The Fundamentals of Gaussian Processes for Regression Problems: Theory, Methodology and Application

James Michael Dodsworth

University of Birmingham

This dissertation is submitted for the Degree of Master of Science

Abstract

This dissertation explores the fundamentals of Gaussian processes for regression problems from three interconnected perspectives: theory, methodology, and application. Theoretical understanding is built through the presentation of key definitions, theorems, and proofs, while application is developed through the analysis of real-world datasets using Python code and visualisations. Methodologically, this work outlines the step-by-step process of applying Gaussian processes to regression problems, covering data transformation, kernel selection, optimization, and the computation of predictive mean and covariance. A generic methodology is stated early on and provides the cohesion for all subsequent chapters.

Throughout, the advantages of Gaussian processes are highlighted, with particular emphasis on their non-parametric nature, which allows for flexibility in modelling. This dissertation also explores the rich variety of kernels and their transformations, and discusses the Bayesian framework that supports Gaussian processes. In addition, model selection techniques are explored, ensuring that the most appropriate Gaussian process model is chosen for a given dataset. Ultimately, this comprehensive approach provides a robust understanding of Gaussian processes, demonstrating their power as a modelling tool for regression problems.

Contents

1 Introduction

1.1 Purpose

The purpose of this dissertation is to explore the fundamentals of Gaussian processes for regression problems in three distinct yet interconnected ways: theoretical, methodological and practical. The understanding of theory is achieved by the stating of definitions, theorems and equations accompanied with their rigorous proofs. The apprehension of practical application is accomplished by the analysis of real-world datasets demonstrated by relevant python code and figures. These two methods are both used in understanding the Gaussian process methodology.

This three-pronged approach allows for Gaussian processes to be scrutinised, determining the successes and pitfalls of such a modelling tool. Additionally, the holistic grasp of Gaussian processes insures the full comprehension of the reader. If in any doubt, the reader is directed appropriately to material that has inspired the work in this dissertation.

1.2 Outline

The Gaussian process journey starts with the wider context of Bayesian statistics and machine learning. Then a Gaussian process introduction is made with a generic methodology described that provides the context for the subsequent chapters. Next, covariance functions are examined, from simple to complex and their particular properties. Finally, model selection is investigated, specifically the choice of covariance functions and their hyperparameters. The dissertation concludes with a brief evaluation on the success of the paper achieving its aims. Moreover, areas of further research are put forth with their corresponding material.

The appendices are split into three: proof of results that were used to prove important claims within the field of Gaussian processes; python code that is of secondary importance in describing the creation of a plot or the application of a Gaussian process model to a dataset; and access to the datasets used throughout the paper. Appendix B is split into subsections that denote chapters. Python code used to produce plots in a given chapter will correspondingly be located in the given Appendix B subsection.

2 Preliminaries & the Definition of Gaussian Processes

2.1 Supervised Learning

Gaussian processes lie within a broader topic known as supervised learning. Supervised learning is the discipline of finding input-output mappings from a given dataset. Unlike unsupervised learning, the dataset includes both a target variable (output) and explanatory variables (inputs). Inputs are denoted as a vector $\mathbf{x}$ and the output is denoted as y. The inputs and output are compiled into a dataset of n observations denoted as $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$. (Williams and Rasmussen, 2006).

The approach of supervised learning is dependent on the type of data of the target variable (e.g. discrete, continuous, binary, text). The two common approaches are classification and regression. (Abu-Mostafa et al., 2012).

This paper focuses solely on Gaussian processes within regression problems. Because of this, input-output mappings can be considered as a continuous target function $f(\mathbf{x})$. Gaussian processes have been used for solving classification problems and for further detail on this topic see Williams and Rasmussen (2006).

2.2 Bayesian vs Frequentist

The following is inspired by Dr. Rowland Seymour's notes on Bayesian Inference and computation. (Seymour, 2023)

Bayesian Inference is a modern approach to statistical analysis compared to its traditional counterpart, Frequentist Theory, and is necessary for Gaussian process application. Therefore, it is prudent to identify the fundamental flaw of the latter to acknowledge the capability of the former.

The underlying principle of Frequentist theory is that the probability of event A is the relative frequency of event A (relative meaning with respect to other events B, C, ...) in the long run.

The following notation will be consistent in both frequentist theory and Bayesian inference.

Let θ be a parameter of a given model that aims to best explain the data $\mathcal{D}$. The parameter θ is unknown and is considered fixed under the frequentist paradigm. Therefore, a maximum likelihood estimate is used to estimate its value.

This distribution $p(\theta|\mathcal{D})$ describes the values that the parameter θ might more likely take given the dataset $\mathcal{D}$. Bayes' theorem allows the distribution to be written as an equation and therefore to be solved.

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta)p(\theta)}{p(\mathcal{D})} \tag{1}$$

The issue is that the distribution $p(\theta)$ is futile because the parameter θ is fixed.

Bayesian inference is different because a probability distribution is assigned to the parameter θ.

2.3 Bayesian Necessity

Before discussing the necessity of the Bayesian paradigm, the terms found in Bayes' theorem, Equation (1), are explained.

2.3.1 Bayes' Theorem

The probability distribution $p(\theta)$ is simply the statistician's subjective prior belief of how the parameter θ represents the true characteristics of a population, from which a dataset is drawn. Therefore, this distribution is known as the prior distribution and is decided before seeing the dataset. (Hoff, 2009).

$p(\mathcal{D} \mid \theta)$ is the likelihood function and describes the success of the parameter θ explaining the dataset, in particular how the inputs $\mathbf{x}$ map to the output y.

marginal likelihood $p(\mathcal{D})$, or evidence, represents the likelihood of observed data with respect to all possible parameter values. Since its calculation marginalises over all possible parameter values, marginal likelihood $p(\mathcal{D})$ can be used within model comparison. (Gelman et al., 1995):

$$p(\mathcal{D}) = \int p(\mathcal{D} \mid \theta)\, p(\theta)\, d\theta \tag{2}$$

A description of $p(\theta|\mathcal{D})$ is given in the previous section 2.2 and is known as the posterior distribution. Hoff states that the posterior distribution $p(\theta|\mathcal{D})$ "describes our belief that the parameter θ is true" (Hoff, 2009, pp. 2), given the dataset $\mathcal{D}$.

2.3.2 Trivial or Consequential?

The choice of a prior distribution $p(\theta)$ might be considered equally trivial compared to the Frequentist's notion of a fixed parameter θ. If this prior distribution must be chosen before looking at the dataset, there is a possibility that a statistician might truly have no belief on what the dataset represents. Bayesian statisticians admit to choosing a prior distribution in an impromptu way and sometimes based on the grounds of computational efficiency, (Hoff, 2009).

However, the choice of a prior distribution $p(\theta)$ over a fixed parameter θ is what enables Gaussian processes to be used. This paper will identify the benefits of Gaussian processes, from their non-parametric properties to their analytical tractability.

Another general advantage to Bayesian inference is the ability to obtain predictive uncertainty. The posterior distribution $p(\theta|\mathcal{D})$ can be found using Bayes' theorem, now that the prior distribution is determined. Any probability distribution must have a mean and variance. In the context of the posterior distribution $p(\theta|\mathcal{D})$, the variance can be used to assess how confident the model is in its predictions. A discussion on confidence or credible intervals is found in section 2.4.3.

2.4 Introduction to Gaussian Processes

2.4.1 Gaussian Evolution: from Univariate to Function Space

The following is inspired by Dr. Rowland Seymour's notes on Bayesian Inference and computation. (Seymour, 2023)

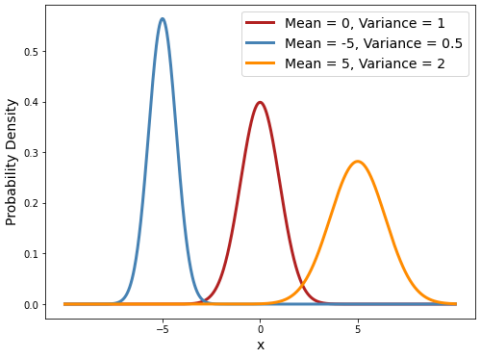
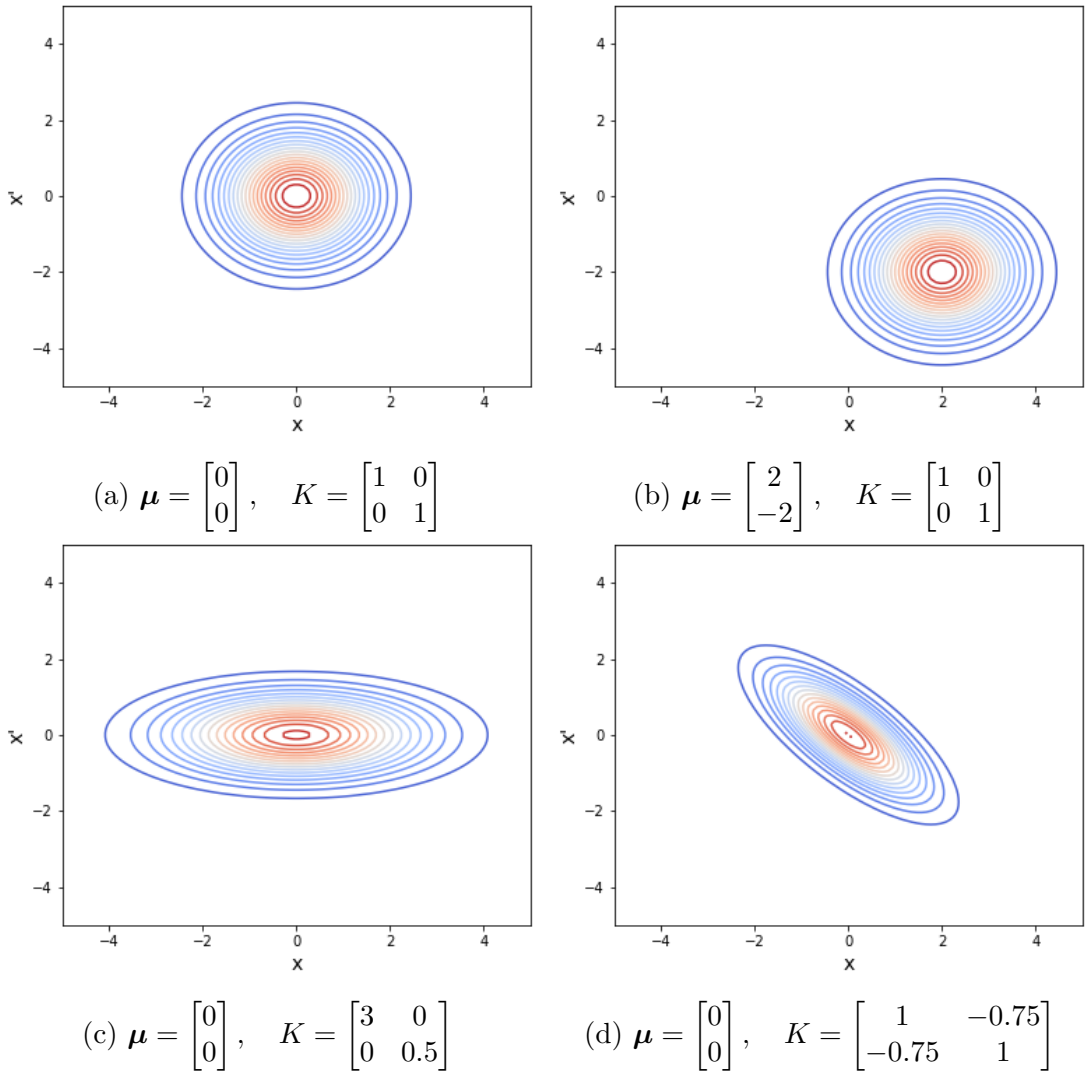


Figure 1: Plot of a univariate normal distribution with varying means and variances and demonstrating their effects on the distributions.

A Gaussian distribution has a mean μ and a variance σ^2 and assigns probabilities to values on the real line $\mathbb{R}$. The Gaussian distribution is colloquially known as the bell curve, and the mean determines the location of the peak of the bell, and the variance determines the width of the bell. These effects are illustrated in Figure 1.

Gaussian multivariate distribution is the Gaussian distribution extended to multiple dimensions $\mathbb{R}^N$. Rather than the mean and variance each taking a single value, they are now a mean vector $\boldsymbol{\mu}$ and a covariance matrix K. The mean vector represents the expected value of each individual component in the vector, while the covariance matrix captures the relationships between every pair of components in the vector. An example of a multivariate Gaussian distribution is shown graphically in Figure 2.



(a) $\boldsymbol{\mu} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

(b) $\boldsymbol{\mu} = \begin{bmatrix} 2 \\ -2 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

(c) $\boldsymbol{\mu} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad K = \begin{bmatrix} 3 & 0 \\ 0 & 0.5 \end{bmatrix}$

(d) $\boldsymbol{\mu} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & -0.75 \\ -0.75 & 1 \end{bmatrix}$

Figure 2: Four contour plots of a two-dimensional Gaussian distribution describing the effects of differing means and covariance functions. Mean affects location, the variance entries within the covariance matrix determine "stretch" $\sigma_i^2 > 1$ or "squash" $\sigma_i^2 < 1$, and the correlation entries σ_{ij} demonstrate a relationship between variables x_i and x_j.

A Gaussian process is a Gaussian multivariate distribution mapped onto a function space. The definition of a Gaussian process is 'a collection of random variables, any finite number of which have a joint Gaussian distribution,' (Williams and Rasmussen, 2006, pp. 13).

The specification of a Gaussian process is by its mean and covariance function:

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$$

The mean function specifies the expected value at each point where the function is evaluated, while the covariance function characterizes the relationship between values at different points of the function. (See Figure 3 for a graphical representation).

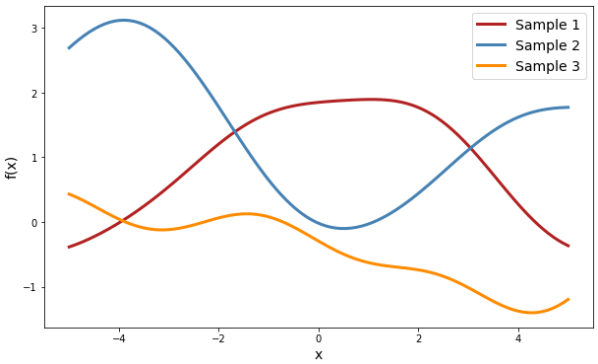


Figure 3: Plot of three samples of a Gaussian distribution on a function space, where the mean vector is zero and covariance function is the squared exponential (see section 3.2).

This particular specification is known as the function-based view. This view places an appropriate importance on the covariance function while ignoring the notion of parameters that would be detailed in the weight-spaced view. This paper will not share those details, yet can be found in Williams and Rasmussen (2006). Individuals who have experience in statistical inference with linear models or machine learning experience may find that the weight-spaced view bridges the gap between these areas and Gaussian processes.

2.4.2 Gaussian Processes Applied to Regression Problems

The choice of the mean and covariance function is dependent on the prior beliefs. The equivalent of the model parameters $\boldsymbol{\theta}$ from equation (1), Bayes' theorem, in a general sense is the mean $m(\mathbf{x})$ and covariance function $k(\mathbf{x}, \mathbf{x}')$ in the specific sense of Gaussian processes. Equivalent because both are decided before training or prediction is done, and that both are optimised to find the estimated function that mirrors most closely the true function. Generally, parameters $\boldsymbol{\theta}$ are optimised; however concerning Gaussian processes, the covariance function $k(\mathbf{x}, \mathbf{x}')$, which ever type is picked, have its own hyperparameters which are optimised. Covariance functions are discussed in length in Sections 3, 4 and 5.

This result is of significance. Gaussian processes are non-parametric meaning theoretically they have an infinite number of parameters $\boldsymbol{\theta}$. In contrast, parametric models have a finite number of parameters which specifies a particular probability distribution. In a modelling scenario, this requires strict assumptions to be made on the form of the data. A non-parametric approach allows more flexibility compared to parametric, since decisions on choice of the mean $m(\mathbf{x})$ and covariance function $k(\mathbf{x}, \mathbf{x}')$ are based on properties of the

function $f(\mathbf{x})$ rather than the exact form of the function $f(\mathbf{x})$. (Seymour et al., 2022). This explanation dispels the supposed equivalence between the Bayesian and Frequentist paradigms mentioned in Section 2.3.2: the choice of covariance function $k(\mathbf{x}, \mathbf{x}')$ is parallel to the choice of the parametric form of a function. This now apparent disparity is further evidenced by the power of varying hyperparameter values within *one* type of covariance function $k(\mathbf{x}, \mathbf{x}')$ seen in Chapter 3.

Within supervised learning for regression problems, the assumption of noisy observations is made. This is to account for measurement errors, other possible explanatory variables that are not captured within the dataset, and inherent randomness.

Therefore, rather than the output simply equal to the learnt function $y = f(x)$, an additional term epsilon is included that represents noise, $y = f(x) + \epsilon$. An extension of this assumption is $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

The joint distribution, Equation (3), below helpfully illustrates the setup before supervised learning is performed. The distribution shows the chosen prior beliefs: the mean function equalling zero, a fair and simple assumption, and the general covariance function being split into its parts, both training and test inputs and noise, σ_n^2. The notation of the covariance function has changed to reflect the multiple inputs by using capital letters to represent matrices.

A key tool heavily deployed within machine learning is separating the dataset into training and test samples. Once the input-output mappings have been learnt from the training examples, the generalisation capability of this function can be tested against unseen data points. For Section 2, a subscript asterisk proceeding a symbol implies a reference to test data points. The lack of implies training.

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X,X) + \sigma_n^2 I & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix}\right) \tag{3}$$

The posterior distribution $p(\theta|\mathcal{D})$ is now found from this joint distribution. Earlier, Section 2 described the posterior distribution as the probability of the parameter θ given the dataset $\mathcal{D}$. Translating these general terms into the specific case of Gaussian processes, the parameter θ is equivalent to the function $\mathbf{f}_*$, and the dataset is $\mathcal{D}$ equivalent to its components, $\mathbf{y}$, X and X_*.

The posterior distribution is otherwise known as the predictive distribution and has the following formula:

$$\mathbf{f}_* \mid X, \mathbf{y}, X_* \sim \mathcal{N}(\bar{\mathbf{f}}_*, \operatorname{cov}(\mathbf{f}_*)),$$

where

$$\bar{\mathbf{f}}_* = K(X_*, X)[K(X,X) + \sigma_n^2 I]^{-1}\mathbf{y} \tag{4}$$

$$\text{cov}(\mathbf{f}_*) = K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma_n^2 I]^{-1} K(X, X_*). \tag{5}$$

Proof:

The joint distribution must be flipped so that when the conditional distribution is taken, it results as the posterior distribution $p(\mathbf{f}_*|X, y_*, X_*)$.

$$\begin{bmatrix} \mathbf{f}_* \\ \mathbf{y} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X_*, X_*) & K(X_*, X) \\ K(X, X_*) & K(X, X) + \sigma_n^2 I \end{bmatrix}\right)$$

Therefore applying Equation (A.1), the conditional posterior distribution is found.

$$\begin{aligned} \mathbf{f}_* \mid \mathbf{X}, \mathbf{y}, \mathbf{X}_* \sim \mathcal{N}(K(\mathbf{X}_*, \mathbf{X})[K(\mathbf{X}, \mathbf{X}) + \sigma_n^2 \mathbf{I}]^{-1} \mathbf{y}, \\ K(\mathbf{X}_*, \mathbf{X}_*) - K(\mathbf{X}_*, \mathbf{X})[K(\mathbf{X}, \mathbf{X}) + \sigma_n^2 \mathbf{I}]^{-1} K(\mathbf{X}, \mathbf{X}_*)) \end{aligned}$$

□

2.4.3 Gaussian Process Methodology

In this methodology, a fictitious one-dimensional dataset is considered to focus solely on the mechanics of Gaussian process regression. Applying a real-world dataset at this early stage is futile and would not aid in the explanation of this methodology. Real-world datasets will be investigated once other topics within Gaussian processes have been explained.

Python code is demonstrated throughout, and the relevant packages to run the code are shown in Listing 1. For the avoidance of repetition, the reader is advised to use this code from the methodology as the basis of understanding future code and its application in subsequent topics. If the reader is in any doubt, consolidate Appendix B.

```
import matplotlib.pyplot as plt
import numpy as np
import GPy
```

Listing 1: Code of importing packages.

In this example, the true function $f(x)$ is known:

$$f(x) = 5\sin\left(\sqrt{x} - 5\right) + \exp(-10x) + x^{\frac{1}{10}} + 2 \tag{6}$$

The true function is algebraically represented in Equation (6), graphically illustrated in Figure 4a, and produced in python by Listing 2.

```
# Define true function f(x)
def f(x):
    return 5 * np.sin(np.sqrt(x) - 5)
             + np.exp(-10*x) + x**(1/10) + 2

# Create noise epsilon
def noise(mean, std_dev):
    return np.random.normal(mean, std_dev)
```

Listing 2: Code of the creation of the function f(x) and noise epsilon.

A dataset is drawn from this function $f(x)$ of corresponding x and y values. x values range from 0 to 100 and y values follow the function $y = f(x) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$ and $f(x)$ is from Equation 6. Therefore, the dataset consists of 101 data points (x_i, y_i) where $i = 0, 1, 2, \ldots, 100$. Listing 3 demonstrates this.

```
# Sample x and y values and reshape to ensure 2D array
x_values = np.arange(0, 100, 1).reshape(-1, 1)
y_values = np.array([f(x) +
                     noise(0, 1)
                     for x in x_values]).reshape(-1, 1)
```

Listing 3: Code of sampling x and y values.

The x and y values are standardised. The dataset is then split into a training set and a test set. The test set is made up of 20 samples, which are evenly distributed across the 101 datapoints. Figure 4b and Listing 4 shows this.

```
# Standardise x and y values (mean = 0, std = 1)
x_values = (x_values - x_values.mean()) / x_values.std()
y_values = (y_values - y_values.mean()) / y_values.std()

# Split the dataset into training (80) and test (20) sets
   equally spaced across the axis

# find 20 evenly distributed indices for the test set and let
   the rest be the training set
test_indices = np.linspace(0, len(x_values)-1, 20, dtype=int)
train_indices = np.setdiff1d(np.arange(len(x_values)),
                             test_indices)

# Create training samples of x and y values using the above
   indices
train_x = x_values[train_indices]
```

```
train_y = y_values[train_indices]

# Create test samples of x and y values using the above
   indices
test_x = x_values[test_indices]
test_y = y_values[test_indices]
```

Listing 4: Code of standardising the x and y values and splitting the values intro training and test samples.

The squared exponential covariance function is chosen (see Section 3.2) and the Gaussian process model is constructed using the training set and covariance function. The model is optimised (see Section 6.1.3). Now Equation (4) and (5) are used to find predictive mean and variance with test samples. Listing 5 describes this and the production of Figure 4d.

```
# Create kernel and fit the Gaussian Process model to training
    set
kernel = GPy.kern.RBF(input_dim=1, variance=1.0, lengthscale
   =1.0)
model = GPy.models.GPRegression(train_x, train_y, kernel)

# Optimise the model parameters
model.optimize()

# Make predictions on the test set
mean_prediction, variance = model.predict(test_x)
```

Listing 5: Code of the Gaussian Process regression, predictive mean and credible interval calculations.

The evaluation of variance is crucial in mapping credible intervals. Credible intervals in the Bayesian paradigm are equivalent to the Frequentist's confidence intervals. This is a key element in the debate between the two paradigms.

A confidence interval in the frequentist paradigm represents a range of values for a population parameter that, based on a sample, is likely to contain the true parameter value with a specified confidence level. For example, a 95% confidence interval means that if samples were repeatedly taken from the population and intervals were constructed, 95% of those intervals would contain the true parameter. The frequentist confidence interval is constructed without assuming any prior knowledge about the parameter and is derived purely from the sample data. In contrast, a Bayesian credible interval provides a direct probabilistic interpretation: it is the range within which the true parameter value lies with a certain probability, conditioned on the data and prior distribution. For instance, a 95% credible interval means that, given the observed data and prior distribution, there is a 95% probability that the true parameter lies within the interval. The latter description

seems more intuitive, at least proponents of the Bayesian paradigm argue.

Figure 4c shows the predictive mean and credible interval, calculated from the posterior distribution, and the success of these predictions can be assessed graphically when plotting the datapoints and the true function on top of these predictions in Figure 4d (Listing 6).

```
# Plot all components of the Guassian Process Methodology
plt.figure(figsize=(8, 6))

# Plot the true function
plt.plot(x_values, true_y_values, label='True Function',
         color='darkorange', linewidth=3)

# PLot the mean function
plt.plot(test_x, mean_prediction, label='GP Mean Prediction',
         color='steelblue', linewidth=3)

# Plot the confidence interval for the test set
plt.fill_between(test_x.flatten(), lower_bound.flatten(),
                 upper_bound.flatten(), color='powderblue',
                 alpha=0.3, label='Credible Interval (95%)')

# Plot the training and test data
plt.scatter(train_x, train_y, color='firebrick', marker='x',
            label='Training Data', s=50)
plt.scatter(test_x, test_y, color='mediumpurple', marker='x',
            label='Test Data', s=50)

# Specify plot parameters
plt.xlabel('x', fontsize=14)
plt.ylabel('f(x)', fontsize=14)
plt.legend(loc='lower right', ncol=2, fontsize=14)
plt.ylim(-4, 3)
plt.show()
```

Listing 6: Code of the Gaussian Process Methodology plot of Figure **??**.

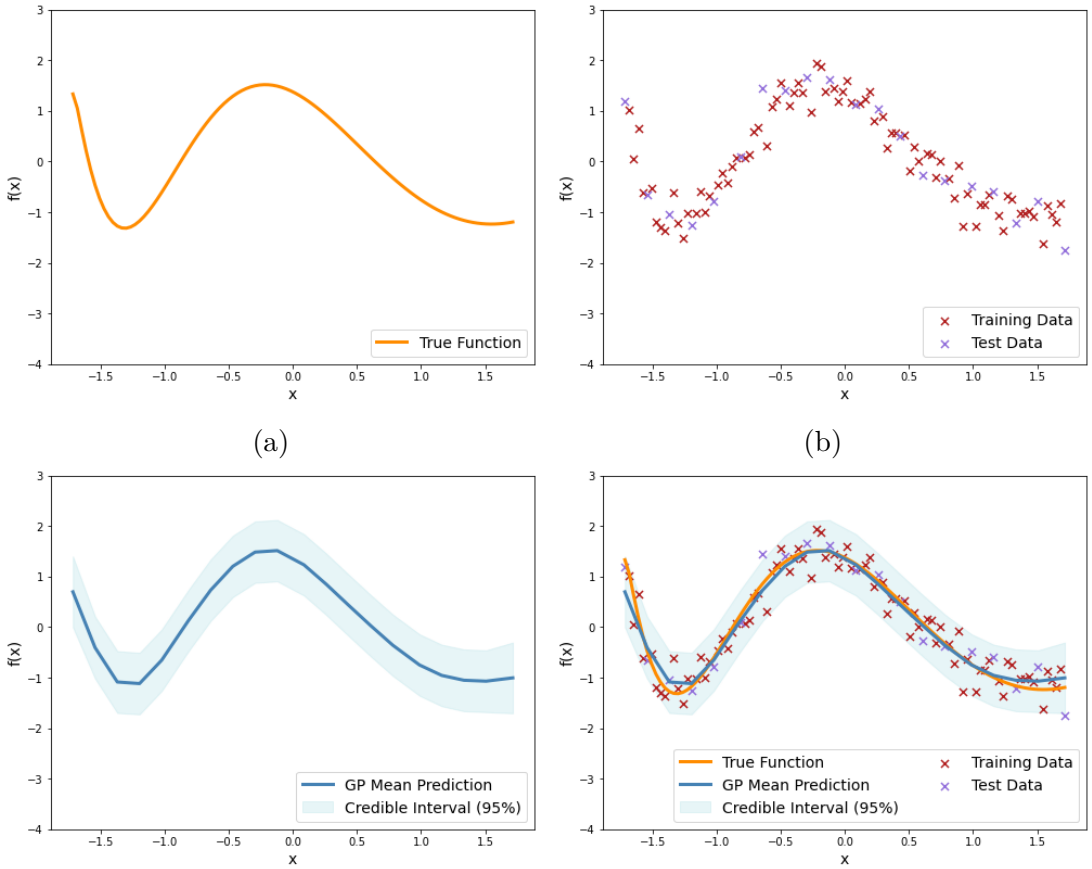


Figure 4: Four plots that show the progression of the Gaussian process methodology. From a graphical assessment, the predictive mean is relatively successful at mapping the true function from the data. Furthermore, the credible intervals encompass nearly all data points.

