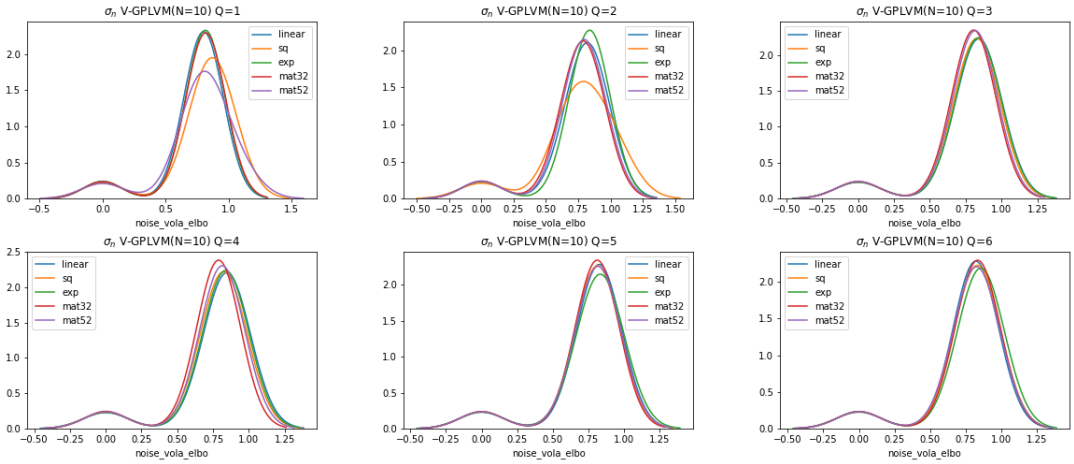


Figure 4.26: V-GPLVM noise distributions for the $N = 10$, $D = 250$ data set.

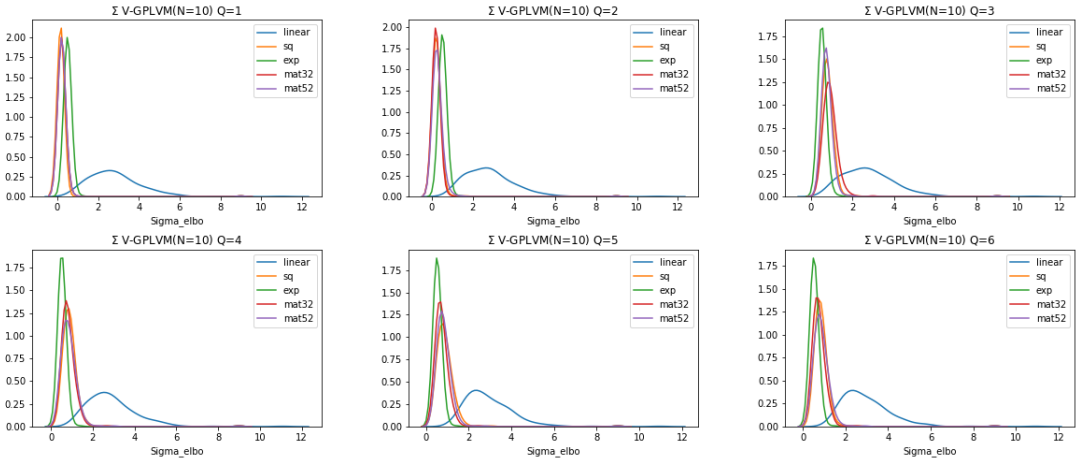


Figure 4.27: V-GPLVM Variance distributions for the $N = 10$, $D = 250$ data set.

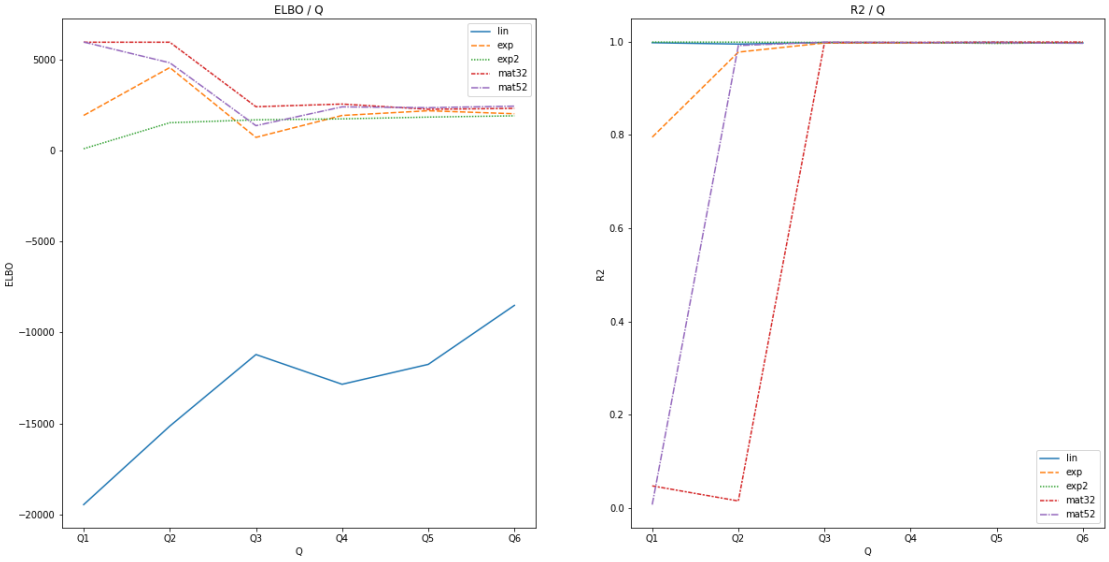


Figure 4.28: ELBO and R2 values grouped by number of latent dimensions (Q) for the V-GPLVM model.

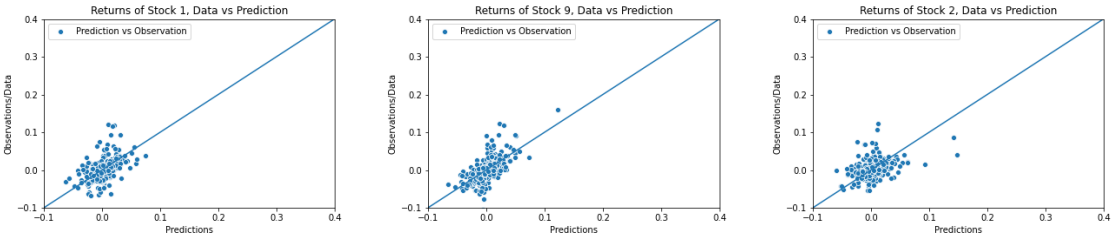


Figure 4.29: Y-$\hat{Y}$ pair plots with the V-GPLVM model $N = 10$, $D = 250$.

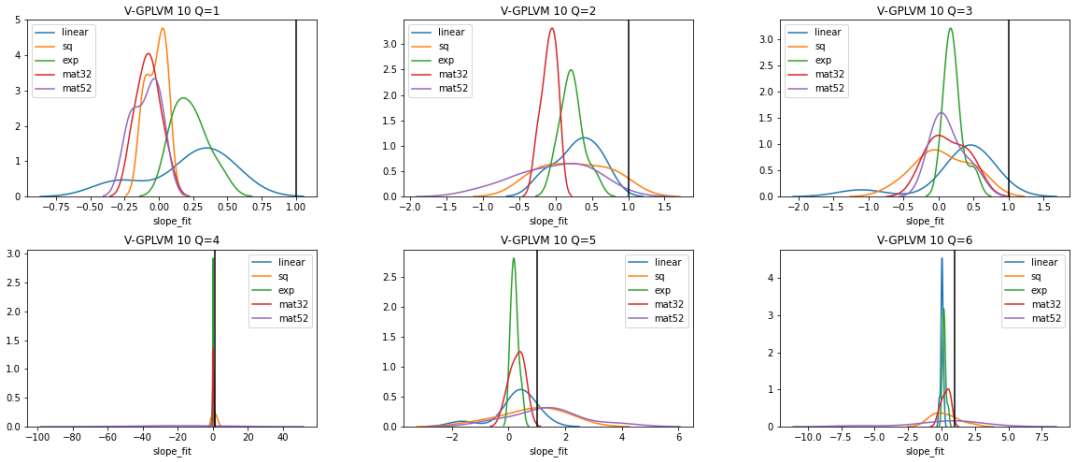


Figure 4.30: V-GPLVM slopes distributions for the $N = 10$, $D = 250$ data set.

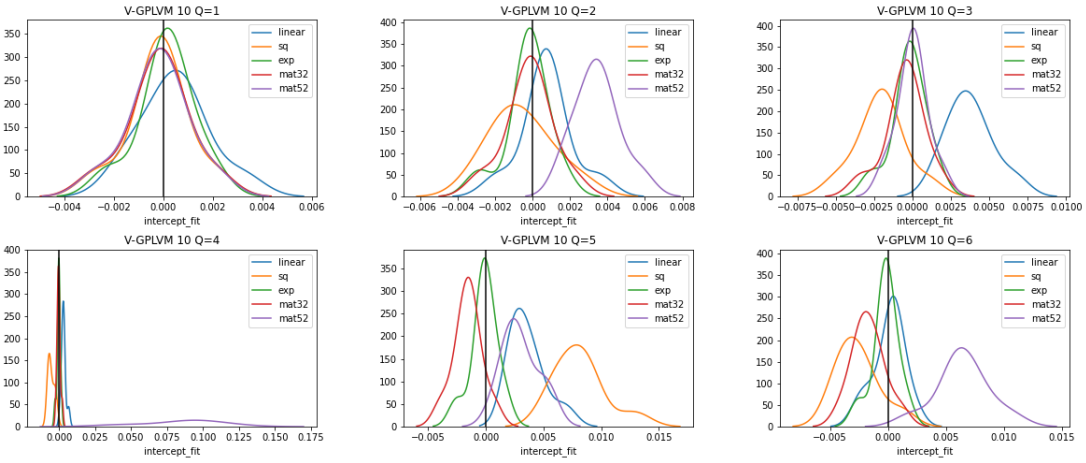


Figure 4.31: V-GPLVM intercept distributions for the $N = 10$, $D = 250$ data set.

