Kıvanç Değirmenci 21803403
Mehmet Batuhan Candar 21801605
08.05.2022

EEE 485 FIRST REPORT OF THE PROJECT

Introduction

The aim of this project is to implement a machine learning algorithm which operates in the diamond dataset which we determined earlier. Diamonds are precious and valuable stones which are used as an investment. There are various factors that affect their prices. Therefore, it is important to decide its price and it is quite a difficult job for both retailer and the buyer. This project tried to solve the future contradictions about the diamonds' price by estimating a price with the features are entered as an input.

To be able to implement such a system, three machine learning algorithms are used. They are Ridge Regression, Neural Networks and Decision Tree. The decision of the usage of these algorithms are taken according to the research conducted before proposal. Besides, as mentioned in the proposal, we continued to do research whether there are other algorithms that we did not know before with more efficient usage to our purpose. However, we did not change the algorithms proposed. Until the week of this First Report, the Ridge Regression and Neural Network algorithms are implemented successfully. In the following weeks until the Final Report and Demonstration, the Decision Tree algorithm is implemented. The results and detailed explanations are in the proper section of this report.

The project consists of four main parts. The first one is determining the dataset. The second is preprocessing the dataset which is chosen in the first step. Then machine learning algorithms which are mentioned are implemented. Lastly, the performance results coming from our models are evaluated.

The Dataset

The dataset that we are using includes essential features of a diamond [1]. After searching in detail of the important factors that affect the value of a diamond and checking various datasets about this issue, the dataset is determined with its comprehensiveness and quality with respect to others. The dataset we chose has 10 columns (9 are the independent features and 1 is the price). The columns are listed below:

- Carat : Carat value
- Cut : Cut type
- Color : Color value
- Clarity : Carat type
- Depth : Depth value
- Table : Flat facet on the surface
- X : Width
- Y : Length
- Z : Height
- Price : Price value

Three of the features (Carat, Cut, Color) have categorical values while other features have numerical values. Since the dataset has categorical values for a regression problem and the numerical values' ranges are not the same, a pre-processing period is required. Besides, this pre-processing is important due to possible missing values or outliers in the dataset. Then, this dataset is used for training, validation and testing since there is not separate datasets for these purposes by dividing original dataset with shuffling with appropriate ratios.

Methodology

1. Pre-Processing

Instead of implementing the algorithms directly, the dataset has to be adjusted and checked in order to get a good learning rate and also higher accuracy. With this way, reliability of the dataset is increased. Since the dataset is found online, there may be missing values or irrational data in it. These malfunctioned parts worsen the efficiency of our algorithms. Thus, all rows and columns of

the dataset is controlled in order to eliminate the missing values. However, dataset does not have any missing values in any row or column.

After this check, the features are arranged whether they affect the price of the diamond or not. If there were any feature which haven't affected the price, it could have been removed. However, all 9 features which is in the original dataset has an effect of the price. This is determined by checking plots of each feature with price and detailed online search. The distribution graphs of these nine features are provided in the Appendix.

Besides, possible outliers were investigated but, any outlier data are determined to remove. Before normalizing numerical valued features, the categorical valued data should be transformed into numerical valued. For this purpose, several online sources are checked for determining in which order these values represent quality of a diamond. According to these results, the range of [0-1] is divided into equal parts for simplicity. There are other possibilities but, according to our prior knowledge, this separation might be enough to understand the correlation with price. For the numerical valued features, all data are mapped into the range between 0 and 1. For this mapping, maximum and minimum values of each feature is determined and the following equation is followed;

$$X_{normalized} = \frac{X - X_{minimum}}{X_{maximum} - X_{minimum}}$$

With normalization step, the consistency is provided between different features. The pre-processed dataset is created and saved to use during the implementation of algorithms. The pearson correlation matrix for 6 features is plotted and can be seen in Figure 1. It is seen in the table that some features have high correlation rates with each other while other do not. The reason of this high correlation is interpreted as: Since a diamond need to be attractive visually, its dimensions should have similar values and therefore; x, y and z values are highly correlated with each other.

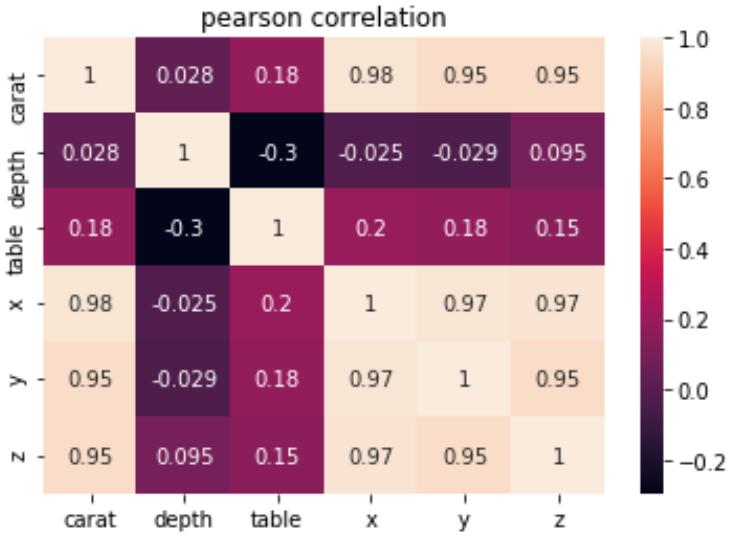

Fig. 1: Correlation matrix between features

2. Neural Network

The second machine learning algorithm which we used in our project is neural networks. It can be used both for regression and classification type of predictions mentions that neural networks are just non-linear statistical models like projection pursuit regressions. [2] Its name is neural network, because its connections are similar with the structure of the neurons in human body as it can be seen

in the figure below. It consists of perceptron which are the unit piece of the network. There are layers between the inputs and the outputs which are called hidden layers. The inputs and the output neurons are connected to each other via hidden layers. The number of the hidden layers are determined according to the dataset and they allow the network to solve complex problems. As the number of hidden layer increases, the neural network become more complex. If there is no hidden layer, the model will be similar with the linear regression model.

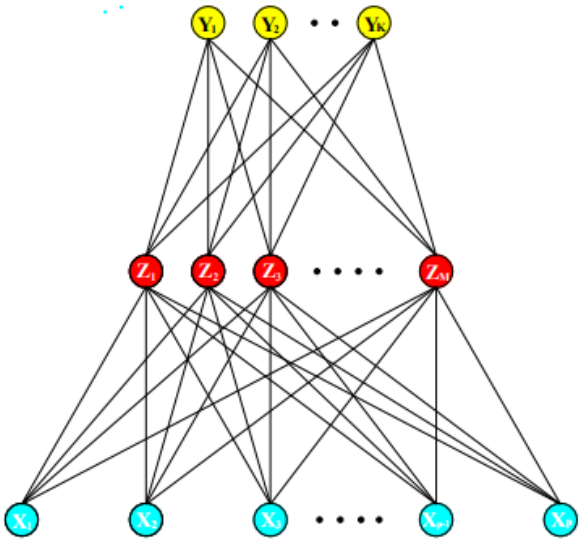

Fig. 2: Schematic of a feed-forward neural network [2]

The output nodes are obtained as the input nodes are multiplied with the weights and added biases. Then the nodes in the hidden layer are added and get into the activation function to obtain the output. Some type of activation functions which are introduced in the lecture are linear, logistic, hyperbolic tangent, soft plus, rectified linear unit (relu).