3 Covariance Functions

The covariance function is one of the most crucial subtopics within the field of Gaussian processes. The responsibility of the model's success of mapping the data lies on the choosing of the covariance function and its hyperparameters. Therefore, this work provides three chapters on the topic, in an extensive attempt to ascertain the paramountcy of covariance functions.

A rigorous mathematical definition of a covariance function is set out, and is used later on when discussing, in detail, the certain properties that different covariance functions possess (see Section 4). The method of exploring each covariance function includes a written and mathematical definition, their advantages for particular types of data (with illustrated examples), and the identification of properties. The input space $\mathcal{X}$ is treated as one-dimensional when describing common examples of covariance functions, simply for ease of understanding. Chapter 5 discusses multi-dimensional covariance functions.

3.1 The Definition of a Covariance Function

A covariance function is a positive semi-definite (PSD) kernel. A kernel is a term for a function k that takes a pair of inputs from an input space ($x, x' \in \mathcal{X}$) and maps them to $\mathbb{R}$. The kernel function can be used to determine the entries of a Gram matrix K. The entries of a Gram matrix are $K_{ij} = k(x_i, x_j)$. K is a covariance matrix if the kernel is a covariance function. Since a PSD kernel is a covariance function therefore, a covariance matrix is also PSD. This property is expressed in quadratic form: $Q(v) = v^T K v \geq 0 \quad \forall v \in \mathbb{R}^n$ and $K \in \mathbb{R}^{n \times n}$. Because of this link between covariance functions and kernels, these terms will be used interchangeably throughout the paper.

3.2 Squared Exponential

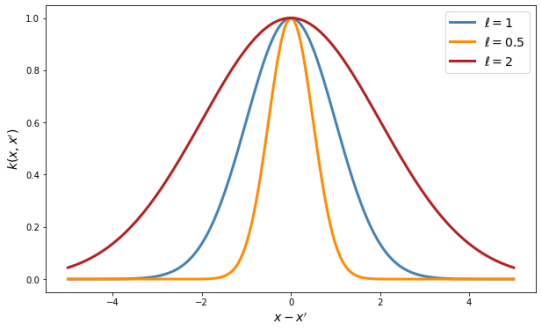
3.2.1 Definition & Properties

The formulation of the squared exponential (SE) is:

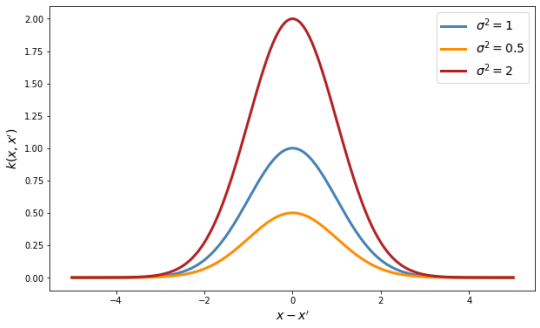
$$k(x, x') = \sigma_f^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right) \tag{7}$$

where σ_f^2 is the output variance (or scale factor) and ℓ is the characteristic length-scale. These are considered to be the hyperparameters of the covariance function.

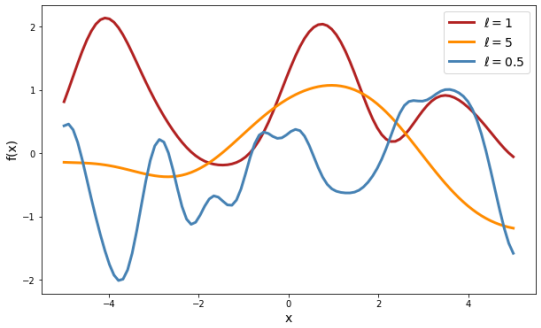
As the values within an explanatory variable grow further apart, the covariance moves



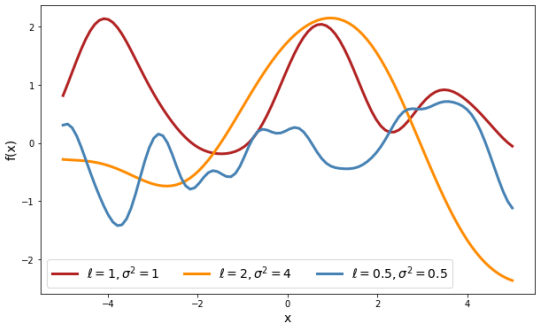
(a) Prior distribution of SE kernel with varying length-scale



(b) Prior distribution of SE kernel with varying scale factor.



(c) Posterior samples drawn from SE kernel with varying length-scale.



(d) Posterior samples drawn from SE kernel with varying scale factor.

Figure 5: Four plots all providing graphical understanding of the effects of varying SE kernel hyperparameters.

away from unity, (Williams and Rasmussen, 2006). This is known as a property called stationarity. In fact, the SE kernel is also isotropic. Furthermore, the SE kernel is known to produce processes that are very smooth. These properties are specified later in Section 4 .

The following description of how varying hyperparameters affect the covariance function $k(x, x')$ and the posterior samples mapped onto the function space aligns with the graphical representation in Figure 5.

The length-scale ℓ controls the characteristic width of the "wiggles" in the function, effectively determining how quickly the function can vary (Figure 5c). Typically, predictions become unreliable for regions more than ℓ units away from the observed data. The output variance σ_f^2 represents the average deviation of the function from its mean, acting as a scaling factor that influences the overall amplitude of the function's variations (Figure 5d). This parameter is a standard component in most kernels, providing flexibility in modelling the range of outputs. (Duvenaud, 2017).

Listing 7 is code for Figure 5c. The code for other plots within Figure 5 can be found in Appendix B.

```
# Define the Squared Exponential (SE) kernel function
def SE_kernel(x, x_prime, length_scale, variance):
    return variance * np.exp(-0.5 * np.sum((x1 - x2)**2) /
       length_scale**2)

# Generate some data points
x = np.linspace(-5, 5, 100)

# Set kernel parameters for each sample
kernel_params = [{"length_scale": 1.0, "variance": 1.0},
                 {"length_scale": 2.0, "variance": 1.0},
                 {"length_scale": 0.5, "variance": 1.0},]

# Plot the GP prior for each set of parameters
plt.figure(figsize=(10, 6))

colours = ['r', 'b', 'g', 'purple']
labels = ['lengthscale=1', 'lengthscale=5', 'lengthscale=0.5']

for i, params in enumerate(kernel_params):
    # Compute the covariance matrix using the SE kernel
    K = np.array([[SE_kernel(xi, xj, params['length_scale'],
                             params['variance'])
                   for xj in x] for xi in x])

    # Sample from the GP prior (mean vector = 0, covariance =
       K)
    np.random.seed(50 + i)  # Slightly different seed for
       reproducibility
    f_prior_samples = np.random.multivariate_normal(mean=np.
       zeros(len(x)), cov=K, size=1)

    # Plot the sample
    plt.plot(x, f_prior_samples[0], color=colours[i], label=
       labels[i])

# Set plot parameters
plt.xlabel("x", fontsize=14)
plt.ylabel("f(x)", fontsize=14)
plt.legend(loc="upper right", fontsize=14)
plt.grid(False)
plt.show()
```

Listing 7: Code of Figure **??**

3.2.2 Characteristic Length-scale Inspection

Rather than simply identifying the consequence of a varying length-scale through graphical representations, a mathematical approach is also considered.

A general 1-d process $f(t)$, drawn from some random distribution, is considered between the interval $[0, T]$. What is the expected number of upcrossings of a specifc level u within the interval? Upcrossings of a given level u mean when the process "crosses" the level u from below the level u.

A formula was found to express this question, known as Rice's formula, Equation (8). Rice (1945) and Kac (1943) found a highly specific formula where the general process $f(t)$ is Gaussian with zero mean and variance equal to one:

$$E\left[N_u^+(0,T)\right] = \frac{\lambda^{1/2}T}{2\pi}e^{-u^2/2} \tag{8}$$

This is considered to be the classic Rice formula. After this discovery, the formula had seen substantial improvements, with proofs being established under progressively more relaxed conditions. Adler (2010) states a version of the formula, Equation (9), where the process is Gaussian, stationary, zero-mean, and almost surely continuous.

$$E[N_u] = \frac{1}{2\pi}\sqrt{\frac{-k''(0)}{k(0)}}\exp\left(-\frac{u^2}{2k(0)}\right) \tag{9}$$

Since a process drawn from an SE kernel fulfils the criterion for the above formula, the specific equation for the expected number of upcrossings can be found.

Let the SE covariance function, Equation (7), instead be written as one variable τ that represents the difference between inputs $\tau = x - x'$ and treat the scale factor σ_f^2 equal to one.

$$k(\tau) = \exp\left(-\frac{(\tau)^2}{2\ell^2}\right) \tag{10}$$

Using Equation (10), the second derivative of the covariance function w.r.t τ needs to be evaluated.

$$k'(\tau) = -\frac{\tau}{\ell^2}\exp\left(-\frac{\tau^2}{2\ell^2}\right), \quad k''(\tau) = \left(\frac{\tau^2}{\ell^4} - \frac{1}{\ell^2}\right)\exp\left(-\frac{\tau^2}{2\ell^2}\right)$$

Substituting $\tau = 0$ into the second derivative $k''(\tau)$ and the original kernel $k(\tau)$, the following results are:

$$k(0) = \exp(0) = 1, \quad k''(0) = -\frac{1}{\ell^2}$$

Therefore, the expectation of the number of upcrossings of this process derived from an SE kernel is:

$$\begin{aligned} E[N_u] &= \frac{1}{2\pi}\sqrt{\frac{-k''(0)}{k(0)}}\exp\left(-\frac{u^2}{2k(0)}\right) \\ &= \frac{1}{2\pi}\sqrt{\frac{1}{\ell^2}}\exp\left(-\frac{u^2}{2}\right) \\ &= \frac{1}{2\pi\ell}\exp\left(-\frac{u^2}{2}\right) \\ &= \frac{1}{2\pi\ell} \quad \text{let } u = 0 \Rightarrow \text{zero-crossings} \end{aligned}$$

This equation confirms the graphical presentations in Figure 5 of how the characteristic length-scale changes the frequency of variations; as $\ell \to \infty$, then $E[N_u] \to 0$.

3.2.3 Example: Microeconomic Collusion

The dataset analysed with the SE kernel is called *CartelStability*. This was originally used to detect whether members of a cartel known as the Joint Executive Committee (JEC) were colluding. JEC controlled eastbound freight shipments from Chicago to the Atlantic seaboard during the 1880s. Collusion is an economics term that describes a few incumbent firms within a market agreeing to control the price or quantity of their goods, preventing possible competitive entrants for one reason: maximising profits. See Porter (1983) for more details.

In order to access the datasets, an additional package is needed. Listing 8 shows how to access *CartelStability* in python and Appendix C directs the reader to downloading the CSV file. 'FILE PATH' denotes the file path in which to find the downloaded CSV file.

```
import pandas as pd

# Access the dataset
file_path = r'FILE PATH\CartelStability.csv'
data = pd.read_csv(file_path)
```

Listing 8: Code of importing the package 'pandas' and accessing the data.

Figure 6 shows a time-series (weekly) of one of the variables in the dataset, quantity (total tonnage of grain shipped). *CartelStability* is split into training and test samples using a proportion (Listing 9) rather than a fixed number seen in Section 2.4.3.

Figure 6 indicates a stationary trend with some local variations. Whilst the dataset may not suggest smoothness (MS infinitely differentiable), the SE kernel is chosen for its other agreeable properties seen in the dataset and to test its easy applicable nature. An important note is that simple kernels require simple dataset structures for the Gaussian process to accurately map the data, yet simple datasets are a rarity. Section 5 discusses cases of more complex kernels that are able to map more complex datasets that are more common.

```
# 20% of the data for the test set
test_size = int(0.2 * len(data))

# Select indices evenly distributed for the test set
test_indices = np.linspace(0, len(data) - 1, test_size,
                           dtype=int)
# The rest of the data will be the training set
train_indices = np.setdiff1d(np.arange(len(data)),
                             test_indices)
```

Listing 9: Code of producing test and training indices from a proportion of the data.

For more interpretable hyperparameters, the quantity variable is standardised (subtracting the mean and dividing by the standard deviation).

Figure 7 shows the predictive distribution, its mean and credible intervals, drawn from the SE covariance function. The characteristics of the dataset suit the SE kernel: local correlations favouring the property of stationarity and the trade-off of the variations suiting both smoothness and appropriate length-scale.

An important note is that when the GP regression model is run with an SE kernel in the GPy package on python, the method includes white noise as a kernel $\sigma_n^2 I$. This term is seen from the posterior distribution in Equations (4) and (5). Therefore, white noise variance σ_n^2 is treated as an additional hyperparameter. That is why it has been optimised in Figure 7. This apparent sum of two kernels does not disrupt the Gaussian

process learning the data, for the sum of two kernels is a kernel (see Section 5.2 for more details). This inclusion of the white noise kernel is true for all GP regression models produced in this dissertation.

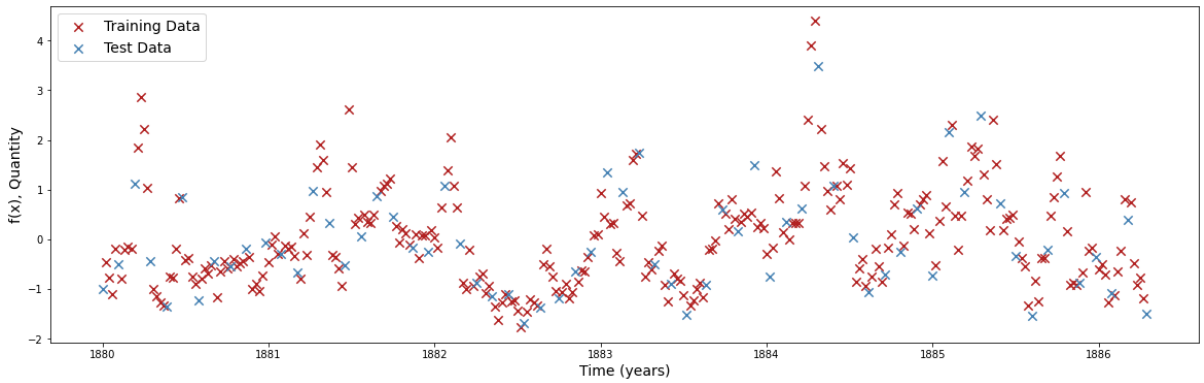


Figure 6: Scatter time-series plot of the quantity of the weekly total tonnage of grain shipped. The dataset is split into training and test samples with ratio 80/20 and with the test samples evenly distributed across the input space.

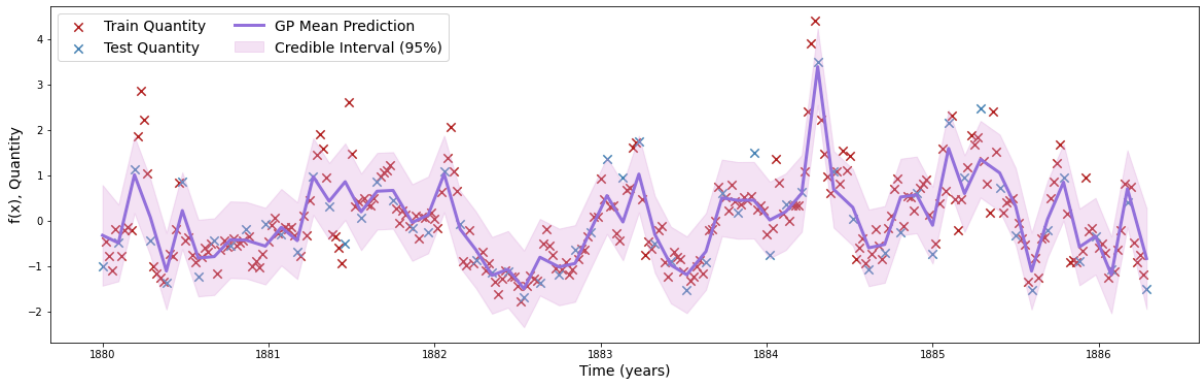


Figure 7: An optimised Gaussian process regression drawn from an SE kernel with predictive mean and credible intervals. The optimised hyperparameter values for the Squared Exponential (SE) kernel are: the variance $\sigma_f^2 = 0.808$, the length-scale $\ell = 0.0463$, and the Gaussian noise variance $\sigma_n^2 = 0.121$.

3.3 Matérn

3.3.1 Definition & Properties

This is a class of covariance functions given by

$$k(x, x') = \sigma_f^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}|x - x'|}{\ell} \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu}|x - x'|}{\ell} \right) \tag{11}$$

where ν is the smoothness parameter, ℓ is the length-scale, σ_f^2 is the output variance, $\Gamma(\nu)$ is the Gamma function, and K_ν. (Abramowitz and Stegun, 1964, see sec. 9.6)

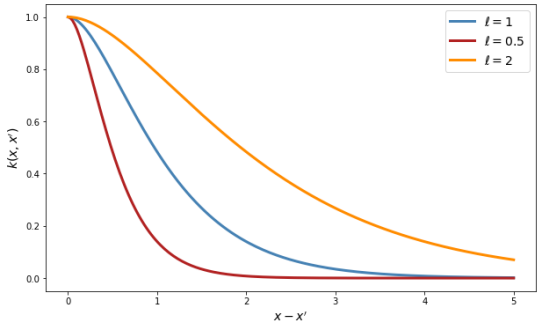
This covariance function is considered a generalisation of SE kernel since as $\nu \to \infty$, the Matérn transforms into the SE kernel.

If ν is a half-integer, Equation (11) simplifies. According to Williams and Rasmussen (2006), the most intriguing cases are $\nu = 3/2$ and $\nu = 5/2$. Equation (11) becomes, respectively:

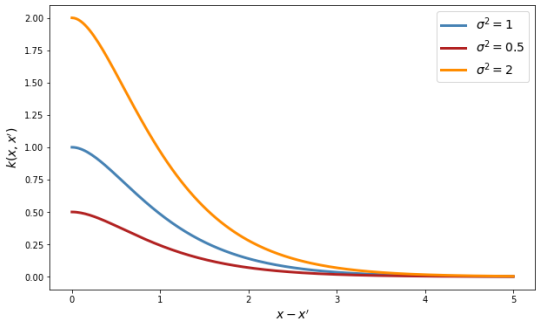
$$k(x, x') = \sigma_f^2 \left(1 + \frac{\sqrt{3}|x - x'|}{\ell}\right) \exp\left(-\frac{\sqrt{3}|x - x'|}{\ell}\right)$$

$$k(x, x') = \sigma_f^2 \left(1 + \frac{\sqrt{5}|x - x'|}{\ell} + \frac{5|x - x'|^2}{3\ell^2}\right) \exp\left(-\frac{\sqrt{5}|x - x'|}{\ell}\right)$$

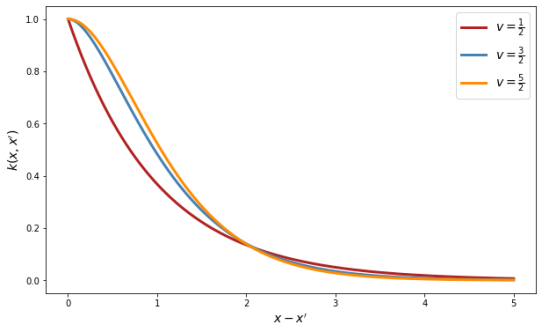
The above equations are derived from a general expression found in Abramowitz and Stegun (1964, eq. 10.2.15).



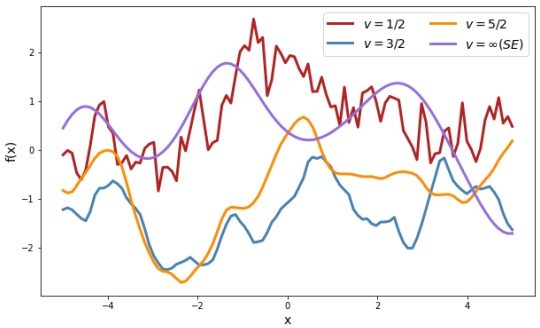
(a) Prior distribution of Matérn kernel with varying length-scale.



(b) Prior distribution of Matérn kernel with varying scale factor.



(c) Prior distribution of Matérn kernel with varying class.



(d) Posterior samples drawn from Matérn kernel with varying class.

Figure 8: Four plots all providing graphical understanding of the effects of varying Matérn kernel hyperparameters.

The case where $\nu = 1/2$ is a particular covariance function which produces "rough" functions. "Rough" implies mean-squared (MS) continuous but not mean-squared (MS) differentiable. Section 4.3 discusses the property of smoothness under varying perspectives, including MS differentiability. This covariance function is known as the exponential covariance function $k(x, x') = \sigma_f^2 \exp -\frac{x-x'}{\ell}$. The process drawn from this kernel is known as the Ornstein-Uhlenbeck (OU) process (Uhlenbeck and Ornstein, 1930).

The case where $\nu \geq 7/2$ is almost redundant (a case could be made if there is prior knowledge of the existence of higher order derivatives), for finite noisy training examples are unable to distinguish between the chosen ν and $\nu \to \infty$, which is simply the SE covariance function.

This parameter ν determines the degree of differentiability of the output function, $f(x)$, specifically k-times MS differentiable if and only if $\nu \geq k$ (Santner et al., 2003, see Section 2.3.4, Example 2.5). This controls the "smoothness" of a function and is graphically represented in Figures 8c and 8d. The length-scale parameter, ℓ, and signal variance, σ_f^2, react in the same way to the squared exponential covariance function and this is shown in Figures 8a and 8b.

Like the SE kernel, The Matérn kernels are considered stationary and isotropic. The low degree of smoothness for $\nu \leq 5/2$ is beneficial for datasets that possess 'roughness'. An appropriate example is stock prices.

3.3.2 Example: Stock Prices

The following dataset is *GAFA stock prices* where GAFA refers to Google, Amazon, Facebook, and Apple. The dataset includes historical daily stock prices from 2014 to 2018. The variable in question which is plotted on a time-series in Figure 9 is the 'Adj_Close'; the variable contains the adjusted closing price for the stock. additionally, the Apple stock prices are only considered. Figure 9 clearly shows a time-series plot with jumps, implying "roughness". This is why a Matérn kernel is suitable in mapping this dataset.

```
GPy.kern.Matern32(input_dim=1, variance=1.0, lengthscale=1.0)
```

Listing 10: Code of creating the Matérn kernel ($\nu = 3/2$)

Figure 10 shows an optimised Gaussian process with a Matérn kernel ($\nu = 3/2$) (Listing **??**) with a predictive mean and credible intervals found from the test set. Similar to the example with the SE kernel, The price variable has been standardised for ease of interpretability.

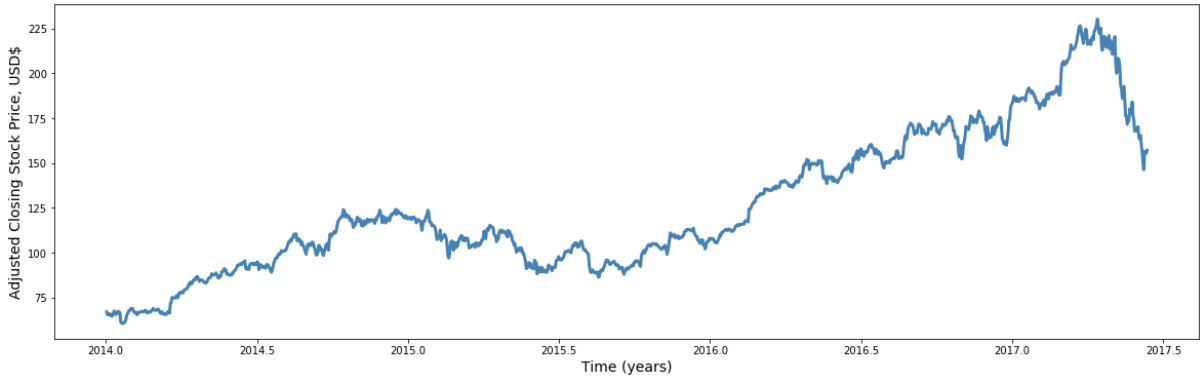


Figure 9: Scatter time-series plot of the adjusted closing price of Apple stock daily. The dataset is split into training and test samples with ratio 80/20 and with the test samples evenly distributed across the input space.

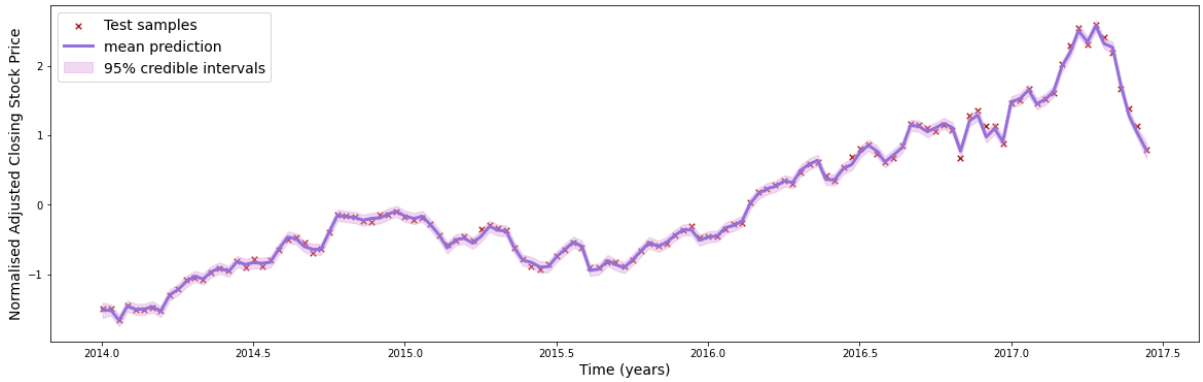


Figure 10: An optimised Gaussian process regression drawn from an Matérn kernel with predictive mean and credible intervals. The optimised hyperparameter values for the Matérn kernel ($\nu = \frac{3}{2}$) are: the variance $\sigma_f^2 = 0.631$, the lengthscale $\ell = 0.0813$, and the Gaussian noise variance $\sigma_n^2 = 0.00119$.

3.4 Periodic

3.4.1 Definition & Properties

The periodic covariance function is denoted as:

$$k(x, x') = \sigma_f^2 \exp\left(-\frac{2\sin^2\left(\frac{\pi|x-x'|}{p}\right)}{\ell^2}\right) \tag{12}$$

where σ_f^2 is the output variance, p is the period, and ℓ is the length-scale parameter.

