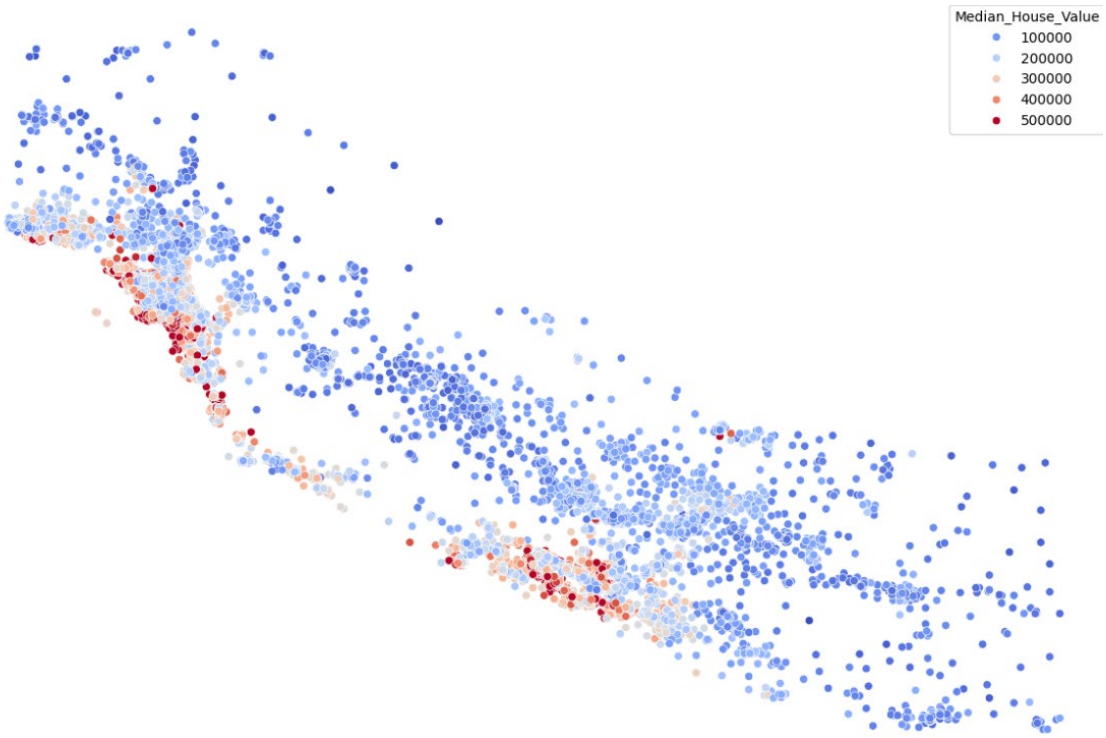
INTRODUCTION TO MACHINE LEARNING

ASSIGNMENT 1



RAHMA ABDULHAMID	7358
MARIAM MOHAMED SAMY	7695
RANA MOHAMED ABO ELAMAYEM	7697

CODE WALK-THROUGH:

```
import numpy as np

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.metrics import accuracy_score, classification_report, confusion_matrix,
ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split
from scipy.stats import mode

from imblearn.under_sampling import RandomUnderSampler
from sklearn.neighbors import KNeighborsClassifier
```

import numpy as np imports the NumPy library, which is a tool for scientific computing with Python. NumPy provides a variety of data structures and functions for working with arrays, and large amounts of data.

import pandas as pd imports the Pandas library, which is a tool for data analysis and visualization with Python. Pandas provides a variety of data structures and functions for working with tabular data, such as DataFrames and Series. DataFrames are two-dimensional data structures that are similar to spreadsheets, and Series are one-dimensional data structures that are similar to lists.

import matplotlib.pyplot as plt imports the Matplotlib library, which is a tool for data visualization that provides a variety of functions for creating plots, such as line charts, bar charts, and histograms.

import seaborn as sns imports the Seaborn library, which is a high-level interface for Matplotlib that provides a variety of pre-defined plot styles and color palettes for visually appealing plots.

From sklearn.metrics:
1. accuracy_score: Calculates the accuracy of a model's predictions, which is the proportion of correct predictions made by the model.
2. classification_report: Generates a classification report, which includes metrics such as accuracy, precision, recall, and F1 score for each class in the dataset.
3. confusion_matrix: Creates a confusion matrix, which is a table that shows the number of correct and incorrect predictions made by the model for each class.
4. ConfusionMatrixDisplay: Visualizes a confusion matrix using a heatmap.

From sklearn.model_selection
train_test_split: Splits a dataset into training and test sets. This is important to do to avoid overfitting, which is when a model learns the training data too well and is unable to generalize to new data.

From scipy.stats
mode: Calculates the mode of a distribution, which is the most frequent value.

From imblearn.under_sampling
RandomUnderSampler: Randomly removes samples from the majority class to create a more balanced dataset. Used for imbalanced classification problems, for classes with unequal number of samples.

From sklearn.neighbors
KNeighborsClassifier: A classification algorithm that predicts the class of a new data point by finding the K most similar data points in the training set and looking at the majority class of those K points.

READING DATA FROM FILE:

```
data=pd.read_csv("D:\\College\\College\\TERM 7\\Machine
Learning\\Assignments_LABS\\Files\\Assignment 1\\magic04.data")
```

The pd.read_csv() function in Python is used to read a comma-separated values (CSV) file into a Pandas DataFrame. A DataFrame is a two-dimensional data structure with labeled axes, which makes it easy to manipulate and analyze data.

DECLARING FEATURE AND TARGET VECTORS:

```
X = data.drop(['c'], axis=1)
y = data['c']
```

In the data file, the first 10 comma separated values in each line represent the features while the 11th represents the class. Class is either 'g' for gamma or 'h' for hadrons.

In the code you provided, the feature vector is represented by the variable X. The feature vector will contain all of the columns in the DataFrame except for the column while the target variable represented by the variable y is created by selecting the column c from the DataFrame data.

BALANCING THE DATASETS (UNDERSAMPLING):

```
# creating dataframe from X
data = pd.DataFrame(X)
# converting NDArray to series
target = pd.Series(y)

# sampling_strategy for majority class
# majority -> resamples only the majority class
undersampler = RandomUnderSampler(sampling_strategy="majority")
# resampled data and target
undersampled_data, undersampled_target = undersampler.fit_resample(data, target)
# class distribution
undersampled_target.value_counts()
```

Here we created a balanced datasets, by undersampling the majority class 'gamma' which had 12332 events to fit with 'hadrons' which only had 6688.

This part of the code uses the RandomUnderSampler class from imblearn to undersample the majority class.

1. `data = pd.DataFrame(X)`
   This line creates a DataFrame from the feature vector X. This is because the RandomUnderSampler class only accepts DataFrames as input.

