Final Year Project Report

Full Unit - Final Report

Regression Algorithms for Learning

Ahmet Cihan

A report submitted in part fulfilment of the degree of

BSc Hons Computer Science and Mathematics

Supervisor: Luo Zhiyuan

Department of Computer Science
Royal Holloway, University of London

April 6, 2024

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 23000

Student Name: Ahmet Cihan

Date of Submission: 12/05/2024

Signature: Ahmet Cihan

Table of Contents

Project Specification 4

Abstract 5

1 Introduction 6

1.1 The Problem 6

1.2 Introduction to Machine Learning 7

1.3 Aims and Goals of the Project 9

1.4 Datasets 10

2 Linear Regression 13

2.1 Introduction to Linear Regression 13

2.2 Least Squares Estimate 13

2.3 Applying on Machine Learning 15

2.4 Assessing the Model 15

2.5 Multiple Linear Regression 16

2.6 Application of Multiple Linear Regression 17

2.7 Comparison of Results 19

3 Ridge Regression 21

3.1 Introduction to Ridge Regression 21

3.2 Ridge Regression Compared to Linear Regression 21

3.3 Application of Ridge with Inverted Matrices 25

3.4 Results 29

4 Gradient Descent Method 31

4.1 Cost Function 31

4.2 Gradient Descent 31

4.3 Data Normalization 34

4.4 Computational Complexity . . . 35
4.5 Application of Gradient Descent . . . 36
5 Lasso Regression . . . 38
5.1 Ridge Regression vs Lasso Regression . . . 38
5.2 Application of Lasso . . . 39
5.3 Tuning Parameter Selection . . . 40
5.4 Extended Boston Housing Dataset . . . 42
5.5 Results . . . 47
6 Inductive Conformal Prediction and Cross-Conformal Prediction . . . 52
6.1 Conformal Prediction . . . 52
6.2 Inductive Conformal Prediction Application . . . 53
6.3 Inductive Conformal Prediction Interval Accuracy . . . 55
6.4 Cross-Conformal Prediction . . . 56
6.5 Application of Cross-Conformal Prediction . . . 57
6.6 Cross-Conformal Prediction Accuracy . . . 58
6.7 Results . . . 60
7 Software Engineering Tools and Techniques . . . 64
7.1 Overview . . . 64
7.2 Software Development Methodologies . . . 64
7.3 Version Control System . . . 65
7.4 Documentation and Coding Standards . . . 65
7.5 Tools and Libraries . . . 66
7.6 Testing . . . 66
8 Professional Issues . . . 67
Bibliography . . . 68
Appendix . . . 72
User Manual . . . 73

Project Specification

Aims: The aim of the project is to implement a popular machine learning technique called Ridge regression, apply it to real-world data, analyse its performance, analyse its performance, and compare it against other methods. **Background:** Regression algorithms play an important role in machine learning. They allow us to learn dependencies between multidimensional attributes and continuous outcomes. Regression algorithms can be applied to different domains.

An example is the Boston housing problem. The Boston housing database consists of records describing houses in Boston (to be precise, average houses in certain neighbourhoods). For each house the values of attributes such as the number of rooms, distance from the city centre, quality of schools etc are known and the price, as evaluated by an expert estate agent, is given. The goal of the learner is to determine the dependency between attributes and prices and to predict the price of a house using the values of the attributes. The program is first shown a number of training examples so that it could learn the dependency and then is tested on test examples. Ridge Regression is known to perform well on this dataset.

Another example is time series data. A significant amount of real world data is temporal in nature, in that the values of the variables are sampled at multiple points over some time period, so that the data stored for each entity has an additional 'time' dimension. Such data are called time series. Examples of time series are the daily closing value of the FTSE 100 index or network traffic measurement on a router. Time series forecasting is the use of a model to predict future values based on previously observed values and regression algorithms can be used here.

Applying a learning algorithm to data is not a straightforward task. One needs to preprocess the data and choose parameters to optimise the performance. The algorithms need to be implemented in such a way so as to run fast and not to suffer from numerical error. Making sense of the results and comparing different algorithms in a fair meaningful way can be tricky. Does one algorithm outperforms another consistently, or is it a mere coincidence?

Early Deliverables

Report:

1. An overview of Ridge regression describing the concepts 'training set' and 'test set', giving the formulas for a regression algorithm and defining all terms in them.
2. Proof of concept program: a regression algorithm applied to a small artificial dataset.
3. Report: Examples of applications of Ridge Regression worked out on paper and checked using the prototype program.
4. Loading a real-world dataset, simple pre-processing and visualisation.

Final Deliverables

- The program must have a full object-oriented design, with a full implementation life cycle using modern software engineering principles.
- The program will work with a real-world dataset, read the data from a file, preprocess the data and apply regression using parameters selected by the user with parameters entered by the user.

- The program will have a graphical user interface.
- The program will automatically perform tests such as comparison of different kernels, parameters, etc and visualise the results.
- The program will implement another learning algorithm such as nearest neighbours or neural networks, and compare regression against it.
- The report will describe the theory of regression and derive the formulas.
- The report will describe the implementation issues (such as the choice of data structures, numerical methods etc) necessary to apply the theory.
- The report will describe the software engineering process involved in generating your software.
- The report will describe computational experiments with different kernels and parameters and draw conclusions.
- The report will describe the competitor algorithm(s), compare the performance and draw conclusions.

Suggested Extensions

- Use of regression in the on-line mode with growing and sliding window.
- Application of regression to different datasets (including those found by the student).
- Implementation of regression-type algorithms such as Kernel Aggregating Algorithm Regression, Kernel Aggregating Algorithm Regression with Changing Dependencies etc and computational experiments with them.

Abstract

This project presents a comparative analysis of regression algorithms applied to the prediction of housing prices. By exploring various machine learning techniques, such as Linear Regression, Ridge Regression, and Lasso Regression, this study seeks to understand their effectiveness in handling multicollinearity and feature selection within different housing datasets. The algorithms are theoretically explained to comprehend their mathematical foundations. The algorithms are applied on practical applications to real-world datasets, including the New York, Washington, Boston, and Extended Boston Housing datasets, as well as the California Housing dataset. This project also delves into the concept of Conformal Prediction, proving its utility in generating prediction intervals with predetermined confidence levels.

Chapter 1: **Introduction**

1.1 The Problem

From the point we wake up every day and take our phones in our hands, we are dealing with data. As we have transitioned into the "Age of Ubiquitous Computing"[17], our daily lives are intertwined with technology, from computers to the internet, and other connected devices. We end up living in a world full of data, leaving traces everywhere we go. These footmarks we all leave end up piling up somewhere we do not see. To make sense of this vast amount of data, we as Computer Scientists, use certain approaches to make inferences out of them. Thus, we end up understanding more of users' behaviours, and this gives more value to this pile of data every day we all create.

Companies run after the data we create to make their business more valuable. Netflix and other streaming services recommend movies or shows depending on our watch history. Twitter, also known as "X" uses our data to show us similar tweets that we liked or interacted with before. This is to pull more users and make it more personal for every user. Other than that Twitter also uses relevant ads depending on your clicks, to make advertisements stronger and grow the company's business value. This can only be done by us; the users and the data we create every time we connect.

In today's computer world, software faces problems and issues every day. In order to solve problems for a program, we can do debugging, or write new algorithms. Algorithms follow the logical structure of a code and solve the problem by applying conditional statements and loops. This could be sorting a list of numbers or finding the minimum value in big data of numbers. As strong as algorithms can get, they also have weaknesses in some of the problems. An example of this can be detecting spam mail. The spam mail issue has been going on for decades now and there is no specific algorithm that can figure out if an email is spam or not. Instead of algorithms that are focused on emails, Computer Scientists have created Machine Learning algorithms, teaching computers what spam mail is and acting accordingly for future cases. "There is no need to learn to sort numbers since we already have algorithms for that, but there are many applications for which we do not have an algorithm but have lots of data."[4] said Ethem Alpaydin in his book clearly stating the difference between algorithms and machine learning.

When we have data, what we also have is patterns and regularities that are happening in the data. Given a house pricing dataset, we can see a pattern when the bedroom numbers increase the price also increases. This could be a simple example of a pattern, but can we always say that a four-bedroom house is always more expensive than two- or three-bedroom houses? However, this may not always be true, we can only make approximations of what affects the house prices. Even though we cannot have 100 percent correct predictions, with Machine Learning we enhance our understanding of how parameters affect our results and predictions. Businesses rely on machine learning algorithms to grow their sales and improve customer relations and employee performance. These algorithms are very useful to businesses as they do not require a lot of investment and the returns are much more valuable. After being built, the ideal system should be able to learn any changes that are made and thus the engineer does not have to interfere future on [4]. This makes machine learning so popular and useful for businesses and companies. Take GPT as an illustrative example. The application is built in a way where we can ask GPT itself to fix any prior mistakes that were made during previous answers. As it is still not the best, we can say it is the best Machine Learning application that is accessible publicly.

1.2 Introduction to Machine Learning

Machine Learning can be thought of as a system that learns from data. It is an automated version of statistical inferences, but it also goes beyond statistics with certain algorithms such as K-Nearest Neighbours and Neural Networks etc. The main idea is to learn using past examples and making statistical sense. This could be as simple as having data for handwritten numbers [20]. Using certain algorithms, we can teach the computer what number is written. This will be done by using past examples of handwritten numbers and labelling them as their according number. A part of this simple dataset with labels will be used for training to teach the computer and another part to test the algorithm. This process with labelled datasets is called supervised learning in Machine Learning. Contrary to this learning method, unsupervised learning is used to train the algorithm with a dataset that does not have any labels [28]. Using supervised learning the computer tries to make inferences from patterns and regularities. This project will focus on supervised learning methods, such as Linear Regression, Ridge Regression, and Lasso Regression.

1.2.1 Introduction to Linear Regression

The Linear Regression stands as a fundamental approach to conducting statistical calculations. Although it is outdated, it is still one of the most powerful tools to this day. The simple Linear Regression equation is;

$$Y = \beta_0 + \beta_1 X$$

Where β_0 and β_1 are the intercept and the slope respectively. Y is the dependent variable and X is the independent variable. In simple terms we are trying to figure out Y by using the predictor variable X [16]. The method involves estimating the parameters β_0 and β_1 by minimizing the least squares criterion [37]. This paper will have a simple algorithm for linear regression. This will be the first algorithm for the project and a basis to compare the rest of the algorithms. Despite its simplicity, linear regression may not be the most reliable algorithm of all. There could be overfittings in complex datasets and the algorithm will lack regularization. Thus, the simple linear regression does not have any way to control complexity [20]. When we are dealing with complex data, it is easy to over-train our model and thus we end up having a model that the predictions do not apply to the real world. This is called overfitting, when we are training our data, we should avoid applying every small feature to our model. There are also other ways to mitigate this issue we face in linear regression [21].

1.2.2 Introduction to Ridge Regression

Ridge regression, a popular machine learning algorithm, used for analysing multicollinearity in data, provides mitigations to linear regression problems. Ridge Regression is estimated the same as Linear Regression, ordinary least squares. However, to overcome multicollinearity in data, the tuning parameter, λ is introduced to the equation. Having λ in the equation helps us shrink the coefficient β_1 and thus our slope is less likely to cause overfittings. There is no specific method to estimate λ. It depends on the dataset to determine the λ and it will not be the same for every dataset. When the λ is equal to 0, there is no shrinkage, thus the calculation we make will be no different than the original linear regression least squares estimate. However, as the λ approaches infinity, we can see that β_1 will also approach 0[16]. This will mean that the variance will go lower as we increase the λ thus, we can prevent overfitting which happens to be the problem in Linear Regression. When we apply the

simple linear regression algorithm to a real-world dataset, take the Boston Housing dataset for an example, we will end up having a high training score, but when it comes to the test score this will not be the case and it will be much lower. If we add a tuning parameter to our algorithm and turn it into Ridge Regression, we should have a lower training score, but our test score will improve[20].

1.2.3 Introduction to Lasso Regression

Lasso Regression, another important machine learning method is a useful tool to address multicollinearity and feature selection. Similar to Ridge Regression, Lasso (Least Absolute Shrinkage and Selection Operator) Regression, introduces a penalty term λ to shrink the model coefficients. The key difference between Lasso Regression and Ridge Regression is that the penalty term will not only shrink the parameters but will also zero out some of the coefficients. This property gives Lasso Regression an advantage against Ridge Regression, specifically when the number of features in the dataset is quite large[16]. This can be observed further in *Section 5.4*, where Lasso Regression is compared to Ridge Regression using the Extended Boston Housing dataset.

Furthermore, selecting the value of λ is also crucial for Lasso Regression. The optimal value will be different for each dataset. When λ is set to 0, the model will become the Linear Regression or Least Squares Estimate model. Similar to Ridge, a large λ value will lead to excluding more coefficients[16]. Thus, the model will be simplified and could mitigate multicollinearity where Ridge Regression is not performing well.

1.2.4 Introduction to Conformal Prediction

What if, we would like to be certain about predictions based on a predetermined probability? How can we be sure that the model's prediction is correct, can we get certainty in predictions? Conformal Prediction, founded by Vladimir Vovk and A. Gammerman, is a way of generating prediction intervals depending on confidence levels. For a 90% confidence level, the prediction intervals that are generated will have at most a 10% error rate. The higher the confidence level, the larger our prediction intervals will get. Conformal Prediction method is a very useful method to make predictions based on certainty. It can be used in many sectors, including health, finance, realtor, etc[35]. In this project, it will be used to determine a price range of a house with a given certainty.

1.2.5 Further Algorithms and Methods

Other than Linear Regression, Ridge Regression, Lasso Regression and Conformal Prediction, other learning algorithms that are a good benchmark to compare are K-Nearest Neighbours(K-NN) and Neural Networks. In simple terms, we can think of the K-NN algorithm as asking your neighbours for more information about the neighbourhood. K-NN provides us with data points. Selecting one datapoint, we try to make inferences from its nearest K datapoints. It is very simple in practice; however, its simplicity comes with trade-offs, the algorithm lacks time efficiency, and it may not create optimal models for datasets with many features[20]. Linear models, such as Linear Regression and Ridge Regression perform better with datasets that have many features.

What about when the dataset we have is not linear or we are dealing with non-linear problems? Neural Networks (**NN**) are a good way to deal with non-linear and complex datasets.

In Computer Science, the mimic of how brains work is Neural Networks. Similar to our brain, Neural Networks has interconnected nodes(neurons). These connections are like layers, helping the computer to learn from data. In a simple **NN** there are three layers. Input layers, hidden layers, and output layers. Input layers are usually basic information regarding the data, compared to the hidden layer it is less complex. Hidden layers have much more complexity and data, which can analyse the data and make inferences from the given input. Hidden layers have a series of nodes which filter the input to get output. We can think of this as the process of seeing a red round object and trying to make a difference whether it's a tomato or an apple. In this case, our vision could be the input, and the smell, texture and taste could be hidden layers which will help us to understand if it is a tomato or an apple. Last but not least, initially the **NN** are prone to making mistakes. As the logic is similar to the brain, the **NN** will also learn from its mistakes and start correcting them by adjusting its nodes to make better decisions for future cases [9].

1.3 Aims and Goals of the Project

The primary aim of this project is to compare the performance of Linear Regression, Ridge Regression, and Lasso Regression across housing datasets. The machine learning techniques will be analysed further by applying different methods to create learning algorithms. A matrix inversion approach will be taken to create the Linear Regression and Ridge Regression algorithms. The Gradient Descent will be introduced to apply Lasso Regression. Furthermore, Conformal Prediction is another method that is applied in this project. Analysis of conformal predictions can be made by combining the regression algorithms that are walked through.

Observing these algorithms theoretically, we can see which ones may perform better than others. Specific goals include evaluating the effectiveness of Ridge Regression in handling multicollinearity. We know for a fact that the Ridge Regression will perform better with data that has multicollinearity. In addition to Ridge Regression, Lasso Regression also aims to mitigate the multicollinearity issue by not only shrinking but also setting the coefficients to zero. This becomes particularly useful when dealing with datasets that have a large number of features. R^2 and MSE values can be compared between datasets and models to analyse the models and their accuracy.

Moreover, the Gradient Descent method introduces an efficient way to optimize the cost function. Gradient Descent is a computationally efficient alternative to the matrix inversion method used in Least Squares Estimate. Gradient Descent method iteratively adjusts the model coefficients to minimize the cost function. This is beneficial when dealing with large datasets, where the computational cost of matrix inversion is too costly. This can be shown by applying the Gradient Descent to an artificial dataset that is in the dimension of nxn and comparing its efficiency to the matrix inversion.

Furthermore, Inductive Conformal Prediction and Cross Conformal Prediction is a way to make predictions based on a given probability threshold. By generating prediction intervals, Inductive Conformal Prediction provides a measure of certainty. Conformal Prediction is vital for practical applications where certainty of the prediction is crucial. In the case of Conformal Predictions, the width of the intervals is valuable for analysis. The width of the intervals can compared between Inductive Conformal Prediction and Cross Conformal Prediction.

All these algorithms and methods have their pros and cons. By assessing these algorithms and methods in various data-driven scenarios, this project seeks to provide valuable insights

into machine learning algorithms. Ultimately, applying real-world datasets to compare and analyse the differences these algorithms have will be the main part of this project.

We must understand we live in a world of data. The people and the businesses who understand the power of data, succeed also in life and business. With Machine Learning we can automate our process for understanding data, as data itself alone does not have much meaning to it. To apply machine learning techniques, depending on what kind of data we have different approaches can be taken. These machine-learning algorithms help us build a future, depending on the past. Our lives become easier thanks to these algorithms. This project will harness the power of advanced machine-learning techniques and pave the way for innovations that can transform our approach to problem-solving and decision-making.

1.4 Datasets

Throughout this report, various datasets are used as part of training and testing the models that are created. For future reference, the datasets are introduced and explained in this section.

New York Housing: A simple dataset, it can be used to predict the housing price based on certain factors like house area, bedrooms, furnished, nearness to the main road, etc.

This dataset has **12** features and target variable being the price:

1. Area of a house
2. Number of bedrooms
3. Number of bathrooms
4. Stories
5. Mainroad
6. Guestroom
7. Basement
8. Hot water heater
9. Airconditioning
10. Parking
11. Prefarea
12. Furnishing Status

This dataset has **545** samples. The dataset is obtained from Kaggle.

Washington Housing: The dataset that is used in *Chapter 2.7*, *Figure 2.5* is the Washington Housing dataset. The dataset has 12 features and the target variable is the price:

1. Number of bedrooms
2. Number of bathrooms,

3. Living room square feet area
4. The square feet of the lot
5. Number of floors
6. If the lot is waterfront or not
7. Rating of the view
8. Condition of the house
9. The square feet of the house above ground floor
10. The square feet of the basement
11. Year it was built
12. Year it was renovated

This dataset has **4600** samples. This dataset is obtained from Kaggle.

Boston Housing Dataset: This dataset is the `sklearn` library's `load_boston` dataset. The Boston Housing dataset has 13 features and the target variable is the median price in $1000's:

1. CRIM - per capita crime rate by town
2. ZN - the proportion of residential land zoned for lots over 25,000 sq. ft.
3. INDUS - proportion of non-retail business acres per town.
4. CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)
5. NOX - nitric oxides concentration (parts per 10 million)
6. RM - average number of rooms per dwelling
7. AGE - proportion of owner-occupied units built before 1940
8. DIS - weighted distances to five Boston employment centres
9. RAD - index of accessibility to radial highways
10. TAX - full-value property-tax rate per $10,000
11. PTRATIO - pupil-teacher ratio by town
12. B - $1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town
13. LSTAT - % lower status of the population

The description of this dataset is obtained from http://lib.stat.cmu.edu/datasets/boston[31]. The dataset has been widely used in literature to test the benchmark of algorithms. The dataset has **506** samples. The order of the samples is also mysterious.

Extended Boston Housing: This dataset is widely used in this project. This dataset is based on the `sklearn` library's `load_boston` dataset.

The Extended Boston Housing is created by taking the simple Boston Housing dataset and applying feature scaling with the `MinMaxScaler`. `MinMaxScaler`, scales the features in between the range of values of 0 and 1. Feature scaling is a very common and important practice

in Machine Learning as it normalizes the effects of the features so that the range of varying values does not have a disproportionate effect on the model[1].

Moving forward, to make the dataset even more complex, which is useful for Ridge Regression and Linear Regression comparison, *Polynomial Feature Generation* is used to make the dataset more complex by combining all the features with the specified degree of 2. Thus, the dataset has **104** features. The complexity added helps to capture a non-linear relationship between the features[16]. This approach generates new features by squaring each future and creating products of all pairs of features, thereby creating new features depending on the obtained values.

California Housing: This dataset is the California Housing dataset, where there are 8 features and the price is the target variable:

1. MedInc - median income in block group
2. HouseAge - median house age in block group
3. AveRooms - average number of rooms per household
4. AveBedrms - average number of bedrooms per household
5. Population - block group population
6. AveOccup - average number of household members
7. Latitude - block group latitude
8. Longitude - block group longitude

The dataset has **20640** samples. The target variable is the median house value for California districts, expressed in hundreds of thousands of dollars($100,000). The information about the description of the California Housing dataset is obtained by using the `california.DESCR` attribute of the dataset object.

Chapter 2: **Linear Regression**

Chapter 2 will delve into Linear Regression a foundational building stone of the regression algorithms for learning. This chapter aims to cover the layers of Linear Regression from understanding its theory, mathematical complexities and real-world applications. We will begin by familiarizing ourselves with the core concepts that form the Linear Regression, gradually building its implementation in machine learning environments. Furthermore, this chapter will compare the results of the implemented algorithm to the one of `sklearn` library's algorithm, proving accuracy.

2.1 Introduction to Linear Regression

A statistical approach, *Linear Regression* has been in our lives for centuries. As it is important to statistics it is also simple. We can have two variables, X and Y. We don't have information about predicting Y but what we know is that X knows Y more than us. Using X we can make inferences about Y, even make predictions. In this context, we can call X the predictor [6]. Take local *temperature*(Celcius) in weather has an effect on *fires* for an example. We can use the *temperature* in a period of time t as the predictor variable or X, and percentage of *fires* could happen. We don't have any predictions about the percentage of fires. Depending on the temperature and some previous percentages, we will predict what could be the percentage of fire. This is a simple case of Linear Regression. We can use the *Least Squares Estimate* method to predict the percentages of fire occurring by looking at the temperature data and the percentage data.

2.2 Least Squares Estimate

Figure 2.1. The Least Squares Estimate is a statistical method to find a line that fits the best for a set of data. This line is found in a way so that the difference between the sums of squares is minimised between the line and the data points. This method also can be thought of as a way to minimize the total "error" between the line and the actual data points. The *Table 2.1* shows us a set of temperature and fire percentage data. We can see that there is

Table 2.1: Canakkale Temperature and Fire Percentage

Temperature(X)	Percentage(Y)
25°C	30%
32°C	40%
30°C	39%
40°C	70%
37°C	66%

a tendency to increase when the temperature increases. Using the Least Squares Estimate method, our goal is to find a line that fits this data the best. By this line, we will have an idea to predict the values of Y. The simplest straight line we know is;

$$Y = \beta_0 + \beta_1 X$$

To find the best-fit line for this data, we have to estimate the slope, β_1 and the intercept, β_0. To find the intercept β_0, we have to use the data we have. The formula to find the slope, β_1

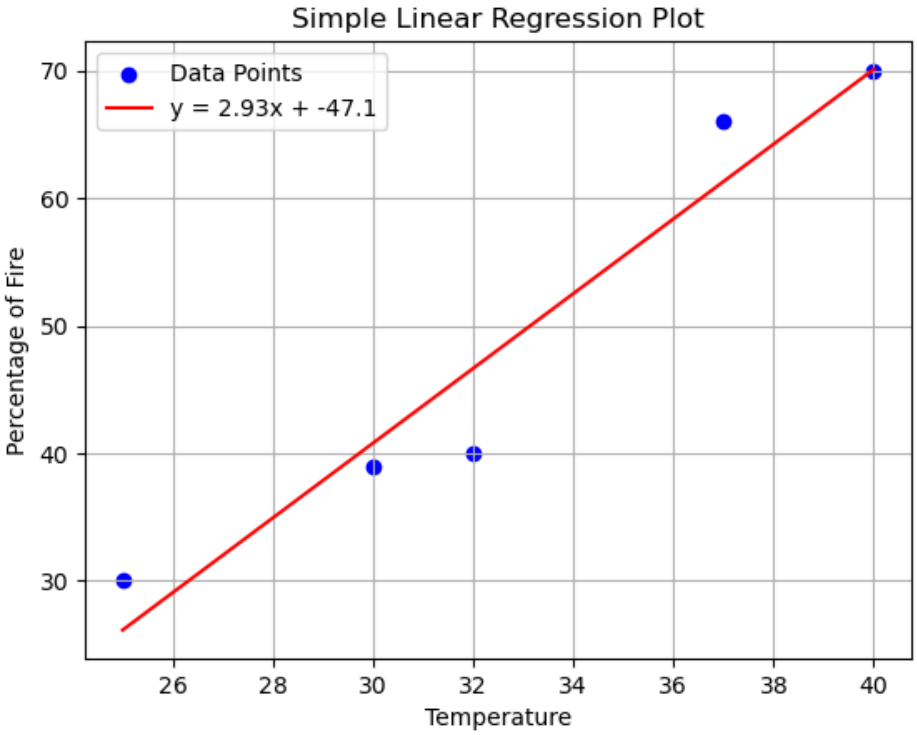

Figure 2.1: Fire Percentages Depending on Temperature

is[6];

$$\beta_1 = \frac{n(\sum_{i=1}^{n} x_i y_i) - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n(\sum_{i=1}^{n} x_i^2) - (\sum_{i=1}^{n} x_i)^2}$$

where:

- n is the number of data points.
- $\sum_{i=1}^{n} x_i y_i$ is the sum of the product of each pair in the dataset.
- $\sum_{i=1}^{n} x_i$ is the sum of all x values.
- $\sum_{i=1}^{n} y_i$ is the sum of all y values.
- $\sum_{i=1}^{n} x_i^2$ is the sum of squares of the x values.