'The period p simply determines the distance between repartitions of the function' Duven-

aud (2017) and output variance σ_f^2 and length-scale parameter ℓ act identically to the previously mentioned covariance functions. The periodic kernel also possesses identical properties with the SE kernel such as stationarity, isotropy and smoothness. In fact, the periodic kernel is infinitely differentiable and therefore produces very smooth processes.

Furthermore, the periodic kernel could be considered a variant of the SE kernel. MacKay et al. (1998) found that an SE kernel in the **u**-space, where $\mathbf{u}(x)$ =(cos(x),sin(x)), produces the periodic kernel.

3.4.2 Mathematical and Graphical Analysis of Hyperparameter Variation

The expected number of upcrossings, Equation (9), can be discovered specifically for the periodic kernel as it satisfies the necessary conditions.

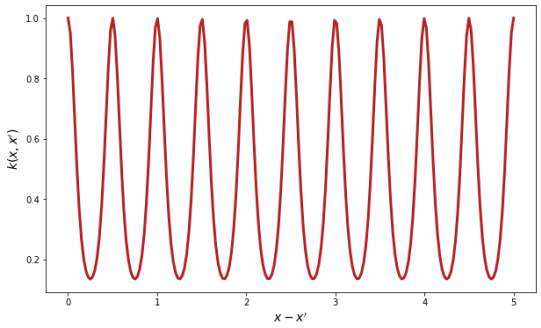
Firstly, the second derivative of the periodic kernel is found, specifically equated when the difference in inputs is zero $r = |x - x'| = 0$. The periodic kernel itself is equated at zero too. The second derivative of the periodic kernel function is cumbersome and requires extensive use of chain and product rules.

$$k(0) = \exp(0) = 1, \quad k''(0) = -\frac{4\pi^2\ell^2 k(0)cos(0)}{p^2\ell^4} = -\frac{4\pi^2}{p^2\ell^2} \tag{13}$$

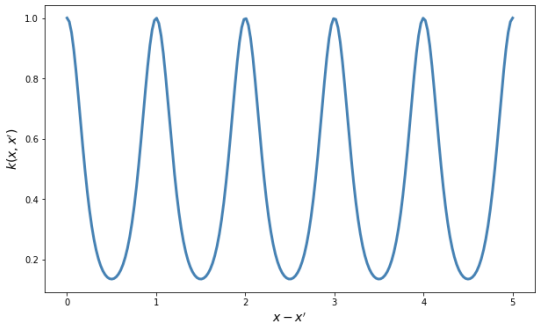
The above results can be plugged into Equation (9):

$$\begin{aligned}
E[N_u] &= \frac{1}{2\pi}\sqrt{\frac{-k''(0)}{k(0)}}\exp\left(-\frac{u^2}{2k(0)}\right) \\
&= \frac{1}{2\pi}\sqrt{\frac{4\pi^2}{p^2\ell^2}}\exp\left(-\frac{u^2}{2}\right) \quad \text{using Equation (13)} \\
&= \frac{1}{2\pi}\frac{2\pi}{p\ell}\exp\left(-\frac{u^2}{2}\right) \\
&= \frac{1}{p\ell} \quad \text{let } u = 0 \Rightarrow \text{zero-crossings}
\end{aligned}$$

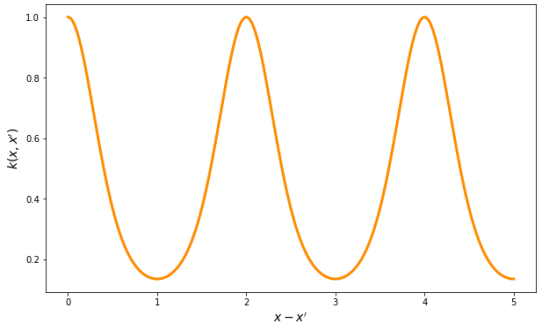
The result above shows the characteristic length-scale in the periodic covariance function acting identically to the previous kernels. Another observation made from the above result is the effect of period p on the expected number of upcossings. The period p acts in the same way as the length-scale and this is verified by the graphical representations in Figure 11.



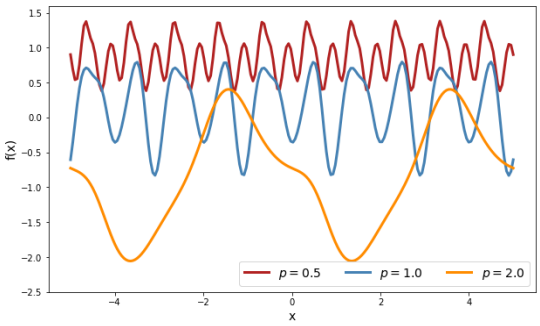
(a) Prior distribution of Periodic kernel with period $p = 0.5$.



(b) Prior distribution of Periodic kernel with period $p = 1$.



(c) Prior distribution of Periodic kernel with period $p = 2$.



(d) Posterior samples drawn from periodic kernel with varying period.

Figure 11: Four plots all providing graphical understanding of the effects of varying the period p hyperparameter on the periodic kernel.

3.4.3 Example: Temperature at Nottingham Castle

Because of the repetitive nature of the periodic kernel, datasets such as weather time-series are well suited to Gaussian processes with periodic kernels. The dataset analysed is known as *nottem*. It measures the average monthly air temperatures at Nottingham Castle in degrees Fahrenheit for 20 years from 1920 to 1939. It was originally used by Anderson (1976) who was applying the Box-Jenkins approach to time-series analysis and forecasting.

Figure 12 is the time-series plot and indicates a repetitive trend which is mapped by a Gaussian process with periodic kernel (Listing 11) in Figure 13. Similarly, the dataset is split into training and test samples at a ratio of 80/20 and are evenly distributed across the input space.

```
kernel = GPy.kern.StdPeriodic(input_dim=1, variance=1.0,
                              lengthscale=1.0, period=1)
```

Listing 11: Code of creating the periodic kernel.

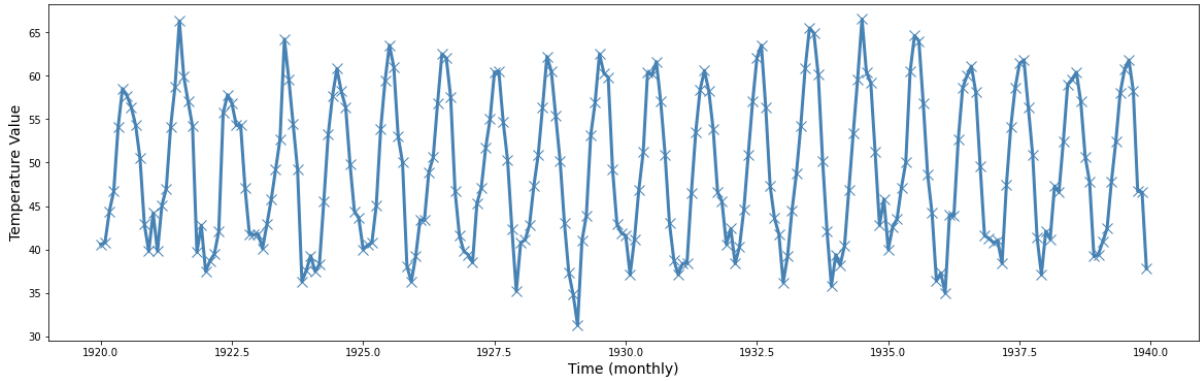


Figure 12: Time-series plot of the average monthly air temperatures at Nottingham Castle in degrees Fahrenheit.

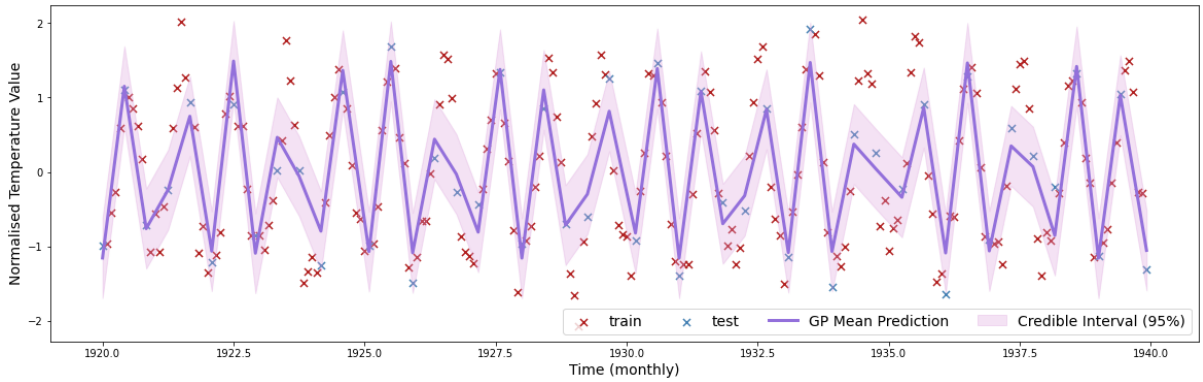


Figure 13: An optimised Gaussian process regression drawn from a periodic kernel with predictive mean and credible intervals. The optimised hyperparameter values for the Periodic kernel are: the variance $\sigma_f^2 = 3.498$, the period $p = 1.001$ the lengthscale $\ell = 1.203$, and the Gaussian noise variance $\sigma_n^2 = 0.0730$.

3.5 Rational Quadratic

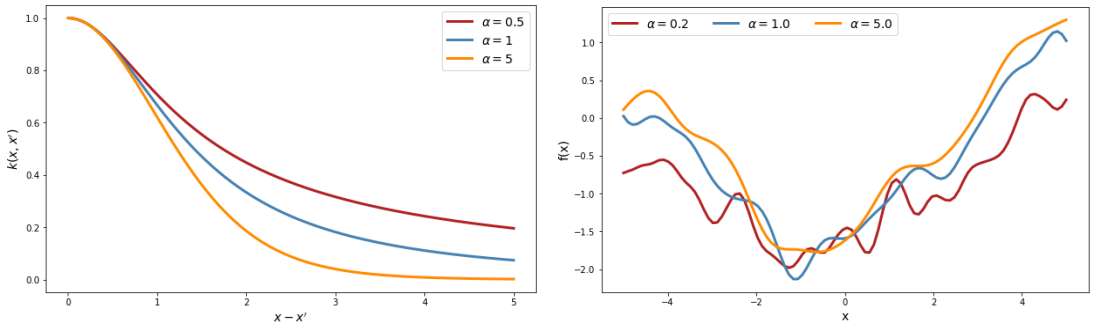
3.5.1 Definition

The rational quadratic (RQ) covariance function has the following formula:

$$k(x, x') = \sigma_f^2 \left(1 + \frac{(x - x')^2}{2\alpha\ell^2}\right)^{-\alpha} \tag{14}$$

where σ_f^2 is the output variance, ℓ is the length-scale, and α is the shape parameter controlling the variation in smoothness.

3.5.2 Properties & Analysis of Hyperparameter Variations



(a) Prior distribution of RQ kernel with varying shape hyperparameter α.

(b) Posterior samples drawn from RQ kernel with varying shape hyperparameter α.

Figure 14: Two plots providing graphical understanding of the effects of varying the shape α hyperparameter on the RQ kernel.

The RQ kernel can be viewed as SE kernels with different length-scales added together (Duvenaud, 2017) and therefore a precursor to later sections on kernel addition (5.2).

The following mathematical understanding comes from Williams and Rasmussen (2006). This is known as a scale mixture (infinite sum) and is expressed through an integral of the product of a length-scale distribution and an SE kernel with respect to length-scale. Stein (1999, see sec. 2.10) provides a general representation of this integral for which an isotropic kernel defines a valid covariance function.

Proof:

The SE kernel (where scale factor $\sigma_f^2 = 1$) uses two transformations $r = x - x'$ and $\tau = \ell^{-2}$ becoming:

$$k_{\text{SE}}(r|\tau) = \exp\left(-\frac{r^2\tau}{2}\right) \tag{15}$$

The specific case for the RQ kernel uses a gamma distribution for the length-scale.

$$
\begin{aligned}
k_{\mathrm{RQ}}(r) &= \int p(\tau|\alpha,\beta) k_{\mathrm{SE}}(r|\tau) d\tau \\
&= \int \frac{\beta^\alpha}{\Gamma(\alpha)} \tau^{\alpha-1} \exp(-\frac{\alpha\tau}{\beta}) \exp\left(-\frac{r^2\tau}{2}\right) d\tau \quad \text{using Equation (A.2)} \\
&\propto \int \tau^{\alpha-1} \exp\left(-\frac{\alpha\tau}{\beta}\right) \exp\left(-\frac{r^2\tau}{2}\right) d\tau \\
&= \int \tau^{\alpha-1} \exp\left(-\tau(\frac{\alpha}{\beta} + \frac{r^2}{2})\right) d\tau \\
&= \int \tau^{\alpha_*-1} \exp\left(-\tau\beta_*\right) d\tau \quad \text{where } \beta_* = \frac{\alpha}{\beta} + \frac{r^2}{2} \text{ and } \alpha_* = \alpha \\
&= \frac{\Gamma(\alpha_*)}{\beta_*^{\alpha_*}} \int \frac{\beta_*^{\alpha_*}}{\Gamma(\alpha_*)} \tau^{\alpha_*-1} \exp\left(-\tau\beta_*\right) d\tau \\
&= \frac{\Gamma(\alpha_*)}{\beta_*^{\alpha_*}} \quad \text{using Equation (A.3)} \\
&= \Gamma(\alpha) \left(\frac{\alpha}{\beta} + \frac{r^2}{2}\right)^{-\alpha} \\
&\propto \left(1 + \frac{\beta r^2}{2\alpha}\right)^{-\alpha} \\
&= \left(1 + \frac{r^2}{2\alpha\ell^2}\right)^{-\alpha}
\end{aligned}
$$

□

Therefore, the shape parameter controls the balance between large-scale and small-scale variations (Figure 14). A small shape parameter α allows for large variations in smoothness within the function and as the shape parameter α tends to infinity, $\alpha \to \infty$, the rational quadratic becomes the squared exponential.

Proof:

$$
\begin{aligned}
\lim_{\alpha\to\infty} k_{\mathrm{RQ}}(r) &= \lim_{\alpha\to\infty} \left(1 + \frac{r^2}{2\alpha\ell^2}\right)^{-\alpha} \\
&= \exp(-\alpha\frac{r^2}{2\alpha\ell^2}) \quad \text{Equation (A.4)} \\
&= \exp(-\frac{r^2}{2\ell^2}) \\
&= k_{\mathrm{SE}}(r)
\end{aligned}
$$

□

The RQ kernel is distinct from Matérn, because the Matérn's smoothness parameter, ν, determines the smoothness for the entire function.

3.5.3 Example: Macroeconomic Time-series

The dataset analysed in this section is called *USMacroB* and is a time-series (quarterly) data on 3 US macroeconomic variables, gross national product, average of the seasonally adjusted monetary base, and average of 3 month treasury-bill rate (p.a.) from 1959-1995. Baltagi (2008) used this dataset as application within his book on Econometrics.

The focus is on one variable, the average of 3 month treasury-bill rate (p.a.). A 3 month treasury-bill rate represents the return earned from investing in a government-issued treasury security with a maturity of 3 months. Figure 15 shows the scatter plot of this variable and a stationary trend with varying variations indicates the suitability of the rational quadratic kernel. Figure 16 shows the Gaussian process regression mapping the data drawn from a rational quadratic kernel (Listing 12).

```
kernel = GPy.kern.RatQuad(input_dim=1, variance=1.0,
                          lengthscale=1, power=1)
```

Listing 12: Code of creating the rational quadratic kernel.

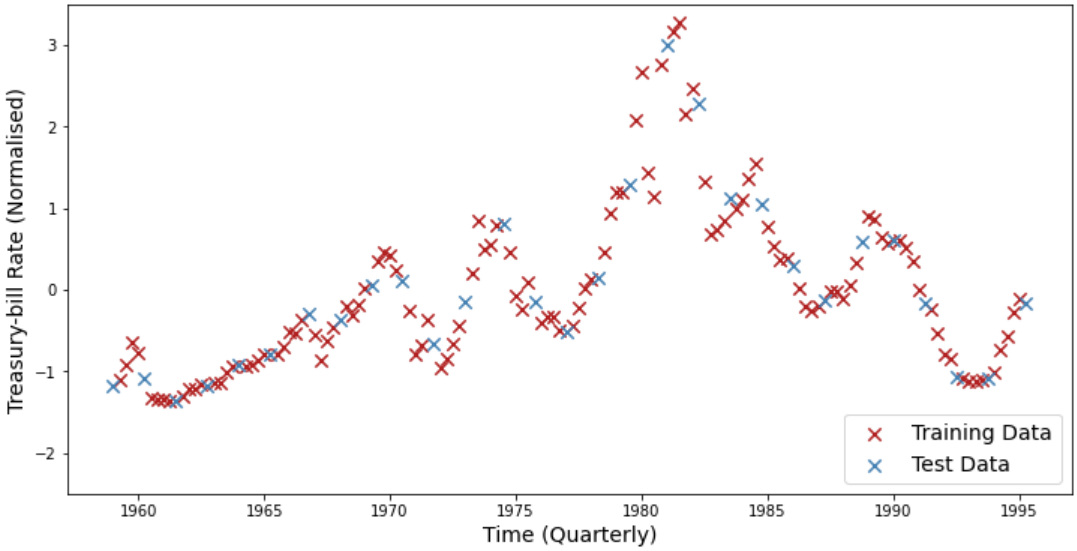


Figure 15: Time-series plot of the average of 3 month treasury-bill rates (p.a.).

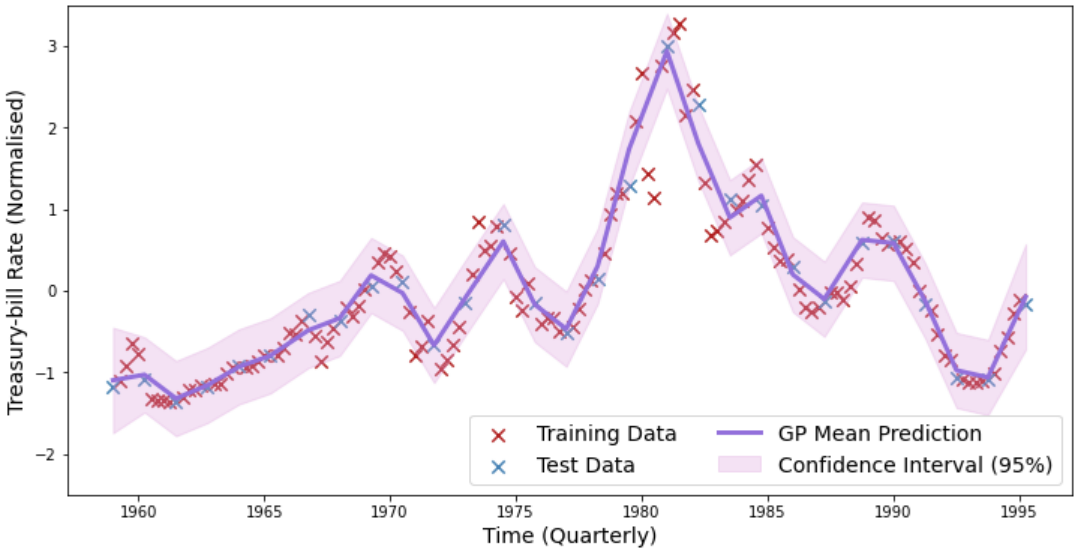


Figure 16: An optimised Gaussian process regression drawn from RQ kernel with predictive mean and credible intervals. The optimised hyperparameter values for the Rational Quadratic kernel are: the variance $\sigma_f^2 = 1.307$, the shape $\alpha = 0.138$ the lengthscale $\ell = 0.450$, and the Gaussian noise variance $\sigma_n^2 = 0.0310$.

3.6 Linear

3.6.1 Definition & Properties

The linear covariance function is given by:

$$k(x, x') = \sigma_b^2 + \sigma_f^2 x x', \tag{16}$$

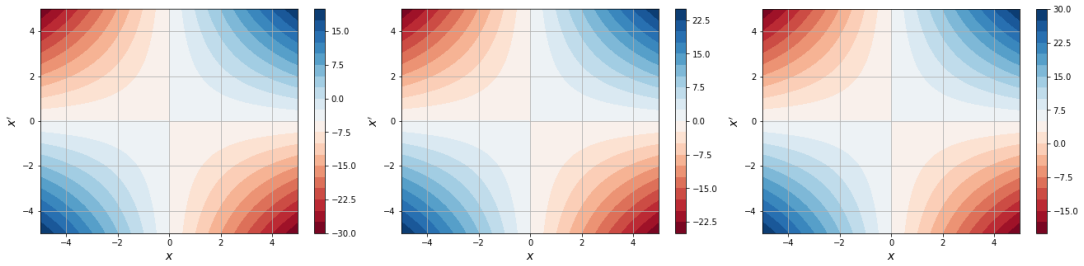
where σ_b^2 is the bias variance and σ_f^2 is the scale factor.

The bias (constant) variance, σ_b^2, influences how far the function's value is likely to deviate from 0 at zero input. Figure 17 illustrates the bias variance hyperparameter on covariance function values. Although it does not directly specify this value, it instead defines a prior distribution over it, adding flexibility to the model. Listing 13 describes the creation of Figure 17.

```
# Define the Linear kernel function
def linear_kernel(x, x_prime, b, sigma_sq=1.0,):
    return b + sigma_sq * (x) * (x_prime)

# Create arrays of x and x' values (from -5 to 5)
x_values = np.linspace(-5, 5, 100)
x_prime_values = np.linspace(-5, 5, 100)
```

```
# Plot Linear Kernel with Varying Bias
bias_values = [-5, 0, 5]
for b in bias_values:
    # Compute the kernel matrix for varying bias
    K = np.array([[linear_kernel(xi, xj, sigma_sq=1.0, b=b)
                   for xj in x_prime_values]
                  for xi in x_values])

    # Create contour plot
    plt.figure(figsize=(7, 5))
    cp = plt.contourf(x_values, x_prime_values, K, levels=20,
       cmap='coolwarm')
    plt.colorbar(cp)
    plt.xlabel("x", fontsize=14)
    plt.ylabel("x'", fontsize=14)
    plt.grid(False)
    plt.show()
```

Listing 13: Code of creating the the varying bias contour plots.



(a) bias $\sigma_b^2 = -5$ (b) bias $\sigma_b^2 = 0$ (c) bias $\sigma_b^2 = 5$

Figure 17: Three contour plots that show the effect of the bias hyperparameter on the covariance function value.

A key distinction of the linear kernel to the previously mentioned kernels is that it possesses the property of non-stationarity. This is evidenced by the covariance formula taking into account the absolute values of x and x' rather than their relative position (i.e. distance), $|x - x'|$.

3.6.2 Example: Relationship between GDP and Energy Consumption

The dataset analysed is called *EuroEnergy*, which contains cross-sectional data on energy consumption for 20 European countries, recorded for the year 1980. The dataset includes two key variables: the real gross domestic product (GDP) for each country in 1980 (measured in million 1975 US dollars) and the total energy consumption (measured in million kilograms of coal equivalence). This data is sourced from Baltagi (2008) and was used for application within his book on Econometrics. The *EuroEnergy* dataset provides

valuable insight into the relationship between economic activity and energy use across Europe.

Before applying Gaussian process regression with a linear kernel (Listing 14), the dataset, as usual, is adjusted. Figure 18 shows the Gaussian process regression with predictive mean and credible intervals. Important note is that a bias hyperparameter is not included in the linear kernel. Figure 18, specifically the datapoints, suggest a global linear non-stationary trend. That is why the linear kernel was applied to this dataset.

```
kernel = GPy.kern.Linear(input_dim=1, variances=1)
```

Listing 14: Code of creating the Linear kernel.

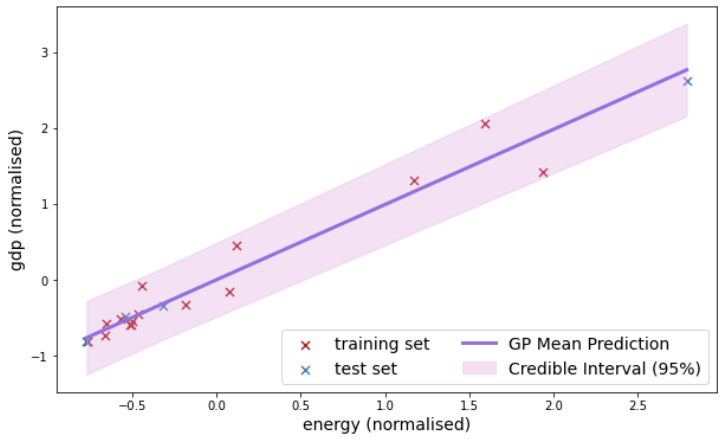


Figure 18: An optimised Gaussian process regression drawn from Linear kernel with predictive mean and credible intervals. The optimised hyperparameter values for the Linear kernel are: the variance $\sigma_f^2 = 0.991$, and the Gaussian noise variance $\sigma_n^2 = 0.0580$.

4 Properties of Covariance Functions

Knowledge of the properties of kernels enhances the ability to choose an appropriate kernel, which is considered the prior distribution in the Bayesian paradigm. If the prior distribution reflects the belief of the true characteristics of a dataset then there is a possibility of translating these characteristics into equivalent properties. A belief of the presence of a given property within a dataset expressed in an appropriate kernel compared to an ambiguous belief might yield stronger results in mapping the data. Therefore, this chapter explores the properties of kernels previously briefly mentioned in Chapter 3.

4.1 Stationarity

The following is from Williams and Rasmussen (2006).

A kernel function $k(x, x')$ that possesses stationarity can be written as $k(r)$ where $r = x - x'$. Therefore, the function is concerned of the relative position of the different input values x in the input space. In other words, the function is 'invariant to translations in the input space', (Williams and Rasmussen, 2006, pp. 79). Papoulis and Pillai (2002) state the difference between weakly and strictly stationary. In stochastic process theory, a process is weakly stationary if it has a constant mean and covariance function that is invariant to translations, while it is strictly stationary if all of its finite-dimensional distributions remain invariant to translations.

An important theorem explains how a covariance function of a stationary process can be expressed as the Fourier transform of a positive finite measure. Bochner's theorem, as stated by Stein (1999) and with a proof by Gihman and Skorokhod (2004), asserts this result. If the measure μ has a density $S(s)$, then $S(s)$ is referred to as the spectral density or power spectrum associated with k.

Bochner's Theorem: A complex-valued function k defined on $\mathbb{R}^D$ is the covariance function of a weakly stationary mean square continuous complex-valued random process on $\mathbb{R}^D$ if and only if it can be represented as:

$$k(r) = \int_{\mathbb{R}^D} e^{2\pi i s \cdot r} \, d\mu(s) \tag{17}$$

where μ is a positive finite measure.

When the spectral density $S(s)$ exists, the covariance function and the spectral density are related through a Fourier transform pair, as shown in Equation (18). This relationship is known as the Wiener-Khinchin theorem, and further details can be found in works by Chatfield and Xing (2019). The covariance function and the spectral density are Fourier

duals, meaning that the covariance function can be recovered from the spectral density and vice versa.

$$k(r) = \int_{-\infty}^{\infty} S(s)e^{2\pi i s \cdot r}\, ds, \quad S(s) = \int_{-\infty}^{\infty} k(r)e^{-2\pi i s \cdot r}\, dr. \tag{18}$$

This result is used later on when discussing the property of smoothness.

4.2 Isotropic

A kernel function $k(x, x')$ that possesses isotropy can be written as $k(r)$ where $r = |x - x'|$. Therefore, the function is concerned of the relative position, ignoring direction, of the different input values x in the input space. In other words, the function is 'invariant to all rigid motions', (Williams and Rasmussen, 2006, pp. 80).