Since the dataset which we used has 9 features, there are 9 input neurons which are carat, cut, color, clarity, depth, table, x, y, z lengths in our model. It has one hidden layer and one activation function. Relu was selected as an activation function. Lastly, the model has one output neuron which is the predicted price of the diamond. The procedure is as follows;

Firstly, we initialized the weights and biases with random values. Their values were set according to the Gaussian distribution with zero mean and variance of 0.3. Then the initialized weights are multiplied with the corresponding input neuron and added the biases. The results of this process added together. Relu activation function is applied to the results in order to compute output. This process is the forward-propagation. Then the loss function is designed in order to observe the error of the model. Lastly, back propagation is applied. Back propagation is a special way to implement gradient descent in a neural network algorithm. By applying the back propagation process for several epochs, the weights and the biases updated to their optimum values. The mathematical expressions of this process are as follow:

- Gaussian distribution ($\sigma^2 = 0.3, \mu = 0$):

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- Forward Propagation:

$$hidden\_layer = x \cdot weight_{hidden} + bias_{hidden}$$
$$\widehat{hidden\_layer} = relu(hidden\_layer) = \max(0, hidden\_layer)$$

$$v_j = x \cdot weight_{output} + bias_{output}$$
$$\hat{y} = relu(\widehat{hidden\_layer} \cdot weight_{output} + bias_{output})$$

$$error = (y - \hat{y})$$
$$MSE = (y - \hat{y})^2$$

- Back Propagation:

$$\frac{\partial \delta(n)}{\partial w_o} = \frac{\partial \delta}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial v_j} \times \frac{\partial v_j}{\partial w_o}$$
$$= -2 \times error \times relu'(v_i) \times (\widehat{hidden\_layer})^T$$

$$\frac{\partial \delta(n)}{\partial b_o} = \frac{\partial \delta}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial v_j} \times \frac{\partial v_j}{\partial b_o}$$
$$= -2 \times error \times relu'(v_i)$$

$$\frac{\partial \delta(n)}{\partial w_h} = \frac{\partial \delta}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial v_j} \times \frac{\partial v_j}{\partial hl} \times \frac{\partial hl}{\partial \widehat{hl}} \times \frac{\partial \widehat{hl}}{\partial w_h}$$
$$= -2 \times error \times relu'(v_i) \times w_{out} \times hl' \times x$$

$$\frac{\partial \delta(n)}{\partial b_h} = \frac{\partial \delta}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial v_j} \times \frac{\partial v_j}{\partial hl} \times \frac{\partial hl}{\partial \widehat{hl}} \times \frac{\partial \widehat{hl}}{\partial b_h}$$
$$= -2 \times error \times relu'(v_i) \times w_{out} \times hl'$$

In order to determine the learning rate and neuron numbers in the hidden layer, validation is performed with the rate of 10% of the data. In order to achieve the minimum error, maximum score, we trained our model several times with various learning rates and neuron numbers combinations. The combination of the arrays [0.01, 0.0075, 0.005, 0.0025, 0.001] and [2, 4, 8, 16, 32] are used as a learning rate and neuron numbers respectively. It is observed that the best score is reached with the values of 0.001 for learning rate and 16 for hidden layer sizes. By using these values, the MSE is converged in 6 epochs as it can be seen in in the figure below. The whole process took 9.695 seconds. We calculated the score with the R-squared method.

```
Test size: 13485
Validation size: 10788
Train size: 29667
Learning rate is: 0.001
Neuron number in hidden layer is: 16
It has taken 9.69538426399231 seconds to train the network
The Score:  0.9293697979187309
```

Fig 3: Results of Neural Network

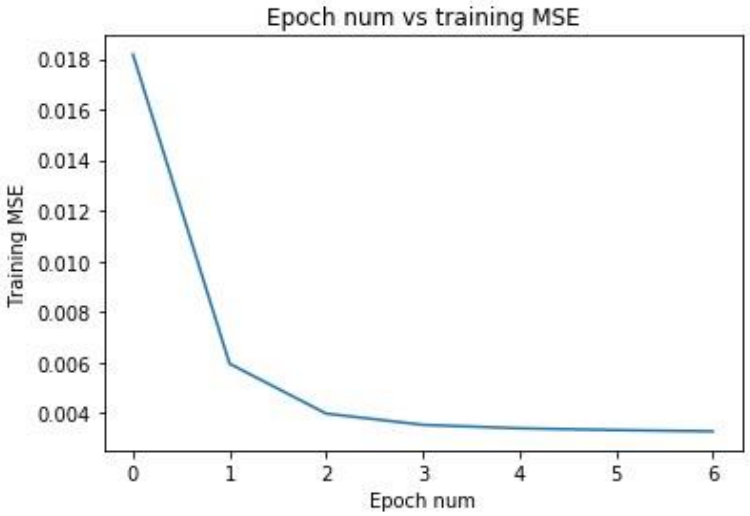

Fig 4: Training MSE vs epoch

Furthermore, the score is 0.93 which means that the prediction of our model is close to the real values. The prediction vs real values graph is in the blow figure. The orange lines are the predicted values and the blue ones are the real values.

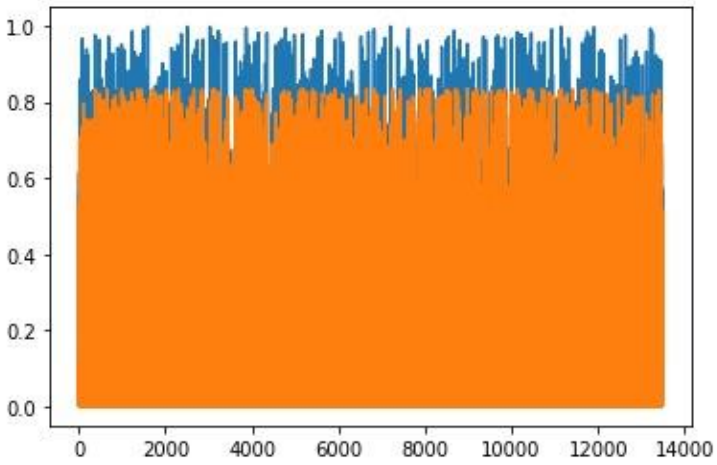

Fig 5: Real test values vs predicted test values
(blue is real and orange is predicted)

3. Ridge Regression

To understand and implement the Ridge Regression algorithm, it is important to understand the Linear Regression algorithm completely. Linear Regression tries to find a relation between the features and the output linearly. However, it is open to overfitting to the train data and cause high error rates for test data. [3] To avoid this possible problem, weights are penalized with a lambda value in Ridge Regression algorithm. With this knowledge and the fact that the group members learn machine learning for the first time, the Ridge Regression is chosen since it is discussed in lectures. After the section of pre-processing, the arranged dataset's 65% is used for training, 10% is used for validation and 25% is used for testing.

The loss function of ridge regression:

$$Loss_R(\beta,\lambda) = \sum_{i=1}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1} \beta_j^2$$

$$= (y - x\beta)^T (y - x\beta) + \lambda \beta^T \beta$$

$$y^T y - y^T x\beta - \beta^T x^T y + \beta^T x^T x\beta + \lambda \beta^T I_p \beta, \quad \text{where } I_p \textit{is a identity matrix}$$

$$\frac{\partial Loss_R(\beta,\lambda)}{\partial \beta} = 0 = -y^T x - y^T x + 2\beta^T x^T x + 2\lambda\beta^T$$

$$(x^T)^T = \left((x^T x + \lambda I)^T \beta\right)^T$$

$$\hat{\beta} = (x^T x + \lambda I)^{-1} x^T y$$

The beta (β) value is calculated as 0.2 and the lambda (λ) values is determined with the consideration of the bias-variance tradeoff. In order not to be over-fit and under-fit, the lambda value is selected as 0.05. We reached that value after various of trials. By applying these values, the following MSE values are obtained:

$$MSE = \frac{1}{n}\sum(y_i - \hat{y}_i)^2$$

```
Validation MSE are:
[0.00456381 0.00453186 0.00451274 0.00449219]
Lambda is: 0.05
It has taken 0.009964704513549805 seconds to train the algorithm
The Score: 0.8449379445647502
```

Fig 6: Results of Ridge Regression

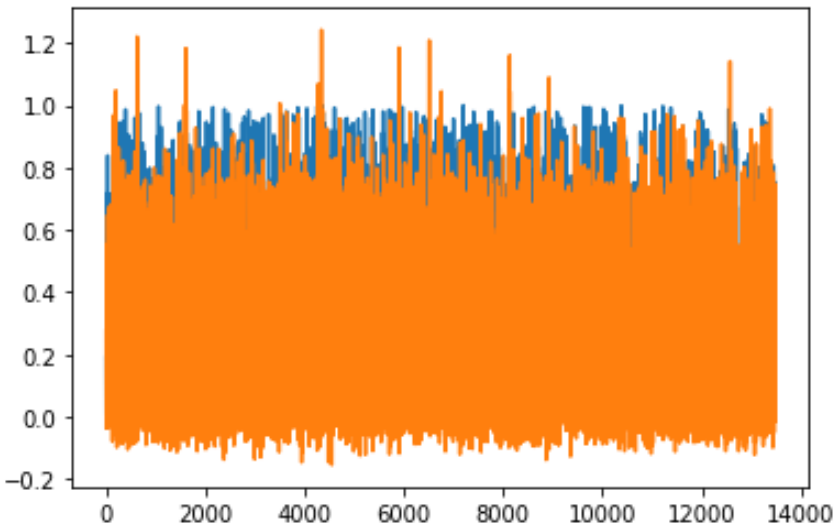

Fig 7: Real test values vs predicted test values
(blue is real and orange is predicted)

In order to optimize we used the gradient descent method. After the application, the new MSE results are obtained;

```
Validation MSE are:
[[0.01556189 0.01582542 0.01491645 0.01512834]
 [0.00910215 0.00871456 0.00876392 0.00924873]
 [0.00596067 0.00599712 0.00611415 0.00601432]
 [0.00547185 0.00537256 0.005301   0.005268  ]
 [0.00543584 0.00528984 0.00532838 0.00521156]
 [0.00520102 0.00523944 0.00522071 0.00522349]
 [0.00499695 0.00517064 0.0050685  0.00508967]
 [0.00485064 0.00484063 0.00489539 0.00485112]]
Learning rate is: 0.2
Lambda is: 0.1
It has taken 78.63296222686768 seconds to train the algorithm
MSE is: 0.004698489316466275
The Score:  0.8871371640286285
```

Fig 8: Results of Ridge Regression with Gradient Descent

Furthermore, the score is 0.84 and 0.88 respectively. This means that the prediction of our model is close to the real values. The prediction vs real values graph is in the blow figure. The orange lines are the predicted values and the blue ones are the real values.