The formula to figure what the intercept, β_1 is;

$$\beta_1 = \frac{\sum_{i=1}^{n} y_i - \beta_0(\sum_{i=1}^{n} x_i)}{n}$$

Using the formulas above let's create a best-fit line for our data. We know that $\sum_{i=1}^{n} x_i = 164$, $\sum_{i=1}^{n} y_i = 245$, $\sum_{i=1}^{n} x_i y_i = 8442$, $\sum_{i=1}^{n} x_i^2 = 5518$ and $n = 5$. Having these data we can calculate that, $\beta_1 \approx 2.93$, and $\beta_0 \approx -47.10$. The best-fit line we have is;

$$Y = 2.93x - 47.10$$

We can see the plotted graph of our best-fit line and our data points on the *Figure 2.1*.

In the *Figure 2.1*, the red dots represent the observed values of the predicted variable Y, while the blue line is the best-fit line for our regression. From looking at the graph it can be inferred that our line is a great fit for our predicted variables. As the *Least Squares Estimate* method minimizes the "*"error"* between the predictor variable and the best-fit line, it can be said that the observed values of predicted variables Y are very close to the line. Thus our model aptly predicts the predicted variable Y, based on the predictor variable X.

2.3 Applying on Machine Learning

For many years Scientists and Statisticians have applied Linear Regression to any kind of data. As the usage of computers grew, it became easier to make statistical calculations. In order to apply this to Machine Learning, we have to create an algorithm, where the algorithm takes a *training set* and a *test set*. For a complete understanding of how the Linear Regression algorithm is coded, see the Appendix, *Linear Regression Proof of Concept Programming*. The algorithm will use the training set to determine the coefficients of the best-fit line, and the test set to make the predictions[7]. To make predictions using Machine Learning, we need data. The data we use, are usually divided in half. *Training Data* and *Test Data*, Training data is used to build a model so that the model can make future predictions. Meanwhile, test data is used to test the model by making predictions and creating some benchmarks regarding the model. Looking at our benchmarks we can analyse if the model we have is a good fit to make predictions, which will be discussed in the *Section 2.4*. To assess the results from the training set and test set, both the training set and test set predictions are calculated separately. By using the `rss_tss()` function, we calculate the RSS and TSS values, to calculate the R^2 statistic, where R^2 statistic are the scores for the model.

2.4 Assessing the Model

Once we have a model, we also would like to know if it is a good fit for our data. Can we rely on the predictions we can make from our model? To assess our model, we can use the R^2 statistic. The R^2 is the proportion of variability in the *predicted variable*(response variable) depending on the *predictor variable* [39]. R^2 can be calculated with the formula;

$$R^2 = 1 - \frac{RSS}{TSS}$$

where TSS is the *total sum of squares*, $TSS = \sum_{i=1}^{n}(y_i - \bar{y})^2$. RSS is the *residual sum of squares*, $RSS = \sum_{i=1}^{n}(y_i - \hat{y_i})$. $\bar{y}$ is the mean of the observed response variables, and $\hat{y}$ is the response variable depending on the predictor variable[16]. The R^2 has a value, $0 < R^2 < 1$. The bigger the value of R^2, the better *predictor variables* explain *response variables*. There is no generalized value for Machine Learning models, we can analyse our models and choose the model with the highest R^2 value for model accuracy.

Take the temperature and the fire percentage for an example. Let the *Table 2.1* as a training set, and assume there is other data for a test set similar to the training set. Looking at the *Figure 2.2* we can see that the `slr()` function returns an array of predictions, the training score and the test score. A score of 0.93 is a very optimistic score and it looks like our model works well for our training set. When we look at the test score, we do see that it does decrease and is around 0.82. A test score of 0.82 is not a bad score, we still do have a decent model for our test dataset.

Assessing the model with the R^2 score is crucial, but for a more comprehensive understanding of model performance Mean Squared Error (MSE) is introduced. MSE is defined as the average of the residual sum of squares, RSS. It is given by the formula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2$$

Here, y is the observed values, $\hat{y}$ is the predicted values from the model and n is the number of samples. The smaller the MSE gets, the closer the model fits to the data. Thus, it will indicate a better predictive performance[16].

```
train = np.array([
        [25, 30],
        [32, 40],
        [30, 39],
        [40, 70],
        [37, 66]
    ])

test = np.array([
        [28, 35],
        [35, 45],
        [33, 42],
        [42, 72],
        [38, 68]
    ])

lr.slr(train, test)
```

✓ 0.1s

```
{'Predictions': array([34.94, 55.45, 49.59, 75.96, 64.24]),
 'Training R-Squared': 0.9336287735849057,
 'Test R-Squared': 0.8201304427369188}
```

Figure 2.2: Input and Output of the `slr()` function

Moving forward with the temperature and the fire percentage example, the MSE of the training data is approximately 16 meanwhile the MSE of the test data is approximately 39. Both of the model assessment scores indicate that the model is well fitted for the training data but it does not do as well in the test data. This is called overfitting and it will be talked through in *Section 2.7*.

A more in-depth discussion will be made in the upcoming sections on how to assess complex models with more than one predictor variable.

2.5 Multiple Linear Regression

Simple Linear Regression is useful for two variable datasets. Even though it is a very powerful tool, when we look at most of our data, we have more than two variables. Fire percentage can be calculated depending on the temperature, but also if we add more independent variables, such as humidity, wind, etc. We will have a model that makes predictions better. Thus, we can claim that the more variables, the better our predictions can be. When we look at the Multiple Linear Regression model;

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_i X_i + \epsilon$$

where Y is the dependent variable, $X_1 ... X_i$ are the independent variables (features) and ϵ is the error term. Since this model has more than one independent variable, when we graph, we end up getting a plane. To do calculations and describe the model simply, we write in a

matrix format[18].

$$\boldsymbol{Y} = \begin{pmatrix} Y_1 \\ Y_2 \\ . \\ . \\ . \\ Y_i \end{pmatrix} \boldsymbol{Z} = \begin{pmatrix} 1 & x_{11} & x_{12} & ... & x_{1K} \\ 1 & x_{21} & x_{22} & ... & x_{2K} \\ . & . & . & ... & . \\ . & . & . & ... & . \\ . & . & . & ... & . \\ 1 & x_{n1} & x_{n2} & ... & x_{nK} \end{pmatrix} \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ . \\ . \\ . \\ \beta_K \end{pmatrix} \boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ . \\ . \\ . \\ \epsilon_i \end{pmatrix} \tag{2.1}$$

where K is the number of independent variables that are in the model and n is the number of samples. We can say that the Multiple Linear Regression Model can also be expressed as[18]:

$$\boldsymbol{Y} = \boldsymbol{Z\beta} + \boldsymbol{\epsilon} \tag{2.2}$$

The matrix $\boldsymbol{Z}$ is also known as the *design matrix*[18]. The design matrix has $K+1$ columns for n samples. When we look at the first column, we can see that it is filled with 1s. This is to account for and calculate the intercept. The rest of the columns are the values of data for the corresponding independent variables. Similar to *SLR*, the *Least Squares Estimate* is also calculated to work out the coefficients $\boldsymbol{\beta}$. Least Squares Estimate method determines the coefficients by minimizing the sum of squares. Let S be the function to minimize the sum of squares and be represented by:

$$S(\boldsymbol{\beta}) = (\boldsymbol{Y} - \boldsymbol{Z\beta})^T(\boldsymbol{Y} - \boldsymbol{Z\beta})$$

To find the value of β that minimizes S we take the derivative of S with respect to β and set it equal to 0 to get the normal equation.

$$\begin{aligned} \frac{\partial S}{\partial \boldsymbol{\beta}} &= 0 \\ -2\boldsymbol{Z}^T\boldsymbol{Y} + 2\boldsymbol{Z}^T\boldsymbol{Z\beta} &= 0 \\ \boldsymbol{Z}^T\boldsymbol{Z\beta} &= \boldsymbol{Z}^T\boldsymbol{Y} \end{aligned}$$

Assuming that $\boldsymbol{Z}^T\boldsymbol{Z}$ is invertible, we can solve for $\hat{\boldsymbol{\beta}}$:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{Z}^T\boldsymbol{Z})^{-1}(\boldsymbol{Z}^T\boldsymbol{Y}) \tag{2.3}$$

is the *Least Squares Estimate* of a Multiple Linear Regression model. Thus we have:

$$\hat{\boldsymbol{Y}} = \boldsymbol{Z}\hat{\boldsymbol{\beta}} \tag{2.4}$$

for our predictions[18][29].

2.6 Application of Multiple Linear Regression

As a learning algorithm compared to Simple Linear Regression, Multiple Linear Regression is different to implement. It does follow the simple Mathematical theory we mentioned in *Section 2.5*, thus we end up using different libraries and functions. Multiple Linear Regression is applied using vectors and matrices; `Numpy` library will be useful in this sense.

```
def train_mlr(train):
  predictions = np.array([])
  Y = train[:, -1]
  Z = train[:, :-1]
  Z = np.column_stack([np.ones(Z.shape[0]), Z])
  beta_hat = np.linalg.inv(Z.T.dot(Z)).dot(Z.T).dot(Y)
  Y_hat = Z.dot(beta_hat)
  predictions = np.append(predictions, Y_hat)
  rss, tss = rss_tss(train, predictions)
  r_sq = r_squared(rss, tss)
  results = {
      "Beta_hat": beta_hat,
      "R-Squared Training": r_sq
  }
  return results
```

The code above is the algorithm for creating the model for Multiple Linear Regression. To initiate the design matrix $\boldsymbol{Z}$, we make sure to assign it to the correct part of our dataset. In future cases this could mean that to be able to create our model we may have to do data processing. Moving forward with $\boldsymbol{Z}$, to complete our design matrix, we use the `Numpy` library to add a column of ones as the first column of our matrix. This is to handle the coefficients(intercepts) for our model. Using the "Linear Algebra" method; `linalg` we calculate the $\hat{\boldsymbol{\beta}}$. We use functions such as `inv()` and `dot()` to inverse and do matrix multiplication. Similarly, we can move forward to calculate the $\hat{\boldsymbol{Y}}$ using the `dot()` function. The `Numpy` library has been a great tool to calculate these mathematical expressions. To estimate the Residual Sum of Squares (RSS) and the Total Sum of Squares (TSS), we use the predictions array that has the $\hat{\boldsymbol{Y}}$ matrix. Similar to **SLR**, the R-squared is also estimated using the RSS and TSS. Inputting these to our `r_squared()` function we should get the corresponding R-Squared statistic for our model. As a result of creating the model, we end up with the vector $\hat{\boldsymbol{\beta}}$, for the coefficients and the R-Squared statistic of the model.

```
def predict_mlr(test, beta_hat):
  predictions = np.array([])
  Z = test[:, :-1]
  Z = np.column_stack([np.ones(Z.shape[0]), Z])
  Y_hat = Z.dot(beta_hat)
  predictions = np.append(predictions, Y_hat)
  rss, tss = rss_tss(test, predictions)
  r_sq = r_squared(rss, tss)
  results = {
      "Predictions": predictions,
      "R-Squared Test": r_sq
  }
  return results
```

Similar to `train_mlr()`, `predict_mlr()` function works the same way. The main difference is that it takes $\hat{\boldsymbol{\beta}}$ and the test dataset as parameters. First, the algorithm processes the data and creates the design matrix $\boldsymbol{Z}$. Using the parameter $\hat{\boldsymbol{\beta}}$, we calculate the $\hat{\boldsymbol{Y}}$ matrix. Predictions are appended in the array corresponding to the results from $\hat{\boldsymbol{Y}}$. Moving forward, to asses our test dataset, RSS and TSS are calculated to estimate the *R-Squared Statistic*. As a result, the `predict_mlr()` function returns a set of predictions and the R-squared statistic respectively.

```
In [ ]: lr.slr(train_data, test_data)

Out[ ]: {'Intercept': 1.92,
         'Slope': 5.16,
         'Predictions': array([29.12727273, 32.41090909, 53.52      , 49.29818182, 22.56      ,
                26.31272727, 23.49818182, 17.4       , 35.69454545, 32.88      ,
                51.17454545, 21.15272727, 14.11636364, 44.13818182, 47.42181818,
                51.64363636, 37.10181818, 12.24      , 19.27636364, 30.53454545]),
         'Training R-Squared': 0.982296479486738,
         'Test R-Squared': 0.9511046101878732}

         Comparing the mlr and slr should give similar intercept and slope and R-Squared.

In [ ]: print(lr.train_mlr(train_data))
        lr.slr(train_data, test_data)

        {'Beta_hat': array([1.94109974, 5.15576782]), 'R-Squared Training': 0.9822971568748751}
Out[ ]: {'Intercept': 1.92,
         'Slope': 5.16,
         'Predictions': array([29.12727273, 32.41090909, 53.52      , 49.29818182, 22.56      ,
                26.31272727, 23.49818182, 17.4       , 35.69454545, 32.88      ,
                51.17454545, 21.15272727, 14.11636364, 44.13818182, 47.42181818,
                51.64363636, 37.10181818, 12.24      , 19.27636364, 30.53454545]),
         'Training R-Squared': 0.982296479486738,
         'Test R-Squared': 0.9511046101878732}

         We can see that we have similar answers for mlr and slr as required.
```

Figure 2.3: *SLR* vs *MLR*

```
In [ ]: # Training data
        train = np.array([
            [1500, 3, 5, 10, 250],   # A relatively new, medium-sized house not too far from the city.
            [2500, 4, 2, 5, 400],    # A new, large house close to the city.
            [1200, 2, 10, 15, 200], # An older, small house a bit far from the city.
            [2200, 3, 3, 7, 350],    # A new, large house moderately close to the city.
            [1800, 3, 8, 8, 290]     # A moderately old, medium-sized house moderately close to the city.
        ])

        # Test data
        test = np.array([
            [1600, 3, 6, 9, 260],    # A moderately new, medium-sized house not too far from the city.
            [2400, 4, 3, 6, 390],    # A new, large house close to the city.
            [1300, 2, 9, 14, 210],  # An older, small house a bit far from the city.
            [2100, 3, 4, 8, 340],    # A new, large house moderately close to the city.
            [1900, 3, 7, 7, 300]     # A moderately old, medium-sized house moderately close to the city.
        ])

In [ ]: beta_hat = lr.train_mlr(train)['Beta_hat']
        lr.train_mlr(train)

Out[ ]: {'Beta_hat': array([ 5.34653465,  0.14455446,  7.62376238, -0.59405941,  0.79207921]),
         'R-Squared Training': 1.0}

In [ ]: lr.predict_mlr(test, beta_hat)

Out[ ]: {'Predictions': array([263.06930693, 385.74257426, 214.25742574, 335.74257426,
                304.25742574]),
         'R-Squared Test': 0.9957771473789029}
```

Figure 2.4: *MLR* Example

2.7 Comparison of Results

As we know from *Section 2.1*, Simple Linear Regression has one independent (X) variable and one dependent variable (Y). Multiple Linear Regression can have more than one independent variable and can give us more complex models. Although they are different models, `mlr()` models can also be created only having one independent variable. This means that we can compare our `slr()` and `mlr()` models. Looking at the *Figure 2.3*, we can see that in the second cell `mlr()` function is printed and the `slr()` function is returned. The first line of the output represents the output from the `mlr()` function. From the second line is the return of the `slr()` function. Logically, we would be expecting similar coefficients from both of the functions. In the results we do see that from the `mlr()` function we have very close coefficients to the ones from the `slr()` function. The intercept is around 1.9 and the slope is around 5.6.

In the *Figure 2.4*, we apply a small dataset we have created. This is a dataset which has some information about a house and its price as being the dependent variable. We apply

```
beta_hat = lr.train_mlr(train_data)["Beta_hat"]
lr.train_mlr(train_data)
```

```
{'Beta_hat': array([ 4.73753216e+06, -6.28002856e+04,  4.22514753e+04,  2.54674175e+02,
        -5.38145409e-01,  5.23130052e+04,  3.70999019e+05,  3.46965455e+04,
         2.06033665e+04,  2.14862150e+01,  2.54674804e+01, -2.44683288e+03,
         1.09238391e+01]),
 'R-Squared Training': 0.4010002043753631}
```

```
lr.predict_mlr(test_data, beta_hat)["R-Squared Test"]
```

```
0.06713160856910139
```

Failure of R-squared, have to understand why the funny numbers.

Removed some columns from the data to make it more sensible for calculations.

```
from sklearn.linear_model import LinearRegression
```

```
X_train = train_data[:, :-1]
y_train = train_data[:, -1]

X_test = test_data[:, :-1]
y_test = test_data[:, -1]

model = LinearRegression().fit(X_train, y_train)
```

```
model.score(X_train, y_train)
```

```
0.40100020437540573
```

```
model.score(X_test, y_test)
```

```
0.06713161020826608
```

Figure 2.5: *MLR, sklearn* Comparison

the training dataset to train and create our ***MLR*** model. Our training R-squared statistic results show that it is a perfect model with a score of 1.0. Using the model we have created, we can make predictions by applying the test set to the model. As a result, we have a set of predictions and a very high R-squared value. From this, we can conclude that given the training dataset and the test set our model can distinctively predict the house prices.

As simple as the Linear Regression models are, the coefficients could have noise, that is affecting our models. A complex dataset will tend to have more noise when it is modelled with Linear Regression. This is because, Linear Regression learning model includes all the independent variables in the model. In reality, some of these independent variables may not affect the dependent variable, in the same way a Linear Regression model estimated. The concept of modelling the smallest variation in the data is having a model with a high amount of noise. Thus this causes the data to **overfit**[21]. *Figure 2.5* uses the dataset which is much more complex compared to the previous example. This dataset is a data of house prices in Washington USA - details can be found in *Section 1.6 Datasets* -. Using this dataset in the *Figure 2.5*, we end up having a model with a lot of noise. This could be seen from the coefficient values. In *Figure 2.5*, we train the model using the training set as usual. When we look to asses our model, we get an R-Squared Statistic of 0.40. This does indicate that our model is not the perfect model for predicting the price but we continue to test our model. When we apply our test set to the model, we get an even lower R-squared statistic with 0.067. This is a simple example of overfitting. The data is overfitting with the Linear Regression model, so when we enter new parameters, the predictions do not correspond to our expectations. Moving forward, a good benchmark for us to compare our algorithm would be the `sklearn` library. The *Figure 2.5* shows us the R-squared statistic from the algorithms we implemented in the *Section 2.6* and the results from the `sklearn` library. Comparing our results R-Squared Statistic to the `sklearn` library results, we do see that our results are correct and reliable.

Chapter 3: **Ridge Regression**

This chapter shifts the focus to Ridge Regression, a method that mitigates the multicollinearity issue Linear Regression presents. Ridge Regression introduces a penalty term to the cost function, to shrink the coefficients of the model, mitigating multicollinearity that could exist in the model. Moreover, the chapter will delve into the mathematical theory behind Ridge Regression and how it is derived. For further analysis, the implementation of the Ridge Regression using matrix inversion will be shown. A comparative analysis between Linear Regression and Ridge Regression is also made to understand the impact of the penalty term.

3.1 Introduction to Ridge Regression

Linear Regression is a great statistical approach to many problems. It is widely used in industries but when the dataset becomes too complex, there will be overfitting, as we discussed in *Section 2.7*. As the dataset becomes more complex, having a lot of features, one or more features could be explained by each other. This phenomenon is multicollinearity. Multicollinearity can cause the coefficients to be unreliable and inefficient, resulting in poor predictive performance of the model. When there is multicollinearity in our dataset, using linear regression, our training score could be performing well but the test score will tend to be lower. This is because the coefficients are affecting the model rigorously, and when there is data that is not introduced before, the model will not be able to make accurate predictions. To minimize the overfitting, we can introduce a *tuning parameter*, λ to the *Least Squares Estimate* method. This tuning term can shrink the effect of the coefficients, thus the model will not be overfitted as it used to be in the Linear Regression model[16]. In conclusion, Ridge Regression brings solutions to multicollinearity by adding the tuning parameter Least Squares Estimate method to decrease the effects of the coefficients, referred to as *L2 Regularization*. Thus, we can have an enhanced model, with a better performance on unseen data.

3.2 Ridge Regression Compared to Linear Regression

Ridge Regression was introduced in 1962 by A.E. Hoerl, to control the variance of the coefficients[14][13]. Then Hoerl and Kennard further developed the concept of Ridge regression by identifying the Ridge estimator as the point on the cost function ellipsoid, which is centred around the LS estimator, that lies nearest to the origin[12]. The Ridge estimator can be observed from the *Figure 3.1*. $\hat{\beta}_{Ridge}$ can be found by[16]:

$$\hat{\beta}_{Ridge} = \text{minimize}_{\beta} \left\{ \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\} \tag{3.1}$$

where y_i is the observed i^{th} value, β_0 is the intercept β_i is the i^{th} coefficent, x_{ij} is the independent variable value at ij^{th} index, n is the number samples and p is the number of features[16];

$$\text{RSS} = \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2$$

thus $\hat{\beta}_{Ridge}$ can also be written as [16];

$$\hat{\beta}_{Ridge} = \text{minimize}_{\beta} \left\{ \text{RSS} + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}$$

Throughout the project, to apply these mathematical formulations to code, it is essential to write formulas in a linear algebra format to ease the complexity of coding. Initially, RSS in linear algebra form can be written as where $\boldsymbol{Z}$ is the design matrix as shown before in Chapter 2;

$$\text{RSS} = (\boldsymbol{Y} - \boldsymbol{Z\beta})^T(\boldsymbol{Y} - \boldsymbol{Z\beta}).$$

Then observing that the penalty term $\lambda \sum_{j=1}^{p} \beta_j^2$ can be written as $\lambda \boldsymbol{\beta^T \beta}$.

When we combine these two parts;

$$S(\boldsymbol{\beta}) = \left\{ (\boldsymbol{Y} - \boldsymbol{Z\beta})^T(\boldsymbol{Y} - \boldsymbol{Z\beta}) + \lambda \boldsymbol{\beta^T \beta} \right\}$$

To find the value of coefficients that minimizes the **RR**, the derivative of the cost function with respect to β has to be taken. Setting this derivative to zero and solving for β will give us the estimator for $\hat{\beta_{ridge}}$.

$$\frac{\partial S}{\partial \boldsymbol{\beta}} = \frac{\partial}{\partial \boldsymbol{\beta}} [(\boldsymbol{Y} - \boldsymbol{Z\beta})^T(\boldsymbol{Y} - \boldsymbol{Z\beta}) + \lambda \boldsymbol{\beta^T \beta}]$$

The derivative consists of two parts, one part is the RSS and the other is the penalty term. The RSS part simplifies to:

$$-2\boldsymbol{Z}^T\boldsymbol{Y} + 2\boldsymbol{Z}^T\boldsymbol{Z\beta}$$

The second part which the penalty term simplifies to:

$$\frac{\partial}{\partial \boldsymbol{\beta}} (\lambda \boldsymbol{\beta}^T \boldsymbol{\beta}) = 2\lambda \boldsymbol{\beta}$$

Then we can combine and set the derivative equal to zero and solve for $\boldsymbol{\beta}$ to find the estimator.

$$\begin{aligned} -2\boldsymbol{Z}^T\boldsymbol{Y} + 2\boldsymbol{Z}^T\boldsymbol{Z\beta} + 2\lambda\boldsymbol{\beta} &= 0 \\ 2\boldsymbol{Z}^T\boldsymbol{Z\beta} + 2\lambda\boldsymbol{\beta} &= 2\boldsymbol{Z}^T\boldsymbol{Y} \\ (\boldsymbol{Z}^T\boldsymbol{Z} + \lambda\boldsymbol{I})\boldsymbol{\beta} &= \boldsymbol{Z}^T\boldsymbol{Y} \end{aligned}$$

Finally, we can multiply both sides by the inverse of $(\boldsymbol{Z}^T\boldsymbol{Z} + \lambda\boldsymbol{I})$ to solve for $\boldsymbol{\beta}$.

This gives us the Ridge estimator:

$$\hat{\beta}_{Ridge} = (\boldsymbol{Z}^T\boldsymbol{Z} + \lambda\boldsymbol{I})^{-1}\boldsymbol{Z}^T\boldsymbol{Y} \tag{3.2}$$

Equation *3.2* is used to estimate the coefficients of the Ridge model. As it can be seen, it is very similar to the Least Squares Estimate, only adding the shrinkage term $\lambda\boldsymbol{I}$. Here, $\boldsymbol{I}$ is the identity matrix, with 1's across its diagonal and the other variables are the same as in the equation *2.3*. Observe that, the λ does not affect the interception, β_0.

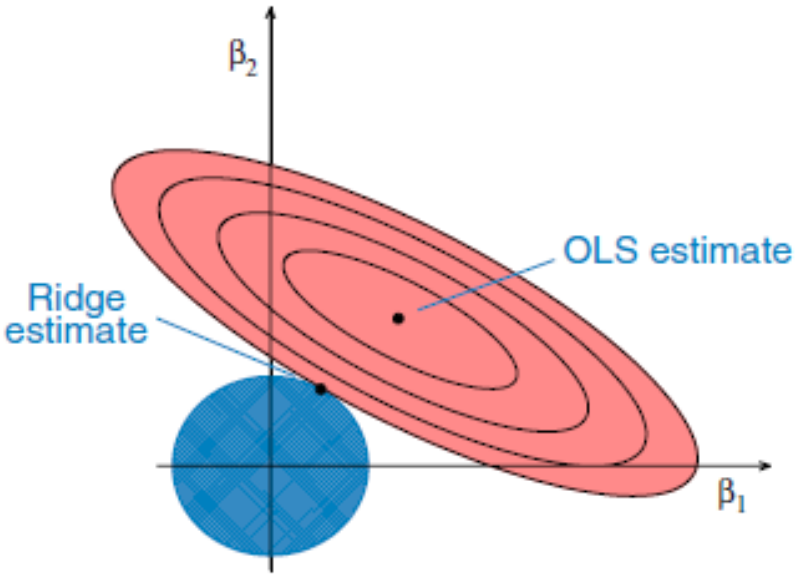

Figure 3.1: *Ridge Estimation Point*

Ridge Regression's advantage over Linear Regression is set on *bias-variance tradeoff* [16]. Unlike Linear Regression, Ridge Regression does not create only one set of coefficients, instead, it creates coefficients depending on the tuning parameter, λ. As the tuning parameter increases, we can see the variance in the model decreases but the bias increases. We will observe in future examples that when there is a very complex dataset, and we have a Linear Regression model, the variance will be higher, after applying our tuning parameter λ the new model will have a lower variance. Note that the tuning parameter $\lambda = 0$, the model will be the Linear Regression model, as the parameter will not have any impact on the Least Squares Estimate. When $\lambda \to \infty$ the model coefficients will tend to 0[16].