Isotropic covariance functions lead to a simplification of the foruier duals mentioned in Section 4.1 (Eqation (18)). Bracewell (1966) finds that the new fourier duals are:

$$k(r) = \frac{2\pi}{r^{p/2-1}} \int_0^{\infty} S(s) J_{p/2-1}(2\pi r s) s^{p/2}\, ds \tag{19}$$

$$S(s) = \frac{2\pi}{s^{p/2-1}} \int_0^{\infty} k(r) J_{p/2-1}(2\pi r s) r^{p/2}\, dr \tag{20}$$

where $J_{p/2-1}$ is a Bessel function of order $p/2 - 1$. A Bessel function of order α is given by:

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!} \Gamma\left(m + \alpha + 1\right) \left(\frac{1}{2}x\right)^{2m+\alpha} \tag{21}$$

4.3 Smoothness

The property of smoothness is outlined in the context of mean-squared (MS) differentiability and further analysed by the use of spectral density.

4.3.1 Differentiability

Smoothness of a process is defined similarly as smoothness of a deterministic function. From real analysis, the smoothness of a function C^k is determined by the degree of differentiability of the function $f^{(k)}(x)$. Mean-squared (MS) differentiability is the appropriate substitute when determining the smoothness of random processes. A general discussion on MS continuity and differentiability is found in Williams and Rasmussen (2006). Furthermore, Adler and Taylor (2009) prove this appropriate substitution where the MS differentiability of degree q means the differentiability of degree q for the sample paths of the process.

Domingo (2019) states a varation of the following if and only if statement. For stationary processes, if the 2^qth-order partial derivative exists and is finite at $r = 0$ then the qth order partial derivative exists for all $r \in R^D$ as a mean square limit.

Using the above statement, proofs of the SE kernel being infinitely MS differentiable and the exponential kernel not being MS differentiable are provided.

Claim 1. The SE kernel is infinitely MS differentiable.

Proof: Since the SE kernel is stationary, the statement above is used. The first-order partial derivative of the SE kernel is considered:

$$k(r) = exp(-\frac{r^2}{2\ell}) \tag{22}$$

$$\begin{aligned} k'(r) &= -\frac{r}{\ell} exp(-\frac{r^2}{2\ell}) \\ &= -\frac{r}{\ell} k(r) \quad \text{using Equation (22)} \end{aligned} \tag{23}$$

Therefore, a recursive formula is identified in Equation (23). Furthermore, from inspection, the qth-order derivatives, where $q \in \mathbb{N}_+$, all have a similar form of a polynomial function with respect to r multiplied by the initial SE kernel. This is demonstrated in Equation (24).

$$k^{(q)}(r) = p(r) \cdot k(r) \quad \text{where } p(r) \text{ is a polynomial function of degree } q \tag{24}$$

This is true for all even degrees of derivatives $2q$:

$$k^{(2q)}(r) = p(r) \cdot k(r) \quad \text{where } p(r) \text{ is a polynomial function of degree } 2q \tag{25}$$

Therefore, equating this 2^qth-order partial derivative, Equation (25), at $r = 0$.

$$\begin{aligned} k^{(2q)}(0) &= p(0) \cdot k(0) \\ &= p(0) \cdot 1 \\ &= p(0) \end{aligned}$$

Any polynomial function equated at zero is simply the constant of the polynomial which by definition is finite. The constant in this case will be determined by the length-scale ℓ.

Therefore, since the 2^qth-order partial derivative of the SE kernel exists and is finite at $r = 0$ for all $2q \in \mathbb{N}_+$, therefore the SE kernel qth order partial derivatives exist for all $q \in \mathbb{N}_+$ and the SE is infinitely MS differentiable. □

Claim 2. The Exponential kernel is not MS differentiable.

Proof: The exponential kernel written as function of τ and then differentiated highlights the issue:

$$\begin{aligned} k(r) &= exp\left(-\frac{r}{\ell}\right) \\ k(\tau) &= exp\left(-\frac{|\tau|}{\ell}\right) \\ k'(\tau) &= -\frac{\tau}{\ell|\tau|}k(\tau) \quad \text{for } \tau \neq 0 \end{aligned}$$

Since the first derivative is not defined at $\tau \neq 0$, the subsequent derivatives are not defined at $\tau \neq 0$. Therefore, from the statement, the exponential kernel is not MS differentiable. □

4.3.2 Spectral Density

The rate of decay of the spectral density corresponds to the degree of smoothness of a process. A slow rate of decay implies a "rougher" process whereas a fast rate of decay implies a "smoother" process. To confirm this statement, an assessment of some of the spectral densities of the previously mentioned kernels is conducted. The spectral densities stated are in one dimension for ease of understanding in Table 1 below.

From Figure 19, the rates of decay align with conclusions made of the smoothness of the four kernels. The SE kernel has the fastest rate of decay, implying the smoothest kernel out of the four. This is confirmed by its property of MS infinite differentiability. The following rates of decay of the spectral densities of the rest of the kernels in descending order are Matérn (5/2), Matérn (3/2), and Exponential. Additionally, this is confirmed by

Covariance Function	**Spectral Density**
Squared Exponential (SE)	$S(s) = \ell\sqrt{2\pi}\exp\left(-2\pi^2 s^2 \ell^2\right)$ (Williams and Rasmussen, 2006)
Matérn (3/2)	$S(s) = \frac{4\sqrt{3}}{\ell^3}\left(\frac{3}{\ell^2} + 4\pi^2 s^2\right)^{-2}$ (Williams and Rasmussen, 2006)
Matérn (5/2)	$S(s) = \frac{16\sqrt{5}}{3\ell^5}\left(\frac{5}{\ell^2} + 4\pi^2 s^2\right)^{-3}$ (Williams and Rasmussen, 2006)
Exponential	$S(s) = \frac{1}{\frac{\pi}{\ell} + \pi\ell s^2}$ (Heinonen, 2017)

Table 1: Covariance Functions and their Spectral Densities assuming variance $\sigma^2 = 1$.

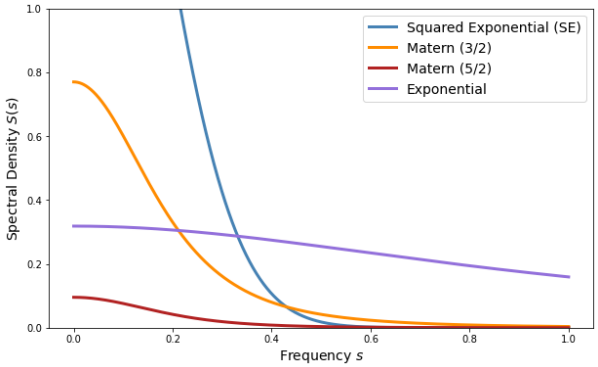


Figure 19: Plot of the spectral densities of the kernels in Table 1 with length-scale equal to one.

their orders of MS differentiability: second-order, first-order and not MS differentiable, respectively. Therefore, each kernel becomes "rougher". Listing 15 is an example of spectral density function created in python code.

```
# Squared Exponential (SE) Spectral Density
def se_spectral_density(s, l):
    return l * np.sqrt(2 * np.pi) * np.exp(-2 * np.pi**2 *
                                           s**2 * l**2)
```

Listing 15: Code of SE spectral density.

5 Exotic Covariance Functions

Later subsections discuss the application of kernels on datasets that include non-numeric types of data. In addition, discussions on the manipulation of kernels to map more complex multi-dimensional input spaces are had.

Basic kernels can be combined with each other to make more complex kernels. Kernels possess characteristics, or 'type[s] of structure' (Duvenaud, 2014, pp. 31), and multiplying or adding kernels together creates more complex structures, allowing the target function to better map the data.

5.1 Multiplication of Kernels

The product of two kernels is a kernel. The premise of the proof of this claim is that two independent random processes, $f_1(x)$ and $f_2(x)$, who have their own covariance functions $k_1(x, x')$ and $k_2(x, x')$, produce a new random process with a new covariance function, Equation (26) after being multiplied together. The proof can also be found in Williams and Rasmussen (2006, pp. 95).

$$k(\mathbf{x}, \mathbf{x'}) = k_1(\mathbf{x}, \mathbf{x'}) \cdot k_2(\mathbf{x}, \mathbf{x'}) \tag{26}$$

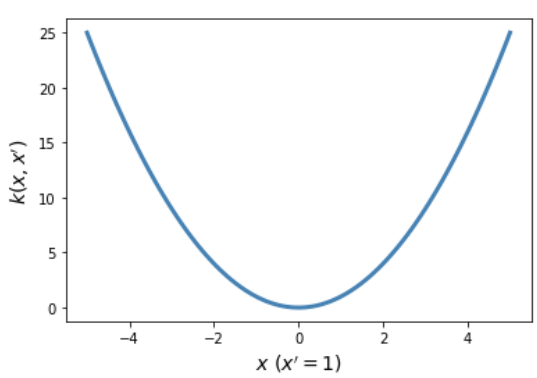
The underlying principle of multiplying kernels is that the value of the new kernel will be high only if the two kernels have high values independently on the input space.

The purpose of multiplying particular kernels is dependent on which basic kernels are combined. The following examples are from Duvenaud (2014):

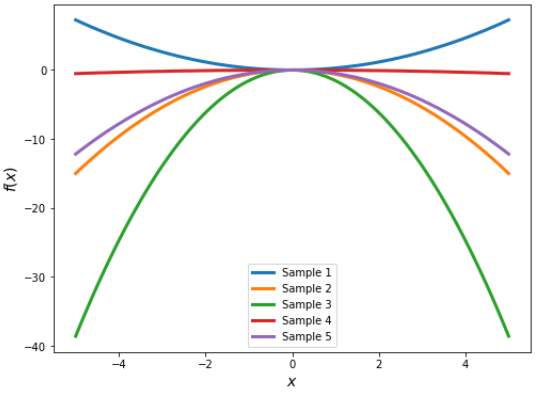
5.1.1 Polynomials

Polynomial structures can be obtained by the product of linear kernels. The number of linear kernels is equivalent to the polynomial degree of the function. Equation (27) show a quadratic kernel and a polynomial kernel of degree p and Figure 20 shows the prior distribution as a contour plot and when $x' = 1$ and posterior samples of a Gaussian process drawn from a quadratic kernel.

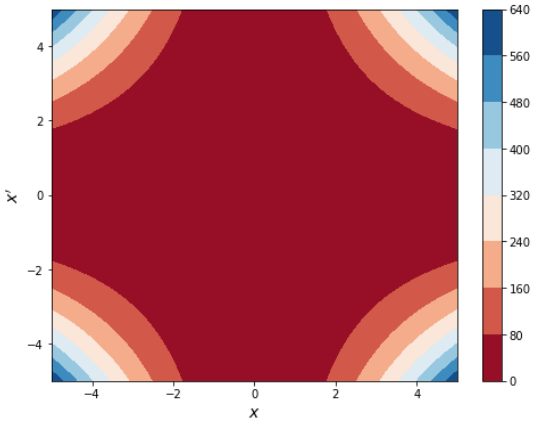
$$k_{\text{QUAD}}(x, x') = (x \cdot x')^2 \quad k_{\text{POLY}}(x, x') = (x \cdot x')^p \tag{27}$$



(a) Prior distribution of Quadratic kernel with $x' = 1$.



(b) Posterior samples drawn from quadratic kernel.



(c) Contour Plot of Prior distribution of Quadratic kernel.

Figure 20: Three plots all providing graphical understanding of quadratic the periodic kernel.

5.1.2 Global and Local

Kernels possess "global" and "local" structures. Linear and periodic kernels are considered a "global" structure because they capture trends that are present across the entire input space. The squared exponential and its variations, Matérn and rational quadratic, identify "local" trends and irregularities. The product of two kernels with different structures (i.e. "local" and "global") allows the new complex kernel to capture both trends in the target function on the input space.

5.2 Addition of Kernels

The addition of two kernels is a kernel. The proof can be found in Williams and Rasmussen (2006, pp. 95) and has similar premise to the product of kernels proof found

in Section 5.1.

$$k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}') \tag{28}$$

Similar to the product of kernels, kernel addition captures multiple features of the input space such as stationary or non-stationary, "global" or "local" trends, and degree of smoothness.

Unlike the product of kernels, the value of the new kernel is high if at least one of the base kernels has a high value. Therefore, the addition of kernels would be considered more flexible than the product of kernels.

5.3 Comparison of the Sum and Product of Kernels

This comparison is described using a real-world example that exhibits structures that suit two kernels. An exploration in how multiplying or adding these two kernels affects the model's capacity to map the data is conducted.

5.3.1 Example: Orange County GNP

This dataset is called *Orange County Employment* and can be found in Baltagi (2008). The dataset is a time-series (quarterly) from 1965-1983 measuring both employment and real gross national product (GNP) of Orange County, a region in southern California. Figure 21 is the time-series of the variable GNP.

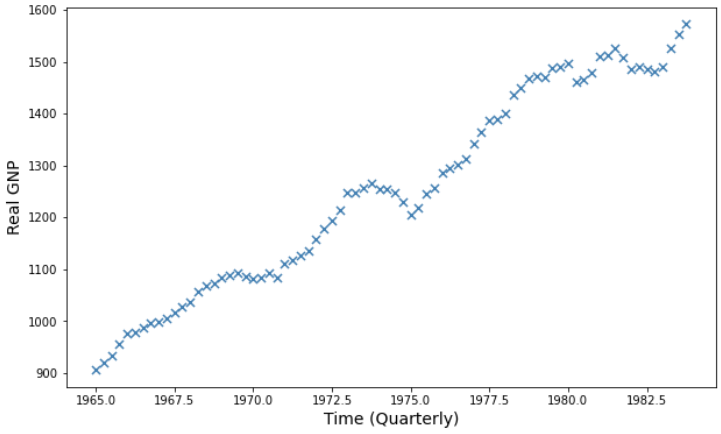


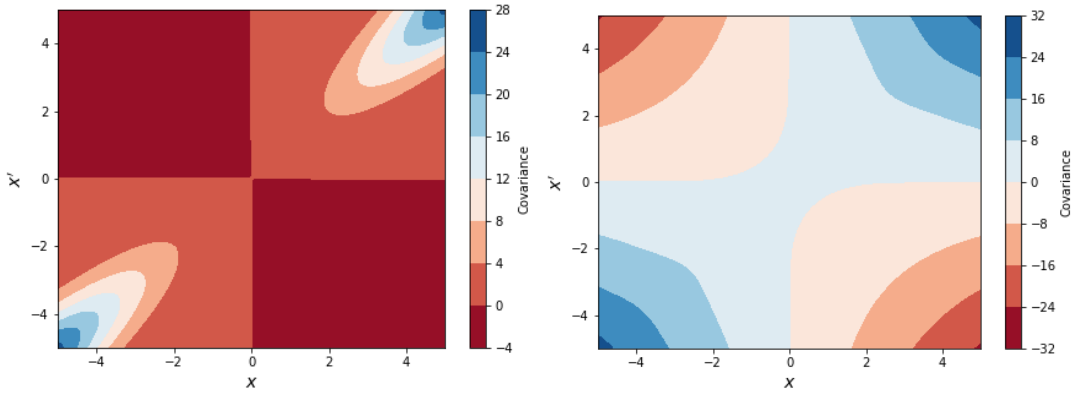
Figure 21: Time-series plot of the GNP of Orange County (quartely) from 1965 to 1983.

From inspection, there is a general "global" linear trend with some "local" variations. A fair conclusion from this inspection is to apply an amalgamation of the linear kernel and

SE kernel. Equations (29) and (30) represent the two possible transformations, sum and product respectively. Furthermore, Figure 22 shows how the prior distribution appears when the product or sum of the two kernels is evaluated. (Listing 16) describes the original code required to produce these contour plots in Figure 22.

$$k_{\text{sum}}(x, x') = k_{\text{SE}}(x, x') + k_{\text{Lin}}(x, x') = \sigma_{\text{SE}}^2 \exp\left(-\frac{(x - x')^2}{2l^2}\right) + \left(\sigma_{\text{Linear}}^2 \cdot x \cdot x'\right) \tag{29}$$

$$k_{\text{product}}(x, x') = k_{\text{SE}}(x, x') \cdot k_{\text{Lin}}(x, x') = \sigma_{\text{SE}}^2 \exp\left(-\frac{(x - x')^2}{2l^2}\right) \cdot \left(\sigma_{\text{Linear}}^2 \cdot x \cdot x'\right) \tag{30}$$



(a) $k_{\text{product}}(x, x') = k_{\text{SE}}(x, x') \cdot k_{\text{Lin}}(x, x')$ (b) $k_{\text{sum}}(x, x') = k_{\text{SE}}(x, x') + k_{\text{Lin}}(x, x')$

Figure 22: Contour plots of the sum kernel and product kernel made up of the SE and Linear kernel.

```
# Create a grid of x and x' values (input space)
x_values = np.linspace(-5, 5, 100)
x, x_prime = np.meshgrid(x_values, x_values)

# Compute the kernel values for the product kernel (SE *
   Linear)
K_product = SE_kernel(x, x_prime, length_scale=1.0,
                      variance=1.0) *
            linear_kernel(x, x_prime, variance=1.0, bias=0)
# Compute the kernel values for the sum kernel (SE + Linear)
K_sum = SE_kernel(x, x_prime, length_scale=1.0,
                  variance=1.0) +
        linear_kernel(x, x_prime, variance=1.0, bias=0)
```

Listing 16: Code of creating a meshgrid and new complex kernels that can be mapped onto a contour plot.

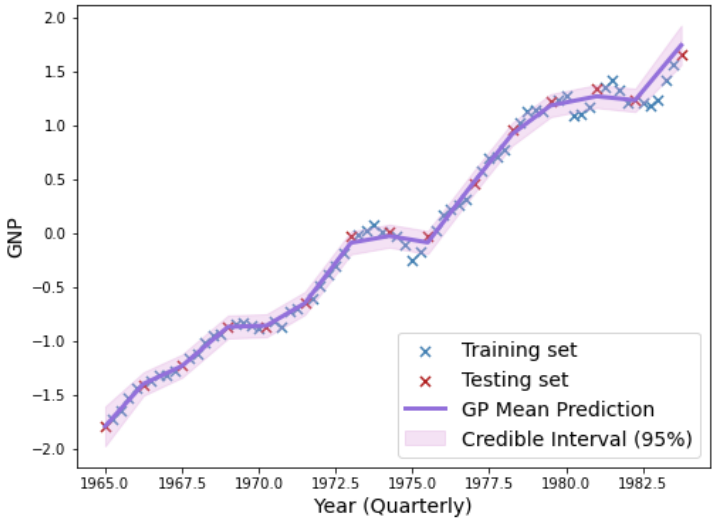


Figure 23: An optimised Gaussian process regression drawn from the sum of the SE and Linear kernel$k_{\text{sum}}(x, x')$. The hyperparemeters are the following: the Linear variance $\sigma_f^2 = 0$, the SE variance $\sigma_f^2 = 0.964$, the length-scale $\ell = 1.476$, and the Gaussian noise variance $\sigma_n^2 = 0.00228$.

Applying both new complex kernels to the dataset, the posterior distribution is found, along with the predictive mean and credible intervals. The results are shown in Figures 23 and 24.

```
# Create the sum of two kernels
kernel_1 =GPy.kern.Linear(input_dim=1, variances=1)
kernel_2 = GPy.kern.RBF(input_dim=1, variance=1.0,
                        lengthscale=1.0)
kernels = [kernel_1,kernel_2]
kernel = GPy.kern.Add(kernels)
```

Listing 17: Code of creating the sum of two kernels.

```
# Create the product of two kernels
kernel_1 =GPy.kern.Linear(input_dim=1, variances=1)
kernel_2 = GPy.kern.RBF(input_dim=1, variance=1.0,
                        lengthscale=1.0)
kernels = [kernel_1,kernel_2]
kernel = GPy.kern.Prod(kernels)
```

Listing 18: Code of creating the product of two kernels.

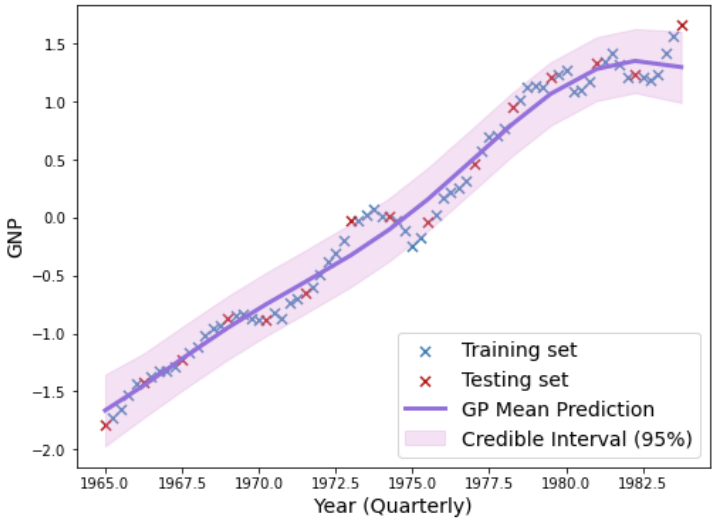


Figure 24: An optimised Gaussian process regression drawn from the product of the SE and Linear kernel $k_{\text{product}}(x, x')$. The hyperparemeters are the following: the Linear variance $\sigma_f^2 = 0.000595$, the SE variance $\sigma_f^2 = 0.000595$, the length-scale $\ell = 6.435$, and the Gaussian noise variance $\sigma_n^2 = 0.0181$.

The results of the two Gaussian process regressions with their respective kernels, $k_{\text{sum}}(x, x')$ and $k_{\text{product}}(x, x')$, suggest that the "sum" kernel provides a more accurate predictive mean with credible intervals having a smaller range. Clearly, the flexibility of the "sum" kernel is better for this Orange County GNP example, as it denies any suggestion of a linear trend (with $\sigma_f^2 = 0$) and is able to be mapped purely by the SE kernel.

5.4 Kernels in Multi-Dimensions

So far, every covariance function in this paper has been examined through one dimension of the input space. More complex models find explanation for the target function by many explanatory variables, a multi-dimensional input space. Using kernel addition and multiplication, new kernels are capable of mapping a function to the input-output multi-dimensional structure. The following two new kernels and their descriptions follow closely to Duvenaud's descriptions in his thesis (Duvenaud, 2014).

5.4.1 SE-ARD

SE-ARD stands for squared exponential automatic relevance determination. The premise of this kernel is the product of SE kernels over different dimensions.

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \prod_{i=1}^{d} \exp\left(-\frac{(x_i - x_i')^2}{2\ell_i^2}\right) = \sigma_f^2 \exp\left(-\frac{1}{2}\sum_{i=1}^{d} \frac{(x_i - x_i')^2}{\ell_i^2}\right) \tag{31}$$

where:

- σ_f^2 is the output variance, controlling the amplitude of the covariance function.
- ℓ_i is the length-scale for the i-th input dimension, controlling how much the input values in that dimension affect the kernel.
- d is the number of dimensions of the input.

The name of the kernel derives from the differing length-scale parameters, as they determine the "relevance" of each dimension. For instance, the feature is irrelevant if ℓ_i is large (Snelson, 2006). One hyperparameter per dimension and their ease of interpretability are the reasons for SE-ARD's ubiquitous use within Gaussian process application.

5.4.2 SE-ARD Example: Cross Country GDP growth rates

The dataset is called *Barro Data* and was initially created and used by Barro and Sala-i Martin (2004) and then later used by Koenker and Machado (1999). *Barro Data* consists of 161 observations on the determinants of cross-country GDP growth rates, the output y. The dataset includes 13 inputs, with variables such as education levels (male and female secondary and higher education), life expectancy, human capital, and various economic factors like investment and public consumption/GDP. The data spans two time periods: the first 71 observations represent countries during the period 1965-75, and the remaining 90 observations cover the period 1977-85. These variables provide insights into how factors like education, political stability, and economic policies influence GDP growth rates across different countries.

The suitability of *Barro Data* for SE-ARD lies in its multi-dimensional nature, with all variables being either numeric or binary, and its relatively stationary structure upon quick inspection. A brief inspection is made using the following code (Listing 19).

```
for column in data.columns[2:]:
    plt.figure(figsize=(8, 6))
    plt.scatter(data[column], target_y, color='blue')
    plt.xlabel(column)
    plt.ylabel('y.net')
    plt.show()
```

Listing 19: Code of creating plots to inspect on the structure of the data.

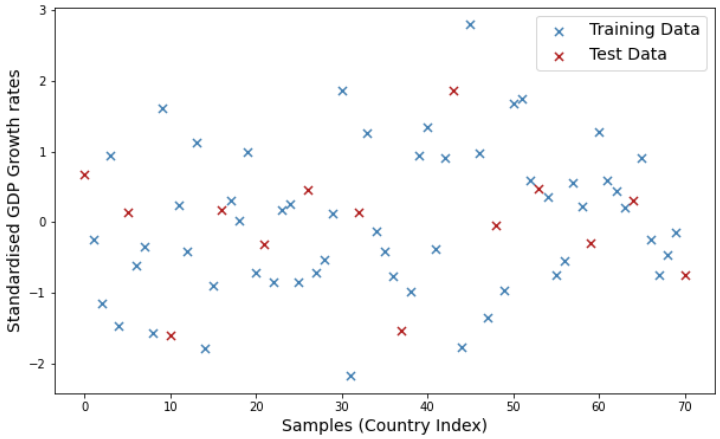


Figure 25: A plot of the *Barro Data* including the countries from the time period 1965-75 and their GDP growth rates split into training and test samples.

Before applying Gaussian processes with the SE-ARD kernel, alterations to *Barro Data* are made. The data analysed is restricted to the first 71 observations that represent countries during the period 1965-75. 20% of the data is taken as test samples, the rest as a training. Figure 25 illustrates all 71 different countries with the two colours referring to training and test sets and their respective GDP growth rate. In addition, the output and inputs are standardised.

```
# Create the SE-ARD kernel (Set lengthscales to 1 for all
   dimensions)
kernel =GPy.kern.RBF(input_dim=train_x.shape[1], variance=1.0,
                     lengthscale=np.full(train_x.shape[1], 6),
                     ARD=True)

# Create the GP Regression model
model = GPy.models.GPRegression(train_x, train_y, kernel)

# Optimize the model
model.optimize()

# Print the optimized model and hyperparameters (including
   lengthscales)
print(model)
print("\nLengthscales of the SE-ARD kernel:")
print(model.kern.lengthscale)
```

Listing 20: Code of running the GP regression with an SE-ARD kernel.

The Gaussian process model is run on the training set with the SE-ARD kernel with initial hyperparameter length scale ℓ value equal to six for all 13 dimensions of the varying inputs. The model is subsequently optimised (Section 6). Listing 20 describes the

relevant code for producing the SE-ARD kernel and printing the results of the optimum hyperparameters. From this, the inputs that possess extremely large length-scales suggest an irrelevancy of the input having explanatory power over the output, the GDP growth rates.

Figure 26 shows the correlation matrix (or the posterior covariance) $K(X', X')$ and Figure 27 illustrates the predictive mean and credible intervals found using Equations (4) and (5) with the test samples. The smaller credible intervals to the right of the 71 countries corresponds with the high correlation seen in the posterior covariance in Figure 26. Furthermore, Figure 27 suggests a high degree of accuracy with the predictive mean, with the distances between predictive mean and actual output y being small across the test samples, with only a few exceptions.

