
电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

学士学位论文

BACHELOR THESIS



Title: Gaussian Process Prediction of
Stock Price Trends

Major: Software Engineering

Student ID: 2017221501010

Name: Etim Ubongabasi Ebong

Supervisor: Dr. Weidong Wang

摘 要

高斯过程回归（GPR）是一种重要的机器学习方法，它利用了贝叶斯理论和推理以及统计研究理论。它能够为我们提供一个相当广泛的结构，用于执行概率回归，也广泛用于为高维、小样本或不一致（非线性）回归问题提供解决方案。它的基本概念可以在功能空间视图或权空间视图中建立。高斯过程回归就其实现方法而言很简单，可总结为非参数推理以及与神经网络和支持向量机并置（比较或对比）的超参数调整。高斯过程模型为其预测提供高斯不确定性估计的显着特征使它们能够顺利地应用于预测控制、自适应控制和贝叶斯过滤技术。最后，本论文给出了实现并给出了未来的研究趋势。

关键词：高斯过程，高斯过程回归，模型拟合，股价预测

ABSTRACT

Gaussian process regression (GPR) is an important machine learning method that makes uses of Bayesian theory and inference as well as statistical studying theory. It is able to provide us with a rather broad structure for the purpose of performing probabilistic regression and is also broadly used for the purpose of proffering solutions to the high-dimensional, small-sample or inconsistent (nonlinear) regression problems. Its fundamental concept is established in either the function-space view or weight-space view. GPR is straightforward with regards to its implementation methods, moldable to nonparametric inference and self-adjusting to often explicitly quantified hyperparameters in juxtaposition (comparison or contrast) with neural network and support vector machines. The striking characteristic that GPR models give Gaussian uncertainty estimates for their forecasts allows them to be smoothly absorbed into prognostic control, adaptive control and Bayesian filtering techniques. Lastly, its implementations are given and future research trends are surveyed.

Keywords: Gaussian process, Gaussian process regression, Model fitting, Stock price prediction

CONTENTS

CHAPTER 1 Introduction.....	1
1.1. Project aims and contributions.....	1
1.2. Thesis structure.....	1
1.3. Gaussian process definition.....	2
1.4. Gaussian process.....	2
CHAPTER 2 Gaussian Process Regression Model.....	5
2.1. Gaussian process regression.....	5
2.1.1. Mean and covariance functions.....	5
2.2. GP prior.....	6
2.3. Hyperparameters.....	6
2.4. The marginal likelihood.....	7
CHAPTER 3 System Design.....	9
3.1. System development life cycle description.....	9
3.2. Project decomposition.....	10
3.3. Algorithm design.....	11
3.3.1. Dataset preparation.....	11
CHAPTER 4 Implementation.....	16
4.1. Dataset.....	16
4.1.1. Dataset preparation.....	16
4.1.2. Dataset preprocess.....	16
4.2. Hyperparameters learning.....	20
4.3. Training (Model fitting).....	20
4.3. Testing (Make predictions).....	23
CHAPTER 5 Conclusions and Observations.....	25
Appendix. Entire Algorithm Design Source Code.....	26
Acknowledgements.....	41
References.....	42

CHAPTER 1 Introduction

1.1. Project aims and contributions

The focus of this project is to acquire knowledge of the mathematical conceptualizations involved in Gaussian Processes and implement them on in real-world problems. The project involves taking prerecorded official stock prices and training them using Bayesian inference to derive a gaussian process regression model, marginal likelihood, etc. to be able to make accurate (sometimes inaccurate) predictions.

Over time, the model can be used to derive other prediction for the stock price data for other companies and organizations. The project was completed in python programming language using SKLEARN and MATPLOTLIB libraries mainly because python programming language is easy to use and is also known for its wide verity of uses especially in the fields of data science and machine learning.

Contributions to this project are mentioned in the references but worth mentioning is the “Gaussian Processes for Machine Learning” by Edward Rasmussen because of the instrumental role it played in the formation of a majority of the reference material in this project.

1.2. Thesis structure

The purpose of this thesis is to point out or give a preamble of the remaining content of this paper.

The next chapter (Chapter 2) is the fundamental introduction of the theory and method of Gaussian process. Section 2.1 gives a basic idea of exactly what a “Gaussian Process Regression Model” should look like. Section 2.2 expatiates on the GP prior and the formulae involved. 2.4 gives the formula for the marginal likelihood. 2.3 is about the Hyperparameters learning or model optimization. And 2.5 is the formula for making predictions.

Chapter 3 is all about the project decomposition, basically an explanation of what the project is and what the project should do including the method/methodology of forecasting.

Chapter 4 is the implementation of the Gaussian process method on stock market. Section 4.1 expatiates on the dataset process and the stock data to be used for the experiment. In section 4.2, we see the graphical representations of the training (model fitting) and testing

(graphical representation of predictions) processes. In section 4.3, we get to see some of the source code used for some of the processes in this project.

Chapter 5 concludes the thesis.

1.3. Gaussian process definition

“Gaussian processes are a collection of random variables (stochastic processes) indexed by time or space such that every definite collection of those said random variables has a multivariate normal distribution.

A machine learning algorithm that uses kernel functions to predict the value of future of unseen points from training data hence the predictions are not just an estimate but also contain some elements of uncertainty” - Wikipedia definitions.

They are powerful tools that enable data scientists to use prior knowledge in order to make predictions. They comprise of regression, clustering and classification.

Basic Concepts of the Gaussian process are thus, “A Gaussian process is a generalization of the Gaussian distribution it represents a probability distribution over functions that are completely specified by a mean and covariance functions (kernels)”. The Mathematical definition would then be written thus, “A Gaussian process is a collection of random variables; in which any definite number of variables will be said to have a joint Gaussian distribution. Whereas a probability distribution describes random variables which are scalars or vectors (for multivariate distributions).

1.4. Gaussian process

We now make it clear that GPs are probability distributions over functions and useful non-parametric models for probabilistic regression. A GP is defined as a set of random variables, any finite number of which is jointly Gaussian distributed. They are fully specified by a mean function m and a covariance function (kernel) k , which are used to compute the mean vector and the covariance matrix of the joint distribution of a finite number of random variables.

A function can be considered an infinitely long vector f_1, f_2, \dots of function values at corresponding input locations $\mathbf{x}_1, \mathbf{x}_2, \dots$. One of the most straightforward ways to place a distribution on these function values would be a Gaussian distribution. However, the Gaussian is only defined for finite-dimensional vectors and not for functions. The Gaussian process generalizes the Gaussian distribution to this setting by treating any finite subset of

function values as a joint Gaussian distribution. This is practically relevant since we are usually only interested in finite training and test sets. If we partition the set of all function values into training set \mathbf{f} , test set \mathbf{f}_* and “other” function values, the marginalization property of the Gaussian allows us to integrate out infinitely many “other” function values to obtain a finite joint Gaussian distribution $p(\mathbf{f}, \mathbf{f}_*)$ with mean \mathbf{m} and covariance matrix \mathbf{K} given by

$$\mathbf{m} = \begin{bmatrix} m_f \\ m_{\mathbf{f}_*} \end{bmatrix}, \mathbf{K} = \begin{bmatrix} K_{ff} & K_{ff_*} \\ K_{f_*f} & K_{f_*f_*} \end{bmatrix}$$

respectively. The entries of the mean vector and the covariance matrix can be computed using the mean and covariance functions of the GP, such that $m_i = m(x_i)$, $K_{ij} = k(x_i, x_j)$, where x_i, x_j are from the training or test sets. Gaussian conditioning on this joint distribution yields the Gaussian predictive distribution of function values \mathbf{f}_* at test time as

$$p(f_{\mathbf{f}_*} | \mathbf{f}) = N(\mu, \Sigma)$$

$$\mu = m_{\mathbf{f}_*} + K_{\mathbf{f}_*f} K_{ff}^{-1} (\mathbf{f} - m_f)$$

$$\Sigma = K_{\mathbf{f}_*\mathbf{f}_*} - K_{\mathbf{f}_*f} K_{ff}^{-1} K_{f\mathbf{f}_*}$$

The GP is a useful and flexible model, and is available that allows us to use GPs as black-box for classification, regression and unsupervised learning. This black-box treatment of GPs can work very well, but it is not uncommon to encounter problems with local optima during optimization, numerical stability etc. The GP is a model that allows us to have a closer look at the sources of these issues, which are often related to the GP’s hyperparameters that control the model. In order to sort out potential issues with the inference in GPs, an intuition of what hyperparameters mean will be helpful. In the following, we will shed some light on the meaning and interpretation of various hyperparameters. Furthermore, we use this intuition to address some common problems that arise when using Gaussian processes. In our discussion, we will focus on stationary covariance functions, which are used commonly.

Using Gaussian processes are advantageous for a couple of reasons, the most important being it usually gives really good prediction results through the optimization of hyperparameters by maximizing marginal likelihood. And usually, it gives really good predictions without the user necessarily understanding how its many concepts work. It is also really advantageous compared with neural networks for instance.

Using Gaussian processes also has its cons. And without a doubt, the most pressing con is its computation time which means that in order to get desired results on time it is only advisable to work with a few thousand data points at a given time. There are mainly two ways in which to look at regression in gaussian processes.

Weight space view. Here, we see the parameters given for the probability distribution as weights and the data or data points are collected as black points. The posterior distribution is calculated using bayed inference and the regression line is calculated by taking the mean of this posterior distribution but in the case where one has an entire distribution, you can also sample from the distribution instead of taking the mean. The regression lines are calculated by sampling the parameters from the posterior distribution.

Function space view. “A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution.

Here we see that the Gaussian process is specified by its mean function and covariance and variance function. Where the mean function $m(x)$ and covariance function $k(x, x')$ are those of a real process $f(x)$.

CHAPTER 2 Gaussian Process Regression Model

2.1. Gaussian process regression

We consider a regression problem

$$y = f(x) + \epsilon,$$

where $x \in R^D$, $y \in R$, and $\epsilon \sim N(0, \sigma_n^2)$ is i.i.d. Gaussian measurement noise. We place a GP prior on the unknown function f , such that the generative process is

$$p(f) = GP(m, k)$$

$$p(y \vee f, x) = N(y \vee f(x), \sigma_n^2),$$

where $p(f)$ is the GP prior and $p(y|f, x)$ is the likelihood. Moreover, m and k are the mean and covariance function of the GP, respectively.

For training inputs $X = [x_1, \dots, x_N]$ and corresponding (noisy) training targets $y = [y_1, \dots, y_N]$ we obtain the predictive distribution

$$p(f_i | X, y, x_i) = N(f_i | \mu_i, \sigma_i^2)$$

$$\mu_i = m(x_i) + k(x_i, X) (k(X, X) + \sigma_n^2 I)^{-1} (y - m(X))$$

$$\sigma_i^2 = k(x_i, x_i) - k(x_i, X) (k(X, X) + \sigma_n^2 I)^{-1} k(X, x_i)$$

at a test point x_i .

2.1.1. Mean and covariance functions

The prior mean function $m(\cdot)$ describes the average function under the GP distribution we expect before seeing any data. Therefore, it is a straightforward way to incorporate prior knowledge about the function we wish to model. For example, any kind of (idealized) mechanical or physics model can be used as a prior. The GP would then model the discrepancy between this prior and the data. In the absence of this type of prior knowledge, a common choice is to set the prior mean function to zero, i.e., $m(\cdot) \equiv 0$. This is the setting we consider in the following. However, everything we will discuss also holds for general prior mean functions.

The covariance function $k(x, x')$ computes the covariance $\text{cov}[f(x), f(x')]$ between the corresponding function values by evaluating the covariance function k at the corresponding inputs x, x' . Practically, the covariance function encodes structural assumptions about the class of functions we wish to model. These assumptions are generally at a high level, and

may include periodicity or differentiability. (For a light, but helpful overview of different covariance functions and their properties for a more comprehensive overview.) Typically, the mean and covariance functions are parameterized. These parameters are the *hyperparameters* of the GP.

