Documentation for Machine Learning Project on Housing Price Predictor

Introduction: -

Reading about the role of machine learning in developing futuristic technologies like self-driving cars got me interested in understanding machine learning over my freshman summer. Therefore, during the vacations, I completed Andrew Ng's Machine Learning Specialization and the first three out of five courses of the Deep Learning Specialization on Coursera.

Even though I felt like I had learnt a lot over the duration of these courses, I was not convinced that I could apply all that I had learnt from the courses. Hence, to challenge myself and build confidence, I decided to build a machine learning project.

This project focuses on one of the most relatable aspects of this field—housing price prediction. Given the significance of real estate as an investment and living necessity, understanding what influences property values is important for buyers, sellers, and investors.

This machine learning project uses the Boston Housing dataset from Kaggle, a well-regarded resource in machine learning communities for benchmarking regression algorithms. The dataset comprises various attributes of housing stocks in the Boston area, which makes it an ideal candidate for deploying and evaluating predictive models.

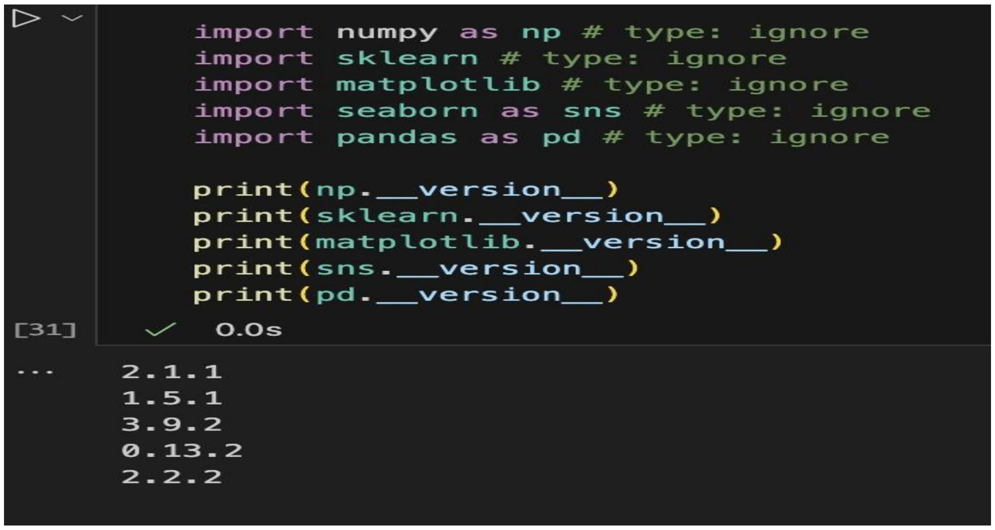
The objective of this project was to develop a model that can predict the median value of homes in Boston, based on features such as crime rates, property taxes, and age of houses. To achieve this, the project utilizes a Random Forest Regressor, which is known for its robustness and efficiency in handling non-linear data without overfitting.

Hyperparameter tuning is employed to optimize the model to check whether the Random Forest Regressor can be further or not. The results of this project are expected to provide insightful predictions and reveal the most influential factors affecting home prices in Boston.

This documentation outlines the project's methodology, discusses the results, and mentions some of the challenges I faced as I was working on my first machine learning project and how I overcame them. By the end of this document, readers will gain a deeper understanding of how machine learning can be applied to solve real-world problems in the real estate market.

Description of the Code (with each point describing an individual block of code): -

- I begin the code by importing libraries that are important for executing several machine learning projects: numpy (for numerical calculations), sklearn (for efficiently executing pre-defined machine learning algorithms), matplotlib (for data visualization), seaborn (for data visualization), and pandas (for data manipulations). I then printed their versions to not just check whether their latest versions had been installed (to ensure compatibility and feature availability), but also confirm that the libraries were downloaded.



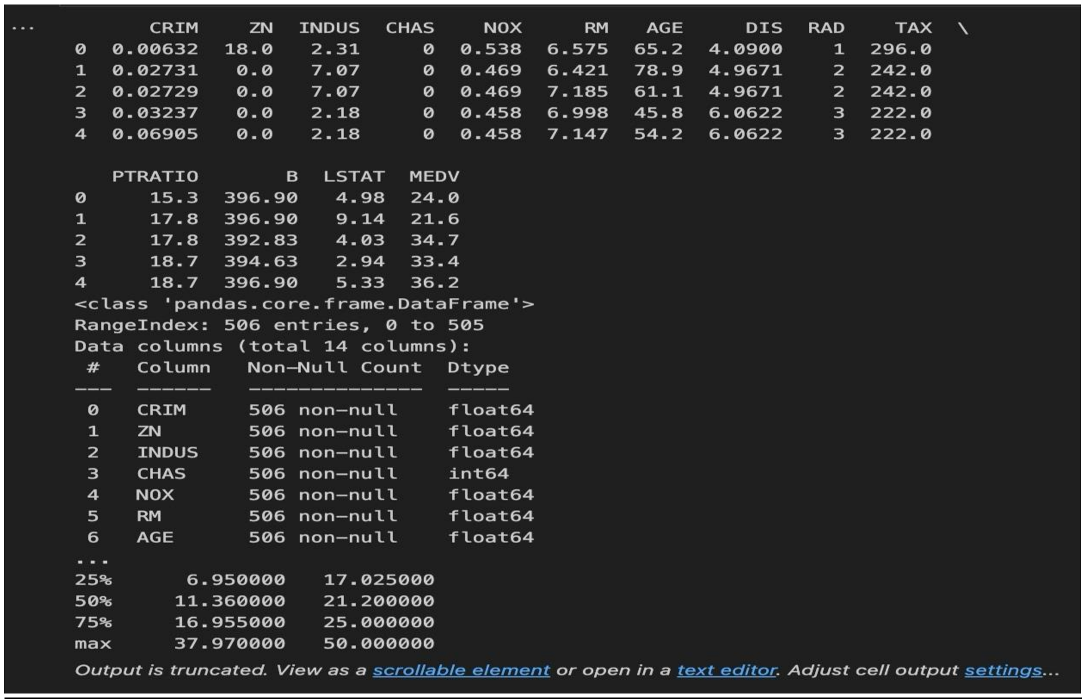
```
import numpy as np # type: ignore
import sklearn # type: ignore
import matplotlib # type: ignore
import seaborn as sns # type: ignore
import pandas as pd # type: ignore

print(np.__version__)
print(sklearn.__version__)
print(matplotlib.__version__)
print(sns.__version__)
print(pd.__version__)
```