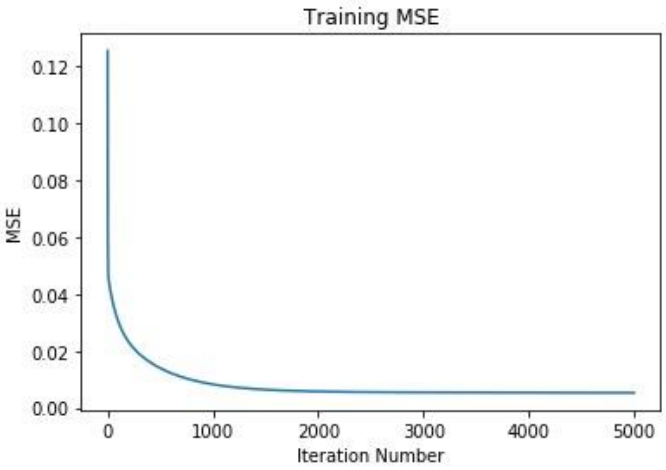

Fig 9: Training MSE vs 5000 iteration

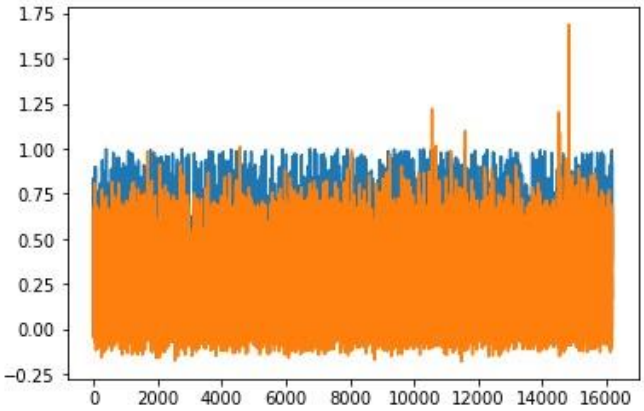

Fig 10: Real test values vs predicted test values
(blue is real and orange is predicted)

4. Decision Tree

Decision tree algorithm is the last method we used to solve the diamond estimation problem. In this method, rules are learnt and the output is decided according to the receiving input [3]. We started to implement this method after the First Report and Demonstration. Our dataset includes 9 different features which have the values in the range of [0,1] after normalization. The aim of the decision tree is to separate top node into two sub nodes that have the possible least error result. This flows until the end. The algorithm's ultimate goal is to find the thresholds in order to divide into two different sub nodes. For this purpose, each input is tried to minimize the error for the current feature. During these repetitions, the algorithm keeps the results to decide the feature-threshold dual. As a result, the most proper threshold value with the feature is chosen and the node is divided. The repetition occurs till the maximum depth or till there would be one entry for each sub node.

For imagination which also helped us a lot, the tree can be thought with leaves and branches. Besides, a sample decision tree schematic is prepared as follows:

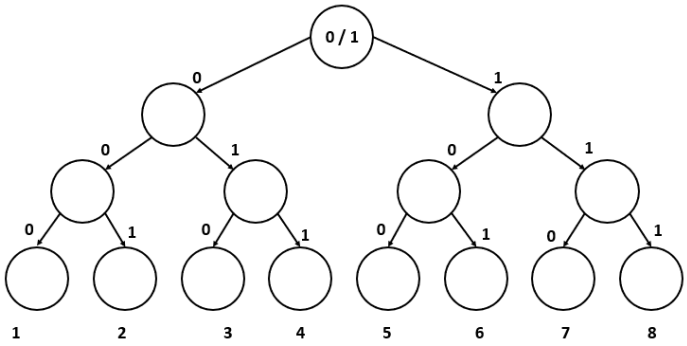

Fig 11: sample decision tree schematic

Implementing the algorithm more easily, a class object named "DecisionTree" is created with the initialization of left and right nodes, feature, threshold and the prediction. The left and right are used to decide the position of the node according to the parent node. The error is calculated at every step to reach the minimum error. By choosing RSS as infinity at the beginning and by updating it after reaching less error value of the error calculated at that step compared to the previous value. As a result, the minimum RSS is found at every step. The formula is:

$$RSS = \sum_{i=1}^{k} (\hat{y}_i - y_i)^2$$

The optimum values for maximum depth is found as 10 by trying several values. The results of this validation process and the R^2 score can be seen in the following figure:

```
Validation MSE are:
[[0.00308759 0.00183517 0.00239954 0.00262244]]
Maximum depth of node is: 10
It has taken 42.049747943878174 seconds to train the network
The Score is:  0.9562909339715471
```

Fig 12: Results of Decision Tree

```
Test size: 13485
Validation size: 10788
Train size: 29667
Maximum depth of node is: 10
It has taken 5.21774435043335 seconds to train the network
The Score:  0.9600982484650987
```

Fig 13: Results of Decision Tree with max depth of node is 10

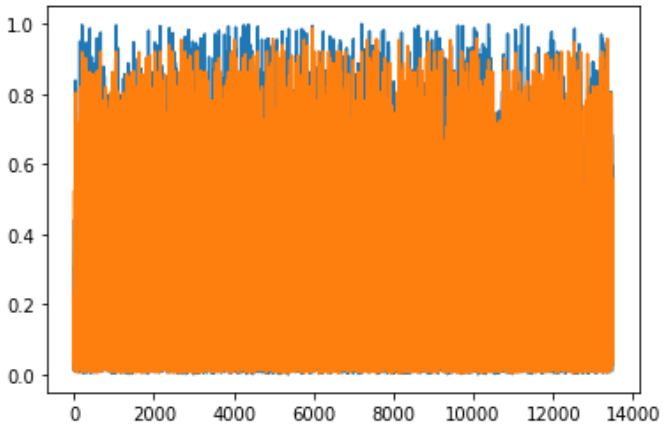

Fig 14: Real test values vs predicted test values
(blue is real and orange is predicted)

Results and Conclusion

According to the Table 1 which can be seen in below, the training lasted 9.695 seconds for Neural Network, 0.010 seconds for Ridge Regression, 0.005 seconds for Ridge Regression with Gradient Descent and 5.22 seconds for Decision Tree algorithms. The longest method to train is the Neural Network and the shortest method to train is the Ridge Regression. The R^2 scores take values between 0 and 1. While 1 represents the high accuracy, 0 represents the low accuracy. When the R^2 scores are checked, the scores are 0.929 for Neural Network, 0.845 for Ridge Regression, 0.887 for Ridge Regression with Gradient Descent and 0.960 for Decision Tree algorithms. The best score is reached with Decision Tree algorithm and the worst result is reached with the Ridge Regression algorithm.

To discuss the method which best fits to the problem that we aimed to solve that was predicting the diamond prices, it is important to find the most efficient algorithm in terms of both required time to train and R^2 score. It can be seen that the Ridge Regression with Gradient Descent gives more accurate results than the Ridge Regression in closed form with a slightly more required time. It can be also observed that the Decision Tree algorithm results in both more accurate and less time requiring than the Neural Network algorithm. As a result, if the Ridge Regression with Gradient Descent and Decision Tree is compared, the choice depends on the user preference. The time required to train Decision Tree algorithm is too much compared to the Ridge Regression with Gradient Descent. However, the R^2 score of the Decision Tree algorithm is much more as well. Therefore, according to our criteria, we decided that the R^2 score is more important than the time required for our case with this dataset and decided on the Decision Tree algorithm as the best fitting model. However, we acknowledge that there might be different datasets with more indices inside and might make the usage of the Decision Tree algorithm harder due to increasing duration to train.