2.2. GP prior

Take the vector $f = [f(x_1), \dots, f(x_n)]$ that is made up of function values in the training set \mathcal{D} it would give us the equation,

$$p(f) = GP(0, K(X, X)),$$

as the prior distribution. In which the mean $m(\cdot) \equiv 0$ for convenience and the covariance or kernel function $k(\cdot, \cdot)$ is used to derive the Covariance matrix $K_{ij} = K(X, X)_{ij} = k(x_i, x_j)$.

Since we derived the prior distribution between training and the test points, and we can locate the points by virtue of matrix X^* whose function values are \mathbf{f}^* , we can conclude that the joint prior of $f = f(X)$, the latent function values at training inputs/locations \mathbf{X} , and $f_{\star} = f(X_{\star})$, the predicted values at testing locations \mathbf{X}^* , with covariance matrix K with $K_{ij} = k_{\theta}(x_i, x_j)$,

$$p(f, f_{\star} | X, X_{\star}, \theta) = N(\mathbf{f}, \mathbf{K})$$

Since we have the training data, we can also make use of Bayes theorem to derive the joint posterior distribution.

The posterior predictive distribution,

$$p(f_{\star} | y, X, X_{\star}, \theta, \sigma_n^2) = N(\mathbf{f}_{\star}, \mathbf{K}_{\star})$$

Consider the noise $\sigma_n^2 I$, the predictive distribution is,

$$p(y_{\star} | y, X, X_{\star}, \theta, \sigma_n^2) = N(\mathbf{y}_{\star}, \mathbf{K}_{\star})$$

2.3. Hyperparameters

When attempting to learn a successful function model data, there are typically parameters that must be defined prior to learning. These hyperparameters are used to either parametrize a family of models to be learned or to train them using optimization techniques. The performance of the model is determined by selecting the right hyperparameters, which usually require manual tuning, which is a source of much irritation. Since model training and evaluation can be costly, we must pick candidate hyperparameters for evaluation using an efficient approach. Gaussian processes are a kind of proxy model that can be used to measure

the degree of uncertainty in a forecast. They apply themselves especially well to surrogate optimization because they incorporate uncertainty. Selecting points that minimize the predicted mean of the Gaussian Method, or/and selecting points where uncertainty about the predicted mean is maximal, can help us optimize exploration in the hyperparameter process.

The covariance matrix \mathbf{K} used by the covariance function comprises the hyperparameters θ . That is $K_{ij}=K(\theta)_{ij}=k(x_i, x_j; \theta)$.

The method of maximizing the marginal likelihood of the data in accordance with the hyperparameters. This can be done by introducing latent function values that will be integrated out. If θ is the set of hyperparameters, those which must be optimized, then by maximizing the marginal likelihood with respect to the hyperparameters θ , the optimal hyperparameters can be learned which will be shown in next section.

2.4. The marginal likelihood

To train the GP, we with respect to the GP hyperparameters, i.e., the parameters of the mean and covariance function, which we summarize by θ . The marginal likelihood in our setting (regression with Gaussian likelihood) can be computed in closed form and is given by

$$p(y|X, \theta) = \int p(y|f, X) p(f|X) df \int N(y|f(X), \sigma_n^2 I) N(f(X)|0, K) df(X) \\ \int N(y|0, K + \sigma_n^2 I),$$

where \mathbf{K} is the kernel matrix with $K_{ij}=k(x_i, x_j)$, $X=[x_1, \dots, x_N]^T$ are the training inputs and $y=[y_1, \dots, y_N]^T$ are the corresponding training targets.

The hyperparameters θ appear non-linearly in the kernel matrix \mathbf{K} , and a closed-form solution to maximizing the marginal likelihood cannot be found in general. In practice, we use gradient-based optimization algorithms (e.g., conjugate gradients or BFGS) to find a (local) optimum of the marginal likelihood.

Maximizing the marginal likelihood behaves much better than finding maximum likelihood or maximum a-posteriori point estimates

$$\arg \max_{f(X), \sigma_n} p(y \vee f(X), \sigma_n^2 I), \text{ maximum likelihood}$$

$$\arg \max_{f(X), \sigma_n} p(y|f(X), \sigma_n^2 I) p(f(X)|\theta), \text{ maximum a posteriori}$$

if we were to compute them. (Making a connection to a linear regression setting, the parameters of a GP would be the function values $f(\mathbf{X})$ themselves.) These two approaches would lead to overfitting, since it is possible to get arbitrarily high likelihoods by placing the function values $f(\mathbf{X})$ on top of the observations \mathbf{y} and letting the noise σ_n tend to zero.

The marginal likelihood does not fit function values directly, but integrates them out, i.e., technically we cannot “overfit” as no fitting happens. By averaging (integrating out) the direct model parameters, i.e., the function values, the marginal likelihood automatically trades off data fit and model complexity. Choose a model that is too inflexible, and the marginal likelihood $p(y \vee X, \theta)$ will be low because few functions in the prior fit the data. A model that is too flexible spreads its density over too many data sets, and so $p(y \vee X, \theta)$ will also be low.

By maximizing the marginal likelihood, or more often, the log expression,

$$\log p(y|X, \theta) = \log N(y|0, K + \sigma_n^2 I) = -\frac{n}{2} \log 2\pi - \frac{1}{2} \log |K + \sigma_n^2 I| - \frac{1}{2} y^\top (K + \sigma_n^2 I)^{-1} y,$$

we find point estimates of the hyperparameters controlling the mean and covariance function (Using a point estimate for the hyperparameter is a convenience only). The complexity of above equation is $\mathcal{O}(N^3)$ in time and $\mathcal{O}(N^2)$ in space.

This log expression can be maximized numerically by python's **SKLEARN** optimizer **fmin_l_bfgs_b** in order to produce the desired hyperparameters,

$$\theta^i = \arg \max_{\theta} \log p(y|X, \theta).$$

The model may still look like overfitting if data is sparse. However, given that usually there are only few hyperparameters, the posterior is usually very peaked, and a point estimate is not a terrible approximation. To be robust to overfitting effects caused by point estimates of the hyperparameters, we could perform Bayesian inference over them as well by, for example, Markov Chain Monte Carlo.), such that the data is explained well by the GP.

CHAPTER 3 System Design

3.1. System development life cycle description

The planning stage (also called the feasibility stage) is exactly what it sounds like: the phase in which plan for the upcoming project are set in motion. It helps to define the problem and scope of the task at hand, as well as determine the objectives for how to fix them. By developing an effective outline for the upcoming development cycle, problems are often theoretically caught before they affect development. The planning phase helps to secure the funding and resources needed to make the plan happen. Perhaps most importantly, the planning stage sets the project schedule, which can be of key importance if development is for a commercial product that must be sent to market by a certain time.

Problem definition

Gaussian process regression for the prediction of stock price trends of apple stock price data AAPL.csv using time series from 2008 to 2016 for prediction of 2017 trends and 2018 first two quarters for the prediction of the trend movement in the last two quarters of 2018.

Constraints and requirements

Requirements: stock price data in .csv format, sublime text editor, python programming language and command line.

Constraints: time series for training and observation 2008 to 2016 and 2018 time series respectively. Predictions from normalized prices only, predictions from adjusted closed prices observation, prediction analysis done in two years only; 2017 and last two quarters of 2018.

Dataset analysis

The analysis stage includes gathering all the specific details required for a new system as well as determining the first ideas for prototypes also described as the first phase of cleaning process, using Jupiter notebook to perform certain operations to ensure that there is no redundancy of data and there are no null values present in the dataset for the purpose for getting proper prediction analysis.

Algorithm design

The design stage is a necessary precursor to the main developer stage.

First, we outline the details for the overall project, alongside specific aspects, typically like model fitting and predictions. The information is then made into a more logical structure that can later be implemented in a programming language. Operation, training, and

maintenance plans will all be drawn up so that developers know what they need to do throughout every stage of the cycle moving forward.

It describes the different modules used in the implementation of the different processes involved in observing and successful prediction.

Implementation

After testing, the overall design for the results will come together. Different modules or designs will be integrated into the primary source code through developer efforts, usually by leveraging training environments to detect further errors or defects. This part mainly consists of the results from model fitting/training the dataset and the testing/prediction results.

Detailed presentation of the processes and their results: dataset preprocess, cleaning phases, model fitting information, initial observation graphs and prediction observation graphs.

3.2. Project decomposition

The products of this project are divided into following categories.

(1) Algorithm design

The algorithm design entails all the steps involved in transforming AAPL data and making sure that it became usable for deriving the somewhat accurate predictions

(2) Dataset preprocess

The dataset preprocess contains the formatting information, the details of the cleaning process and the hyperparameters used

(3) Price trend observations

The upward or downward trends in the prices of AAPL stocks are observed and then a graph of normalized price trends is created from the normalized prices (normalized: not necessarily affected by splits or dividends).

(4) Observations of the normalized price trends

The algorithm is used to observe the upward or downward trends in adjusted or normalized closed prices. The idea is to forecast or predict the trend in prices as accurately as possible against the actual observations taken during the training period.

(5) Price trend predictions

The trend of the predictions is plotted against the actual price trend observations in order to see the level of accuracy that the algorithm is capable of.

3.3. Algorithm design

In earlier parts of this project, it is made known that the entirety of this project was completed in python programming language with the help of the pandas, MATLAB and sklearn libraries.

This part of the project however, focuses on the detailed (in some sense) description of the different python files that were used to compile and run this project and what they may entail.

It is however worth mentioning that this project was compiled and able to run with the help of command prompt using a python command line statement (`python - datapresenter.py`).

3.3.1. Dataset preparation

In order to be able to carry out this project, the data has to be made available and this is made possible by downloading the dataset(s) (year of inception to 2018) in .csv format and storing in the “data” file folder. This is where the “`import.py`” file will have access to the dataset and will be able read the contents of the dataset to the data handler. See figure 3.1.