2. `target = pd.Series(y)`
   This line converts the target variable y to a Series. This is also necessary because the RandomUnderSampler class only accepts Series as input.

3. `undersampler = RandomUnderSampler(sampling_strategy="majority")`
   The sampling strategy sampling_strategy="majority" tells the RandomUnderSampler class to only resample the majority class.

4. `undersampled_data, undersampled_target = undersampler.fit_resample(data, target)`
   The fit_resample() method on the RandomUnderSampler object takes the DataFrame data and the Series target as input and returns a resampled DataFrame and a resampled Series.

5. `undersampled_target.value_counts()`
   This line prints the class distribution of the resampled target Series undersampled_target. This is useful for checking to make sure that the undersampling process was successful in balancing the class distribution.

```
c
g    6688
h    6688
Name: count, dtype: int64
```

As shown in figure, class g was successfully under-sampled to match h, having only 6688 events each.

SPLITTING DATASETS INTO TRAINING, VALIDATION AND TEST SETS:

```
train_ratio = 0.7
validation_ratio = 0.15
test_ratio = 0.15

# train is now 70% of the entire data set
# test_size=1 - train_ratio
x_train, x_rest, y_train, y_rest = train_test_split(undersampled_data,
undersampled_target, test_size=0.3)

# test is now 15% of the initial data set
# validation is now 15% of the initial data set
#test_size=test_ratio/(test_ratio + validation_ratio)
x_val, x_test, y_val, y_test = train_test_split(x_rest, y_rest, test_size=0.5)
```

The code splits the undersampled data into three sets: training, validation, and test. The training set (70% of dataset) is used to train the model, the validation set (15%) is used to evaluate the model, and the test set (15%) is used to evaluate the final performance of the model.

```
1. x_train, x_rest, y_train, y_rest = train_test_split(undersampled_data,
   undersampled_target, test_size=0.3)
```

This line splits the undersampled data into training and rest sets using the train_test_split() where the test_size parameter is set to 1 - train_ratio, which ensures that the training set is 70% of the entire data set.

```
2. x_val, x_test, y_val, y_test = train_test_split(x_rest, y_rest, test_size=0.5)
```

The line splits the validation and test sets from the rest set. The train_test_split() function is used to split the data into two sets, with a specified test size. The test size is set to 0.5, which means that 50% of the rest set will be used for the test set and the other 50% will be used for the validation set.

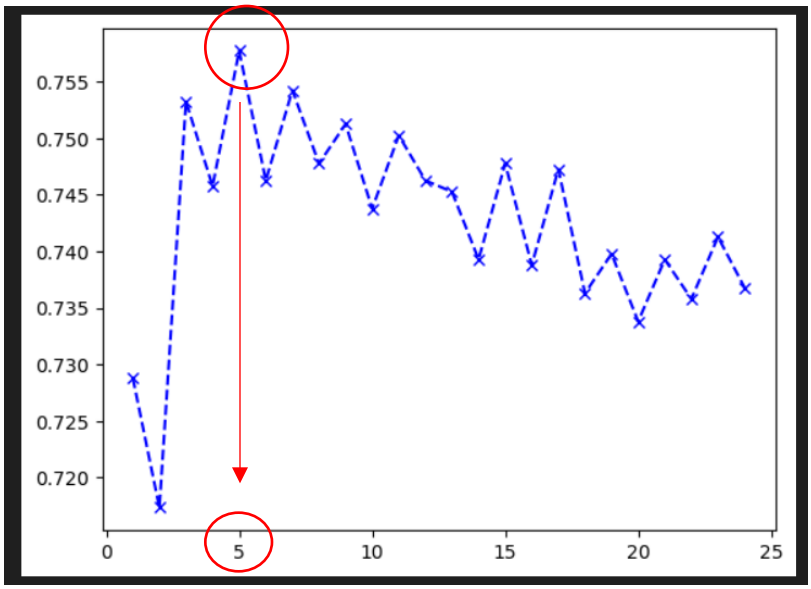
CREATING THE MODEL AND TRAININIG IT:

In this part, we use the validation set to find the K value for best results, and for the chosen K value, we test our model.

```
accuracy=[]
for i in range(1,25):
    model=KNeighborsClassifier(n_neighbors=i)
    model.fit(x_train, y_train)
    y_pred = model.predict(x_val)
    acc = accuracy_score(y_val,y_pred)
    print("Accuracy: ", acc)
    accuracy.append(acc)

plt.plot(range(1,25), accuracy, color='blue', marker='x', linestyle='dashed')
```

1. `accuracy=[]`
   We create an array to store the accuracy value for each k we try.
2. `For i in range(1,25):`
   Symbolizes the different values of k = 1→25 we are trying.
3. `model=KNeighborsClassifier(n_neighbors=i)`
   KNeighborsClassifier function from sklearn.neighbors library finds k-nearest neighboring points.
4. `model.fit(x_train, y_train)`
   .fit() from sklearn.model_selection is used store the dataset in an efficient Data Structure for searching to train the model on the training data. The model learns the patterns in the training data and uses those patterns to make predictions on new data.
5. `y_pred = model.predict(x_val)`
   The .predict() function classifies a data point from the validation test by finding the 'k' nearest neighbours from the Train Data and classify the query point based on the majority vote.
6. `acc = accuracy_score(y_val,y_pred)`
   The accuracy_score() function compares the predicted value and the actual value of the new data point




```
Accuracy:  0.7288135593220338
Accuracy:  0.7173479561316052
Accuracy:  0.7532402791625125
Accuracy:  0.7457627118644068
Accuracy:  0.7577268195413759
Accuracy:  0.7462612163509471
Accuracy:  0.7542372881355932
Accuracy:  0.7477567298105683
Accuracy:  0.751246261216351
Accuracy:  0.7437686939182453
Accuracy:  0.7502492522432702
Accuracy:  0.7462612163509471
Accuracy:  0.7452642073778664
Accuracy:  0.7392821535393819
Accuracy:  0.7477567298105683
Accuracy:  0.7387836490528414
Accuracy:  0.747258225324028
Accuracy:  0.7362911266201396
Accuracy:  0.7397806580259222
Accuracy:  0.7337986041874377
Accuracy:  0.7392821535393819
Accuracy:  0.7357926221335992
Accuracy:  0.7412761714855434
Accuracy:  0.7367896311066799
```

From the plot of accuracy against different values of k, we find that the model performed best at K=5. Consequently, we will use K=5 to test our set. (at approximately 76%)

BY TESTING THE TEST SET AT K=5:

(using KNeighborsClassifier, model.fit(), model.predit() functions like explained above):

```
    testDataAccuracy = accuracy_score(y_test,yPred)
    print("Accuracy: ",testDataAccuracy )
```

```
Accuracy:  0.7797708021923269
```

Accuracy is at approximately 78%

WHAT IS A CLASSIFICATION REPORT? It is a report that showcases the performance of a machine learning model on a test set. The report includes the following metrics:

- Precision: The fraction of positive predictions that were correct.
  precision = true_positives / (true_positives + false_positives)
- Recall: The fraction of positive examples that were correctly identified.
  recall = true_positives / (true_positives + false_negatives)
- F1-score: A harmonic mean of precision and recall.
  2 * (precision * recall) / (precision + recall)
- Support: The number of examples in each class.

	precision	recall	f1-score	support
g	0.75	0.83	0.79	999
h	0.81	0.73	0.77	1008
accuracy			0.78	2007
macro avg	0.78	0.78	0.78	2007
weighted avg	0.78	0.78	0.78	2007

CONCLUSIONS FROM CLASSIFICATION REPORT:

- The report shows that the model has an overall accuracy of 78% (model correctly predicted the class label for 78% of the test examples.)
- The F1-score for the model in this case is 0.78, which is a good score.