We benefitted from working on this project and completing it successfully since this project provided the chance to implement three of the methods that are covered in the class this semester. Not using the built-in Machine Learning libraries during the implementation of the algorithms we used really challenged us but, it has provided us to test and develop our coding and analytical thinking skills. We believe that we learnt a lot from this project.

Algorithm	Time required (s)	R^2 Score
Neural Network	9.695	0.929
Ridge Regression	0.010	0.845
Ridge Regression – Gradient Descent	0.005	0.887
Decision Tree	5.220	0.960

Table 1: Required time and R^2 Score for each algorithm

Gannt Chart

The Gannt Chart is prepared according to the roles of the team members with at which week these workloads are completed. The chart is colored according to the responsible member.

Gannt Chart of EEE485 Term Project

	Weeks		Proposal		First Report						Final Report				
What is done	Start Week	End Week	Week 3	week 4	week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14	Week 15
Project Proposal	3	4													
Project Search	3	4													
Dataset Search	3	4													
Method Search	3	4													
Literature Review	3	4													
Report Writing	3	4													
First Report	5	10													
Dataset Pre-processing	5	6													
Ridge Regression Algorithm	6	8													
Neural Network Algorithm	6	8													
Performance Analysis	8	10													
Report Writing	8	10													
Final Report	11	15													
Improvements according to feedback	11	13													
Decision Tree Algorithm	12	14													
Performance Analysis	13	15													
Report Writing	13	15													

Both Group Members
Batuhan
Kıvanç

Table 2: Gannt Chart

References:

[1] Kaggle, *diamonds*, 2021. [.csv]. Available: https://www.kaggle.com/datasets/resulcaliskan/diamonds. [Accessed: 07-Apr-2022]

[2] T. Hastie, J. Friedman, and R. Tisbshirani, The elements of Statistical Learning: Data Mining, Inference, and prediction. New York: Springer, 2009.

[3] G. James, D. Witten, T. Hastie, and R. Tibshirani, An introduction to statistical learning: With applications in R. New York: Springer, 2013.

