Part 1 - Exploring and processing the data

House sale prices for King County (USA), which includes Seattle, are included in the dataset utilized in this study. Homes sold between May 2014 and May 2015 are included.
The dataset can be found in this link:
https://www.kaggle.com/harlfoxem/housesalesprediction [1]. This dataset includes 21613 rows and 21 columns.

Before importing the data from the dataset, the libraries need to import. The libraries that were used are the following:

- import pandas as pd – This library manipulates and analyse the data.
- from sklearn.preprocessing import StandardScaler - Remove the mean and scale to unit variance to standardize characteristics.
- from sklearn.model_selection import train_test_split - Divide arrays or matrices into train and test subsets at random.
- import keras - Keras is a library for artificial neural networks.
- from keras.models import Sequential - A sequential model is a useful approach for a neural network because it connects the layers sequentially.
- from keras.layers import Dense, Dropout - The regular deeply linked neural network layer is the dense layer. At each step during training period, the Dropout layer sets input units to 0 at random with a rate frequency.
- from tensorflow.keras.optimizers import Adam – Adam is an optimization algorithm for Stochastic Gradient Descent (SGD) for training models.
- import numpy as np - NumPy is a library that is used to perform a range of mathematical operations on arrays.
- import matplotlib.pyplot as plt – This is a plotting library.
- import seaborn as sns – This library makes statistical graphics.

Then the dataset was imported and store in the variable “df” by using the following function:

```
df = pd.read_csv("kc_house_data.csv ")
```

To analyse the dataset a function was used to get the first 5 rows of the dataset (df.head(5).T). This function can be used to rapidly determine whether an item includes the correct data type. In addition, another function (df.info) was used to obtain additional dataset information such as the number of columns, column labels, column data types, and memory usage.

[5]:

	0	1	2	3	4
id	7129300520	6414100192	5631500400	2487200875	1954400510
date	20141013T000000	20141209T000000	20150225T000000	20141209T000000	20150218T000000
price	221900.0	538000.0	180000.0	604000.0	510000.0
bedrooms	3	3	2	4	3
bathrooms	1.0	2.25	1.0	3.0	2.0
sqft_living	1180	2570	770	1960	1680
sqft_lot	5650	7242	10000	5000	8080
floors	1.0	2.0	1.0	1.0	1.0
waterfront	0	0	0	0	0
view	0	0	0	0	0
condition	3	3	3	5	3
grade	7	7	6	7	8
sqft_above	1180	2170	770	1050	1680
sqft_basement	0	400	0	910	0
yr_built	1955	1951	1933	1965	1987
yr_renovated	0	1991	0	0	0
zipcode	98178	98125	98028	98136	98074
lat	47.5112	47.721	47.7379	47.5208	47.6168
long	-122.257	-122.319	-122.233	-122.393	-122.045
sqft_living15	1340	1690	2720	1360	1800
sqft_lot15	5650	7639	8062	5000	7503