It is often observed that when there is a linear relationship between the predictor variables and the response(predicted) variable, the linear regression will exhibit low bias but potentially high variance. This indicates that when a small change in the training data can affect the Linear Regression coefficients in large values. Particularly in situations where the training data has almost the same amount of predictor variables as the observations, the Linear Regression model can suffer from very high variance. If the predictor variables are more than the number of observations, Linear Regression will not be able to come up with a model. However, Ridge Regression can still provide a model with a small bias and a decrease in variance[16]. Thus, we can conclude that Ridge Regression works well when there is a high variance between the predictor variables.

Observing the *Figure 3.2*, we can see the impact of λ on our models. Using the Extended Boston Housing dataset - details of the dataset can be found in `Section 1.6, Datasets`-, I have developed both a linear regression model and a Ridge regression model to analyse their coefficients. In the bar plots of *Figure 3.2*, the first graph is a bar plot where the tuning parameter is set to, $\lambda = 1$. From the graph, it is evident that the Ridge Regression coefficients are much smaller than those of Linear Regression coefficients. The tuning parameter constrains the coefficients effectively, thus the difference between the coefficients is very high. Moving forward, when we lower λ to 0.0001, (getting closer to 0), we see that the coefficients are less regularized, resulting in less bias relative to $\lambda = 1$. The coefficient values of Ridge Regression are more visible in the graph, not very close to 0 anymore. In contrast, for $\lambda = 1000$, the green bars (Ridge Regression Coefficients) look like dots as they are very close to 0. The regularization of the coefficients is very high so, this model will have the most bias and the least variance between predictor variables.

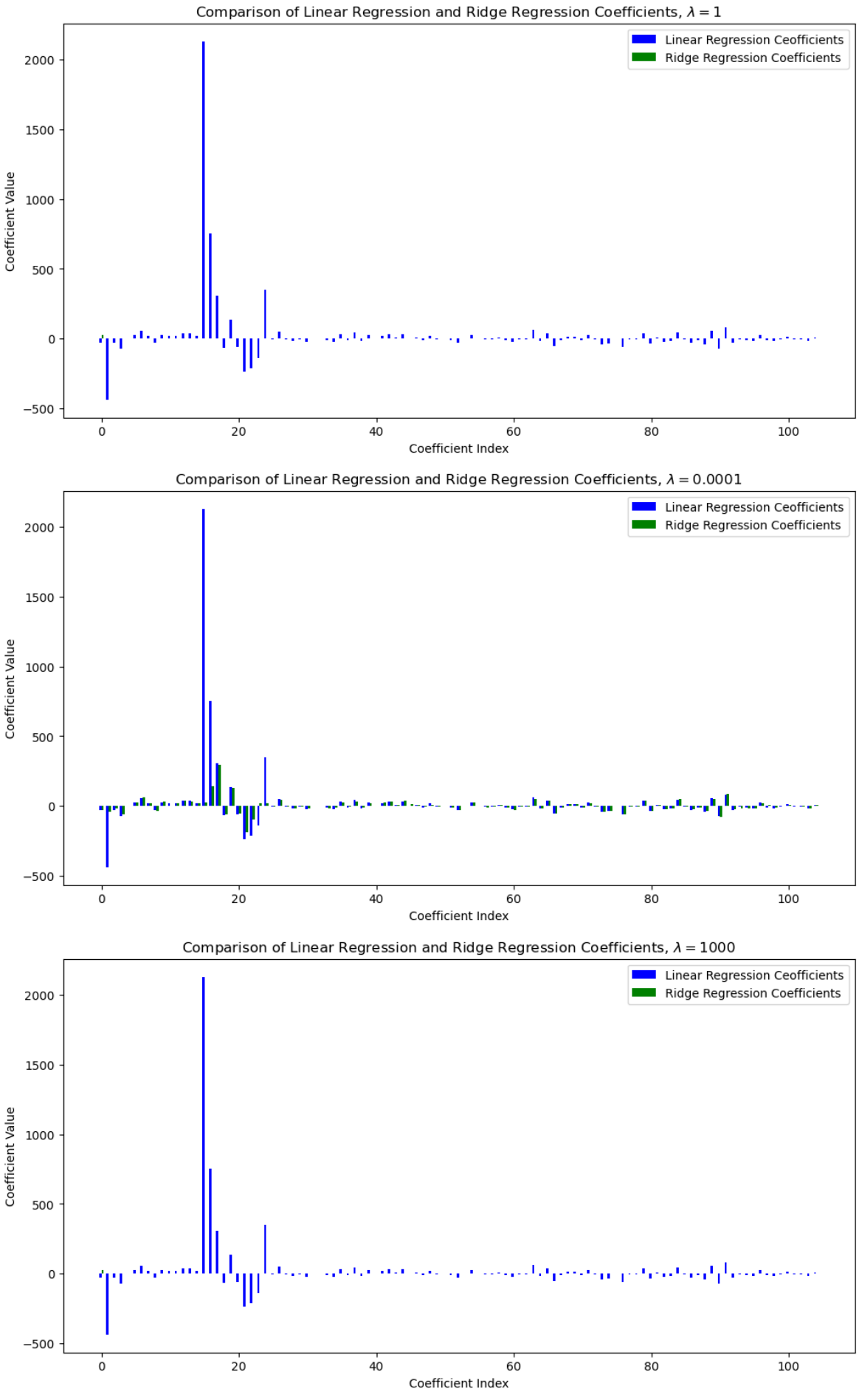

Figure 3.2: *Ridge vs Linear Regression, Coeffeicients*

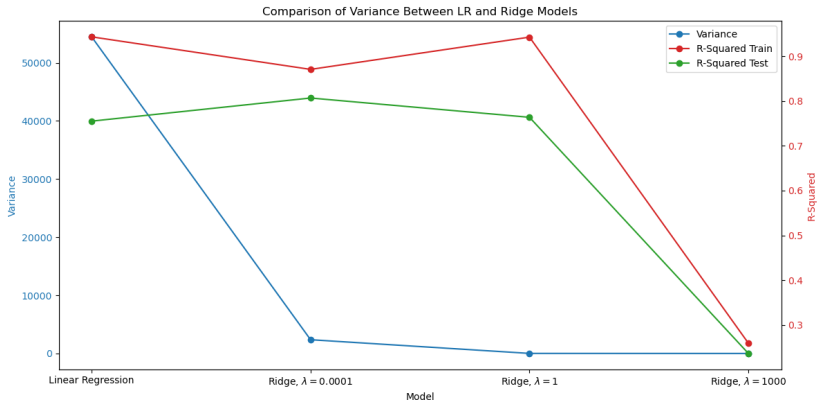

Figure 3.3: *LR and RR Variance and R-Squared Results*

In the *Figure 3.3*, the extended Boston Housing Dataset is used again to construct various models. Examining the Linear Regression model it is evident that it has a very high variance and but a very high training R-squared score of 0.94. Due to the high variance between the coefficients, this model tends to perform poorly with unknown data. This is confirmed when is the model applied to the test dataset, we do indeed get a lower R-squared score of 0.75. Applying our Ridge Regression algorithm to the training set reveals that with a low tuning parameter, $\lambda = 0.0001$, the variance between coefficients becomes much lower compared to the Linear Regression model. However, a notable downside is that the R-squared score for the training set is 0.87, lower than that of the Linear Regression model. Observing the test R-squared score for the Ridge model, we do end up having a higher test set score with, 0.80. This indicates that this model does better compared to the Linear Regression model, with data that was not trained previously. Moving forward, in the *Figure 3.3*, we also can observe that as the tuning parameter goes towards infinity, variance is also converging to 0. Additionally, as the tuning parameter increases, we have noted that bias increases. Observing the graph, when $\lambda = 1000$, we do see that the R-squared scores are very low for both training and testing, suggesting that this model is not an ideal model for predictions.

3.3 Application of Ridge with Inverted Matrices

Remember from the equation *3.2*;

$$\hat{\beta}_{Ridge} = (\boldsymbol{Z}^T\boldsymbol{Z} + \lambda\boldsymbol{I})^{-1}\boldsymbol{Z}^T\boldsymbol{Y}$$

Applying this equation to code was a similar process to the Multiple Linear Regression Algorithm that was developed in the *Section 2.6*. The code below is the implementation of the Ridge regression algorithm. It is a very straightforward algorithm that is dependent on the equation *3.1*. Similar to **MLR**, the data is processed by using numpy array properties. The design matrix is assigned then the first row of the design matrix is changed to ones using the `np.ones()` function. The difference compared to **MLR** is that the `I_p` variable. The `I_p` variable is the penalty term with an identity matrix, that has 1's across its diagonal except for the (0,0) index. Moving forward, the coefficients of the model are calculated using another `Numpy` function, `np.linalg.pinv()`. Note that in the **MLR** algorithm I have used `np.linalg.inv()`, but the function only inverses matrices which are square or invertable [2]. During testing of **MLR**, the algorithm failed to give appropriate R-Squared Scores, and this was due to the design matrix not being able to be inverted, either was not a square matrix

or it just was not invertible. Ridge Regression is expected to do well in complex and varied data sets, the design matrix we end up getting may not be a square matrix, or be invertible. Thus I have used `np.linalg.pinv()` function to be able to compute more

```
def train_ridge_regression(train, alpha=1.0):
  Y = train[:, -1]
  Z = train[:, :-1]
  Z = np.column_stack([np.ones(Z.shape[0]), Z])

  I_p = np.eye(Z.shape[1]) # Create the Identity Matrix
  I_p[0, 0] = 0            # that has 1's across the diago-
                           # nal except for the (0, 0)
                           # index.
  beta_hat = np.linalg.pinv(Z.T.dot(Z) + alpha * I_p).dot(Z.T).dot(Y)
  predictions = Z.dot(beta_hat)

  rss, tss = rss_tss(train, predictions)
  r_sq = r_squared(rss, tss)
  variance = var(beta_hat)
  mse = mse_(Y, predictions)

  results = {
      "Beta Hat": beta_hat,
      "R-Squared Training": r_sq,
      "Variance": variance
      "MSE": mse
  }

  return results
```

complex design matrix inverses. This is the Moore-Penrose Pseudoinverse approach to invert non-square matrices, especially in the context of least squares estimate[5]. After applying this function to **RR** it also was added to the **MLR** algorithm to be able to make better comparisons. Moving forward, from the code we can also observe that variance is added compared to the **MLR** algorithm. The `variance` variable is the sample variance between the coefficients. It is a very basic function, `var()` to compute the sample variance, it uses the `Numpy` library `np.var()` function. The MSE of the model is also calculated by using the `mse_()` function. Thus we have the trained **RR** model and its coefficients returned within a dictionary with various other variables. The testing of the **RR** model is the same as **MLR** which can be observed in *Section 2.6*.

Applying the right approach of Test Driven Development is very important for our algorithms. So to test the models easier, I have created an easier access function to the "Extended Boston Housing Dataset" from mglearn. This is the code below,

```
def load_extended_boston(r_state):
  X, y = mglearn.datasets.load_extended_boston()
  X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=r_state)
  y_train = y_train.reshape(-1,1)
  y_test = y_test.reshape(-1,1)
  train_data = np.concatenate((X_train, y_train), axis=1)
  test_data = np.concatenate((X_test, y_test), axis=1)
  return train_data, test_data
```

The code first loads the dataset, then using the `sklearn` library, `train_test_split()` function is called to create the design matrix depending on a given random state parameter. This is to remember the results for a specific random shuffle of the data. Observe that `y_train` and `y_test` are turned into vectors and concatenated with the training data and test data respectively. Using the returned training and test datasets, we can now train and test our data faster.

3.3.1 Testing the Algorithm

In *Section 3.3*, it was mentioned that `np.linalg.inv()` function can not be inverse if the design matrix is not a square matrix. Using the "Extended Boston Housing Dataset", we run into the problem of the design matrix not being a square matrix and not an invertable one. Thus, by using the Moore-Penrose Pseudoinverse, which is a more generalized method of inverting non-square matrices, we can invert our design matrix. Although inverting matrices is not computationally efficient, we will apply the *Gradient Descent* method in Chapter 4 to develop our models.

Observing the *Figure 3.4*, we can see that we have the same score for the Linear Regression model compared to the sklearn library, thus, we can say that our algorithm is working accurately using the `np.linalg.pinv()`, Moore-Penrose Pseudoinverse. In figure *Figure 3.5*, we can observe the test score of the Linear Regression model and the results from the Ridge Regression algorithm. Looking at our test score of the Linear Regression model, there is a sign of overfitting as the training score was 0.94 and the test score is 0.75. Testing our results compared to `sklearn` library, we can confirm that our model is accurate. Moving forward, we apply our training dataset to the Ridge Regression algorithm and create a model with the default tuning parameter, $\lambda = 1$. The training model returns a lower score compared to the one of Linear Regression but we move forward to test our model with the test dataset we gathered. The test score returns a higher score compared to the one of Linear regression with 0.80, thus this model will perform better with unknown data. We can also observe that the results of the scores are very close to each other compared with the `sklearn` library, thus, it can be confirmed that our algorithms are accurate.

Looking at *Figure 3.6*, this time we use another dataset to test our algorithms. This is the "California Housing Dataset", - details of the dataset can be found in Section 1.6, Datasets - which is a less complex dataset compared to the "Extended Boston Housing Dataset" but has more samples. After applying the training dataset to the Linear Regression algorithm, we see that we get a score of 0.6126 and when the test dataset is applied, the score further decreases

In [ ]:
```
X, y = mglearn.datasets.load_extended_boston()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=20)
lin = LinearRegression().fit(X_train, y_train)
lin.score(X_train, y_train)
```

Out[ ]:
```
0.9434644739413218
```

In [ ]:
```
y_train = y_train.reshape(-1,1)
y_test = y_test.reshape(-1,1)
train_data = np.concatenate((X_train, y_train), axis=1)
test_data = np.concatenate((X_test, y_test), axis=1)
model = lr.train_mlr(train_data)
model["R-Squared Training"]
```

Out[ ]:
```
0.9434644739413219
```

Figure 3.4: *Linear Regression Training Score*

```
In [ ]: print("Test score of sklearn: " + str(lin.score(X_test, y_test)))
        print("Test score of my linear regression: " + str(lr.predict_mlr(test_data, model["Beta_hat"])["R-Squared Test"]))

        Test score of sklearn: 0.7552361218471089
        Test score of my linear regression: 0.7552361218484293

In [ ]: ridge_model = lr.train_ridge_regression(train_data)
        ridge_model["R-Squared Training"]

Out[ ]: 0.8708317636707955

In [ ]: ridge = Ridge().fit(X_train, y_train)
        ridge.score(X_train, y_train)

Out[ ]: 0.8708317636707952

In [ ]: print("Test score of sklearn: " + str(ridge.score(X_test, y_test)))
        print("Test score of my ridge regression: " + str(lr.test_ridge_regression(test_data, ridge_model['Beta Hat'])["R-S

        Test score of sklearn: 0.8068661033562764
        Test score of my ridge regression: 0.8068661033562888
```

Figure 3.5: *Ridge Regression Result Comparison*

```
In [ ]: california_housing = fetch_california_housing()
        X = california_housing.data
        y = california_housing.target

        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
        y_train = y_train.reshape(-1, 1)
        y_test = y_test.reshape(-1, 1)
        train_data = np.concatenate((X_train, y_train), axis=1)
        test_data = np.concatenate((X_test, y_test), axis=1)
        linear_model = lr.train_mlr(train_data)
        linear_model

Out[ ]: {'Beta_hat': array([-3.70232774e+01,  4.48674910e-01,  9.72425757e-03, -1.23323343e-01,
                 7.83144908e-01, -2.02962049e-06, -3.52631849e-03, -4.19792484e-01,
                -4.33708061e-01]),
         'R-Squared Training': 0.612551191396695}

In [ ]: lr.predict_mlr(test_data, linear_model["Beta_hat"])["R-Squared Test"]

Out[ ]: 0.5757877060542553

In [ ]: ridge_model = lr.train_ridge_regression(train_data, 0.8)
        ridge_model

Out[ ]: {'Beta Hat': array([-3.70201927e+01,  4.48543661e-01,  9.72562388e-03, -1.23075882e-01,
                 7.81761778e-01, -2.02657537e-06, -3.52595050e-03, -4.19788052e-01,
                -4.33686269e-01]),
         'R-Squared Training': 0.6125511485601522}

In [ ]: lr.test_ridge_regression(test_data, ridge_model["Beta Hat"])["R-Squared Test"]

Out[ ]: 0.5758415523925382
```

Figure 3.6: *California Housing Dataset Results*

to 0.5757. There is a slight overfitting in this model but we will try to mitigate it again by regularization. Using the same training dataset, to develop our Ridge Regression model with a tuning parameter $\lambda = 0.8$, we end up getting a very similar training score. When the test dataset is applied to the Ridge Regression model, the score decreases to 0.5758. This indicates that there is a slight improvement but with the chosen tuning parameter this was the best model we can get. There could be another tuning parameter value, that would further improve the model's performance. We will discuss this in *Section 5.3*, which focuses on parameter selection.

3.4 Results

Table 3.1: Comparison of Linear and Ridge Regression Models for $\lambda = 1$ with R^2

Dataset	LR R^2 Train	LR R^2 Test	RR R^2 Train	RR R^2 Test
New York	0.6849	0.6690	0.6848	0.6721
Washington	0.2147	0.2147	0.2146	0.2275
Boston	0.7304	0.7663	0.7281	0.7607
Extended Boston	0.9424	0.8386	0.8624	0.8532
California	0.6078	0.6007	0.6077	0.6006

Table 3.2: Comparison of Linear and Ridge Regression Models for $\lambda = 1$ with MSE

Dataset	LR MSE Train	LR MSE Test	RR MSE Train	RR MSE Test
New York	113120×10^7	88718×10^7	11314×10^7	87868×10^7
Washington	26836×10^7	19021×10^7	26836×10^7	19015×10^7
Boston	23.3622	18.1147	23.5556	18.5556
Extended Boston	4.9921	12.5149	11.9210	11.3808
California	0.5243	0.5256	0.5243	0.5256

Continuing from the previous section's analysis, further analysis is made by the given tables. Observing the tables given above, an analysis of the Linear Regression and Ridge Regression models are made using the R^2 and the MSE. The training and test dataset is split by using the `train_test_split()` function and the `random_state=17`. The models are trained and tested on all the available datasets. It is analysed that except for the Washington Housing dataset, the models are somewhat decent.

From analysing the `New York dataset` and R^2 scores for Linear Regression models are decent. It can be observed that there is a slight overfitting due to multicollinearity but the Ridge Regression model seems to mitigate this very efficiently.

`Washington dataset` seems to not perform well with these regression algorithms, it may have to do with that the dataset is not linear. Regression algorithms do not perform well with non-linear datasets. This data can be experimented with on Neural Networks for further analysis.

Moving forward, `Boston Housing dataset` returns unusual results. The R^2 of the test datasets are larger than the training datasets. This could have to do with how the data was split and by chance, we may have created a test dataset that performs better on the model compared to the training dataset. This issue can be fixed with cross-validation. In Chapter 5, we will apply cross-validation to mitigate this issue.

Furthermore, `Extended Boston Housing dataset` has informative results. Observing *Table*

3.1 and *Table 3.2*, it can be seen that there is a large overfitting that is present from the Linear Regression model. The R^2 decreases from 0.94 to 0.83 therefore, shows clear signs of overfitting. Examining the MSE values for the Linear Regression model, it can be seen that the training dataset fits very well with the model with an MSE of 4.9921. When the test data is fitted, then the MSE jumps to 12.5149, showing clear signs of overfitting. Again, Ridge Regression mitigates the overfitting issues very well in this data and lowers the difference between model assessment scores for training and test datasets drastically. The accuracy of the Ridge Regression model can be increased for this dataset by choosing a different λ value.

Finally, `California dataset` can be observed, and the results show us that our models are not the best for this dataset. But better than the Washington models. Results for the California dataset show us that there is not much multicollinearity in this data and Linear Regression looks sufficient for this dataset. This is due to the fact that the R^2 and MSE values are very close to each other. Observe that the MSE is very low for both of the models in training and testing. Since the R^2 is around 0.60 and MSE is so low can indicate some outliers are lowering the R^2 score. Further steps like dealing with outliers can be taken to increase the accuracy of this model as R^2 value around 0.60 is not very ideal.

In conclusion, the results show us that Ridge Regression indeed does a good job when multicollinearity is present and improves model generalization. The upcoming Chapters will build upon these insights by exploring feature selection. Lasso Regression will be introduced as a method to mitigate multicollinearity and create simpler models by reducing the number of features.

Chapter 4: **Gradient Descent Method**

Machine Learning has one question in mind, and that is how can we find the most optimized model? One popular method for optimization is gradient descent. This chapter will introduce the gradient descent algorithm, a powerful iterative algorithm to create accurate models. Gradient descent works by iteratively adjusting the model's parameters. It can be thought of as trying to find the lowest point in a valley if you start from the top of the mountain. This algorithm is widely used in the optimization world.

This chapter will explain how gradient descent finds this "lowest point" using a cost function. This cost function will measure the error of predictions, and the main goal of gradient descent is to minimize this error. To run this process smoothly and with the least amount of time, it is recommended to apply data normalization.

The computational cost of the matrix inversion we have applied in the previous chapters can be too expensive. When creating models, computational efficacy issues can be mitigated by using the gradient descent algorithm. The gradient descent method has advantages as well as matrix inversion. We will see how we can leverage the best out of the two methods.

4.1 Cost Function

The cost function quantifies the error between the predicted and the actual values and expresses this error as a single value[29]. The cost functions can vary depending on the model we are creating. For instance, in a regression model, Mean Squared Error (MSE) is commonly used as the cost function. The empirical goal of a machine learning model is to find the set of parameters that minimize the value of the cost function.

4.2 Gradient Descent

In the previous chapters, we have introduced the *Least Squares Estimate* method. Using this method, we have created algorithms that could predict values depending on other features. For instance, the price of a house can be predicted depending on the number of rooms and bedrooms it has. Whilst useful, The LS method could be computationally expensive particularly due to taking an inverse of a matrix is not an efficient way when you are dealing with a vast amount of data. To mitigate this issue, *Gradient Descent* method is introduced. The gradient Descent method estimates the parameters of a model by minimizing the cost function. In this chapter, the cost function will be regarded as the Mean Squared Error (MSE).

The Gradient Descent method is an iterative method to estimate the parameter vector, $\boldsymbol{\theta}$ of the model. By optimizing the cost function, the parameters are iteratively updated. The method can be applied to any kind of model which has a differentiable cost function[29].

Let $f(\boldsymbol{\theta})$ be the cost function. The method will initially assume that the parameter vector $\boldsymbol{\theta}$ has only 0's as its values. While initializing $\boldsymbol{\theta}$ as zeros is common, other initializations are also possible. Then from this point, iteratively we look towards the global minimum of the function.

$$\boldsymbol{\theta_{new}} = \boldsymbol{\theta_{old}} - \mu \nabla f(\boldsymbol{\theta_{old}}) \tag{4.1}$$

To look for the global minimum, we have to take steps by taking derivatives. These steps in the equation are, μ, which is called the *learning step*. This step is a scalar to determine the size of the steps taken towards the minimum[4]. Using this method, the gradient points towards the direction of the greatest increase of f around $\boldsymbol{\theta_{old}}$, using the learning step the algorithm makes a step towards the opposite direction, thus minimising the cost function. Then the output will be the last vector of $\boldsymbol{\theta}$[28].

4.2.1 Mathematical Simplification

The definition of Mean Squared Error gives us;

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{4.2}$$

Where $\hat{y}$ is the predicted values, y is the observed values and n is the number of data points.

Now let $\boldsymbol{Z}$ be the design matrix of data with all the features and let $\vec{\theta}$ be the parameter vector. Then from LS we know that $\hat{y} = \boldsymbol{Z}\vec{\theta}$.

Then using the euclidean norm, where $||\vec{v}|| = \sqrt{\sum(v_i)}$, we can get $||\boldsymbol{Z}\vec{\theta} - \vec{y}||^2 = \sum(\vec{\hat{y}_i} - \vec{y_i})^2$. Using this we end up having a new form of MSE;

$$MSE = \frac{1}{n}||\boldsymbol{Z}\vec{\theta} - \vec{y}|| \tag{4.3}$$

To apply MSE as a cost function to the gradient descent method, we can find the gradient of the MSE and apply it to the method.

$$\frac{\partial MSE}{\partial \theta_j} = \frac{2}{n}\sum_{i=1}^{n}(\boldsymbol{Z}\vec{\theta} - \vec{y})\boldsymbol{Z} \tag{4.4}$$

Using matrix algebra, this equation can then be simplified to;

$$\nabla MSE = \frac{2}{n}\boldsymbol{Z}^T\sum_{i=1}^{n}(\boldsymbol{Z}\vec{\theta} - \vec{y}) \tag{4.5}$$

This gives us the equation for the gradient descent as;

$$\boldsymbol{\theta_{new}} = \boldsymbol{\theta_{old}} - \mu \nabla MSE \tag{4.6}$$

Using the design matrices and vectors will ease the coding of the algorithm.