4.4 Time-dependent Latent Space GPLVM

The Time-dependent Latent Space GPLVM model allows the GPLVM extra flexibility through the stocks representations in latent space having mobility. The distance between the instances in latent space implies higher correlation if it is smaller, and lower correlation if it is larger. We proceed with the same sanity checks and model evaluations as before. It is necessary to note, that the model did not produce symmetric covariance matrices for kernels of the Matern-class. Therefore, these are not part of the evaluation. Also, due to the size of the matrices needed to express this model, only the small data matrix that was used in the V-GPLVM ($N = 10$, $D = 250$) could be used. Figure 4.32 shows good agreement with expectations, observation noise is comparably low for all different versions of the model. On the other hand figure 4.33, displaying the variance, shows erroneous behavior, and probably would have been better approximated with some other, most likely asymmetric distribution. At last figure 4.34 shows values close to 0 dominating, implying comparably small covariance between stocks. Figure 4.35 then provides a problem of interpretation. While the ELBO values of the linear kernel are lower, the lower dimensional representation of the reconstruction in form of the R^2 values is considerably higher for the linear kernel. Overarching still is the fact, that the ELBO values and R^2 values still can not compete with the GPLVM model, just like the V-GPLVM model. This conclusion is further exemplified in figures 4.36 and 4.36, where intercepts can compete with the other models, but slopes of the Huber regression still produce distributions with values so far off from the desired value. Last, but not least, evaluating some examples of the Y-$\hat{Y}$-pair plots makes abundantly clear, that the model still has issues in need to be fixed. All examples show, that distributions match only in lower dimensional representations. The covariance structure of 10 stocks was not enough to be representative of the underlying structure.

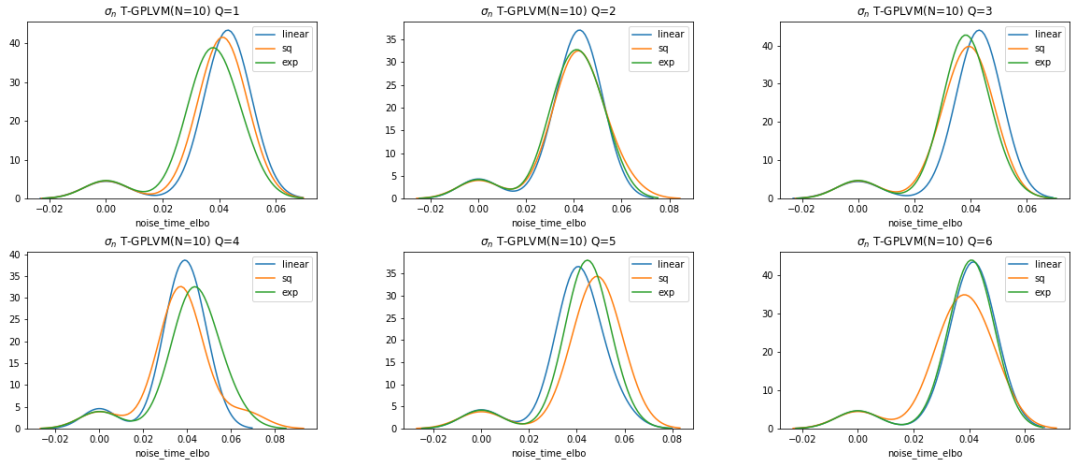


Figure 4.32: T-GPLVM data space (Y) observation noise distributions.

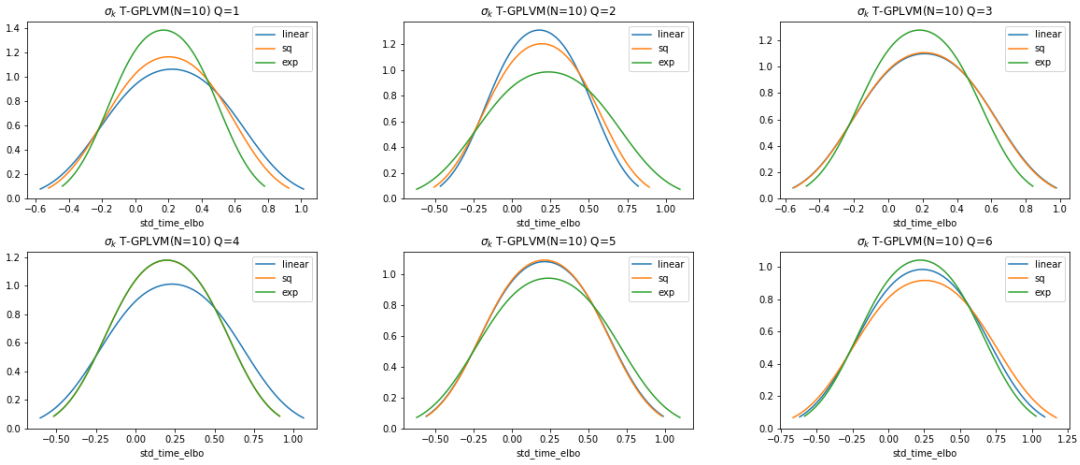


Figure 4.33: T-GPLVM data space (Y) variance distributions.

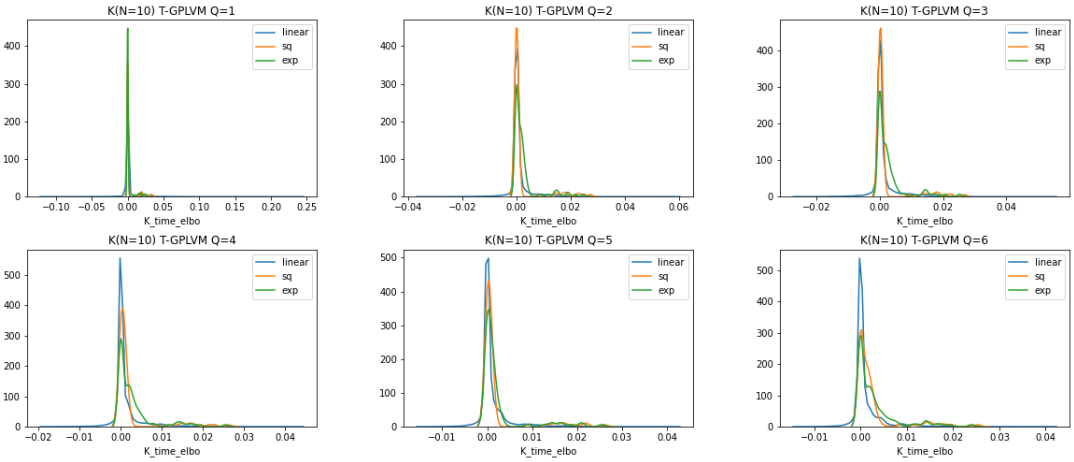


Figure 4.34: Covariance matrix entries distributions from the T-GPLVM model run.

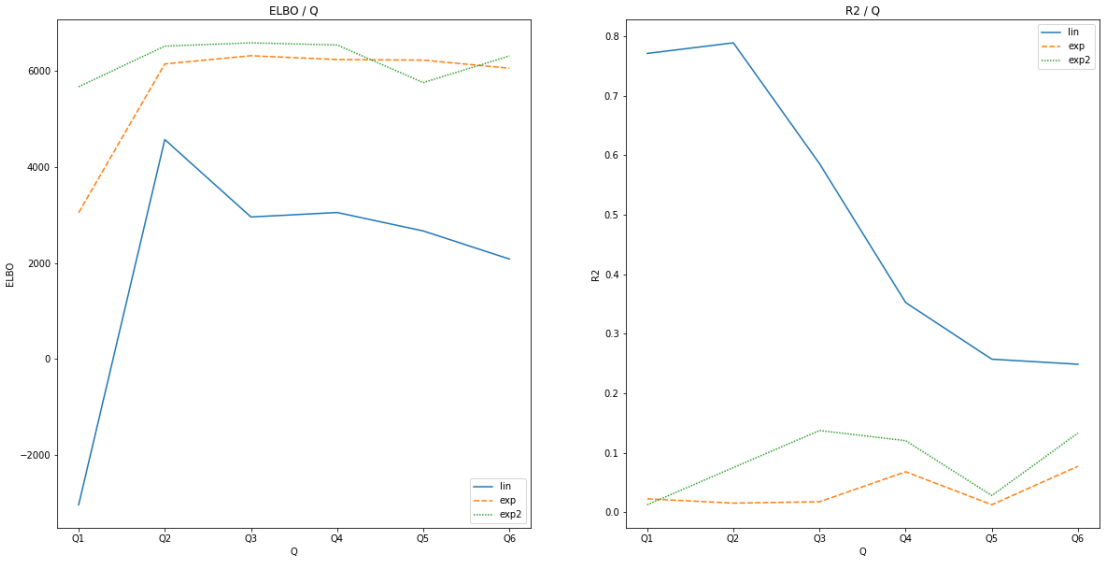


Figure 4.35: ELBO and R^2 values from the T-GPLVM model run, without the Matern class kernels.

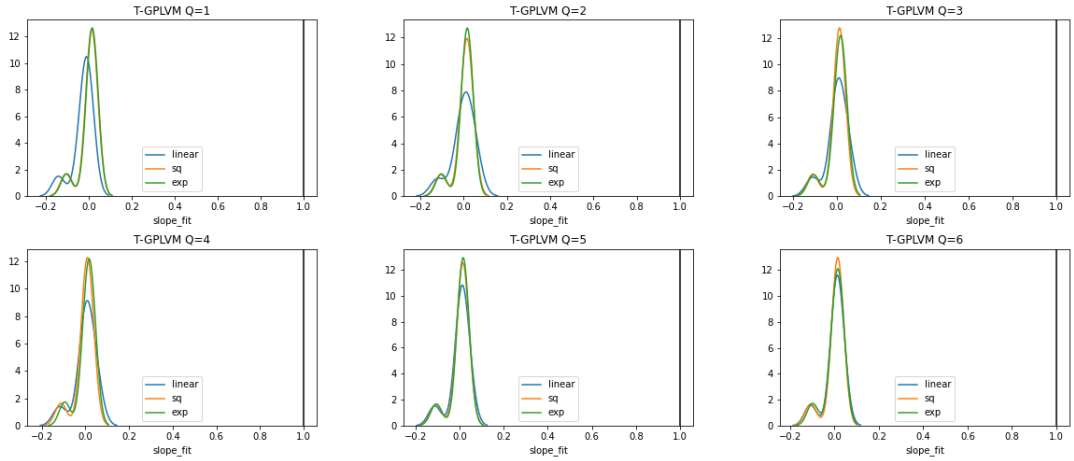


Figure 4.36: Distributions of slopes from the Y-$\hat{Y}$ pairs for the T-GPLVM model.