[31] ✓ 0.0s

```
...  2.1.1
     1.5.1
     3.9.2
     0.13.2
     2.2.2
```

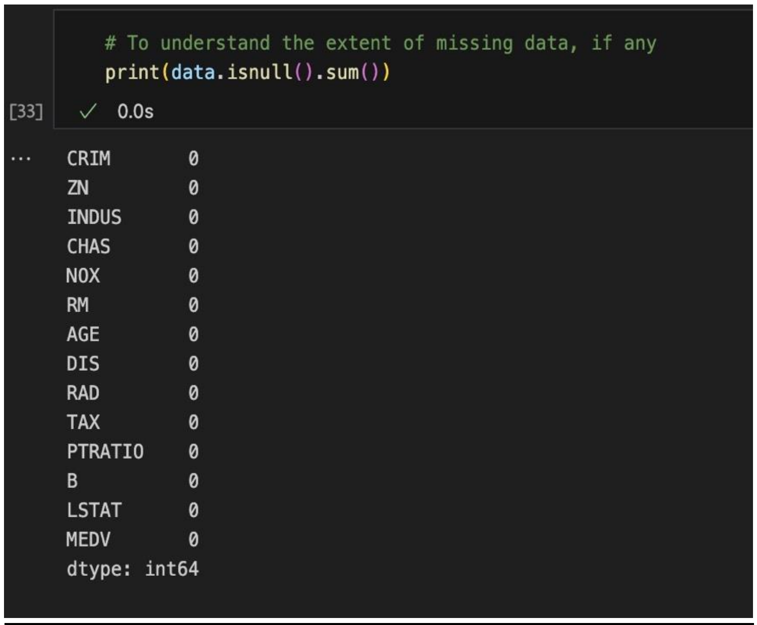
- I loaded the Boston CSV dataset file into the ‘data’ variable. I printed “data.head()” to display the data highlighted in the first few rows of the file, which helps us understand the structure and the type of data that we are dealing with. “data.info()” is a pandas function that prints the number of columns, column labels, column data types, and the number of cells in each columns that are non-null values (among other information). “data.describe()” is another pandas function that is a convenient way to get the count, mean, standard deviation, minimum, 25th percentile (Q1), median (Q2), 75th percentile (Q3), and maximum of the columns. These functions help us understand the dataset a lot better.



```
...       CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
   0  0.00632  18.0   2.31     0  0.538  6.575  65.2  4.0900    1  296.0
   1  0.02731   0.0   7.07     0  0.469  6.421  78.9  4.9671    2  242.0
   2  0.02729   0.0   7.07     0  0.469  7.185  61.1  4.9671    2  242.0
   3  0.03237   0.0   2.18     0  0.458  6.998  45.8  6.0622    3  222.0
   4  0.06905   0.0   2.18     0  0.458  7.147  54.2  6.0622    3  222.0

      PTRATIO       B  LSTAT  MEDV
   0     15.3  396.90   4.98  24.0
   1     17.8  396.90   9.14  21.6
   2     17.8  392.83   4.03  34.7
   3     18.7  394.63   2.94  33.4
   4     18.7  396.90   5.33  36.2
   <class 'pandas.core.frame.DataFrame'>
   RangeIndex: 506 entries, 0 to 505
   Data columns (total 14 columns):
    #   Column   Non-Null Count  Dtype
   ---  ------   --------------  -----
    0   CRIM     506 non-null    float64
    1   ZN       506 non-null    float64
    2   INDUS    506 non-null    float64
    3   CHAS     506 non-null    int64
    4   NOX      506 non-null    float64
    5   RM       506 non-null    float64
    6   AGE      506 non-null    float64
   ...
   25%      6.950000   17.025000
   50%     11.360000   21.200000
   75%     16.955000   25.000000
   max     37.970000   50.000000
   Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

- I print "data.isnull().sum()" to check whether the dataset is complete or not. Since all the respective columns have 0 printed in front of them, I understand that the dataset is complete and we do not need to complete any missing data. This is rare in the real-world.

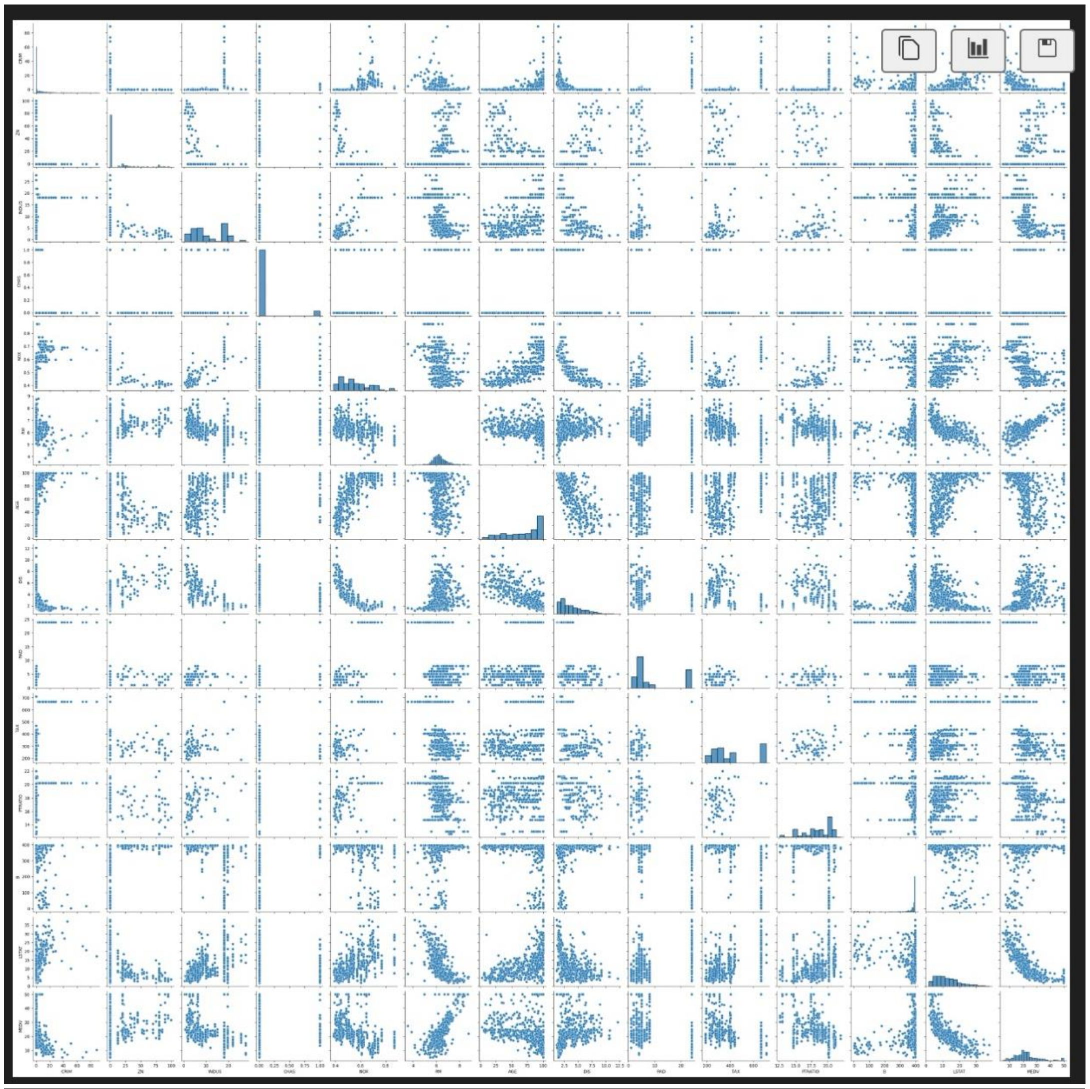