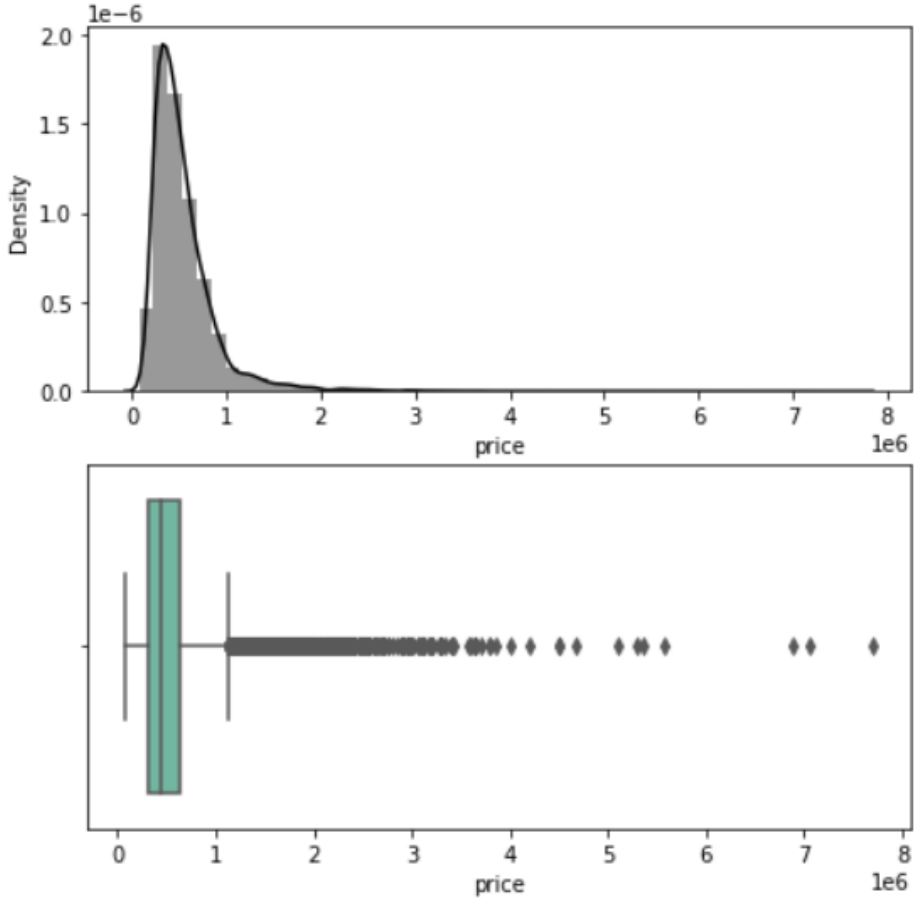
[6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   id             21613 non-null  int64  
 1   date           21613 non-null  object 
 2   price          21613 non-null  float64
 3   bedrooms       21613 non-null  int64  
 4   bathrooms      21613 non-null  float64
 5   sqft_living    21613 non-null  int64  
 6   sqft_lot       21613 non-null  int64  
 7   floors         21613 non-null  float64
 8   waterfront     21613 non-null  int64  
 9   view           21613 non-null  int64  
 10  condition      21613 non-null  int64  
 11  grade          21613 non-null  int64  
 12  sqft_above     21613 non-null  int64  
 13  sqft_basement  21613 non-null  int64  
 14  yr_built       21613 non-null  int64  
 15  yr_renovated   21613 non-null  int64  
 16  zipcode        21613 non-null  int64  
 17  lat            21613 non-null  float64
 18  long           21613 non-null  float64
 19  sqft_living15  21613 non-null  int64  
 20  sqft_lot15     21613 non-null  int64  
dtypes: float64(5), int64(15), object(1)
memory usage: 3.5+ MB
```

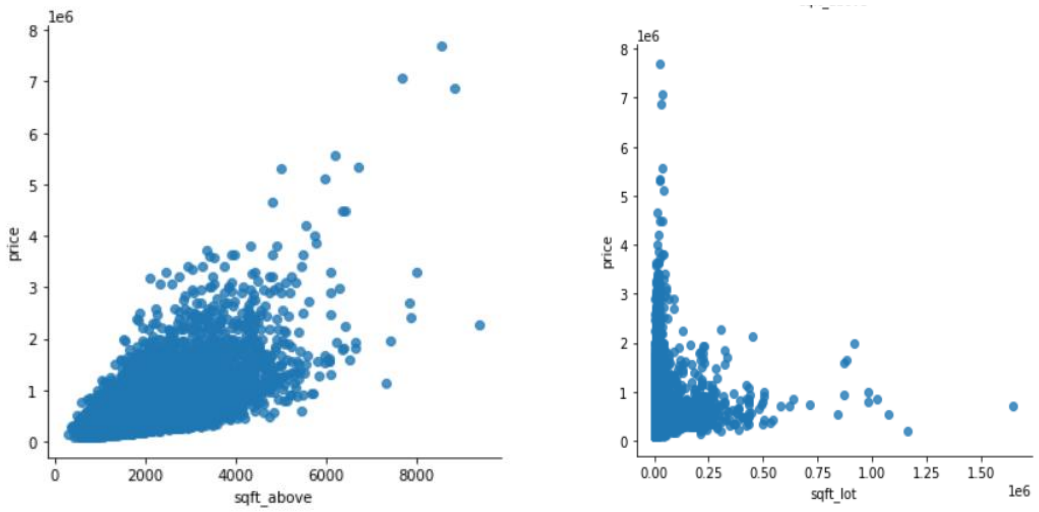
The dataset contains the following features:
Id: The ID of each house sold
Date: The Date when the house was sold
Price: Price of each house
Bedrooms: No. of bedrooms per House
Bathrooms: No. of bathrooms per House
Sqft_Living: Square footage of each house
Sqft_Lot: Square footage of the land space
Floors: No. of floors
Waterfront: This feature is about whether a House has a view to a waterfront
View: This feature displays of how good the view of the house was
Condition: How good is the condition of each house
Grade: This feature shows that each house is given a rating based on the King County grading system
Sqft_Above: The square footage of house except from basement
Sqft_Basement: The square footage of the basement
Yr_Built: The year that the house built
Yr_Renovated: The year when house was renovated
Zipcode: The zipcode of the houses
Lat: Latitude values
Long: Longitude values
Sqft_Living15: The square footage of the living room for the nearest 15 neighbors
Sqft_Lot15: The square footage of the land lots of the nearest 15 neighbors

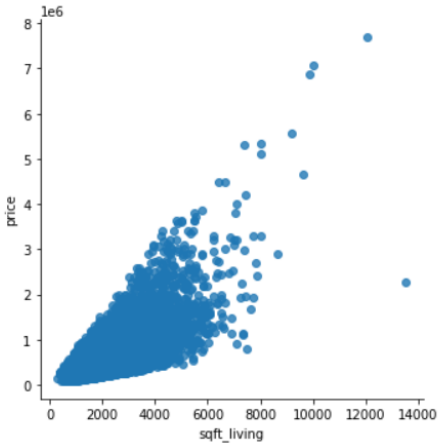
The features were then plotted to help comprehend the dataset better. The graphs show that the majority of the costs are between 0 and 1 million dollars, with a few

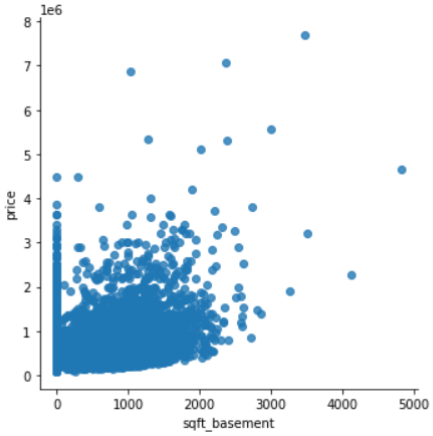
outliers near 8 million dollars.

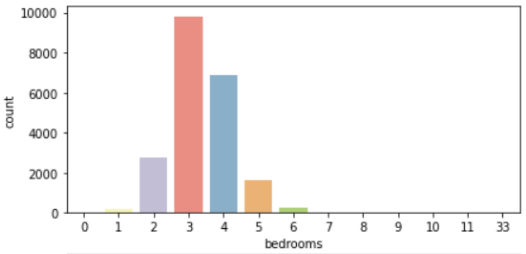


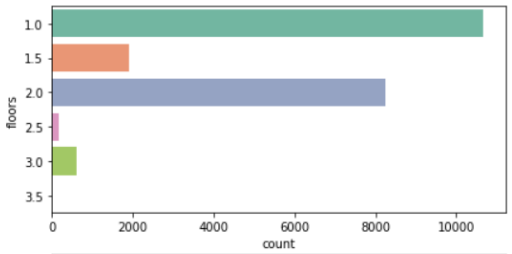
To better comprehend and visualise the dataset's features, a brief analysis of various feature distribution versus house price was plotted.

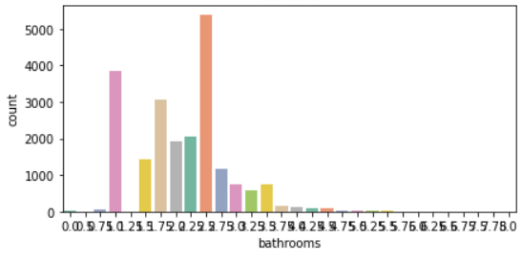


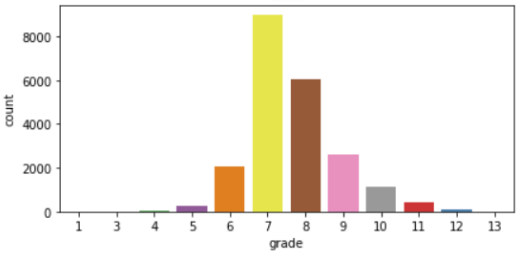








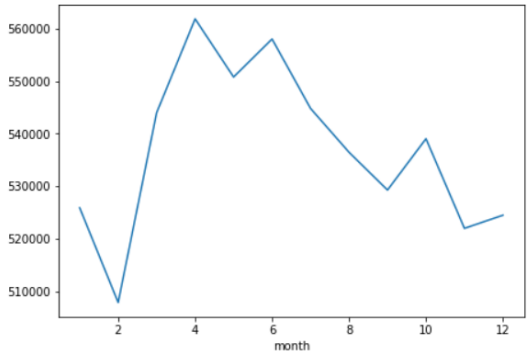


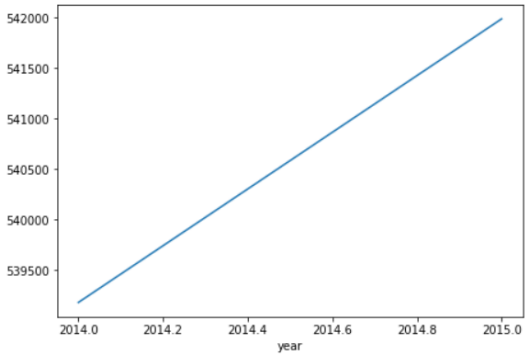