Appendices

Appendix A

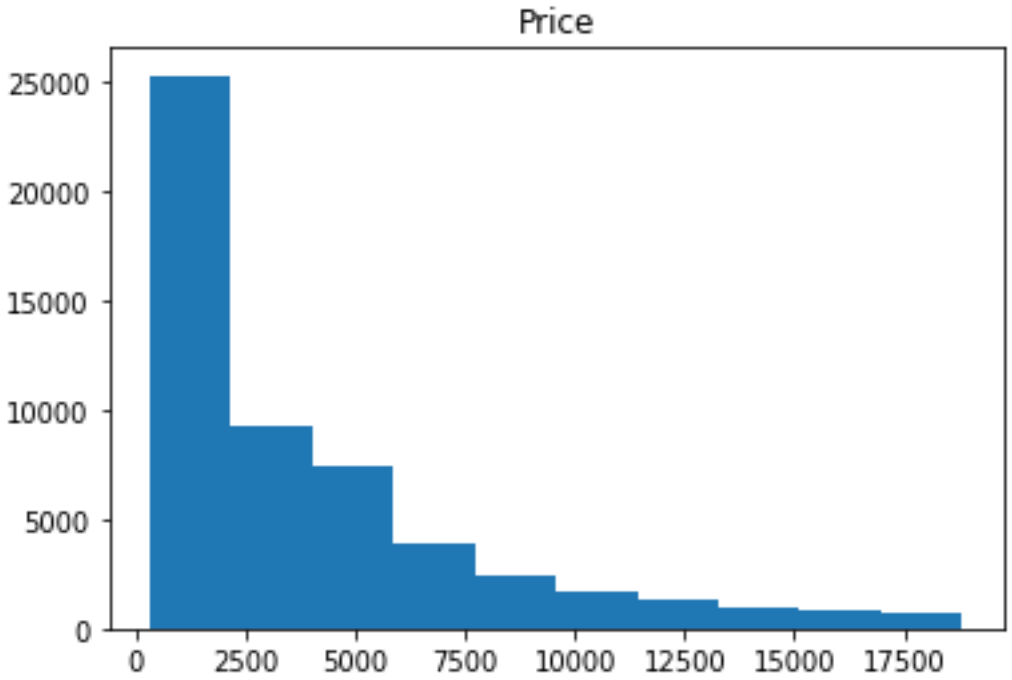

Fig 15: Price label distribution

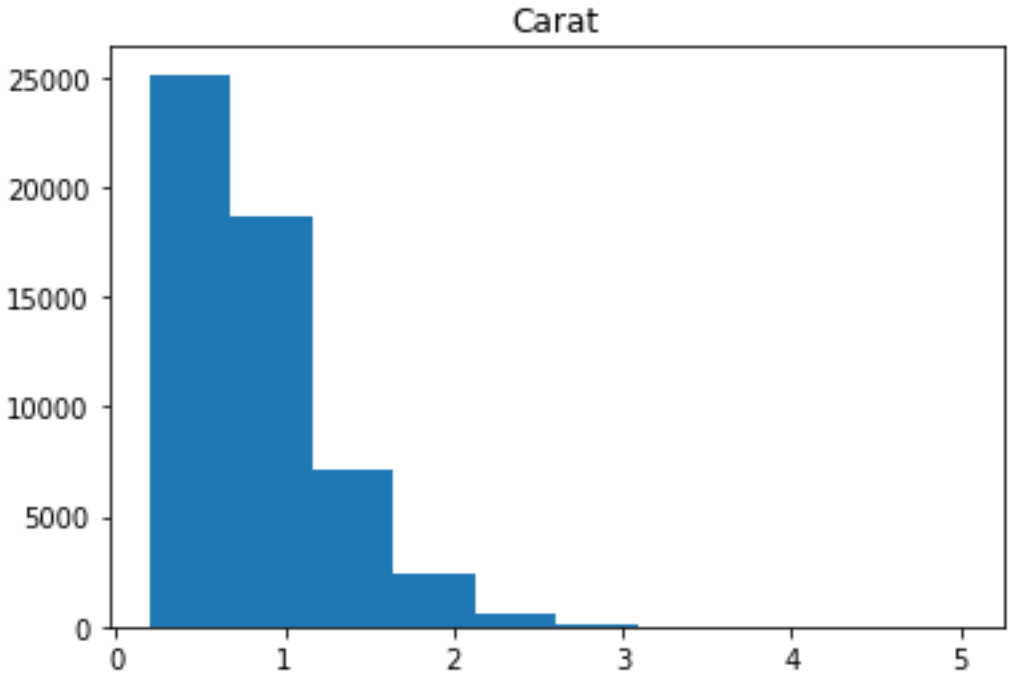

Fig 16: Carat feature distribution

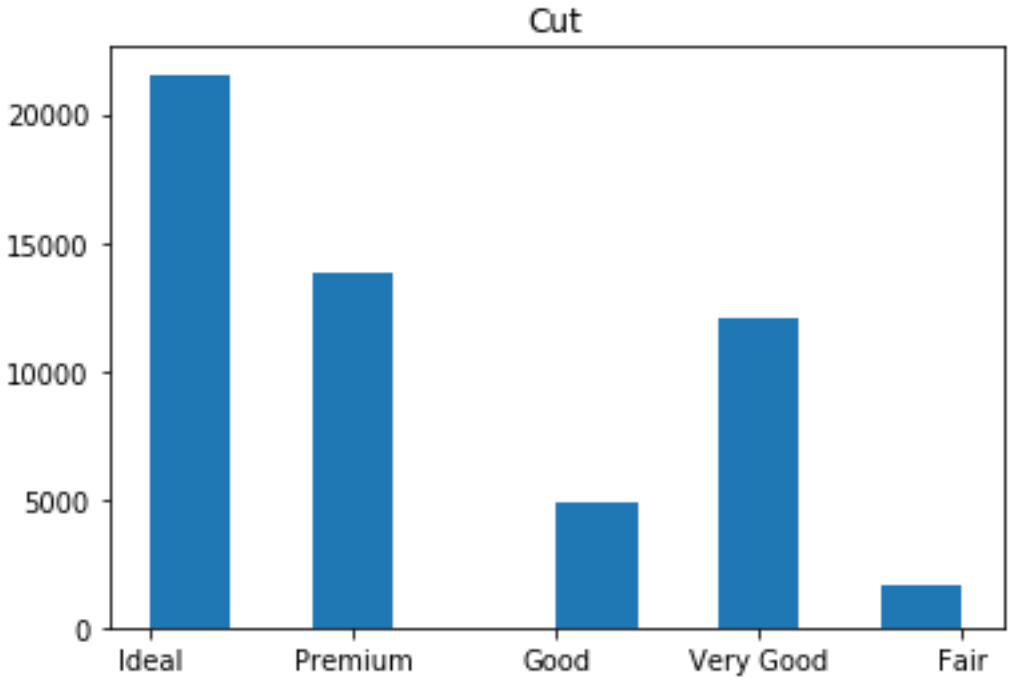

Fig 17: Cut feature distribution

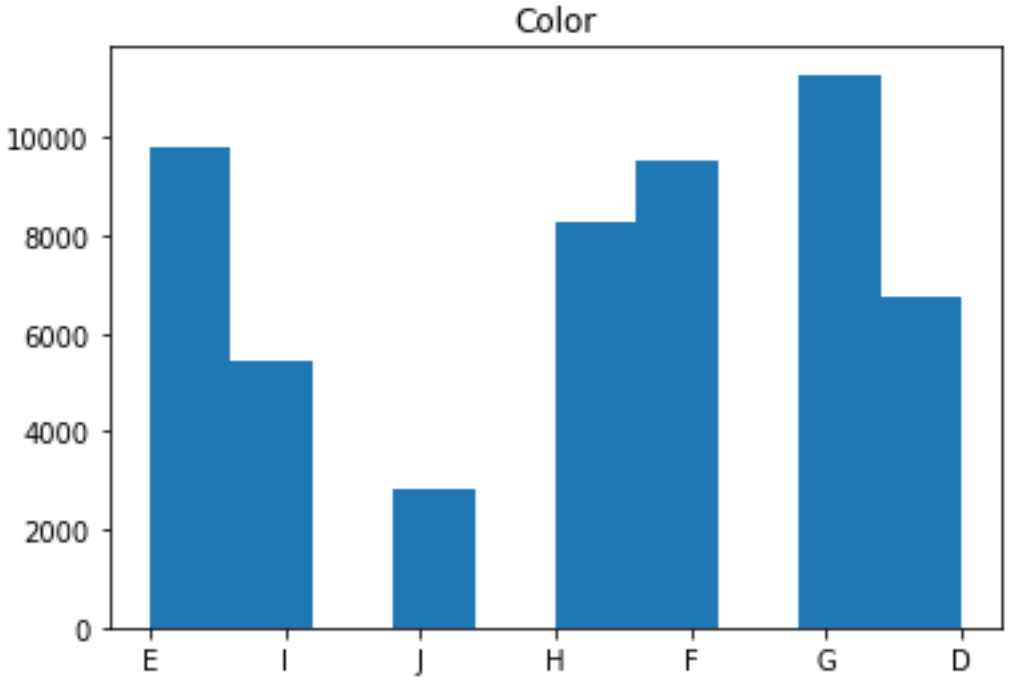

Fig 18: Color feature distribution

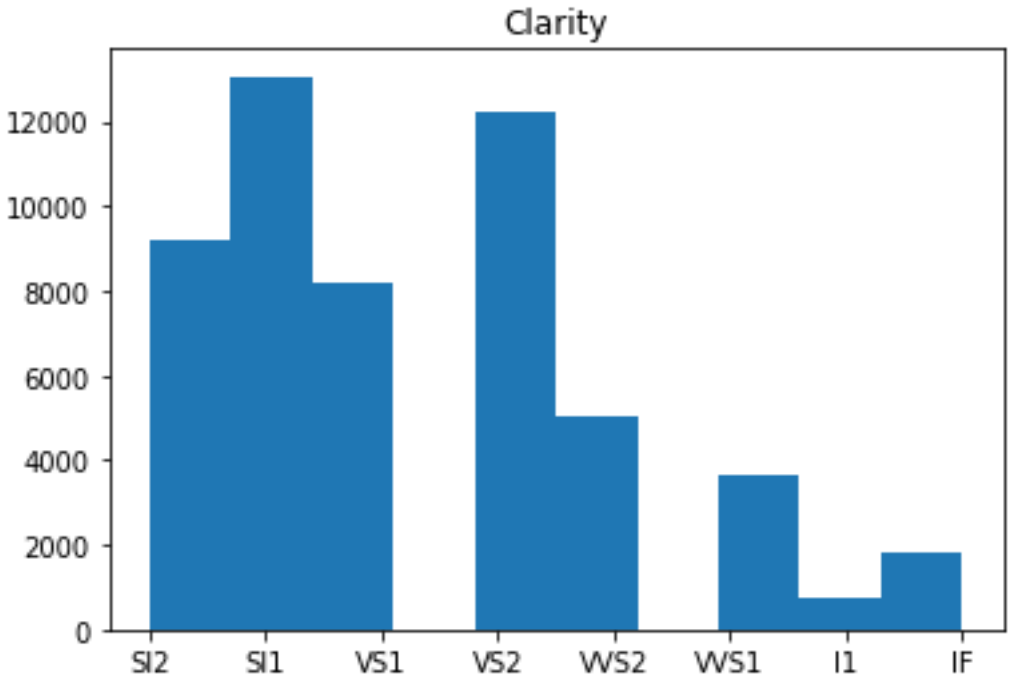

Fig 19: Clarity feature distribution

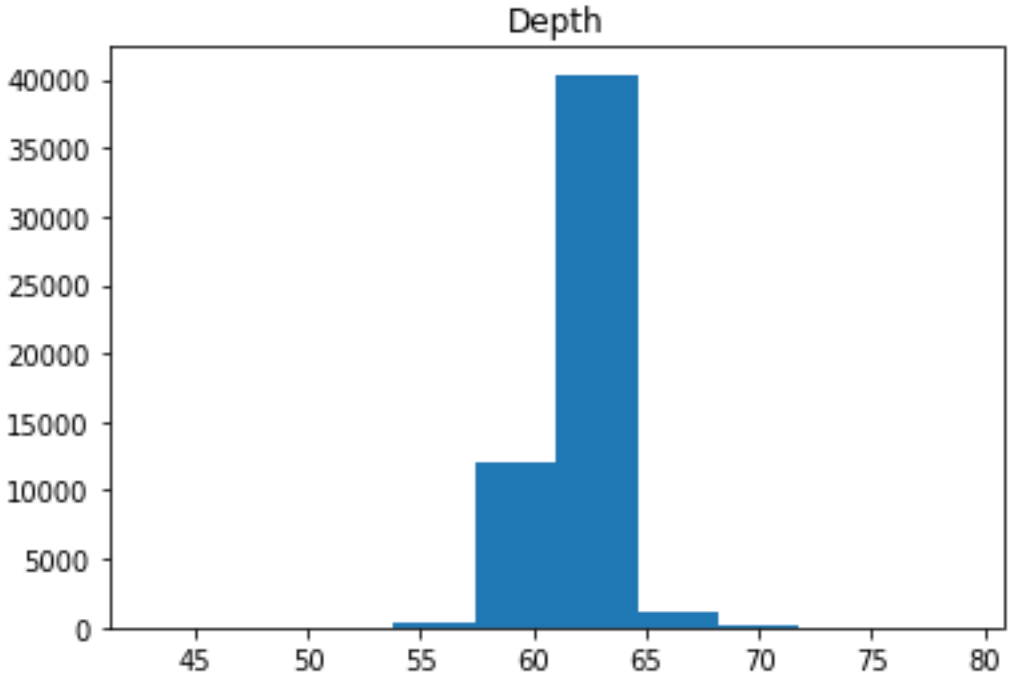

Fig 20: Depth feature distribution

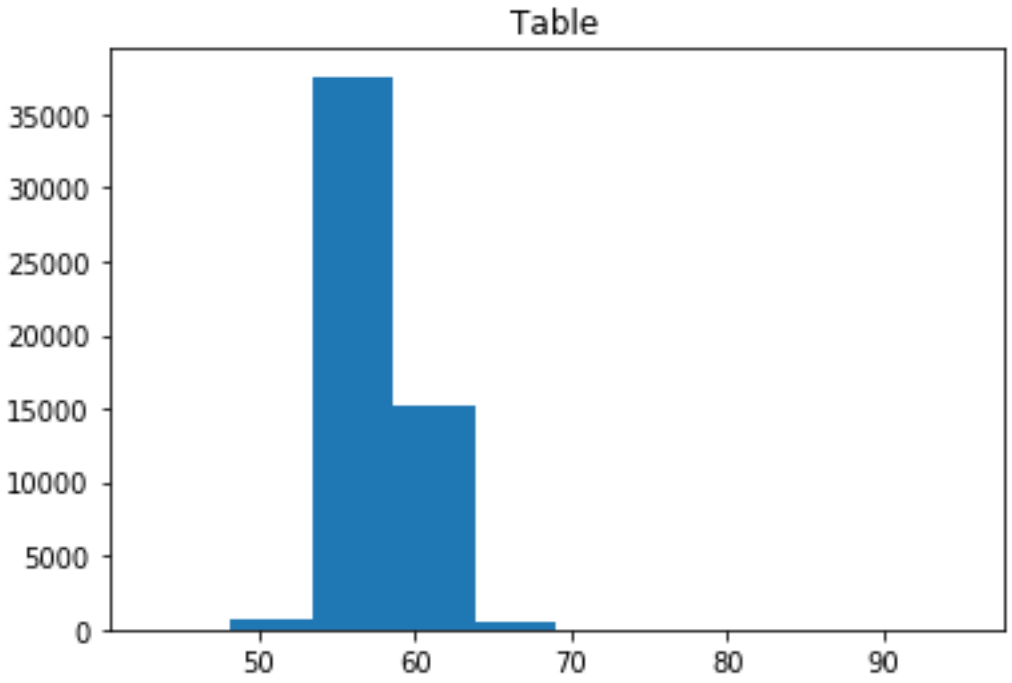

Fig 21: Table feature distribution

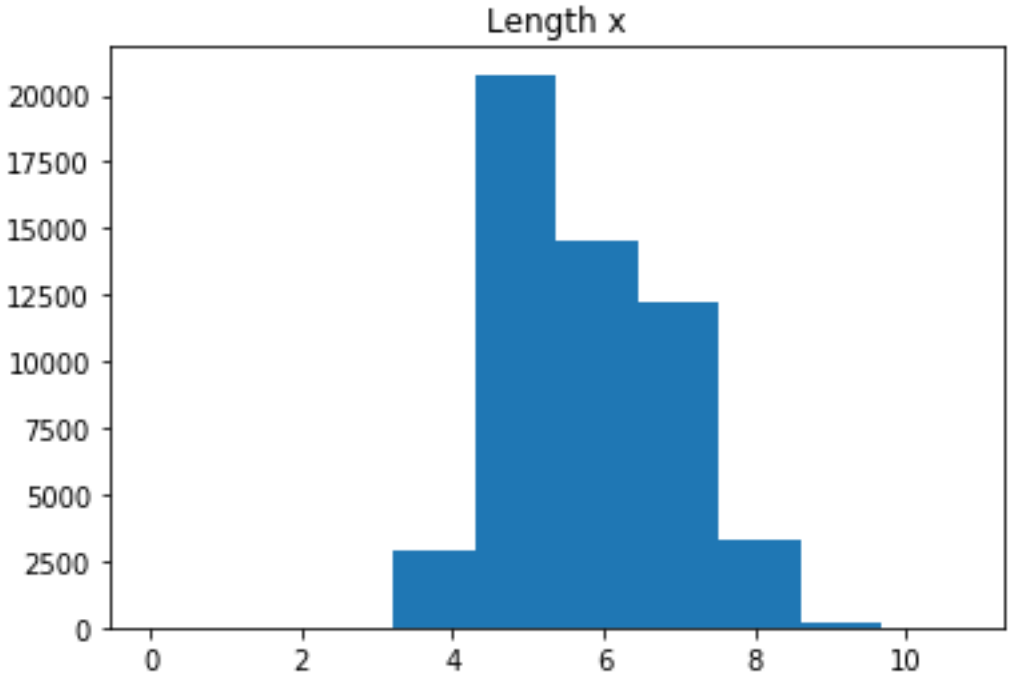

Fig 22: X value feature distribution

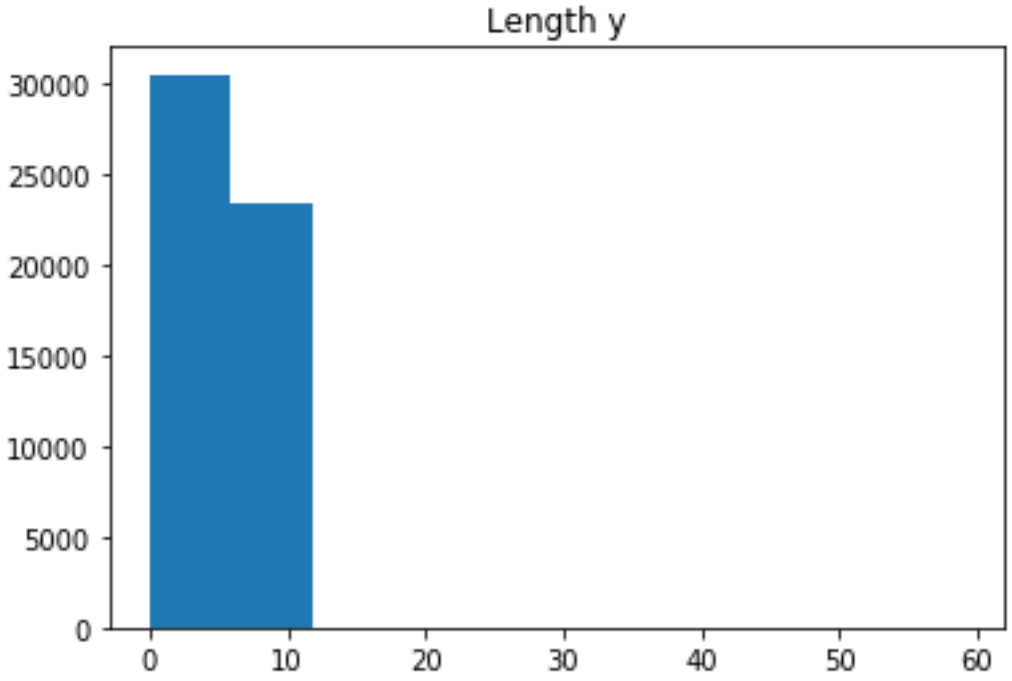

Fig 23: Y value feature distribution

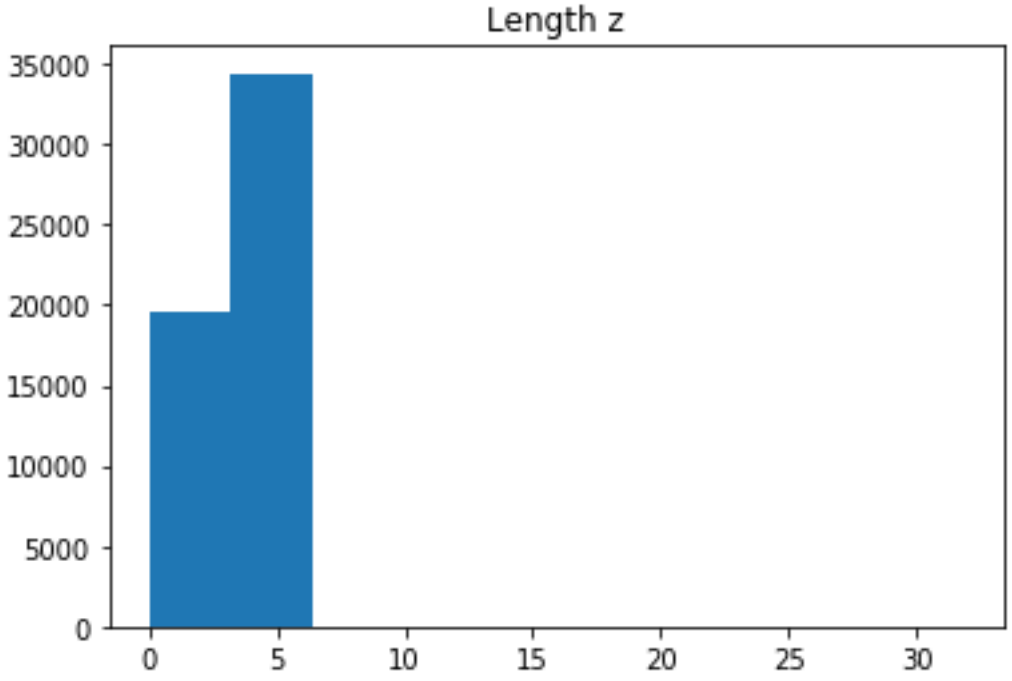

Fig 24: Z value feature distribution

Appendix B

Pre-processing Codes

```
#Importing Libraries
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd

diamond_data = pd.read_csv("diamonds.csv")
#diamond_data = diamond_data.dropna()

features = diamond_data.columns

plt.figure()
plt.title("Carat")
plt.hist(diamond_data.carat)

plt.figure()
plt.title("Cut")
plt.hist(diamond_data.cut)

plt.figure()
plt.title("Color")
plt.hist(diamond_data.color)

plt.figure()
plt.title("Clarity")
plt.hist(diamond_data.clarity)

plt.figure()
plt.title("Depth")
plt.hist(diamond_data.depth)

plt.figure()
plt.title("Table")
plt.hist(diamond_data.table)

plt.figure()
plt.title("Length x")
plt.hist(diamond_data.x)

plt.figure()
plt.title("Length y")
plt.hist(diamond_data.y)

plt.figure()
plt.title("Length z")
plt.hist(diamond_data.z)
```

```
plt.figure()
plt.title("Price")
plt.hist(diamond_data.price)

for i in range(53940):
    if diamond_data.cut[i] == "Ideal":
        diamond_data.cut[i] = 1
    elif diamond_data.cut[i] == "Premium":
        diamond_data.cut[i] = 0.75
    elif diamond_data.cut[i] == "Very Good":
        diamond_data.cut[i] = 0.5
    elif diamond_data.cut[i] == "Good":
        diamond_data.cut[i] = 0.25
    else:
        diamond_data.cut[i] = 0

for i in range(53940):
    if diamond_data.color[i] == "D":
        diamond_data.color[i] = 1
    elif diamond_data.color[i] == "E":
        diamond_data.color[i] = 0.83
    elif diamond_data.color[i] == "F":
        diamond_data.color[i] = 0.67
    elif diamond_data.color[i] == "G":
        diamond_data.color[i] = 0.5
    elif diamond_data.color[i] == "H":
        diamond_data.color[i] = 0.33
    elif diamond_data.color[i] == "I":
        diamond_data.color[i] = 0.16
    else:
        diamond_data.color[i] = 0

for i in range(53940):
    if diamond_data.clarity[i] == "IF":
        diamond_data.clarity[i] = 1
    elif diamond_data.clarity[i] == "VVS1":
        diamond_data.clarity[i] = 0.86
    elif diamond_data.clarity[i] == "VVS2":
        diamond_data.clarity[i] = 0.72
    elif diamond_data.clarity[i] == "VS1":
        diamond_data.clarity[i] = 0.58
    elif diamond_data.clarity[i] == "VS2":
        diamond_data.clarity[i] = 0.44
    elif diamond_data.clarity[i] == "SI1":
        diamond_data.clarity[i] = 0.28
    elif diamond_data.clarity[i] == "SI2":
        diamond_data.clarity[i] = 0.14
    else:
        diamond_data.clarity[i] = 0
```