4.2.2 Application of the Gradient Descent

Using the simplified version of the Gradient Descent for easier programming, it is straightforward to code the algorithm. Similar to the LS algorithms that were written in the previous

chapters, we start with defining the design matrix, $\boldsymbol{Z}$ and the observed values $\boldsymbol{Y}$. Then we initiate the parameters vector $\vec{\theta}$ and an empty list to keep track of the value of the cost function.

```
def gradient_descent(data, learning_rate, threshold):
  Z = data[:, :-1]
  ones = np.ones((Z.shape[0], 1))
  Z = np.hstack((ones, Z))
  Y = data[:, -1]

  theta = np.zeros(Z.shape[1])
  cost_function = []
```

Moving forward, a `while` loop starts to converge the cost function to a minimum. Matrix multiplication is applied to compute the $\hat{y}$ values. Using the `mse()` function from the `model_evaluation` module, the value of the cost function is calculated and then appended to the cost function list. Then using the gradient, which we have found in section 4.2.1, the value of the gradient is estimated. To assess the minimum point of the cost function, this algorithm uses the norm of the gradient. Using the `linalg` module of the `Numpy` library, the norm is calculated. Furthermore, the new parameter is calculated using the `learning_rate` parameter multiplied by the gradient. If at any point the norm of the gradient is less than the given `threshold` parameter, the algorithm will exit the loop and return the $\vec{\theta}$.

```
def gradient_descent(data, learning_rate, threshold):
  .
  .
  .
  while True:
    y_hat = Z@(theta)
    mse = model_evaluation.mse(Y, y_hat)
    cost_function.append(mse)

    gradient = 2/len(Z)*Z.T@(y_hat - Y)
    gradient_norm = np.linalg.norm(gradient)
    theta -= learning_rate*gradient

    if gradient_norm < threshold:
        break
```

4.2.3 Learning Rate and Convergence Threshold

Learning rate and *threshold* parameters are key hyperparameters in Gradient Descent that can significantly affect the performance of the algorithm. The learning rate controls the size of the steps we take towards the minimum of the cost function. A learning rate that is too high may cause the algorithm to overshoot the minimum, resulting in divergence. Conversely, a learning rate that is too small can slow down the convergence progress. Learning rate is a value in the range of 0 to 1. There is not a generally accepted rule for determining the learning rate. The rate can be determined by trial and error, or adaptive methods that adjust the learning rate depending on the progress of the algorithm[38].

The convergence threshold is a small, positive value, that specifies if the algorithm has converged to a minimum. In section 4.2.2 it can be observed that the norm of the gradient is estimated in each iteration. If this value is less than the given *threshold*, it is assumed that the algorithm has reached to a minimum, thus the loop can be terminated. It is important to set the convergence threshold well, because if it is too large, then the algorithm may terminate before it reaches the minimum, if it is too small, then the algorithm may run for an excessive number of iterations[29].

When the gradient descent algorithm is run, it is vital to choose optimal values for both the learning rate and the convergence threshold. These values are not always the same, and it is problem-specific. It may require a few test runs to tune the parameters. To start tuning the parameters, one can start with 0.1 for the learning rate and a value of 0.0001 for the threshold. These values can be changed depending on the result, and one can choose a random set of numbers if wanted to do so.

4.3 Data Normalization

The gradient Descent algorithm comes with many advantages, but one thing that could put the algorithm back is the convergence of the algorithm. When the data has a high variance and is riddled, the gradient descent algorithm will have trouble converging to a minimum. To mitigate this issue data normalization is introduced which is also called *feature scaling*. Data normalization will increase the learning speed of the algorithm[36].

In the previous section 4.2.3 learning rate and convergence threshold were introduced. These two parameters are crucial to get to a global minimum in the cost function. Alongside these parameters, feature scaling is also another important aspect for the algorithm to converge quickly. There are many ways of applying feature scaling, for our case, we will apply *standard normalization*. This will mean that our features will end up having a 0 mean and a standard deviation of 1. This process will standardize the value of the features, thereby facilitating a more efficient convergence process.

Standard Normalization is done by replacing each data point x_n with its mean-centred unit deviation.

$$\frac{x_n - \mu}{\sigma} \rightarrow x_n \tag{4.7}$$

where the sample mean of data points μ is defined as:

$$\mu = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{4.8}$$

and the sample standard deviation of the data points σ is defined as:

$$\sigma = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(x_n - \mu)^2} \tag{4.9}$$

```
class StandardScaler():
  def __init__(self):
      self.means = None
      self.stds = None

  def fit(self, data):
      self.means = np.mean(data, axis=0)
      self.std = np.std(data, axis=0)

  def transform(self, data):
      return (data - self.means)/self.std

  def fit_transform(self, data):
      self.fit(data)
      return self.transform(data)
```

The code above is the implementation of the standard normalization. It is a straightforward implementation using the `numpy` library and the mathematical explanation provided before.

4.4 Computational Complexity

In the previous chapter we have introduced the least squares estimate using the matrix inversion. In this section, we will delve into the computational complexity of matrix inversion and how the gradient descent algorithm gives us better results in this sense.

4.4.1 Matrix Inversion Complexity

To recap, the computation for the least squares estimate was;

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{Z}^T\boldsymbol{Z})^{-1}(\boldsymbol{Z}^T\boldsymbol{Y})$$

where $\hat{\boldsymbol{\beta}}$ is the least square estimates, $\boldsymbol{Z}$ is the design matrix and $\boldsymbol{Y}$ is the observed values. When computational efficacy is considered, taking inverses of matrices is as particularly cumbersome as doing it manually. Specifically, when we are dealing with a matrix which is the size of (p x p), the computational time of the algorithm would be $O(p^3)$. However, if the matrix has a size of (p x k), the complexity of the algorithm can be $O(p^2k)$. This underscores that when p(number of features) gets large, the computational complexity can get too excessive[19].

4.4.2 Gradient Descent Complexity

When examining the gradient descent algorithm, it becomes apparent that few parameters affect the algorithm's runtime. However, if we were to calculate the computational complexity of each iteration, it amounts to $O(np)$ where n is the number of samples and p is the number of features. The dilemma with gradient descent lies in its sensitivity to datasets with different values for parameters such as the learning rate and the convergence threshold. These factors lead to a fluctuating complexity. Consequently, while the most important goal of the algorithm is that it ideally converges to a global minimum, this fluctuation implies that the final computational complexity is not always determinable.

4.4.3 Comparison

Gradient Descent and matrix inversion are two different methods of finding a solution for optimization problems. There are advantages and disadvantages of each approach. For instance, when we are dealing with a dataset where the matrix is (p x p) and p is very large, then matrix inversion is computationally too excessive with $O(p^3)$. Meanwhile, gradient descent, although it could take many more iterations, each iteration is computationally cheaper, having a complexity of $O(p^2)$. Matrix inversions also take up a lot of space in memory. It requires storing ($\boldsymbol{Z}^T\boldsymbol{Z}$) and its inverse, which can be a problem when dealing with large datasets. On the other hand gradient descent, only handles the current values of the parameters and the gradient. This means that it uses less memory compared to the matrix inversion. Moving forward, if we are dealing with a situation where our data gets updated regularly or have online learning, it will be computationally demanding to use matrix inversion. It would require to do matrix multiplication and inversion every time there is new data introduced. However gradient descent will keep updating its parameters and may not be as computationally demanding as matrix inversion. Furthermore, if there is no linear relationship between the independent and the dependent variables, then matrix inversion will not be able to create an accurate model, meanwhile gradient descent algorithm can handle the non-linearity between the variables.

Despite its computational complexity, matrix inversion has advantages in certain types of problems. When we try to create a model using the gradient descent method, we end up using parameters such as learning rate and convergence threshold, the value of these parameters is crucial for an appropriate model. Finding the right values for these parameters can take time using trial and error. Meanwhile, matrix inversion can provide an exact solution in one step, which would be both accurate and efficient if the dataset is not too large. Specifically for applications with a small number of features and samples, matrix inversion then can be the optimal choice due to its direct approach to solving linear equations.

In conclusion, while gradient descent can be favoured for datasets with large amounts of features and samples, matrix inversion has its advantages which can be fully leveraged. The choice between matrix inversion and gradient descent depends on the problem at hand, including the size of the dataset, the computational resources available, and the accuracy and simplicity needed for the solution.

4.5 Application of Gradient Descent

To observe the efficiency of gradient descent, we will use simulated data. This simulated data is created using the `sklearn` library and the `datasets` module, applying the `make_regression` function.

```
from sklearn.datasets import make_regression

# Generating a large artificial dataset using make_regression
np.random.seed(10) # For reproducibility
X, y = make_regression(n_samples=10000, n_features=10000, noise=100, bias=100)

# Adding a bias column with ones to X
X_b = np.c_[np.ones((X.shape[0], 1)), X]
```

Using the simulated data, we create our training and test datasets, by applying the function

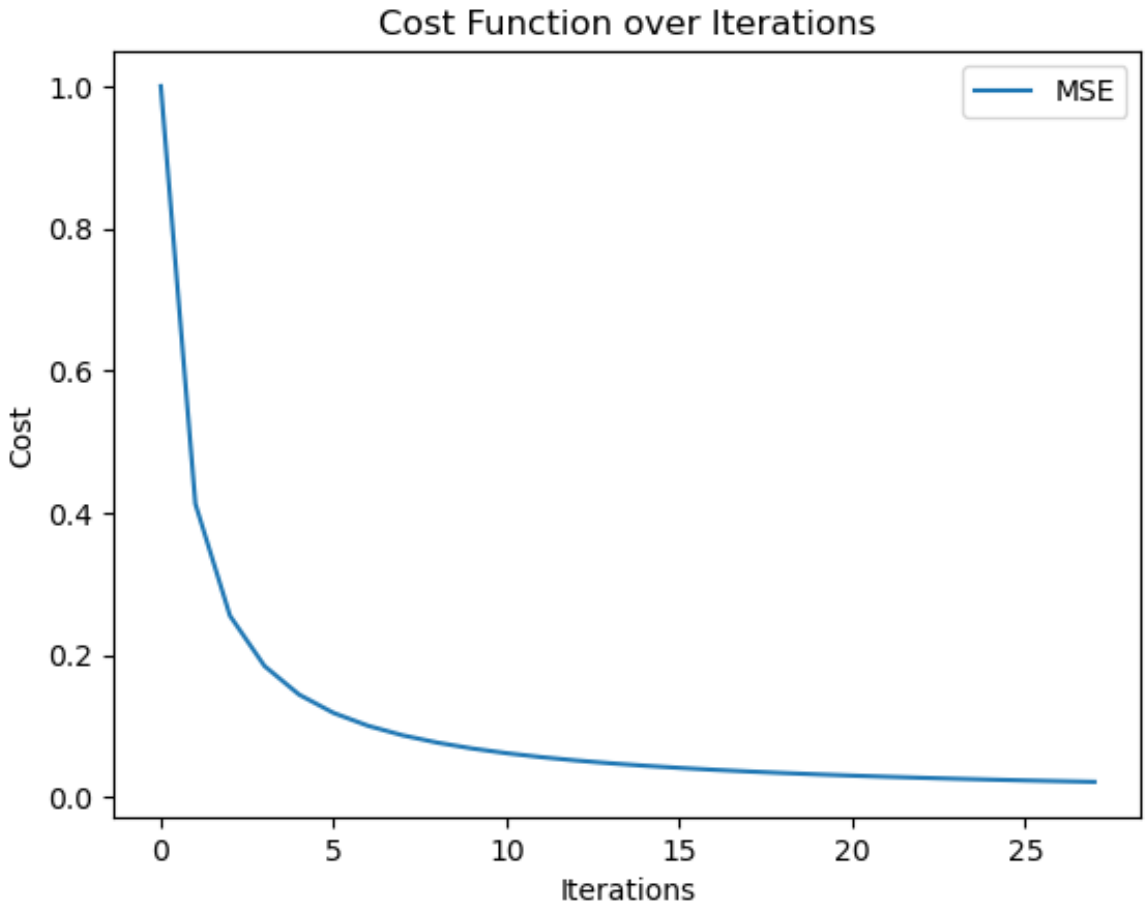

Figure 4.1: *Cost Function over Iterations*

`train_test_split()` from the `sklearn` library. Observing this simulation, the number of samples and the number of features is as large as 10000. This does not look computationally efficient using a learning algorithm based on matrix inversion. For this dataset, the computational complexity for matrix inversion is $O(n^3)$. Before applying the gradient descent algorithm, we will apply data normalization using the class we have created in the previous section. This is done to shorten the amount of time it takes to find the global minimum of the cost function. When we apply the normalized data to the gradient descent algorithm with a learning rate and a convergence threshold of 0.1, the graph of the cost function can be observed in Figure 4.1.

It is important to note that when we apply our linear regression function which uses matrix inversion, it takes longer than 5 minutes to return with a model. Meanwhile, the gradient descent method finds the global minimum in less than 5 seconds. This is a great example of computational efficiency between the two methods. Although the computational complexity can grow by changing the learning rate and the convergence threshold, it will not be as excessive as the matrix inversion. Note that this could change depending on the dataset.

Chapter 5: **Lasso Regression**

The previous chapters have introduced, Multiple Linear Regression and Ridge Regression. The multicollinearity analysis was made and Ridge Regression showed clear improvement to mitigate overfitting. This chapter will dive into the deeper fundamentals of Ridge Regression and compare it to the Least Absolute Shrinkage and Selection Operator, Lasso Regression. In the previous chapter 3, the Ridge Regression algorithm was coded based on matrix inversion and the formula -closed form- given was easy to code. In this chapter, the proof is shown of how to drive a closed-form formula for Ridge Regression.

Ridge Regression is based on an L2 penalty, this penalty will shrink the model coefficients towards 0 but will not set them to 0. This has been helpful so far in overcoming overfitting problems, but what if we want a simpler model? This means that if we are only interested in the features that have the most impact on the model, then Lasso Regression can be introduced with an L1 penalty. L1 penalty will zero out features and will make the model simpler[16].

Ridge and Lasso Regression penalties are based on the tuning parameter, λ. In this chapter, tuning parameter selection is talked through using cross-validation. The importance of data normalization will be clear when we are looking for the ideal tuning parameter.

A deeper comparative analysis is made by applying the models to a real-life dataset. The Extended Boston Housing dataset is applied to make a comparison between the Ridge and Lasso models. An idea of different advantages will be given between these two models.

5.1 Ridge Regression vs Lasso Regression

Ridge Regression **RR** effectively addresses the issue of overfitting in models. In this section, we will explore the unique advantages that Lasso Regression offers over the Ridge regression algorithm.

The **RR** incorporates all parameters, denoted as $\hat{\beta}$ into the model. Although **RR** shrinks these parameters towards 0, it does not set any parameter exactly to zero unless the penalty term, λ is infinite. This is called the **L2** penalty, in chapter 3 L2 penalty was introduced. L2 penalty adds the square of the magnitude of the coefficients to the cost function. L2 penalty might not pose a problem for making predictions, but it can complicate model interpretation, especially dealing with a large number of parameters $\hat{\beta}$[16]. For instance, in housing datasets, features such as the number of bedrooms, number of bathrooms, neighbourhood rating and year of construction are more indicative of a house's price than other variables. In such cases, the Lasso penalty effectively reduces the coefficients of less significant predictors to 0, thereby simplifying the model interpretation. Lasso penalty is also called the **L1** penalty. The L1 penalty works by adding the absolute value of the magnitude of the coefficients to the cost function. Basically, by setting some coefficients to 0, it selects a simpler model that excludes some features.

In summary, the L2 penalty shrinks the coefficients toward zero but does not set them to zero, it keeps all features with a moderated influence. Meanwhile, the L1 penalty can zero out coefficients and apply a feature selection.

5.1.1 Mathematical Approach

Comparing the formulation of **RR** to Lasso we can observe that they have similar formulations. The **RR** coefficient estimates are defined as:

$$\hat{\beta}_{Ridge} = \text{minimize}_{\beta} \left\{ \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\} \tag{5.1}$$

The Lasso estimator is defined as:

$$\hat{\beta}_{Lasso} = \text{minimize}_{\beta} \left\{ \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\} \tag{5.2}$$

It can be observed that the estimators for Lasso and Ridge are very similar. The difference is the L1 and the L2 penalties. Furthermore, it can be examined that the Ridge estimator is what is called the closed form of its formula. This closed-form approach can not be applied to the Lasso estimator because of its absolute value in the penalty term. This is because the absolute value function is not differentiable. Therefore, an optimization method like gradient descent is needed to find the values of the Lasso Regression coefficients.

The RSS is given by:

$$\text{RSS} = \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2$$

The RSS can also be written in the form of Euclidean distance;

$$\text{RSS} = ||\boldsymbol{Y} - \boldsymbol{Z\beta}||^2$$

Then the Lasso estimator can be defined as:

$$\hat{\beta}_{Lasso} = ||\boldsymbol{Y} - \boldsymbol{Z\beta}||^2 + \lambda ||\boldsymbol{\beta}|| \tag{5.3}$$

Where $||\boldsymbol{\beta}||$ is the L1 norm, which is the sum of absolute values of the vector[15].

5.2 Application of Lasso

The proof of concept programming of the Lasso estimator is very similar to the one of the gradient descent function that was coded a chapter ago. This time `max_iterations` parameter is given to handle the issues where the cost function does not converge. First, the given data is arranged, and then `theta` is created as a zero vector. Then the code follows the same logic as gradient descent but when calculating gradient, it also adds the product of `Lasso_scalar` parameter and the `sign(theta)` to the cost function. The `sign()` function, is used to compute the sign of each element in a given array. It returns -1 for negative values, 0 for zero and 1 for positive values[22]. This function is crucial for applying the L1 penalty as it determines the direction of the penalty for each coefficient. This means that the function is used to apply the appropriate sign to the penalty term that will be subtracted from the coefficients during the update step.

```
def lasso_gradient_descent(data, learning_rate, threshold,
                           mse_test, lasso_scalar,
                           max_iterations = 1000000):
  Z = data[:, :-1]
  ones = np.ones((Z.shape[0], 1))
  Z = np.hstack((ones, Z))
  Y = data[:, -1]

  theta = np.zeros(Z.shape[1])
  cost_function = []

  for i in range(max_iterations):
    y_hat = Z@(theta)
    mse = model_evaluation.mse(Y, y_hat)
    cost_function.append(mse)
    gradient = 2/len(Z)*Z.T@(y_hat - Y) + lasso_scalar*np.sign(theta)
    gradient_norm = np.linalg.norm(gradient)
    theta -= learning_rate*gradient

    if gradient_norm < threshold:
        break
```

5.3 Tuning Parameter Selection

Lasso and Ridge Regression have one thing in common which is the tuning parameter λ, which specifies the strength of the penalty term. To tackle the issue of selecting the best performing λ, cross-validation is introduced. Cross-validation divides the training data into subsets, also called *folds*. One of the folds is left out to test the model, this way the rest of the folds are tested on the one fold that is left out. In the context of selecting the tuning parameter, a range of λ values are tested using cross-validation. This way an optimal value for λ can be reached to [16].

```
def k_fold_cross_validation(model, data, k, score="R-Squared"):
  fold_size = len(data) // k # rounding to the nearest integer
  metrics = []

  for i in range(k):
      start, end = i * fold_size, (i+1) * fold_size
      test_data = data[start:end]
      train_data = np.concatenate([data[:start], data[end:]])

      model.train(train_data)
      predictions = model.test(test_data)["Predictions"]
      if score == "R-Squared":
          rss, tss = m_e.rss_tss(test_data, predictions)
          metric = m_e.r_squared(rss, tss)
          metrics.append(metric)
      elif score == "MSE":
          metric = m_e.mse(test_data[:, -1], predictions)
          metrics.append(metric)
  return np.mean(metrics)
```

```
lambda_values = [0.01, 0.1, 1, 10, 100]

final_model = m_s.lambda_ridge(data, 5, lambda_values)
final_model
```

```
53.71096889147755
32.98063920904538
21.423064166503238
29.40038302108283
45.55559343146818
Best lambda: 1, with score: 21.423064166503238
```

Figure 5.1: *The λ Value Applying Cross-Validation*

The above code for k-fold cross-validation is straightforward. First, the fold size is estimated by the given parameter `k`, based on this, training and test data sets are created. Next, the function initializes the model using the specified `model` parameter. Within the current loop, the model is tested with the available test data. By applying the evaluation metric, we can deduce a mean value of all evaluations that were conducted. This approach prevents us from relying on 'luck' to create a model, thereby strengthening our model's foundation.

As well as our model, we can also choose to apply this logic to our selection for the tuning parameter, λ. By going through a set of data with specific values of λ, we can deduce the best λ which the model works with. Compared to trial and error, applying cross-validation will make the model creation more automated and less time-consuming.

```
def lambda_ridge(data, k, lambda_values):
  best_lambda = None
  best_score = None
  for l in lambda_values:
      model = RidgeRegression(alpha=l)
      cv_score = k_fold_cross_validation(model, data, k, score='MSE')
      if best_score is None or cv_score < best_score:
          best_lambda = l
          best_score = cv_score
  final_model = RidgeRegression(alpha=best_lambda)
  return final_model
```

The code given above is an application of finding the best λ, value from a given array of λ values. This function uses the `k_fold_cross_validation()` function to find the λ value that returns the best model. Iterating through the lambda values, the score and λ are initialized using the values from the iteration. This function returns a model which has the `best_lambda` as its parameter. Observing *Figure 5.3*, it is shown that in a given array of λ values to be applied to various Ridge regression models, using the Boston Housing dataset, the best value for λ is 1. This means that a model with the tuning parameter set to 1 performs better than the models with other values for the tuning parameters. Further analysis shows that there is an overfitting issue that is present. This does show that, whilst Ridge regression mitigates overfitting to some extent, there is a risk that the model may not perform optimally after applying cross-validation. In conclusion, although cross-validation is a beneficial method for iterating through data to find the best model, it may not always return the optimal model with Ridge regression.

Normalizing the data can overcome the issue of not identifying the best-performing model after applying cross-validation. When normalized data is applied to cross-validation, it is easier to handle data and run the algorithms. It also provides an opportunity to analyse

the data thoroughly with much lower variance. The lower variance will facilitate a more accurate analysis. Modifying the given `k_fold_cross_validation()` function will add the `StandardScaler()` as follows.

```
def k_fold_cross_validation(model, data, k, score="R-Squared"):
  .
  .
  .
  scaler = StandardScaler() - added lines
  for i in range(k):
    start, end = i * fold_size, (i+1) * fold_size
    test_data = data[start:end]
    train_data = np.concatenate([data[:start], data[end:]])
    train_data_scaled = scaler.fit_transform(train_data) -> added lines
    test_data_scaled = scaler.transform(test_data) -> added lines
```

By adding these lines to the cross-validation function to utilise the `StandardScaler()`, the analysis becomes more robust. This is also crucial for the next implementation, which is for the *Lasso Regression*. In the previous section 4.3, we talked about how important data normalization is for gradient descent variants. When Lasso regression is run through the cross-validation, if the data is not normalized it will take a long time for it to converge to a minimum. By adding normalization to the cross-validation function we also manage to overcome this issue as well. Similarly, for the Lasso regression, a function is provided to iterate through an array of λ values. Then returns the model with the best-performing λ value. By automating the trial-and-error process of finding the best-performing model with a specific λ value, cross-validation is a useful tool to replace the trial-and-error approach.

5.4 Extended Boston Housing Dataset

In this section, the Extended Boston Housing dataset is applied to the Lasso Regression and Ridge Regression for comparative analysis. The Extended Boston Housing dataset is chosen for this analysis due to its large number of features, which Lasso Regression can be beneficial. In the analysis that is provided, it can be observed how the cost function converges within time for Lasso Regression and the comparison of coefficients between the Ridge Regression model and the Lasso Model can be observed. We can also compare the R^2 values and see how different coefficients result in different values. Firstly look at *Figure 5.2* it can be seen that a Lasso Regression, with a learning rate of 0.0001, a convergence threshold of 0.01, and $\lambda = 0.001$. From the figure the value of the cost function can be observed thoroughly, it is examined that the cost function reaches a minimum after 500000 iterations. The R^2 value of this model is 0.9242 which indicates a high level of model reliability. Notably, it can be seen that the L1 penalty has not reduced most of the coefficients. The coefficients suggest that the L1 penalty's strength is not pronounced in this example, however, upcoming examples will observe higher values of L1 penalty term insights. Observe the Ridge Regression model looking at *Figure 5.3*, and understand how Lasso and Ridge models are similar and different from each other. Examining Figure 5.2 and Figure 5.3 it can be observed that the values of coefficients are very different between the two models. We can deduce the difference L1 and L2 penalty terms provided to the models. This is because of L1 penalty zeros out some coefficients and the L2 penalty reduces the magnitude of the coefficients but does not set them to zero. Despite the spread of the coefficient values are considerably high, both of the models have high reliability. The Ridge Regression model has an R^2 value of 0.9447. To conclude the comparison between the regression algorithms, although the values of the coefficients are different, the model accuracy is very high in both of the models for the seen

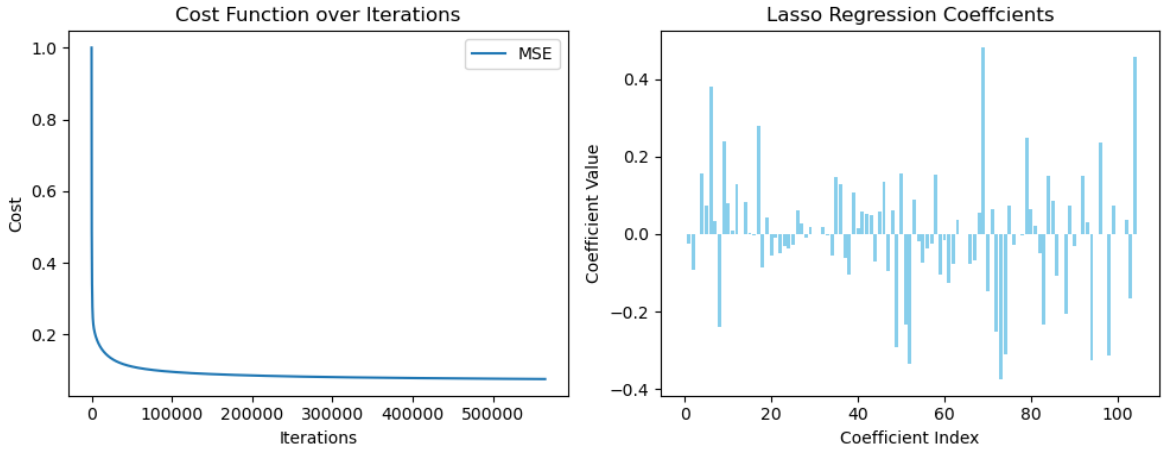

Figure 5.2: *Lasso Regression* $\lambda = 0.001$ *on Boston Housing*

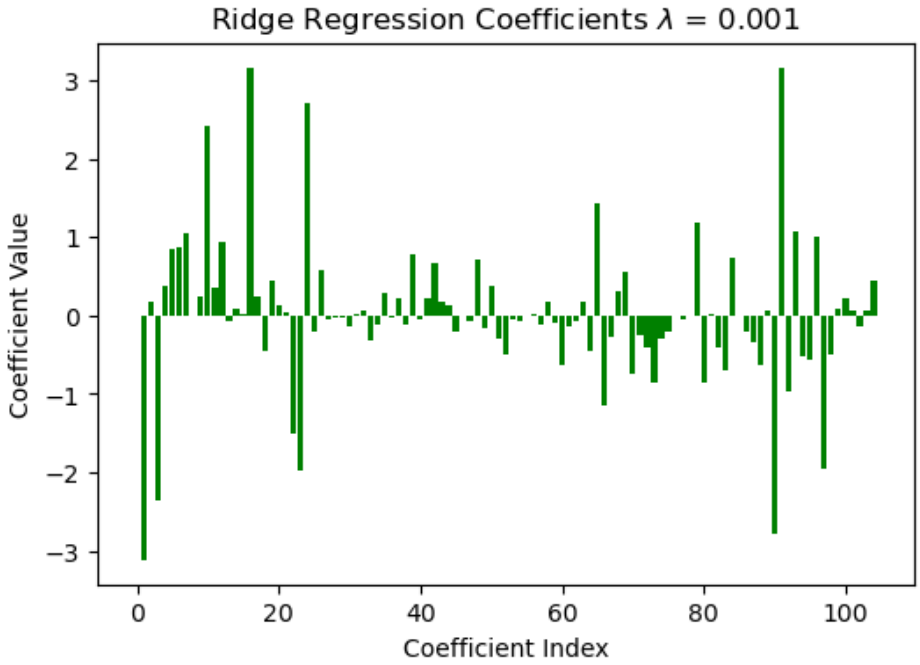

Figure 5.3: *Ridge Regression on Boston Housing*

data. For unseen data, the model can perform poorly, which will be discussed later in the chapter.