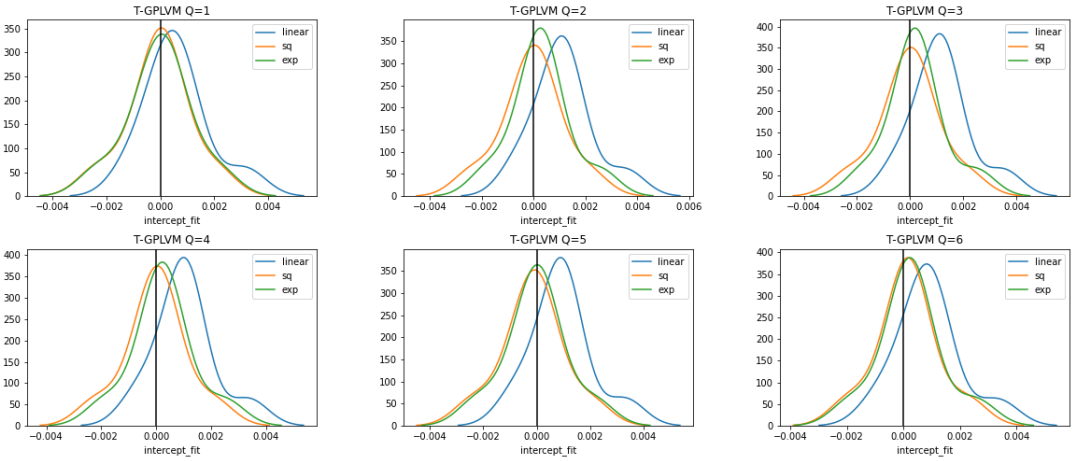


Figure 4.37: Distributions of intercepts from the Y-$\hat{Y}$ pairs for the T-GPLVM model.

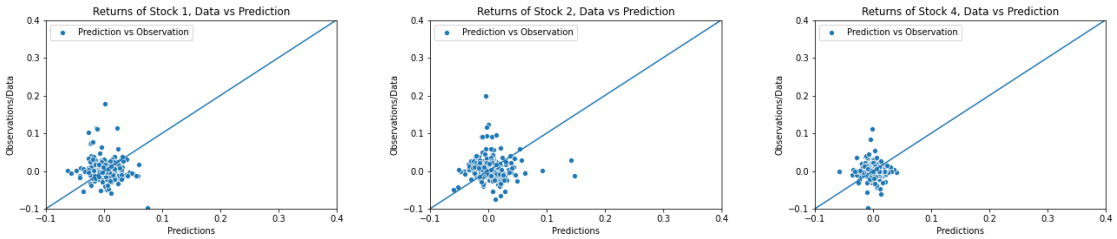


Figure 4.38: Plots from the T-GPLVM reconstruction with the $N = 10$, $D = 250$ dataset.

4.5 Time-dependent Latent Space SPLVM

The Time-dependent Latent Space SPLVM model (T-SPLVM) is in comparison to the T-GPLVM what the SPLVM is to the GPLVM. Figure 4.39 is showing distributions of the data space observation noise. These plots show healthy behavior, even though some stocks are inferred by the model to have 0 observation noise. Figure 4.40 follows, displaying the same unhealthy behavior as the data space process variance did in the T-GPLVM chapter 4.4. The entries of the covariance matrix then also show unexpected kernel density estimates, ranging to extremely high values. This leads to the conclusion, that the T-SPLVM model still lacks in quality. Checking the ELBO and R^2 values, after identifying erroneous behavior is but a formality, still it is easy to see that the model, scoring comparably to the T-GPLVM model, is not able to compete with the GPLVM or SPLVM model. Here, as the initial expectation suggests, the linear kernel performs considerably worse than the more sophisticated kernels. The second to last test of the model, looking at slope and intercept distributions, again proves this model as not competitive. While intercepts, as with all models, are close enough to the desired value of 0 to meet expectations. The last test of the model, comparing Y-$\hat{Y}$-pair plots 4.45 provides us with the evidence to also not consider this model for further evaluation, showing no behavior close to the expectation. We find defective behavior to hold for all examples of these plots, in all variants of the model.