```
#Finding parameters for Normalization
max_carat = max(diamond_data.carat)
min_carat = min(diamond_data.carat)
max_depth = max(diamond_data.depth)
min_depth = min(diamond_data.depth)
max_table = max(diamond_data.table)
min_table = min(diamond_data.table)
max_x = max(diamond_data.x)
min_x = min(diamond_data.x)
max_y = max(diamond_data.y)
min_y = min(diamond_data.y)
max_z = max(diamond_data.z)
min_z = min(diamond_data.z)
max_price = max(diamond_data.price)
min_price = min(diamond_data.price)

#Normalization
diamond_data.carat = (np.array(diamond_data.carat)-min_carat)/(max_carat-min_carat)
diamond_data.depth = (np.array(diamond_data.depth)-min_depth)/(max_depth-min_depth)
diamond_data.table = (np.array(diamond_data.table)-min_table)/(max_table-min_table)
diamond_data.x = (np.array(diamond_data.x)-min_x)/(max_x-min_x)
diamond_data.y = (np.array(diamond_data.y)-min_y)/(max_y-min_y)
diamond_data.z = (np.array(diamond_data.z)-min_z)/(max_z-min_z)
diamond_data.price = (np.array(diamond_data.price)-min_price)/(max_price-min_price)

shuffled = diamond_data.sample(frac = 1)
#shuffled.to_csv("preprocessed_diamond_data.csv", index=False)
```

Neural Network Codes

```
#Importing Libraries and Functions defined
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from time import time
from last_functions import *

#Getting the data
diamond_dataset = pd.read_csv("preprocessed_diamond_data.csv")
Y = np.array(diamond_dataset.price)                  #labels
X = np.array(diamond_dataset.drop("price",axis = 1))    #features

#Separating the dataset
length = Y.shape[0]
datum = np.arange(0, length)
test_length = int(np.round(0.25 * length))
print("Test size: {0}".format(test_length))
validation_length = int(np.round(0.1 * length))
```

```
print("Validation size: {0}".format(validation_length))
train_length = length - test_length - validation_length
print("Train size: {0}".format(train_length))

test_datum = datum[0: test_length]
validation_datum = datum[test_length: test_length + validation_length]
train_datum = datum[test_length + validation_length:]

test_Y = Y[test_datum]
test_X = X[test_datum, :]
validation_Y = Y[validation_datum]
validation_X = X[validation_datum, :]
train_Y = Y[train_datum]
train_X = X[train_datum, :]

#Training Data
start_time = time()
nn = NeuralNetwork(neuron_number_in_hidden_layer = 16, num_of_features = 9, learning_rate =
0.001)
loss_history = nn.training(train_X, train_Y)
print("Learning rate is: 0.001")
print("Neuron number in hidden layer is: 16")
print("It has taken {0} seconds to train the network".format(time() - start_time))

plt.figure()
plt.plot(loss_history)
plt.title("Epoch num vs training MSE")
plt.xlabel("Epoch num")
plt.ylabel("Training MSE")

test_Y_predicted = nn.predict(test_X)
print("The Score: ", (R_squared(test_Y_predicted, test_Y)))

plt.figure()
plt.plot(test_Y, label="line1")
plt.plot(test_Y_predicted, label="line2")
```

Ridge Regression Codes

```
#Importing Libraries and Functions defined
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from time import time
from last_functions import *

#Getting the data
diamond_dataset = pd.read_csv("preprocessed_diamond_data.csv")