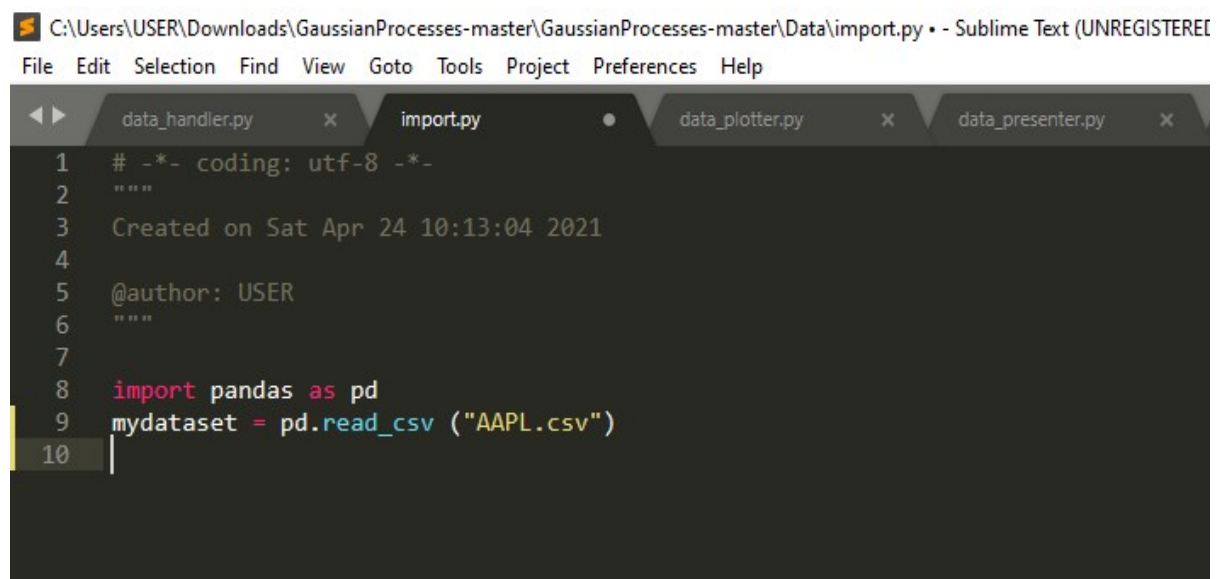


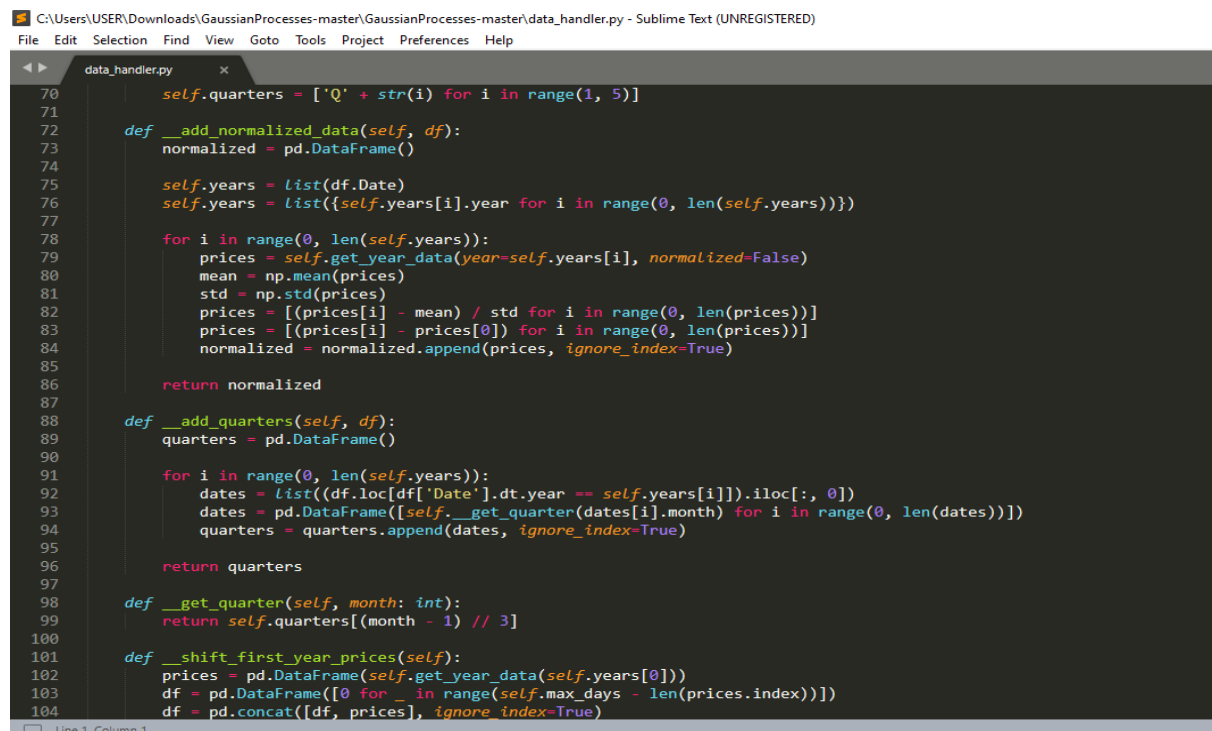
Figure 3.1 Dataset preparation

(1) `data_handler.py`

Data handler file is configured in order to be able to transform data from one format to another. This means that it is used to transform the data from csv format to the format required for making observations arranging the data into four different categories (time series) that would then go on to be used for making observations about the price trends of

those categories and later the predictions. These four categories (time series) as will be seen later on include; year of inception till 2018, 2008 to 2016 for the price trend prediction of the year 2017, trend of normalized prices throughout the years, then observations from 2008 to the first two quarters of the 2018 trading period/year. This file parses .csv files using the python class (csvhandler) and returns an iterable object to fetch the data in order to commence the dataset fitting/training process.

The data handler also contains source code that is used in the cleaning process, particularly in getting an equal number of trading days for every year observed for prediction and also deriving normalized prices. Figure 3.2 shows the definition of the data handler file.



```

70     self.quarters = ['Q' + str(i) for i in range(1, 5)]
71
72     def __add_normalized_data(self, df):
73         normalized = pd.DataFrame()
74
75         self.years = list(df.Date)
76         self.years = list({self.years[i].year for i in range(0, len(self.years))})
77
78         for i in range(0, len(self.years)):
79             prices = self.get_year_data(year=self.years[i], normalized=False)
80             mean = np.mean(prices)
81             std = np.std(prices)
82             prices = [(prices[i] - mean) / std for i in range(0, len(prices))]
83             prices = [(prices[i] - prices[0]) for i in range(0, len(prices))]
84             normalized = normalized.append(prices, ignore_index=True)
85
86         return normalized
87
88     def __add_quarters(self, df):
89         quarters = pd.DataFrame()
90
91         for i in range(0, len(self.years)):
92             dates = list(df.loc[df['Date'].dt.year == self.years[i]].iloc[:, 0])
93             dates = pd.DataFrame([self.__get_quarter(dates[i].month) for i in range(0, len(dates))])
94             quarters = quarters.append(dates, ignore_index=True)
95
96         return quarters
97
98     def __get_quarter(self, month: int):
99         return self.quarters[(month - 1) // 3]
100
101     def __shift_first_year_prices(self):
102         prices = pd.DataFrame(self.get_year_data(self.years[0]))
103         df = pd.DataFrame([0 for _ in range(self.max_days - len(prices.index))])
104         df = pd.concat([df, prices], ignore_index=True)

```

Figure 3.2 Data handler definition

(2) gp_wrapper.py

The gp_wrapper.py file is written by importing “gaussianprocessregressor” and “RBF” from the sklearn_gaussianprocess and sklearn_gaussianprocess.kernels both from python's sklearn library, it also takes the data from the datahandler.py file. The purpose of this file is simple. Here the data taken from the data handler is sampled without noise. Fitting a model without noise means that the regression will most likely pass through each and every data point. Hence the purpose of this file is for training the dataset/model fitting. This is done by calling the “wrapper” class which defines and organizes the data in the order.

company_data (AAPL data)

- `__prices_data` (AAPL price data)
- `__quarters` (2008 to first two of 2018 for the last two of 2018)
- `__max_days` (total number of trading days)
- `__alpha` (for specifying the gaussian noise level)
- `__iterations` (from data handler file)

And then applies the `__kernels` (specifies covariance) and `__gp` model from the `gaussianprocessregressor`.

(3) data_plotter.py

The data plotter file makes use of matplotlib's libraries and also imports the data to be used in plotting the different graphs both of observation(training) and plotting predictions(testing) from both the data handler and the `gpr` wrapper modules. This module/file is used for the purpose of plotting the data in the order in which they are demarcated by the `gpr` wrapper module/file.

This is done by calling the “`plotter`” class which organizes the data in the order.

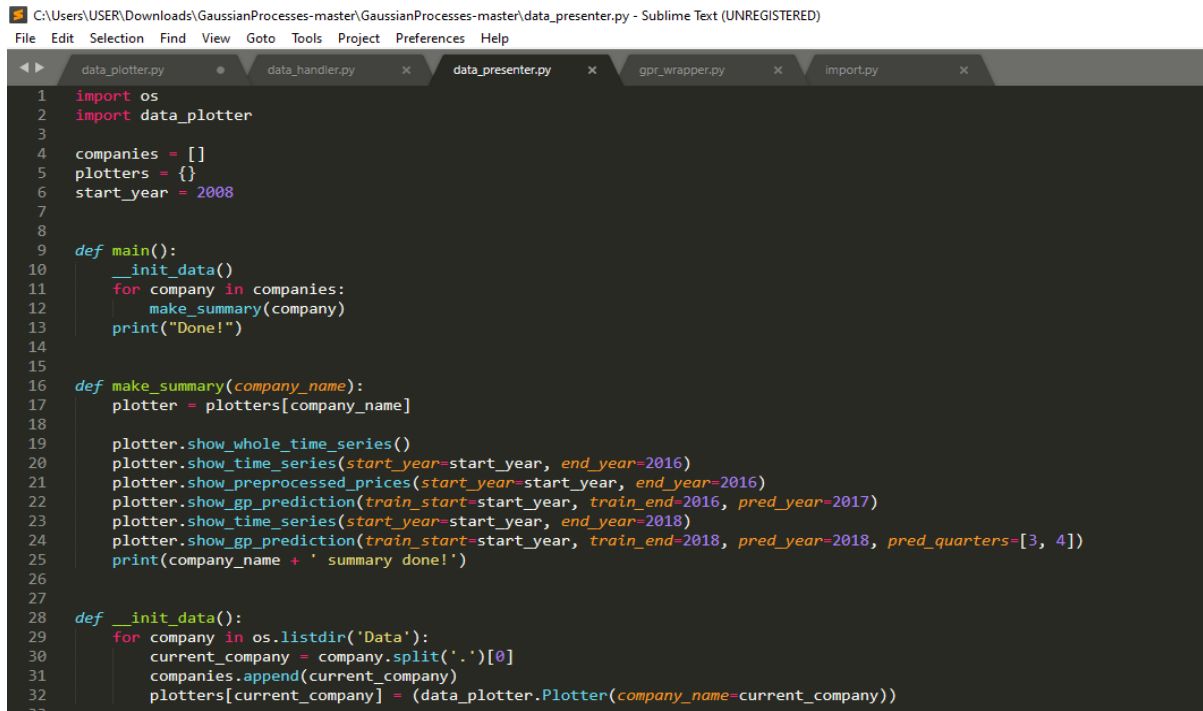
- `__company_name` (AAPL)
- `__company_handler` (from the `datahandler` file)
- `__prices_data` (all the price trend data for initial and normalized prices)
- `__quarters` (2008 to first two of 2018 for the last two of 2018)
- `__years` (years for observation graphs as well as prediction graphs)
- `__max_days` (maximum number of trading days)
- `__quarter_length` (defines the length of a single quarter)

And then applies `__gpr` from the `Gaussianprocessregressor` to show or plot the prediction data.

(4) data_presenter.py

The data presenter module takes the information in form of the plotted graphs from the data plotter module and is responsible for showing each time series and the two predictions (year2017 and the last two quarters of 2018) according to the coordinates stipulated by the data plotter module. After successful execution, it prints “`company name + summary done`”.

The code below shows the order in which the data presenter shows plotted graphs as different time series and the predictions for each timeseries.



```

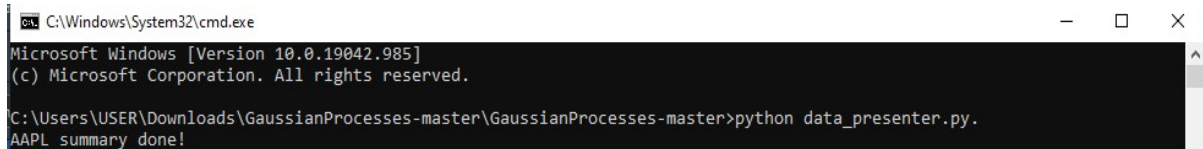
1 import os
2 import data_plotter
3
4 companies = []
5 plotters = {}
6 start_year = 2008
7
8
9 def main():
10     __init_data()
11     for company in companies:
12         make_summary(company)
13     print("Done!")
14
15
16 def make_summary(company_name):
17     plotter = plotters[company_name]
18
19     plotter.show_whole_time_series()
20     plotter.show_time_series(start_year=start_year, end_year=2016)
21     plotter.show_preprocessed_prices(start_year=start_year, end_year=2016)
22     plotter.show_gp_prediction(train_start=start_year, train_end=2016, pred_year=2017)
23     plotter.show_time_series(start_year=start_year, end_year=2018)
24     plotter.show_gp_prediction(train_start=start_year, train_end=2018, pred_year=2018, pred_quarters=[3, 4])
25     print(company_name + ' summary done!')
26
27
28 def __init_data():
29     for company in os.listdir('Data'):
30         current_company = company.split('.')[0]
31         companies.append(current_company)
32         plotters[current_company] = (data_plotter.Plotter(company_name=current_company))
33

```

Figure 3.3 Data presenter exemplification

(5) Command line statement

Command prompt is used to execute the simple command line statement “python data_presenter.py” which compiles the program to show the results of all plots of all the time series in .png format. The statement has to however be executed from the proper file location.



```

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

C:\Users\USER\Downloads\GaussianProcesses-master\GaussianProcesses-master>python data_presenter.py.
AAPL summary done!