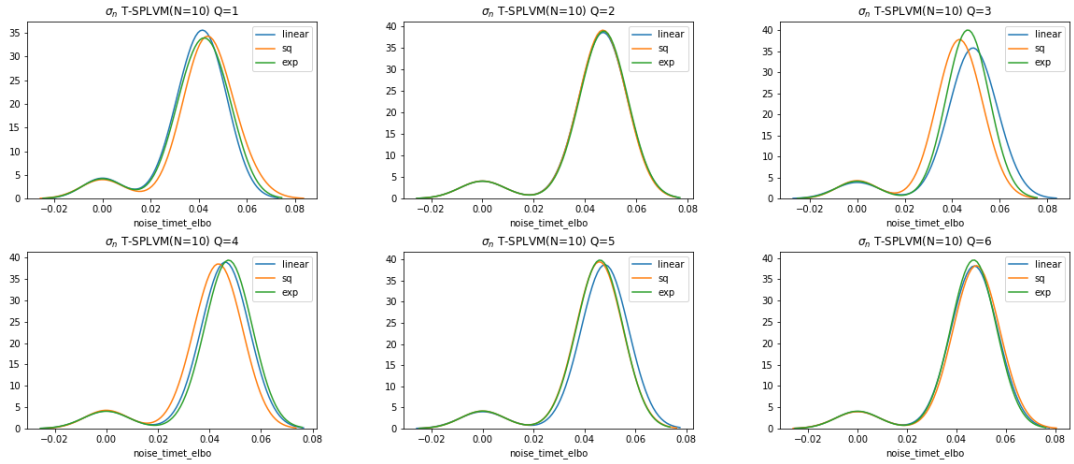


Figure 4.39

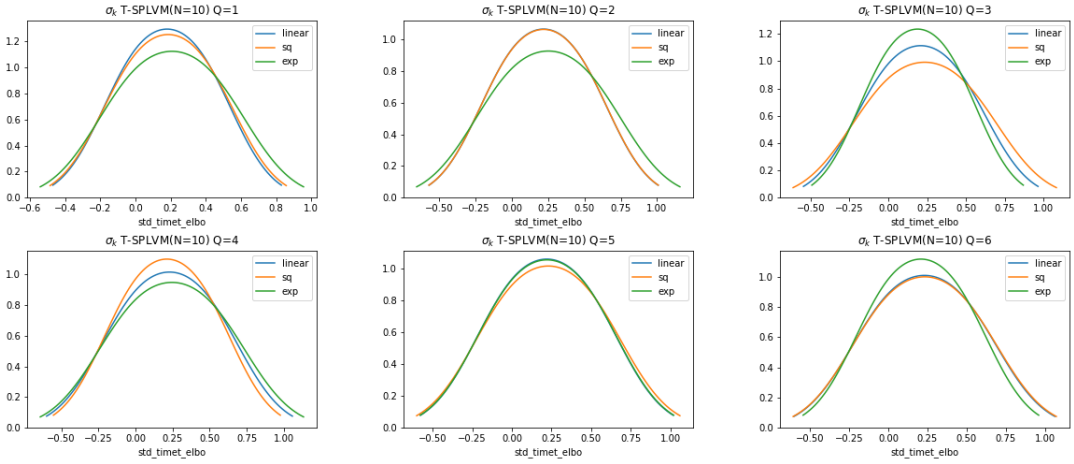


Figure 4.40

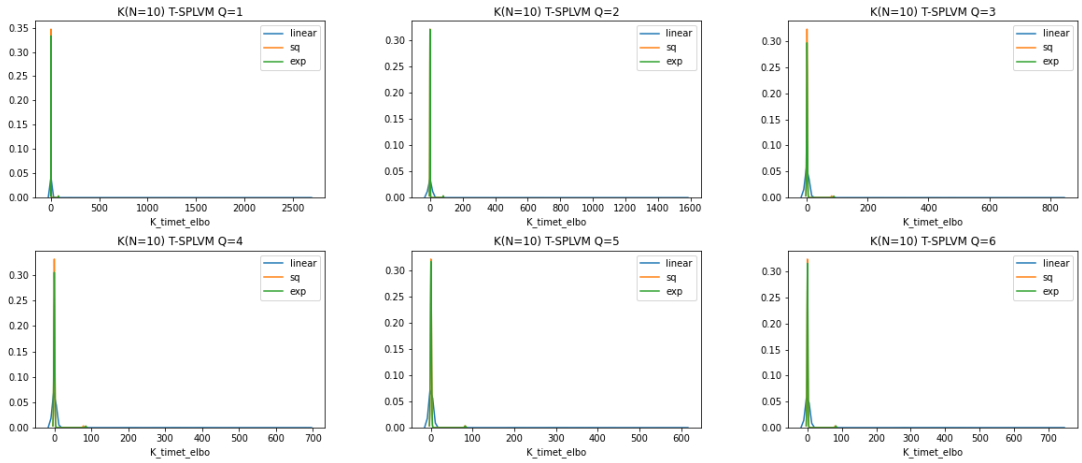


Figure 4.41

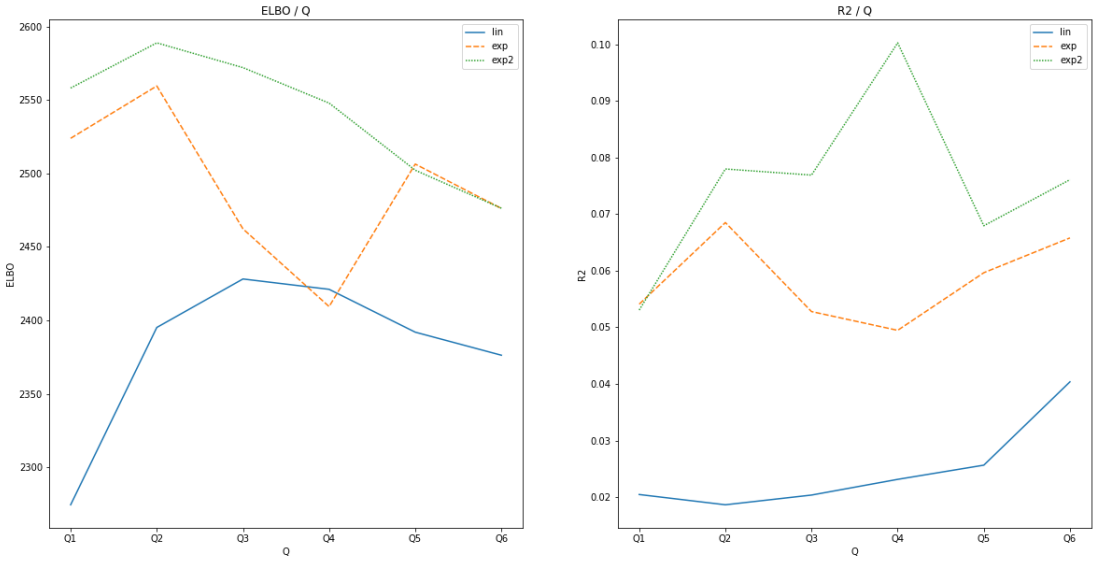


Figure 4.42

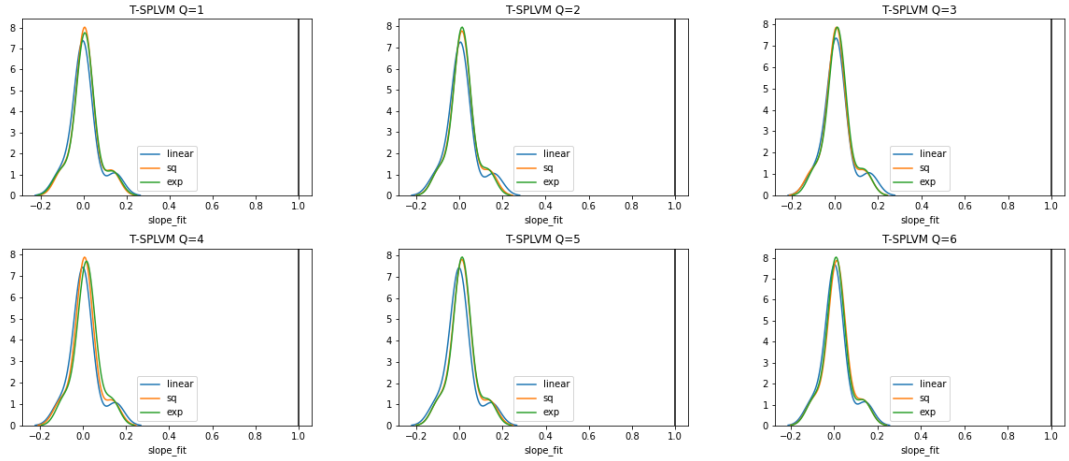


Figure 4.43

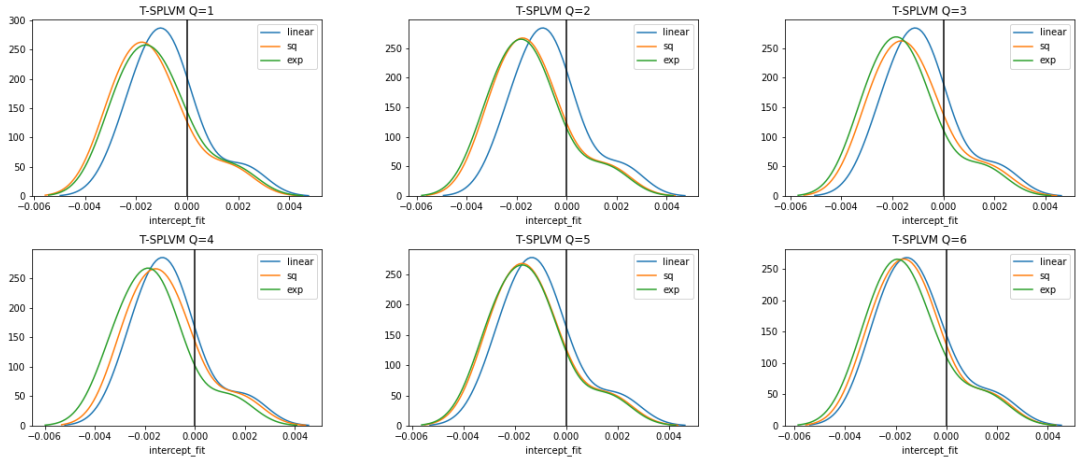


Figure 4.44

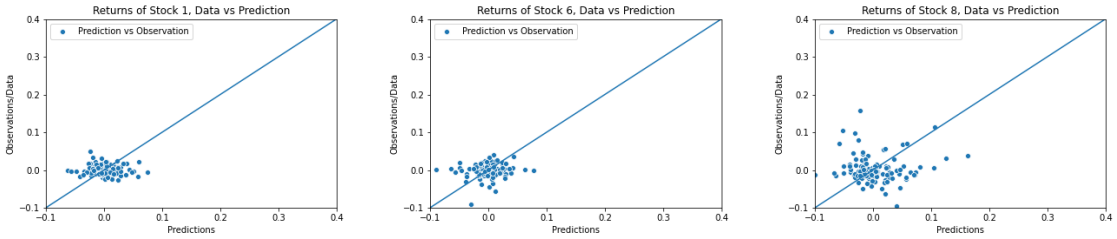


Figure 4.45: Plots from the SPLVM reconstruction with the $N = 10$, $D = 250$ dataset.

5 Conclusion and Outlook

In this thesis, we provide alternative models for the previously used GPLVM for financial applications [12]. The added models include the Student-t version of the GPLVM, the SPLVM 2.3.1, as well as the generalization of the GPLVM with both time dependent volatility (V-GPLVM) in section 3.1.3, as well as a time dynamic on the latent space of the GPLVM in section 3.1.2. The latter is then generalized to the T-SPLVM, the Student-t Process Latent Variable Model with a time dynamic on the latent space. All models are written down in terms of their mathematical formulation, and the generating process is laid out. After that, the application of these models using the stan [45] language is described, where the variational bayes algorithm is used to evaluate the models. The main focus of this work is laid on the development of the models, which shows in the results chapter 4.5. Here, also due to restraints of the size of calculations on the used computer cluster, the models including a time dynamic could only be solved with insufficiently high numbers of stock observations. The GPLVM and SPLVM model, in sections 4.1.1 and 4.2 on the other hand were carried out with much higher numbers of parallel observations.

In the results section 4.5 the models are again introduced and their pitfalls are explained. At first, the GPLVM is shown, and the central systematic error in the reconstruction of the covariance structure is shown. Several sanity tests for the models are introduced and explained, which mainly consist of checking distributions of the inferred parameter values, comparing them to the expected values. Then, several other factors are taken into account, like the distributions of slope and intercept values of the stocks fit with Huber Regression, which directly relate to the observed systematical error. This systematical error is analyzed thoroughly, with the main factor being the number of stocks in the data matrix Y. A data set of returns normalized on volatility, using a HCMC optimized GARCH process, is also applied to the GPLVM, but fails due to problems that seem related to the GARCH process quality breaking down after half a year of trading days with observations. Afterwards the Student-t Process Latent Variable Model is applied to the same problem. The broader flanks of the Student-t distribution reduces the extent of the systematic error, but does not completely rid the model of it. The three models with a time dynamic are introduced afterwards, but they already fail during the sanity checks of the models, all while not being competitive with ELBO and coefficient of determination values. To make better assumptions about the models though, they would need to be compared not only to each other, but the GPLVM and SPLVM with entirely equal problem settings. Putting the models into context, especially GPLVM and the SPLVM provide a high quality way of obtaining good covariance matrix estimates, that are necessary for e.g. market analysis, and even market predictions.

Further research and development into these models is needed, especially considering they can be used for all kings of regression and even classification tasks. The potential of the V-GPLVM, T-GPLVM and T-SPLVM, when applied with larger sizes of data matrices is also an untapped resource. As a statistical machine learning model these models have the potential to be applied usefully in different areas of research, since they have the potential to find correlation and infer the degree of causality. Also, when applied to financial markets, the sensible number of latent market variables, as they correspond to the number of latent dimensions of these models, can be approximated.

6 Appendix

6.1 Appendix A - Huber Regression

When analyzing Y-$\hat{Y}$-plots, a single higher deviation from the optimal line is possibly attributing for a high percentage of how the slope and intercept value turns out. But since way more points are located around the suspected optimal line, this value should reflect the accuracy of reconstruction better. Therefore Huber regression was used to fit the values of slope and intercept onto the data-prediction pairs. Huber regression is also a linear regression, but outside of an interval of *beta* times the standard deviation of data, the data points contribute less to the least squares optimization of the linear fit, by contributing linearly and not quadratically. To see that this enhances the predictions, we show an example of some stocks data-prediction plots, Figure **??**, with the optimized fits of Huber regression (hr, green lines), linear regression (lr, red lines), and the optimal line (y=x, blue lines). Huber regression shows more robustness towards outliers. This is also reflected when looking at the population of all days of a single stock, fitted once with linear regression (left, Figure 6.2a) and once with Huber regression (right, Figure 6.2b).