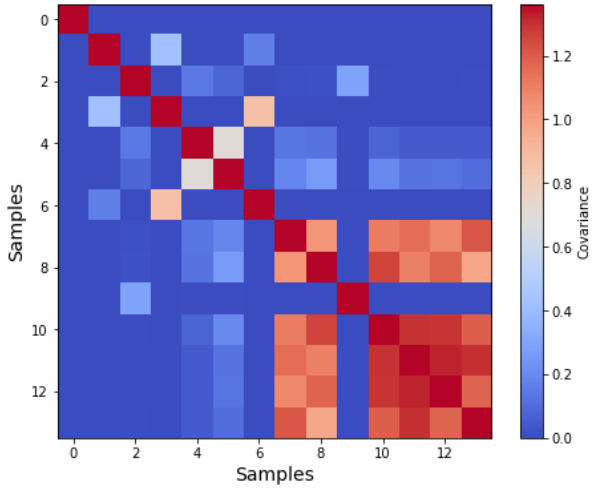


Figure 26: A correlation matrix (or posterior covariance) of the test samples $K(X', X')$.

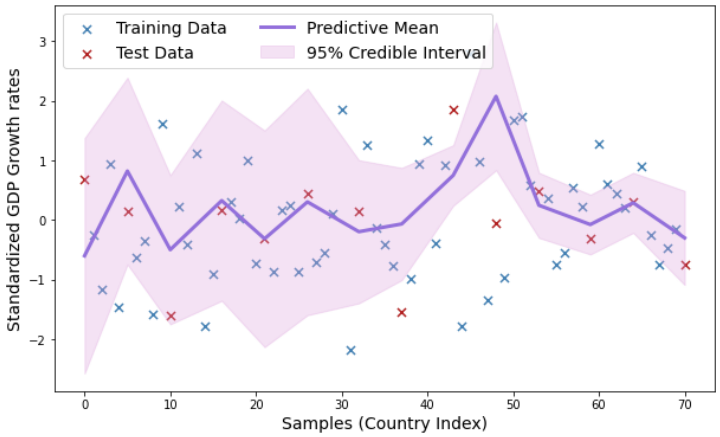


Figure 27: An optimised Gaussian process regression drawn from an SE-ARD kernel with predictive mean and credible intervals. The optimised hyperparameter values are found using the code in Listing 20.

5.4.3 Additive Model

A target function that has multiple inputs can be explained by a Gaussian process where the kernel is the sum of the same number of kernels as their dimensions within the input space such that each kernel takes a unique subset of the input space. This is illustrated in the below equation:

$$f(\mathbf{x}) \sim \mathcal{GP}(0, k_1(x_1, x_1') + k_2(x_2, x_2') + \cdots + k_d(x_d, x_d'))$$

5.4.4 Additive Example: Cross Country GDP Growth Rates

The dataset analysed is the *Barro Data* from 5.4.2. This section closely follows Duvenaud (2014) and his explanations of the additive model, including similar equations yet the difference is the dataset. The one Duvenaud (2014) uses is called *Compressive strength of concrete mixtures*, which was initially analysed using design of experiments and neural networks by Yeh (2006). It was later used in Kuhn (2013) on applied predictive modelling. This section focuses on the theory.

The suggested model, Equation (32), is the summation of one-dimensional functions f_i where $i = \{1, 2, \cdots, 13\}$ which are modelled using Gaussian processes with SE kernels.

$$f(\mathbf{x}) = f(x_1) + f(x_2) + \cdots + f(x_{13}) + \epsilon \quad \text{where } \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{32}$$

The following describes how the posterior mean and covariance of the additive model are found and is similar to the basic one-dimensional case in Section 2.4.2.

The joint prior distribution of this model, Equation (32), is considered:

$$\begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_{13}(\mathbf{x}) \\ \sum_{i=1}^{13} f_i(\mathbf{x}) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{13} \\ \sum_{i=1}^{13} \mu_i \end{bmatrix}, \begin{bmatrix} \mathbf{K}_1 & 0 & \dots & 0 & \mathbf{K}_1 \\ 0 & \mathbf{K}_2 & \dots & 0 & \mathbf{K}_2 \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \dots & \mathbf{K}_5 & \mathbf{K}_5 \\ \mathbf{K}_1 & \mathbf{K}_2 & \cdots & \mathbf{K}_5 & \sum_{i=1}^{13} \mathbf{K}_i \end{bmatrix} \right) \tag{33}$$

where $f_i(\mathbf{x})$ and $\mathbf{K}_i$, for $i = \{1, 2, \cdots, 13\}$, are the functions and Gram matrices, respectively, for the inputs.

The proof of Equation (5) can be used to show the conditional distribution from the joint distribution, Equation (33).

$$f_i(\mathbf{x}) \mid \sum_{i=1}^{13} f_i(\mathbf{x}) \sim \mathcal{N}\left(\mu_i + K_i^T \left(\sum_{i=1}^{13} K_i\right)^{-1} \left[\sum_{i=1}^{13} f_i(\mathbf{x}) - \sum_{i=1}^{13} \mu_i\right], K_i - K_i^T \left(\sum_{i=1}^{13} K_i\right)^{-1} K_i\right) \tag{34}$$

Equation (34) has the same uses mentioned in previous sections: a predictive mean and covariance that can be used to find credible intervals.

An additional equation derived is the posterior covariance of the additive components. This allows analysis between inputs, whether there is correlation. The equation is as follows:

$$\mathrm{Cov}\left[f_i(\mathbf{x}), f_j(\mathbf{x}) \mid f_i(\mathbf{x}) + f_j(\mathbf{x})\right] = -K_i^T (K_i + K_j)^{-1} K_j \quad \text{where } i, j = 1, \cdots, 13$$

5.5 Kernels on Categorical Variables

The types of data dealt with so far have been numeric data. Datasets contain a variety of data types including categorical data and transformations must be made to kernels to allow for categorical explanatory variables to influence a model's ability of mapping inputs.

A simple method is known as the one-of-k encoding. This reshapes the categorical explanatory variable into columns that act as true or false binaries where the data points correspond to a particular category.

For example, the one-of-k encoding of x, which takes one of three values ($x \in \{A, B, C\}$), corresponds to 3 binary inputs where one-of-k(A)=[1,0,0], one-of-k(B)=[0,1,0], one-of-k(C)=[0,0,1].

This transformation from one-dimensional categorical variable into a three-dimensional structure of binary variables means that multi-dimensional kernel is suitable to characterise the categorical variable. A common kernel is the SE-ARD, discussed previously, and illustrated for a categorical variable in Equation (35).

$$k_{\text{categorical}}(x, x') = \text{SE-ARD}(\text{one-of-k}(x), \text{one-of-k}(x')) \tag{35}$$

The hyperparameters play the same role as seen in Section 5.4.1, determining relevance of a variable.

5.5.1 Example: House Prices

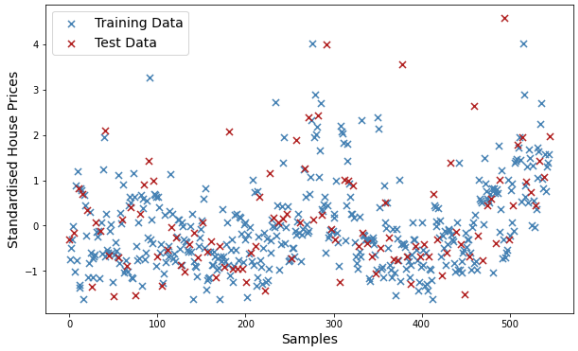
The *HousePrices* dataset contains sales data for houses sold in Windsor, Canada, during July, August, and September of 1987. It includes 546 observations on 12 variables, such as the sale price, lot size, number of bedrooms and bathrooms, number of stories, and other house features. The data set is used to study the factors that influence house prices in the city. It is cited in the context of semi-parametric estimation of a hedonic price function, as outlined by Anglin and Gencay (1996).

The focus is on the input of the number of stories. Whilst this is discrete, for the purpose of the example, it is treated as categorical with 5 options: 1, 2, 3, 4 stories. This one input is encoded into 4 new variables described above (Section 5.5). Listing 21 states the relevant code. *HousePrices*, as usual, is split into training and test examples and is illustrated in Figure 28a.

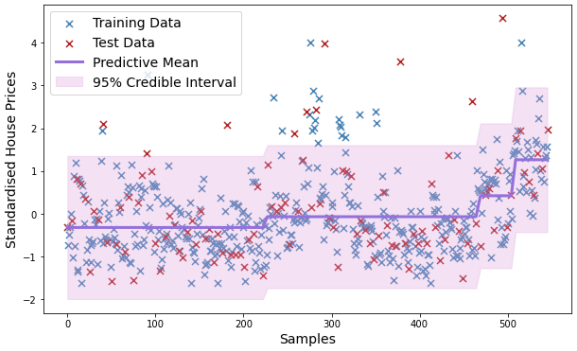
```
from sklearn.preprocessing import OneHotEncoder

# One-Hot Encoding of the x values
encoder = OneHotEncoder(sparse_output=False)  # Use
   sparse_output=False for dense array
encoded_stories = encoder.fit_transform(data[['stories']])
print(encoded_stories)

# Convert the encoded data to a DataFrame
encoded_data = pd.DataFrame(encoded_stories, data['rownames'],
    columns=encoder.categories_[0])
```

Listing 21: Code of encoding cateogrical variable into spearate binary variables.

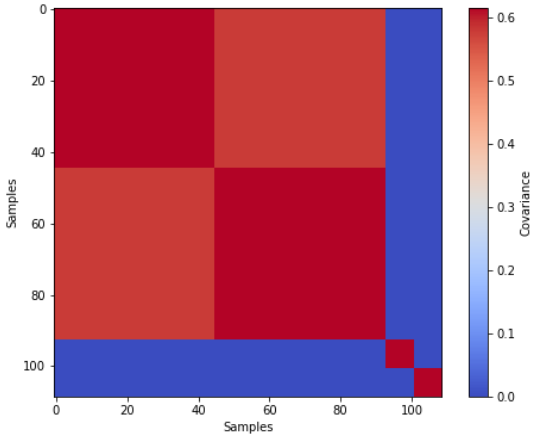
Figure 28a shows the houses sold at different prices in a particular order; from left to right the house increase in the number of stories. Figure 28b shows this change in the number of stories through the predictive mean and credible intervals produced from an SE-ARD drawn Gaussian process. Figure 28c is the correlation matrix on the test samples. The darker red boxes indicate the correlation of the samples where they share the same input. The input space is the 5 new variables. Figure 28c also indicates a positive correlation between house prices with one and two stories.



(a) A plot of House Prices in ascending order of level of stories, split into training and test samples.



(b) A plot of a Gaussian Process regression with SE-ARD kernel with predictive mean and credible intervals.



(c) Correlation matrix of the test samples with the level of stories of house in ascending order within the samples.

Figure 28: Three plots relating to the example using SE-ARD for categorical variables.

6 GP Model Selection

Having discussed the essential building blocks of Gaussian processes (kernels and their hyperparameters), the next logical step is how kernels and the value for their hyperparameters are chosen? A discussion on the method of model accuracy is conducted.

6.1 Marginal Likelihood

The following is from Williams and Rasmussen (2006).

Marginal likelihood, as described previously in Section 2.3.1, is a metric that evaluates the probability of output values y given input values by marginalising over function values f. Marginalisation is demonstrated by the integral in the following equation in the calculation of the marginal likelihood.

$$p(y|X) = \int p(y|f, X)p(f|X)df \tag{36}$$

Equation (36), while in principle is the same as Equation (2), is laid out differently. Firstly, the dataset $\mathcal{D}$ is split into its components of output values y and input values X.

Furthermore, the marginalisation is over parameters θ in Equation (2) rather than functions f. Equation (2) describes the marginal likelihood in the general sense of Bayesian statistics. On the other hand, Equation (36) above describes marginal likelihood in the context of Gaussian processes, a non-parametric model. The Gaussian process is described by its prior which is Gaussian and has a particular mean and covariance function, $f|X \sim \mathcal{N}(0, K)$. The covariance function has particular hyperparameters.

6.1.1 Proof of Analytical Tractability

The claim is that the marginal likelihood of a Gaussian process model has a closed solution.

There are two proofs; one is provided below and the other is found in Section A.4:

1: firstly, the formula of the joint probability density of a multivariate Gaussian distribution is:

$$p(\mathbf{x}|\mu, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu)^T\Sigma^{-1}(\mathbf{x} - \mu)\right) \tag{37}$$

- $\mathbf{x}$ is a d-dimensional vector of random variables.
- μ is the mean vector of the distribution, also d-dimensional.
- Σ is the $d \times d$ covariance matrix.
- $|\Sigma|$ is the determinant of the covariance matrix.
- Σ^{-1} is the inverse of the covariance matrix.

By the definition of a Gaussian process, the prior is Gaussian:

$$p(\theta) \sim \mathcal{N}(0, K) \tag{38}$$

Additionally, the likelihood function, $p(y|f, X)$, is a factorised Gaussian, since $y = f(x)+\epsilon$ and $\epsilon \sim \mathcal{N}(0, \sigma^2)$:

$$p(\theta) \sim \mathcal{N}(f, \sigma_n^2 \mathbf{I}) \tag{39}$$

The product of two Gaussians is a (normalised) Gaussian with an additional normalising constant (Williams and Rasmussen, 2006, see A.7).

$$\mathcal{N}(x|a, A)\mathcal{N}(x|b, B) = Z^{-1}\mathcal{N}(x|c, C) \tag{40}$$

$$Z^{-1} = (2\pi)^{-d/2}|A + B|^{-1/2} \exp\left(-\frac{1}{2}(a - b)^T(A + B)^{-1}(a - b)\right) \tag{41}$$

The last piece of information that will aid the proof is that the integral of joint probability density of a multivariate Gaussian distribution is one, describing all events occurring with their respective probabilities.

$$\int_{\mathbb{R}^d} \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu)^T\Sigma^{-1}(\mathbf{x} - \mu)\right) \, d\mathbf{x} = 1 \tag{42}$$

Now, using the above results, a sequential transformation can show that the marginal likelihood can be represented by a three-term equation that may be solved analytically.

$$
\begin{aligned}
p(\mathbf{y}|X) &= \int p(\mathbf{y}|\mathbf{f}, X)p(\mathbf{f}|X)\, d\mathbf{f} \\
&= \int \mathcal{N}(\mathbf{y}|\mathbf{f}, \sigma_n^2 I)\mathcal{N}(\mathbf{f}|0, K)\, d\mathbf{f} \quad \text{using Equations (38) and (39)} \\
&= \int \mathcal{N}(\mathbf{f}|\mathbf{y}, \sigma_n^2 I)\mathcal{N}(\mathbf{f}|0, K)\, d\mathbf{f} \\
&= \int Z^{-1}\mathcal{N}(\mathbf{f}|c, C) \quad \text{using Equation (40)} \\
&= Z^{-1} \cdot 1 \quad \text{using Equation (42)} \\
&= (2\pi)^{-D/2}|\sigma_n^2 I + K|^{-1/2} \exp\left(-\frac{1}{2}\mathbf{y}^T\left(\sigma_n^2 I + K\right)^{-1}\mathbf{y}\right) \quad \text{using Equation (41)}
\end{aligned} \tag{43}
$$

From this, taking logs of both sides produces the more interpretable equation.

$$
\log p(\mathbf{y}|X) = -\frac{D}{2}\log(2\pi) - \frac{1}{2}\log|\sigma_n^2 I + K| - \frac{1}{2}\mathbf{y}^T(\sigma_n^2 I + K)^{-1}\mathbf{y} \tag{44}
$$

□

Since the sum of covariance functions is also a covariance function (Section 5.2), the white noise term $\sigma_n^2 I$ in the log marginal likelihood $\log p(\mathbf{y}|X)$ is enveloped within the covariance function. Therefore, Equation (44) is simplified to:

$$
\log p(\mathbf{y}|X) = -\frac{D}{2}\log(2\pi) - \frac{1}{2}\log|K_y| - \frac{1}{2}\mathbf{y}^T K_y^{-1}\mathbf{y} \tag{45}
$$

where K_y represents the more complex covariance function of the sum of covariance function chosen for the prior distribution and the white noise covariance function.

6.1.2 Analysis of Log Marginal Likelihood

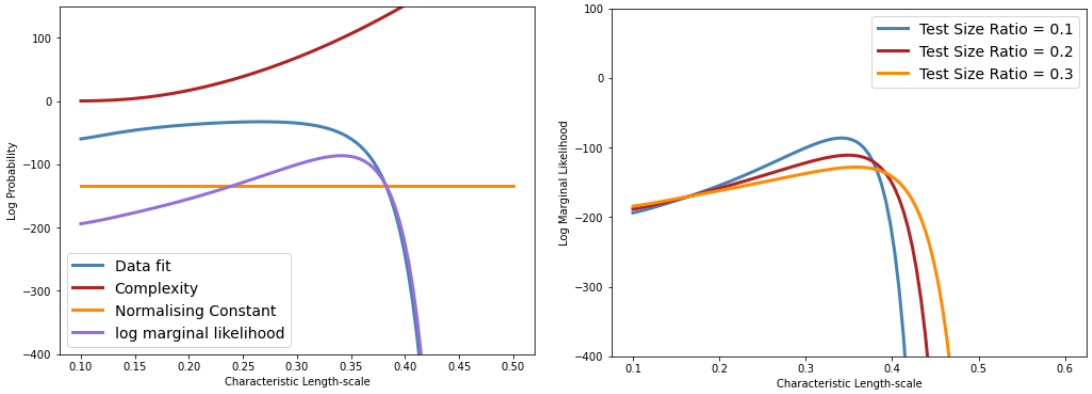
As seen in Equation (45), there are three terms that possess interpretable roles. The data fit term is $-\frac{1}{2}\mathbf{y}^T K_y^{-1}\mathbf{y}$ as it is the only term that includes the target y. The covariance function in this term, $-\frac{1}{2}\log|K_y|$, means it is the complexity penalty term and $-\frac{D}{2}\log(2\pi)$ is the normalisation constant.

The following Figure 29a, is a demonstration of how the the terms interact with the total log marginal likelihood. The data used for this illustration is *USMacroB* from Section 3.5.3. In this case, the SE kernel is used rather than RQ kernel. The usual Gaussian process regression is run on the *USMacroB* after being split into training and

test examples. From this, the three terms are equated for varying length-scales. Listing 22 describes the implementation of the terms of Equation (45) to produce Figure 29a.

```
# Create a list of different lengthscales to test
lengthscales = np.linspace(0.1, 2, 100)

# Lists to store term values
data_fits = []
complexitys = []
norm_consts = []
lmls = []

# Loop over each lengthscale and calculate the terms within
   marginal likelihood
for lengthscale in lengthscales:
    # Create an SE kernel with the given lengthscale
    kernel = GPy.kern.RBF(input_dim=1, variance=1,
                          lengthscale=lengthscale)

    # Fit the GP model using the training data
    model = GPy.models.GPRegression(train_x, train_y, kernel)
    K = np.linalg.inv(model.kern.K(train_x))

    # Calculate data fit term
    data_fit = np.dot(train_y.T, np.dot(K, train_y))
    data_fit =  -1/2 * data_fit.item()
    # Append to the list
    data_fits.append(data_fit)

    # Calculate complexity term
    complexity = - 1/2 * np.log(np.linalg.det(K)
    # Append to the list
    complexitys.append(complexity)

    # Calculate normalising constant term
    norm_const = - len(train_y)/2 *np.log(2*np.pi)
    # Append to the list
    norm_consts.append(norm_const)

    # Evaluate the marginal likelihood of the model
    lml = data_fit + complexity + norm_const
    # Append to the list
    lmls.append(lml)
```

Listing 22: Orginal Code of the the values of the three terms within Equation (45) with varying lengthscale.



(a) A plot of log marginal likelihood terms and their summation with varying characteristic length-scales (complexity).

(b) A plot of the log marginal likelihood against characteristic length-scale for varying sizes of data.

Figure 29: Two plots aiding the analysis of the log marginal likelihood.

The data fit decreases as the characteristic length-scale increases, particularly rapidly after the optimum length-scale is found. Williams and Rasmussen (2006) interprets the length-scale as the complexity of the model: a smaller length-scale allows for the model to map the finer variations within the data whereas a model with a larger length-scale can only capture the broader nature of the data. This rapid decrease makes intuitive sense since there exists a given length-scale where the model is simply unable to map the datapoints and therefore resulting in a very small datafit term. Again intuitively, the complexity term increases as the complexity of the model (characteristic length-scale) increases seen in Figure 29a. These terms combined with the normalising constant provide the log marginal likelihood.

Figure 29a identifies an important observation of the marginal likelihood. This property is that marginal likelihood already incorporates trade-off between model fit and model complexity.

This is known as Occam's razor; a principle that prefers explanations with the fewest elements. In the field of statistics, marginal likelihood favours a "goldilocks approach" where compromises are made on both sides of the spectrum of model fit and complexity.

Figure 29b demonstrates how log marginal likelihood interacts with varying sizes of datasets. An increase in the test size proportion means a reduction in the training set and therefore a decrease in datapoints, since K_y and $\mathbf{y}$ are made of training samples.

For very small datasets, the slope of the log marginal likelihood is relatively flatter, as both very small and moderate length-scale values are equally consistent with the observed data. However, as more data is gathered, the complexity term becomes more influential, penalizing excessively short length-scales. This is evidenced by Figure 29b and Williams and Rasmussen (2006)

6.1.3 Optimising Hyperparameters

The discrete nature of covariance function types means that the model with a particular covariance function that has the largest marginal likelihood is chosen.

The continuous nature of hyperprameters, $\theta \in \mathbb{R}$, allows for the partial derivative w.r.t. the given hyperparameter to be taken of the log marginal likelihood equation, set to zero and therefore find the hyperparameter value that maximises the log marginal likelihood.

The covariance function from equation, K_y is now treated as K for simplicity.

Claim:

$$\frac{\partial}{\partial \theta_j} \log p(\mathbf{y}|X, \theta) = \frac{1}{2}\mathrm{Tr}\left(\left(\alpha\alpha^T - K^{-1}\right)\frac{\partial K}{\partial \theta_j}\right) \quad \text{where} \quad \alpha = K^{-1}\mathbf{y} \tag{46}$$

Proof:

Three results are stated, one of those also being proved in Section A.5, that will aid in proving the original claim.

1. The derivative of an inverse matrix is the following:

$$\frac{\partial}{\partial \theta} K^{-1} = -K^{-1}\frac{\partial K}{\partial \theta}K^{-1} \tag{47}$$

2. The derivative of a log determinant matrix is the following:

$$\frac{\partial}{\partial \theta} \log |K| = \mathrm{Tr}\left(K^{-1}\frac{\partial K}{\partial \theta}\right) \tag{48}$$

These two results can be found in Williams and Rasmussen (2006, see A.14 and A.15).

3. A matrix identity (proof in Section A.5):

$$\mathbf{v}^T A \mathbf{v} = \mathrm{Tr}(\mathbf{v}\mathbf{v}^T A) \tag{49}$$

Now, the derivative of log marginal likelihood is revisited:

$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} \log p(\mathbf{y}|X,\theta) &= \frac{\partial}{\partial \theta_j}\left(-\frac{D}{2}\log(2\pi) - \frac{1}{2}\log|K| - \frac{1}{2}\mathbf{y}^T K^{-1}\mathbf{y}\right) \\
&= -\frac{1}{2}\mathrm{Tr}(K^{-1}\frac{\partial K}{\partial \theta_j}) + \frac{1}{2}\mathbf{y}^T K^{-1}\frac{\partial K}{\partial \theta_j}K^{-1}\mathbf{y} \quad \text{using Equation (47) and Equation (48)} \\
&= \frac{1}{2}\alpha^T\frac{\partial K}{\partial \theta_j}\alpha - \frac{1}{2}\mathrm{Tr}(K^{-1}\frac{\partial K}{\partial \theta_j}) \\
&= \frac{1}{2}\mathrm{Tr}(\alpha\alpha^T\frac{\partial K}{\partial \theta_j}) - \frac{1}{2}\mathrm{Tr}(K^{-1}\frac{\partial K}{\partial \theta_j}) \quad \text{using Equation (49)} \\
&= \frac{1}{2}\mathrm{Tr}\left(\left(\alpha\alpha^T - K^{-1}\right)\frac{\partial K}{\partial \theta_j}\right)
\end{aligned}
$$

□

The reasoning for the derivative to be represented in this way is two-fold: a more easily interpretable and computationally efficient formula. The trace function of the product of two matrices is an $\mathcal{O}(n^2)$ operation because their diagonal elements are only considered. If the matrix-matrix multiplication $\left(\alpha\alpha^T - K^{-1}\right)\frac{\partial K}{\partial \theta_j}$ occurs first then the trace is found, this operation costs $\mathcal{O}(n^3)$.

The only computationally taxing element of the derivative is finding the inverse of the covarinace matrix K^{-1}. This method for optimising the hyperparameters is known as gradient-based and is considered advantageous due to its computational efficiencies.

6.1.4 Optimisation Example: Orange County

Within the GPy package there is a method of optimising the model. The method chooses the 'preferred' optimiser and solves the problem, finding the hyperparameter values that maximise the log marginal likelihood.

This method is applied to the *Orange County Employment* data from Section 5.3.1. In that context two kernels were combined (sum and product), however in this case, the simple SE kernel is used.

```
print(model)
print(f'The value of maximised log maringal likelihood is:
      {model.log_likelihood()}')
```

Listing 23: Code provding the results.

Listing 23 provides the results of the optimised hyperparameters and the maximised value of the log marginal likelihood. A comparison is made to this result with a graphical

approach of finding the maximised value. Appendix B.5 describes a mesh grid of varying length-scales and Gaussian noises and a zero matrix is populated with the log marginal likelihood at each specific coordinate of length-scale and Gaussian noise. This is plotted as a contour and Figure 30 shows the results. From inspection, there is a discrepancy between the maximum log marginal likelihoods. If the number of points is increased in the mesh grid, the accuracy of the maximum log marginal likelihood increases. The optimise method uses optimisers such as the scaled conjugate gradient method which is extremely accurate compared to the above graphical method.

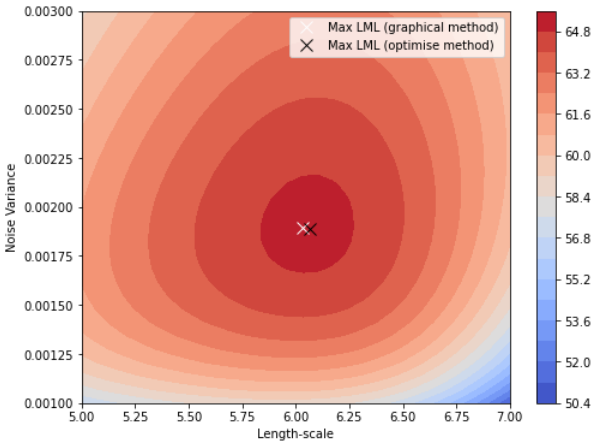


Figure 30: A contour plot of log marginal likelihood (LML) with varying Gaussian noise and length-scale.

7 Conclusion

7.1 Purpose

The aim of this dissertation was to develop a rigorous fundamental understanding of Gaussian Processes in theory, methodology, and application. In this work, definitions and proofs were outlined, accompanied with descriptions of how they fit into the wider Gaussian process narrative. An understanding of methodology was achieved by outlining the application of Gaussian processes to a dataset. Proceeding this, the paper went into detail in each step of the methodology: data transformation, kernel choice, optimisation and predictive mean and covariance. Throughout the dissertation, various datasets were extensively used to demonstrate the capability of Gaussian processes and described using figures. This procedure provided the reader with ways to grasp each subtopic of Gaussian processes.