```
# To understand the extent of missing data, if any
print(data.isnull().sum())
```

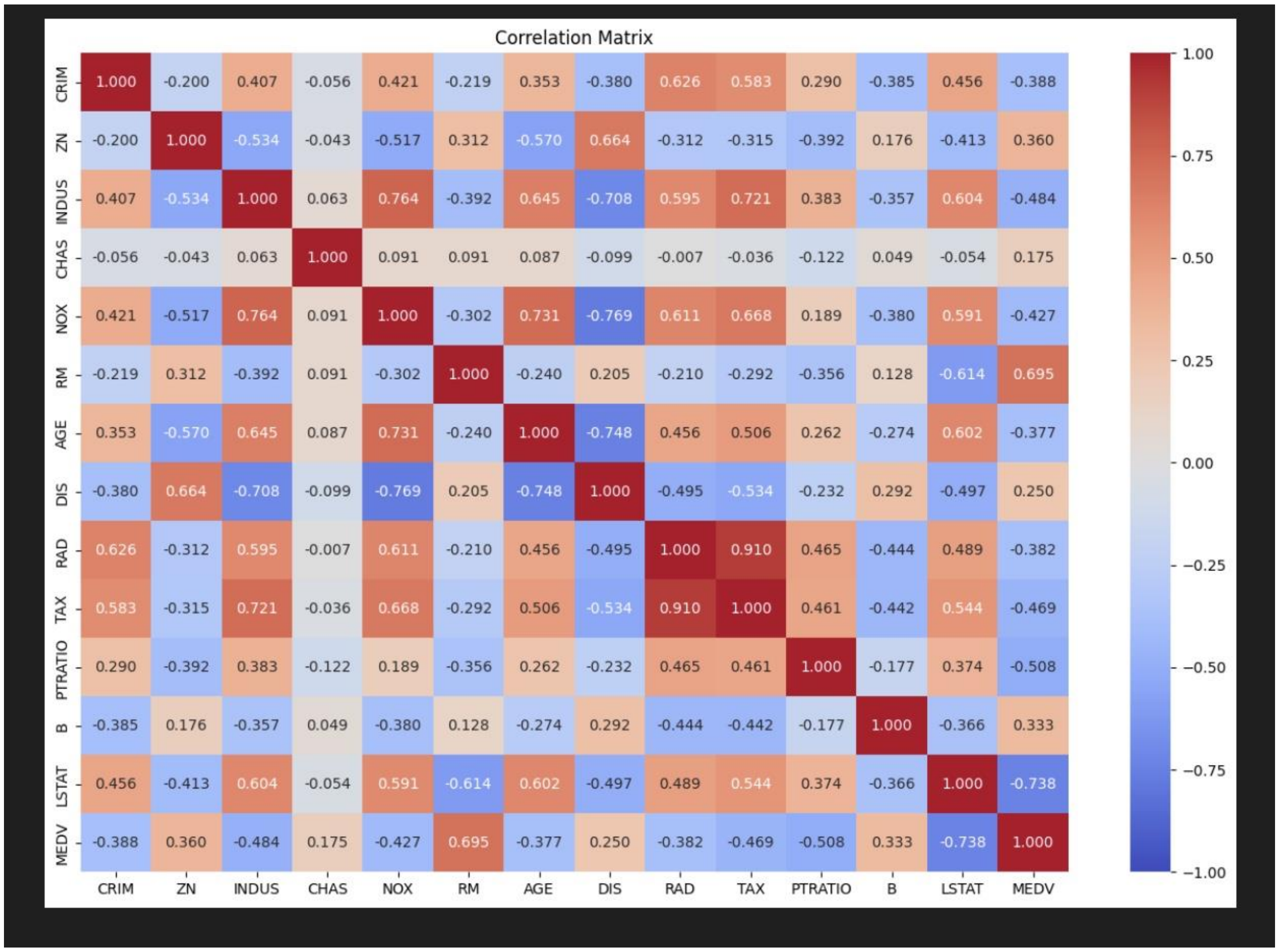
```
[33]  ✓ 0.0s
```