Y = np.array(diamond_dataset.price)                #labels
```

```
X = np.array(diamond_dataset.drop("price",axis = 1))    #features

#Separating the dataset
length = Y.shape[0]
datum = np.arange(0, length)
test_length = int(np.round(0.25 * length))
print("Test size: {0}".format(test_length))
validation_length = int(np.round(0.1 * length))
print("Validation size: {0}".format(validation_length))
train_length = length - test_length - validation_length
print("Train size: {0}".format(train_length))

test_datum = datum[0: test_length]
validation_datum = datum[test_length: test_length + validation_length]
train_datum = datum[test_length + validation_length:]

test_Y = Y[test_datum]
test_X = X[test_datum, :]
validation_Y = Y[validation_datum]
validation_X = X[validation_datum, :]
train_Y = Y[train_datum]
train_X = X[train_datum, :]

#Training Data Direct
start1 = time()
training_mse, beta_values, bias_value = ridge_regression_algorithm_direct(train_X, train_Y, 0.05)
print("The training takes {0} seconds".format(time() - start1))
test_Y_predicted = bias_value + test_X.dot(beta_values)
print("The Score: ", (R_squared(test_Y_predicted, test_Y)))

#Training Data with Gradient Descent
start2 = time()
MSE_values_array, beta_values, bias_value = ridge_regression_algorithm(train_Y, train_X, 0.2,
0.1, 5000)
print()
print("The training takes {0} seconds".format(time() - start2))
print("MSE is: {0}".format(MSE_values_array[-1]))

plt.figure()
plt.plot(MSE_values_array,'r')
plt.title("MSE Values of Train Set")
plt.xlabel("Iteration")
plt.ylabel("MSE Values")

test_Y_predicted = bias_value + test_X.dot(beta_values)
print("The Score: ", (R_squared(test_Y_predicted, test_Y)))

plt.figure()
plt.plot(test_Y, label="line1")
plt.plot(test_Y_predicted, label="line2")
```

Decision Tree Codes

```
#Importing Libraries and Functions defined
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from time import time
from last_functions import *

#Getting the data
diamond_dataset = pd.read_csv("preprocessed_diamond_data.csv")
Y = np.array(diamond_dataset.price)
X = np.array(diamond_dataset.drop("price", axis = 1))

#Separating the dataset
length = Y.shape[0]
datum = np.arange(0, length)
test_length = int(np.round(0.25 * length))
print("Test size: {0}".format(test_length))
validation_length = int(np.round(0.2 * length))
print("Validation size: {0}".format(validation_length))
train_length = length - test_length - validation_length
print("Train size: {0}".format(train_length))

test_datum = datum[0: test_length]
validation_datum = datum[test_length: test_length + validation_length]
train_datum = datum[test_length + validation_length:]

test_Y = Y[test_datum]
test_X = X[test_datum, :]
validation_Y = Y[validation_datum]
validation_X = X[validation_datum, :]
train_Y = Y[train_datum]
train_X = X[train_datum, :]


start_time = time()
dt = training_dt(train_X, train_Y, 0, max_depth = 10)
print("Maximum depth of node is: 10")
print("It has taken {0} seconds to train the network".format(time() - start_time))

test_Y_predicted = predict_set(test_X, dt)
print("The Score: ", (R_squared(test_Y_predicted, test_Y)))

plt.figure()
plt.plot(test_Y, label="line1")
plt.plot(test_Y_predicted, label="line2")
```