In *Figure 5.4* the Lasso regression model is with the same hyperparameters except for the tuning parameter with the value of $\lambda = 0.01$. The coefficient values and the value of the cost function over iterations can be followed from the figure. It is easy to analyse that as the tuning parameter value grows, more coefficients are zeroed out by the L1 penalty. Then looking at the next graph which is *Figure 5.5*, it can be seen that although many coefficients are close to zero, they are not zero out similar to the L1 penalty. This does show the difference between L1 and L2 penalties. Examining the R^2 values, meanwhile, the Lasso model has an R^2 value of 0.8997, and the R^2 value for the Ridge model is 0.9443. This time there is a slight difference between the evaluation of the models. The Ridge model looks more accurate on the training set, though we have seen before that sometimes a lower value in training can result in less overfitting compared to higher values of R^2 in the training of the model.

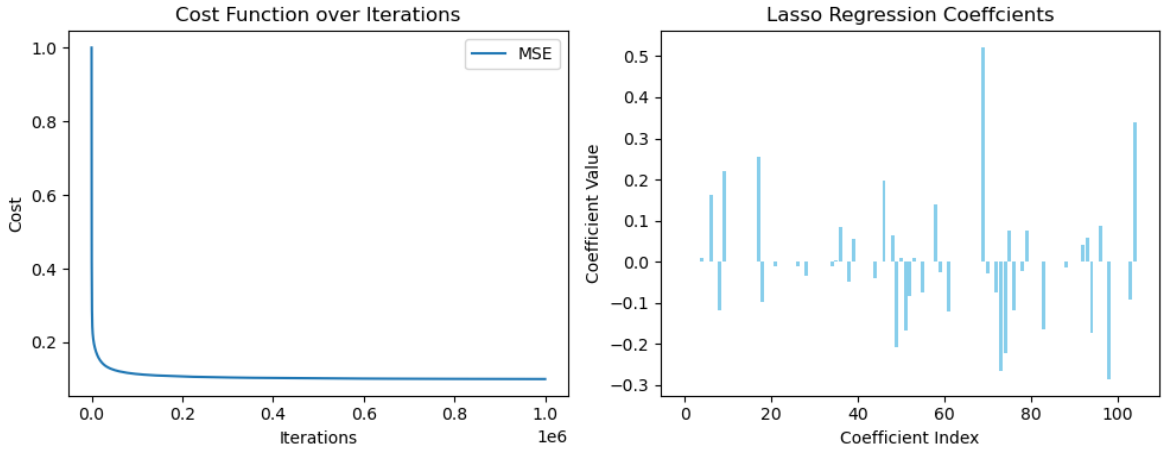

Figure 5.4: *Lasso Regression* $\lambda = 0.01$ *on Boston Housing*

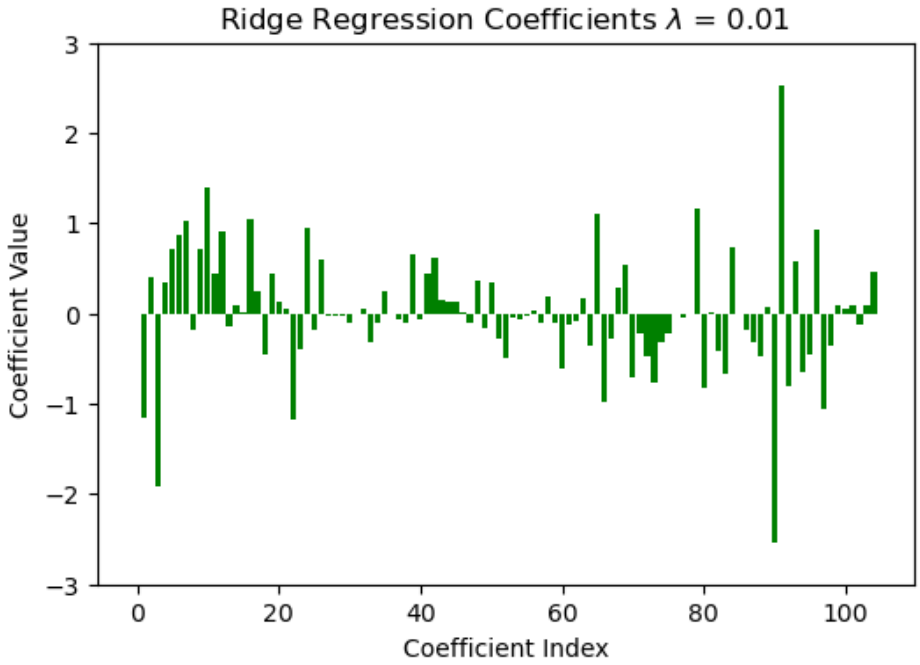

Figure 5.5: *Ridge Regression* $\lambda = 0.01$ *on Boston Housing*

This time for the Lasso model in *Figure 5.6*, the tuning parameter λ has a value of 1. The cost function value over iterations and the coefficients can be followed from the figure. From the figures and the given graphs, it is easy to deduce that when $\lambda \to \infty$ the coefficients $\to 0$. The larger the value of λ, the more features will be excluded from the model. Similarly in Ridge Regression, it was discussed before that larger values of λ will shrink the value of the coefficients towards zero. In *Figure 5.7* the coefficient values of the Ridge model can observed. All the coefficients are shrunk towards 0 and the impact of each feature is lower than the previous models with lower λ values. This is the key difference between the L1 and L2 penalty. As the value for the tuning parameter got larger, the L1 penalty excluded most of the features. Meanwhile, the L2 penalty reduced the effect of each coefficient towards zero. Examining the R^2 values for these models, the Lasso model has a value of 0.3888 which looks like a poor-performing model, and the Ridge model has a value of 0.9327, which still looks like a strong model with the given training data.

In conclusion, from the given figures, the difference in the coefficient values can be observed.

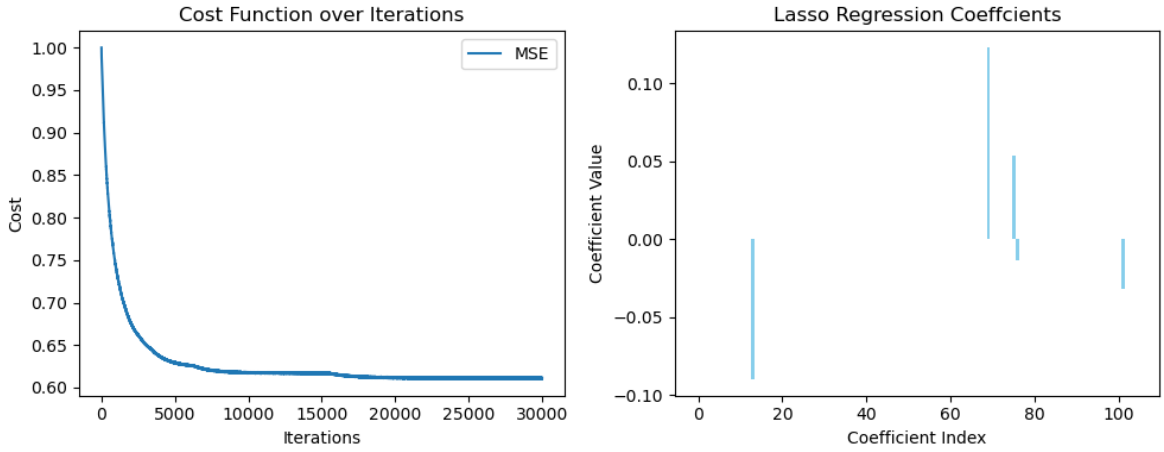

Figure 5.6: *Lasso Regression* $\lambda = 1$ *on Boston Housing*

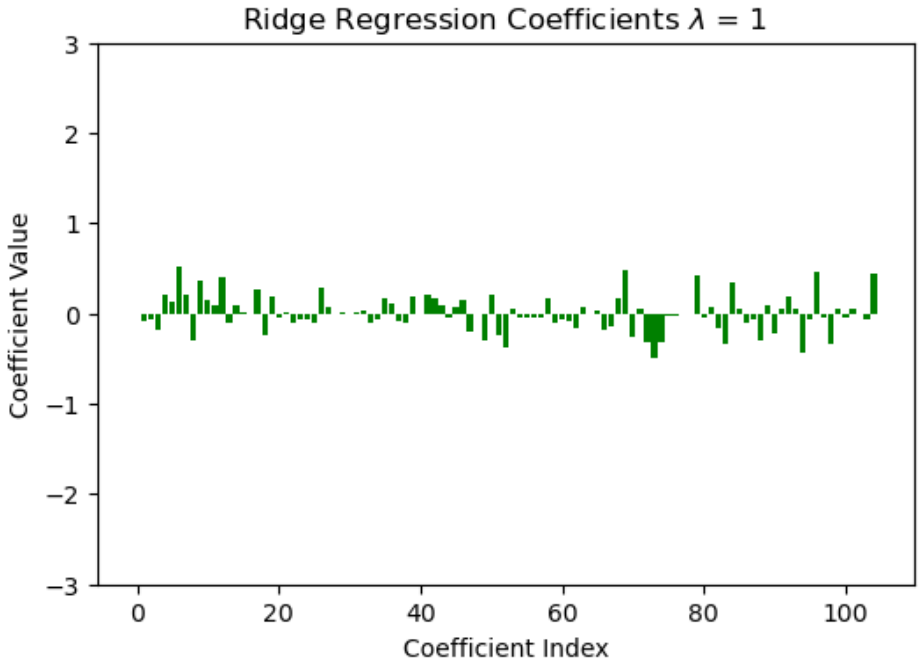

Figure 5.7: *Ridge Regression* $\lambda = 1$ *on Boston Housing*

There is a tendency for the Lasso models that the evaluation metrics such as R^2 got lower every time λ grew. This is expected as most of the features are being cancelled out, important features may also have been cancelled out. This will lower the R^2 score and result in a poor-performing model. Meanwhile, Ridge will still have a good foundation as all the features will be included in the model.

Moving forward, in this part of the section, the testing data will also be applied to the models. This way this will give a better understanding of which models perform better with which values. Observe that when $\lambda = 0.001$ the Ridge model has an R^2 value of 0.94 and when it is tested on unseen data, the R^2 value decreases to 0.83. Showing clear signs of overfitting. Applying various λ values and using trial and error, the best model can be deduced. Instead of this approach, the cross-validation method that was introduced in the previous section will be applied. In *Figure 5.8* given data is the Boston Housing data, and 5 folds are applied with an array of given λ values. As a result, the model with the value of 10 for λ is the best-performing model. This can be tested by training the model and testing the model on

```
lambda_values = [0.001, 0.01, 0.1, 1, 10]

final_model = m_s.lambda_ridge(data, 5, lambda_values)
final_model
```

```
1.7038962227207821
1.2966505782246256
0.719496812012608
0.6901385423679948
0.4988223722041951
Best lambda: 10, with score: 0.4988223722041951
```

Figure 5.8: *Ridge Regression Cross-Validation on Boston Housing*

```
final_model = model_selection.lambda_lasso(data, 5, lambda_values, model)
final_model
```

```
0.786132884498945
0.6995257631764643
0.45582689944997934
0.2914405359620017
0.9035434202925856
Best lambda: 0.1, with score: 0.2914405359620017
```

Figure 5.9: *Lasso Regression Cross-Validation on Boston Housing*

an unseen set of data. After training our model, the training model has an R^2 value of 0.90. Testing the model on unseen data, the model ends up being overfitted, with the R^2 value of 0.85. Furthermore, analysing all the λ values in the given array, it can be seen that the overfitting grows with other values of λ. By applying cross-validation with normalizing the data, the function returns the best possible model which has the least overfitting compared to the rest of the models. Looking at the Lasso model, the same logic can be applied. The analysis of different λ values will show overfitting and specifically when $\lambda = 1$, the model will not perform well at all - as it was shown before. When cross-validation is applied, for the best performing model, the best value for λ is 0.1 with a given set of hyperparameters, `learning_rate`=0.001, `threshold`=0.001, and `max_iterations`=15000. The evaluation metric for this model when it is trained is a score of R^2=0.81 and when the model is tested on unseen data the score is R^2= 0.81 too. This shows that with this specific Lasso model, the model will perform consistently even when there is unseen data that is introduced to the model.

Furthermore, if one has to choose between two of the models, one would have to consider what they are expecting from the model. If the model should have a high predictive score, then it would recommended that they should go with the Ridge model, as it had a better predictive score of 0.85 compared to the Lasso, 0.81. If the model should be consistent between seen data and unseen data and there should be low generalization, then the Lasso model seems to be the better option. If model simplicity is important, Lasso is the model to choose as it will zero out features with higher values of λ, thus creating simpler models. In case it is important to have all the model features, then the Ridge model will be a better option. At this point, it can be said that depending on what is expected from the model, model choice can be made.

5.5 Results

This section will discuss and analyse results from all of the models by applying them to the available `datasets` except for Washington. The New York Housing dataset and Extended Boston Housing dataset are used to analyse the models using various penalty coefficients (λ). Washington Housing dataset is not used for this analysis because from *Section 3.4* it was deduced that this dataset is not linear and the regression models do not perform well on this dataset. The Boston Housing dataset will be analysed using cross-validation and penalty coefficient parameter selection for better analysis, preventing the aspect of chance in model creation. Further, the California dataset will also be analysed to figure out the best possible model for this dataset, using cross-validation and penalty coefficient parameter selection.

Table 5.1: Comparison of R^2 scores for New York Housing

R^2 Scores/Model	Linear Regression	Ridge Regression	Lasso Regression
R^2 Train, $\lambda = 0.01$	0.6849	0.6849	0.6849
R^2 Test, $\lambda = 0.01$	0.6690	0.6690	0.6690
R^2 Train, $\lambda = 0.1$	0.6849	0.6849	0.6849
R^2 Test, $\lambda = 0.1$	0.6690	0.6693	0.6690
R^2 Train, $\lambda = 1$	0.6849	0.6849	0.6849
R^2 Test, $\lambda = 1$	0.6690	0.6722	0.6690
R^2 Train, $\lambda = 1000$	0.6849	0.2973	0.6849
R^2 Test, $\lambda = 1000$	0.6690	0.3673	0.6703

Table 5.2: Comparison of MSE scores for New York Housing

MSE Value/Model	Linear Regression	Ridge Regression	Lasso Regression
MSE Train, λ=0.01	113120×10^7	113120×10^7	113120×10^7
MSE Test, λ=0.01	88718×10^7	88709×10^7	88718×10^7
MSE Train, λ=0.1	113120×10^7	113120×10^7	113120×10^7
MSE Test, λ=0.1	88718×10^7	88628×10^7	88718×10^7
MSE Train, λ=1	113120×10^7	113144×10^7	113120×10^7
MSE Test, λ=1	88718×10^7	87868×10^7	88718×10^7
MSE Train, λ=10000	113120×10^7	252285×10^7	113136×10^7
MSE Test, λ=10000	88718×10^7	169580×10^7	88366×10^7

Examine the *Table 5.1* and *Table 5.2*, respective to the regression models, R^2 and MSE values are given using the New York dataset. Different λ values are provided to show the difference between different models. For this data, the Lasso model uses, the `learning_rate=0.01`, `threshold=0.1`, `max_iterations=1000000` as parameters. Remark that for Ridge or Lasso, when $\lambda = 0$ the model is simply the Linear Regression model. This can be proven from the tables. When the $\lambda = 0.01$, which is a low value, the R^2 and MSE values are all the same for training and test sets. This proves that as λ goes closer to 0, the Ridge and Lasso models will perform the same way as the Linear Regression model.

Next, when $\lambda = 0.1$, it can be observed that Lasso has the same scores as Linear Regression, but Ridge shows a slight difference in the test set scores. Similarly, when $\lambda = 1$ Ridge is the only model that has different scores compared to Linear Regression and in this case, it mitigates the slight overfitting there is between the training set and the test set.

Furthermore, $\lambda = 1000$ shows different results compared to the rest of the λ values. This value of λ penalizes the Ridge coefficients at a level such that the training of the model nor the test is accurate. Although Ridge does not perform well with this penalty coefficient,

the Lasso model seems to be performing well. The difference between training and test set scores is not wide and the accuracy is decent for both sets. This shows that there are penalty coefficients that work in benefit Lasso but not Ridge. This can also work and vice versa as well.

Table 5.3: Comparison of R^2 scores for Extended Boston Housing

R^2 Scores/Model	Linear Regression	Ridge Regression	Lasso Regression
R^2 Train, λ=0.001	0.9424	0.9412	0.9236
R^2 Test, λ=0.001	0.8386	0.8490	0.8475
R^2 Train, λ=0.01	0.9424	0.9329	0.9179
R^2 Test, λ=0.01	0.8386	0.8468	0.8507
R^2 Train, λ=0.1	0.9424	0.9127	0.8913
R^2 Test, λ=0.1	0.8386	0.8601	0.8519
R^2 Train, λ=1	0.9424	0.8624	0.8005
R^2 Test, λ=1	0.8386	0.8532	0.8042

Table 5.4: Comparison of MSE scores for Extended Boston Housing

MSE Value/Model	Linear Regression	Ridge Regression	Lasso Regression
MSE Train, λ=0.001	4.992058	5.094776	6.623917
MSE Test, λ=0.001	12.514891	11.708657	11.823062
MSE Train, λ=0.01	4.992058	5.817922	7.110313
MSE Test, λ=0.01	12.514891	11.877098	11.580209
MSE Train, λ=0.1	4.992058	7.565534	9.417083
MSE Test, λ=0.1	12.514891	10.852412	11.481876
MSE Train, λ=1	4.992058	11.920993	17.284409
MSE Test, λ=1	12.514891	11.380822	15.180454

Moving forward, the Extended Boston Housing dataset is analysed further with the *Table 5.3* and *Table 5.4*. The R^2 and MSE values of the models are given using the Extended Boston Housing dataset by applying different λ values. Similar to the New York dataset, when λ is as small as 0.001, the performance of all of the models is very similar to each other. As λ grows and becomes larger, the difference can be observed clearly. When $\lambda = 0.01$, All the models perform very well with the training model but the models do not perform well when the testing set is applied. This shows clear signs of overfitting. It can be seen that when $\lambda = 0.1$, Ridge and Lasso models are slowly mitigating the overfitting issue. Lasso model shows clear signs of improvement regarding overfitting although, the accuracy is lower the difference between the R^2 and MSE values is much less compared to Linear Regression and Ridge Regression. Finally, $\lambda = 1$ shows great improvement in generalization using the Ridge model. The accuracy of the model is very well in the training set and test set. The MSE is as low as 11.9 for the training set and 11.4 for the test set. Showing great accuracy and a very small difference between MSE and R^2 scores. Concluding that this model overcomes the multicollinearity issue presented by the Linear Regression model. Observe that Lasso Regression, similar to Ridge mitigates this issue. In this case, Lasso has a very low accuracy compared to the Ridge model. Lasso's feature selection could be eliminating too many features, thus losing relevant information. Comparing all the models to each other, it can be observed that the best model we have for the Extended Boston dataset is the Ridge model when $\lambda = 1$. Note that, when cross-validation is applied the results that can be obtained are more reliable.

Table 5.5 represents the analysis of the Boston Housing dataset by applying cross-validation. Results in earlier *Section 3.4* showed us there is an anomaly for the Boston Housing dataset, where the test scores were higher than the training scores. It was deduced that this could be

Table 5.5: Comparison of R^2 scores for Boston Housing using cross-validation

R^2 Scores/Model	Linear Regression	Ridge Regression	Lasso Regression
R^2 Train, $k = 3$	0.7431	0.7431	0.7285
R^2 Test, $k = 3$	0.7109	0.7109	0.6957
R^2 Train, $k = 5$	0.7426	0.7426	0.7278
R^2 Test, $k = 5$	0.6952	0.6952	0.6794
R^2 Train, $k = 10$	0.7420	0.7420	0.7268
R^2 Test, $k = 10$	0.6986	0.6987	0.6847
R^2 Train, $k = 20$	0.7414	0.7404	0.7261
R^2 Test, $k = 20$	0.6756	0.6761	0.6607

mitigated by cross-validation and now this mitigation can be analysed by the table given in the next page. In this table, corresponding R^2 scores are given for Linear, Ridge and Lasso models with a given k, which represents the number of folds for cross-validation.

Analysing Table 5.5 we can see that when $k = 3$, the models have decent accuracy. It can be observed that the accuracy for Linear and Ridge models is very similar. Looking at Lasso scores, although it has a lower accuracy compared to Linear and Ridge models, the Lasso manages to lower the difference between training and test scores, showing a better generalization compared to the rest of the models. Moving forward, the overfitting issue seems to be more present when $k = 5$ as the difference between the training scores and test scores is growing for all of the models. Although there is a slight improvement when $k = 10$ this improvement is not sufficient for better results. Finally, in the given table when $k = 20$, for all models the trend for overfitting grows and reaches its peak.

In conclusion, the results show us that when cross-validation is applied, the anomaly of having larger test score values compared to training is mitigated. The analysis using our models gives us a trend of overfitting as the k values grow. This can be due to the fact of minimising the amount of data that is tested in each fold as the value of k grows. When $k = 3$ we have decent results from the models. For the best model, Lasso could be preferred if one needs a model where the accuracy of the unseen data is important compared to the training data. If prediction accuracy is preferred, then one can choose Linear or Ridge.

Table 5.6: Comparison of MSE scores for California Housing using cross-validation

MSE Value/Model	Linear Regression	Ridge Regression	Lasso Regression
MSE Train, $k = 3$	0.3915	0.3915	0.4962
MSE Test, $k = 3$	0.5221	0.5221	0.4962
MSE Train, $k = 5$	0.3922	0.3922	0.5107
MSE Test, $k = 5$	0.5670	0.5669	0.5107
MSE Train, $k = 10$	0.3936	0.3936	0.4321
MSE Test, $k = 10$	0.3992	0.3993	0.4321
MSE Train, $k = 20$	0.3936	0.3937	0.4326
MSE Test, $k = 20$	0.3995	0.3995	0.4326

Table 5.6 represents the analysis of the California Housing dataset by applying cross-validation. Results in earlier *Section 3.4* showed us the models are not accurate and there could be outliers in the data. By cross-validation, we will try to improve the accuracy scores and get the best possible model. In this table, corresponding MSE scores are given for Linear, Ridge and Lasso models with a given k, which represents the number of folds for cross-validation.

Examining Table 5.6 for values ($k = 3, 5$) observe that the MSE values for training scores are as small as 0.39 for Linear and Ridge models, but test MSE values are as large as

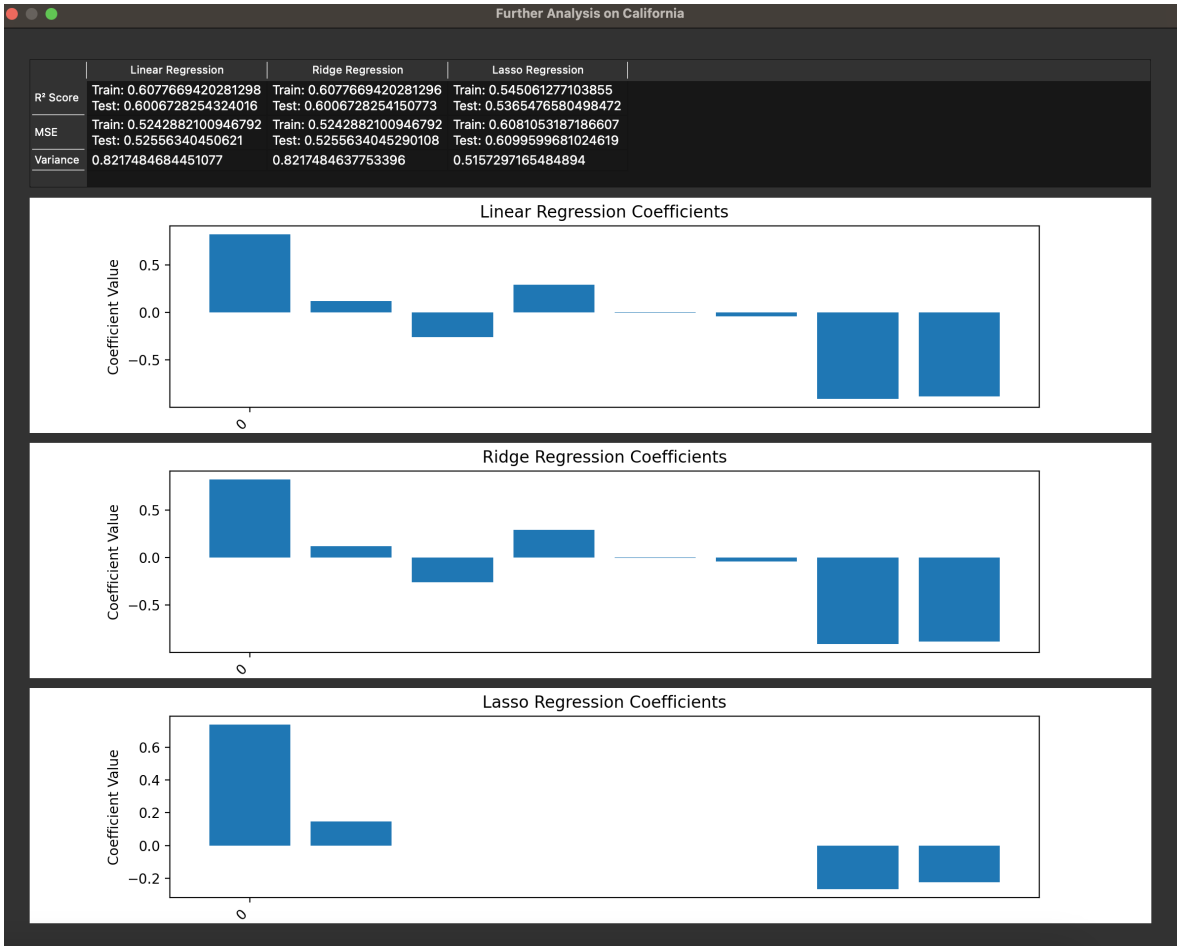

Figure 5.10: *Further Analysis on California Housing*

0.52. Showing clear signs of overfitting. When the Lasso model is observed, Lasso shows efficient generalization by removing some of the features, thus the difference of MSE values for training and test is very close to each other(the MSE scores are rounded to the nearest 4 decimal points). Moving forward, when the fold has values, $(k = 10, 20)$, the Linear and Ridge models are not overfitting anymore and the test MSE is as small as 0.40. On the other hand, the Lasso model seems to be consistent across all the k values regarding overfitting, but the model accuracy is lower compared to Linear and Ridge Regression models.