```
price            1.000000
sqft_living      0.702035
grade            0.667434
sqft_above       0.605567
sqft_living15    0.585379
bathrooms        0.525138
view             0.397293
sqft_basement    0.323816
bedrooms         0.308350
lat              0.307003
waterfront       0.266369
floors           0.256794
yr_renovated     0.126434
sqft_lot         0.089661
sqft_lot15       0.082447
yr_built         0.054012
condition        0.036362
long             0.021626
id              -0.016762
zipcode         -0.053203
Name: price, dtype: float64
```

A correlation table was also created to verify the variable price's association with others. The variables id and zip code have the least correlation with the price in this table, hence these two columns will be removed from the dataset.

The date column was then broken down into years and months to see how the house price has changed over time.





Part 2- Building and training the Neural Network

Before beginning to develop the neural network, unnecessary data such as the id and zip code were removed. The date field was also removed from the dataset because it was split into two columns: one for the year and one for the month. The data needed to be converted into arrays before being fed into the neural network, therefore a function called dataset ("dataset = df.values") was used to do that.

After that, the data must be separated into inputs X, which will be utilized to create predictions, and the target value y, which will be forecasted based on the features. The price of the houses will be the target variable for this project. Except from the price column, all of the columns are allocated to a variable called X. The y variable is used to represent the price column. The following is the code to assign the columns as X and y:

```
X=dataset[:,1:20]
y = dataset[:, 0]
```

Every value before the comma refers to the array's rows, and every value after the comma refers to the array's columns in this code. Since nothing was inserted before the comma in the code, the rows did not change and the X variable will take all of the rows from the dataset. The columns that are used are indicated by the values after the comma. Columns 1 to 20 were inserted as the features for the variable X in this case (19 columns). Column 0 was added to the y variable, which refers to the price column. To begin training the model, the data must be separated into two sets: training and test. The train test split library was used for this stage.

```
x_train, x_test, y_train, y_test = train_test_split(X,y, test_size=0.20)
```

This code shows how the data was divided into training and test data, with the tested data accounting for 20% of the total data. In conclusion, this phase has four variables:

- x_train (19 input features, 80% of the dataset)
- x_test (19 input features, 20% of the dataset)
- y_train (1 output, 80% of the dataset)
- y_test (1 output, 20% of the dataset)

The y_train and y_test data were also converted from a row to a column array:

```
y_train = y_train.reshape((-1, 1))

y_test = y_test.reshape((-1, 1))
```

The characteristics were then scaled to unit variance after being normalised by subtracting the mean. On the training data, the fit.transform function was used to scale them and learn the data's scaling parameters.

```
x_scaler = StandardScaler()

x_train = x_scaler.fit_transform(x_train)

y_scaler = StandardScaler()