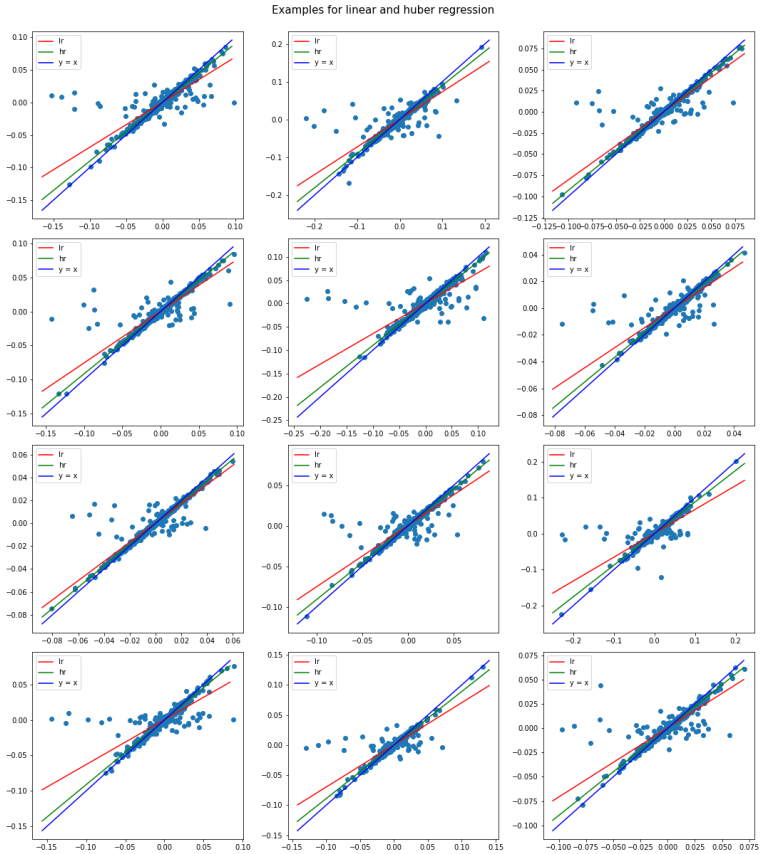
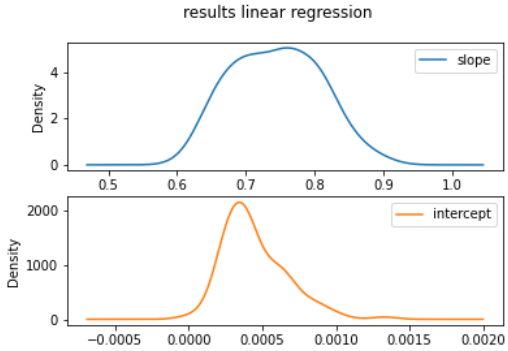
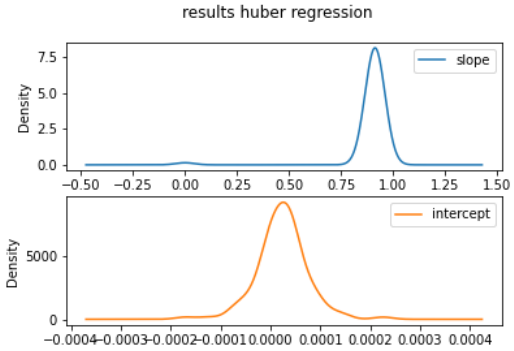


Figure 6.1: Several stocks, that exhibit strong cases of outliers in data-prediction pairs are shown, all with linear regression fits using least squares optimization (red), huber regression using least squares inside the interval of $alpha = 1.35$ standard deviations around the fit and linear contribution of outliers (green) and the optimal line with slope 1 and intercept 0 (blue). Using the convention of this thesis, the data are shown on the x-axis, and predictions on the y-axis.



(a) The populations of slope (top panel) and intercept (bottom panel) values as densities of a dataset of a single stock using linear least squares regression.



(b) The populations of slope (top panel) and intercept (bottom panel) values as densities of a dataset of a single stock using Huber regression.

6.2 Appendix B - Stan model code

6.2.1 gplvm_finance.stan

```
functions {
    matrix cov_linear(vector[] X1, vector[] X2, real sigma){
        int N = size(X1);
        int M = size(X2);
        int Q = num_elements(X1[1]);
        matrix[N,M] K;
        {
            matrix[N,Q] x1;
            matrix[M,Q] x2;
            for (n in 1:N)
                x1[n,] = X1[n]';
            for (m in 1:M)
                x2[m,] = X2[m]';
            K = x1*x2';
        }
        return square(sigma)*K;
    }

    matrix cov_matern32(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        for (n in 1:N)
            for (m in 1:M){
                dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
                K[n,m] = square(sigma)*(1+sqrt(3)*dist/l)*exp(-sqrt(3)*dist/l);}
        return K;
    }

    matrix cov_matern52(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        for (n in 1:N)
            for (m in 1:N){
                dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
                K[n,m] = square(sigma)*(1+sqrt(5)*dist/l+5*square(dist)/(3*square(l)))*exp(-sqrt(5)*dist/l);}
        return K;
    }

    matrix cov_exp_l2(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        for (n in 1:N)
            for (m in 1:M){
                dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
                K[n,m] = square(sigma) * exp(-0.5/l * dist);}
        return K;
    }

    matrix cov_exp(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
        for (m in 1:M){
            dist = 0;  //sqrt(squared_distance(X1[n], X2[m]) + jitter);
            for (i in 1:Q)
                dist = dist + fabs(X1[n,i] - X2[m,i]);
            K[n,m] = square(sigma) * exp(-0.5/l * dist);}
        return K;
    }

    matrix kernel_f(vector[] X1, vector[] X2, real sigma, real l,
    real a, int kernel, vector diag_stds, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        if (kernel==1)
        K = cov_linear(X1, X2, a);                // K = a^2*X1*X2.T
        else if (kernel==2){
            K = cov_exp_quad(X1, X2, sigma, l);
            for (n in 1:N)
                K[n,n] = K[n,n] + jitter;
            K = quad_form_diag(K, diag_stds);}
        else if (kernel==3){
```

```
83              K = cov_exp(X1, X2, sigma, l, jitter);
84              K = quad_form_diag(K, diag_stds);}
85          else if (kernel==4){
86              K = cov_matern32(X1, X2, sigma, l, jitter);
87              K = quad_form_diag(K, diag_stds);}
88          else if (kernel==5){
89              K = cov_matern52(X1, X2, sigma, l, jitter);
90              K = quad_form_diag(K, diag_stds);}
91          return K;
92      }
93  }
94  data {
95      int<lower=1> N;
96      int<lower=1> D;
97      int<lower=1> Q;
98      matrix[N,D] Y;
99      int<lower=1,upper=5> kernel;
100     real<lower=0> jitter;
101 }
102 transformed data {
103     vector[N] mu = rep_vector(0, N);
104 }
105 parameters {
106     //vector[N] mu;
107     vector[Q] X[N];                          // latent space
108     real<lower=0> kernel_lengthscale;        // kernel lengthscale
109     vector<lower=0>[N] diag_stds;            // std for each stock
110     vector<lower=0>[N] noise_std;            // observation noise ... non isotropic a la factor model
111     real<lower=0> alpha;                     // kernel std for linear kernel
112 }
113 transformed parameters {
114     matrix[N,N] L;
115     real R2 = 0;
116     {
117         matrix[N,N] K = kernel_f(X, X, 1., kernel_lengthscale,
118         alpha, kernel, diag_stds, jitter);
119
120         for (n in 1:N)
121         K[n,n] = K[n,n] + pow(noise_std[n], 2) + jitter;
122         L = cholesky_decompose(K);
123
124         R2 = sum(1 - square(noise_std) ./diagonal(K) )/N;
125     }
126 }
127 model {
128     for (n in 1:N)
129     X[n] ~ normal(0, 1);
130
131     diag_stds ~ normal(0, .5);
132     noise_std ~ normal(0, .5);
133     kernel_lengthscale ~ inv_gamma(3.0,1.0);// inv_gamma for zero-avoiding prop
134     alpha ~ inv_gamma(3.0,1.0);              // kernel std for linear kernel
135
136     for (d in 1:D)
137     col(Y,d) ~ multi_normal_cholesky(mu, L);
138 }
139 generated quantities {
140     matrix[N,N] K = kernel_f(X, X, 1., kernel_lengthscale,
141     alpha, kernel, diag_stds, jitter);
142 }
```

6.2.2 gplvm_vola.stan

```
functions {
    matrix cov_linear(vector[] X1, vector[] X2, real sigma){
        int N = size(X1);
        int M = size(X2);
        int Q = num_elements(X1[1]);
        matrix[N,M] K;
        {
            matrix[N,Q] x1;
            matrix[M,Q] x2;
            for (n in 1:N)
            x1[n,] = X1[n]';
            for (m in 1:M)
            x2[m,] = X2[m]';
            K = x1*x2';
        }
        return square(sigma)*K;
    }

    matrix cov_matern32(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        for (n in 1:N)
        for (m in 1:M){
            dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
            K[n,m] = square(sigma)*(1+sqrt(3)*dist/l)*exp(-sqrt(3)*dist/l);}
        return K;
    }

    matrix cov_matern52(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        for (n in 1:N)
        for (m in 1:N){
            dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
            K[n,m] = square(sigma)*(1+sqrt(5)*dist/l+5*square(dist)/(3*square(l)))*exp(-sqrt(5)*dist/l);}
        return K;
    }

    matrix cov_exp_l2(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        for (n in 1:N)
        for (m in 1:M){
            dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
            K[n,m] = square(sigma) * exp(-0.5/l * dist);}
        return K;
    }

    matrix cov_exp(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
        for (m in 1:M){
            dist = 0;  //sqrt(squared_distance(X1[n], X2[m]) + jitter);
            for (i in 1:Q)
            dist = dist + fabs(X1[n,i] - X2[m,i]);
            K[n,m] = square(sigma) * exp(-0.5/l * dist);}
        return K;
    }

    matrix kernel_f(vector[] X1, vector[] X2, real sigma, real l,
    real a, int kernel, vector diag_stds, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        if (kernel==1)
        K = cov_linear(X1, X2, a);                // K = a^2*X1*X2.T
        else if (kernel==2){
            K = cov_exp_quad(X1, X2, sigma, l);
            for (n in 1:N)
            K[n,n] = K[n,n] + jitter;
            K = quad_form_diag(K, diag_stds);}
        else if (kernel==3){
            K = cov_exp(X1, X2, sigma, l, jitter);
            K = quad_form_diag(K, diag_stds);}
        else if (kernel==4){
            K = cov_matern32(X1, X2, sigma, l, jitter);
```

```
87              K = quad_form_diag(K, diag_stds);}
88          else if (kernel==5){
89              K = cov_matern52(X1, X2, sigma, l, jitter);
90              K = quad_form_diag(K, diag_stds);}
91          return K;
92      }
93  }
94  data {
95      int<lower=1> N;                         // number of stocks
96      int<lower=1> D;                         // number of days
97      int<lower=1> Q;                         // number of latent dimensions
98      matrix[N,D] Y;                          // input Data
99      int<lower=1,upper=5> kernel;
100     real<lower=0> jitter;
101 }
102 transformed data {
103     vector[N] mu = rep_vector(0, N);
104 }
105 parameters {
106     vector[Q] X[N];
107     vector<lower=0>[N] noise_std;
108     real<lower=0> kernel_l;
109     matrix<lower=0>[N,D] Sigma;
110 }
111 transformed parameters{
112     matrix[N,N] K;
113     matrix[N,N] L_;
114     matrix[N,N] L[D];
115
116     K = kernel_f(X, X, kernel_l,  kernel, rep_vector(0,N), jitter);
117     L_ = cholesky_decompose(K);
118
119     {
120     for (d in 1:D){
121             L[d] = diag_post_multiply(L_, Sigma[:,d]);
122         }
123     }
124 }
125 model {
126     for (n in 1:N){               //latent space process
127         X[n] ~ normal(0.0, 1.0);
128         Sigma[n] ~ normal(0.0, 1.0);
129     }{
130         noise_std ~ normal(0.0, 1.0);
131         kernel_l ~ normal(0.0, 1.0);
132     }
133     for (d in 1:D)
134         col(Y,d) ~ multi_normal_cholesky(mu, L[d]);
135 }
```