Macro average:

- The macro average calculates the average performance metrics (precision, recall, F1-score) for all classes, without considering class imbalance.
- The macro average precision, recall, and F1-score are all 0.78.

Weighted average:

- The weighted average calculates the average performance metrics, taking into account the number of samples in each class.
- The weighted average precision, recall, and F1-score are all 0.78.

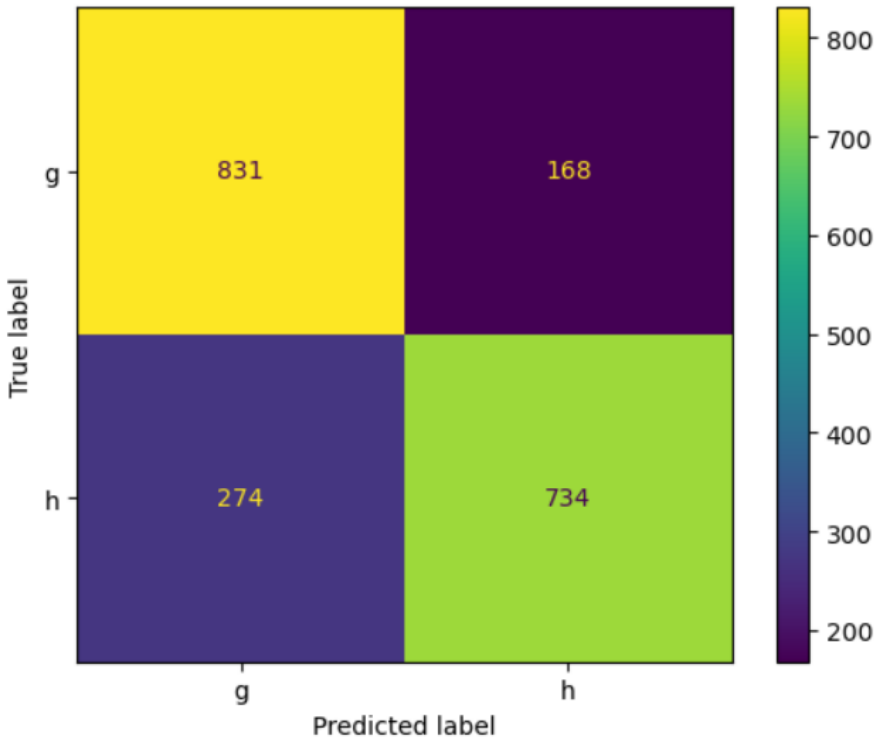
CONFUSION MATRIX DISPLAY:

A confusion matrix is a table that shows the number of correct and incorrect predictions made by a machine learning model for each class.

```
ConfusionMatrixDisplay(confusion_matrix(y_test, yPred),
display_labels=model.classes_).plot()
plt.show()
```

The confusion_matrix() function is used to calculate the confusion matrix for a given model and test set. The ConfusionMatrixDisplay() class takes the output of the confusion_matrix() function as an input.

The display_labels parameter of the ConfusionMatrixDisplay() class is used to specify the labels for the rows and columns of the confusion matrix. In this case, the model.classes_ attribute is used to get the class labels from the model.



From the above figure, we can deduce the number of correct and incorrect predictions made by the model for each class. For predicted label 'g', 831 predictions were correct, while 274 were incorrect.

As for class 'h', 734 predictions were correct, while 168 were incorrect (predicted as g)

Note: The total number of samples = 831+274+168+734 = 2007 (as previously mentioned before)

PART TWO:

```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression , Lasso,LassoCV ,Ridge
from sklearn.preprocessing import StandardScaler
from sklearn import metrics
from sklearn.model_selection import GridSearchCV
```

1. **sklearn.metrics**: provides functions for evaluating the performance of machine learning models.
2. **sklearn.model_selection**: provides functions for splitting data into training and test sets, and for performing cross-validation.
3. **sklearn.linear_model**: provides functions for implementing linear regression, lasso and Ridge models.
4. **sklearn.preprocessing**: provides functions for preprocessing data. The function StandardScaler() is used to standardize the data, which involves scaling the features to have a mean of zero and a standard deviation of one.
5. **sklearn**: contains the main functionality of scikit-learn, including the metrics, model_selection, and linear_model submodules.
6. **gridsearchcv**: provides a function for performing grid search on machine learning models. The function GridSearchCV() is used to search for the best hyperparameters for a given model.

DRAWING HISTOGRAM OF DATA: Given the data set has a large number of data points (20,640), a histogram size of figsize=(15,10) provides clear and readable display of data, while also allowing for some flexibility in customizing the appearance of the histogram.

PREPROCESSING OF DATA:

```
#Preprocessing
houses['Tot_Rooms']=np.log(houses['Tot_Rooms']+1)
houses['Tot_Bedrooms']=np.log(houses['Tot_Bedrooms']+1)
houses['Population']=np.log(houses['Population']+1)
houses['Households']=np.log(houses['Households']+1)
houses.hist(figsize=(15,10))
```

Preprocessing is needed to make the data more suitable for machine learning algorithms. In our case, the log transformation is used to:

1. Normalize the data, which can make it easier for the algorithms to learn the relationships between the variables.
2. Reduce skewness, which can make the model more robust to outliers.