y_train = y_scaler.fit_transform(y_train)
```

The epochs and batch size must be entered before the model can be created. The batch size refers to how many samples are processed until the model is finished. The number of epochs is the number of times the training test has been run. To create a neural network, the model must be divided into layers. The neural network will be separated into three layers for this project (input layer, hidden layer, output layer). The model = sequential() function comes next, and it specifies the model layer by layer (sequentially). This model's input layer will be a dense layer of sixty-four neurons. A dense layer means that all the neurons of this layer will be connected with each neuron of its preceding layer. In the case of the input layer, the input_shape was implemented to say that this layer will be the first. Also, this layer has to have the same shape as the training data so the number 19 were inserted which indicates the number of columns of the training data. The ReLU activation function was employed to determine whether or not a neuron should be activated. If the input is positive, the rectified linear activation function (ReLU) will output it directly, else it will output zero.

Then, a dropout layer was used to drop randomly 20% of nodes. This dropout layer was added to improve the model's accuracy and prevent it from overfitting. The middle/hidden layer goes through the same procedure. The last layer is a dense layer with one neuron that employs a linear activation function that does not affect the sum of the inputs and simply returns the value.

A loss function and a Keras optimizer were used to compile the model. The mean squared error (MSE) loss function is utilised, which determines the average of the squared difference between the expected response and the actual true value for each observation. The Adam optimizer is an optimization algorithm for stochastic gradient descent that is used for training deep learning models.

The model.fit() function was used to train the model. The train data, as well as the batch size and epochs, were inserted into this function. Furthermore, two other variables were added to this function (verbose, validation_split). The output of the Neural network can be seen in a verbose mode while it is training. The validation split Keras function allows the user to quickly separate the training dataset into train and validation. For instance, if the validation_split= 0.1 means that the Keras will use the last 10% of the data before starting training for validation.

```
batch_size = 128
epochs = 400

model = Sequential()
model.add(Dense(64, activation= "relu",
                input_shape=(19,)))
model.add(Dropout(0.2))
model.add(Dense(64, activation= "relu"))
model.add(Dropout(0.2))

model.add(Dense(1, activation= "linear"))

model.compile(loss= "mse", optimizer= "Adam")

hist=model.fit(x_train, y_train, batch_size=batch_size,
          epochs=epochs, verbose =1, validation_split=(0.1))
```

Once the model has been trained, it can be assessed on the test set to determine the loss:

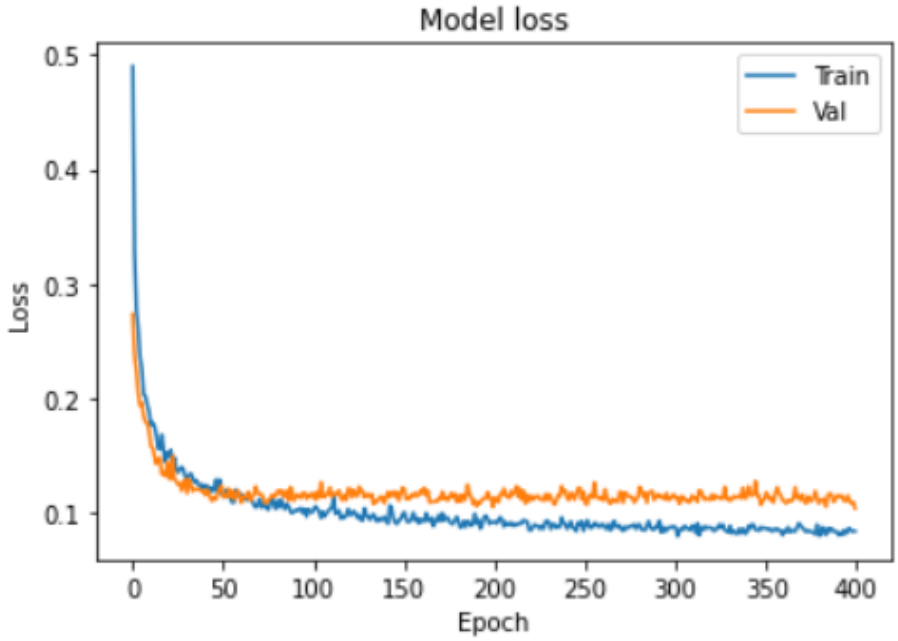
```
x_test = x_scaler.transform(x_test)
y_test = y_scaler.transform(y_test)

model.evaluate(x_test, y_test)
```

To obtain the median house price of a new sample, the following procedure was implemented:

```
new_sample = [[3,1.00,1180,5650,1.0,0,0,3,7,1180,0,1955,0,47.5112,-122.257,1340,5650,10,2014]]
new_sample_normalised = x_scaler.transform(new_sample)
raw_output = model.predict(new_sample_normalised)
output = y_scaler.inverse_transform(raw_output)
print(output)
```

This is the house price of the new sample: `[[270223.16]]`

The training loss per epoch was plotted to see if the model is overfitting. Since both lines fall close at the same time, there is no overfitting in this situation.



This section's final step is to summarise the model.

model.summary()

The strength of the relationship between the model and the dependent variable is shown in the model summary table below.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                1280
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                4160
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 5,505
Trainable params: 5,505
Non-trainable params: 0
_________________________________________________________________
```