An additional aim was to assert the advantages of Gaussian processes. Section 2.4.2 highlighted the main advantage in that Gaussian processess are non-parametric: decisions on choice of mean function and covariance function are based of properties of the function rather than the exact form of the function within the parametric approach. Thus Gaussian processes, as a modelling tool, are very flexible. This advantage is reinforced by the plethora of choice of both kernels and their transformations in Sections 3 and 5. Section 2.3.2 discussed the benefit of the Bayesian paradigm where the posterior distribution provides invaluable information on the Gaussian process model, particularly its predictive mean and covariance. Furthermore, Section 2.4.3 provided an explanation for the arguably more intuitive definition of credible intervals. Lastly, The result of Gaussian processes producing an analytically tractable solution for the marginal likelihood seen in Section 6.1.1 affords the reader to ignore the possible numerical approximations.

7.2 Further Research

Each chapter of the dissertation is finite and therefore gives rise to the opportunity of exploring further within the subtopics of Gaussian processes.

Further research for Chapter 2 could be an exploration in the debate between the Bayesian and frequentist paradigm. The debate was most lively in the 20th century with most published articles tending to favour one side over the other, often misrepresenting the opposing viewpoint (Hoff, 2009). An introduction to the debate that recounts both sides is the paper by Little (2006). An investigation might provide significant context for why the existence of Gaussian processes is dependent on the existence of the Bayesian paradigm.

The research avenues for Chapters 3, 4 and 5 are numerous. Firstly, other kernels could

be investigated such as the neural network kernel (Radford and Neal, 1996) which has the property of non-stationarity however does not possess the dot product term $x \cdot x'$ seen in the linear kernel (Section 3.6). The property of smoothness being determined by Eigen-analysis (Williams and Rasmussen, 2006, see section 4.3) is another research idea that might help readers with a computer science background. Alternative transformations of kernels could be explored, such as changepoints explained by Duvenaud (2014). Changepoints kernels model abrupt changes in the underlying function at specific points, allowing the covariance structure to shift at those locations.

A different approach to exotic kernels mapping complex data structures is the idea of manipulating data in such way that simple kernels might have greater success in mapping. This is known as pre-processing. Datasets in this dissertation have undergone pre-processing in a rudimentary sense: standardisation on the output and inputs and re-ordering. Pre-processing can be applied to the output space such as warped Gaussian processes (Snelson et al., 2003) or the input space. Calandra et al. (2016) introduce Manifold Gaussian Processes, an innovative supervised approach that simultaneously learns a transformation of the data into a feature space and performs Gaussian processes regression from the feature space to the observed space. These are possible research areas that would increase the number of tools that a statistician would have at their disposal.

The assumption made throughout the paper is of the prior mean function $m(\mathbf{x})$ being equal to zero. An exploration of eliminating this assumption could be conducted, with the discussion by Williams and Rasmussen (2006) being the initial line of inquiry. Ultimately, knowledge of setting an appropriate prior mean function will only aid model accuracy.

Cross-validation is an alternative method to marginal likelihood seen in Chapter 6. The focus of cross-validation is to determine the generalisation capabilities of the model. An additional measure would aid a statistician in deciding the most appropriate kernel and hyperparameters. An automated model selection method is described by Duvenaud (2014). It tests every kernel sequentially on a dataset, increasing in complexity if necessary by adding or multiplying more kernels, and using marginal likelihood for model comparison. A test of this algorithm on the datasets mentioned in this paper and comparison with the "human" analysis might contribute interesting insights.

Furthermore, the end of Section 6.1.3 discusses the most important issue within Gaussian processes: computational inefficiency. Therefore, an investigation into numerical approximations could be a basis for future research. Williams and Rasmussen (2006) suggests multiple ways to reduce computational inefficiency, such as reduced-rank approximations to the Gram matrix, approximate GP regression problems, and approximate marginal likelihood and its derivatives.

7.3 Final Remark

In conclusion, this dissertation has successfully provided both the theoretical foundation and practical insights into the power of Gaussian processes, demonstrating their versatility in modelling for regression problems.

A Appendix: Proof Aids

A.1 Joint and Conditional Distributions

Let $\mathbf{x}$ and $\mathbf{y}$ be jointly Gaussian random vectors such that

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_{\mathbf{x}} \\ \boldsymbol{\mu}_{\mathbf{y}} \end{bmatrix}, \begin{bmatrix} A & C^T \\ C & B \end{bmatrix} \right)$$

Therefore, the conditional distribution of $\mathbf{x}$ given $\mathbf{y}$ is the following:

$$\mathbf{x}|\mathbf{y} \sim \mathcal{N}(\boldsymbol{\mu}_{\mathbf{x}} + CB^{-1}(\mathbf{y} - \boldsymbol{\mu}_{\mathbf{y}}), A - CB^{-1}C^T) \tag{A.1}$$

This can be found in Von Mises (1967, see section 9.3).

A.2 Gamma Distribution

The gamma distribution is:

$$p(x; \alpha_*, \beta_*) = \frac{\beta_*^{\alpha_*}}{\Gamma(\alpha_*)} x^{\alpha_* - 1} \exp - \frac{\alpha_* x}{\beta_*} \tag{A.2}$$

The integral of a pdf is equal to one:

$$\int p(x; \alpha_*, \beta_*) = \int \frac{\beta_*^{\alpha_*}}{\Gamma(\alpha_*)} x^{\alpha_* - 1} \exp(-\frac{\alpha_* x}{\beta_*}) = 1 \tag{A.3}$$

A.3 Binomial Approximation

For a small $|x|$ yet large $|\alpha x|$, The binomial approximation is:

$$(1 + x)^{\alpha} \approx \exp(\alpha x) \tag{A.4}$$

Therefore in the specific proof in Section 3.5.2, the Rational quadratic kernel can be treated as a binomial expansion with $x = r^2/2\alpha\ell^2$.

A.4 Second Proof of Analytical Tractability

The target y is the sum of the latent function f and noise ϵ, randomly distributed. Since the prior$p(\theta)$ and noise ϵ are both normally distributed, and the sum of two normally distributed variables is also a normally distributed random variable. Therefore, the target y is normally distributed $\mathbf{y} \sim \mathcal{N}(0, K + \sigma_n^2 I)$

Using Equation (37) from the first approach, an equation for the marginal likelihood is directly found. This is the same as Equation (43).

A.5 Proof of a Matrix Identity

Proof of Equation (49).

Let $\mathbf{v}$ be an $n \times 1$ vector, $\mathbf{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$, and A be an $n \times n$ matrix with elements $a_{ij} \in \mathbb{R}$,

where:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$$

Now, consider the quadratic form $\mathbf{v}^T A \mathbf{v}$:

$$
\begin{aligned}
\mathbf{v}^T A\mathbf{v} &= \begin{pmatrix} v_1 & \cdots & v_n \end{pmatrix} \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \\
&= \begin{pmatrix} v_1 & \cdots & v_n \end{pmatrix} \begin{pmatrix} \sum_{i=1}^n a_{1i} v_i \\ \vdots \\ \sum_{i=1}^n a_{ni} v_i \end{pmatrix} \\
&= \sum_{i=1}^n v_i \left(\sum_{j=1}^n a_{ij} v_j \right) \\
&= v_1 \left(\sum_{i=1}^n a_{1i} v_i \right) + \cdots + v_n \left(\sum_{i=1}^n a_{ni} v_i \right) \\
&= v_1 \left(a_{11} v_1 + \cdots + a_{1n} v_n \right) + \cdots + v_n \left(a_{n1} v_1 + \cdots + a_{nn} v_n \right) \\
&= v_1^2 a_{11} + \cdots + v_1 a_{1n} v_n + \cdots + v_n a_{n1} v_1 + \cdots + a_{nn} v_n^2
\end{aligned}
$$

Now, consider the trace form on the RHS of Equation (49):

$$
\begin{aligned}
\mathrm{Tr}(\mathbf{v}\mathbf{v}^T A) &= \mathrm{Tr}(\begin{pmatrix} v_1 & \cdots & v_n \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}) \\
&= \mathrm{Tr}(\begin{pmatrix} v_1^2 & \cdots & v_1 v_n \\ \vdots & \ddots & \vdots \\ v_n v_1 & \cdots & v_n^2 \end{pmatrix} \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}) \\
&= \mathrm{Tr}(\begin{pmatrix} v_1^2 a_{11} + \cdots + v_1 v_n a_{n1} & \cdots & \cdot \\ \vdots & \ddots & \vdots \\ \cdot & \cdots & v_1 v_n + \cdots + a_{n1} v_n^2 \end{pmatrix} \text{ omission of non-diagonal entries sin} \\
&= v_1 \left(\sum_{i=1}^{n} a_{1i} v_i \right) + \cdots + v_n \left(\sum_{i=1}^{n} a_{ni} v_i \right) \\
&= v_1^2 a_{11} + \cdots + v_1 a_{1n} v_n + \cdots + v_n a_{n1} v_1 + \cdots + a_{nn} v_n^2
\end{aligned}
$$

The two terms are equivalent to one another. The proof of equation (49) is complete. □

B Appendix: Python Code

The following is the code used to produce the figures within this dissertation. This code has been reserved for the appendix as no new code is being displayed that has not already appeared in the main text. The purpose of this code is to further aid the reader in grasping implementation of Gaussian processes in python if the code outlined in the main text is insufficient.

B.1 Chapter 2

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
```

```
# Define the range of x values
x = np.linspace(-10, 10, 1000)


# Parameters for the three distributions
params = [(0, 1, 'firebrick'), (-5, 0.5, 'steelblue'),
          (5, 2, 'darkorange')]

# Plot the three distributions
plt.figure(figsize=(8, 6))
for mean, variance, colour in params:
    plt.plot(x, norm.pdf(x, loc=mean,
                         scale=np.sqrt(variance)),
             label=f'Mean = {mean}, Variance = {variance}',
             color=colour, linewidth=3)

# Add legend and labels
plt.xlabel('x', fontsize=14)
plt.ylabel('Probability Density', fontsize=14)
plt.legend(fontsize=14)

# Set x-axis ticks to only display the means
plt.xticks([mean for mean, _, _ in params])

# Display the plot
plt.show()
```

Listing 24: Code of the Creation of Figure 1

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function for 2D Gaussian PDF
def gaussian_pdf(x, x_prime, mu, K):

    diff = np.stack([x - mu[0], x_prime - mu[1]], axis=-1)
    inv_K = np.linalg.inv(K)
    norm_const = 1 / (2 * np.pi * np.sqrt(np.linalg.det(K)))

    return norm_const * np.exp(-0.5 * np.sum(diff @ inv_K *
                                             diff, axis=-1))

# Create a grid of points for the 2D space
x_1 = np.linspace(-4, 4, 100)
x_2 = np.linspace(-4, 4, 100)
```

```
X_1, X_2 = np.meshgrid(x_1, x_2)

# Parameters for the four plots
params = [([0, 0], np.array([[1, 0], [0, 1]])),
          ([2, -2], np.array([[1, 0], [0, 1]])),
          ([0, 0], np.array([[3, 0], [0, 0.5]])),
          ([0, 0], np.array([[1, -0.75], [-0.75, 1]]))]

# Loop over each plot and create separate contour plots
for i, (mu, K) in enumerate(params):
    Z = gaussian_pdf(X_1, X_2, mu, K)

    plt.figure(figsize=(6, 5))
    contour = plt.contour(X_1, X_2, Z, 20, cmap='coolwarm')
    plt.xlabel("x", fontsize=14)
    plt.ylabel("x'", fontsize=14)
    plt.show()
```

Listing 25: Code of the Creation of Figure 2

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Squared Exponential (SE) kernel function
def SE_kernel(x1, x2, length_scale=2.0, variance=1.0):
    return variance * np.exp(-0.5 * np.sum((x1 - x2)**2) /
                        length_scale**2)

# Generate 100 data points between -5 and 5
x = np.linspace(-5, 5, 100)

# Compute the covariance matrix using the SE kernel
K = np.array([[SE_kernel(xi, xj) for xj in x] for xi in x])

# Sample from the GP prior (mean vector = 0, covariance = K)
np.random.seed(100)  # For reproducibility
f_prior_samples =
np.random.multivariate_normal(mean=np.zeros(len(x)),cov=K,
                              size=3)

# Plot three different samples from the GP prior
plt.figure(figsize=(10, 6))
colours = ['firebrick', 'steelblue', 'darkorange']
for i, colour in enumerate(colours):
    plt.plot(x, f_prior_samples[i], color=colour,
             label=f'Sample {i+1}', linewidth=3)
```

```
# Add labels and legend
plt.xlabel("x", fontsize=14)
plt.ylabel("f(x)", fontsize=14)
plt.legend(fontsize=14)
# Show the plot
plt.show()
```

Listing 26: Code of the Creation of Figure 3

```
import numpy as np
import GPy
import matplotlib.pyplot as plt

# Define true function f(x)
def f(x):
    return 5 * np.sin(np.sqrt(x) - 5) + np.exp(-10*x) + x
       **(1/10) + 2

# Create noise epsilon
def noise(mean, std_dev):
    return np.random.normal(mean, std_dev)

# set random seed for reproducibility
np.random.seed(2)

# Sample 100 x and y values and reshape to ensure 2D array
x_values = np.arange(0, 100, 1).reshape(-1, 1)
y_values = np.array([f(x) + noise(0, 1) for x in x_values]).
   reshape(-1, 1)
true_y_values = np.array([f(x) for x in x_values]).reshape(-1,
    1)

# Standardize target_y (mean = 0, std = 1)
x_values = (x_values - x_values.mean()) / x_values.std()
y_values = (y_values - y_values.mean()) / y_values.std()
true_y_values = (true_y_values - true_y_values.mean()) /
   true_y_values.std()


# Split the dataset into training (80) and test (20) sets
   equally spaced across
# the x axis
test_indices = np.linspace(0, len(x_values)-1, 20, dtype=int)
train_indices = np.setdiff1d(np.arange(len(x_values)),
   test_indices)
```

```
# Create training samples of x and y values using the above
    indices
train_x = x_values[train_indices]
train_y = y_values[train_indices]

# Create training samples of x and y values using the above
    indices
test_x = x_values[test_indices]
test_y = y_values[test_indices]

# Create kernel and fit the Gaussian Process model to training
     set
kernel = GPy.kern.RBF(input_dim=1, variance=1.0, lengthscale
    =1.0)
model = GPy.models.GPRegression(train_x, train_y, kernel)

# Optimise the model parameters
model.optimize()

# Make predictions on the test set
mean_prediction, variance = model.predict(test_x)


# Plot the true function f(x)
plt.figure(figsize=(8, 6))
plt.plot(x_values, true_y_values, label='True Function', color
    ='darkorange', linewidth=3)
plt.xlabel('x', fontsize=14)
plt.ylabel('f(x)', fontsize=14)
plt.legend(loc='lower right', fontsize=14)
plt.ylim(-4, 3)
plt.show()

# Plot the test and training data
plt.figure(figsize=(8, 6))
plt.scatter(train_x, train_y, color='firebrick', marker='x',
    label='Training Data', s=50)
plt.scatter(test_x, test_y, color='mediumpurple', marker='x',
    label='Test Data', s=50)
plt.xlabel('x', fontsize=14)
plt.ylabel('f(x)', fontsize=14)
plt.legend(loc='lower right', fontsize=14)
plt.ylim(-4, 3)
plt.show()
```

```
# plot the mean prediction for the test set
plt.figure(figsize=(8, 6))
plt.plot(test_x, mean_prediction, label='GP Mean Prediction',
    color='steelblue', linewidth=3)

# Calculate the 95% credible interval
lower_bound = mean_prediction - 1.96 * np.sqrt(variance)
upper_bound = mean_prediction + 1.96 * np.sqrt(variance)

# Plot the confidence interval for the test set
plt.fill_between(test_x.flatten(), lower_bound.flatten(),
    upper_bound.flatten(), color='powderblue', alpha=0.3, label=
    'Credible Interval (95%)')

plt.xlabel('x', fontsize=14)
plt.ylabel('f(x)', fontsize=14)
plt.legend(loc='lower right', fontsize=14)
plt.ylim(-4, 3)
plt.show()

# Plot all components of the Guassian Process Methodology
plt.figure(figsize=(8, 6))

# Plot the true function
plt.plot(x_values, true_y_values, label='True Function', color
    ='darkorange', linewidth=3)

# PLot the mean function
plt.plot(test_x, mean_prediction, label='GP Mean Prediction',
    color='steelblue', linewidth=3)

# Plot the confidence interval for the test set
plt.fill_between(test_x.flatten(), lower_bound.flatten(),
    upper_bound.flatten(), color='powderblue', alpha=0.3, label=
    'Credible Interval (95%)')

# Plot the training and test data
plt.scatter(train_x, train_y, color='firebrick', marker='x',
    label='Training Data', s=50)
plt.scatter(test_x, test_y, color='mediumpurple', marker='x',
    label='Test Data', s=50)

# Specify plot parameters
plt.xlabel('x', fontsize=14)
plt.ylabel('f(x)', fontsize=14)
plt.legend(loc='lower right', ncol=2, fontsize=14)
```

```
plt.ylim(-4, 3)
plt.show()
```

Listing 27: Code of the Creation of Figure 4

B.2 Chapter 3

B.2.1 Section 3.2

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Squared Exponential (SE) kernel function
def SE_kernel(r, sigma_sq, lengthscale):
    return sigma_sq * np.exp(-0.5 * ((r) ** 2) / lengthscale
       **2)

# Create an array of r values (e.g., from -5 to 5)
r_values = np.linspace(-5, 5, 200)

# Create plot of prior distribution with varying length-scales
plt.figure(figsize=(10, 6))
y_values_2a = SE_kernel(r_values, 1.0, 0.5)
y_values_2b = SE_kernel(r_values, 1.0, 2.0)
y_values_1 = SE_kernel(r_values, 1.0, 1.0)
plt.plot(r_values, y_values_1, label=r"$\ell=1$", color='
   steelblue', linewidth=3)
plt.plot(r_values, y_values_2a, label=r"$\ell=0.5$", color='
   darkorange', linewidth=3)
plt.plot(r_values, y_values_2b, label=r"$\ell=2$", color='
   firebrick', linewidth=3)
plt.xlabel(r"$x - x'$", fontsize=14)
plt.ylabel(r"$k(x, x')$", fontsize=14)
plt.legend(fontsize=14)
plt.show()

# Create plot of prior distribution with varying scale factor
plt.figure(figsize=(10, 6))
y_values_3a = SE_kernel(r_values, 0.5, 1.0)
y_values_3b = SE_kernel(r_values, 2.0, 1.0)
y_values_1 = SE_kernel(r_values, 1.0, 1.0)
plt.plot(r_values, y_values_1, label=r"$\sigma^2=1$", color='
   steelblue', linewidth=3)
```

```
plt.plot(r_values, y_values_3a, label=r"$\sigma^2=0.5$", color
   ='darkorange', linewidth=3)
plt.plot(r_values, y_values_3b, label=r"$\sigma^2=2$", color='
   firebrick', linewidth=3)
plt.xlabel(r"$x - x'$", fontsize=14)
plt.ylabel(r"$k(x, x')$", fontsize=14)
plt.legend(fontsize=14)
plt.show()
```

Listing 28: Code of the Creation of Figures 5a and 5b

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Squared Exponential (SE) kernel function
def SE_kernel(x, x_prime, length_scale, variance):
    return variance * np.exp(-0.5 * np.sum((x - x_prime)**2) /
        length_scale**2)

# Generate some data points
x = np.linspace(-5, 5, 100)

# Set kernel parameters for each sample
kernel_params = [{"length_scale": 1.0, "variance": 1.0}, {"
   length_scale": 2.0, "variance": 1.0},{"length_scale": 0.5, "
   variance": 1.0}]

# Plot the GP prior for each set of parameters
plt.figure(figsize=(10, 6))

colours = ['firebrick', 'darkorange', 'steelblue']
labels = [r'$\ell=1$', '$\ell=5$', '$\ell=0.5$']

for i, params in enumerate(kernel_params):
    # Compute the covariance matrix using the SE kernel
    K = np.array([[SE_kernel(xi, xj, params['length_scale'],
       params['variance']) for xj in x] for xi in x])

    # Sample from the GP prior (mean vector = 0, covariance =
       K)
    np.random.seed(50 + i)  # Slightly different seed for
       reproducibility
    f_prior_samples = np.random.multivariate_normal(mean=np.
       zeros(len(x)), cov=K, size=1)

    # Plot the sample
```

```
    plt.plot(x, f_prior_samples[0], color=colours[i], label=
        labels[i],linewidth=3)

# Set plot parameters
plt.xlabel("x", fontsize=14)
plt.ylabel("f(x)", fontsize=14)
plt.legend(loc="upper right", fontsize=14)
plt.show()
```

Listing 29: Code of the Creation of Figures 5c

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Squared Exponential (SE) kernel function
def SE_kernel(x, x_prime, length_scale, variance):

    return variance * np.exp(-0.5 * np.sum((x - x_prime)**2) /
        length_scale**2)

# Generate some data points
x = np.linspace(-5, 5, 100)  # 100 points between -5 and 5

# Set kernel parameters for each sample
kernel_params = [
    {"length_scale": 1.0, "variance": 1.0},  # Red
    {"length_scale": 2.0, "variance": 4.0},  # Blue
    {"length_scale": 0.5, "variance": 0.5},  # Green
]

# Plot the GP prior for each set of parameters
plt.figure(figsize=(10, 6))

colours = ['firebrick', 'darkorange', 'steelblue']
labels = [r'$\ell=1, \sigma^2=1$', r'$\ell=2, \sigma^2=4$', r'
    $\ell=0.5, \sigma^2=0.5$']

for i, params in enumerate(kernel_params):
    # Compute the covariance matrix using the SE kernel
    K = np.array([[SE_kernel(xi, xj, params['length_scale'],
        params['variance']) for xj in x] for xi in x])

    # Sample from the GP prior (mean vector = 0, covariance =
        K)
    np.random.seed(50 + i)  # Slightly different seed for
        reproducibility
```

```
    f_prior_samples = np.random.multivariate_normal(mean=np.
        zeros(len(x)), cov=K, size=1)

    # Plot the sample
    plt.plot(x, f_prior_samples[0], color=colours[i], label=
        labels[i], linewidth=3)

# Add labels, title, and legend
plt.xlabel("x", fontsize=14)
plt.ylabel("f(x)", fontsize=14)
plt.legend(loc="lower left", ncol=3, fontsize=14)
plt.grid(False)

# Show the plot
plt.show()
```

Listing 30: Code of the Creation of Figure 5d

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import GPy


# Accessing the dataset
file_path = r'C:\Users\student-local\OneDrive - University of
    Birmingham\Dissertation\CartelStability.csv'
data = pd.read_csv(file_path)

# Create a suitable time variable and extract quantity
data['time'] = 1880 + data.index / 52
x = data['time'].values.reshape(-1,1)
y = data['quantity'].values.reshape(-1, 1)
y = (y - np.mean(y)) / np.std(y)
# Set the seed for reproducibility
np.random.seed(1)

# 20% of the data for the test set
test_size = int(0.2 * len(data))  # 15% of the data

# Evenly Distributed
# Select indices evenly distributed for the test set
test_indices = np.linspace(0, len(data) - 1, test_size, dtype=
    int)

# The rest of the data will be the training set
```

```
train_indices = np.setdiff1d(np.arange(len(data)),
    test_indices)

# Split the data into training and test sets
train_data = data.loc[train_indices]
test_data = data.loc[test_indices]

# Extract time (x) and quantity (y) for the full dataset
#x = data.index.values.reshape(-1, 1)  # Time (index) as a 2D
    array
#y = data['quantity'].values.reshape(-1, 1)  # Quantity as a 2
    D array

# Split the dataset into train and test sets
train_x = x[train_indices]  # Training x values (time)
train_y = y[train_indices]  # Training y values (quantity)

test_x = x[test_indices]    # Test x values (time)
test_y = y[test_indices]

# Plot the test and training data
plt.figure(figsize=(20, 6))
plt.scatter(train_x, train_y, color='firebrick', marker='x',
    label='Training Data', s=70)
plt.scatter(test_x, test_y, color='steelblue', marker='x',
    label='Test Data', s=70)
plt.xlabel('Time (years)', fontsize=14)
plt.ylabel('f(x), Quantity', fontsize=14)
plt.legend(loc='upper left', fontsize=14)
plt.show()

# 5. Fit the Gaussian Process model
kernel = GPy.kern.RBF(input_dim=1, variance=1.0, lengthscale
    =2.0)
print(kernel.K(train_x))
model = GPy.models.GPRegression(train_x, train_y, kernel)
model.optimize(messages=True)
print(model)
# 6. Make predictions on the test set
mean_prediction, variance = model.predict(test_x)

plt.figure(figsize=(20, 6))
plt.scatter(train_x, train_y, label='Train Quantity', color='
    firebrick', marker='x', s=70)
plt.scatter(test_x, test_y, label='Test Quantity', color='
    steelblue', marker='x', s=70)
```

```
plt.plot(test_x, mean_prediction, label='GP Mean Prediction',
   color='mediumpurple', linewidth=3)

# Calculate the 95% confidence interval
lower_bound = mean_prediction - 1.96 * np.sqrt(variance)
upper_bound = mean_prediction + 1.96 * np.sqrt(variance)

# Plot the confidence interval for the test set
plt.fill_between(test_x.flatten(), lower_bound.flatten(),
   upper_bound.flatten(), color='plum', alpha=0.3, label='
   Credible Interval (95%)')
plt.xlabel('Time (years)', fontsize=14)
plt.ylabel('f(x), Quantity', fontsize=14)
plt.legend(loc='upper left', ncol=2, fontsize=14)
plt.show()
```

Listing 31: Code of the Creation of Figures 6 and 7

B.2.2 Section 3.3

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Matern kernel function
def matern_kernel(r, v, sigma_sq, lengthscale):

    if v == 1/2:
        return sigma_sq * np.exp(-r / lengthscale)
    elif v == 3/2:
        return sigma_sq * (1 + np.sqrt(3) * r / lengthscale) *
            np.exp(-np.sqrt(3) * r / lengthscale)
    elif v == 5/2:
        return sigma_sq * (1 + np.sqrt(5) * r / lengthscale +
           (5 / 3) * (r**2) / lengthscale**2) * np.exp(-np.sqrt
           (5) * r / lengthscale)

# Create an array of x - x' values (e.g., from 0 to 5)
r_values = np.linspace(0, 5, 200)

# Second Plot: Matern Kernels with fixed var=1 and varying
   lengthscale (below and above 1)
plt.figure(figsize=(10, 6))
y_values_2a = matern_kernel(r_values, 3/2, 1.0, 0.5)  #
   lengthscale below 1
```

```
y_values_2b = matern_kernel(r_values, 3/2, 1.0, 2.0)  #
    lengthscale above 1
y_values_1 = matern_kernel(r_values, 3/2, 1.0, 1.0)
plt.plot(r_values, y_values_1, label=r"$\ell=1$", color='
    steelblue')
plt.plot(r_values, y_values_2a, label=r"$\ell=0.5$", color='
    firebrick')
plt.plot(r_values, y_values_2b, label=r"$\ell=2$", color='
    darkorange')
plt.xlabel(r"$x - x'$", fontsize=14)
plt.ylabel(r"$k(x, x')$", fontsize=14)
plt.legend(fontsize=14)
plt.grid(False)
plt.show()

# Third Plot: Matern Kernels with fixed lengthscale=1 and
    varying variance (below and above 1)
plt.figure(figsize=(10, 6))
y_values_3a = matern_kernel(r_values, 3/2, 0.5, 1.0)  #
    variance below 1
y_values_3b = matern_kernel(r_values, 3/2, 2.0, 1.0)  #
    variance above 1
y_values_1 = matern_kernel(r_values, 3/2, 1.0, 1.0)
plt.plot(r_values, y_values_1, label=r"$\sigma^2=1$", color='
    steelblue')
plt.plot(r_values, y_values_3a, label=r"$\sigma^2=0.5$", color
    ='firebrick')
plt.plot(r_values, y_values_3b, label=r"$\sigma^2=2$", color='
    darkorange')
plt.xlabel(r"$x - x'$", fontsize=14)
plt.ylabel(r"$k(x, x')$", fontsize=14)
plt.legend(fontsize=14)
plt.grid(False)
plt.show()

# Fourth Plot: Matern Kernels with varying v (1/2, 3/2, 5/2)
    and fixed lengthscale and variance
plt.figure(figsize=(10, 6))
y_values_v_half = matern_kernel(r_values, 1/2, 1.0, 1.0)
y_values_v_three_half = matern_kernel(r_values, 3/2, 1.0, 1.0)
y_values_v_five_half = matern_kernel(r_values, 5/2, 1.0, 1.0)
plt.plot(r_values, y_values_v_half, label=r"$v=\frac{1}{2}$",
    color='firebrick')
plt.plot(r_values, y_values_v_three_half, label=r"$v=\frac
    {3}{2}$", color='steelblue')
```

```
plt.plot(r_values, y_values_v_five_half, label=r"$v=\frac
   {5}{2}$", color='darkorange')
plt.xlabel(r"$x - x'$", fontsize=14)
plt.ylabel(r"$k(x, x')$", fontsize=14)
plt.legend(fontsize=14)
plt.grid(False)
plt.show()
```