```
...  CRIM       0
     ZN         0
     INDUS      0
     CHAS       0
     NOX        0
     RM         0
     AGE        0
     DIS        0
     RAD        0
     TAX        0
     PTRATIO    0
     B          0
     LSTAT      0
     MEDV       0
     dtype: int64
```

- "sns.pairplot()" is a seaborn function used to create pairwise plots of the variables involved in the dataset. We then plot this using matlab.pyplot.show().



- The pairwise plot was hard to understand due to several different variables being present in the dataset. Therefore, I decided to then use the Correlation Matrix Heatmap. For that, I use the .corr() pandas function, which finds the correlation between columns of the dataset. I use the matplotlib.pyplot.figure() to plot a figure that is 15 inches wide and 10 inches tall.
  I used the seaborn.heatmap() to create the heatmap using the correlation matrix found. I added the correlation coefficient values directly onto each cell in the heatmap using "annot=True". I used the "coolwarm" color and set the numerical annotation to display three decimal places. Furthermore, I set the correlation range to be between -1 and 1 (to ensure that the colors are scaled appropriately across the entire correlation matrix). I then show this heatmap with the title "Correlation Matrix".



	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
CRIM	1.000	-0.200	0.407	-0.056	0.421	-0.219	0.353	-0.380	0.626	0.583	0.290	-0.385	0.456	-0.388
ZN	-0.200	1.000	-0.534	-0.043	-0.517	0.312	-0.570	0.664	-0.312	-0.315	-0.392	0.176	-0.413	0.360
INDUS	0.407	-0.534	1.000	0.063	0.764	-0.392	0.645	-0.708	0.595	0.721	0.383	-0.357	0.604	-0.484
CHAS	-0.056	-0.043	0.063	1.000	0.091	0.091	0.087	-0.099	-0.007	-0.036	-0.122	0.049	-0.054	0.175
NOX	0.421	-0.517	0.764	0.091	1.000	-0.302	0.731	-0.769	0.611	0.668	0.189	-0.380	0.591	-0.427
RM	-0.219	0.312	-0.392	0.091	-0.302	1.000	-0.240	0.205	-0.210	-0.292	-0.356	0.128	-0.614	0.695
AGE	0.353	-0.570	0.645	0.087	0.731	-0.240	1.000	-0.748	0.456	0.506	0.262	-0.274	0.602	-0.377
DIS	-0.380	0.664	-0.708	-0.099	-0.769	0.205	-0.748	1.000	-0.495	-0.534	-0.232	0.292	-0.497	0.250
RAD	0.626	-0.312	0.595	-0.007	0.611	-0.210	0.456	-0.495	1.000	0.910	0.465	-0.444	0.489	-0.382
TAX	0.583	-0.315	0.721	-0.036	0.668	-0.292	0.506	-0.534	0.910	1.000	0.461	-0.442	0.544	-0.469
PTRATIO	0.290	-0.392	0.383	-0.122	0.189	-0.356	0.262	-0.232	0.465	0.461	1.000	-0.177	0.374	-0.508
B	-0.385	0.176	-0.357	0.049	-0.380	0.128	-0.274	0.292	-0.444	-0.442	-0.177	1.000	-0.366	0.333
LSTAT	0.456	-0.413	0.604	-0.054	0.591	-0.614	0.602	-0.497	0.489	0.544	0.374	-0.366	1.000	-0.738
MEDV	-0.388	0.360	-0.484	0.175	-0.427	0.695	-0.377	0.250	-0.382	-0.469	-0.508	0.333	-0.738	1.000

- I print the “data.shape” to check the number of rows and columns, which is 506 rows and 14 columns. Looking at this, I determine that the training data is not big enough to have a cross-validation set with a training and test set.

- I import multiple libraries to begin training the model. The train_test_split from sklearn.model_selection is used to split the dataset into the training and test sets. The StandardScaler from sklearn.preprocessing is used to standardize features by ensuring that all features are on the same scale, which can improve the performance of machine learning algorithms. The LinearRegression is imported from sklearn.linear_model and is used to find a linear relationship between input features X and the target variable Y. This is used commonly for regression problems.

  I had to use #type: ignore in my code to ignore any type checking warnings, but it does not affect the functionality of the code.

  The ”.drop()” function is used to remove the ”MEDV” column present in the “data” dataset. Use of ”axis=1”suggests that we are dropping a column and not a row and that the rest of the columns are to be loaded as input X. We load y to be the MEDV column from the dataset using "data['MEDV']".

I print the shape of X and y to confirm whether the data has correctly loaded or not and I get the following output for the same:
X = (506, 13)
y = (506,)

- I used the train_test_split() function to split the dataset into the training and test sets. I specified the feature matrix to be X and the target vector to be y. I specified the test set to be 20% of the data (with the training data being the other 80%). I initialized the random_state to be equal to 30 to ensure the randomness of the split is reproducible. This maintains consistency in the results that we get instead of getting a different split of data each time. Using this, we get X_train (training data for features), X_test (test data for features), y_train (training data for the target variable, i.e., the housing prices), and y_test (test data for the target variable).

  I then standardize the features by scaling the features so that they have a mean of 0 and a standard deviation of 1. This ensures that the features are evaluated on the same scale, which helps the algorithm's performance by making them converge faster. This is important as unscaled features may disproportionately affect the model.

  I initialize a variable named "scaler" to be an instance of the StandardScaler class, which is used to standardize the features. The .fit_transform() is a function of the sklearn library, which is used to fits the scaler on the training data and also transforms it. It does this by computing the mean and standard deviation of X_train and then standardizing the data using these values. The .transform() method of the sklearn library is applied to the test data using the same mean and standard deviation computed from the training data. This makes the test data to be standardized in the same way as the training data.

  I initialize "model" to be an instance of the LinearRegression() model of the sklearn library. Regression looks to find the best-fitting line that minimizes the difference between the predicted values and the actual target values (house prices).

  The .fit() method trains the linear regression model using the standardized training data (X_train_scaled) and the corresponding target values (y_train).

- I use .predict() method of sklearn is used to generate predictions from a trained model. Then, I import mean_squared_error, mean_absolute_error, and r2_score from sklearn to find out how well our trained model is doing by comparing the actual y values (y_test) with the predicted y values found post training the model (y_pred).

  Mean squared error (mse) measures the average of the squared differences between actual and predicted values. A lower MSE indicates that the predicted values are closer to the actual values. Mean absolute error (mae) is used to measure the average of the absolute differences between actual and predicted values. a lower value of MAE indicates better performance. Lastly, R^2 score (r2) shows how well the model's predictions fit the data. It gives a value between 0 and 1, where 1 means perfect predictions and 0 means extremely poor performance (random guessing).

The output that I received was:

```
Mean Squared Error: 17.933042188699112
Mean Absolute Error: 3.225410406976328
R² Score: 0.7181173900062088
```

- Looking at the outputs above, I realized that the algorithm was not performing well enough for the dataset at hand. Therefore, I decided to use the Random Forest Regressor. It is an algorithm that combines multiple decision trees to make more accurate predictions. This algorithm fetches a better performance for non-linear relationships and complex datasets.

  I have the same split and random_state as I had in LinearRegression() used previously. I also created 2 more variables (X_train2_scaled and X_test2_scaled). The .fit_transform() is used to fit the scaler on the training data and also transform it.

  This time, I used the model to be RandomForestRegressor() through the variable model2. The .fit() function trains the Random Forest model using the scaled training data (X_train2_scaled) and the actual housing prices (y_train2). The .predict() function is used to predict the housing price y_pred2 using the input, that is, X_test2_scaled.

  Just like for Linear Regression, I will find the MSE, MAE, and R^2 for the variables y_test2 and y_pred2. The 3 scores in the previously mentioned order are:

  ```
  Random Forest Mean Squared Error: 6.398625627450983
  Random Forest Mean Absolute Error: 1.904745098039215
  Random Forest R² Score: 0.8994224586514574
  ```

  As shown, after using this algorithm, the performance of the model improves tremendously and we end up getting much better predictions than we did for Linear Regression.

- The function plot_learning_curve is used to plot learning curves for the Random Forest model by evaluating by evaluating how the model's performance changes as the size of the training dataset increases. Plotting a learning curve helps you know whether your model is overfitting, underfitting, or well-generalized to the dataset on which it is trained.

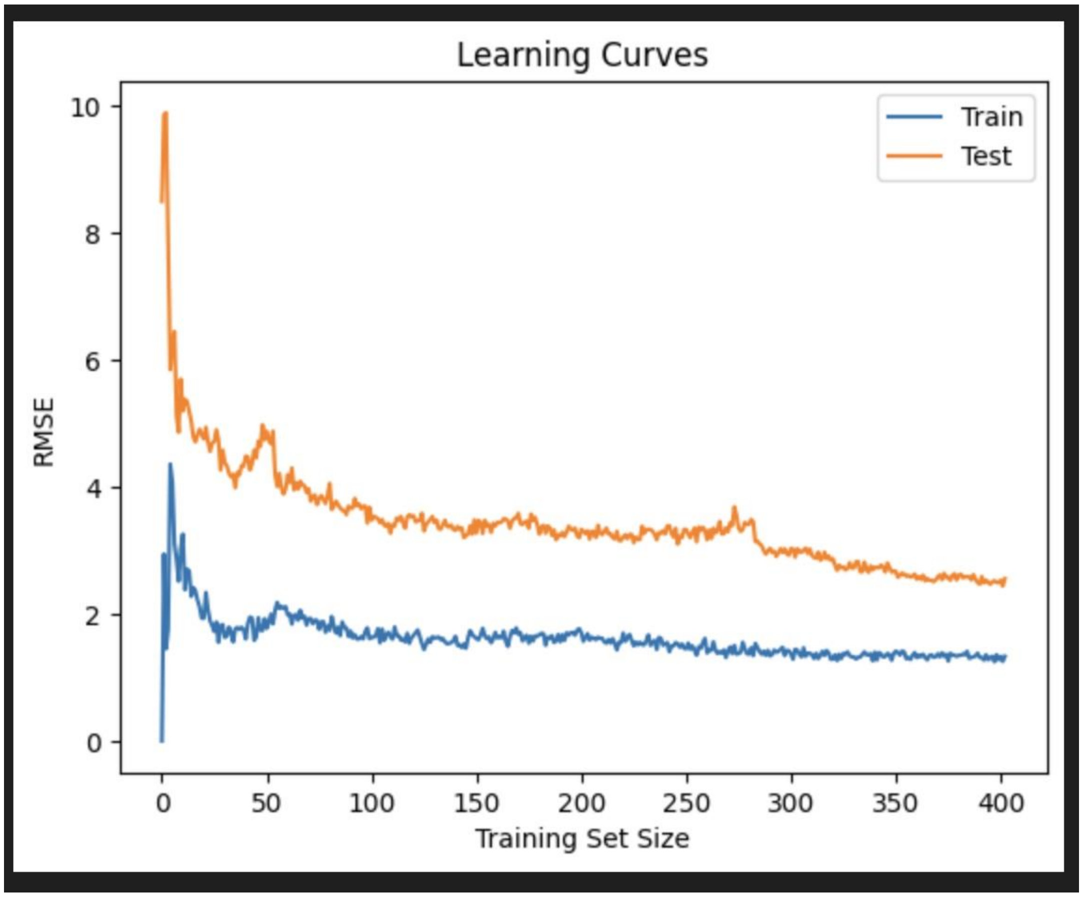
I created the train_errors and test_errors variables to keep track of the Mean Squared Errors at each step of the for loop to follow.

The range of the loop is from 1 to the length of the training set. The for loop iterates over the training data in increasing sizes to see how the model is doing as the amount of information processed by the model gradually increases.

The first m rows of X_train2 and y_train2 (X_train2[:m], y_train2[:m]) can be used to train the model using the .fit() function.
y_train_predict2 is used to make predictions on the subset of the training data X_train2[:m] and y_test_predict2 is used to make predictions on the entire test set X_test2. This helps us examine how the model's generalization error changes as it leans gradually from the training set.

Then, y_train2[:m] (the actual target values) are compared with y_train_predict2 (the predicted target value) and the MSE for this is appended to the train_errors.
y_test2 is compared with y_test_predict2 and the MSE for this is appended to test_errors.

As a final step, we plot these results through the matplotlib.pyplot library.



- The plot above shows that as the amount of data that the model2 uses to train increases, so does its performance, that is, it gradually generalizes to the dataset.