```

Figure 3.4 Data presenter execution exemplification

The following flowchart shows the various steps explained above in order (it would be important to state that in the algorithm flowchart, processes like the dataset cleaning are represented by the datahandler.py which is used for the second and most important phase of the cleaning process).

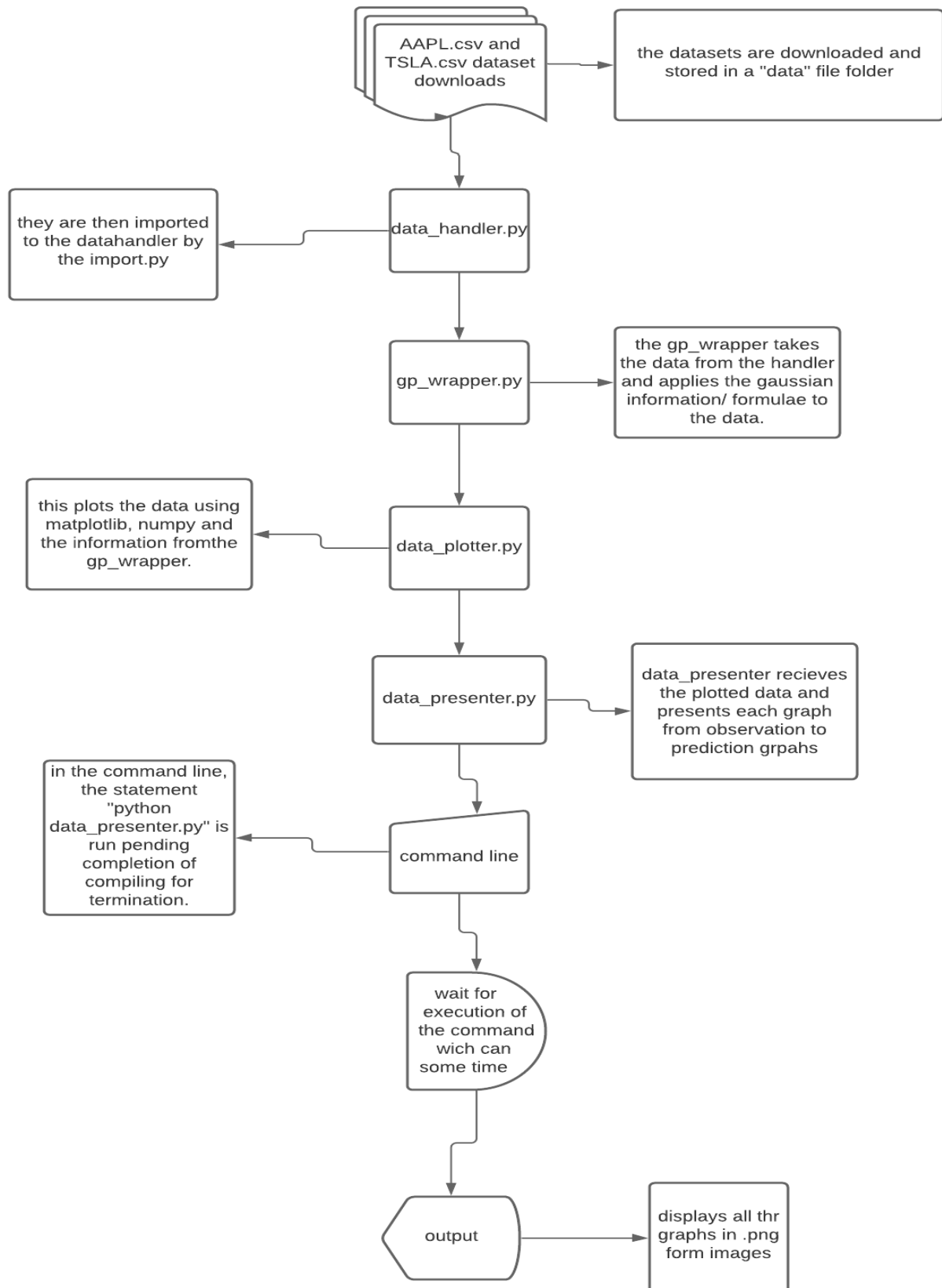


Figure 3.5 System flowchart

CHAPTER 4 Implementation

4.1. Dataset

4.1.1. Dataset preparation

For this project, the stock information that was used is Apple (AAPL) stocks from the year it was founded 1985 to the year 2018 the data was downloaded from the yahoo finance website in form of a .csv file and stored in the data directory of the project.

4.1.2. Dataset preprocess

(1) Formatting

The dataset was downloaded in the proper format (csv) file format in order to remove or eliminate the need to format or change the dataset from a relational database format to its intended format (csv file format) for instance.

(2) Cleaning

The cleaning process is divided into two parts.

In the first part, some observations are done in order to ensure that the dataset is completely independent of redundant or repeating data (values) and that there are no empty spaces where values are supposed to be. This process was carried out using Jupyter's notebook for python and the demonstration is as follows:

First the dataset is imported unto jupyter, and goes by the name “AdjClose” and then read by the file. The command “AdjClose.head()” is then run to show in summary what the dataset looks like. “AdjClose.shape()” is then run to show the number of rows and column in counted in the dataset which number “(9594, 6)”. The processes are shown in Figure 4.1.

A command is run which then shows the datatypes of all the columns in the dataset to determine if they are appropriate datatypes that can be used for the analysis (shown in Figure 4.2).

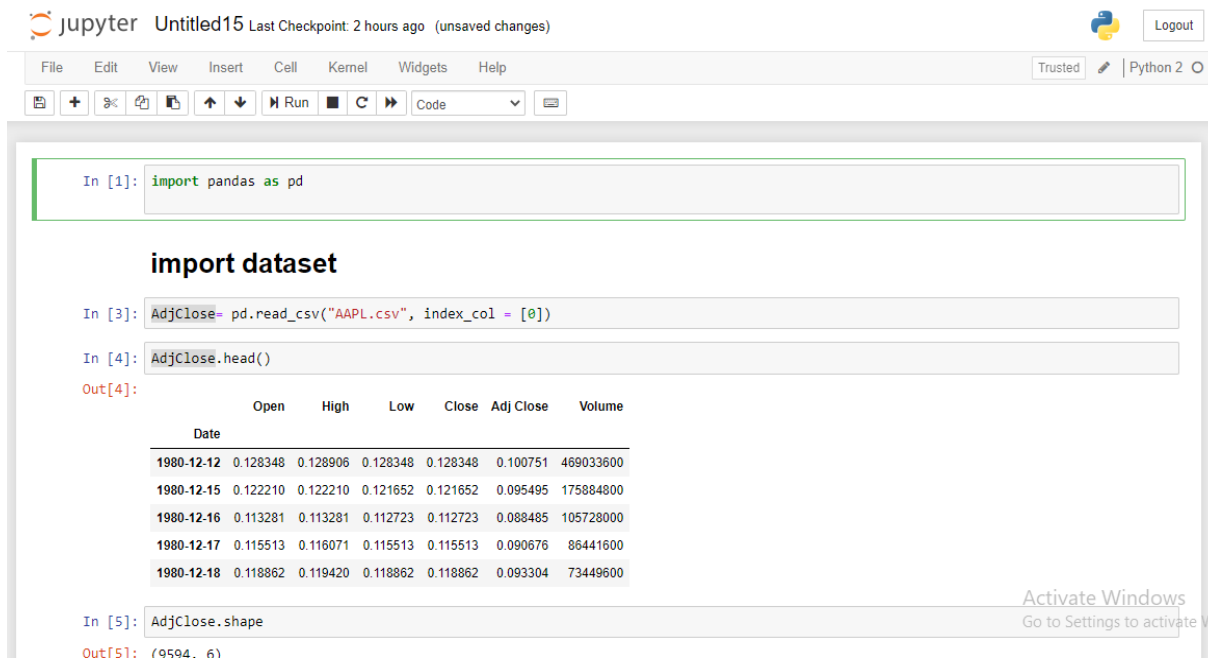


Figure 4.1 Dataset used in the thesis

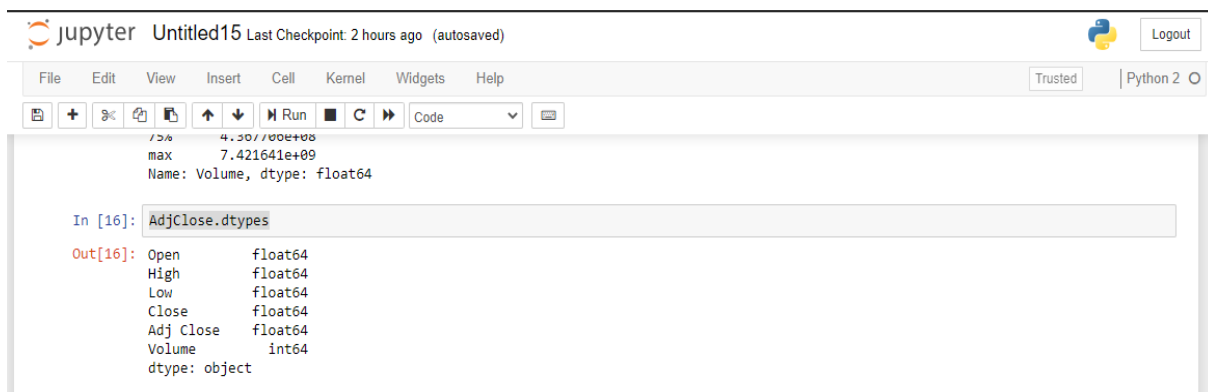


Figure 4.2 Datatypes of the dataset

The datatypes include floats and integers, which means the datatypes are appropriate.

Next, “`AdjClose.duplicated()`” is run to check the dataset for duplicated values or values that appear more than once. The results show “false” as displayed in Figure 4.3 means there are none in the dataset.