Why we use log function:

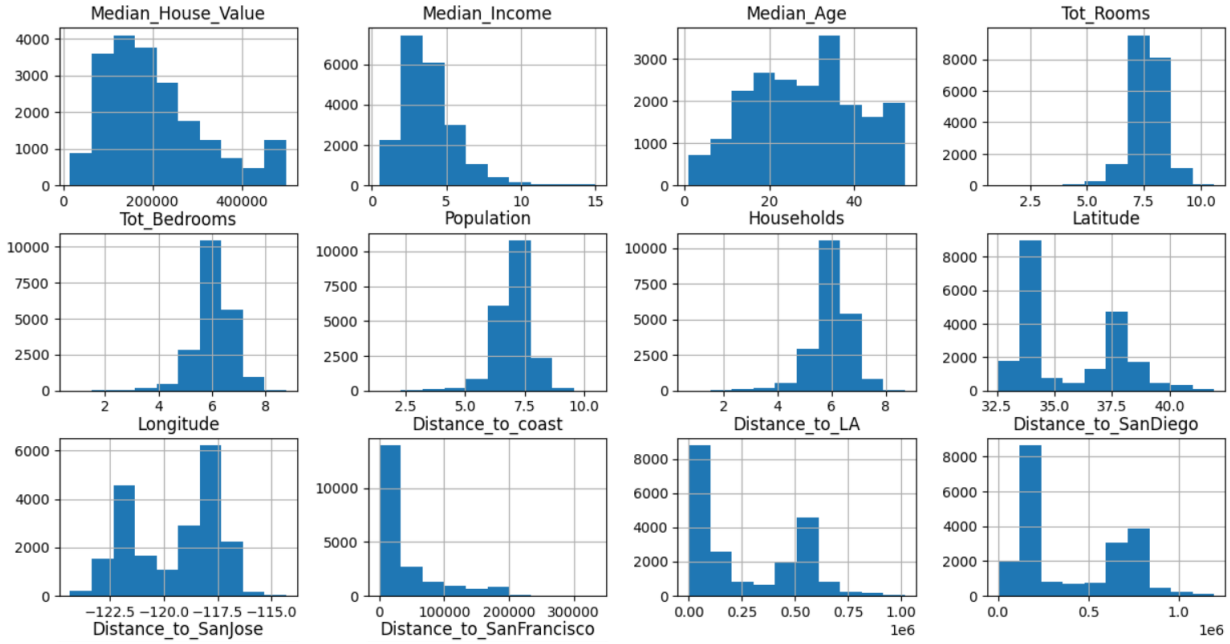
```
houses['Tot_Rooms']=np.log(houses['Tot_Rooms']+1)
```

In our context, the log transformation is used to make the data more suitable for the linear regression, lasso regression, and ridge regression algorithms.

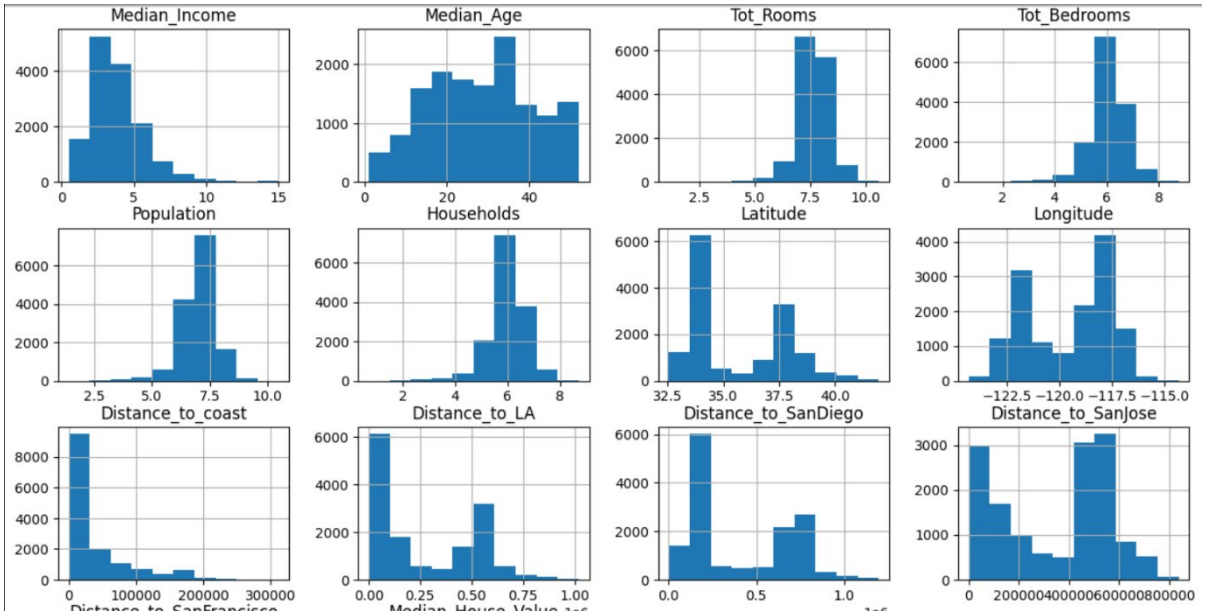
Since these algorithms are all based on the assumption that the relationship between the variables is linear, log transforming the data makes the relationship between the variables more linear, which can improve the performance of these algorithms.

NOTE: we add +1 to 'Tot_Rooms' to avoid taking the logarithm of zero in case Tot_Rooms or any value was read to be zero as logarithm of zero is undefined. Additionally, the Tot_Bedrooms and Population columns are both skewed to the right. This means that there are a few data points with very high values, and many data points with lower values. Adding 1 to the content of the logarithm helps to reduce the skewness of the data and makes it more interpretable.

DISPLAYED BELOW ARE PREPROCESSED NORMALISED VALUES OF DATA:



VS. After distributing data into training set (taking up 70% of total data set), validation(15%) and test sets(15%):