  The next step would be to check whether the performance of this model can be improved or not. For this, we use hyperparameter tuning. There are 2 primary methods that I used to do this. The first method that I use for the tuning is Randomized Search Cross-Validation technique instead of Grid Search as then we explore the hyperparameter set randomly instead of gradually exploring various hyperparameters in order. This can often be more efficient and faster, especially when dealing with a large hyperparameter space.

  For this, we import RandomizedSearchCV from sklearn that helps in finding the optimal hyperparameters for a model using the method stated earlier, that is, it tries different combinations of hyperparameters from a defined search space. radint is imported from scipy.stats that generates random integers within a specified range, which is used here to define range for hyperparameter values.

  The variable model3 is used to store the RandomForestRegressor() algorithm to test the hyperparameters. param_distributions is a dictionary that defines the range of hyperparameters to be tuned – number of trees in the random forest (n_estimators – randomly samples between 10 and 200), maximum depth of each decision tree (max depth – having None/unrestricted level, 10, 20, or 30 levels), minimum number of samples required to split an internal node (min_samples_split – ranges between 2 and 10), and the minimum number of samples required to be at a leaf node (min_samples_leaf – ranges between 1 and 10).

  Next, we initialize the random_search variable to run the RandomizedSearchCV. We specify the estimator to be model3 and the param_distributions to contain the hyperparameters to be tuned. The number of iterations is 50. cv is equal to 5, which means that we create 5 equal folds. The model will be trained and validated 5 times, each time using a different fold for validation and the remaining 4 folds for training.

  Since RandomizedSearchCV minimizes the score, I use the negative Mean Squared Error to inform that the lower MSE means better performance. The random_state is kept the same as when training the models (30) to ensure that the results are not only consistent, but also reproducible.

  Then, we fit the X_train2_scaled and y_train2 to the random_search variable, which runs the search process. The 50 randomly chosen hyperparameter combinations from param_distributions use the 5-fold cross-validation to evaluate the combination. This means that for each fold, the model will train on 4 folds and validate on the remaining folds, which happens 5 times for each hyperparameter combination. The performance metrics (negative MSE) from each fold are averaged to judge the performance.

  Finally, .best_params_ is used to returns the combination of hyperparameters that performed best based on the cv process. The .best_score_ is used to display the performance of the best hyperparameter combination. Since the score is negative (due to negative MSE), we negate it again to get a positive value. The result for RandomizedSearchCV is as follows: -

```
Best Parameters: {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 3,
'n_estimators': 114}
Best Score: 13.3852300372311
```

- The RandomizedSearchCV keeps track of the best model it found during the 50 iterations and .best_estimator_ retrieves that model for us. We store this in a new variable called best_model. We fit the X_train2_scaled and y_train2 to the best_model and then find the prediction of the housing prices based on X_test2_scaled using best_model and store it in y_pred_final.

  We then find the MSE, MAE, and R^2 values for this new trained model and print it in this order. The results are as follows: -