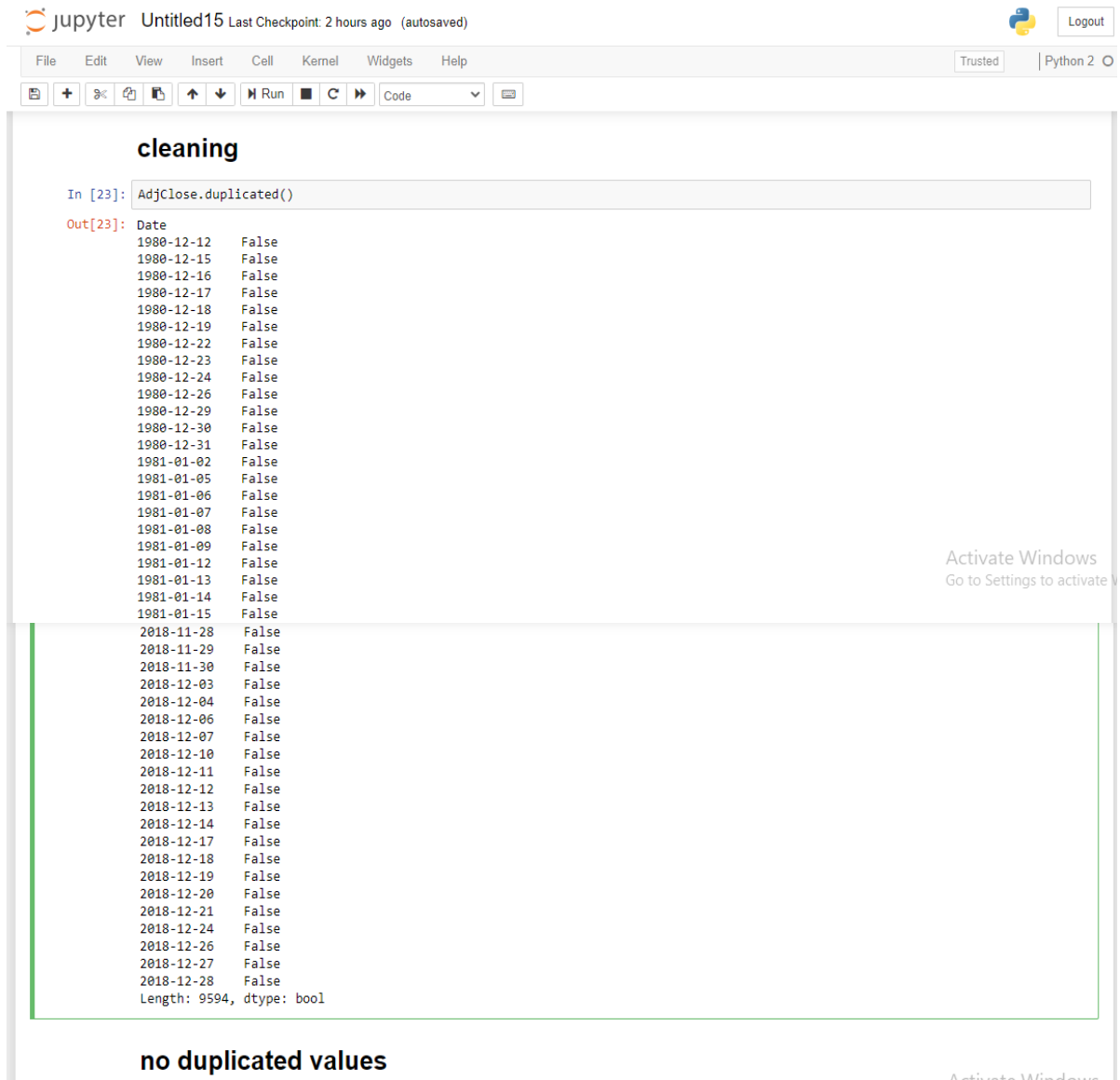


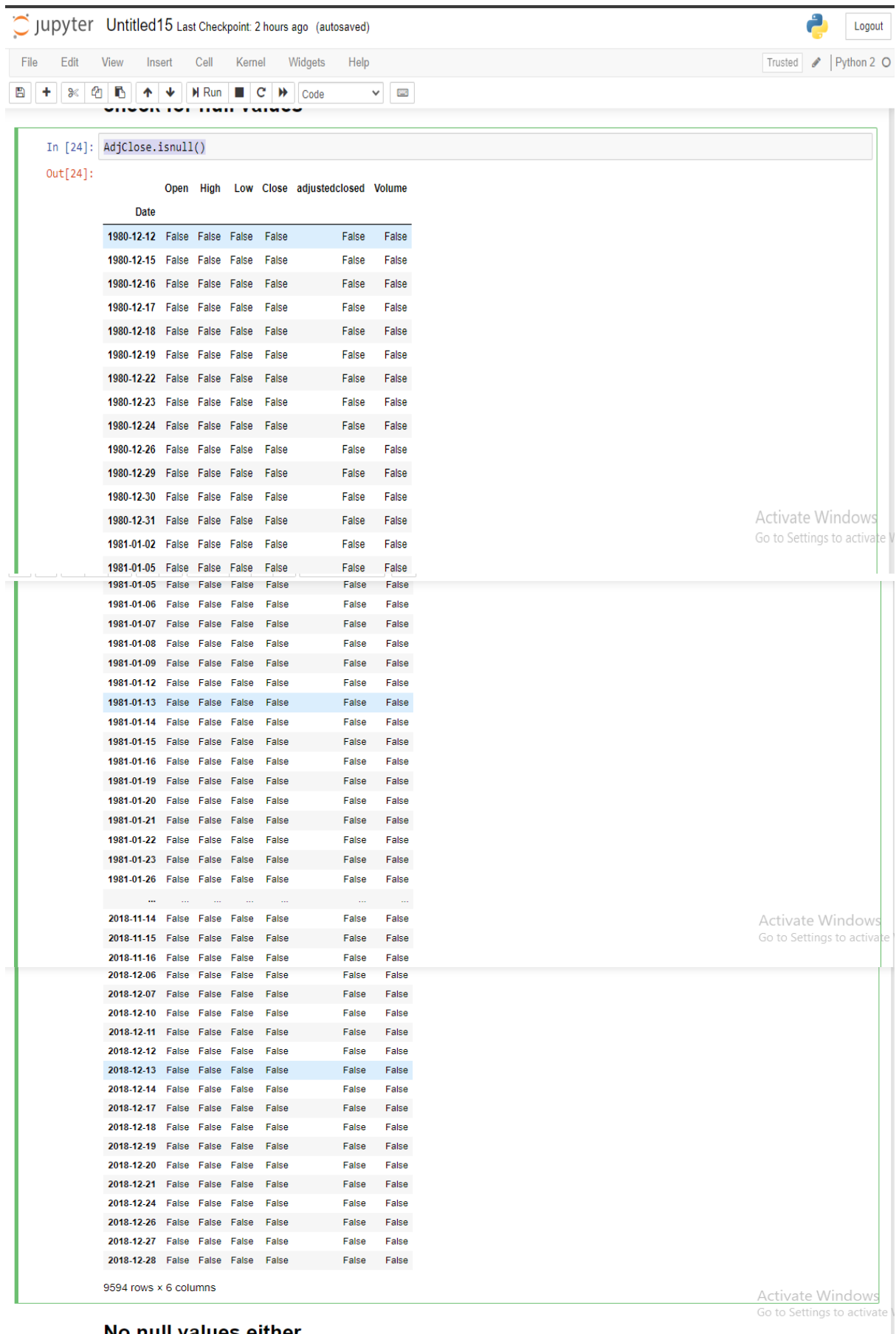
Figure 4.3 Cleaned dataset

Finally, the dataset is checked for null values by executing the command “`AdjClose.isnull()`”. The results are shown in Figure 4.4.

With the absence of any null or redundant values in the dataset, it is ready for the second phase of cleaning which concentrates on getting equal number of trading days in every year and also derives normalized prices.

The daily changes of adjusted closing prices of these stocks were examined and the historical data was downloaded in the form of csv file from the yahoo finance section. There are two sample periods taken for the indices,

- (1) first sample period is based on years 2008-2016 for prediction of whole year 2017,
- (2) and second based on years 2008-2018 (up to end of second quarter) for prediction of the rest of the 2018 year.



No null values either

Figure 4.4 Cleaned dataset

There are about 252 days of trading per year since no data is observed on weekends. However, some years have more than 252 days of trading and some less so we choose to ignore the days higher 252+ days and for those with less trading days, the remaining days are filled up with the mean of the year to have equal number of trading days for all years. The reason for choosing to use adjusted close prices because the aim is to predict the trend of the stocks and not the prices. The adjusted close price is used to avoid the effect of dividends and splits because when stocks have splits, their prices drop by half. The adjusted close prices are standardized to zero mean and unit standard deviation. The prices are also normalized in each year to avoid the variation from previous years by subtracting the first day in order to start from zero.

4.2. Hyperparameters learning

The method used is called optimization. This is done by maximizing the marginal likelihood (log marginal likelihood) as described in section 2.4,

$$\log p(y|X, \theta) = \log N(y|0, K + \sigma_n^2 I) = -\frac{n}{2} \log 2\pi - \frac{1}{2} \log |K + \sigma_n^2 I| - \frac{1}{2} y^\top (K + \sigma_n^2 I)^{-1} y,$$

to get the desired hyperparameters θ , which are $\theta^{\hat{c}} = \arg \max_{\theta} \log p(y|X, \theta)$.

This is achievable by making use of python's SKLEARN optimizer **fmin_1_bfgs_b**.

4.3. Training (Model fitting)

The first graph is a representation (graphic) of the observed stock prices at Apple.NC from the year 1980 to the year 2018 and the last two vertical lines on the graph represent the year 2018.

Consider a set of N real time series each of length $\{y_t^i\}, i=1, \dots, N, t=1, \dots, M_i$. In this application each i represents a different year, and the series is the sequence of the particular prices during the period where it is traded. Considering the length of the stock market year, usually M will be equal to 252 and sometimes less if incomplete series is considered (for example this year) assuming that the series follow an annual cycle. Thus, knowledge from past series can be transferred to a new one in order for them to be forecast.

Each trade year of data is treated as a separate time series and the corresponding year is used as an independent variable in regression model.

The forecasting/prediction problem is that given observations from the complete series $i=1, \dots, N-1$ and (optionally) from a partial last series $\{y_t^N\}, t=1, \dots, M_N$, we want to

extrapolate the last series until a particular predetermined endpoint (usually a multiple of a quarter length during a year) to characterize the joint distribution of $\{y_\tau^N\}$, $\tau = M_N + 1, \dots, M_N + H$ for some H . We also have a set of non-stochastic explanatory variables specific to each series, $\{x_t^i\}$, where $x_t^i \in \mathbb{R}^d$. The goal is to be able to find an effective representation of $P\left(\{y_\tau^N\}_{\tau=M_N+1, \dots, M_N+H} \vee \{x_t^i, y_t^i\}_{t=1, \dots, M_i}^{i=1, \dots, N}\right)$, with τ, i and t ranging over the forecasting horizon, the available time series and the observations within a series.

Everything mentioned in this section was implemented in Python using the `sklearn` library and `sklearn.gaussian_process` in particular.

This section shows the list of graphs showing price trend observations.

First, we have the graph for price observation from the year 1980 to the year 2018 with the price ranging from 0 to 50. Then the two graphs that follow are the time series used for the 2017 prediction and the last two quarters of 2018 in order and also the graph showing the price trend of the normalized prices. As we can see from the price trend observations for these different time series, their prices begin and peak at different levels making the use of normalized prices necessary in order to give us a sense of what the price trends for different years would look like on the same scale in terms of Prices and number of trading days and trading quarters in the year. This is very important because it gives us a context from which predictions can be understood with the same consistency when being looked at.

The first graph (Figure 4.5) is a representation (graphic) of the observed stock prices at Apple.NC from the year 1980 to the year 2018 and the last two vertical lines on the graph represent the year 2018.