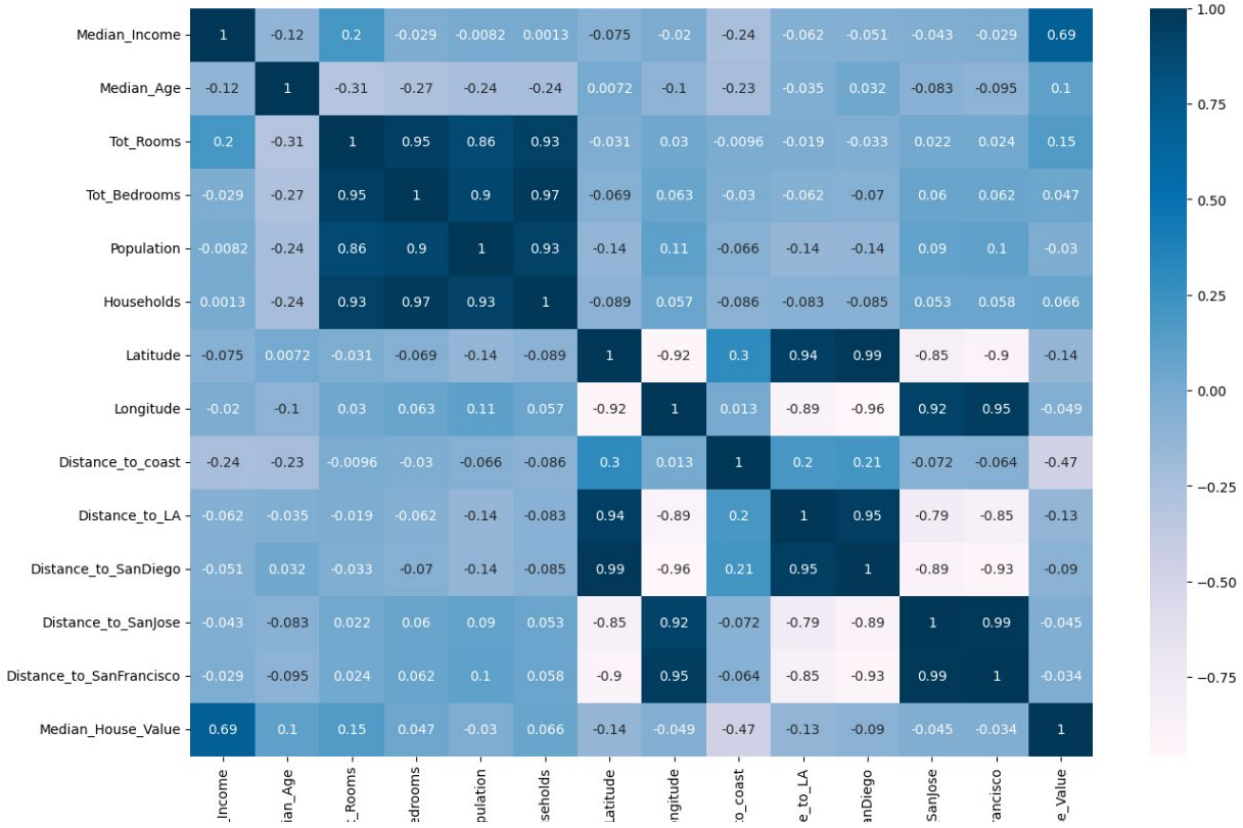


```
#Heat map of training set
plt.figure(figsize=(15,10))
sns.heatmap(train_data.corr(),annot=True,cmap="PuBu")
```

1. plt.figure(figsize=(15,10)): This line creates a figure with the specified dimensions.
2. sns.heatmap(train_data.corr(),annot=True,cmap="PuBu"): This line creates a heatmap of the correlation matrix of the training data. The annot=True argument specifies that the correlation coefficients should be displayed in the heatmap cells. The cmap="PuBu" argument specifies that the PuBu colormap should be used to color the heatmap cells.

WHY WE USE HEATMAPS?

1. Identifying correlated features and allows us to remove redundant features or to inform the choice of machine learning model.
2. Heatmaps can also be used to identify outliers in the data to be removed or handled appropriately before training a machine learning model.
3. Heatmaps can help to understand the distribution of the data. This information can be used to choose the appropriate machine learning model and to set the hyperparameters of the model.



The heatmap will show the correlation between each pair of features in the training data. The darker the color of a cell in the heatmap, the higher the correlation between the corresponding features.

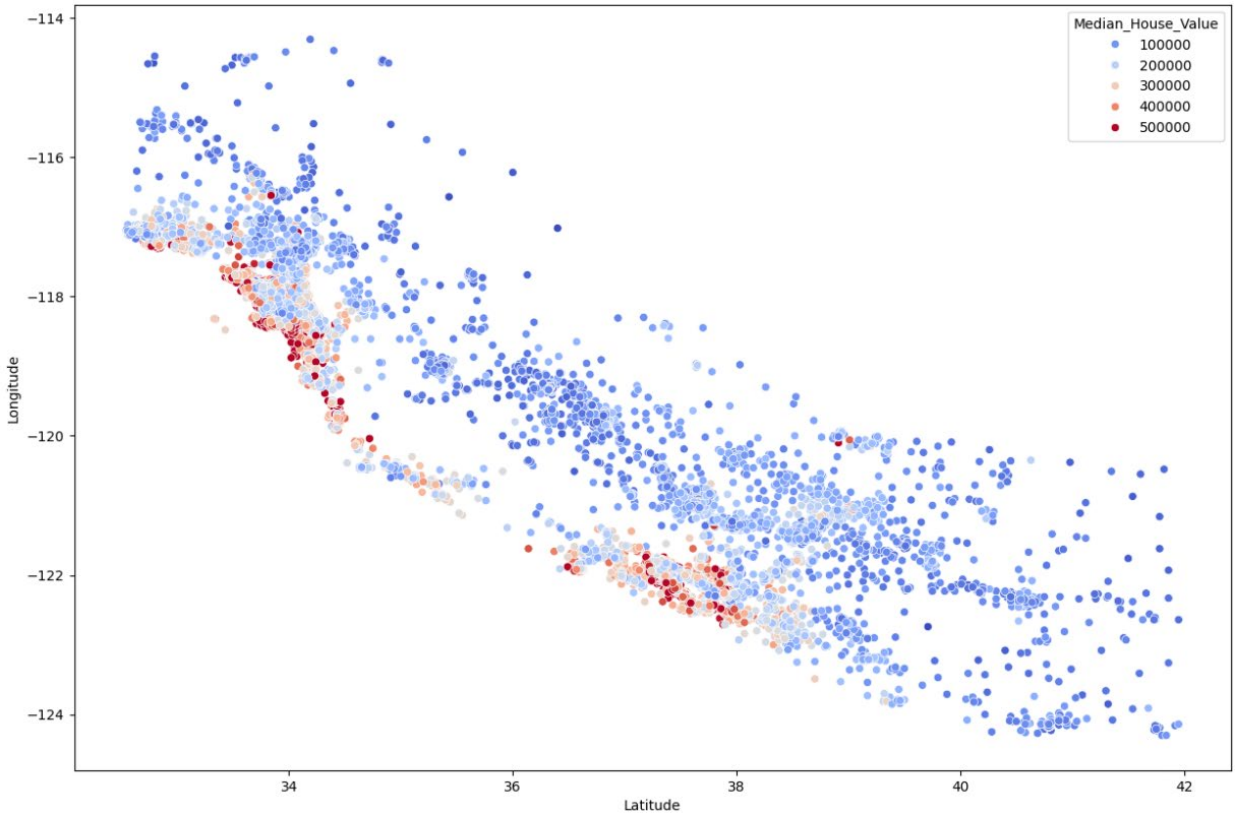
For an example, from the graph we deduce a high correlation between # of bedrooms, and # of rooms vs low correlation between latitude and longitude.

```
#Showing how the position of features block affect the price target
plt.figure(figsize=(15,10))
sns.scatterplot(x="Latitude",y="Longitude",data=train_data,hue="Median_House_Value",palette="coolwarm")
```

This line is used to visualize the relationship between latitude, longitude, and median house value in the training data. This information can be used to understand the factors that affect median house values.