```
Final Model Mean Squared Error: 6.617221082786907
Final Model Mean Absolute Error: 1.9425342495672748
Final Model R² Score: 0.8959864405551135
```

- Based on the results shown above, we see that the performance of the best_model is very close to the model that we had for RandomForestRegressor() without the hyperparameter tuning. This observation stayed true despite changing the different hyperparameter ranges and other variable values multiple times. Therefore, I decided to try out Grid Search Cross-Validation technique to check if it can yield better results through the more systematic approach that it follows. Unlike RandomizedSearchCV, it exhaustively tries all possible combinations of the hyperparameters you provide in a grid. Therefore, it is more thorough, but can also take longer if you have several hyperparameters to search through.

  I imported GridSearchCV from sklearn.model_selection and used the same hyperparameters inside a new dictionary variable called param_grid: n_estimators (having 50/100/150 trees in the forest), max_depth (10/20/30 or no limit/None), min_samples_split (minimum 2/5/10 samples needed to split a node to prevent overfitting), and min_samples_leaf (minimum 1/2/4 samples required to be in the leaf node to control tree complexity).

  We define a model4 to contain the RandomForestRegressor(). The grid_search variable is set up to use GridSearchCV with the model4, cv=5 (like with RandomizedSearchCV), and the dictionary of hyperparameters being param_grid. We use the negative MSE again to check the performance.
  We initialize n_jobs = -1, which instructs the model to use all available CPU cores on your machine to perform the task in parallel. It’s the most efficient setting as it will distribute the work across all the available resources, which would significantly speed up the process.

Furthermore, defining verbose = 1 prints basic information about the progress of the grid search. For example, it shows which hyperparameter combination is being tested.

Then, the remaining steps are similar to what we saw with RandomizedSearchCV – we fit X_train2_scaled and y_train2 to grid_search, print the best combination of hyperparameters found and the best score found with that through negative MSE, and retrieve the best model trained with the optimal hyperparameters to declare the variable best_model.

We predict the housing prices using this new best_model found and inputting the X_test2_scaled in the .predict() function.

Finally, we find the MSE, MAE, and R^2 scores of the new model and print those scores. The results are as follows: -

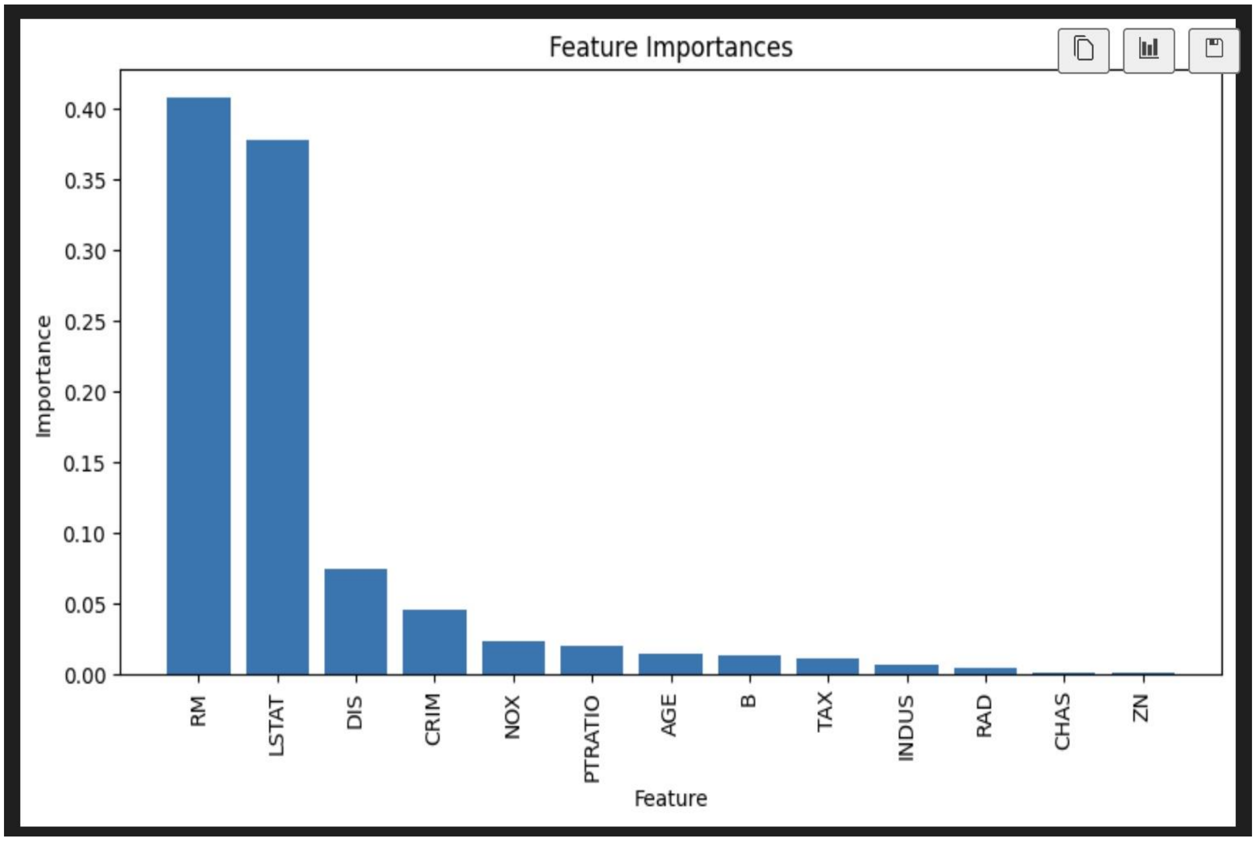
```
Fitting 5 folds for each of 108 candidates, totalling 540 fits
Best Parameters: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 5,
'n_estimators': 100}
Best Score: 13.077491446357246