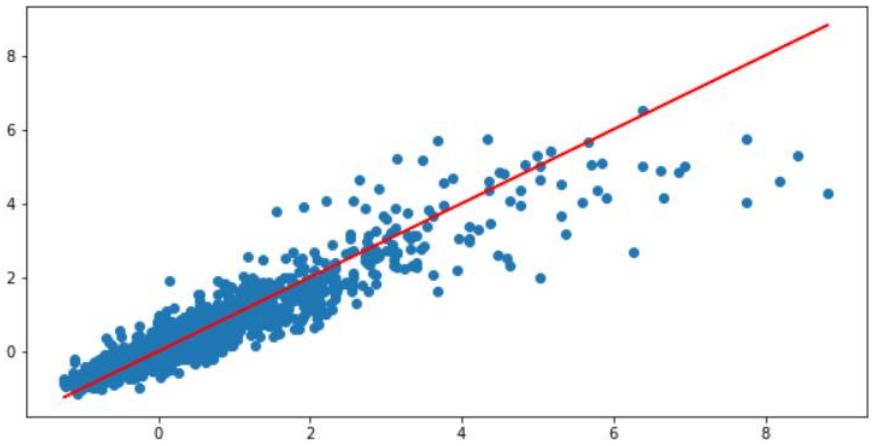
Part 3 – Evaluation on test data

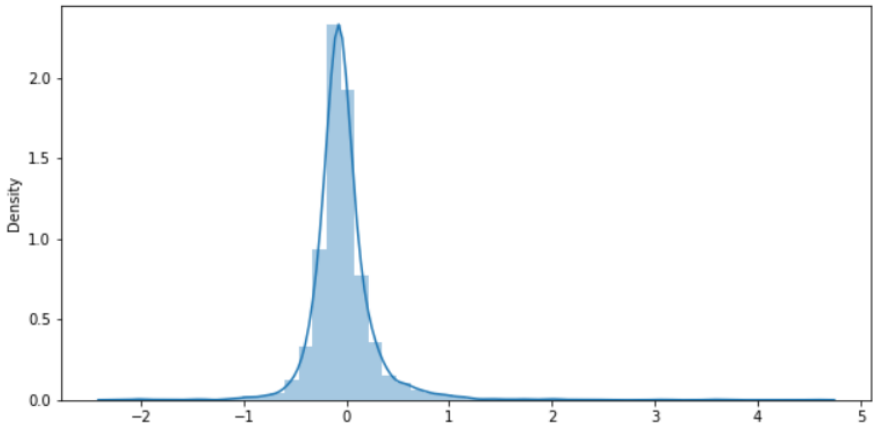
For this part, three different regression evaluation metrics were used. These are:

- **Mean Squared Error** (MSE) – The average error of the absolute value of the model.
- **Mean Absolute Error** (MAE) - The average error of the model.
- **Root Mean Squared Error** (RMSE) - The square root of the absolute value of the model's average error.

In this part, the model will be fed with the test set to generate a list of predictions. Then, this list will be compared with the accurate numbers. Different measures were used to compare the predictions such as MAE, MSE, RMSE and Variance Regression Score.

Then a graph was made to display the model predictions versus the perfect fit to see how accurate the model is. The red line in the first graph represents the perfect prediction. Some outliers indicate the expensive houses. The second graph visualizes residuals which is the difference between the predicted y values and the y observed values.





Part 4 - The server-client communication

A UDP socket was chosen to link the client to the server for this phase of the project. UDP is an unreliable and unpredictable transmission protocol. The delivery of every byte is not guaranteed by UDP. The transmission rate is calculated by the application as well. Importing the libraries was the initial step in writing the code for the server-client communication. The first library which inserted into the code was the socket library. This library implements sockets, which can be used to communicate with the Internet. Then, the sys module was used. The sys library in python contains several methods and variables for manipulating various aspects of the python runtime environment.

After this process, two sockets named SOCK_DGRAM and AF_INET were added to the server to begin socket programming. The SOCK_DGRAM socket provides connectionless messages with a maximum length of a certain number of characters. The AF_INET socket specifies the sort of address that a socket can communicate with. IPv4 addresses are used in the majority of cases. After creating the server socket the s.bind(Host, Port) function is used to bind the socket to the localhost and port.