The scatter plot will show the latitude and longitude of each data point, colored according to the median house value at that location. By examining the scatter plot, you can identify areas with high and low median house values. You can also identify patterns in the data, such as whether median house values are higher or lower in certain areas.

This information can be used to understand the factors that affect median house values. For example, you may find that median house values are higher in areas that are closer to major cities or that have better schools.




```
#Scaling features for all sets(Gets better results)
sc = StandardScaler()

x_train_scaled = sc.fit_transform(x_train)
x_valid_scaled = sc.fit_transform(x_valid)
x_test_scaled = sc.fit_transform(x_test)
```

Some algorithms, such as linear regression, logistic regression, and support vector machines, are more sensitive to the scale of the data than others. For these algorithms, it is generally recommended to apply standard scaling after normalization using log.

Standard scaling helps to ensure that all of the features are on the same scale. This can improve the performance of machine learning algorithms by making it easier for the algorithms to learn the relationships between the features.

LINEAR REGRESSION MODEL:

```
#1)Linear regression model

#Building model and fitting
Linmodel =LinearRegression()

Linmodel.fit(x_train_scaled,t_train)
```

1. Linmodel = LinearRegression(): This line creates a new LinearRegression object.
2. Linmodel.fit(x_train_scaled,t_train): This line trains the linear regression model on the training data.

```
#Evaluate Linear reg model using CV

param_grid = {'fit_intercept': [True, False], 'positive': [True, False]}
grid_search_linear = GridSearchCV(Linmodel, param_grid, cv=5)
grid_search_linear.fit(x_train_scaled, t_train)
best_linear =grid_search_linear.best_estimator_

y_valid_linear= best_linear.predict(x_valid_scaled)

linear_Vscore = metrics.r2_score(t_valid,y_valid_linear)
print("Validation set accuracy: " ,linear_Vscore)
```

3. param_grid={'fit_intercept': [True, False], 'positive': [True, False]}:

This line defines a parameter grid for the linear regression model. The fit_intercept parameter controls whether the model should fit an intercept term. The positive parameter controls whether the model should predict positive values only.

4. grid_search_linear = GridSearchCV(Linmodel, param_grid, cv=5): This line creates a GridSearchCV object to perform cross-validation on the linear regression model. The GridSearchCV object will try all possible combinations of the parameters in the param_grid dictionary and select the model that performs best on the cross-validation folds.
5. grid_search_linear.fit(x_train_scaled, t_train): This line fits the linear regression model on the training data using cross-validation.
6. best_linear =grid_search_linear.best_estimator_: This line selects the best linear regression model from the cross-validation.
7. y_valid_linear= best_linear.predict(x_valid_scaled): This line makes predictions on the validation data using the best linear regression model.
8. linear_Vscore = metrics.r2_score(t_valid,y_valid_linear): This line calculates the R-squared score of the linear regression model on the validation data. The R-squared score is a measure of the goodness of fit of the model.
9. print("Validation set accuracy: " ,linear_Vscore): This line prints the R-squared score of the linear regression model on the validation data to the console.

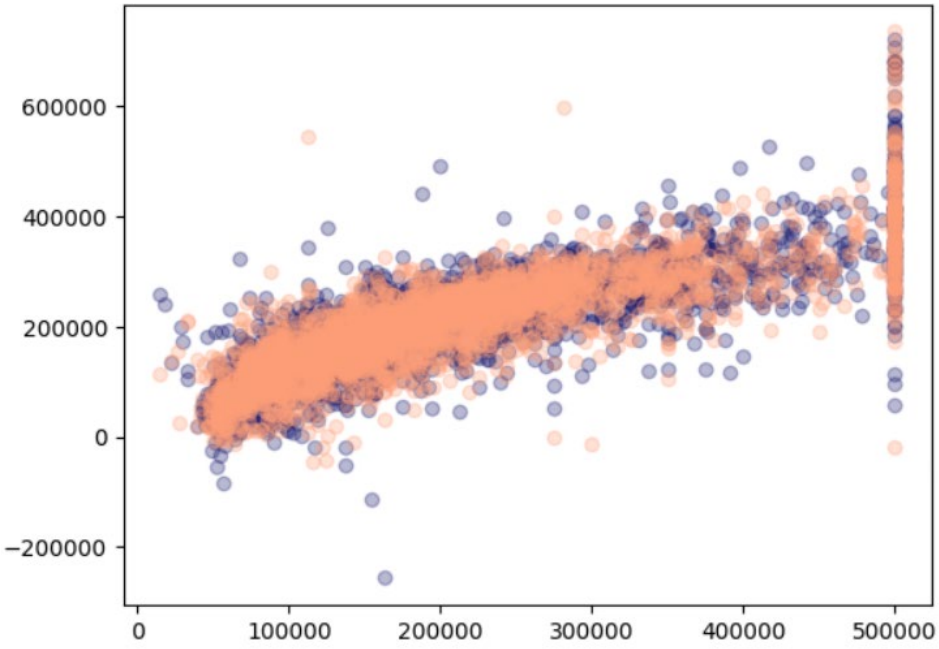
After also testing the test set, by making predictions on it using the best linear regression model, and analyzing its R squared score, we find that: Validation set accuracy → 0.675167512281825 while test set accuracy → 0.655731173500683.

CALCULATING MEAN SQUARE ERROR AND MAIN ABSOLUTE ERROR OF VALIDATION AND TEST SETS:

```
Mean squared error(Valid set) : 4270642478.55
Mean Absolute error(Valid set) : 47243.40
Mean squared error(Test set) : 4511669899.87
Mean Absolute error(Test set) :  48755.27
```