Listing 32: Code of the Creation of Figures 8c,8a and 8b

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Matern kernel function for different v
def matern_kernel(x1, x2, v, sigma_sq=1.0, lengthscale=1.0):
    r = np.abs(x1 - x2)
    if v == 1/2:
        return sigma_sq * np.exp(-r / lengthscale)
    elif v == 3/2:
        return sigma_sq * (1 + np.sqrt(3) * r / lengthscale) *
            np.exp(-np.sqrt(3) * r / lengthscale)
    elif v == 5/2:
        return sigma_sq * (1 + np.sqrt(5) * r / lengthscale +
           (5 / 3) * (r**2) / lengthscale**2) * np.exp(-np.sqrt
           (5) * r / lengthscale)
    else:
        return sigma_sq * np.exp(-r**2 / (2 * lengthscale**2))
             # SE kernel when v -> infinity

# Generate some data points
x = np.linspace(-5, 5, 100)  # 100 points between -5 and 5

# Set kernel parameters for each sample
kernel_params = [
    {r'$v$': 1/2, "label": r"$v=1/2$", "color": 'firebrick'},
         # Red for v=1/2
    {r'$v$': 3/2, "label": r"$v=3/2$", "color": 'steelblue'},
         # Blue for v=3/2
    {r'$v$': 5/2, "label": r"$v=5/2$", "color": 'darkorange'},
          # Green for v=5/2
    {r'$v$': np.inf, "label": "$v∞= (SE)$", "color": '
       mediumpurple'}  # Purple for SE kernel (v∞->)
]

# Plot the GP prior for each set of parameters
plt.figure(figsize=(10, 6))
```

```
for i, params in enumerate(kernel_params):
    # Compute the covariance matrix using the Matern kernel
    K = np.array([[matern_kernel(xi, xj, v=params[r'$v$'],
       sigma_sq=1.0, lengthscale=1.0) for xj in x] for xi in x
       ])

    # Sample from the GP prior (mean vector = 0, covariance =
       K)
    np.random.seed(100+i)  # Same seed for reproducibility
    f_prior_samples = np.random.multivariate_normal(mean=np.
       zeros(len(x)), cov=K, size=1)

    # Plot the sample
    plt.plot(x, f_prior_samples[0], label=params['label'],
       color=params['color'], linewidth=3)

# Add labels, title, and legend
plt.xlabel("x", fontsize=14)
plt.ylabel("f(x)", fontsize=14)
plt.legend(loc="upper right",ncol=2, fontsize=14)
# Show the plot
plt.show()
```

Listing 33: Code of the Creation of Figure 8d

```
import numpy as np
import pandas as pd
import GPy
from matplotlib import pyplot as plt

# Accessing, and giving variable labels.
# access dataset from file
file_path = r'C:\Users\student-local\OneDrive - University of
   Birmingham\Dissertation\gafa_stock.csv'
data = pd.read_csv(file_path)
#print(data)
# Filter dataset for AAPL only
aapl_data = data[data['Symbol'] == 'AAPL']  # Assuming 'symbol
   ' column identifies stock ticker

# Create the 'time' variable as a float, starting from 2014
aapl_data['time'] = 2014 + (aapl_data.index + 1) / 365
x = aapl_data['time'].values.reshape(-1,1)
y = aapl_data['Adj_Close'].values.reshape(-1, 1)
```

```
plt.figure(figsize=(20, 6))
plt.plot(x, y, color='steelblue', linewidth=3)

# Add labels and title
plt.xlabel('Time (years)', fontsize=14)
plt.ylabel('Adjusted Closing Stock Price, USD\$', fontsize=14)
plt.show()

# Standardise y
y = (y - np.mean(y)) / np.std(y)

# 10% of the data for the test set
test_size = int(0.1 * len(aapl_data))
test_indices = np.linspace(0, len(aapl_data) - 1, test_size,
   dtype=int)
train_indices = np.setdiff1d(np.arange(len(aapl_data)),
   test_indices)

# Split the dataset into train and test sets
train_x = x[train_indices]
train_y = y[train_indices]
test_x = x[test_indices]
test_y = y[test_indices]

kernel = GPy.kern.Matern32(input_dim=1, variance=1.0,
   lengthscale=1.0)
model = GPy.models.GPRegression(train_x, train_y, kernel)
model.optimize()
print(model)

mean_prediction, variance = model.predict(test_x)

plt.figure(figsize=(20, 6))

# Scatter plot of normalized data
plt.scatter(test_x, test_y, color='firebrick', marker='x',
   label='Test samples', s=30)

# Plot the predictive mean for this kernel
plt.plot(test_x, mean_prediction, label='mean prediction',
   color='mediumpurple', linewidth=3)
# Calculate the 95% confidence intervals
lower_bound = mean_prediction - 1.96 * np.sqrt(variance)
upper_bound = mean_prediction + 1.96 * np.sqrt(variance)
# Plot the confidence intervals
plt.fill_between(test_x.flatten(), lower_bound.flatten(),
```

```
    upper_bound.flatten(),label='95% credible intervals', color=
    'plum', alpha=0.4)

# Add labels, title, and legend
plt.xlabel('Date', fontsize=14)
plt.ylabel('Normalised Adjusted Closing Stock Price', fontsize
    =14)
plt.legend(loc='upper left', fontsize=14)
plt.grid(False)
# Show the plot
plt.show()
```

Listing 34: Code of the Creation of Figures 9 and 10

B.2.3 Section 3.4

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Periodic kernel function
def periodic_kernel(r, period=1.0, sigma_sq=1.0, lengthscale
    =1.0):
     return sigma_sq * np.exp(-2 * np.sin(np.pi * r / period)
        **2 / lengthscale**2)

# Create an array of r values (e.g., from 0 to 5)
r_values = np.linspace(0, 5, 200)  # 200 points between 0 and
    5

# Plot 1: Periodic Kernel with Period = 0.5
plt.figure(figsize=(10, 6))
y_values_1 = np.array([periodic_kernel(ri, period=0.5) for ri
    in r_values])
plt.plot(r_values, y_values_1, color='firebrick')
plt.xlabel(r"$x-x'$", fontsize=14)
plt.ylabel(r"$k(x,x')$", fontsize=14)
plt.show()

# Plot 2: Periodic Kernel with Period = 1
plt.figure(figsize=(10, 6))
y_values_2 = np.array([periodic_kernel(ri, period=1) for ri in
     r_values])
plt.plot(r_values, y_values_2, color='steelblue')
plt.xlabel(r"$x-x'$", fontsize=14)
```

```
plt.ylabel(r"$k(x,x')$", fontsize=14)
plt.show()

# Plot 3: Periodic Kernel with Period = 2
plt.figure(figsize=(10, 6))
y_values_3 = np.array([periodic_kernel(ri, period=2) for ri in
    r_values])
plt.plot(r_values, y_values_3, color='darkorange')
plt.xlabel(r"$x-x'$", fontsize=14)
plt.ylabel(r"$k(x,x')$", fontsize=14)
plt.show()
```

Listing 35: Code of the Creation of Figures 11a,11b and 11c

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Periodic kernel function
def periodic_kernel(x, x_prime, period=1.0, sigma_sq=1.0,
   lengthscale=1.0):
    """Periodic Kernel Function"""
    r = np.abs(x - x_prime)
    return sigma_sq * np.exp(-2 * np.sin(np.pi * r / period)
       **2 / lengthscale**2)

# Generate some data points
x = np.linspace(-5, 5, 200)  # 100 points between -5 and 5

# Set kernel parameters for each sample (varying period, fixed
    variance and lengthscale)
kernel_params = [
    {"period": 1.0, "label": r"$p = 0.5$", "color": 'firebrick
       '},   # Red for period = 0.5
    {"period": 2.0, "label": r"$p = 1.0$", "color": 'steelblue
       '},   # Blue for period = 1.0
    {"period": 5.0, "label": "$p = 2.0$", "color": 'darkorange
       '},   # Green for period = 2.0
]

# Plot the GP prior for each set of parameters
plt.figure(figsize=(10, 6))

for i, params in enumerate(kernel_params):
    # Compute the covariance matrix using the Periodic kernel
    K = np.array([[periodic_kernel(xi, xj, period=params['
       period'], sigma_sq=1.0, lengthscale=1.0) for xj in x]
```

```
        for xi in x])

    # Sample from the GP prior (mean vector = 0, covariance =
        K)
    np.random.seed(10000+i)  # Same seed for reproducibility
    f_prior_samples = np.random.multivariate_normal(mean=np.
        zeros(len(x)), cov=K, size=1)

    # Plot the sample
    plt.plot(x, f_prior_samples[0], label=params['label'],
        color=params['color'])

# Add labels, title, and legend
plt.xlabel("x", fontsize=14)
plt.ylabel("f(x)", fontsize=14)
plt.legend(loc="lower right", ncol=3, fontsize=14)
plt.ylim(-2.5,1.6)
plt.grid(False)
# Show the plot
plt.show()
```

Listing 36: Code of the Creation of Figure 11d

```
import numpy as np
import pandas as pd
import GPy
from matplotlib import pyplot as plt

# Accessing, and giving variable labels.
# access dataset from file
file_path = r'C:\Users\student-local\OneDrive - University of
    Birmingham\Dissertation\nottem.csv'
data = pd.read_csv(file_path)


plt.figure(figsize=(20, 6))  # Create a new figure for each
    variable
plt.plot(data['time'], data['value'], color='steelblue',
    marker='x', linewidth=3, markersize=10)
# Adding labels and title
plt.xlabel('Time (monthly)', fontsize=14)
plt.ylabel('Temperature Value', fontsize=14)

# Show the plot
plt.show()
```

```
# Extract time (x) and quantity (y) for the full dataset
x = data['time'].values.reshape(-1, 1)  # Time (index) as a 2D
    array
y = data['value'].values.reshape(-1, 1)  # Quantity as a 2D
   array
# Standardise y
y = (y - np.mean(y)) / np.std(y)

# 10% of the data for the test set
test_size = int(0.2 * len(data))
test_indices = np.linspace(0, len(data) - 1, test_size, dtype=
   int)
train_indices = np.setdiff1d(np.arange(len(data)),
   test_indices)

# Split the dataset into train and test sets
train_x = x[train_indices]
train_y = y[train_indices]
test_x = x[test_indices]
test_y = y[test_indices]

kernel = GPy.kern.StdPeriodic(input_dim=1, variance=3.5,
   lengthscale=1.2, period=1)
model = GPy.models.GPRegression(train_x, train_y, kernel)
model.optimize(messages=True)
print(model)

# 6. Make predictions on the test set
mean_prediction, variance = model.predict(test_x)

# 8. Plot the test and training data
plt.figure(figsize=(20, 6))
plt.scatter(train_x, train_y, color='firebrick', marker='x',
   label='train', s=50)
plt.scatter(test_x, test_y, color='steelblue', marker='x',
   label='test', s=50)
plt.plot(test_x, mean_prediction, label='GP Mean Prediction',
   color='mediumpurple', linewidth=3)

# Calculate the 95% confidence interval
lower_bound = mean_prediction - 1.96 * np.sqrt(variance)
upper_bound = mean_prediction + 1.96 * np.sqrt(variance)

# Plot the confidence interval for the test set
plt.fill_between(test_x.flatten(), lower_bound.flatten(),
   upper_bound.flatten(), color='plum', alpha=0.3, label='
```

```
   Credible Interval (95%)')
plt.xlabel('Time (monthly)', fontsize=14)
plt.ylabel('Normalised Temperature Value', fontsize=14)
plt.legend(loc='lower right',ncol=4, fontsize=14)
plt.show()
```

Listing 37: Code of the Creation of Figures 12 and 13

B.2.4 Section 3.5

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Rational Quadratic kernel function
def RQ_kernel(r, alpha, sigma_sq=1.0, lengthscale=1.0):
    return sigma_sq * (1 + (r**2) / (2 * alpha * lengthscale
       **2))**(-alpha)

# Create an array of r values (e.g., from 0 to 5)
r_values = np.linspace(0, 5, 200)  # 200 points between 0 and
   5


# Set varying alpha values for the Rational Quadratic kernel
alphas = [0.5, 1.0, 5.0]  # Different alpha values
colours = ['firebrick', 'steelblue', 'darkorange']  # Colors
   for each line
labels = [r"$\alpha=0.5$", r"$\alpha=1$", r"$\alpha=5$"]

# Plot all variations on a single plot
plt.figure(figsize=(10, 6))

for i, alpha in enumerate(alphas):
    y_values = np.array([RQ_kernel(ri, alpha=alpha) for ri in
       r_values])
    plt.plot(r_values, y_values, label=labels[i], color=
       colours[i], linewidth=3)

# Add labels, title, and legend
plt.xlabel(r"$x-x'$", fontsize=14)
plt.ylabel(r"$k(x,x')$", fontsize=14)
plt.legend(loc="upper right", fontsize=14)
# Show the plot
plt.show()
```

Listing 38: Code of the Creation of Figures 14a

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Rational Quadratic kernel function
def RQ_kernel(x, x_prime, alpha, sigma_sq=1.0, lengthscale
   =1.0):
    r = np.abs(x - x_prime)
    return sigma_sq * (1 + (r**2) / (2 * alpha * lengthscale
       **2))**(-alpha)

# Generate some data points
x = np.linspace(-5, 5, 100)  # 100 points between -5 and 5

# Set kernel parameters for each sample (varying alpha, fixed
   lengthscale and variance)
alpha_values = [0.2, 1.0, 5.0]
colors = ['firebrick', 'steelblue', 'darkorange']
labels = [r"$\alpha = 0.2$", r"$\alpha = 1.0$", r"$\alpha =
   5.0$"]

# Plot the GP prior for each set of parameters
plt.figure(figsize=(10, 6))

for i in range(len(alpha_values)):
    # Compute the covariance matrix using the Rational
       Quadratic kernel
    K = np.array([[RQ_kernel(xi, xj, alpha=alpha_values[i])
       for xj in x] for xi in x])

    # Sample from the GP prior (mean vector = 0, covariance =
       K)
    np.random.seed(10)  # Same seed for reproducibility
    f_prior_samples = np.random.multivariate_normal(mean=np.
       zeros(len(x)), cov=K, size=1)

    # Plot the sample
    plt.plot(x, f_prior_samples[0], label=labels[i], color=
       colors[i], linewidth=3)

# Add labels, title, and legend
plt.xlabel("x", fontsize=14)
plt.ylabel("f(x)", fontsize=14)
```

```
plt.legend(loc="upper left", ncol=3, fontsize=14)
plt.grid(False)

# Show the plot
plt.show()
```

Listing 39: Code of the Creation of Figures 14b

```
import numpy as np
import pandas as pd
import GPy
from matplotlib import pyplot as plt

# Accessing, and giving variable labels.
# access dataset from file
file_path = r'C:\Users\student-local\OneDrive - University of
   Birmingham\Dissertation\USMacroB.csv'
data = pd.read_csv(file_path)

data['time'] = 1959 + data.index / 4
x = data['time'].values.reshape(-1,1)
y = data['tbill'].values.reshape(-1,1)
y = (y - np.mean(y)) / np.std(y)

# 20% of the data for the test set
test_size = int(0.2 * len(data))  # 15% of the data

# Evenly Distributed
# Select indices evenly distributed for the test set
test_indices = np.linspace(0, len(data) - 1, test_size, dtype=
   int)

# The rest of the data will be the training set
train_indices = np.setdiff1d(np.arange(len(data)),
   test_indices)

# Split the data into training and test sets
train_data = data.loc[train_indices]
test_data = data.loc[test_indices]

# Split the dataset into train and test sets
train_x = x[train_indices]
train_y = y[train_indices]
test_x = x[test_indices]
test_y = y[test_indices]
```

```
# Plot the test and training data
plt.figure(figsize=(12, 6))
plt.scatter(train_x, train_y, color='firebrick', marker='x',
   label='Training Data', s=70)
plt.scatter(test_x, test_y, color='steelblue', marker='x',
   label='Test Data', s=70)
plt.xlabel('Time (Quarterly)', fontsize=14)
plt.ylabel('Treasury-bill Rate (Normalised)', fontsize=14)
plt.legend(loc='lower right', fontsize=14)
plt.ylim(-2.5,3.5)
plt.show()


kernel = GPy.kern.RatQuad(input_dim=1, variance=1.4,
   lengthscale=1, power=0.1)
model = GPy.models.GPRegression(train_x, train_y, kernel)
model.likelihood.variance = 0.1
model.optimize(messages=True)
print(model)
# 6. Make predictions on the test set
mean_prediction, variance = model.predict(test_x)

# 8. Plot the test and training data
plt.figure(figsize=(12, 6))
plt.scatter(train_x, train_y, color='firebrick', marker='x',
   label='Training Data', s=70)
plt.scatter(test_x, test_y, color='steelblue', marker='x',
   label='Test Data', s=70)
plt.plot(test_x, mean_prediction, label='GP Mean Prediction',
   color='mediumpurple', linewidth=3)

# Calculate the 95% confidence interval
lower_bound = mean_prediction - 1.96 * np.sqrt(variance)
upper_bound = mean_prediction + 1.96 * np.sqrt(variance)

# Plot the confidence interval for the test set
plt.fill_between(test_x.flatten(), lower_bound.flatten(),
   upper_bound.flatten(), color='plum', alpha=0.3, label='
   Confidence Interval (95%)')
plt.xlabel('Time (Quarterly)', fontsize=14)
plt.ylabel('Treasury-bill Rate (Normalised)', fontsize=14)
plt.legend(loc='lower right', ncol=2, fontsize=14)
plt.ylim(-2.5,3.5)
plt.show()
```

Listing 40: Code of the Creation of Figures 15 and 16

B.2.5 Section 3.6

```
import numpy as np
import matplotlib.pyplot as plt

# Define the Linear kernel function with fixed x' = 1
def linear_kernel(x, x_prime, sigma_sq=1.0, b=0.0):
    return b + sigma_sq * (x) * (x_prime)

# Create arrays of x and x' values (e.g., from -5 to 5)
x_values = np.linspace(-5, 5, 100)  # 100 points between -5
   and 5
x_prime_values = np.linspace(-5, 5, 100)  # Same grid for x'
   values

# Plot Linear Kernel with Varying Bias
bias_values = [-5, 0, 5]  # Different bias values for Plot 1
for b in bias_values:
    # Compute the kernel matrix for varying bias
    K = np.array([[linear_kernel(xi, xj, sigma_sq=1.0, b=b)
       for xj in x_prime_values] for xi in x_values])

    # Create contour plot
    plt.figure(figsize=(7, 5))
    cp = plt.contourf(x_values, x_prime_values, K, levels=20,
       cmap='coolwarm')
    plt.colorbar(cp)
    plt.xlabel(r'$x$', fontsize=14)
    plt.ylabel(r"$x'$", fontsize=14)
    plt.show()
```

Listing 41: Code of the Creation of Figures 17

```
import numpy as np
import pandas as pd
import GPy
from matplotlib import pyplot as plt

# Accessing, and giving variable labels.
# access dataset from file
file_path = r'C:\Users\student-local\OneDrive - University of
   Birmingham\Dissertation\EuroEnergy.csv'
data = pd.read_csv(file_path)

# Extracting the target variable (gdp) and input variable (
```

```
    energy)
x = data['energy'].values.reshape(-1, 1)  # reshape to 2D
    array for GPy
y = data['gdp'].values.reshape(-1, 1)     # reshape to 2D
    array for GPy

# Standardising x and y
y = (y - np.mean(y)) / np.std(y)
x = (x - np.mean(x)) / np.std(x)

# Sort test_x in increasing order
sorted_indices = np.argsort(x.flatten())

# Reorder test_x and test_y according to the sorted indices
sorted_x = x[sorted_indices]
sorted_y = y[sorted_indices]

# 10% of the data for the test set
test_size = int(0.2 * len(data))
test_indices = np.linspace(0, len(data) - 1, test_size, dtype=
    int)
train_indices = np.setdiff1d(np.arange(len(data)),
    test_indices)

# Split the dataset into train and test sets
train_x = sorted_x[train_indices]
train_y = sorted_y[train_indices]
test_x = sorted_x[test_indices]
test_y = sorted_y[test_indices]


kernel = GPy.kern.Linear(input_dim=1, variances=1)
model = GPy.models.GPRegression(train_x, train_y, kernel)
model.optimize(messages=True)
print(model)

# 6. Make predictions on the test set
mean_prediction, variance = model.predict(test_x)

# 8. Plot the test and training data
plt.figure(figsize=(10, 6))
plt.scatter(train_x, train_y, color='firebrick', marker='x',
    label='training set', s=50)
plt.scatter(test_x, test_y, color='steelblue', marker='x',
    label='test set', s=50)
plt.plot(test_x, mean_prediction, label='GP Mean Prediction',
```

```
   color='mediumpurple', linewidth=3)

# Calculate the 95% confidence interval
lower_bound = mean_prediction - 1.96 * np.sqrt(variance)
upper_bound = mean_prediction + 1.96 * np.sqrt(variance)

# Plot the confidence interval for the test set
plt.fill_between(test_x.flatten(), lower_bound.flatten(),
   upper_bound.flatten(), color='plum', alpha=0.3, label='
   Credible Interval (95%)')
plt.xlabel('energy (normalised)', fontsize=14)
plt.ylabel('gdp (normalised)', fontsize=14)
plt.legend(loc='lower right',ncol=2, fontsize=14)
plt.show()
```

Listing 42: Code of the Creation of Figures 18

B.3 Chapter 4

```
import numpy as np
import matplotlib.pyplot as plt

# Define the spectral densities for the different kernels

# 1. Squared Exponential (SE) Spectral Density
def se_spectral_density(s, l):
    return l * np.sqrt(2 * np.pi) * np.exp(-2 * np.pi**2 * s
       **2 * l**2)

# 2. Matérn (3/2) Spectral Density
def matern_3_2_spectral_density(s, l):
    return (4 * np.sqrt(3) / l**3) * ( (3 / l**2) + (4 * np.pi
       **2 * s**2) )**(-2)

# 3. Matérn (5/2) Spectral Density
def matern_5_2_spectral_density(s, l):
    return (16 * np.sqrt(5) / (3 * l**5)) * ( (5 / l**2) + (4
       * np.pi**2 * s**2) )**(-3)

# 4. Exponential Spectral Density
def exponential_spectral_density(s, l):
    return 1 / ( (np.pi / l) + (np.pi * l * s**2) )

# Create the frequency range for plotting
```

```
s_values = np.linspace(0, 1, 1000)  # frequency range from 0
    to 10

# Lengthscale parameter
l = 1 # Can be adjusted

# Compute the spectral densities for each kernel
se_values = se_spectral_density(s_values, l)
matern_3_2_values = matern_3_2_spectral_density(s_values, l)
matern_5_2_values = matern_5_2_spectral_density(s_values, l)
exponential_values = exponential_spectral_density(s_values, l)

# Plot the spectral densities
plt.figure(figsize=(10, 6))

plt.plot(s_values, se_values, label='Squared Exponential (SE)'
    , color='steelblue', linewidth=3)
plt.plot(s_values, matern_3_2_values, label='Matern (3/2)',
    color='darkorange', linewidth=3)
plt.plot(s_values, matern_5_2_values, label='Matern (5/2)',
    color='firebrick', linewidth=3)
plt.plot(s_values, exponential_values, label='Exponential',
    color='mediumpurple', linewidth=3)

# Add labels
plt.xlabel('Frequency $s$', fontsize=14)
plt.ylabel('Spectral Density $S(s)$', fontsize=14)
plt.ylim(0, 1)
plt.legend(fontsize=14)
plt.show()
```

Listing 43: Code of the Creation of Figures 19

B.4 Chapter 5

B.4.1 Section 5.1

```
import numpy as np
import matplotlib.pyplot as plt

# Define the quadratic (polynomial) kernel function
def polynomial_kernel(x, x_prime, p):
    return (x * x_prime)**p
```

```
# Generate a range of x values
x_values = np.linspace(-5, 5, 100)  # Varying x from -5 to 5
x_prime = 1  # Fixed value for x'
p = 2  # Degree of the polynomial (quadratic)

# Compute the kernel (covariance) values for each x
y_values = polynomial_kernel(x_values, x_prime, p)

# Create the plot
plt.plot(x_values, y_values, label=f'Polynomial Kernel (p={p})
    ', color='steelblue', linewidth=3)
plt.xlabel('$x$ $(x\' = 1)$', fontsize=14)
plt.ylabel('$k(x, x\')$', fontsize=14)
plt.show()

# Generate a grid of x and x' values
x_values = np.linspace(-5, 5, 100)  # x from -5 to 5
x_prime_values = np.linspace(-5, 5, 100)  # x' from -5 to 5

# Create meshgrid for the contour plot
X, X_prime = np.meshgrid(x_values, x_prime_values)

# Compute the kernel values (covariance) for each (x, x') pair
Y = polynomial_kernel(X, X_prime, p=2)  # Quadratic kernel (p
    =2)

# Create the contour plot
plt.figure(figsize=(8, 6))
cp = plt.contourf(X, X_prime, Y, cmap='RdBu')
plt.colorbar(cp)  # Show color scale
plt.xlabel('$x$', fontsize=14)
plt.ylabel("$x'$", fontsize=14)
plt.show()

# Simulate a Gaussian Process with the quadratic kernel

# Define the mean (which is 0 for this case)
mean = np.zeros(len(x_values))

# Compute the covariance matrix using the quadratic kernel
K = polynomial_kernel(x_values[:, None], x_values[None, :], p
    =2)

# Add some noise (optional, for realistic simulation)
noise = 1e-5 * np.eye(len(x_values))  # Small noise to make
    the GP slightly more realistic
```

```
K += noise

# Sample functions from the GP
samples = np.random.multivariate_normal(mean, K, size=5)

# Plot the samples
plt.figure(figsize=(10, 6))
for i in range(5):
    plt.plot(x_values, samples[i, :], label=f'Sample {i+1}',
       linewidth=3)
plt.xlabel('$x$', fontsize=14)
plt.ylabel('$f(x)$', fontsize=14)
plt.legend()
plt.show()
```

Listing 44: Code of the Creation of Figures 20

B.4.2 Section 5.3