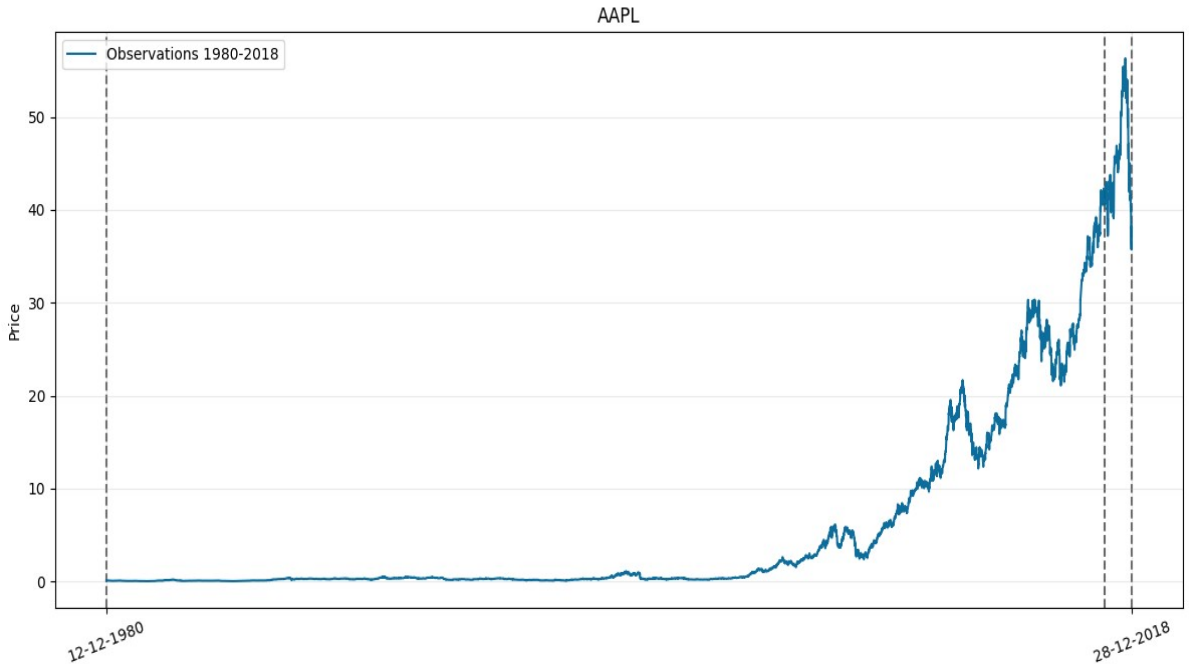


Figure 4.5 Observations of the data (stock prices)

This second graph (Figure 4.6) is a representation of the observation of the price movement from the year 2008 to the year 2016 this is the first sample period that was used in training or model fitting for the prediction of price trends in the year 2017.

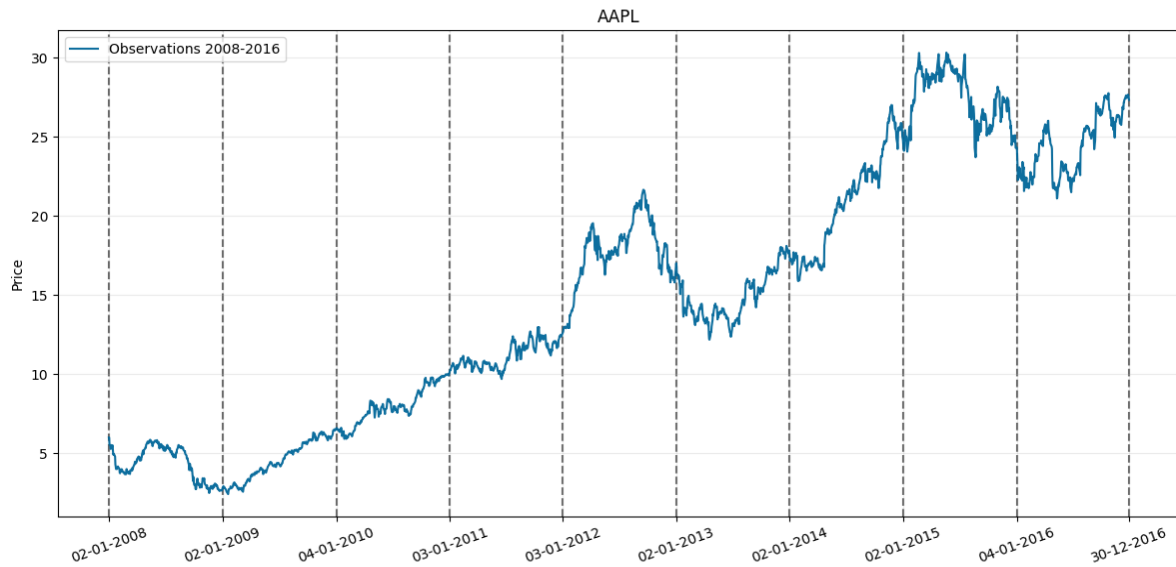


Figure 4.6 Observations from 2008 to 2016

Figure 4.7 shows the normalized prices of AAPL stock that would be used for the price predictions of year 2017 and for the quarterly prediction of the last two quarters in 2018. The different lines represent how well the prices for each year did across all the four quarters of a yearly trading period. This is arguably the most important graph in the group of observation graphs because of the context it gives for understanding the predictions.

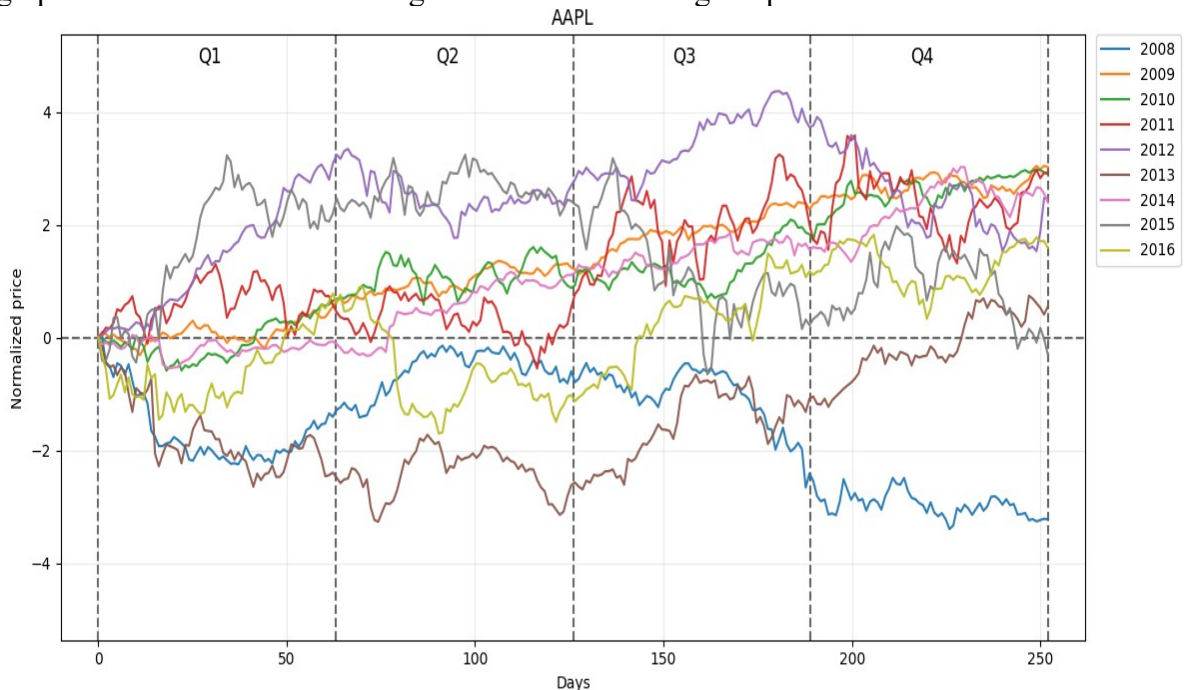


Figure 4.7 Normalized prices of AAPL stock

Figure 4.8 represents the price trend observations from 2008 to year 2018 but only the first two quarters of the year 2018.

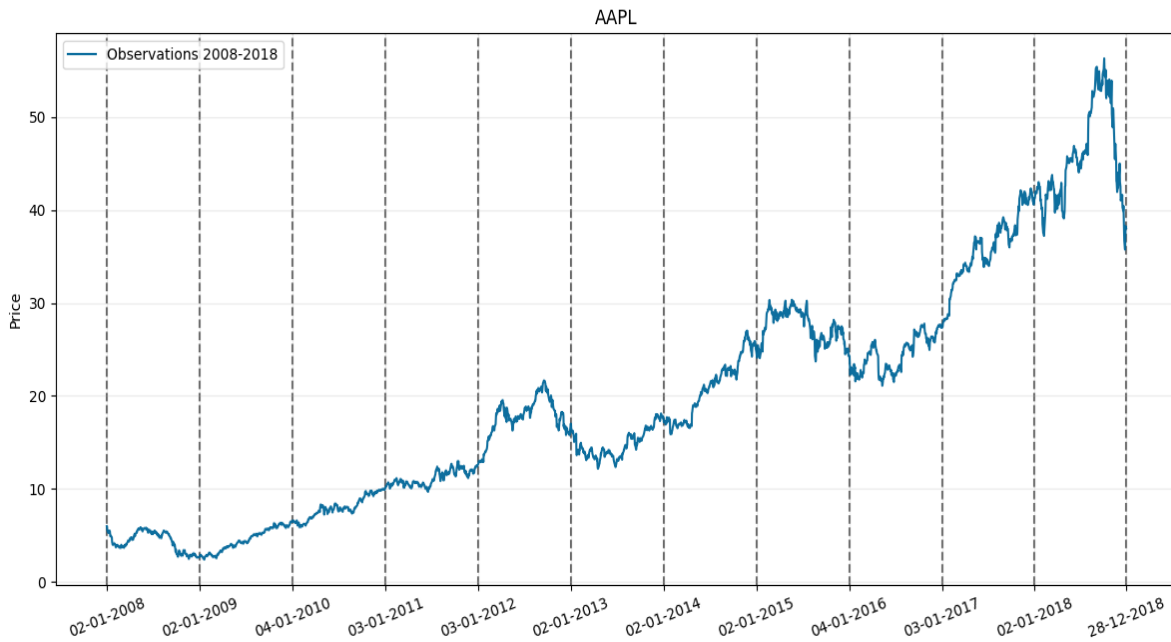


Figure 4.8 Price trend from 2008 to 2018

4.3. Testing (Make predictions)

In Figure 4.9 we can see the normalized price trend observations for Apple.NC in the year 2017 plotted against the GPR prediction for the same year across the four quarters of the trading period. From observations, the predictions were only accurate at the beginning of the first quarter, somewhere between the end of the third quarter and the beginning of the fourth quarter and also again towards the end of the fourth quarter.

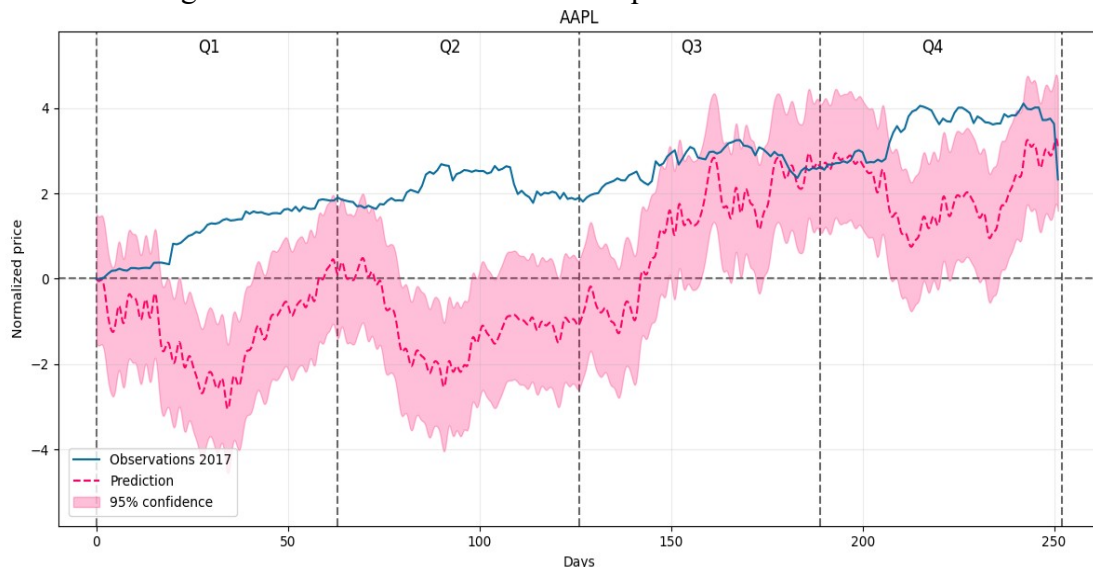


Figure 4.9 Normalized price trend observations for Apple.NC in the year 2017

Here the normalized Price trend information for the year 2018 in APPLE.NC are observed but the predictions are only for the third and fourth quarter of the same year.