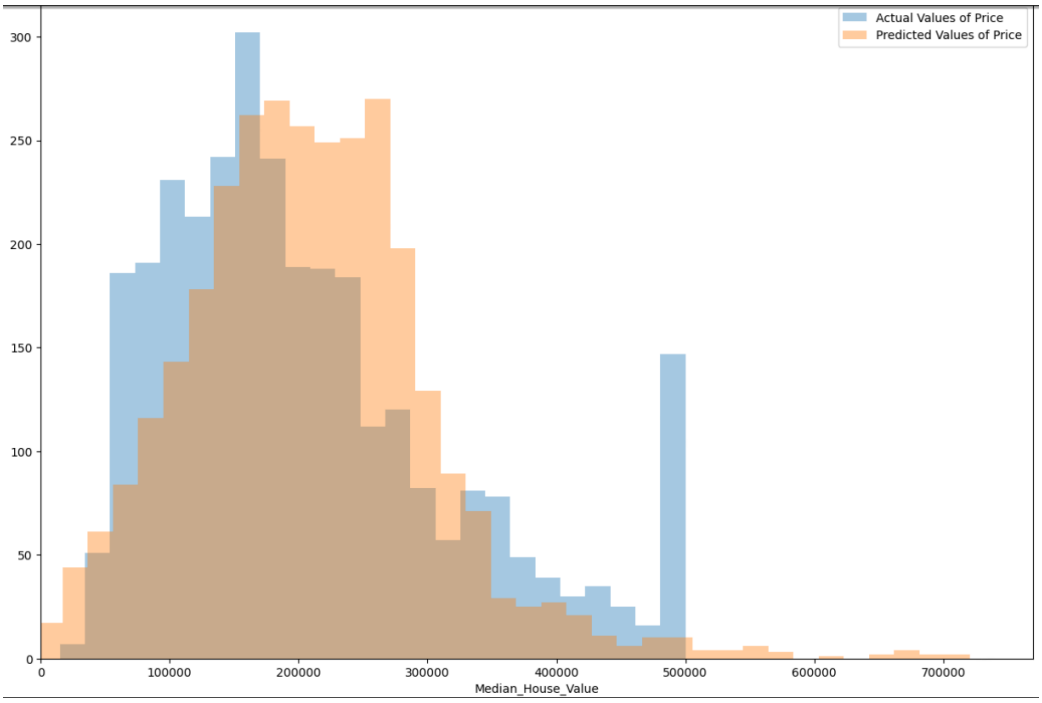
We calculated the (MSE) and (MAE) to evaluate the performance of our machine learning model on the validation and test sets. The MSE is calculated by averaging the squares of the errors between actual and predicated values. The MAE is calculated by averaging the absolute values of the errors.

SHOWING HOW THE VALIDATION SET HELPED THE TEST SET: (BLUE → TEST, SALMON → VALIDATION)



The test model mainly generates similar results, with only very few deviations.

```
#Showing how well the model predicted the targets
```



The results displayed, show correct predictions for the most part, except for slight variations. This indicates overall good accuracy of the model.

LASSO REGRESSION MODEL:

```
#2)Lasso regression model
#Building model and fitting
lasso= Lasso(random_state= 42)
lasso.fit(x_train_scaled,t_train)
```

The random state is a seed value that controls how the model is initialized. By setting it to a fixed value, we ensure that the model is initialized in the same way every time it is trained.

```
range1= [0.00005, 0.00006, 0.00007, 0.00008, 0.00009, 0.0001, .0002, .0003,
.0004, .0005, .0006, .0007, .0008, .0009, .001]
```

This indicates a list of hyperparameters (alpha) that controls the amount of L1 regularization applied by the Lasso model.

```
params_grid= {'alpha': range1}
grid_search_lasso= GridSearchCV(estimator= lasso,
                        param_grid= params_grid,
                        cv= 3,
                        scoring= 'neg_mean_absolute_error',
                        return_train_score= True,
                        n_jobs= -1,
                        verbose= 1)
```

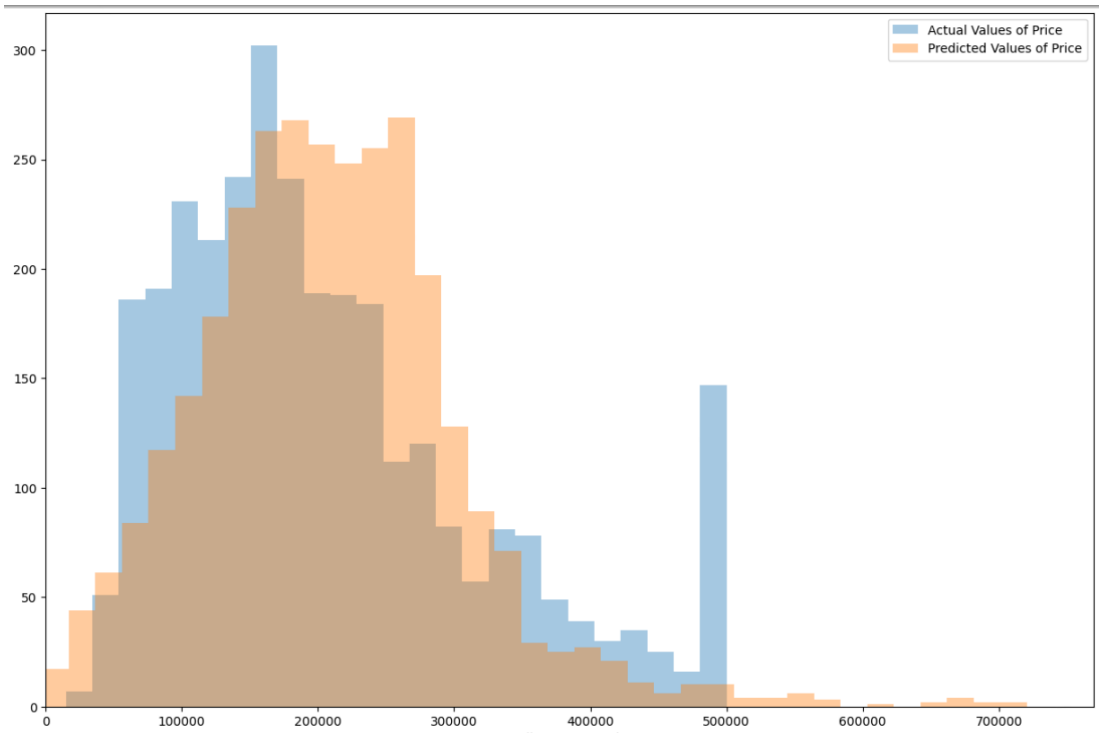
The GridSearchCV object will try all possible combinations of the hyperparameters in the parameter grid and select the model that performs best on the cross-validation folds.

```
grid_search_lasso.fit(x_train_scaled, t_train)
best_lasso =grid_search_lasso.best_estimator_
y_valid_lasso= best_lasso.predict(x_valid_scaled)
lasso_Vscore =metrics.r2_score(t_valid,y_valid_lasso)
```

best_lasso =grid_search_lasso.best_estimator: selects the best Lasso model from the cross-validation.
y_valid_lasso= best_lasso.predict(x_valid_scaled): makes predictions on the validation data using the best Lasso model.
lasso_Vscore =metrics.r2_score(t_valid,y_valid_lasso): calculates the R-squared score of the Lasso model on the validation data. The R-squared score is a measure of the goodness of fit of the model.

```
Results:
Validation set accuracy:  0.6751705443220783
Test set accuracy:  0.6557189275159471

Mean squared error(Valid set) : 4270602615.67
Mean Absolute error(Valid set) : 47242.89
Mean squared error(Test set) : 4511830384.46
Mean Absolute error(Test set) :  48757.05
```



