CSE 587 Spring 2023
Project Phase 2

Team
Name: Pavana Lakshmi Venugopal
UBIT: pavanala
Name: Vaidurya Malathesha
UBIT: vaidurya

Title: Analysis of property for taxes in West Roxbury.

Problem Statement: We will study and examine the information to identify the primary causes for the total value of the property to rise, and then determine the amount of property taxes the owner will have to pay every year.

a. Background: Sending auditors to each property to access the property value is a cumbersome task, quite time consuming and expensive for the government. If we can save some amount for the government, they can be used for other essentials citizens might need. We will go through all the data points or values for West Roxbury, what features lead to the value of the property increasing or decreasing, based on that we would find out if we could predict the Total value and hence calculate the tax. If a website is created where homeowners add the details or update the details of the property regularly then this goal can be achieved. We will be analyzing the predictors to calculate the property value.

b. Potential: It would no longer be necessary for assessors to devote considerable time inspecting each home, saving governments hundreds of thousands of dollars. Machine Learning models can be used to predict the total value of the homes.

The following are the questions that will be addressed:

Which data can be retained for Total value?

Does any data need modification for our models to interpret?

Which data does not contribute to the predictions?

Which are some of the main reasons for change in total value?

We would basically analyze the data, find which features or predictors are needed, to gain an understanding and predict the response variable. Here it is Total value.

Link to dataset: https://github.com/reisanar/datasets/blob/master/WestRoxbury.csv

Deliverables **[50 marks total]**

1. **Algorithms/Visualizations [25 marks]:** Apply **5 different significant and relevant algorithms** (ML, MR, and/or statistical models) to your data and create visualizations for the results. **For 487 students**: at least 1 of the 5 algorithms must be one that was not discussed in class. **For 587 students**: at least 2 must be from outside of class. Algorithms discussed in class are: Linear Regression, k-Means, k-NN, Naive Bayes, and Logistic Regression. The outside algorithms can come from the class textbooks, or other sources. **Cite the appropriate sources for each outside algorithm you choose to apply.**
2. **Explanation and Analysis [25 marks]:** For each of the 5 above algorithms, provide justification for why you chose the particular algorithm for your particular problem, work you had to do to tune/train the model, and discuss the effectiveness of the algorithm when applied to your data to answer questions related to your problem statement. This should include discussion of any relevant metrics for demonstrating model effectiveness, as well as any intelligence you were able to gain from application of the algorithm to your data.

Based on the insights gained from the pre-processing and exploratory data analysis conducted in Phase-1, we have documented the distribution of each feature and the resulting insights.During this phase of the project, we will utilize the processed data obtained from Phase 1. With this processed data, we will make necessary adjustments and apply various machine learning models to predict the taxes.Taking into account the size of the data and the computational resources required to process it, we aim to identify the most significant features from the correlation matrix and other relevant factors. As a result, we scale down the features and choose those that are relevant in predicting the target variable, Total_value.The selected features are shown below.

LOT_SQFT YR_BUILT LIVING_AREA FLOORS ROOMS BEDROOMS FULL_BATH HALF_BATH KITCHEN FIREPLACE REMODEL_Old REMODEL_Recent

1. Multiple Linear Regression

When it comes to modeling the variation of a dependent variable based on a group of independent variables, linear regression is one of the most appropriate techniques to use.In this particular analysis, we will use linear regression to make predictions about the target variable, which is the total value.

Justification: Since linear regression is an effective approach for predicting continuous numerical values, it is frequently used to solve regression issues. The method involves fitting a

straight line that most accurately depicts the relationship between the independent and dependent variables.Since the WestRoxbury dataset contains features such as the number of rooms, lot square foot, and living area, which are continuous numerical values, linear regression is a valid option to use. Therefore, using linear regression on the WestRoxbury dataset is a valid decision.

Tune/train: They can be divided into a few steps as below.

1. Since pre-processing is already done in phase 1 we now divide the data into training and test sets.
2. Train the Linear Regression model.
3. Then we evaluate the model with certain evaluation metrics like mean squared error, root mean squared error. Also we then find the accuracy to determine how well our model is performed.