Furthermore, the coefficient values of the models can be examined from the `Figure 5.10`, which is a screenshot from the GUI. In this figure, the results are from the training and test sets that are created by using `train_test_split()` function with `random_state=17`. It can be seen that from the small table in Figure 5.10, there is no overfitting over the models and the accuracy of Linear and Ridge is higher compared to Lasso Regression similar to Table 5.6. From the figure, the model coefficient values for all the models can be compared. After applying cross-validation and finding the best penalty coefficient, these are the models with corresponding coefficients. From this figure, the difference between L1 and L2 penalty can be observed clearly. Using cross-validation, the selected λ values for Ridge and Lasso in these models are 0.00001 and 0.1 respectively. This will mean Ridge will not show a big difference in values of coefficients compared to the Linear model but the coefficients will be shrunk regardless. Meanwhile, Lasso will be more effective and will remove several features, and it can be seen that Lasso has removed 4 features from the model with an L1 penalty. Meanwhile, Ridge has shrunk the model coefficients compared to Linear Regression with the L2 penalty.

Thus we can conclude that, for the California Housing dataset when cross-validation is applied, unlike the Boston Housing, as the value of k grows we end up having more accurate models with less overfitting. It can be observed that when $k = 10$, Linear and Ridge Regression models are decent and there are no overfitting issues. This means that Linear or Ridge Regression models can be chosen to make future predictions.

Chapter 6: **Inductive Conformal Prediction and Cross-Conformal Prediction**

This chapter introduces conformal prediction with its specific properties and approach to predictions made from models. Inductive conformal prediction is then introduced, and an implementation of the algorithm is made using the "quantile" approach. Furthermore, the accuracy of the inductive conformal prediction and prediction intervals are made by analysing the error rate and the calibration graph. Next, cross-conformal prediction is introduced for efficiency and better accuracy. Cross-conformal prediction is implemented as an "extension" of the inductive conformal prediction and further analysis is made comparing the prediction intervals of inductive conformal prediction and cross-conformal prediction.

6.1 Conformal Prediction

Conformal prediction is a statistical approach that is based on past data to assign confidence measures to new predictions. Given a specific confidence interval, conformal prediction will return an interval of values assigned to the prediction. Conformal prediction works with any nonconformity score estimating function. The "Absolute Error" value is used to estimate the nonconformity scores in this implementation. By using the nonconformity scores, conformal prediction assesses how much a new example deviates from previous examples. This methodology generates prediction intervals that contain the true value with a predetermined probability[35]. With the goal of prediction intervals, nonconformity scores and the quantile estimate are needed to estimate the values of the intervals.

Nonconformity score is a way to understand how different a new data is compared to the old data. It is a way of measuring the "strangeness" of new data compared to a set of old data. For a new data point, its nonconformity score could be measured by the absolute value difference of the new data point ($\hat{y}_i$) prediction and the actual data point (y_i) value (for i^{th} data point).

$$\alpha_i = |y_i - \hat{y}_i|$$

This way it can be seen how far away the new example is from the existing data points[35]. In a regression model quantile is estimated using the nonconformity scores depending on a significance level. The significance level is directly used to determine the quantile of the nonconformity scores.

With the objective to understand how "strange" new data is, p-values are needed to be estimated. After estimating the nonconformity score for the new data, this score can be used further to assess whether the data is "strange" or "conforming". To determine whether the data is conforming or not, the nonconformity scores are compared. The goal is to find out what fraction of the data set's nonconformity scores are as high as or higher than the score of the new data. This fraction is called the p-value. If the p-value is very small, the data is said to be "strange" compared to the rest. Conversely, if it is large the data fits in well. This way, it can be understood if the data is an outlier (nonconforming) or not (conforming)[35][27]. In a classification problem, p-values can be used to determine the label or a set of labels for predictions. P-values can be calculated by the given formula:

$$p(y) = \frac{\#\{i = 1, \ldots, n+1 \mid \alpha_i^y \geq \alpha_{n+1}^y\}}{n+1} \tag{6.1}$$

Where α_i is the i^{th} nonconformity score.

Quantiles that are obtained from nonconformity scores are used to generate prediction intervals. This interval is based on a given significance level, for example, 0.05% meaning that the true value will fall in this range 95% of the time. This will provide a statistically valid way to account for the uncertainty in the predictions[35].

Conformal prediction is a state-of-the-art statistical method that provides reliability for the predictive models. This is crucial for a data-driven world. Its advantage lies in its ability to offer theoretically guaranteed measures of confidence intervals for each prediction it makes. This is useful in fields like healthcare, finance, face recognition systems, etc. where decisions based on predictions can have significant consequences.

For instance, in the healthcare industry, conformal prediction can be used to diagnose patients with given confidence intervals[35]. It also can be applied to estimate the efficacy of a new treatment. This way doctors can understand the uncertainty that is associated with the new treatment. This can help the doctors do a better risk assessment and make more informed decisions. Similarly, in financial markets such as forecasting stock prices, conformal prediction can be useful. Conformal prediction can assist traders and investors about the risks and reliability of their model's predictions. This way conformal predictors can guide investment strategies by indicating when a prediction is particularly trustworthy or when caution should be applied due to uncertainty.

6.2 Inductive Conformal Prediction Application

In this section, *Inductive Conformal Prediction* is implemented. **ICP** is chosen instead of *Conformal Prediction* due to its computational efficiency compared to **CP**[33] and **CP** can not be applied to all regression algorithms [11]. To understand the difference between **ICP** and **CP**, the training data is separated into two different sets, the calibration set and the new training set. To calculate the nonconformity scores and the p-values for **ICP**, the calibration set is used instead of the training set.

```
def nonconformity_scores(self, train_data):
  if self.beta_hat is None:
      raise ValueError("Model must be trianed before measuring
      nonconformity scores.")
  y = train_data[:, -1]
  y_hat = self.predictions
  return sorted(np.abs(y_hat - y))
```

Given the code above, this is the function to estimate the nonconformity scores given training data and a list of predictions. This function is part of the `RidgeRegression()` class and can be called after a ridge regression model is trained. From earlier the definition of nonconformity scores, we apply the logic in the function. By looking at the absolute difference between our predictions and the observed values, we get the nonconformity scores for each of the predictions that are made. The scores are then used to find the p-values and generate the confidence intervals. The same logic and function is applied to the `LassoRegression` class.

```
def p_vals(self, nonconformity_scores, train_data, new_data, predictions):
  p_values = []
  n = train_data.shape[0]
  y = new_data[:, -1]
  test_nonconformity_scores = np.abs(y - predictions)
```

```
    for test_nonconformity_score in test_nonconformity_scores:
        count = 0
        for nonconformity_score in nonconformity_scores:
            if test_nonconformity_score >= nonconformity_score:
                count += 1
        p_value = count/n
        p_values.append(p_value)
    return p_values
```

Moving forward, p-values are crucial for determining if the prediction is conforming or not. From the earlier definition given, it can be deduced that the p-values are the proportion of training examples that are at least as nonconforming as the new prediction[27]. In the given code above, p-values are determined by making comparisons between the nonconformity scores. The nonconformity scores in this case are calculated based on the `nonconformity_scores()` function. Thus, the nonconformity scores of the training data will be passed to the `p_vals()` function. Depending on the conditional statement p-value is estimated by incrementing the count variable. At last, the function returns a list of p-values. The returned p-values can be used further for analysis of the predictions that were made.

```
def conformal_prediction(self, train_data, calibration_data, test_data,
  significance_level):
    self.train(train_data)

    # Using the calibration data, estimate the nonconformity scores
    calibration_predictions = self.predict(calibration_data)
    nonconformity_scores = self.nonconformity_scores(calibration_data,
    calibration_predictions)

    # Apply the test data to make predictions
    test_predictions = self.predict(test_data)

    # Create intervals using the nonconformity scores
    prediction_intervals = []
    quantile = np.quantile(nonconformity_scores, significance_level)
    # Given the confidence level, returns specific quantiles according
    to the nonconformity scores
    for prediction in test_predictions:
        lower_bound = prediction - quantile
        upper_bound = prediction + quantile
        prediction_intervals.append((lower_bound, upper_bound))
    return prediction_intervals
```

In order to finalize the **ICP** and generate confidence intervals, the function given above is applied. The function is passed as training data, calibration data, test data, and a significance level. The algorithm begins by training the model with the given `training_data`. The nonconformity scores, which are used to generate prediction intervals, should not be estimated using the training dataset. Instead, they should be calculated by another small dataset, referred to as the `calibration_data`. The calibration data can be obtained by using the `sklearn` library's, `train_test_split()` function. Separated from the training data, the calibration dataset is used to calculate the nonconformity scores. Nonconformity scores can be determined by applying the `nonconformity_scores()` function. Moving forward, the given test data is applied to the trained model to make predictions based on the `test_data`. Then, an empty `prediction_intervals` list is created. Next, the `quantile` is estimated by

```
prediction_intervals = model.conformal_prediction(new_train_data, calibration_data, test_data, 0.90)
model.calculate_error_rate(test_data, prediction_intervals)
```

```
0.10236220472440945
```

Figure 6.1: *Error rate,* $\gamma = 0.10$ *with a Ridge Regression Model*

using the `numpy` library's, `quantile()` function. A *quantile* is where data is divided into equal-sized subsets[30]. This function first sorts the nonconformity scores in an ascending order. Then, it calculates the quantile, q, by $q = \gamma(n)$ where γ is the significance level, and n is the number of nonconformity scores [23]. From the given `quantile`, we can deduce the intervals. The quantile approach gives us a more efficient algorithm. This method will be walked through in detail in the *Section 6.4*. The algorithm continues with a `for` loop. The *lower bounds* and *upper bounds* of the prediction intervals are estimated using the estimated `quantile` value. Moving forward, the code iterates through the predictions based on test data. Then, the prediction interval is returned. The interval is symmetric around the prediction, meaning that it is calculated by subtracting and adding the quantiles to the prediction. The same function is also implemented to `LassoRegression()` class and can be used the same way as `RidgeRegression()` class.

6.3 Inductive Conformal Prediction Interval Accuracy

The intervals can be found with the given function in the previous section. To understand if the given intervals are accurate or not, there are several methods we will use to analyse the accuracy. The *error rates* will be calculated to understand how much of the observed values fall inside the prediction interval. Usually, the error rate will be around the value of the significance level. For instance, given a confidence level of 0.95% - which is (1-significance level (γ)), the error rate for the prediction intervals should be around ≈ 0.05. Another method to understand if the prediction interval is accurate is to plot a calibration curve. This graph plots observed values falling in the intervals against the expected values based on the confidence levels[34]. By drawing the line, $y = x$ on the graph, the prediction results can be assessed respectively. If the prediction interval is around the $y = x$ line or on the line, this is considered to be an accurate interval. More analysis can also be made, such as if the interval is too wide or too narrow. The rest of this section will prove how accurate the predictions are by analysing the error rate and the calibration graph.

Observing the *Figure 6.1*, the prediction intervals for a Ridge Regression model using the Extended Boston Housing dataset, are estimated by `conformal_prediction()` function with a confidence level of 90%. Then the `calculate_error_rate()` function is used to estimate the error rate of the prediction intervals. The result is approximately 0.10 or 10%. This shows that the observed values that fall outside of the intervals are only 10% of the whole intervals. The result indicates that the prediction intervals are accurate. From the given *Figure 6.2* the calibration curve can be examined. It can also be observed that the calibration curve represents the (1-error rate) proportion compared to the error rate. On the graph, for the confidence level of 90%, the proportion of observed values within prediction intervals is on the $y = x$ line. This also shows that the prediction interval is accurate. Moving forward, similar logic is applied for the confidence level, 99%. Similarly, the results can be observed from *Figure 6.2* from the calibration curve and *Figure 6.3*. The error rate is approximately 0.008 and it is very close to 0.01 to the significance level, which is $\gamma = 0.01$. This is the value required to be a well-calibrated interval. As well as calibration, the prediction set should also be small for more confident analysis[32]. Furthermore, the proportion of observed values within prediction intervals is on the line as expected and it does show that when the confidence

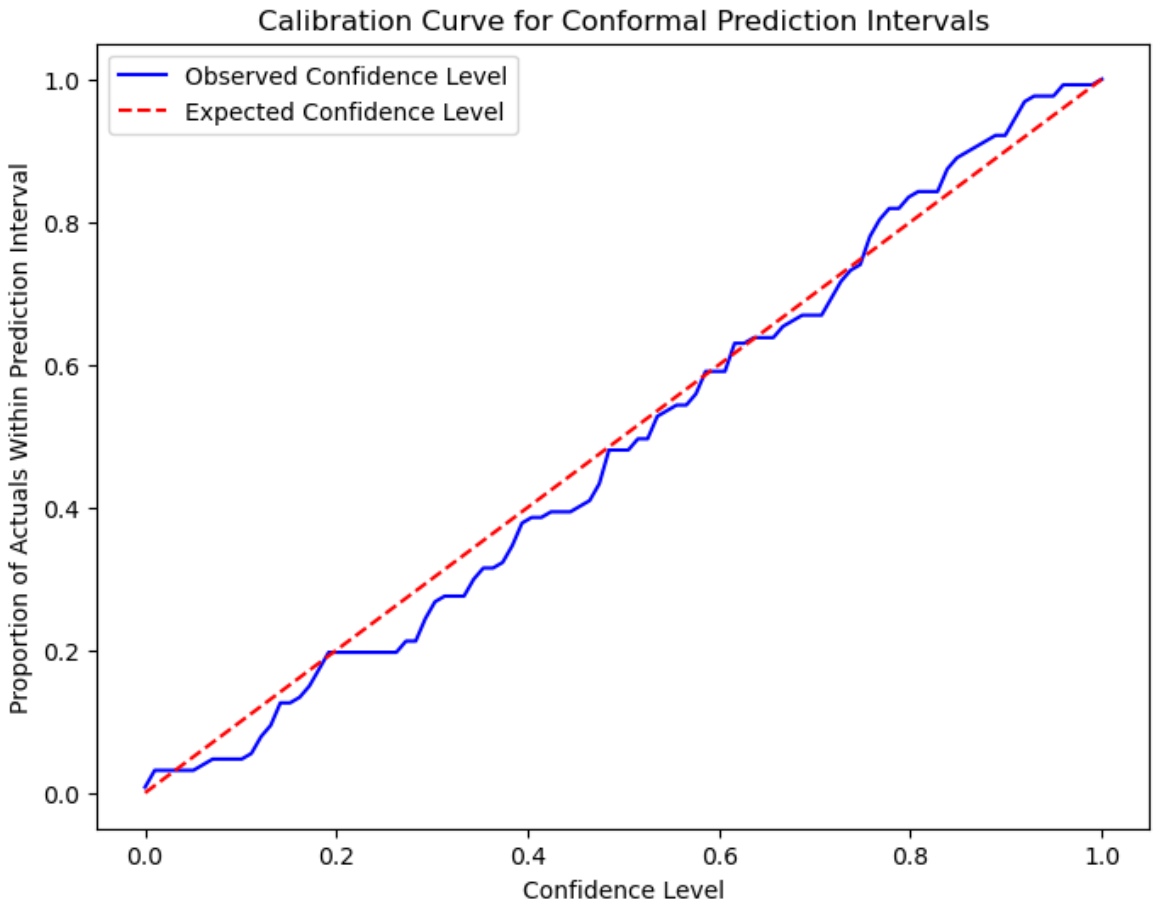

Figure 6.2: *Calibration Curve, Inductive Conformal Prediction with a Ridge Regression Model*

```
prediction_intervals = model.conformal_prediction(new_train_data, calibration_data, test_data, 0.99)
model.calculate_error_rate(test_data, prediction_intervals)
```

```
0.007874015748031496
```

Figure 6.3: *Error rate,* $\gamma = 0.01$ *with a Ridge Regression Model*

level is 99%. Thus, the Ridge Regression model makes accurate intervals as required.

So far, **ICP** is introduced, and it is applied to models such as Ridge Regression and Lasso Regression. A Conformal Prediction algorithm that is implemented is good for being able to make predictions based on a predetermined probability. The accuracy of the intervals is checked with the error rate and calibration graph, but what if we wanted to be more concise and have narrower prediction intervals? Next, cross-conformal prediction will be implemented and the power of cross-validation can be seen again.

6.4 Cross-Conformal Prediction

ICP was a good mitigation for adaptability to different regression models and computationally efficient. The downside of the **ICP** is that compared to **CP**, it may suffer from accuracy in prediction intervals. To mitigate both computational efficiency and prediction accuracy, cross-conformal prediction is introduced[33]. In Chapter 5, *Section 5.3*, tuning parameter selection was introduced based on cross-validation. Similar logic can be applied to the **ICP**

and extended it using the cross-validation[11].

Compared to the standard approach of conformal prediction, where the intervals are returned depending on p-values, in this project the implementation has the quantile approach. Quantile was introduced in the previous section, and this approach shows that it is computationally more efficient when applied to **CCP**. In the standard approach given in Vladimir Vovk's machine learning module, the intervals are estimated using the p-values. When this is applied to cross-conformal prediction for each prediction the p-values are needed. The calculation of each p-value, for every prediction can be computationally extensive. By using quantiles intervals are approximated based on the distribution of the nonconformity scores across all folds. Applying the IID (Independent and Identically Distributed) assumption of the data, we can also assume that the nonconformity scores are stable across all folds. Given a confidence level, a quantile of the nonconformity scores is found. This quantile will be a nonconformity score such that a given significance level, γ a proportion of nonconformity scores will fall below it[30]. The downside of this method is that, if the nonconformity scores exhibit outliers or anomalies, the quantile might be influenced. This can cause less accuracy in the prediction intervals.

In summary, **CCP** is introduced to find more accurate prediction intervals with computational efficiency. **CCP** is an extension of **ICP** and can be applied in a similar logic by applying cross-validation. Unlike the standard **CP** approach using p-values, in this project, the implementations are based on quantiles.

6.5 Application of Cross-Conformal Prediction

In this section how **ICP**, that was implemented in the previous section, can be extended to **CCP** implementation will be shown. In the code given below, `cross_conformal_prediction` function takes parameters `k`, `data`, `new_data`, and `significance_level`. The parameter `k` is the fold value, and depending on the value of the `k`, `data` is folded into `k` parts. `data` is the whole dataset that is needed to be analysed. `new_data` is the data we want to introduce to the model so prediction intervals can be made using the model. Lastly, `significance_level` is the determined probability to find the interval bounds. The main difference from **ICP** is that **CCP** applies cross-validation. It uses the `sklearn` library's `KFold()` function to separate the data into subsets. A fold will be used as a calibration set and the rest will be used as the training data in each iteration. Another fact that distinguishes **CCP** from **ICP** is scaling data when finding the nonconformity scores. This is important in cross-validation algorithms to make the algorithm more efficient and accurate. Then the nonconformity scores are calculated using the calibration set for each fold. Next, by training the model we make predictions using the given `new_data`. Quantile is deduced in the same way as in the **ICP** algorithm. Moving forward, the algorithm iterates through the list of predictions and the boundaries of the interval are deduced using the quantile.

```
def cross_conformal_prediction(self, k, data, new_data, significance_level):
  kf = KFold(n_splits=k, shuffle=True, random_state=17)
  scaler = StandardScaler()

  # This will accumulate all nonconformity scores across all folds
  nonconformity_scores = []

  # Perform k-fold cross-validation
  for train_index, test_index in kf.split(data):
      # Split data to training and calibration data
```

```
        train_data, calibration_data = data[train_index], data[test_index]

        # Scale data
        X_train, Y_train = scaler.fit_transform(train_data[:, :-1]),
                                                train_data [:, -1]
        X_cal, Y_cal = scaler.transform(calibration_data[:, :-1]),
                                        calibration_data[:, -1]

        # Recombine feature and target data
        train_data = np.hstack((X_train, Y_train.reshape(-1, 1)))
        calibration_data = np.hstack((X_cal, Y_cal.reshape(-1, 1)))

        # Train data and estimate the nonconformity scores based
        on calibration data
        self.train(train_data)
        calibration_predictions = self.predict(calibration_data)
        nc = self.nonconformity_scores(calibration_data,
                                       calibration_predictions)
        nonconformity_scores.extend(nc)

    # Train the model on the whole dataset and predict values on a
    given new data
    self.train(data)
    test_predictions = self.predict(new_data)

    # Similar to the conformal prediction function the intervals are
    estimated using quantiles
    prediction_intervals = []
    quantile = np.quantile(nonconformity_scores, significance_level)
    for prediction in test_predictions:
        lower_bound = prediction - quantile
        upper_bound = prediction + quantile
        prediction_intervals.append((lower_bound, upper_bound))
    return prediction_intervals
```

Furthermore, similar to **ICP**, for **CCP** the prediction intervals can be tested by observing the error rate and the calibration curve.

6.6 Cross-Conformal Prediction Accuracy

In the previous section, testing the accuracy of **ICP** showed that there is a linear relationship between the proportion of observed values that fall inside the intervals and the expected confidence level if the prediction intervals are accurate. For instance, a confidence level, of 0.95 should have 95% of the observed values in the prediction intervals.

Examine the given *Figure 6.4*, using a Ridge Regression model trained on the `Extended Boston Housing dataset`, and the calibration curve is given. The confidence level is a set of evenly spaced 100 confidence levels from 0 to 1. The graph shows us for each confidence level the, proportion of the observed values that fall inside the prediction intervals. It is shown in the graph that the calibration curve follows the $y = x$ as required. This is an indicator that our prediction intervals have a decent accuracy.

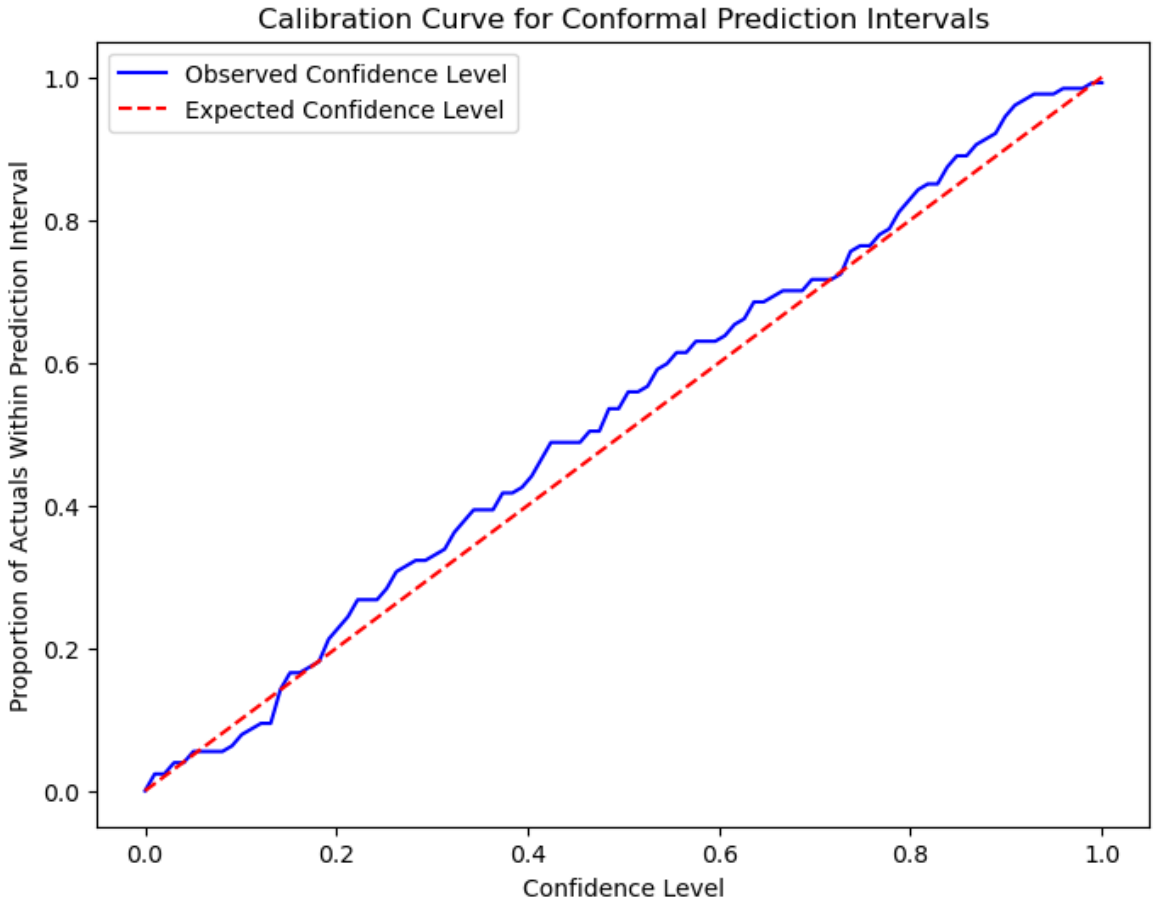

Figure 6.4: *Calibration Curve of the CCP with a Ridge Regression Model*

Moreover, the error rate of the **CCP** can be compared to **ICP**. Observing, *Figure 6.6*, shows us the error rates with confidence levels, 0.90 and 0.95 respectively. Remark that the respective error rates were, for confidence levels, 0.90 was around 10% and for 0.99 was around 0.8% levels. These results can be compared to the information given in *Figure 6.6*. Observe that the error rate for confidence level 0.99 is approximately 0.8%, similar to the result from the **ICP**. This shows that both of the prediction intervals acquired from **CCP** and **ICP** are accurate. Moving forward, observe the error rate for confidence level, 0.90 in *Figure 6.6*. The error rate is around 5.5% and it is different than the error rate that was obtained from the **ICP**. This means that even though the confidence level is 90%, observed values fall in the intervals 94.5% of the time. Compared to the 10% threshold that is given by the confidence level, the **CCP** made fewer errors in prediction intervals. This can be interpreted in several ways, either the intervals are more conservative compared to the one of the conformal prediction and results in wider intervals or the intervals are narrower but

```
prediction_intervals = rrmodel.cross_conformal_prediction(5, data, test_data, 0.90)
rrmodel.calculate_error_rate(test_data, prediction_intervals)
```

```
0.05511811023622047
```

```
prediction_intervals = rrmodel.cross_conformal_prediction(5, data, test_data, 0.99)
rrmodel.calculate_error_rate(test_data, prediction_intervals)
```

```
0.007874015748031496
```

Figure 6.5: *The Error Rates of the CCP,* $\gamma = 0.10, 0.05$ *with a Ridge Regression Model*

the accuracy is higher. A short statistical analysis can be made by analysing the intervals.