6.2.3 gplvm_time.stan

```
functions{
    matrix cov_lin(vector[] X1, vector[] X2, real sigma){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        {
            for (n in 1:N){
                for (m in 1:M){
                    K[n,m] = square(sigma) * X1[n,n] * X2[m,m];
                }
            }
        }
        return K;
    }

    matrix cov_linear(vector[] X1, vector[] X2, real sigma){
        int N = size(X1);
        int M = size(X2);
        int Q = rows(X1[1]);
        matrix[N,M] K;
        {matrix[N,Q] x1;
        matrix[M,Q] x2;
        for (n in 1:N)
            x1[n,:] = X1[n]';                //'=Transpose
        for (m in 1:M)
            x2[m,:] = X2[m]';
        K = x1*x2';}
        return square(sigma)*K;
    }

    matrix cov_exp(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
            for (m in 1:M){
                dist = 0;
                for (q in 1:Q)
                    dist = dist + fabs(X1[n,q] - X2[m,q]);
        K[n,m] = square(sigma) * exp(-0.5/l * dist);}
        return K;
    }

    matrix cov_exp_2(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
            for (m in 1:M){
                dist = 0;
                for (q in 1:Q)
                    dist = dist + square(fabs(X1[n,q] - X2[m,q]));
        K[n,m] = square(sigma) * exp(-0.5/l * dist);}
        return K;
    }

    matrix cov_matern32(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
            for (m in 1:M){
                dist = 0;
                for (q in 1:Q)
                    dist = dist + fabs(X1[n,q] - X2[m,q]);
        K[n,m] = square(sigma)*(1+sqrt(3)*dist/l)*exp(-sqrt(3)*dist/l);}
        return K;
    }
    matrix cov_matern52(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
            for (m in 1:N){
                dist = 0;
                for (q in 1:Q)
                    dist = dist + fabs(X1[n,q] - X2[m,q]);
        K[n,m] = square(sigma)*(1+sqrt(5)*dist/l+5*square(dist)/(3*square(l)))*exp(-sqrt(5)*dist/l);}
```

```
87          return K;
88      }
89
90      matrix kernel_f_time(vector[] X1, vector[] X2, real l, real sigma, real jitter, int kernel_n){
91          int D = size(X1);
92          matrix[D,D] L;
93          matrix[D,D] K;
94          if (kernel_n==1){
95              K = cov_lin(X1, X2, l);
96              for (d in 1:D)
97                  K[d,d] = K[d,d] + pow(sigma, 2) + jitter;
98              L = cholesky_decompose(K);}
99          else if (kernel_n==2){
100             K = cov_exp_2(X1, X2, 1., l, jitter);
101             for (d in 1:D)
102                 K[d,d] = K[d,d] + pow(sigma, 2) + jitter;
103             L = cholesky_decompose(K);}
104         else if (kernel_n==3){
105             K = cov_exp(X1, X2, 1., l, jitter);
106             for (d in 1:D)
107                 K[d,d] = K[d,d] + pow(sigma, 2) + jitter;
108             L = cholesky_decompose(K);}
109         else if (kernel_n==4){
110             K = cov_matern32(X1, X2, 1., l, jitter);
111             for (d in 1:D)
112                 K[d,d] = K[d,d] + pow(sigma, 2) + jitter;
113             L = cholesky_decompose(K);}
114         else if (kernel_n==5){
115             K = cov_matern52(X1, X2, 1., l, jitter);
116             for (d in 1:D)
117                 K[d,d] = K[d,d] + pow(sigma, 2) + jitter;
118             L = cholesky_decompose(K);}
119         return L;
120     }
121     matrix kernel_f_y(vector[] X1, vector[] X2, real l, vector sigma, vector noise, real jitter, int kernel_n){
122         int N = size(X1);
123         matrix[N,N] K;
124         matrix[N,N] L;
125         if (kernel_n==1){
126             K = cov_linear(X1, X2, l);
127             K = quad_form_diag(K, sigma);
128             for (n in 1:N){
129                 K[n,n] = K[n,n] + pow(noise[n], 2) + jitter;}
130             L = cholesky_decompose(K);}
131         else if (kernel_n==2){
132             K = cov_exp_2(X1, X2, 1., l, jitter);
133             K = quad_form_diag(K, sigma);
134             for (n in 1:N){
135                 K[n,n] = K[n,n] + pow(noise[n], 2) + jitter;}
136             L = cholesky_decompose(K);}
137         else if (kernel_n==3){
138             K = cov_exp(X1, X2, 1., l, jitter);
139             K = quad_form_diag(K, sigma);
140             for (n in 1:N){
141                 K[n,n] = K[n,n] + pow(noise[n], 2) + jitter;}
142             L = cholesky_decompose(K);}
143         else if (kernel_n==4){
144             K = cov_matern32(X1, X2, 1., l, jitter);
145             K = quad_form_diag(K, sigma);
146             for (n in 1:N){
147                 K[n,n] = K[n,n] + pow(noise[n], 2) + jitter;}
148             L = cholesky_decompose(K);}
149         else if (kernel_n==5){
150             K = cov_matern52(X1, X2, 1., l, jitter);
151             K = quad_form_diag(K, sigma);
152             for (n in 1:N){
153                 K[n,n] = K[n,n] + pow(noise[n], 2) + jitter;}
154             L = cholesky_decompose(K);}
155         return L;}
156 }
157 data {
158     // data and model parameters
159     int<lower=1> N;
160     int<lower=1> D;
161     int<lower=1> Q;
162     matrix[N,D] Y;
163     int<lower=1,upper=5> kernel_number_x;
164     int<lower=1,upper=5> kernel_number_y;
165     real<lower=0> jitter;
166     // lengthscale dist parameters
167     real<lower=0> alpha_x;
168     real<lower=0> beta_x;
169     real<lower=0> alpha_y;
170     real<lower=0> beta_y;
171     // standard deviation parameters
172     real <lower=0,upper=1> std_x;
173     real <lower=0,upper=0.1> noise_x;
174     real <lower=0,upper=1> std_y;
175     real <lower=0,upper=0.1> noise_y;
```

```
}
transformed data {
    vector[N] zeros_N = rep_vector(0, N);
    vector[D] zeros_D = rep_vector(0, D);
    vector[D] time[D];
    for (d in 1:D){
        time[d] = rep_vector(d,d);
    }
}
parameters {
    // X
    vector[Q] X[D,N];
    real<lower=0> kernel_lengthscale_x;
    real<lower=0> kernel_std_x;
    real<lower=0> noise_std_x;
    // Y
    real<lower=0> kernel_lengthscale_y;
    vector<lower=0>[N] diag_std_y;
    vector<lower=0>[N] noise_std_y;
}
transformed parameters {
    real R2 = mean(square(diag_std_y)./(square(diag_std_y)+square(noise_std_y)));
}
model {
    // X
    kernel_lengthscale_x ~ gamma(alpha_x, beta_x);
    kernel_std_x ~ normal(0, std_x);
    noise_std_x ~ normal(0, noise_x);
    // Y
    kernel_lengthscale_y ~ inv_gamma(alpha_y, beta_y);
    diag_std_y ~ normal(0, std_y);
    noise_std_y ~ normal(0, noise_y);
    {
        matrix[D,D] L_x;
        L_x = kernel_f_time(time, time, kernel_lengthscale_x, noise_std_x, jitter, kernel_number_x);
        for (q in 1:Q)
        for (n in 1:N)
        to_vector(X[:,n,q]) ~ multi_normal_cholesky(zeros_D, L_x);
    }
    {
        matrix[N,N] L_y[D];
        matrix[N,N] K_y;
        //vector[Q] X[D,N];
        for (d in 1:D) {
            L_y[d] = kernel_f_y(X[d], X[d], kernel_lengthscale_y, diag_std_y, rep_vector(0,N), jitter, kernel_number_y);
        for (d in 1:D)
        col(Y,d) ~ multi_normal_cholesky(zeros_N, L_y[d]);
    }
}
generated quantities {
    matrix[N,N] K_y[D];

    for (d in 1:D){
        K_y[d] = kernel_f_y(X[d], X[d], kernel_lengthscale_y, diag_std_y, rep_vector(0,N), jitter, kernel_number_y);}
}
```