These steps are shown in the code below

```
# partition data into train and test sets
X = predictor_df_normalized
y = response_df
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.3, random_state=1)
```

```
# train the LR model
linear_model = LinearRegression()
linear_model = linear_model.fit(train_X, train_y)
```

```
# print performance metrics on training set using regressionSummary()
predicted_y_training = linear_model.predict(train_X)
regressionSummary(train_y, predicted_y_training)
```

```
Regression statistics

                    Mean Error (ME) : 0.0000
       Root Mean Squared Error (RMSE) : 44.1851
            Mean Absolute Error (MAE) : 33.5804
          Mean Percentage Error (MPE) : -1.1689
Mean Absolute Percentage Error (MAPE) : 8.7926
```

```
# deploy the model on the test data
predicted_y_test = linear_model.predict(test_X)

result = pd.DataFrame({'Predicted': predicted_y_test, 'Actual': test_y,
                       'Residual': test_y - predicted_y_test})
```

```
# Checking the model performance in prediction
regressionSummary(test_y, predicted_y_test)
```

```
Regression statistics

                    Mean Error (ME) : 0.3204
       Root Mean Squared Error (RMSE) : 44.2349
            Mean Absolute Error (MAE) : 33.1262
          Mean Percentage Error (MPE) : -1.0285
Mean Absolute Percentage Error (MAPE) : 8.6670
```

```
# Checking for accuracy
accuracy = linear_model.score(test_X, test_y)
accuracy
```

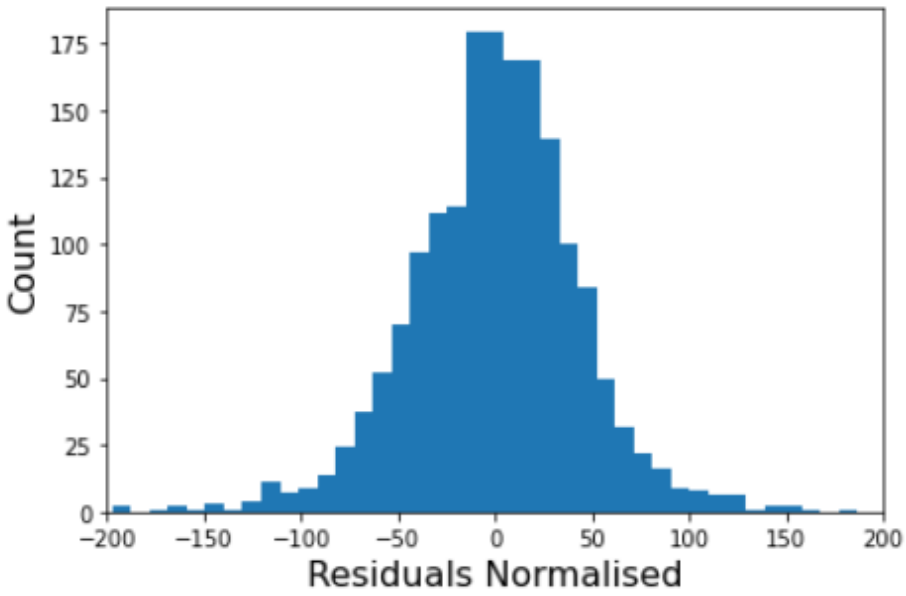
```
0.8003645918117761
```

Effectiveness: Based on the output for training and testing the statistics such as Mean error, Root mean squared error, Mean absolute error, Mean Percentage error and Mean absolute percentage error, we can infer that there is only a small difference in values. Hence, we can say

that the model is not overfit. When it comes to checking for accuracy it shows it's 80% accurate in the prediction task.