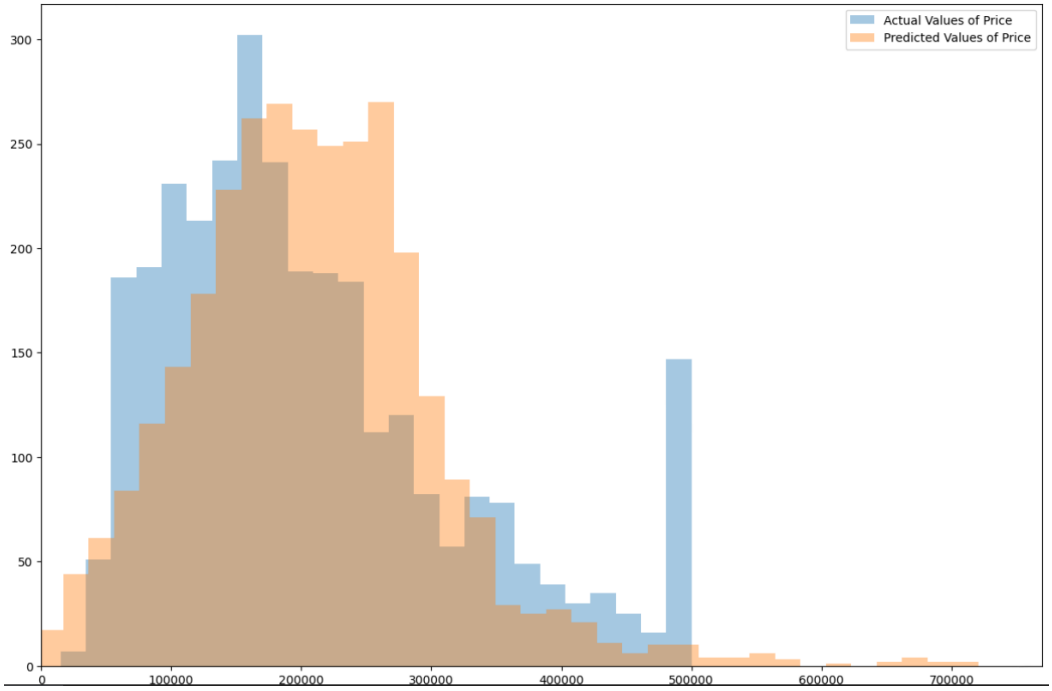
Model shows similar results to that of Linear Regression Model. The results displayed, show correct predictions for the most part, except for slight variations. This indicates overall good accuracy of the model.

RIDGE REGRESSION MODEL

```
params_grid = {'alpha': [0.0001, 0.001, 0.01, 0.05, 0.1,
 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0, 3.0,
 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]}
```

The grid search will try all possible combinations of the hyperparameters in the parameter grid and select the model that performs best on the cross-validation folds. The range of alpha values from 0.0001 to 10.0 suggests that the Ridge model is very sensitive to the alpha hyperparameter.

```
Fitting 3 folds for each of 23 candidates, totalling 69 fits
Validation set accuracy:  0.6751675151411787
Test set accuracy:  0.6557311712466787
Mean squared error(Valid set) : 4270642440.96
Mean Absolute error(Valid set) : 47243.40
Mean squared error(Test set) : 4511669929.41
Mean Absolute error(Test set) :  48755.27
```



Model shows similar results to that of Linear Regression and Lasso Model. The results displayed, show correct predictions for the most part, except for slight variations. This indicates overall good accuracy of the model.

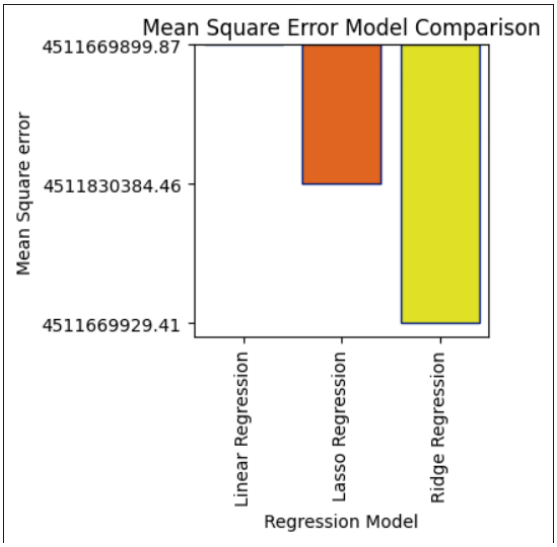
SUMMARY OF MODELS (COMPARRING RESULTS)

	Regression Model	Test set performance	Mean Square error	Mean Absolute error
0	Linear Regression	0.655731	4511669899.87	48755.27
1	Lasso Regression	0.655719	4511830384.46	48757.05
2	Ridge Regression	0.655731	4511669929.41	48755.27

The three models have very similar test set performance, with all three models achieving a test set performance of over 0.65. However, there are some very minor differences in the mean square error and mean absolute error of the three models.

1. Lasso Regression has the highest mean square error, indicating that it is making slightly larger errors in predicting the target variable.
2. Linear Regression has the lowest mean square error, indicating that it is making the smallest errors in predicting the target variable.
3. Ridge Regression has a mean square error that is similar to Linear Regression.

As for the mean absolute error, all three models are very similar.



FOR DIFFERENT VALUES OF ALPHA (HYPERPARAMETER):

Shown below is a comparison of randomly chosen values of alpha (hyperparameter), and its effect on Test set performance, as well as MSE and MAE. The results show highly similarity between the three regression models performance.

	Regression Model	Test set performance	Mean Square error	Mean Absolute error
0	Linear Regression	0.661515	4518313042.83	47974.35
1	Lasso Regression	0.661499	4518525475.09	47974.68
2	Ridge Regression	0.661515	4518313105.67	47974.35

	Regression Model	Test set performance	Mean Square error	Mean Absolute error
0	Linear Regression	0.669265	4222398709.83	46987.24
1	Lasso Regression	0.669254	4222536921.61	46987.54
2	Ridge Regression	0.669265	4222398733.64	46987.24

	Regression Model	Test set performance	Mean Square error	Mean Absolute error
0	Linear Regression	0.675152	4479672008.15	48622.83
1	Lasso Regression	0.675153	4479658734.03	48624.02
2	Ridge Regression	0.675152	4479671998.47	48622.83

	Regression Model	Test set performance	Mean Square error	Mean Absolute error
0	Linear Regression	0.692347	4188903421.09	47347.16
1	Lasso Regression	0.692344	4188943477.50	47347.35
2	Ridge Regression	0.692347	4188903450.41	47347.16

	Regression Model	Test set performance	Mean Square error	Mean Absolute error
0	Linear Regression	0.681353	4208088935.30	47739.05
1	Lasso Regression	0.681349	4208136143.52	47740.30
2	Ridge Regression	0.681353	4208088971.99	47739.05