Two functions (sendto() and recvfrom()) are also used to communicate between the server and the client. The client first sends a request to the server using the sendto() function. Then, the server reads this request with the help of the recvfrom() function that the client sent. Depending on the value placed inside the brackets, recvfrom() will read a certain number of bytes. Since the client's request contains a large number of bytes, the number in the brackets will be increased to 2048 for this situation. The server will next process this message to determine the predicted value, which in this case is the house price. The server then sends a message back with the sendto() function and the client receives it by using the recvfrom() function. Finally, the close() method closes the socket associated with the server and releases the socket's resources.

The UDP sockets receive a UDP datagram at the time and the UDP socket servers do not listen for connections in the same way that TCP socket servers do. The sending rate of the UDP protocol is controlled by the clients and the server instead of TCP where there is an operating system for the sending rate. One of the drawbacks of a UDP socket is that a network can easily flood with packets.

Both of the server and client codes are included below. The server code is on the left, and the client code is on the right.

```python
HOST = ''
PORT = 8888

# Datagram (udp) socket

try:
    s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    print("Socket created")
except socket.error as msg:
    print("Failed to create socket. Error Code :" + str(msg[0]) + " Message " + msg[1])
    sys.exit()

# Bind socket to local host and port
try:
    s.bind((HOST,PORT))
except socket.error as msg:
    print('Bind failed. Error Code :' + str(msg[0]) + "Message" + msg[1])
    sys.exit()

print("Socket bind complete")

# now keep talking with the client
while 1:
    # receive data from client (data, addr)
    d = s.recvfrom(2048)
    data = d[0].decode()
    addr = d[1]

    if not data:
        break
    reply = "This is the price of this house = 261020.12"

    s.sendto(reply.encode(), addr)
    print("Message[" + addr[0] + ":" + str(addr[1]) + "] - " + data.strip())

s.close()
```

```python
HOST = ''
PORT = 8888

# Datagram (udp) socket

try:
    s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    print("Hello, I'm the Oracle. How can I help you today?")
except socket.error as msg:
    print("Failed to create socket")
    sys.exit()

host = 'localhost'
port =8888

while (1):
    msg=input("Enter message to send").encode()

    try:
            # Set the whole string
            s.sendto(msg, (host, port))

            # receive data from client (data, addr)
            d= s.recvfrom(2048)
            reply = d[0].decode()
            addr = d[1]

            print("Server reply : " + reply)

    except socket.error as msg:
            print("Error Code: " + str(msg[0]) + " Message " + msg[1])
            sys.exit()
```

When the server connects to the client, it will deliver the message “Hello, I am the Oracle how can I help you?.” Then the client will reply by asking what is the price of a house by giving specific details about that. These details will be related to the features of the dataset such as the number of floors, grade, latitude. After that, the server will respond with the house's price. An example of this communication is the image below:

```
Hello, I'm the Oracle. How can I help you today?
Enter message to send I would like the median value of this house with these details [3,1.00,1180,5650,1.0,0,0,3,7,1180,0,1955,0,47.5112,-122.257,1340,
5650,10,2014]
Server reply : This is the price of this house = 261020.12
Enter message to send
```

References

1.Kaggle. House Sales in King County, USA. 2021 [online] Available from: https://www.kaggle.com/datasets/harlfoxem/housesalesprediction [Accessed 25 March 2022].

Appendices

The actual code:

Part 1 - Exploring and processing the data:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import  Adam
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
df = pd.read_csv("kc_house_data.csv ")
print(df)
df.head(5).T
df.info()
#Visualizing house prices
fig = plt.figure(figsize=(7,7))
fig.add_subplot(2,1,1)
sns.distplot(df['price'],color="k")
fig.add_subplot(2,1,2)
sns.boxplot(df['price'], palette="Set2")
#Visualizing square footage of (home,lot,above and basement)
fig = plt.figure(figsize=(16,7))
sns.lmplot(x='sqft_above',y='price',data=df,fit_reg=False)
sns.lmplot(x='sqft_lot',y='price',data=df,fit_reg=False)
```

```
sns.lmplot(x='sqft_living',y='price',data=df,fit_reg=False)
sns.lmplot(x='sqft_basement',y='price',data=df,fit_reg=False)
#Visualizing bedrooms,bathrooms,floors,grade
fig = plt.figure(figsize=(15,7))
fig.add_subplot(2,2,1)
sns.countplot(df['bedrooms'],data=df,palette="Set3")
fig.add_subplot(2,2,2)
sns.countplot(y='floors',data=df,palette="Set2")
fig.add_subplot(2,2,3)
sns.countplot(df['bathrooms'],palette="Set2")
fig.add_subplot(2,2,4)
sns.countplot(df['grade'],palette="Set1")
price_correlation = df.corr()['price'].sort_values(ascending=False)
print(price_correlation)
#Data visualization house price vs months and years
df['date'] = pd.to_datetime(df['date'])
df['month'] = df['date'].apply(lambda date:date.month)
df['year'] = df['date'].apply(lambda date:date.year)
fig = plt.figure(figsize=(16,5))
fig.add_subplot(1,2,1)
df.groupby('month').mean()['price'].plot()
fig.add_subplot(1,2,2)
df.groupby('year').mean()['price'].plot()
```

Part 2- Building and training the Neural Network:

```
df.drop(['id', 'date','zipcode'], axis=1, inplace=True)
dataset = df.values
```

```
X = dataset[:, 1:20]

y = dataset[:, 0]

x_train, x_test, y_train, y_test = train_test_split(X,y, test_size=0.20)

y_train = y_train.reshape((-1, 1))

y_test = y_test.reshape((-1, 1))

#Standardization scaler

x_scaler = StandardScaler()

x_train = x_scaler.fit_transform(x_train)

y_scaler = StandardScaler()

y_train = y_scaler.fit_transform(y_train)

#Train and create the model

batch_size = 128

epochs = 400

model = Sequential()

model.add(Dense(64, activation= "relu",input_shape=(19,)))

model.add(Dropout(0.2))

model.add(Dense(64, activation= "relu"))

model.add(Dropout(0.2))

model.add(Dense(1, activation= "linear"))

model.compile(loss= "mse", optimizer= "Adam")

hist=model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose =1,
validation_split=(0.1))

x_test = x_scaler.transform(x_test)

y_test = y_scaler.transform(y_test)

model.evaluate(x_test, y_test)

new_sample = [[3,1.00,1180,5650,1.0,0,0,3,7,1180,0,1955,0,47.5112,-
122.257,1340,5650,10,2014]]

new_sample_normalised = x_scaler.transform(new_sample)
```

```
raw_output = model.predict(new_sample_normalised)
output = y_scaler.inverse_transform(raw_output)
print(output)
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='upper right')
plt.show()
model.summary()
```

Part 3 – Evaluation on test data:

```
y_pred = model.predict(x_test)
from sklearn import metrics
print('MAE:', metrics.mean_absolute_error(y_test, y_pred))
print('MSE:', metrics.mean_squared_error(y_test, y_pred))
print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print('VarScore:',metrics.explained_variance_score(y_test,y_pred))
# Visualizing the predictions
fig = plt.figure(figsize=(10,5))
plt.scatter(y_test,y_pred)
# Perfect predictions
plt.plot(y_test,y_test,'r')
fig = plt.figure(figsize=(10,5))
residuals = (y_test- y_pred)
sns.distplot(residuals)
```

Part 4 - The server-client communication:

```
import socket
import sys
# Server code
HOST = ''
PORT = 8888
# Datagram (udp) socket
try:
   s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
   print("Socket created")
except socket.error as msg:
   print("Failed to create socket. Error Code :" + str(msg[0]) + " Message " + msg[1])
   sys.exit()
# Bind socket to local host and port
try:
   s.bind((HOST,PORT))
except socket.error as msg:
   print('Bind failed. Error Code :' + str(msg[0]) + "Message" + msg[1])
   sys.exit()
print("Socket bind complete")
# now keep talking with the client
while 1:
   # receive data from client (data, addr)
   d = s.recvfrom(2048)
   data = d[0].decode()
   addr = d[1]
```

```
    if not data:
        break
    reply = "This is the price of this house = 261020.12"
    s.sendto(reply.encode(), addr)
    print("Message[" + addr[0] + ":" + str(addr[1]) + "] - " + data.strip())
s.close()

# Client code
import socket
import sys
HOST = ''
PORT = 8888
# Datagram (udp) socket
try:
    s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    print("Hello, I am the Oracle. How can I help you today?")
except socket.error as msg:
    print("Failed to create socket")
    sys.exit()
host = 'localhost'
port =8888
while (1):
    msg=input("Enter message to send").encode()
    try:
        # Set the whole string
        s.sendto(msg, (host, port))
        # receive data from client (data, addr)
```

```
        d= s.recvfrom(2048)
        reply = d[0].decode()
        addr = d[1]
        print("Server reply : " + reply)
    except socket.error as msg:
        print("Error Code: " + str(msg[0]) + " Message " + msg[1])
        sys.exit()
```