Functions used in different script

```
import numpy as np

#Score Calculating
def R_squared(prediction_y, real_y):
    above = np.sum(np.square(real_y - prediction_y))
    below = np.sum(np.square(prediction_y - np.mean(real_y)))
    return (1- above/below)

#Ridge Regression
def ridge_regression_algorithm_direct(X, Y, lamda_value):
    I = np.identity(X.shape[1]) #identity matrix
    beta_values = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X) + lamda_value * I), X.T), Y)   #Direct formula
    return 0, beta_values, 0

def ridge_regression_algorithm(train_Y, train_X, decided_learning_rate, decided_lamda, iteration):
    feature_number = 9
    data_number = 53940
    bias_value = np.random.normal(0, 0.1, 1)
    beta_values = np.random.normal(0, 0.1, feature_number)
    MSE_values_array = np.zeros(iteration)

    for i in range(0, iteration):
        y_predicted = train_X.dot(beta_values) + bias_value

        error_value = train_Y - y_predicted            #Direct error
        squared_error_value = np.square(error_value)    #Take square
        MSE_value = np.mean(squared_error_value)        #Take mean
        MSE_values_array[i] = MSE_value

        Derivated_MSE = -2 * (train_X.T).dot(error_value)
        Derivated_Lamda = 2 * decided_lamda * beta_values
        Derivated_Beta = (Derivated_MSE + Derivated_Lamda) / data_number
        Derivated_Bias = -2 * np.mean(np.sum(error_value)) / data_number

        beta_values = beta_values - decided_learning_rate * Derivated_Beta
        bias_value = bias_value - decided_learning_rate * Derivated_Bias

    return MSE_values_array, beta_values, bias_value


#Neural Network
def relu(x_value):              #relu activation func
    return np.maximum(0, x_value)

def relu_grad(x_value):
    return (x_value > 0) * 1
```

```
class NeuralNetwork():
    def __init__(self, neuron_number_in_hidden_layer, num_of_features, learning_rate):
        self.bias_hidden_layer = np.random.normal(0, 0.3, size = (1, neuron_number_in_hidden_layer))  #Bias for hidden layer
        self.weight_hidden_layer= np.random.normal(0, 0.3, size = (num_of_features, neuron_number_in_hidden_layer)) #Weight for hidden layer
        self.bias_output = np.random.normal(0, 0.3, size = (1, 1))  #Bias for output
        self.weight_output = np.random.normal(0, 0.3, size = (neuron_number_in_hidden_layer, 1)) #Weight for output
        self.learning_rate = learning_rate  #learning rate

    def train_one_iteration(self, train_X, train_Y):
        loss = 0
        num_of_samples = train_Y.shape[0]
        x_value = np.zeros((1, train_X.shape[1]))

        for i in range(num_of_samples):
            x_value[0, :] = train_X[i, :]
            y_value = train_Y[i]

            hidden_layer_before = x_value.dot(self.weight_hidden_layer) + self.bias_hidden_layer
            hidden_layer = relu(hidden_layer_before)  #putting into ReLU function
            output = hidden_layer.dot(self.weight_output) + self.bias_output  #output
            test_Y_predicted = relu(output)  #putting into ReLU function

            error_value = y_value - test_Y_predicted  #Error calculation
            loss = loss + error_value * error_value/num_of_samples  #Loss calculation

            weight_output_grad = -2 * error_value * hidden_layer.T * relu_grad(output)  #derivative
            bias_output_grad = -2 * error_value * relu_grad(output)  #derivative
            weight_hidden_layer_gradient = -2 * error_value * relu_grad(output) * x_value.T * (self.weight_output.T * relu_grad(hidden_layer_before)) #derivative
            bias_hidden_layer_gradient = -2 * error_value * relu_grad(output) * (self.weight_output.T * relu_grad(hidden_layer_before))  #derivative

            self.weight_output = self.weight_output - self.learning_rate * weight_output_grad  #update
            self.bias_output = self.bias_output - self.learning_rate * bias_output_grad  #update
            self.weight_hidden_layer = self.weight_hidden_layer - self.learning_rate * weight_hidden_layer_gradient #update
            self.bias_hidden_layer = self.bias_hidden_layer - self.learning_rate * bias_hidden_layer_gradient  #update

        return loss

    def training(self, train_X, train_Y, maximum_epoch = 25, threshold_to_stop = 0.00005):
        loss_record = []
        for i in range(maximum_epoch):  #Iteration on each epoch
            loss = NeuralNetwork.train_one_iteration(self, train_X, train_Y)  #Training each epoch
```

```
                loss_record.append(loss[0][0])                                  #Finding and recording loss after
each epoch
                if (i >= 1) and (loss_record[-2] - loss_record[-1] < threshold_to_stop):  #Condition to stop
the training when converged
                    break
            return loss_record

        def predict(self, train_X):
            test_Y_predicted_array = np.zeros(train_X.shape[0])     #create the array
            x_value = np.zeros((1, train_X.shape[1]))               #create the array

            for i in range(train_X.shape[0]):
                x_value[0, :] = train_X[i, :]

                hidden_layer_before = x_value.dot(self.weight_hidden_layer) + self.bias_hidden_layer
                hidden_layer = relu(hidden_layer_before)            #hidden layer update

                output = hidden_layer.dot(self.weight_output) + self.bias_output
                test_Y_predicted = relu(output)         #Getting the predictions
                test_Y_predicted_array[i] = test_Y_predicted

            return test_Y_predicted_array

    #Decision Tree
    class DecisionTree():
        def __init__(self):
            self.first = None             # first (left) node
            self.sec = None               # second (right) node
            self.feature = None
            self.threshold_to_stop = None  #threshold value
            self.prediction_y = None

    def RSS(first_node, sec_node):
        first_rss = np.sum(np.square(first_node - np.mean(first_node)))
        sec_rss = np.sum(np.square(sec_node - np.mean(sec_node)))
        rss = first_rss + sec_rss
        return rss

    def basepredict_dt(test, dt):               #predicts the initial condition
        while (dt.prediction_y is None):
            feature = dt.feature
            threshold_to_stop = dt.threshold_to_stop
            if test[feature] < threshold_to_stop:
                dt = dt.sec
            else:
                dt = dt.first
        return dt.prediction_y

    def predict_set(test_X, dt):              #next prediction acc. to previous
        test_Y_predicted_array = np.zeros(np.shape(test_X)[0])
```

```
    for i in range(np.shape(test_X)[0]):
        test_Y_predicted_array[i] = basepredict_dt(test_X[i, :], dt)
    return test_Y_predicted_array

def training_dt(train_X, train_Y, depth, max_depth = 10):
    if  depth >= max_depth:
        rule = DecisionTree()
        rule.prediction_y = np.mean(train_Y)
        return rule

    rule = DecisionTree()
    rule_rss = np.inf
    feature_numb = np.shape(train_X)[1]

    for feature_no in range(feature_numb):
        feature_value = np.unique(train_X[:, feature_no])
        feature_value = np.sort(feature_value)
        feature_value = feature_value[1: -1]
        for val in feature_value:
            sec_datum = train_X[:, feature_no] > val
            sec_Y = train_Y[sec_datum]
            first_datum = ~sec_datum
            first_Y = train_Y[first_datum]
            node_rss = RSS(sec_Y, first_Y)
            if rule_rss > node_rss:
                rule_rss = node_rss
                rule.feature = feature_no
                rule.threshold_to_stop = val

    if rule.threshold_to_stop is None or rule.feature is None:
        rule.prediction_y = np.mean(train_Y)
        return rule

    sec_datum = train_X[:, rule.feature] > rule.threshold_to_stop
    sec_train_X = train_X[sec_datum, :]
    sec_train_Y = train_Y[sec_datum]
    first_datum = ~sec_datum
    first_train_X = train_X[first_datum, :]
    first_train_Y = train_Y[first_datum]
    depth = depth + 1

    rule.sec = training_dt(sec_train_X, sec_train_Y, depth, max_depth)
    rule.first = training_dt(first_train_X, first_train_Y, depth, max_depth)
    return rule
```