Final Model Mean Squared Error: 6.846677400397343
Final Model Mean Absolute Error: 1.9753186278590686
Final Model R² Score: 0.8923797047315423
```

- The results of the GridSearchCV were slightly worse than RandomizedSearchCV. Therefore, it turns out that the original model, that is, the model without any hyperparameter tuning, ends up doing slightly better than both the techniques used of hyperparameter tuning.

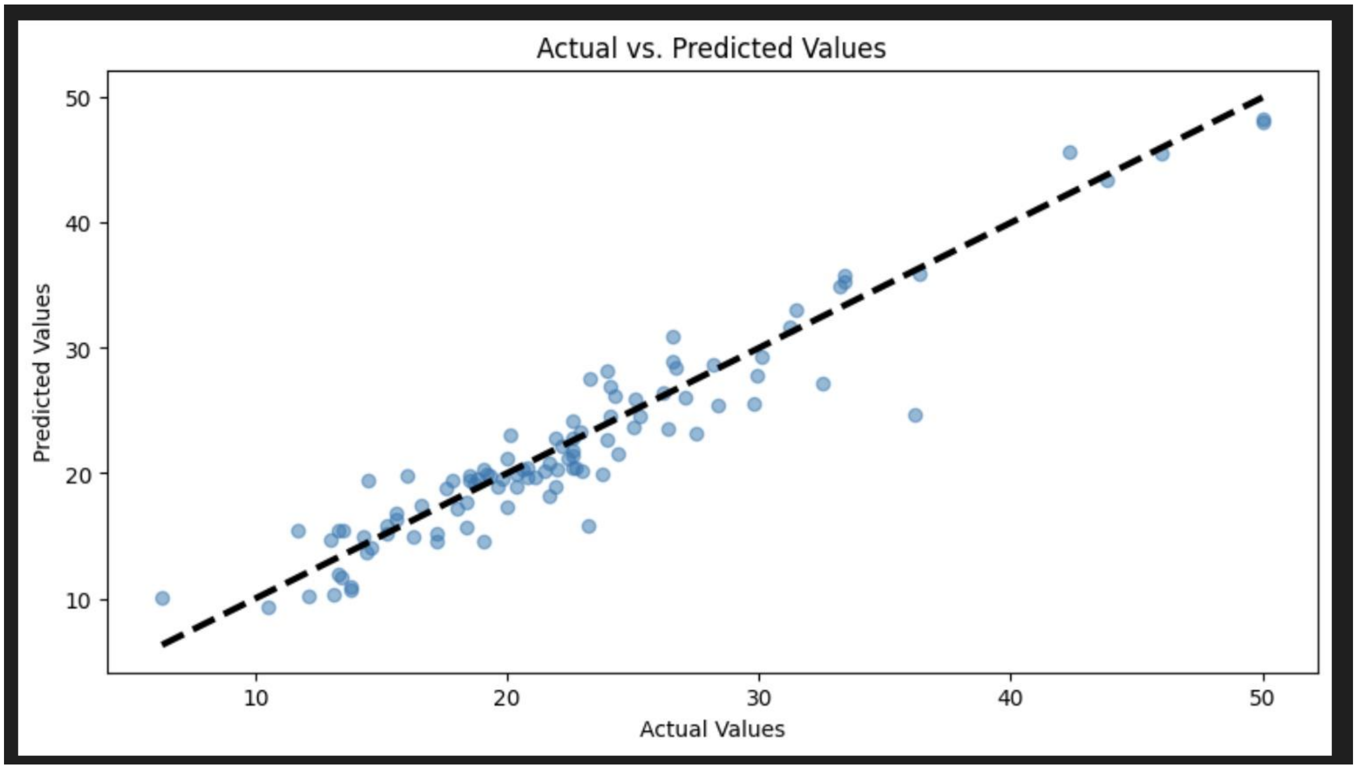
  I, therefore, use model2 as the final model (the RandomForestRegressor() without any hyperparameter tuning). I use .feature_importances_ to store the importance (metric that tells us how relevant each feature was in predicting the target value) of each feature in determining the target variable MEDV.

  The .argsort(importances) returns the indices of the features sorted in ascending order of importance. We then reverse its order to see the result in descending order (to see which features impact the result the most).

  We then plot the result using matplotlib.pyplot library.



- Finally, I use the matplot.pyplot library again to plot the final selected model in comparison to the actual data to visually see how well our trained model is doing for the given dataset.



Some Challenges I Faced or Observations I Made: -

- I found it interesting to note that the RandomizedSearchCV yielded slightly better results than GlidSearchCV even though the latter searches for different hyperparameter combinations exhaustively. I did not know that this was possible. Turns out that this is possible as GridSearchCV only checks combinations at predefined intervals and might miss better combinations that fall in between those grid points.

- I did not expect to spend so much time trying out different combinations of hyperparameters for tuning. It was an unexpectedly tedious process despite importing the algorithm to search for optimal hyperparameters from the sklearn library.

- It was challenging to think about what the flow of this project should be for training the model. Since I was teaching myself how to do this project, I had to rely on platforms like ChatGPT, Stack Overflow, and W3Schools.

- The dataset for this project was complicated, but really organized. It is rare to find datasets in the real world that are complete and often, you must spend time on feature selection and filling in the missing data. I did not have to do that for this project, but it is a skill that I am looking forward to acquiring when I work on future projects.

- I used Random Forest algorithm for this project instead of XGBoost as I was facing a lot of difficulty importing it. I wanted to try XGBoost given its reputation for being a powerful algorithm, but I could not do so due to limited time availability. Nonetheless, it’s something that I would want to use in upcoming projects.