6.2.4 student-t.stan

```
functions {
    matrix cov_linear(vector[] X1, vector[] X2, real sigma){
        int N = size(X1);
        int M = size(X2);
        int Q = num_elements(X1[1]);
        matrix[N,M] K;
        {
            matrix[N,Q] x1;
            matrix[M,Q] x2;
            for (n in 1:N)
            x1[n,] = X1[n]';
            for (m in 1:M)
            x2[m,] = X2[m]';
            K = x1*x2';
        }
        return square(sigma)*K;
    }

    matrix cov_matern32(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        for (n in 1:N)
        for (m in 1:M){
            dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
            K[n,m] = square(sigma)*(1+sqrt(3)*dist/l)*exp(-sqrt(3)*dist/l);}
        return K;
    }

    matrix cov_matern52(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        for (n in 1:N)
        for (m in 1:M){
            dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
            K[n,m] = square(sigma)*(1+sqrt(5)*dist/l+5*square(dist)/(3*square(l)))*exp(-sqrt(5)*dist/l);}
        return K;
    }

    matrix cov_exp_l2(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        for (n in 1:N)
        for (m in 1:M){
            dist = sqrt(squared_distance(X1[n], X2[m]) + jitter);
            K[n,m] = square(sigma) * exp(-0.5/l * dist);}
        return K;
    }

    matrix cov_exp(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
        for (m in 1:M){
            dist = 0;  //sqrt(squared_distance(X1[n], X2[m]) + jitter);
            for (i in 1:Q)
            dist = dist + fabs(X1[n,i] - X2[m,i]);
            K[n,m] = square(sigma) * exp(-0.5/l * dist);}
        return K;
    }

    matrix kernel_f(vector[] X1, vector[] X2, real sigma, real l,
    real a, int kernel, vector diag_stds, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        if (kernel==1)
        K = cov_linear(X1, X2, a);                  // K = a^2*X1*X2.T
        else if (kernel==2){
            K = cov_exp_quad(X1, X2, sigma, l);
            for (n in 1:N)
            K[n,n] = K[n,n] + jitter;
            K = quad_form_diag(K, diag_stds);}
        else if (kernel==3){
            K = cov_exp(X1, X2, sigma, l, jitter);
            K = quad_form_diag(K, diag_stds);}
        else if (kernel==4){
            K = cov_matern32(X1, X2, sigma, l, jitter);
```

```
87              K = quad_form_diag(K, diag_stds);}
88          else if (kernel==5){
89              K = cov_matern52(X1, X2, sigma, l, jitter);
90              K = quad_form_diag(K, diag_stds);}
91          return K;
92      }
93  }
94  data {
95      int<lower=1> N;                         // number of stocks
96      int<lower=1> D;                         // number of days
97      int<lower=1> Q;                         // number of latent dimensions
98      matrix[N,D] Y;                          // input Data
99      int<lower=1,upper=5> kernel;     // used by function 'kernel_f()' for model choice
100     real<lower=0> jitter;
101 }
102 transformed data {
103     vector[N] mu = rep_vector(0, N);
104 }
105 parameters {
106     vector[Q] X[N];
107     real<lower=0> kernel_lengthscale;
108     vector<lower=0>[N] diag_stds;
109     vector<lower=0>[N] noise_std;
110     real<lower=0> alpha;
111     real<lower=2> t_nu;
112 }
113 transformed parameters {
114     real R2 = 0;
115     matrix[N,N] K = kernel_f(X, X, 1., kernel_lengthscale, alpha, kernel, diag_stds, jitter);
116     for (n in 1:N)
117         K[n,n] = K[n,n] + pow(noise_std[n], 2) + jitter;
118     R2 = sum(1 - square(noise_std) ./diagonal(K) )/N;
119 }
120 model {
121     diag_stds ~ normal(0, .5);
122     noise_std ~ normal(0, .5);
123     kernel_lengthscale ~ inv_gamma(3.0,1.0);
124     alpha ~ inv_gamma(3.0,1.0);
125     for (n in 1:N)
126         X[n] ~ normal(0, 1);
127     for (d in 1:D)
128         col(Y,d) ~ multi_student_t(t_nu, mu, K);
129 }
```

6.2.5 gplvm_time-studt.stan

```
functions{
    matrix cov_lin(vector[] X1, vector[] X2, real sigma){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        {
            for (n in 1:N){
                for (m in 1:M){
                    K[n,m] = square(sigma) * X1[n,n] * X2[m,m];
                }
            }
        }
        return K;
    }

    matrix cov_linear(vector[] X1, vector[] X2, real sigma){
        int N = size(X1);
        int M = size(X2);
        int Q = rows(X1[1]);
        matrix[N,M] K;
        {
            matrix[N,Q] x1;
            matrix[M,Q] x2;
            for (n in 1:N)
            x1[n,:] = X1[n]';                    //'=Transpose
            for (m in 1:M)
            x2[m,:] = X2[m]';
            K = x1*x2';
        }
        return square(sigma)*K;
    }

    matrix cov_exp(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
        for (m in 1:M){
            dist = 0;  //sqrt(squared_distance(X1[n], X2[m]) + jitter);
            for (q in 1:Q)
            dist = dist + fabs(X1[n,q] - X2[m,q]);
            K[n,m] = square(sigma) * exp(-0.5/l * dist);}
        return K;
    }

    matrix cov_exp_2(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
        for (m in 1:M){
            dist = 0;  //sqrt(squared_distance(X1[n], X2[m]) + jitter);
            for (q in 1:Q)
            dist = dist + square(fabs(X1[n,q] - X2[m,q]));
            K[n,m] = square(sigma) * exp(-0.5/l * dist);}
        return K;
    }

    matrix cov_matern32(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
        for (m in 1:M){
            dist = 0;  //sqrt(squared_distance(X1[n], X2[m]) + jitter);
            for (q in 1:Q)
            dist = dist + fabs(X1[n,q] - X2[m,q]);
            K[n,m] = square(sigma)*(1+sqrt(3)*dist/l)*exp(-sqrt(3)*dist/l);}
        return K;
    }
    matrix cov_matern52(vector[] X1, vector[] X2, real sigma, real l, real jitter){
        int N = size(X1);
        int M = size(X2);
        matrix[N,M] K;
        real dist;
        int Q = rows(X1[1]);
        for (n in 1:N)
        for (m in 1:N){
            dist = 0;  //sqrt(squared_distance(X1[n], X2[m]) + jitter);
            for (q in 1:Q)
```

```
87              dist = dist + fabs(X1[n,q] - X2[m,q]);
88              K[n,m] = square(sigma)*(1+sqrt(5)*dist/l+5*square(dist)/(3*square(l)))*exp(-sqrt(5)*dist/l);}
89          return K;
90      }
91
92      matrix kernel_f_time(vector[] X1, vector[] X2, real l, real sigma, real jitter, int kernel_n){
93          int D = size(X1);
94          matrix[D,D] K;
95          if (kernel_n==1){
96              K = cov_lin(X1, X2, l);
97              for (d in 1:D)
98              K[d,d] = K[d,d] + pow(sigma, 2) + jitter;}
99          else if (kernel_n==2){
100             K = cov_exp_2(X1, X2, 1., l, jitter);
101             for (d in 1:D)
102             K[d,d] = K[d,d] + pow(sigma, 2) + jitter;}
103         else if (kernel_n==3){
104             K = cov_exp(X1, X2, 1., l, jitter);
105             for (d in 1:D)
106             K[d,d] = K[d,d] + pow(sigma, 2) + jitter;}
107         else if (kernel_n==4){
108             K = cov_matern32(X1, X2, 1., l, jitter);
109             for (d in 1:D)
110             K[d,d] = K[d,d] + pow(sigma, 2) + jitter;}
111         else if (kernel_n==5){
112             K = cov_matern52(X1, X2, 1., l, jitter);
113             for (d in 1:D)
114             K[d,d] = K[d,d] + pow(sigma, 2) + jitter;}
115         return K;
116     }
117
118     matrix kernel_f_y(vector[] X1, vector[] X2, real l, vector sigma, vector noise, real jitter, int kernel_n){
119         int N = size(X1);
120         matrix[N,N] K;
121         if (kernel_n==1){
122             K = cov_linear(X1, X2, l);
123             K = quad_form_diag(K, sigma);
124             for (n in 1:N){
125                 K[n,n] = K[n,n] + pow(noise[n], 2) + jitter;}}
126         else if (kernel_n==2){
127             K = cov_exp_2(X1, X2, 1., l, jitter);
128             K = quad_form_diag(K, sigma);
129             for (n in 1:N){
130                 K[n,n] = K[n,n] + pow(noise[n], 2) + jitter;}}
131         else if (kernel_n==3){
132             K = cov_exp(X1, X2, 1., l, jitter);
133             K = quad_form_diag(K, sigma);
134             for (n in 1:N){
135                 K[n,n] = K[n,n] + pow(noise[n], 2) + jitter;}}
136         else if (kernel_n==4){
137             K = cov_matern32(X1, X2, 1., l, jitter);
138             K = quad_form_diag(K, sigma);
139             for (n in 1:N){
140                 K[n,n] = K[n,n] + pow(noise[n], 2) + jitter;}}
141         else if (kernel_n==5){
142             K = cov_matern52(X1, X2, 1., l, jitter);
143             K = quad_form_diag(K, sigma);
144             for (n in 1:N){
145                 K[n,n] = K[n,n] + pow(noise[n], 2) + jitter;}}
146         return K;
147     }
148 }
149 data {
150     // data and model parameters
151     int<lower=1> N;
152     int<lower=1> D;
153     int<lower=1> Q;
154     matrix[N,D] Y;
155     int<lower=1,upper=5> kernel_number_x;
156     int<lower=1,upper=5> kernel_number_y;
157     real<lower=0> jitter;
158     // lengthscale dist parameters
159     real<lower=0> alpha_x;
160     real<lower=0> beta_x;
161     real<lower=0> alpha_y;
162     real<lower=0> beta_y;
163     // standard deviation parameters
164     real <lower=0,upper=1> std_x;
165     real <lower=0,upper=0.1> noise_x;
166     real <lower=0,upper=1> std_y;
167     real <lower=0,upper=0.1> noise_y;
168 }
169 transformed data {
170     vector[N] zeros_N = rep_vector(0, N);
171     vector[D] zeros_D = rep_vector(0, D);
172     vector[D] time[D];
173     for (d in 1:D){
174         time[d] = rep_vector(d,d);}
175 }
```

```
parameters{
    // X
    vector[Q] X[D,N];
    real<lower=0> kernel_lengthscale_x;
    real<lower=0> kernel_std_x;
    real<lower=0> noise_std_x;
    real<lower=2> t_nu_x;
    // Y
    real<lower=0> kernel_lengthscale_y;
    vector<lower=0>[N] diag_std_y;
    vector<lower=0>[N] noise_std_y;
    real<lower=2> t_nu_y;
}
transformed parameters {
    real R2 = mean(square(diag_std_y)./(square(diag_std_y)+square(noise_std_y)));
}
model {
    // X
    kernel_lengthscale_x ~ gamma(alpha_x, beta_x);
    kernel_std_x ~ normal(0, std_x);
    noise_std_x ~ normal(0, noise_x);
    // Y
    kernel_lengthscale_y ~ inv_gamma(alpha_y, beta_y);
    diag_std_y ~ normal(0, std_y);
    noise_std_y ~ normal(0, noise_y);
    //prior on X
    {
        matrix[D,D] K_x;
        K_x = kernel_f_time(time, time, kernel_lengthscale_x, noise_std_x, jitter, kernel_number_x);
        for (q in 1:Q)
            for (n in 1:N)
                to_vector(X[:,n,q]) ~ multi_student_t(t_nu_x, zeros_D, K_x);
    }
    //likelihood
    {
        matrix[N,N] K_y[D];
        for (d in 1:D) {
            K_y[d] = kernel_f_y(X[d], X[d], kernel_lengthscale_y, diag_std_y, rep_vector(0,N), jitter, kernel_number_y);
        }
        for (d in 1:D)
            col(Y,d) ~ multi_student_t(t_nu_y, zeros_N, K_y[d]);
    }
}
```

Bibliography

[1] Tipping, Michael E. ; Bishop, Christopher M.: Probabilistic Principal Component Analysis. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 61 (1999), Nr. 3, 611-622. http://dx.doi.org/10.1111/1467-9868.00196. – DOI 10.1111/1467–9868.00196