To understand whether **CCP** is narrowing the intervals or not, the interval widths can be computed by finding the difference between the upper bound and the lower bound of the intervals. Then, the mean of the interval widths can be computed for further analysis. Let the mean of the interval widths be, μ_{icp} and μ_{ccp} respectively. Moving forward, after finding the mean of the interval widths for **CCP** -$k = 5$ for folds- and **ICP**, when the confidence level is 0.90, the $\mu_{icp} \approx 11.21$ and $\mu_{ccp} \approx 9.55$. Shows that **CCP** has narrow intervals compared to **ICP**. This makes the **CCP** more accurate and more efficient with narrower intervals. This will be delved further in the next section.

In conclusion, **CCP** brings another perspective to conformal prediction by combining cross-validation with the **ICP** method. It has been shown that **CCP** can come up with narrower and more accurate prediction intervals compared to **ICP**. An in-detail analysis can be made using the statistical properties of the prediction intervals to conclude the comparative analysis between **CCP** and **ICP**.

6.7 Results

In this section, we will analyse further the difference between **ICP** and **CCP**. The mean width and the error rate of the prediction intervals will be compared by using different datasets. The best models from the results that were gathered from *Section 5.5* will be used for conformal prediction for each dataset. This will mean that Ridge ($\lambda = 1$) will be used for the New York Housing and Extended Boston Housing dataset, Ridge ($\lambda = 10$) will be used for Boston Housing, and Linear Regression will be used for California Housing.

Method/Measure	**Mean Width**			**Errors (%)**		
	90%	**95%**	**99%**	**90%**	**95%**	**99%**
ICP	2800601	4981366	8116084	0.125	0.0147	0.0073
CCP $k = 3$	3464865	4466636	6934533	0.0588	0.0221	0.0074
CCP $k = 5$	3360657	4330331	6732799	0.0735	0.0221	0.0074
CCP $k = 10$	3344532	4323354	6713494	0.0735	0.0221	0.0074
CCP $k = 20$	3390318	4361738	6722107	0.0662	0.0221	0.0074

Table 6.1: Ridge Regression CP with New York Housing

Method/Measure	**Mean Width**			**Errors (%)**		
	90%	**95%**	**99%**	**90%**	**95%**	**99%**
ICP	11.209	15.353	32.489	0.0314	0.0157	0.0078
CCP $k = 3$	10.569	13.501	21.850	0.0394	0.0236	0.0079
CCP $k = 5$	9.974	12.131	24.733	0.0551	0.0236	0.0079
CCP $k = 10$	10.078	12.895	25.280	0.0551	0.0236	0.0079
CCP $k = 20$	10.598	12.900	25.630	0.0394	0.0236	0.0079

Table 6.2: Ridge Regression CP with Extended Boston Housing

Given the tables *Table 6.1* and *Table 6.2*, the mean interval widths are given with error rates for Inductive Conformal Prediction and Cross-Conformal Prediction with different values of k. The model that is used is the best Ridge model we have observed from *Section 5.5*, where $\lambda = 1$. This model is applied to both datasets, New York and Extended Boston Housing. From both of the tables, it can be observed that, as the confidence level is smaller the mean width is also smaller compared to larger values of confidence intervals. This was talked through in the previous section and we can confirm it from the results above. Both of these

tables show that the **CCP** method, lowers the mean width compared to the **ICP**.

Remark that the number of samples for the New York dataset is 545 and for Boston Housing is 506. The mean widths of the **CCP** grow as k changes, this could mean that for these datasets, the number of samples is not large enough for large values of k. This could be observed from the New York Housing, the mean width increases after $k = 10$. Similarly, with Extended Boston Housing, the mean width shows an increase after $k = 3$.

Moreover, in Table 6.1 the error rate logic continues from the earlier section, and for a 90% confidence level, the error rate is 12.5% which is close to the 10% expected error rate value. This means that the prediction interval is too narrow and the proportion of observed values that fall inside the intervals is less than expected. We will try to mitigate this with **CCP**. Conversely, a 95% confidence level returns an error rate below 5%, this means that the prediction intervals can be made tighter and that they are too wide for this confidence level. Finally, for 99% the error rate is very close to 1% with 0.7% and the prediction interval seems to be accurate.

Furthermore, the **CCP** error rate values can be compared to **ICP** and it can be seen that for 90% confidence level, mitigates the overshooting of error rates, but this time it seems to undershoot the error rate threshold of 10% with a 6% error rate. Similar results can be followed for the 95% confidence level. For 99% confidence level, it looks like the error rate of 0.7 is stable across all the **CCP**s.

In Table 6.2 the error rate for **ICP** is 3% for a 90% confidence interval and for **CCP**s the error rate is between 4%-6%. This clearly shows that the prediction intervals are too wide and we can make them narrower and pull our intervals to a 10% error rate for a 90% confidence interval. Similar logic can be applied to a 95% confidence interval. For a 99% confidence interval, similar to the New York Housing dataset, the error rate is very stable and has an error rate of around 0.8%, making it an accurate prediction interval.

In conclusion, these results show how **CCP** narrows the prediction interval widths compared to **ICP**. The error rate results can be interpreted as the prediction intervals can be narrowed down for **CCP** results. This could be specific for these datasets, this can be observed in the following parts of this section. Furthermore, the stability across 99% confidence levels shows that the algorithm performs very conservatively under this confidence level, thus the the error rate is very close to 1% and stable across all of the **CCP**s.

Method/Measure	**Mean Width**			**Errors (%)**		
	90%	**95%**	**99%**	**90%**	**95%**	**99%**
ICP	32.342	37.503	47.605	0.0787	0.0078	0.0
CCP $k = 3$	13.629	20.066	35.856	0.0472	0.0236	0.0078
CCP $k = 5$	14.019	18.794	33.098	0.0314	0.0236	0.0078
CCP $k = 10$	13.838	19.227	34.435	0.0472	0.0236	0.0078
CCP $k = 20$	13.928	19.149	33.714	0.0472	0.02362	0.0078

Table 6.3: Ridge Regression CP with Boston Housing

Method/Measure	**Mean Width**			**Errors (%)**		
	90%	**95%**	**99%**	**90%**	**95%**	**99%**
ICP	81.895	82.974	84.583	0.0986	0.0536	0.0110
CCP $k = 3$	2.155	3.002	4.937	0.0990	0.0500	0.0103
CCP $k = 5$	2.151	3.007	4.951	0.0992	0.0498	0.0103
CCP $k = 10$	2.155	3.004	4.949	0.0990	0.0500	0.0103
CCP $k = 20$	2.157	3.013	4.942	0.0988	0.0496	0.0103

Table 6.4: Linear Regression CP with California Housing

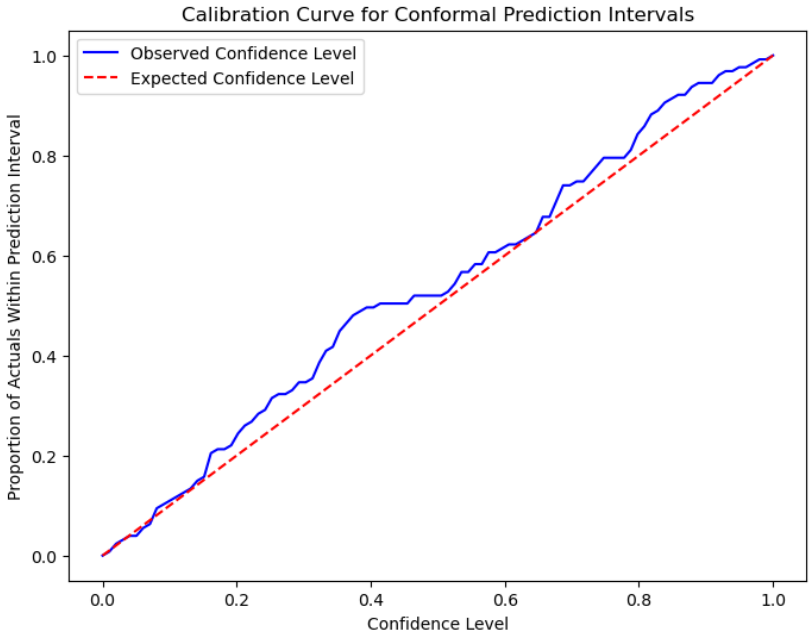

Figure 6.6: *Calibration Curve of the CCP for Boston Housing, with a Ridge($\lambda = 10$) Regression Model*

In the tables *Table 6.3* and *Table 6.4*, the mean interval widths are given with error rates for Inductive Conformal Prediction and Cross-Conformal Prediction with different values of k. The model that is used is Ridge ($\lambda = 10$) for Boston Housing and Linear Regression for California Housing datasets. Similar to tables 6.1 and 6.2, from both of the tables 6.3 and 6.4 it can be observed that, as the confidence level is smaller the mean width is also smaller compared to larger values of confidence intervals. Both of these tables show that the **CCP** method, lowers the mean width compared to the **ICP**.

Examining Table 6.3, it can be observed that for **ICP** the error rate of 90% confidence level is around 8%, which is considerably close to the 10% threshold, so we can say it has a good calibration. When a 95% confidence level is observed, the error rate is as small as 0.7% when the threshold is around 5%. This indicates our intervals are too wide and they can be narrower and the interval is not very well calibrated for this confidence level. The calibration graph for the Boston Housing dataset when **CCP** is applied with $k = 5$ is given in the *Figure 6.6*. The figure does show that this model is not calibrated well as the observed confidence level line (blue) is off the given $x = y$ line in dashed red. Furthermore, the error rate for a 99% confidence level is 0, this means that all the observed values have fallen into the intervals and there are no errors in the interval. This makes the confidence interval too conservative and implies that the prediction intervals are broader. Moreover, observing **CCP** values for Table 6.3 shows that the error rates drop from 8% to 5% for 90%, although the widths are narrower compared to **ICP**, the error rates seem to be conservative. 95% confidence level seems to be less conservative compared to the **ICP** with a 2% error rate. This shows that **CCP** narrows the intervals closer to the error rate threshold, 5% for this confidence level. Finally, 99% confidence level with **CCP** has better calibration compared to **ICP**, with an error rate of 0.8% which is close to the 1% threshold.

Therefore, when conformal prediction is applied to the Boston Housing dataset, the prediction intervals for **ICP** are well calibrated for 90% but not very well for 95% and 99% confidence levels. When **CCP** is applied mean width levels drop compared to **ICP** and the intervals are less conservative compared to **ICP**. Furthermore, the mean intervals are much smaller in the Extended Boston Housing dataset(Table 6.2) compared to the Boston Housing dataset(Table 6.3), this is because of the feature scaling and applying polynomial features generation in

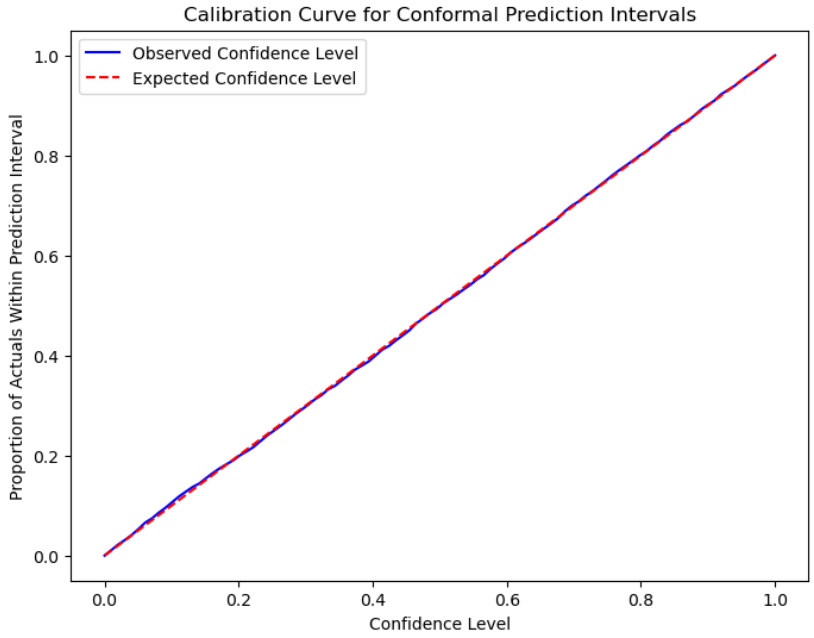

Figure 6.7: *Calibration Curve of the CCP for California Housing, with a Linear Regression Model*

the Extended Boston Housing dataset. This shows that if the data is processed well, the conformal prediction will also have narrower results. Finally, the **CCP** error rates show that the undershooting issue that is present in **ICP** for 95% and 99% confidence levels are somewhat mitigated. In conclusion, compared to **ICP**, this dataset can have better prediction intervals when **CCP** is applied.

Examine that the results from Table 6.4 are the standard results for a conformal prediction method. The **CCP** lowers the prediction interval widths compared to **ICP** and the error rates correspond to the significance level thresholds, 90% confidence level has 10% error rate, 95% has 5% and 99% has 1% error rates. This shows that this data is optimal for conformal prediction and cross-conformal predictions. For $k = 3$ **CCP**, throughout the given confidence levels we have the narrowest intervals and very well-calibrated error rates. This shows that these prediction intervals are as narrow as possible for given confidence levels. This can also be observed from the calibration graph *Figure 6.7*, of the California Housing dataset with **CCP** where $k = 5$ and the observed confidence level line (blue) is on the $x = y$ line (dashed red) which is the expected confidence level line, showing perfect calibration.

In conclusion, we can deduce that **CCP** successfully narrows the prediction interval widths compared to **ICP**. The Extended Boston Housing dataset performs better with **ICP** and **CCP** compared to the simple Boston Housing dataset. Depending on the datasets, the fluctuations of the error rates can be observed. For given confidence intervals the analysis showed that all of the data except for California Housing, presented an overshooting or undershooting of the error rates. The **CCP** results showed some mitigation concerning overshooting or undershooting of the error rates. Moreover, Ridge Regression with $\lambda = 1$ (for tables 6.1 and 6.2) and $\lambda = 10$ (for table 6.3) was used and Linear Regression is used for table 6.4, this shows that the model that is chosen for **CCP**, for it to have accurate and narrowed prediction intervals, the model should have high accuracy. This is due to the nonconformity scores which will be larger in general for low-accurate models. Therefore by applying the models that have been implemented throughout the project, the results show the difference between **ICP** and **CCP**.

Chapter 7: **Software Engineering Tools and Techniques**

The Software Engineering process of the project. The tools that are used, and the methodologies that are followed. The importance of the version control system, and how it is used. The coding standards that are applied in the project. The tools and the libraries that are used in the project. The testing and debugging procedures. Usage of documentation, and its importance.

7.1 Overview

In the development of this project, similar to other software projects, the integration of software engineering skills and methodologies is essential. The necessary methodologies and tools will create a strong and reliable algorithm. This chapter will cover the used tools and methods, going in-depth with their application and importance.

7.2 Software Development Methodologies

During the development of this project, the methodologies of Agile and Waterfall have been used in a hybrid approach.

7.2.1 Waterfall Methodology

The Waterfall methodology is a sequential approach to the development of the product. One stage of the development process should be finished before moving to the next one. The development stage begins with the necessary specifications of the project and moves forward with planning, modelling, construction and deployment. The waterfall methodology is the oldest software engineering methodology, thus it does come with efficiency problems. The fact that it is a sequential development, it is hard to go back to change or apply an update to the past stages. In the current complexity of software and its developments, the Waterfall methodology is not the most popular one amongst the other methodologies[24]. In this project, the waterfall approach is used in the development of the learning models, as the process is more linear and structured.

7.2.2 Agile Methodology

Unlike Waterfall, the Agile methodology is not static and has an iterative approach to software development. Agile development focuses on adaptivity. The product owner is always included in development and frequently gives necessary feedback for adaptive development. Thus this iterative approach is more compelling for complex and highly structured software and development[24]. The Agile methodology is more popular in software development compared to Waterfall due to its flexibility and adaptivity. In this project, the algorithms needed an iterative and continuous development which is a part of the Agile which allowed flexibility for any necessary updates. In conclusion, the hybrid approach has been practical for creating

strong and reliable algorithms and necessary development for the project.

7.3 Version Control System

Version Control Systems (VCS), have become an integral part of software development, which helps developers manage source codes and maintain the versions of the projects. It also makes collaboration between developers easier and more effective. There are two primary types of VCS: Centralized Version Control Systems(CVCS) and Distributed Version Control Systems(DVCS). The CVCS works with a centralised model with a single repository, meanwhile, DVCS, such as Git, do not have a central repository but have a local repository for each user[40]. This project uses the DVCS, Git and the system, GitLab as its VCS. In the project, Git is utilized for its full range of advantages, including efficient branching and version tracking. Branches are used to add or make any changes in the project, this way the main is always kept in a working condition. This is because if anything we change or add breaks the project, we can always go back to the working main. Tags are used to create releases. Releases help to track the development of the project and understand the developmental needs of the project.

7.4 Documentation and Coding Standards

Documentation and Coding Standards are very important aspects of a well-put software engineering project. Coding standards are a set of rules and guidelines for more consistent, readable, maintainable, and reliable software development. The coding standards unify the developers to write in a common way to make the understanding of the code easier [10].

Documentation also plays a crucial role in software development. A code without documentation is a map without a legend. It helps explain the code, its purpose, parameters, returns, and any exceptions or errors. This is very important, as if a new team member were to join the team they will have an easier time comprehending the already written code. Thus, good documentation and coding standards, will make the development more maintainable and comprehensible by others [10].

This project has followed certain coding standards and good documentation practices. All the code files and modules begin with a comment describing the module and its purpose, an example is provided in the code below. This is to enhance the maintainability and clarity of the project. Functions and methods are also documented with docstrings, explaining their purpose, parameters and returns. Moving forward, this project has followed the *PEP 8* Python coding convention. The PEP 8 is a guideline for Python developers to write clear and reliable code[25].

```
#model_evaluation.py is dedicated to evaluating the models.

import numpy as np

def r_squared(rss, tss):
    """
    Get the r-squared value for a linear model.

    :param rss: The residual sum of squares.
    :param tss: The total sum of squares.

    return: The R-squared value using the formula.
```

```
    """
    return 1 - (rss/tss)
```

This code is the first 15 lines of the `model_evaluation.py`.

It is easy to follow from the code snippet above that the placement of import statements follows the standard Python convention. The naming of the classes and the functions are also followed by the PEP 8 convention. The functions and classes also follow object-oriented programming principles. The variable naming has been very descriptive throughout the project for clarity and maintainability. In conclusion, following necessary coding standards and writing documentation, this project has been easy to maintain and clear to anyone who is an outside party.

7.5 Tools and Libraries

A variety of Software Engineering Tools and Libraries have been used throughout this project. `NumPy` library is the leading library as it does most of the mathematical calculations. It is used in all of the modules in this project. `NumPy` is essential for efficient mathematical operations, which is the core of this project. Moving forward, `Matplotlib` is another model that is primarily used in the module such as `plotting.py` and `app.py`. This library helps with creating visual graphs and plots which is very important for analysis of the results. The `Qt` framework libraries have been a big part of the `app.py` as they help to create and design the GUI of this project. Finally, `mglearn`, and `sklearn` libraries are used widely throughout the project for testing and accessing datasets. Therefore, the diverse use of software engineering tools and libraries has made the project highly efficient and functional. The combination of these tools and libraries has enabled adequate data handling, well-maintained analysis, and a GUI, making this project more variable and accessible.

7.6 Testing

Software testing is the most crucial part of software development as it validates the performance and quality of the software[3]. It is the only way to understand if a piece of work developed by developers has achieved its goal. Software testing has been thoroughly applied throughout this project. The testing has ensured the correctness and efficiency of the developed algorithms. As another important part of the testing, corner and extreme cases were tested and were maintained according to their outcomes. Moving forward, the implementation of unit testing, using the Test-Driven Development(TDD) approach, has helped me develop robust algorithms. A challenge encountered during the testing was determining the accuracy of the created models. To validate the outcomes of the models, `sklearn` library was used as a benchmark. The results and the scores were compared with the `sklearn`, to understand if there were any miscalculations or not. Finally, Jupyter notebooks were used to facilitate the analysis and drawing comparisons between the developed algorithms and the state-of-the-art `sklearn` library.

Chapter 8: **Professional Issues**

Machine Learning is a field that is based on data, and when data is the case many ethical issues arise. There have been many cases in the past where society was in shock at how monopoly companies like Facebook sell users' data to politicians for manipulation, etc. When using data, it is crucial to know where its source is, whether is it private, and traceable, does it comply with legal obligations. It is important to obligate laws and legal rights when gathering data to overcome any kind of ethical issues. As the value of data rises, companies try to find loopholes in the systems to access data and develop machine learning models, or systems that will benefit them. For everyone's privacy and their safety, it is very important to comply with laws and regulations regarding data. If not, in the case of data leakage, private data can end up with people who can pose a danger to others. That is why it is very important when dealing with data to follow regulations that protect everyone in society.

Moreover, how you gather, and what dependent variables are used in the data can be a factor to create societal bias. Before using the data, the data source should be read and understood for what kind of purpose the data was collected. This could be understanding what dependent variables are used for what reason. If the data consists of dependent variables, that discriminate a group of people or are offensive to some people, it may be unethical to use this data to conclude the independent variable. When doing research or, building a machine learning model, researchers and developers use many data. It is up to them to decide to use data for their research or development, thus it is their utmost duty to conclude if something about the data is unethical. If this is not thought through thoroughly it could result in unethical results. Thus, in this project, dependent variables were analysed before the use of data to prevent any kind of problems for the end product.

Throughout this project, several datasets were used and applied to the learning models. As mentioned before, data brings an ethical issue by itself to this project. It is crucial that the data is used in legal ways and it is not anyone's data or can not be traced back to the individual. Having said this, the housing datasets that were used in this project were gathered from educational sources, and they can not be traced back to an individual. The California and Boston Housing datasets are gathered from the `sklearn` and `mglearn` libraries, which are educational libraries for machine learning practices.New York and Washington datasets were found in kaggle and did not present any ethical issues. Originally, the Washington House dataset had some private information, such as the address and the postcode of the houses but they were removed and not used in this project and prevented any ethical issues that could arise.

Another ethical issue that arises in this project is the Boston Housing dataset. The authors of this dataset engineered a non-invertible variable "B" assuming that racial self-segregation had a positive impact on house prices[26]. Thus, the dataset has ethical issues when an analysis based on features is made. Although the Boston Housing dataset was widely used in this project it was taken into careful consideration to not include any kind of feature-based analysis. Furthermore, this project's primary goal is to compare the regression algorithms and predict house prices. When predicting the house prices Boston Housing datasets were not taken into consideration. When an analysis was made with these datasets, it was focused on the theory of the datasets. For example, the Extended Boston dataset was an example of well-preprocessed data compared to Boston Housing or California Housing has the most samples compared to any of the datasets.

Moreover, using the GUI you can make predictions based on given states or you can either select a state or import data and analyse the model. For prediction, simply, the values of the features can be entered and a price prediction will be returned. Although you can make

model analysis using the Boston Housing datasets, it is not possible to make house price predictions using the Boston Housing datasets, -Extended or Normal- overcoming any kind of ethical concerns. In conclusion, the data can have societal biases given in the Boston Housing dataset[8]. Data should be carefully analysed and processed before working on it to prevent any kind of societal biases and ethical issues. Throughout this project, this was carefully taken into consideration and acted upon respectively, by removing some of the private information that could indicate problems for the Washington dataset and preventing any kind of prediction from being made using the Boston Housing dataset.

Bibliography

[1] Data normalization google developers, 2022-07-18. https://developers.google.com/machine-learning/data-prep/transform/normalization.

[2] Numpy library. https://numpy.org/devdocs/reference/generated/numpy.linalg.inv.html.

[3] S. Ahamed. Studying the feasibility and importance of software testing: An analysis. *arXiv preprint arXiv:1001.4193*, 2010.

[4] E. Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[5] J. C. A. Barata and M. S. Hussein. The moore–penrose pseudoinverse: A tutorial review of the theory. *Brazilian Journal of Physics*, 42:146–165, 2012.

[6] N. H. Bingham and J. M. Fry. *Regression: Linear models in statistics*. Springer Science & Business Media, 2010.

[7] J. Brownlee. How to implement simple linear regression from scratch with python. https://machinelearningmastery.com/implement-simple-linear-regression-scratch-python/, May 2020.

[8] M. Carlisle. Racist data destruction? https://medium.com/@docintangible/racist-data-destruction-113e3eff54a8, 2019. An article on Boston Housing Dataset.

[9] M. Cilimkovic. Neural networks and back propagation algorithm. *Institute of Technology Blanchardstown, Blanchardstown Road North Dublin*, 15(1), 2015.