Here one can say that the price prediction is largely more accurate than that of the previous prediction with its most accurate points yet again being towards the middle and end of the third and the beginning of the fourth quarter

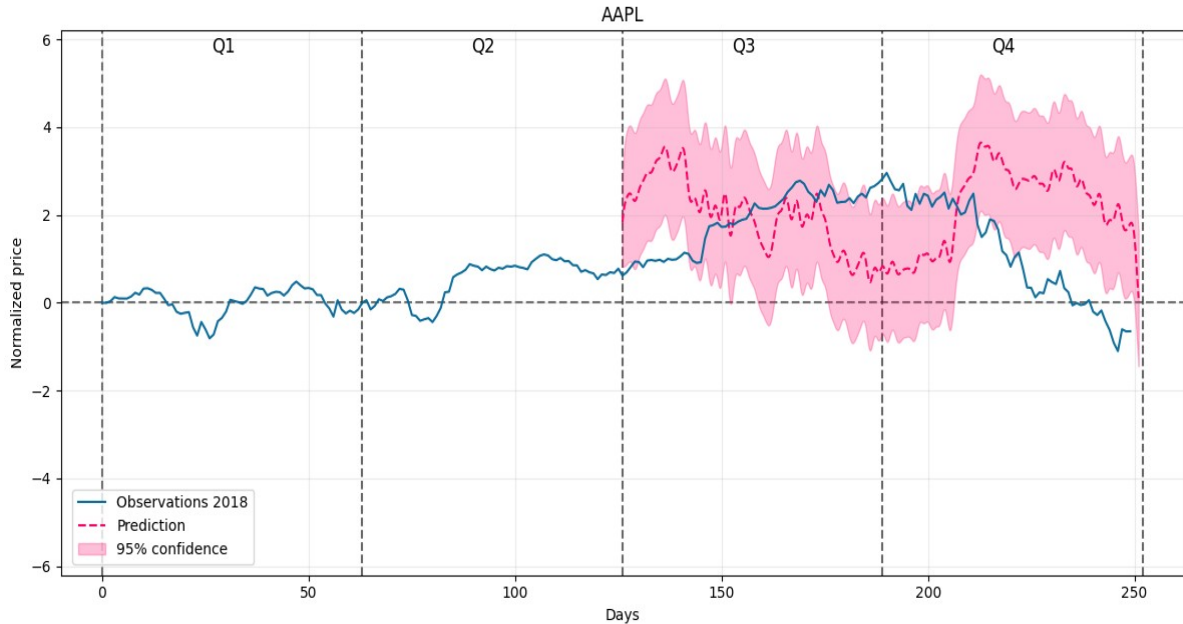


Figure 4.10 Price predictions for the third and fourth quarter of year 2018

CHAPTER 5 Conclusions and Observations

The two experiments conducted here show the power of using Gaussian processes for predicting stock prices. There were a number of different ways to test how to make a good prediction based on the training data. By varying the lengths of time considered for training, we achieved different results and accuracies. Using a long time period in experiment 1 resulted in fairly good stock predictions, while using shorter times resulted in sub-optimal predictions. In experiment 2, on the other hand, converse was true.

We have seen that some of the larger scale predictions in this project have been far from accurate, however, the predictions over a shorter period of time seem to have been more accurate and could prove to be somewhat useful for instance to people who would want to invest in these companies on the stock market for whatever reasons they may have.

However, through the duration of this project, I have learnt how gaussian processes can be used to predict and forecast trend directions of stock prices through regression. It featured a rather simple demonstration of how gaussian processes (regression) can be used to predict the upward or downward trend of prices in the stock market. We also saw the connection between Gaussian processes and Bayesian thinking by looking at how Bayesian inference can be used to derive a posterior distribution over certain function. Practically speaking however, using Gaussian process are not the most efficient method to use, because the computational load for predictions is high. To make money in the stock market using a machine learning technique, a faster method of computation would be required.

Gaussian processes can be very complex and yet very interesting but they are more interesting because they are really useful in almost any thing that has to do with uncertainty and that means that in our world today, gaussian processes are way more important than we give them credit for. A few areas where they can be and are used today include machine learning, stock market trend predictions, statistical modelling, predicting Brownian motion, oil and gas exploration, robotics and many more fields of innovation and expertise.

A downside of gaussian processes is however that it sometimes takes long computation time and this significantly limits the number of figures that you work with especially on smaller projects. This makes it computationally expensive.

The concept is named after world renown scientist and mathematician Carl Friedrich Gauss for the numerous significant contributions he made during his lifetime in the fields of

Mathematics and Science. These contributions have proven to be invaluable and will continue to do so in time to come.

Appendix. Entire Algorithm Design Source Code

1. Import.py

```
import pandas as pd
mydataset = pd.read_csv ("AAPL.csv")
```

2. Data_handler.py

```
import pandas as pd
import numpy as np

class CsvHandler:
    df = None
    quarters = None
    years = None
    max_days = None

    def __init__(self, csv_name: str):
        self.__load_data(csv_name)
        self.df['Norm Adj Close'] = self.__add_normalized_data(self.df)
        self.df['Quarter'] = self.__add_quarters(self.df)
        self.max_days = 252

    def get_equal_length_prices(self, normalized=True):
        df = self.__shift_first_year_prices()
        for i in range(1, len(self.years)):
            df = pd.concat([df,
pd.DataFrame(self.get_year_data(year=self.years[i], normalized=normalized))],
axis=1)

        df = df[:self.max_days]

        quarters = []
```

```
for j in range(0, len(self.quarters)):
    for i in range(0, self.max_days // 4):
        quarters.append(self.quarters[j])
quarters = pd.DataFrame(quarters)

df = pd.concat([df, quarters], axis=1)
df.columns = self.years + ['Quarter']
df.index.name = 'Day'

self.__fill_last_rows(df)

return df

def get_year_data(self, year: int, normalized=True):
    if year not in self.years:
        raise ValueError('\n' +
                           'Input year: {} not in available years:
{}'.format(year, self.years))

    prices = (self.df.loc[self.df['Date'].dt.year == year])
    if normalized:
        return np.asarray(prices.loc[:, 'Norm Adj Close'])
    else:
        return np.asarray(prices.loc[:, 'Adj Close'])

def get_whole_prices(self, start_year: int, end_year: int):
    if start_year < self.years[0] or end_year > self.years[-1]:
        raise ValueError('\n' +
                           'Input years out of available range! \n' +
                           'Max range available: {}-{}\n
n'.format(self.years[0], self.years[-1])) +
                           'Was: {}-{}'.format(start_year, end_year))
```



```
df = (self.df.loc[(self.df['Date'].dt.year >= start_year) &
(self.df['Date'].dt.year <= end_year)])
df = df.loc[:, ['Date', 'Adj Close']]

return df

def show(self, max_rows=None, max_columns=None):
    with pd.option_context('display.max_rows', max_rows,
'display.max_columns', max_columns):
        print(self.df)

def __load_data(self, csv_name: str):
    self.df = pd.read_csv('Data/' + csv_name + '.csv')
    self.df = self.df.iloc[:, [0, 5]]
    self.df = self.df.dropna()
    self.df.Date = pd.to_datetime(self.df.Date)
    self.quarters = ['Q' + str(i) for i in range(1, 5)]

def __add_normalized_data(self, df):
    normalized = pd.DataFrame()

    self.years = list(df.Date)
    self.years = list({self.years[i].year for i in range(0,
len(self.years))})

    for i in range(0, len(self.years)):
        prices = self.get_year_data(year=self.years[i], normalized=False)
        mean = np.mean(prices)
        std = np.std(prices)
        prices = [(prices[i] - mean) / std for i in range(0, len(prices))]
        prices = [(prices[i] - prices[0]) for i in range(0, len(prices))]
        normalized = normalized.append(prices, ignore_index=True)

    return normalized
```

```
def __add_quarters(self, df):
    quarters = pd.DataFrame()

    for i in range(0, len(self.years)):
        dates = list((df.loc[df['Date'].dt.year == self.years[i]]).iloc[:,
0]))
        dates = pd.DataFrame([self.__get_quarter(dates[i].month) for i in
range(0, len(dates))])
        quarters = quarters.append(dates, ignore_index=True)

    return quarters

def __get_quarter(self, month: int):
    return self.quarters[(month - 1) // 3]

def __shift_first_year_prices(self):
    prices = pd.DataFrame(self.get_year_data(self.years[0]))
    df = pd.DataFrame([0 for _ in range(self.max_days -
len(prices.index))])
    df = pd.concat([df, prices], ignore_index=True)

    return df

def __fill_last_rows(self, df):
    years = self.years[:-1]

    for year in years:
        mean = np.mean(df[year])
        for i in range(self.max_days - 1, -1, -1):
            current_price = df.iloc[i, df.columns.get_loc(year)]
            if np.isnan(current_price):
                df.iloc[i, df.columns.get_loc(year)] = mean
            else:
```

```
break
```