[2] Wiener, N.: *Extrapolation, Interpolation, and Smoothing of Stationary Time Series*. MIT press Cambridge, 1949

[3] Kolmogoroff, A.: Interpolation and extrapolation stationary random sequences. In: *Izv. Akad. Nauk SSSR Ser. Mat.* 5 (1941), Nr. 1, 3-14. http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=im&paperid=3775&option_lang=eng

[4] Thiele, T. N.: ANALYTISKE STUDIER OM DEN RENE MATHEMATIKS PRINCIPER. In: *Tidsskrift for mathematik* 4 (1880), 33–62. http://www.jstor.org/stable/24539888. – ISSN 09092528, 24460737

[5] Cox, D. R. ; Gudmundsson, Gudmundur ; Lindgren, Georg ; Bondesson, Lennart ; Harsaae, Erik ; Laake, Petter ; Juselius, Katarina ; Lauritzen, Steffen L.: Statistical Analysis of Time Series: Some Recent Developments [with Discussion and Reply]. In: *Scandinavian Journal of Statistics* 8 (1981), Nr. 2, 93–115. http://www.jstor.org/stable/4615819. – ISSN 03036898, 14679469

[6] Matheron, G.: The intrinsic random functions and their applications. In: *Advances in Applied Probability* 5 (1973), Nr. 3, S. 439–468. http://dx.doi.org/10.2307/1425829. – DOI 10.2307/1425829

[7] AG Journel, CJ H.: *Mining geostatistics*. Jan 1976

[8] Ripley, B.D.: *Spatial Statistics*. Wiley, 2005 (Wiley Series in Probability and Statistics). https://books.google.de/books?id=BDDPTdohXeYC. – ISBN 9780471725206

[9] Jay M. Ver Hoef, Noel C.: Multivariate Spatial prediction. In: *Mathematical Geology* 25 (1993), 219-240. http://dx.doi.org/10.1007/BF00893273. – DOI 10.1007/BF00893273

[10] CKI Williams, CE R.: Gaussian processes for regression. In: *Advances in Neural Information Processing Systems* 8 (1996), S. 514–520

[11] Neal, RM: Bayesian Learning for Neural Networks. In: *Lecture Notes in Statistics* 118 (1996)

[12] Nirwan, Rajbir-Singh ; Bertschinger, Nils: *Applications of Gaussian Process Latent Variable Models in Finance*. 2019

[13] Markowitz, Harry: PORTFOLIO SELECTION*. In: *The Journal of Finance* 7 (1952), Nr. 1, 77-91. http://dx.doi.org/10.1111/j.1540-6261.1952.tb01525.x. – DOI 10.1111/j.1540–6261.1952.tb01525.x

[14] Jobson, J D. ; Korkie, Bob M.: Performance Hypothesis Testing with the Sharpe and Treynor Measures. In: *Journal of Finance* 36 (1981), Nr. 4, 889-908. https://EconPapers.repec.org/RePEc:bla:jfinan:v:36:y:1981:i:4:p:889-908

[15] SHARPE, WILLIAM F.: Capital Asset Prices with and without Negative Holdings. In: *The Journal of Finance* 46 (1991), Nr. 2, 489-509. http://dx.doi.org/10.1111/j.1540-6261.1991.tb02671.x. – DOI 10.1111/j.1540–6261.1991.tb02671.x

[16] Ledoit, Olivier ; Wolf, Michael: Honey, I Shrunk the Sample Covariance Matrix. In: *The Journal of Portfolio Management* 30 (2004), Nr. 4, 110–119. http://dx.doi.org/10.3905/jpm.2004.110. – DOI 10.3905/jpm.2004.110. – ISSN 0095–4918

[17] Nevmyvaka, Yuriy ; Feng, Yi ; Kearns, Michael: Reinforcement Learning for Optimized Trade Execution. In: *Proceedings of the 23rd International Conference on Machine Learning*. New York, NY, USA : Association for Computing Machinery, 2006 (ICML '06). – ISBN 1595933832, 673–680

[18] Gately, Edward: *Neural Networks for Financial Forecasting*. 1995. – ISBN 0471112127

[19] Chapados, Nicolas ; Bengio, Yoshua: Augmented functional time series representation and forecasting with Gaussian processes. In: *Advances in neural information processing systems*, 2008, S. 265–272

[20] Heaton, J. B. ; Polson, N. G. ; Witte, J. H.: *Deep Learning in Finance*. 2018

[21] Wilson, Andrew G. ; Knowles, David A. ; Ghahramani, Zoubin: *Gaussian Process Regression Networks*. 2011

[22] Lawrence, Neil: Probabilistic Non-Linear Principal Component Analysis with Gaussian Process Latent Variable Models. In: *J. Mach. Learn. Res.* 6 (2005), Dezember, S. 1783–1816. – ISSN 1532–4435

[23] Rasmussen, Carl E.: *Gaussian processes for machine learning*. MIT Press, 2006

[24] Thompson, MJ Buckle; M Buckle; J.: *The UK Financial System: Fourth Edition*. Manchester University Press, 2018

[25] *Kapitel* 1. In: Kuhn H.W., Tucker A.: *Nonlinear Programming*. Birkhäuser, 2014

[26] *Kapitel* Chapter 1. In: ROSS, STEPHEN A.: *The Arbitrage Theory of Capital Asset Pricing*. 2013, 11-30

[27] Roll, Richard ; Ross, Stephen A.: The Arbitrage Pricing Theory Approach to Strategic Portfolio Planning. In: *Financial Analysts Journal* 51 (1995), Nr. 1, 122-131. http://dx.doi.org/10.2469/faj.v51.n1.1868. – DOI 10.2469/faj.v51.n1.1868

[28] Cochrane, J.H. ; Fama, E.F. ; Moskowitz, T.J.: *The Fama Portfolio: Selected Papers of Eugene F. Fama*. University of Chicago Press, 2017 (Online access with DDA: Askews). https://books.google.de/books?id=S4M4DwAAQBAJ. – ISBN 9780226426846

[29] FAMA, EUGENE F. ; FRENCH, KENNETH R.: Multifactor Explanations of Asset Pricing Anomalies. In: *The Journal of Finance* 51 (1996), Nr. 1, 55-84. http://dx.doi.org/10.1111/j.1540-6261.1996.tb05202.x. – DOI 10.1111/j.1540–6261.1996.tb05202.x

[30] EF Farma, KR F.: The Capital Asset Pricing Model: Theory and Evidence. In: *Journal of Economic Perspectives* 18 (2004), Nr. 3, S. 25–46

[31] Everitt, B.: An Introduction to Latent Variable Models. In: *An Introduction to Latent Variable Models*, 1984, S. 13–31

[32] Bishop, Christopher M.: *Pattern recognition and machine learning*. New York, NY : Springer, 2006 (Information science and statistics). https://cds.cern.ch/record/998831. – Softcover published in 2016

[33] Tipping, Michael E. ; Bishop, Christopher M.: Mixtures of Probabilistic Principal Component Analyzers. In: *Neural Computation* 11 (1999), Nr. 2, 443-482. http://dx.doi.org/10.1162/089976699300016728. – DOI 10.1162/089976699300016728

[34] Tipping, Michael E. ; Bishop, Christopher M.: Probabilistic Principal Component Analysis. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 61 (1999), Nr. 3, 611-622. http://dx.doi.org/10.1111/1467-9868.00196. – DOI 10.1111/1467–9868.00196

[35] Amar Shah, Zoubin G. Andrew Gordon Wilson W. Andrew Gordon Wilson: Student-t Processes as Alternatives to Gaussian Processes. In: *Journal of Machine Learning Research* 33 (2014)

[36] DAWID, A. P.: Some matrix-variate distribution theory: Notational considerations and a Bayesian application. In: *Biometrika* 68 (1981), 04, Nr. 1, 265-274. http://dx.doi.org/10.1093/biomet/68.1.265. – DOI 10.1093/biomet/68.1.265. – ISSN 0006–3444

[37] *Kapitel* 1. In: Geyer, Chalres J.: *Introduction to Markov Chain Monte Carlo*. 2011, S. 3–48

[38] C Andrieu, et a. MI Jordan J. MI Jordan: An Introduction to MCMC for Machine Learning. In: *Machine Learning* 50 (2003), S. 5–43

[39] Duane, Simon ; Kennedy, A.D. ; Pendleton, Brian J. ; Roweth, Duncan: Hybrid Monte Carlo. In: *Physics Letters B* 195 (1987), Nr. 2, 216 - 222. http://dx.doi.org/https://doi.org/10.1016/0370-2693(87)91197-X. – DOI https://doi.org/10.1016/0370–2693(87)91197–X. – ISSN 0370–2693

[40] Kucukelbir, Alp ; Ranganath, Rajesh ; Gelman, Andrew ; Blei, David: Automatic Variational Inference in Stan. Version: 2015. http://papers.nips.cc/paper/5758-automatic-variational-inference-in-stan.pdf. In: Cortes, C. (Hrsg.) ; Lawrence, N. D. (Hrsg.) ; Lee, D. D. (Hrsg.) ; Sugiyama, M. (Hrsg.) ; Garnett, R. (Hrsg.): *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc., 2015, 568–576

[41] Team, Stan D.: *Stan Modeling Language Users Guide and Reference Manual*. 2.18.0, 2018. http://mc-stan.org

[42] John Duchi, Yoram S. Elad Hazan H. Elad Hazan: Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. In: *Journal of Machine Learning Research* 12 (2011), S. 2121–2159

[43] Deutschen Normenausschuss (DNA), Fachnormenaussschuss Informationsverarbeitung (.: Sinnbilder fuer Datenfluss und Programmablaufplaene. In: *Deutsche Industrienorm* (1966)

[44] https://de.finance.yahoo.com/

[45] Bob Carpenter, et a.: Stan: A probabilistic programming language. In: *Journal of Statistical Software* 76(1) (2017). http://dx.doi.org/10.18637/jss.v076.i01. – DOI 10.18637/jss.v076.i01

[46] Kumar, Ravin ; Carroll, Colin ; Hartikainen, Ari ; Martin, Osvaldo A.: ArviZ a unified library for exploratory analysis of Bayesian models in Python. In: *The Journal of Open Source Software* (2019). http://dx.doi.org/10.21105/joss.01143. – DOI 10.21105/joss.01143

[47] Taylor, Ross: *PyFlux: An open source time series library for Python*. 2016