[10] V. Dixit. Coding standards and guidelines: A comprehensive guide with examples and best practices. https://www.lambdatest.com/learning-hub/coding-standards, Sep 2023.

[11] A. Gammerman, V. Vovk, and H. Papadopoulos. *Statistical learning and data sciences*. Springer, 2015.

[12] M. Gruber. *Improving efficiency by shrinkage: The James–Stein and Ridge regression estimators*. Routledge, 2017.

[13] A. E. Hoerl and R. W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[14] A. Horel. Application of ridge analysis to regression problems. *Chemical Engineering Progress*, 58:54–59, 1962.

[15] D. Jackson. The lasso. =https://bookdown.org/ssjackson300/Machine-Learning-Lecture-Notes/the-lasso.html, Mar 2023.

[16] G. James, D. Witten, T. Hastie, R. Tibshirani, et al. *An introduction to statistical learning*, volume 112. Springer, 2013.

[17] S. A. Kakvi. Computer and network security notes, 2022.

[18] A. A. Koloydenko. Multiple linear regression lecture notes. https://moodle2223.royalholloway.ac.uk/pluginfile.php/1105406/mod_label/intro/MultRegressionV2023.pdf, 2023.

[19] Y. Meng. Least squares data fitting. =https://courses.grainger.illinois.edu/cs357/fa2021/notes/ref-17-least-squares.html.

[20] A. C. Müller and S. Guido. *Introduction to machine learning with Python: a guide for data scientists.* " O'Reilly Media, Inc.", 2016.

[21] K. Murphy. *Machine learning: a probabilistic approach*, volume 1163. MIT Press, 2013.

[22] numpy. Numpy.sign. `https://numpy.org/doc/stable/reference/generated/numpy.sign.html`.

[23] NumPy Developers. numpy.quantile — numpy v2.0.dev0 manual. `https://numpy.org/devdocs/reference/generated/numpy.quantile.html`, 2024. Accessed: 2024-03-19.

[24] R. S. Pressman. *Software engineering: a practitioner's approach.* Palgrave macmillan, 2005.

[25] G. v. Rossum. Pep 8 – style guide for python code. =https://peps.python.org/pep-0008/.

[26] Scikit-Learn. Boston housing dataset. `https://scikit-learn.org/1.0/modules/generated/sklearn.datasets.load_boston.html`, 2024. Boston Housing Documentation.

[27] G. Shafer and V. Vovk. A tutorial on conformal prediction. *Journal of Machine Learning Research*, 9(3), 2008.

[28] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms.* Cambridge university press, 2014.

[29] S. Theodoridis. *Machine learning: a Bayesian and optimization perspective.* Academic press, 2015.

[30] S. H. To. Quantile: Definition and how to find them in easy steps. `https://www.statisticshowto.com/quantile-definition-find-easy-steps/`, 2024. Accessed: 2024-03-19.

[31] T. University. Boston dataset from toronto university. `https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html`.

[32] V. Vovk. Introduction to conformal prediction lecture notes. `https://moodle.royalholloway.ac.uk/pluginfile.php/191090/mod_resource/content/38/03_1.pdf`. Lecture notes from Machine Learning module.

[33] V. Vovk. Chapter 6: Data preprocessing, parameter selection, and inductive conformal prediction. `https://moodle.royalholloway.ac.uk/pluginfile.php/195264/mod_resource/content/32/06_1.pdf`, 2023. Lecture notes from Machine Learning module.

[34] V. Vovk. Chapter 9: Pipelines and conformal prediction. `https://moodle.royalholloway.ac.uk/pluginfile.php/202273/mod_resource/content/24/09_1.pdf`, 2023. Lecture notes from Machine Learning module.

[35] V. Vovk and A. Gammerman. Conformal prediction. `https://cml.rhul.ac.uk/cp.html#:~:text=Abstract-,Conformal%20prediction%20uses%20past%20experience%20to%20determine%20precise%20levels%20of,y%20with%20probability%201%20%20`μ.

[36] J. Watt, R. Borhani, and A. K. Katsaggelos. *Machine learning refined: Foundations, algorithms, and applications.* Cambridge University Press, 2020.

[37] S. Weisberg. *Applied linear regression*, volume 528. John Wiley & Sons, 2005.

[38] X. Wu, R. Ward, and L. Bottou. Wngrad: Learn the learning rate in gradient descent. *arXiv preprint arXiv:1803.02865*, 2018.

[39] Zach. How to calculate r-squared by hand. `https://www.statology.org/calculate-r-squared-by-hand/`, July 2021.

[40] N. N. Zolkifli, A. Ngah, and A. Deraman. Version control system: A review. *Procedia Computer Science*, 135:408–415, 2018.

Appendix

Linear Regression Proof of Concept Programming

For many years Scientists and Statisticians have applied Linear Regression to any kind of data. As the usage of computers grew, it became easier to make statistical calculations. In order to apply this to Machine Learning, we have to create an algorithm, where the algorithm takes a *training set* and a *test set*. The algorithm will use the training set to determine the coefficients of the best fit line, and the test set to make the predictions[7]. To make predictions using Machine Learning, we need data. The data we use, are usually divided in half. *Training Data* and *Test Data*, Training data is used to build a model, so that the model can make future predictions. Meanwhile test data is used to test the model by making predictions and create some benchmarks regarding the model. Looking at our benchmarks we can analyse, if the model we have is a good fit to make predictions, which will be discussed in the section **2.4**. Implementation of a Linear Regresssion algorithm for learning is as follows. Using TDD we can create a simple function to estimate our least estimate square coefficients.

```
def sum_x_y(data):
  total = sum(data[:, 0]*data[:, 1])
  return total

def intercept(data):
  n = data.shape[0]
  a = ((sum_y(data))-slope(data)*sum_x(data))/n
  return round(a, 2)

def slope(data):
  n = data.shape[0]
  b = (n*(sum_x_y(data))-(sum_x(data))*(sum_y(data)))/
      ((n*(sumsq_x(data)))-(sum_x(data))**2)
  return round(b, 2)
```

The two functions `intecept()` and `slope()` calculates the coefficients of our best fit line using the least squares estimate method. The functions returns a number rounded to two decimal places. To determine the n we use the `shape()` function. The `shape()` function returns a tuple and the first element is the number of rows we have in the data which stands for our n. The `sum_x_y()` and all the other `sum()` functions are basic sum functions of the values of data.

```
def slr(train, test):
  predictions = np.array([])
  intercept_ = intercept(train)
  slope_ = slope(train)
  # Train, test and asses the model for datasets.
  yhat = intercept_ + slope_ * train[:, 0]
  train_predictions = np.append(predictions, yhat)
  rss_train, tss_train = rss_tss(train, train_predictions)
  yhat = intercept_ + slope_ * test[:, 0]
  test_predictions = np.append(predictions, yhat)
  rss_test, tss_test = rss_tss(test, test_predictions)
  r_sq_train = r_squared(rss_train, tss_train)
  r_sq_test = r_squared(rss_test, tss_test)
```

```
Installing missing packages...
Requirement already satisfied: PySide6_Addons==6.6.3 in /home/cim/ug/zkac149/Desktop/PROJECT/venv/lib/pyt
hon3.10/site-packages (6.6.3)
Requirement already satisfied: PySide6_Essentials==6.6.3 in /home/cim/ug/zkac149/Desktop/PROJECT/venv/lib
/python3.10/site-packages (6.6.3)
Requirement already satisfied: shiboken6==6.6.3 in /home/cim/ug/zkac149/Desktop/PROJECT/venv/lib/python3.
10/site-packages (from PySide6_Addons==6.6.3) (6.6.3)
All packages are installed.
```

Figure 8.1: *The command line when bootstrap is finished*

Figure 8.2: *The further analysis resulsts page for New York Dataset*

```
results = {
    "Predictions": predictions,
    "Training R-Squared": r_sq_train,
    "Test R-Squared": r_sq_test
}
return results
```

The `slr()` function is the *Simple Linear Regression* learning algorithm. It takes in two parameters, one the training set and the other is the test set. It uses the `slope()` and `intercept()` functions to create a model using the training dataset. Then the algorithm moves to a stage where it asseses the model. Using the *intercept* and *slope* values, we get new sets of predictions. In order to assess (*Check Section 2.4*) the results from training set and test set, both training set and test set predictions are calculated separately. By using the `rss_tss()` function, we calculate the RSS and TSS values, to calculate the R^2 statistic, where R^2 statistic are the scores for the model.

User Manual

The GUI of this project has two main functionalites. One of them is to predict and give prediction intervals for house prices from the given datasets, -it can not be imported-. The other functionality is to do an in-depth analysis of the models from chosen or imported dataset. Further analysis creates all the models using the dataset that was chosen and at the end it prompts a window to be able to do a model analysis. The analysis depends on the model's assesment scores, R^2, MSE and variance. The analysis also consists of an coefficent comparison to understand the difference between L1 and L2 regularizations.

The GUI, of this project can be ran by running the `app.py` module in the source file. If your

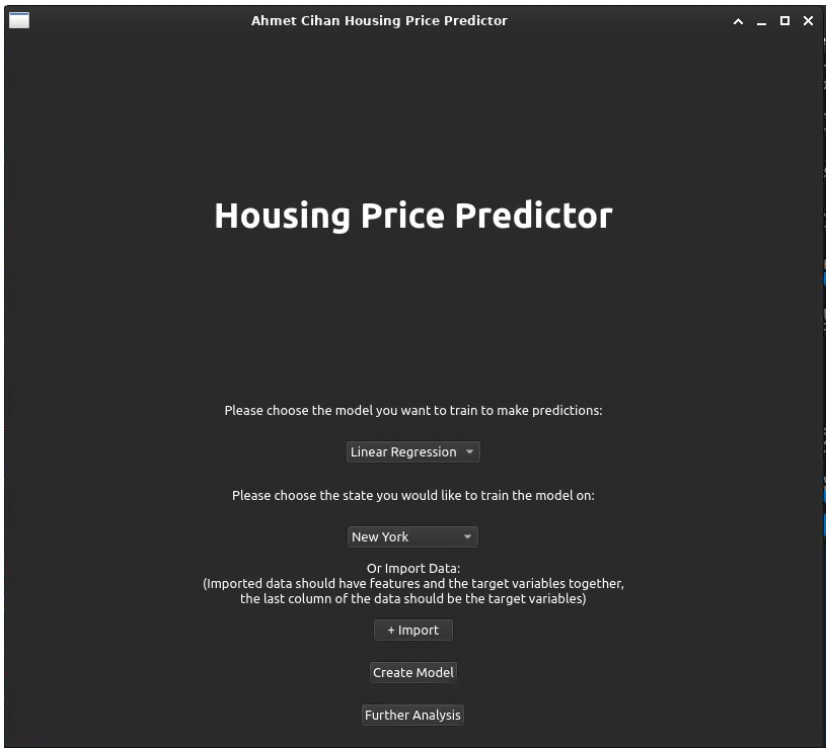

Figure 8.3: *The mainwindow when the app is run*

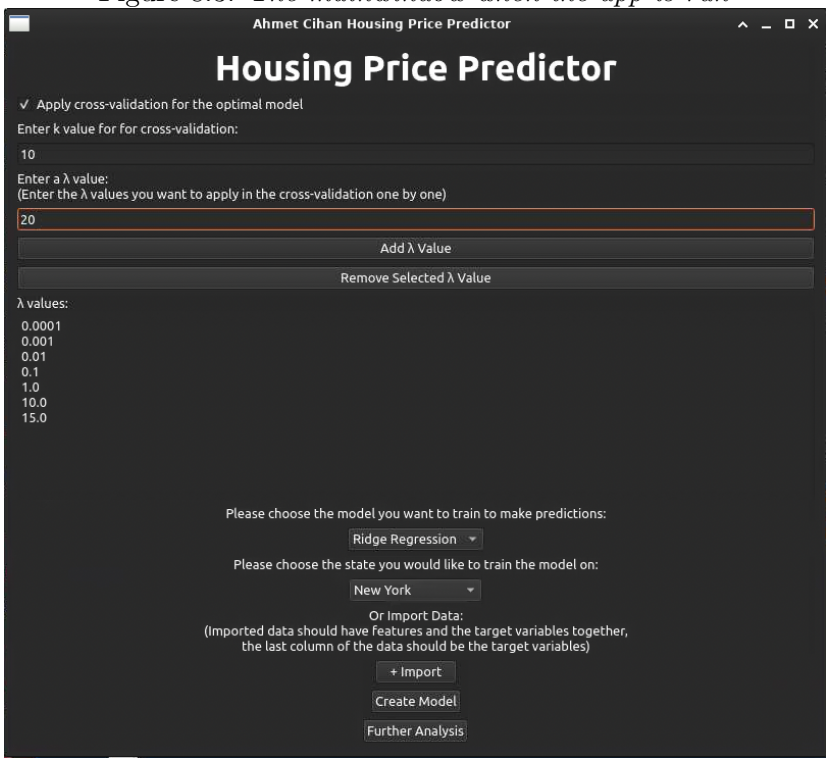

Figure 8.4: *The cross-validaiton for Ridge Models*

environment has pip and python installed, `app.py` should run without an issue. The `app.py` has a bootstrap script to install any missing or required libraries to run the modules of this project. If you are running the app for the first time, the app may take a while to install all the dependencies and requirements so it is normal for it to take up to 5 minutes to start the app for the first time. If for any reason, the app quits itself after bootstrap, please do try again to run the `app.py` as it sometimes runs the program in the second time. As shown from *Figure 8.1*, the command line can be followed.

When the program successfully downloads all the libraries and run, a mainwindow appears given in *Figure 8.3*. In this window you can either choose a dataset to train your model and make predictions with by **first**, pressing create model and **then** make predictions button. If Ridge or Lasso models are chosen for predictions, cross-validation can be applied (*Figure 8.4*). If cross-validation is chosen, then the user has to choose a k value and add the λ values to apply the cross-validation. In case of Lasso model, all the parameteres should be entered for cross-validation and normal model creation.

Regarding the further analysis button, user can use this button to analyse any **imported** data or chosen data. If you want to **analyse imported** data, you should import the data, then press further analysis. Note that for accurate analysis the data should have the target variable as the last column of the imported data. Furthermore, pressing further analysis button will create a small dialog saying Linear Regression is created and create Ridge model for next step, similar to prediction of the prices, cross-validation can be applied or a simple λ value can be given to process to next step. User has to **press create model** before pressing next. After that Lasso model can be created using a similar logic. Again after **pressing create model** and next, a results page (*Figure 8.2*) will appear regarding your data you have chosen or imported.

If at any point at prediction or further analysis if you want to cancel and go back to the main page you can press the escape button or click on the x to close the dialogs.

Reflection

The dissertation has been a significant learning journey. Implementing Linear regression and Ridge was an easy task because of my background from statistics modules last year. Reading the papers and books regarding these topics was understandable and easy to follow. When the analysis was made the results were the same as the `sklearn` library, it showed accuracy. Deriving the ridge regression formula was a tough task to do, readings were done to understand the theoretical approach behind Ridge.

It was important to understand the difference between the inverted matrices approach and gradient descent. From statistics, the inverted matrices approach was a familiar approach but gradient descent was a method I learned during the development of this project. It took me the longest time of all the algorithms to build a functional gradient descent algorithm. This was due to several factors, it was a different concept and it took me time to understand how it can be approached. First, I tried and tested the algorithm against the coefficient variables and it looked like gradient descent sometimes does not find the same coefficients compared to the inverted matrices approach. This pushed me back a bit because I was testing and comparing the wrong part of the methods, after figuring this out I moved to test the accuracy of the results with a R^2 score. After comparing the results, I was able to deduce that my gradient descent method was accurate and working fine. Another challenge with the gradient descent algorithm was that it had two more hyperparameters compared to the closed-form approach of the Ridge Regression algorithm. These parameters were key

to faster results. In the beginning, it took me time to understand and sometimes I ended up running my gradient descent algorithm for hours. After understanding and reading more about the hyperparameters it was easier to understand how the runtime of the algorithm can be shortened.

Moreover, applying gradient descent to lasso regression was not a simple task, I had to understand how I could approach to L1 penalty. Since it was the absolute value of the coefficients, it was not a straightforward task to add the penalty term to gradient descent. I used the numpy library's sign function to understand which direction the penalty is going to. Moving forward another critical part of this project is understanding how cross-validation works and using it to choose the penalty coefficient. A function was created to divide the data into k parts for training and testing. This function was then used to determine the optimal penalty coefficient This was a straightforward task compared to the gradient descent algorithm, as it was an introduction to cross-validation for me. With gradient descent, another important topic that arose was data normalization. Before gradient descent and lasso regression, data normalization was not applied to train and test the data, after gradient descent was introduced it showed that normalized data with a variance of 1 was able to reduce the runtime of the gradient descent algorithm. The results I have obtained from section 5.5 showed clear results between the difference between Linear Ridge and Lasso Regressions, the results were as expected. Moving forward, it was valuable for me to write and learn more about conformal prediction which I learned in my Machine Learning module first term. It was a good revision and learning and reading more about conformal prediction made me more interested in the topic. Applying cross-validation to regression was a challenging task but the motivation of Vladimir Vovk saying it can be applied to any machine learning method, made me push till the end. I took a different approach to my conformal prediction algorithm, I did not use p-values to estimate the intervals instead I used quantiles to deduce what the bounds are for the intervals. Furthermore, the implementation of cross-conformal prediction was less challenging after implementing quantiles for interval prediction. However, I realised that I need a different approach for Lasso's cross-conformal prediction as it distributes predictions differently compared to Ridge Regression.

Furthermore, the GUI itself was another challenge of this project. Using a framework, Qt, which I didn't know before, required me to do documentation readings, to understand which functions are more useful in what cases. Since the project was more of a comparative, research topic, I couldn't focus solely on the GUI thus, I did it as simply as possible, which led me to have code smells and not the best documentation for my `app. py`. Despite this, I managed to get a functional and working application that can use the models to make house price predictions and do further analysis about the models by comparing their coefficients and assessment scores such as MSE and R^2.

Finally, this project was a success, comparative analysis was done based on Ridge Regression, a functional GUI was implemented to do predictions and analyse the models. It was a fulfilling journey, theoretically and practically. The project has given me great experience in the field I want to work in the future. Thus, I am willing to do more in-depth research about machine learning models for future use.

Diary

Term 1

Week 1 and Week 2:

- Done research about the project.

- Found references regarding Linear Regression, Ridge Regression, K-NN, and Neural Network Algorithms.

- Wrote the abstract.

- Wrote the Plan for the project to submit.

Week 3:

- Research about Mathematics of Linear Regression.

- Template for the report is arranged.

- Wrote an introductory paragraph about Linear Regression and gave an example.

- Introduced the Least Squares Estimate method.

- Wrote a paragraph about Least Squares Estimate and continued with an example.

- Started the programming of Linear Regression algorithm.

- Created a new branch for the development of the Linear Regression Algorithm.

- Least Squares Estimate functions are implemented.

- Simple Linear Regression Algorithm is implemented.

- Report is written about Training and Test Data sets.

Week 4:

- Report about the code written for linear regression algorithm.

- Report R-Squared and RSE for assessing the model.

- Code the R-Squared and apply it in previous dataset.

- Wrote a small analysis of the example data

- Multiple Linear Regression introduction.

- Programming Multiple Linear Regression.

- Testing the MLR algorithm with various datasets.

Week 5:

- Comparison of the Multiple Linear Regression algorithm between the sklearn library.

- Report about the results from the MLR.

Week 6:

- Ridge Regression reading.

- Finishing of the comparison of MLR and SLR in the report.

Week 7:

- Ridge Regression implementation.(Proof of concept programming)

- Ridge Regression report.

- Ridge Regression introduction.

- Variance estimation added to the linear_r module.

Week 8:

- Ridge vs Linear Regression Report.

- Ridge vs LR Coef Analysis Report.

- Variance calculation and analysis.

- R-Squared comparison between models and variance.

- Testing the Algorithm part of the report. The tests were made by comparing the Project's algorithm's to the ones of Scikit Learn.

Week 9:

- Start to building the GUI.

- I have decided to use PySide6 for my GUI application.

- Found some tutorials, how to use PySide6 and apply it to the project.

- Had some problems adding widgets to the layout. Learned that PySide6 works with widgets that are initiated when `QVBoxLayout()` is called.

- I have decided that it is smarter to do the data, algorithm choosing from the menu bar. Thus I have added these two options to the menu bar.

- I was using the linear_r.py module for everything. For more professionalised approach of software development, I have refactored my code into smaller modules and classes.

- I have been working on the GUI, and I have implemented a way to output the linear regression vs ridge regression coefficients bar graph. This at first did not work as expected. As I struggled with not having the output in the same window. In order to fix this I used a buffer to save the graph and output is a png file.

- Worked on a slider to dedicate the lambda, tuning parameter for Ridge Regression. Used a new layout to add the slider. I will need more layouts for future cases to display the results of the algorithms.

- Finished a small text box for ridge regression tuning parameter. Now it can be manually inputted as well as using the slider.

- Worked on literature review.

- Working on k-fold-cross-validation, testing for the perfect algorithm. Ran into dimensions not matching issue, I think there is something wrong with how I assigned the division of the data. I fixed that issue, as it was a small bug where I had a typo. I compared my results to sklearn, but I have realised the answers were not the same. This was due to the fact that I do integer division when deciding the `fold_size`. `sklearn` library does better managing this. When the number of features is folded into a number that can be divided, then the values we end having are very similar.

Week 10:

- Datasets section created for the introduction. Wrote a concise summary of the datasets that are used in the project.

- Writing the software tools and techniques section of the report. Doing readings from the Software Engineering by Roger S. Pressman.

- Wrote the Methodologies part of the report.

- Doing research for Version Control System section of the Software Engineering Tools and Techniques.

- Documentation and Coding Standards section of the report.

- The documentation and coding standards section is done. On top of this finished the Tools and Libraries section.

- Testing section started.

- Worked on the presentation.

Week 11:

- Research on Gradient Descent method. Getting ready for the presentation. Making a handout and practicing the presentation.

Term 2

Weeks 1 - 2:

- Planning the structure of the term 2.

- Meeting with the supervisor to finalise the plan.

Week 3:

- Readings about the Gradient Descent Method from different sources.

- Research about the cost function, understanding mathematical background and foundation of gradient descent method.

- Report, loss function is introduced.

- Pseudocoding the gradient descent method.

- Tried a direct application of the taking partial derivatives of the cost function, ended up being too 'costly'.

- Derived another way of partial derivation of the cost function using matrix algebra.

- Proof of concept programming of the gradient descent using the recent matrix algebra method.

- Struggling with the graident descent algorithm, need to do more research on how to determine the learning rate and threshold to determine the minimum for the cost function.

- Gradient descent is introduced in the report.

Week 4:

- Report of the Gradient Descent.

- Mathematical simplification is introduced to ease the coding of the algorithm in the report.

- Testing the algorithm on boston housing dataset. Standard gradient descent does not seem to perform well in datasets as big as boston housing dataset.

- Looking for other gradient descent methods to imporve the algorithm.

- Realized that the data itself is the problem. When there is an artifical dataset, with proper scaling the algorithm works very well.

- For a more effiecient way of computing large datasets, Stochastic Gradient Descent will be worked on.

- Stochastic gradient descent is implemented for future use.

- Report of the gradient descent implementation.

- Learning rate and Convergence Threshold report is written.

- Research about the lasso regression.

- Implemented and tested the lasso regression with the gradient descent method.

Week 5:

- Lasso Regression Report

Week 6:

- Data Normalization research, and a report of standard normalization with its mathematical approach.

- Computational Complexity research, learned more about gradient descents computational complexity. There is not much articles related to this, I ended using my own initiative related to this.

- Wrote the Matrix inversion complexity, gradient descent complexity and comparison sections in the gradient descent chapter.

- Finished the computational complexity part of the Gradient Descent Method chapter.

- Research about testing of the lasso algorithm.

Week 7:

- Tests are done on the lasso regression algorithm.

- Coefficient analysis has been made, but struggling to get proper results when lasso scalar increases.

- Wrote a proof that the least squares estimate ridge formula can be written in the previously provided ridge estimator with linear algebra format.

- Application of Lasso Regression is written in the report, with the explanation of the code.

Week 8:

- Tuning parameter section is written in the report.

- Proof of concept programming is implemented and tested for finding the tuning parameter both for ridge and lasso regression.

- Cross-validation is modified for assesing the models with normalized data.

- Further analysis is made using the Boston Housing Dataset.

- Introduction to conformal prediction, research is done.

Week 9:

- Readings about conformal prediction.

- Written conformal prediction algorithm.

- Proof of concept programming of cross conformal prediction.

- Report of the conformal prediction.

- Bug fixes on the conformal prediction and cross conformal prediction.

- Issues with prediction intervals being not correct.

Week 10:

- Bug fixes on conformal prediction and cross conformal prediction continues.

- Proof of concept programming of the error rate and calibration curve to test and understand the prediction intervals.

- More efficient progress is made as the error rate and calibration curves were good way to understand if the intrevals were correct.

- Interval prediction algorithm is modified and now uses calibration sets to calculate nonconformity scores. This has given me more credible prediction intervals compared to before. Before I was getting the nonconformity scores from the whole training set and this was giving me noncredible intervals with high error rates for high confidence intervals such 95% it would give 52% error rate.

- Similar logic has been implemented to cross conformal prediction and I now have better prediction intervals on given confidence levels.

Week 11:

- Decided to use quantiles for prediction intervals.

- After applying quantiles to nonconformity scores the results were more credible and accurate compared to the error rates.

- Report on Cross-conformal prediction about interval widths.

- GUI work, created several releases. It followed a step by step approach, first the main layout was built to create models, then a further analysis dialog was created for analysing the models. After, a prediction dialog was created for house price prediction chosen the dataset.

Week 12:

- Worked on results, decided to write more comprehensive results section for Chapter 3,

Chapter 5 and Chapter 6.

- Linear Regression and Ridge Regression results were made by applying the models to all of the datasets.

- Linear, Ridge and Lasso were analysed using all the datasets, cross-validation was also added to the results for more detailed analysis.

- Inductive Conformal Prediction was tested against cross-conformal prediction. An analysis is made comparing the interval widths across all the datasets.

- Professional Issues section is written considering the data usage in this project.

- Reflection is added for reflecting to the whole of the project.