3. Data_plotter.py

```
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import numpy as np
import data_handler
import gpr_wrapper
```

```
class Plotter:
```

```
    __company_name = None
    __company_handler = None
    __prices_data = None
    __quarters = None
    __years = None
    __max_days = None
    __quarter_length = None
    __gpr = None
```

```
    def __init__(self, company_name: str):
        self.__company_name = company_name
        self.__company_handler = data_handler.CsvHandler(company_name)
        self.__prices_data = self.__company_handler.get_equal_length_prices()
        self.__quarters = self.__company_handler.quarters
        self.__years = self.__company_handler.years
        self.__max_days = self.__company_handler.max_days
        self.__quarter_length = int(self.__max_days / 4)
        self.__gpr = gpr_wrapper.Wrapper(company_name)
```

```
    def show_preprocessed_price(self, year: int):
        self.show_preprocessed_prices(start_year=year, end_year=year)
```

```
    def show_preprocessed_prices(self, start_year: int, end_year: int):
```

```
self.__validate_dates(start_year=start_year, end_year=end_year)

fig = plt.figure(num=self.__company_name + ' normalized prices')
ax = plt.gca()
fig.set_size_inches(12, 6)
y_low, y_high = 0, 0
for year in range(start_year, end_year + 1):
    y = self.__prices_data[year]
    if y_low >= min(y):
        y_low = min(y)
    if y_high <= max(y):
        y_high = max(y)
    x = np.linspace(0, len(y), len(y))
    plt.plot(x, y, alpha=.95, label=year)
    plt.legend(bbox_to_anchor=(1.01, 1), loc=2, borderaxespad=0.)

y_max = max(abs(y_low) - 1, abs(y_high) + 1)
x_min, x_max = -10, self.__max_days + 10
ax.set_ylim(bottom=-y_max, top=y_max)
ax.set_xlim(left=x_min, right=x_max)

for i in range(0, 5):
    plt.vlines(x=(self.__max_days / 4) * i, ymin=-y_max, ymax=y_max,
color='black', linestyle='--', alpha=.6,
zorder=-1)
    if i < 4:
        ax.text((self.__max_days / 4) * i + self.__max_days / 8 - 5,
y_max - 0.5, self.__quarters[i],
fontSize=12)
    plt.hlines(y=0, xmin=x_min, xmax=x_max, color='black',
linestyle='--', alpha=.6, zorder=-1)

plt.grid(True, alpha=.25)
plt.title(self.__company_name)
```

```
plt.xlabel('Days')
plt.ylabel('Normalized price')

plt.tight_layout()

fname = '{}_{}_{}_normalized_prices.png'.format(self.__company_name,
start_year, end_year)
fig.savefig(fname, dpi=fig.dpi)
plt.clf()

def show_gp_prediction(self, train_start: int, train_end: int, pred_year:
int, pred_quarters: list = None):
    self.__validate_dates(start_year=train_start, end_year=pred_year)

    prices = self.__prices_data[pred_year]
    prices = prices[prices.iloc[:].notnull()]

    fig = plt.figure(num=self.__company_name + ' prediction')
    ax = plt.gca()
    fig.set_size_inches(12, 6)

    x_obs = list(range(prices.index[0], prices.index[-1] + 1))
    x_mesh, y_mean, y_cov =
self.__gpr.get_eval_model(start_year=train_start, end_year=train_end,
                           pred_year=pred_year,
pred_quarters=pred_quarters)
    y_lower = y_mean - np.sqrt(np.diag(y_cov))
    y_upper = y_mean + np.sqrt(np.diag(y_cov))
    y_max = max(abs(min(y_lower) - 1), abs(max(y_upper) + 1))
    ax.set_ylim(bottom=-y_max, top=y_max)

    x_min, x_max = -10, self.__max_days + 10
    ax.set_xlim(left=x_min, right=x_max)
```

```
plt.plot(x_obs, prices, color='#006699', alpha=.95,
label=u'Observations ' + str(pred_year), zorder=10)
plt.plot(x_mesh, y_mean, color='#ff0066', linestyle='--',
label=u'Prediction')
plt.fill_between(x_mesh, y_lower, y_upper,
alpha=.25, label='95% confidence', color='#ff0066')

handles, labels = plt.gca().get_legend_handles_labels()
new_labels, new_handles = [], []
for handle, label in zip(handles, labels):
    if label not in new_labels:
        new_labels.append(label)
        new_handles.append(handle)
plt.legend(new_handles, new_labels, bbox_to_anchor=(0.01, 0.02),
loc='lower left', borderaxespad=0.)

for i in range(0, 5):
    plt.vlines(x=self.__quarter_length * i, ymin=-y_max, ymax=y_max,
color='black', linestyle='--', alpha=.6,
zorder=-1)
    if i < 4:
        ax.text(self.__quarter_length * i + self.__quarter_length / 2
- 5, y_max - 0.5, self.__quarters[i],
fontSize=12)
    plt.hlines(y=0, xmin=x_min, xmax=x_max, color='black',
linestyle='--', alpha=.6, zorder=-1)

plt.grid(True, alpha=.25)
plt.title(self.__company_name)
plt.xlabel('Days\n')
plt.ylabel('Normalized price')

plt.tight_layout()
```

```
fname = '{}{}_prediction.png'.format(self.__company_name, pred_year)
fig.savefig(fname, dpi=fig.dpi)
plt.clf()

def show_whole_time_series(self, intermediate: bool = False):
    self.show_time_series(start_year=self.__years[0],
end_year=self.__years[-1], intermediate=intermediate)

def show_time_series(self, start_year: int, end_year: int, intermediate:
bool = True):
    self.__validate_dates(start_year=start_year, end_year=end_year)

    prices_data =
self.__company_handler.get_whole_prices(start_year=start_year,
end_year=end_year)

    fig = plt.figure(num=self.__company_name + ' prices')
    fig.set_size_inches(12, 6)
    plt.plot(prices_data.iloc[:, 0], prices_data.iloc[:, 1],
color='#006699', alpha=.95,
            label=u'Observations ' + str(start_year) + '-' +
str(end_year), zorder=10)
    ax = plt.gca()

    x_ticks = []
    for year in range(start_year, end_year + 2):
        if year == end_year + 1:
            current_date = prices_data[prices_data['Date'].dt.year ==
end_year].iloc[-1, 0]
        else:
            current_date = prices_data[prices_data['Date'].dt.year ==
year].iloc[0, 0]
        x_ticks.append(current_date)
```

```
x_formatter = mdates.DateFormatter('%d-%m-%Y')
ax.xaxis.set_major_formatter(x_formatter)
if not intermediate:
    x_ticks = [x_ticks[0], x_ticks[-2], x_ticks[-1]]
    ax.set_xticks([x_ticks[0], x_ticks[-1]])
else:
    ax.set_xticks(x_ticks)
plt.xticks(rotation=20)
y_min, y_max = ax.get_ylim()
x_min, x_max = ax.get_xlim()
ax.set_ylim(bottom=y_min, top=y_max)
ax.set_xlim(left=x_min, right=x_max)

for i in range(0, len(x_ticks)):
    plt.vlines(x=x_ticks[i], ymin=y_min, ymax=y_max, color='black',
linestyles='--', alpha=.6,
              zorder=-1)

plt.grid(True, alpha=0.25)
plt.legend()
plt.title(self.__company_name)
plt.ylabel('Price')

plt.tight_layout()

fname = '{}_{}_{}_prices.png'.format(self.__company_name, start_year,
end_year)
fig.savefig(fname, dpi=fig.dpi)
plt.clf()

def __validate_dates(self, start_year: int, end_year: int):
    if start_year < self.__years[0] or end_year > self.__years[-1]:
        raise ValueError('\n' +
```



```
        'Input years out of available range! \n' +  
        'Max range available: {}-{}\n'  
n'.format(self.__years[0], self.__years[-1]) +  
        'Was: {}-{}'.format(start_year, end_year))
```

4. Gp_wrapper.py

```
import numpy as np  
import pandas as pd  
from sklearn.gaussian_process import GaussianProcessRegressor  
from sklearn.gaussian_process.kernels import RBF  
  
import data_handler  
  
class Wrapper:  
    __company_data = None  
    __prices_data = None  
    __quarters = None  
    __max_days = None  
    __alpha = None  
    __iterations = None  
    __kernels = None  
    __gp = None  
  
    def __init__(self, company_name: str):  
        self.__company_data = data_handler.CsvHandler(company_name)  
        self.__prices_data = self.__company_data.get_equal_length_prices()  
        self.__quarters = self.__company_data.quarters  
        self.__years = self.__company_data.years  
        self.__max_days = self.__company_data.max_days  
  
        kernel = 63 * RBF(length_scale=1)  
        self.__alpha = 1e-10
```

```
self.__iterations = 10
self.__kernels = [kernel]
self.__gp = GaussianProcessRegressor(kernel=self.__kernels[0],
alpha=self.__alpha,

n_restarts_optimizer=self.__iterations,

                                normalize_y=False)

def get_eval_model(self, start_year: int, end_year: int, pred_year: int,
pred_quarters: list = None):
    years_quarters = list(range(start_year, end_year + 1)) + ['Quarter']
    training_years = years_quarters[:-2]
    df_prices =
self.__prices_data[self.__prices_data.columns.intersection(years_quarters)]

    possible_days = list(df_prices.index.values)
    X = np.empty([1,2], dtype=int)
    Y = np.empty([1], dtype=float)

    first_year_prices = df_prices[start_year]
    if start_year == self.__company_data.years[0]:
        first_year_prices = (first_year_prices[first_year_prices.iloc[:] !
= 0])
        first_year_prices = (pd.Series([0.0],
index=[first_year_prices.index[0]-1])).append(first_year_prices)

    first_year_days = list(first_year_prices.index.values)
    first_year_X = np.array([[start_year, day] for day in
first_year_days])

    X = first_year_X
    Y = np.array(first_year_prices)
    for current_year in training_years[1:]:
        current_year_prices = list(df_prices.loc[:, current_year])
```

```

        current_year_X = np.array([[current_year, day] for day in
possible_days])
        X = np.append(X, current_year_X, axis=0)
        Y = np.append(Y, current_year_prices)

    last_year_prices = df_prices[end_year]
    last_year_prices =
last_year_prices[last_year_prices.iloc[:].notnull()]

    last_year_days = list(last_year_prices.index.values)
    if pred_quarters is not None:
        length = 63 * (pred_quarters[0] - 1)
        last_year_days = last_year_days[:length]
        last_year_prices = last_year_prices[:length]
        last_year_X = np.array([[end_year, day] for day in last_year_days])

    X = np.append(X, last_year_X, axis=0)
    Y = np.append(Y, last_year_prices)

    if pred_quarters is not None:
        pred_days = [day for day in
                        range(63 * (pred_quarters[0]-1), 63 *
pred_quarters[int(len(pred_quarters) != 1)])]
    else:
        pred_days = list(range(0, self.__max_days))
    x_mesh = np.linspace(pred_days[0], pred_days[-1]
                        , 2000)
    x_pred = ([[pred_year, x_mesh[i]] for i in range(len(x_mesh))])

    self.__gp = self.__gp.fit(X, Y)
    self.__kernels.append(self.__gp.kernel_)

    y_mean, y_cov = self.__gp.predict(x_pred, return_cov=True)

```

```
        return x_mesh, y_mean, y_cov

    def get_kernels(self):
        return self.__kernels
```

5. Data_presenter.py

```
import os
import data_plotter

companies = []
plotters = {}
start_year = 2008

def main():
    __init_data()
    for company in companies:
        make_summary(company)
    print("Done!")

def make_summary(company_name):
    plotter = plotters[company_name]

    plotter.show_whole_time_series()
    plotter.show_time_series(start_year=start_year, end_year=2016)
    plotter.show_preprocessed_prices(start_year=start_year, end_year=2016)
    plotter.show_gp_prediction(train_start=start_year, train_end=2016,
pred_year=2017)
    plotter.show_time_series(start_year=start_year, end_year=2018)
    plotter.show_gp_prediction(train_start=start_year, train_end=2018,
pred_year=2018, pred_quarters=[3, 4])
    print(company_name + ' summary done!')
```

```
def __init_data():
    for company in os.listdir('Data'):
        current_company = company.split('.')[0]
        companies.append(current_company)
        plotters[current_company] =
(data_plotter.Plotter(company_name=current_company))

if __name__ == "__main__":
    main()
```

Acknowledgements

I would like to sincerely appreciate certain personages who have continued to offer all of their unconditional love, support and succor to myself during the course of this research endeavor.

I am also extremely full of gratitude to my thesis supervisor, Dr. Weidong Wang who is currently at the University of Electronic science and Technology China, for his time and patience and willingness shown in trying to help me where I had problems to ensure the successful resolution of this endeavor. I also want to thank my friends for their help in pushing me and making sure I didn't give up and even offering help when I needed it.

Lastly, I want to also sincerely express my gratitude to my parents and other unmentioned family members for their continued financial and upstanding support and also for their constant motivation and support during this period and above all I would like to thank God Almighty for His favor, help, goodwill and sustenance throughout the duration and entire process of the writing and completion of this project.

References

1. C. E. Rasmussen and C. K. I. Williams. Gaussian Processes for Machine Learning. MIT Press, 2006.
2. Bengio, Chapados, Forecasting and Trading Commodity Contract Spreads with Gaussian Processes, 2007.
3. Correa, Farrell, Gaussian Process Regression Models for Predicting Stock Trends, 2007, <https://github.com/gdroguski/GaussianProcesses>.
4. Zexun Chen, Gaussian process regression methods and extensions for stock market prediction, 2017.
5. Correa, Farrell, Gaussian Process Regression Models for Predicting Stock Trends, 2007.
6. <https://stats.stackexchange.com/questions/470261/linear-regression-from-a-weight-space-view-gaussian-process>
7. <https://distill.pub/2019/visual-exploration-gaussian-processes/>
8. <https://andrewcharlesjones.github.io/posts/2020/11/gaussian-processes/>
9. https://deisenroth.cc/teaching/2018-19/foundations-of-machine-learning/GPSS_Lab1_2018.ipynb
10. https://deisenroth.cc/teaching/2018-19/foundations-of-machine-learning/lecture_gaussian_processes.pdf
11. <http://inverseprobability.com/talks/notes/gaussian-processes.html>
12. GPy: A gaussian process framework in python. <https://gpy.readthedocs.io/en/deploy/>.
13. Fitting Gaussian Process Models in Python, March 8, 2017, written by Chris Fonnesbeck, Assistant Professor of Biostatistics, Vanderbilt University Medical Centre.