```
import numpy as np
import pandas as pd
import GPy
from matplotlib import pyplot as plt

# Accessing, and giving variable labels.
# access dataset from file
file_path = r'C:\Users\student-local\OneDrive - University of
   Birmingham\Dissertation\OrangeCounty.csv'
data = pd.read_csv(file_path)


# Extracting the target variable (gnp) and input variables (
   all columns except 'gnp')
y = data['gnp'].values.reshape(-1,1)
data['time'] = 1965 + data.index / 4
x = data['time'].values.reshape(-1,1)


# Plot the 'employment' data against the quarterly timestamps
plt.figure(figsize=(10, 6))
plt.scatter(x, y, color='steelblue', marker='x', label='GNP',
   s=50)
# Labeling the axes
plt.xlabel('Time (Quarterly)', fontsize=14)
```

```
plt.ylabel('Real GNP', fontsize=14)
plt.show()

y = (y - np.mean(y)) / np.std(y)

# Randomly select 15% of the data for the test set
test_size = int(0.2 * len(data))  # 15% of the data

# 20% of the data for the test set
test_size = int(0.2 * len(data))  # 15% of the data

# Evenly Distributed
# Select indices evenly distributed for the test set
test_indices = np.linspace(0, len(data) - 1, test_size, dtype=
   int)

# The rest of the data will be the training set
train_indices = np.setdiff1d(np.arange(len(data)),
   test_indices)

# Split the data into training and test sets
train_data = data.loc[train_indices]
test_data = data.loc[test_indices]

# Split the dataset into train and test sets
train_x = x[train_indices]
train_y = y[train_indices]
test_x = x[test_indices]
test_y = y[test_indices]

# Plot the 'quantity' time series for the training set
plt.figure(figsize=(10, 6))
plt.scatter(train_x, train_y, label='Train', color='steelblue'
   , marker='x', s=50)
plt.scatter(test_x, test_y, label='Test', color='firebrick',
   marker='x', s=50)
plt.xlabel('Year (Quarterly)', fontsize=14)
plt.ylabel('GNP', fontsize=14)
plt.show()

# 5. Fit the Gaussian Process model
kernel_1 =GPy.kern.Linear(input_dim=1, variances=1)
kernel_2 = GPy.kern.RBF(input_dim=1, variance=1.0, lengthscale
   =1.0)
kernels = [kernel_1,kernel_2]
kernel = GPy.kern.Add(kernels)
```

```
model = GPy.models.GPRegression(train_x, train_y, kernel)
model.optimize(messages=True)
print(model)
# 6. Make predictions on the test set
mean_prediction, variance = model.predict(test_x)

plt.figure(figsize=(8, 6))
plt.scatter(train_x, train_y, label='Training set', color='
   steelblue', marker='x', s=50)
plt.scatter(test_x, test_y, label='Testing set', color='
   firebrick', marker='x', s=50)

plt.plot(test_x, mean_prediction, label='GP Mean Prediction',
   color='mediumpurple', linewidth=3)

# Calculate the 95% confidence interval
lower_bound = mean_prediction - 1.96 * np.sqrt(variance)
upper_bound = mean_prediction + 1.96 * np.sqrt(variance)

# Plot the confidence interval for the test set
plt.fill_between(test_x.flatten(), lower_bound.flatten(),
   upper_bound.flatten(), color='plum', alpha=0.3, label='
   Credible Interval (95%)')
plt.xlabel('Year (Quarterly)', fontsize=14)
plt.ylabel('GNP', fontsize=14)
plt.legend(loc='lower right', fontsize=14)
plt.show()

kernel = GPy.kern.Prod(kernels)
model = GPy.models.GPRegression(train_x, train_y, kernel)
model.optimize(messages=True)
print(model)
# 6. Make predictions on the test set
mean_prediction, variance = model.predict(test_x)

plt.figure(figsize=(8, 6))
plt.scatter(train_x, train_y, label='Training set', color='
   steelblue', marker='x', s=50)
plt.scatter(test_x, test_y, label='Testing set', color='
   firebrick', marker='x', s=50)

plt.plot(test_x, mean_prediction, label='GP Mean Prediction',
   color='mediumpurple', linewidth=3)

# Calculate the 95% confidence interval
```

```
lower_bound = mean_prediction - 1.96 * np.sqrt(variance)
upper_bound = mean_prediction + 1.96 * np.sqrt(variance)

# Plot the confidence interval for the test set
plt.fill_between(test_x.flatten(), lower_bound.flatten(),
   upper_bound.flatten(), color='plum', alpha=0.3, label='
   Credible Interval (95%)')
plt.xlabel('Year (Quarterly)', fontsize=14)
plt.ylabel('GNP', fontsize=14)
plt.legend(loc='lower right', fontsize=14)
plt.show()
```

Listing 45: Code of the Creation of Figures 21,23, 24

```
import numpy as np
import matplotlib.pyplot as plt

# Define the SE (Squared Exponential) kernel
def se_kernel(x, x_prime, variance=1.0, lengthscale=1.0):
    return variance * np.exp(-0.5 * (x - x_prime)**2 /
       lengthscale**2)

# Define the Linear kernel
def linear_kernel(x, x_prime, variance=1.0):
    return variance * x * x_prime

# Create a grid of x and x' values (input space)
x_values = np.linspace(-5, 5, 100)
X, X_prime = np.meshgrid(x_values, x_values)

# Compute the kernel values for the product kernel (SE *
   Linear)
K_product = se_kernel(X, X_prime) * linear_kernel(X, X_prime)

# Plot the contour for the product kernel (SE * Linear)
plt.figure(figsize=(7, 5))
cp1 = plt.contourf(X, X_prime, K_product, cmap='RdBu')
plt.colorbar(cp1, label="Covariance")
plt.xlabel('$x$', fontsize=14)
plt.ylabel("$x'$", fontsize=14)
plt.show()

# Compute the kernel values for the sum kernel (SE + Linear)
K_sum = se_kernel(X, X_prime) + linear_kernel(X, X_prime)

# Plot the contour for the sum kernel (SE + Linear)
```

```
plt.figure(figsize=(7, 5))
cp2 = plt.contourf(X, X_prime, K_sum, cmap='RdBu')
plt.colorbar(cp2, label="Covariance")
plt.xlabel('$x$', fontsize=14)
plt.ylabel("$x'$", fontsize=14)
plt.show()
```

Listing 46: Code of the Creation of Figure 22

B.4.3 Section 5.4

```
import numpy as np
import pandas as pd
import GPy
from matplotlib import pyplot as plt
from sklearn.preprocessing import StandardScaler

# Accessing, and giving variable labels.
# Access dataset from file
file_path = r'C:\Users\student-local\OneDrive - University of
   Birmingham\Dissertation\barro.csv'
data = pd.read_csv(file_path)

# Only consider the first 71 rows
data = data.iloc[:71]

# Target variable y creation
y = data['y.net'].values.reshape(-1,1)
y = (y - y.mean()) / y.std()
X = data.iloc[:, 2:15]
scaler = StandardScaler()
X = scaler.fit_transform(X)

# 20% of the data for the test set
test_size = int(0.2 * len(data))  # 15% of the data

# Evenly Distributed
# Select indices evenly distributed for the test set
test_indices = np.linspace(0, len(data) - 1, test_size, dtype=
   int)
# The rest of the data will be the training set
train_indices = np.setdiff1d(np.arange(len(data)),
   test_indices)
```

```
train_x = X[train_indices]  # Training x values
train_y = y[train_indices]  # Training y values

test_x = X[test_indices]  # Test x values
test_y = y[test_indices]  # Test y values


# Plot the training and test points with respect to 'y.net'
plt.figure(figsize=(10, 6))
plt.scatter(train_indices, train_y,marker='x', color='
   steelblue', label='Training Data', s=50)
plt.scatter(test_indices, test_y, marker='x', color='firebrick
   ', label='Test Data', s=50)
plt.xlabel('Samples (Country Index)', fontsize=14)
plt.ylabel('Standardised GDP Growth rates', fontsize=14)
plt.legend(fontsize=14)
plt.show

# Create the SE-ARD kernel (Set lengthscales to 6 for all
   dimensions)
kernel = GPy.kern.RBF(input_dim=train_x.shape[1], variance
   =1.0, lengthscale=np.full(train_x.shape[1], 6), ARD=True)

# Create the GP Regression model
model = GPy.models.GPRegression(train_x, train_y.reshape(-1,
   1), kernel)

# Optimize the model
model.optimize(messages=True)

# Print the optimized model and hyperparameters (including
   lengthscales)
print(model)
print("\nLengthscales of the SE-ARD kernel:")
print(model.kern.lengthscale)

# Plot the posterior covariance matrix
posterior_covariance = model.kern.K(test_x)
plt.figure(figsize=(8, 6))
plt.imshow(posterior_covariance, cmap='coolwarm',
   interpolation='nearest')
plt.colorbar(label="Covariance")
plt.xlabel("Samples", fontsize=14)
plt.ylabel("Samples", fontsize=14)
plt.show()
```

```
# Predictive Mean and Confidence Intervals for the Test Set
mean, variance = model.predict(test_x)

# 95% Confidence Intervals (1.96 * std deviation)
confidence_interval = 1.96 * np.sqrt(variance)


plt.figure(figsize=(10, 6))


plt.scatter(train_indices, train_y,marker='x', color='
   steelblue', label='Training Data', s=50)
plt.scatter(test_indices, test_y, marker='x', color='firebrick
   ', label='Test Data', s=50)
# Plot the predictive mean
plt.plot(test_indices, mean, color='mediumpurple', label='
   Predictive Mean', linewidth=3)

# Plot the confidence intervals
plt.fill_between(test_indices, mean.flatten() -
   confidence_interval.flatten(), mean.flatten() +
   confidence_interval.flatten(),
                color='plum', alpha=0.3, label='95% Credible
                   Interval')

# Add labels and title
plt.xlabel('Samples (Country Index)', fontsize=14)
plt.ylabel('Standardized GDP Growth rates', fontsize=14)
plt.legend(loc='upper left', ncol=2, fontsize=14)
plt.show()
```

Listing 47: Code of the Creation of Figures25,26, 27

B.4.4 Section 5.5

```
import pandas as pd
import random
import GPy
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder,
   StandardScaler
from sklearn.preprocessing import StandardScaler
```

```
# Load the dataset from the URL
file_path = r'C:\Users\student-local\OneDrive - University of
   Birmingham\Dissertation\HousePrices.csv'
data = pd.read_csv(file_path)
data = data[['rownames','price','stories']]
data = data.sort_values('stories')
print(data)

# Step 2: One-Hot Encoding of the 'x' values
encoder = OneHotEncoder(sparse_output=False)  # Use
   sparse_output=False for dense array
encoded_stories = encoder.fit_transform(data[['stories']])

# Convert the encoded data to a DataFrame for better
   visualization
encoded_data = pd.DataFrame(encoded_stories, data['rownames'],
    columns=encoder.categories_[0])
encoded_inputs = pd.DataFrame(encoded_stories, columns=encoder
   .categories_[0])


y = data['price'].values.reshape(-1,1)
# Standardize target_y (mean = 0, std = 1)
y = (y - y.mean()) / y.std()

# Input matrix variable x creation (features)
X = encoded_data.iloc[:, 0:5]  # Assuming columns 3 to 14 are
   the input features
scaler = StandardScaler()
X = scaler.fit_transform(X)


# Randomly select 15% of the data for the test set
test_size = int(0.2 * len(data))  # 15% of the data

# Evenly Distributed
# Select indices evenly distributed for the test set
test_indices = np.linspace(0, len(data) - 1, test_size, dtype=
   int)
#print(test_indices)
# The rest of the data will be the training set
train_indices = np.setdiff1d(np.arange(len(data)),
   test_indices)

train_x = X[train_indices]  # Training x values
train_y = y[train_indices]  # Training y values
```

```
test_x = X[test_indices]  # Test x values
test_y = y[test_indices]  # Test y values

# Step 5: Plot the training and test points with respect to 'y
    .net'
plt.figure(figsize=(10, 6))

# Plot the training data
plt.scatter(train_indices, train_y,marker='x', color='
    steelblue', label='Training Data', s=50)
# Plot the test data
plt.scatter(test_indices, test_y, marker='x', color='firebrick
    ', label='Test Data', s=50)
# Add labels and title
plt.xlabel('Samples', fontsize=14)
plt.ylabel('Standardised House Prices', fontsize=14)

# Add a legend
plt.legend(fontsize=14)
plt.show

# Step 1: Create the SE-ARD kernel (Set lengthscales to 1 for
    all dimensions)
kernel = GPy.kern.RBF(input_dim=train_x.shape[1], variance
    =1.0, lengthscale=np.full(train_x.shape[1], 1), ARD=True)

# Step 2: Create the GP Regression model
model = GPy.models.GPRegression(train_x, train_y.reshape(-1,
    1), kernel)

# Step 3: Optimize the model
model.optimize(messages=True)

# Print the optimized model and hyperparameters (including
    lengthscales)
print(model)
print("\nLengthscales of the SE-ARD kernel:")
print(model.kern.lengthscale)


# Step 4: Predictive Mean and Confidence Intervals for the
    Test Set
mean, variance = model.predict(test_x)

# 95% Confidence Intervals (1.96 * std deviation)
```

```
credible_interval = 1.96 * np.sqrt(variance)

plt.figure(figsize=(10, 6))
plt.scatter(train_indices, train_y,marker='x', color='
    steelblue', label='Training Data', s=50)

# Plot the test data
plt.scatter(test_indices, test_y, marker='x', color='firebrick
    ', label='Test Data', s=50)
# Plot the predictive mean
plt.plot(test_indices, mean, color='mediumpurple', label='
    Predictive Mean', linewidth=3)

# Plot the confidence intervals
plt.fill_between(test_indices, mean.flatten() -
    credible_interval.flatten(), mean.flatten() +
    credible_interval.flatten(),
                 color='plum', alpha=0.3, label='95% Credible
                    Interval')

# Add labels and legend
plt.xlabel('Samples', fontsize=14)
plt.ylabel('Standardised House Prices', fontsize=14)
plt.legend(fontsize=14)
plt.show()

# Step 4: Plot the posterior covariance matrix
posterior_covariance = model.kern.K(test_x)
plt.figure(figsize=(8, 6))
plt.imshow(posterior_covariance, cmap='coolwarm',
    interpolation='nearest')
plt.colorbar(label="Covariance")
plt.xlabel("Samples")
plt.ylabel("Samples")
plt.show()
```

Listing 48: Code of the Creation of Figure 28

B.5 Chapter 6

```
import numpy as np
import pandas as pd
import GPy
from matplotlib import pyplot as plt
```

```
# Accessing, and giving variable labels.
# access dataset from file
file_path = r'C:\Users\student-local\OneDrive - University of
    Birmingham\Dissertation\USMacroB.csv'
data = pd.read_csv(file_path)

data['time'] = 1959 + data.index / 4
x = data['time'].values.reshape(-1,1)
y = data['tbill'].values.reshape(-1,1)
y = (y - np.mean(y)) / np.std(y)

# 20% of the data for the test set
test_size = int(0.1 * len(data))  # 15% of the data

# Evenly Distributed
# Select indices evenly distributed for the test set
test_indices = np.linspace(0, len(data) - 1, test_size, dtype=
    int)

# The rest of the data will be the training set
train_indices = np.setdiff1d(np.arange(len(data)),
    test_indices)

# Split the data into training and test sets
train_data = data.loc[train_indices]
test_data = data.loc[test_indices]

# Split the dataset into train and test sets
train_x = x[train_indices]
train_y = y[train_indices]
test_x = x[test_indices]
test_y = y[test_indices]

# 5. Create a list of different lengthscales to test
lengthscales = np.linspace(0.1, 0.5, 100)  # Lengthscale range

# List to store marginal likelihood values
complexitys = []
norm_consts = []
data_fits = []
lmls = []

# 6. Loop over each lengthscale and calculate the marginal
    likelihood
for lengthscale in lengthscales:
```

```
    # Create an SE kernel with the given lengthscale
    kernel = GPy.kern.RBF(input_dim=1, variance=1, lengthscale
       =lengthscale)

    # Fit the GP model using the training data
    model = GPy.models.GPRegression(train_x, train_y, kernel)

    data_fit = np.dot(train_y.T, np.dot(np.linalg.inv(model.
       kern.K(train_x)), train_y))
    data_fit =  -1/2 * data_fit.item()
    data_fits.append(data_fit)

    complexity = - 1/2 * np.log(np.linalg.det(model.kern.K(
       train_x)))
    complexitys.append(complexity)

    norm_const = - len(y)/2 *np.log(2*np.pi)
    norm_consts.append(norm_const)

    lml = data_fit + complexity + norm_const
    lmls.append(lml)



# 7. Plot the marginal likelihood as a function of lengthscale
plt.figure(figsize=(8, 6))
plt.plot(lengthscales, data_fits, label='Data fit', color='
   steelblue', linewidth=3)
plt.plot(lengthscales, complexitys, label='Complexity', color=
   'firebrick', linewidth=3)
plt.plot(lengthscales, norm_consts, label='Normalising
   Constant', color='darkorange', linewidth=3)
plt.plot(lengthscales, lmls, label='log marginal likelihood',
   color='mediumpurple', linewidth=3)
plt.ylim(-400, 150)
plt.xlabel('Characteristic Length-scale')
plt.ylabel('Log Probability')
plt.grid(False)
plt.legend(fontsize=14)
plt.show()

# 3 test sizes
test_sizes = [0.1, 0.2, 0.3]

# Lengthscale range for the kernel
lengthscales = np.linspace(0.1, 0.6, 100)  # Lengthscale range
```

```
# List to store log likelihood values for different test sizes
lmls_list = []

for test_size in test_sizes:
    # Split the dataset into train and test sets based on the
       current test_size
    test_size_int = int(test_size * len(data))

    # Evenly distribute test indices across the data
    test_indices = np.linspace(0, len(data) - 1, test_size_int
       , dtype=int)
    train_indices = np.setdiff1d(np.arange(len(data)),
       test_indices)

    # Create training and test sets
    train_x = x[train_indices]
    train_y = y[train_indices]
    test_x = x[test_indices]
    test_y = y[test_indices]

    lmls = []

    # Loop over each lengthscale and calculate the log
       marginal likelihood
    for lengthscale in lengthscales:
        kernel = GPy.kern.RBF(input_dim=1, variance=1,
           lengthscale=lengthscale)
        model = GPy.models.GPRegression(train_x, train_y,
           kernel)

        data_fit = np.dot(train_y.T, np.dot(np.linalg.inv(
           model.kern.K(train_x)), train_y))
        data_fit =  -1/2 * data_fit.item()
        data_fits.append(data_fit)

        complexity = - 1/2 * np.log(np.linalg.det(model.kern.K
           (train_x)))
        complexitys.append(complexity)

        norm_const = - len(y)/2 *np.log(2*np.pi)
        norm_consts.append(norm_const)
        # Get the marginal likelihood of the model
        lml = data_fit + complexity + norm_const
        lmls.append(lml)
    lmls_list.append(lmls)
```

```
# Plot logmlikelis for different test sizes
colour=['steelblue','firebrick','darkorange']
plt.figure(figsize=(8, 6))
for i, test_size in enumerate(test_sizes):
    plt.plot(lengthscales, lmls_list[i], label=f'Test Size
       Ratio = {test_size}',color=colour[i], linewidth=3)

plt.xlabel('Characteristic Length-scale')
plt.ylabel('Log Marginal Likelihood')
plt.legend(fontsize=14)
plt.ylim(-400, 100)
plt.grid(False)
plt.show()
```

Listing 49: Code of the Creation of Figure 29

```
import numpy as np
import pandas as pd
import GPy
from matplotlib import pyplot as plt

# Accessing, and giving variable labels.
# access dataset from file
file_path = r'C:\Users\student-local\OneDrive - University of
   Birmingham\Dissertation\OrangeCounty.csv'
data = pd.read_csv(file_path)


print(data)
# Extracting the target variable (gnp) and input variables (
   all columns except 'gnp')
y = data['gnp'].values.reshape(-1,1)
x = data.index.values.reshape(-1, 1)
y = (y - np.mean(y)) / np.std(y)

# 20% of the data for the test set
test_size = int(0.1 * len(data))  # 15% of the data

# Evenly Distributed
# Select indices evenly distributed for the test set
test_indices = np.linspace(0, len(data) - 1, test_size, dtype=
   int)

# The rest of the data will be the training set
```

```
train_indices = np.setdiff1d(np.arange(len(data)),
    test_indices)

# Split the data into training and test sets
train_data = data.loc[train_indices]
test_data = data.loc[test_indices]

# Split the dataset into train and test sets
train_x = x[train_indices]
train_y = y[train_indices]
test_x = x[test_indices]
test_y = y[test_indices]

kernel = GPy.kern.RBF(input_dim=1, variance=1.0, lengthscale
    =1.0)
model = GPy.models.GPRegression(train_x, train_y, kernel)
model.optimize()
print(model)
print(f'The value of maximised log maringal likelihood is: {
    model.log_likelihood()}')

# Create a grid of noise values and lengthscale values
noise_values = np.linspace(0.001, 0.003, 30)  # Noise values
lengthscales = np.linspace(5, 7, 30)  # Lengthscale values

# Prepare the meshgrid
X_noise, X_lengthscale = np.meshgrid(noise_values,
    lengthscales)

# Prepare a matrix to store the marginal likelihood values
lmls = np.zeros_like(X_noise)

# Loop over the noise and lengthscale values and compute the
    marginal likelihood
for i in range(len(noise_values)):
    for j in range(len(lengthscales)):
        noise = noise_values[i]
        lengthscale = lengthscales[j]
        # Create an SE kernel with the given noise and
            lengthscale
        kernel = GPy.kern.RBF(input_dim=1, variance=1,
            lengthscale=lengthscale)
        # Fit the GP model using the training data
        model = GPy.models.GPRegression(train_x, train_y,
            kernel, noise_var=noise)
        lml = model.log_likelihood()
```

```
        lmls[j, i] = lml


# Find the index of the maximum log marginal likelihood
max_idx = np.unravel_index(np.argmax(lmls), lmls.shape)
max_lengthscale = X_lengthscale[max_idx]
max_noise = X_noise[max_idx]
print(np.max(lmls))
print(max_lengthscale)
print(max_noise)

# Create the contour plot
plt.figure(figsize=(8, 6))
contour = plt.contourf(X_lengthscale, X_noise, lmls, 20, cmap=
    'coolwarm')
plt.colorbar(contour)
plt.ylabel('Noise Variance')
plt.xlabel('Lengthscale')
# Add a white cross at the maximum point
plt.plot(max_lengthscale, max_noise, 'wx', markersize=10,
    label='Max LML (graphical method)')
plt.plot(6.067, 0.00189, 'kx', markersize=10, label='Max LML (
    optimise method)')
plt.grid(False)
plt.legend()
plt.show
```

Listing 50: Code of the Creation of Figure 30

C Appendix: Datasets

The reader is directed to available R datasets, a website with a list of over 2000 datasets that are free to use. CSV files and brief explanations are provided for each dataset. The names of the dataset that are mentioned in the paper can be entered into the search engine and the dataset can be downloaded by clicking CSV.

References

Abramowitz, M. and Stegun, I. A. (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, ninth dover printing, tenth gpo printing edition.

Abu-Mostafa, Y. S., Magdon-Ismail, M., and Lin, H.-T. (2012). *Learning from data*, volume 4. AMLBook New York.

Adler, R. J. (2010). *The geometry of random fields*. SIAM.

Adler, R. J. and Taylor, J. E. (2009). *Random fields and geometry*. Springer Science & Business Media.

Anderson, O. D. (1976). *Time series analysis and forecasting: the Box-Jenkins approach*. Butterworths.

Anglin, P. M. and Gencay, R. (1996). Semiparametric estimation of a hedonic price function. *Journal of applied econometrics*, 11(6):633–648.

Baltagi, B. H. (2008). *Econometric analysis of panel data*, volume 4. Springer.

Barro, R. and Sala-i Martin, X. (2004). Economic growth second edition.

Bracewell, R. N. (1966). The fourier transform and its applications.

Calandra, R., Peters, J., Rasmussen, C. E., and Deisenroth, M. P. (2016). Manifold gaussian processes for regression. In *2016 International joint conference on neural networks (IJCNN)*, pages 3338–3345. IEEE.

Chatfield, C. and Xing, H. (2019). *The analysis of time series: an introduction with R*. Chapman and hall/CRC.

Domingo, D. (2019). Gaussian process emulation: Theory and applications to the problem of past climate reconstruction.

Duvenaud, D. (2014). *Automatic model construction with Gaussian processes*. PhD thesis.

Duvenaud, D. (2017). Kernel Cookbook — cs.toronto.edu. `https://www.cs.toronto.edu/~duvenaud/cookbook/`. [Accessed 16-12-2024].

Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. B. (1995). *Bayesian data analysis*. Chapman and Hall/CRC.

Gihman, I. I. and Skorokhod, A. V. (2004). *The Theory of Stochastic Processes I*. Springer Berlin Heidelberg.

Heinonen, M. (2017). Learning with spectral kernels. Accessed: 2025-01-19.

Hoff, P. D. (2009). *A first course in Bayesian statistical methods*, volume 580. Springer.

Kac, M. (1943). On the average number of real roots of a random algebraic equation. *Bulletin of the American Mathematical Society*, 49(4):314–320.

Koenker, R. and Machado, J. A. (1999). Goodness of fit and related inference processes for quantile regression. *Journal of the american statistical association*, 94(448):1296–1310.

Kuhn, M. (2013). Applied predictive modeling.

Little, R. J. (2006). Calibrated bayes: a bayes/frequentist roadmap. *The American Statistician*, 60(3):213–223.

MacKay, D. J. et al. (1998). Introduction to gaussian processes. *NATO ASI series F computer and systems sciences*, 168:133–166.

Papoulis, A. and Pillai, S. U. (2002). *Probability, Random Variables, and Stochastic Processes*. McGraw Hill, Boston, fourth edition.

Porter, R. H. (1983). A study of cartel stability: The joint executive committee, 1880-1886. *The Bell Journal of Economics*, 14(2):301.

Radford, M. and Neal, N. (1996). *Bayesian learning for neural networks*. Springer.

Rice, S. O. (1945). Mathematical analysis of random noise. *Bell System Technical Journal*, 24(1):46–156.

Santner, T. J., Williams, B. J., and Notz, W. I. (2003). *The Design and Analysis of Computer Experiments*. Springer New York.

Seymour, R. (2023). Bayesian inference and computation. https://rowlandseymour.github.io/BIC/. Accessed: 2024-12-16.

Seymour, R. G., Kypraios, T., and O'Neill, P. D. (2022). Bayesian nonparametric inference for heterogeneously mixing infectious disease models. *Proceedings of the National Academy of Sciences*, 119(10):e2118425119.

Snelson, E. (2006). Tutorial: Gaussian process models for machine learning. `http://www.gatsby.ucl.ac.uk/~/snelson/Tutorial/`. Gatsby Computational Neuroscience Unit, UCL, 26th October 2006.

Snelson, E., Ghahramani, Z., and Rasmussen, C. (2003). Warped gaussian processes. *Advances in neural information processing systems*, 16.

Stein, M. (1999). Interpolation of spatial data. springer series in statistics.

Uhlenbeck, G. E. and Ornstein, L. S. (1930). On the theory of the brownian motion. *Physical Review*, 36(5):823–841.

Von Mises, R. (1967). Mathematical theory of probability and statistics.

Williams, C. K. and Rasmussen, C. E. (2006). *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA.

Yeh, I.-C. (2006). Analysis of strength of concrete using design of experiments and neural networks. *Journal of Materials in Civil Engineering*, 18(4):597–604.