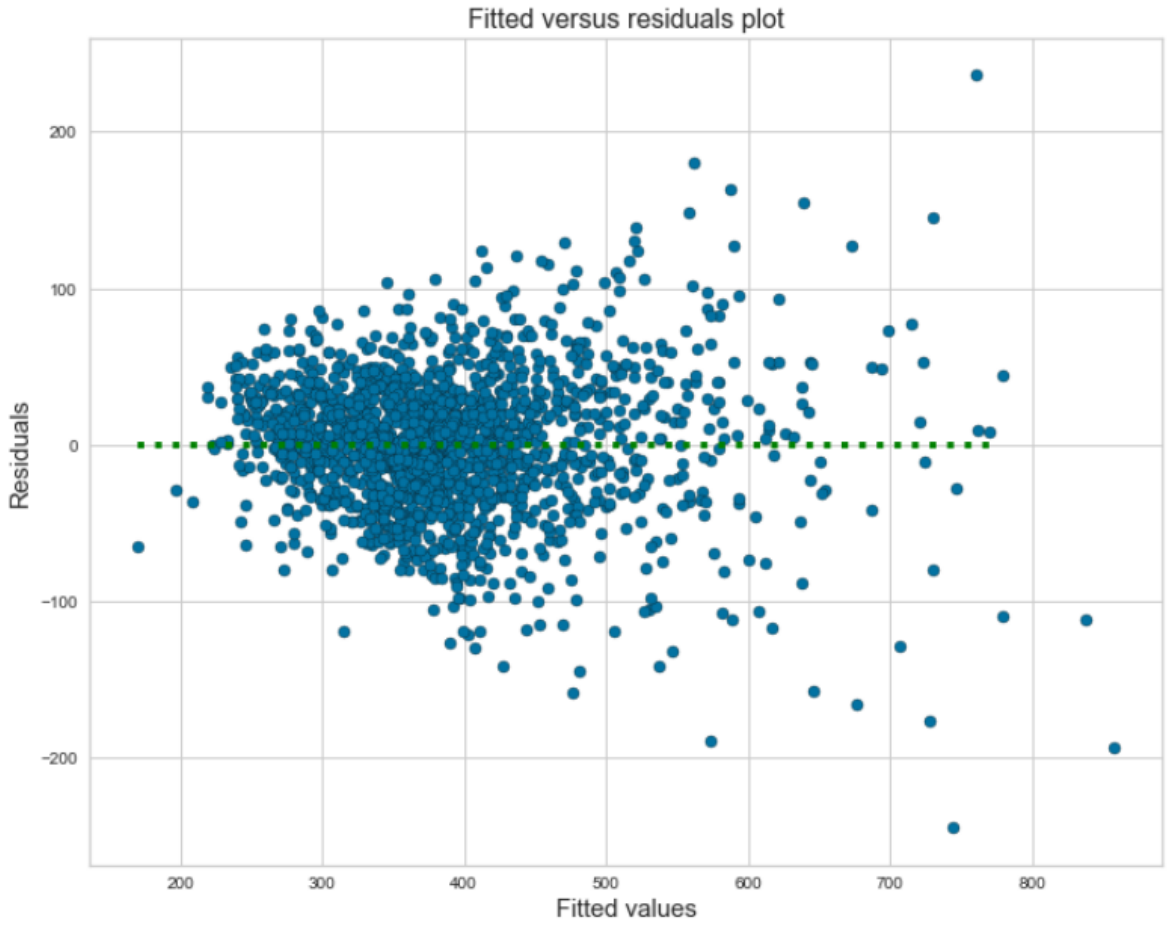
Some of the and their visualizations along with their python code and their explanation are below

```
# Checking if our residuals are normally distributed
residuals = test_y - predicted_y_test
plt.hist(residuals, bins = 50)
plt.xlim([-200,200])
plt.xlabel('Residuals Normalised',fontsize=16)
plt.ylabel('Count',fontsize=16)
plt.tight_layout()
```



This snippet of Python code plots a histogram of the residuals from a regression model, which are the discrepancies between the test set target values test_y and the projected target values predicted_y_test. Residuals calculate the difference between the actual test set target values and the predicted target values from the regression model. And then a histogram is created for these residuals.This enables us to see how the model's prediction errors are distributed.The model is functioning properly and the errors are random if the residuals have a normal distribution . Based on the above histogram we can say that the residuals are normally distributed. This is true because the histogram is evenly distributed around zero. This indicates normality. It is not perfectly bell shaped, but it is close to being symmetric. So we consider it to be normally distributed.

```
: # Fitted vs. Residuals Plot
  plt.figure(figsize=(10,10))
  p=plt.scatter(x=predicted_y_test,y=residuals,edgecolor='k')
  xmin = predicted_y_test.min()
  xmax = max(predicted_y_test)
  plt.hlines(y=0,xmin=xmin*1,xmax=xmax*0.9,color='green',linestyle=':',lw=4)
  plt.xlabel("Fitted values",fontsize=15)
  plt.ylabel("Residuals",fontsize=15)
  plt.title("Fitted versus residuals plot",fontsize=16)
  plt.grid(True)
  plt.tight_layout()
```



This code generates a fitted versus residuals plot, which allows us to visualize the relationship between the predicted target values and the residuals. The regression model is working well if the residuals have a random pattern with no obvious trend. The regression model is working well if the residuals have a random pattern as in the scatter plot above with no obvious trend.

Apart from these values and visualizations we also use Statsmodel, another python library for data analysis. Statsmodels is a helpful tool for fitting, evaluating, and comparing various regression models even if it is not a specific regression model in and of itself.

```
# train the  model
train_X = sm.add_constant(train_X)
test_X = sm.add_constant(test_X)

linear_model2 = sm.OLS(list(train_y), train_X).fit()
linear_model2.params
```

```
predicted_y_training2 = linear_model2.predict(train_X)
regressionSummary(train_y, predicted_y_training2)
```

```
Regression statistics

                    Mean Error (ME) : 0.0000
     Root Mean Squared Error (RMSE) : 44.1851
          Mean Absolute Error (MAE) : 33.5804
        Mean Percentage Error (MPE) : -1.1689
Mean Absolute Percentage Error (MAPE) : 8.7926
```

```
: # deploy the model on the test data
predicted_y_test2 = linear_model2.predict(test_X)
```

```
: # Checking the model performance in prediction
regressionSummary(test_y, predicted_y_test2)
```

```
Regression statistics

                    Mean Error (ME) : 0.3204
     Root Mean Squared Error (RMSE) : 44.2349
          Mean Absolute Error (MAE) : 33.1262
        Mean Percentage Error (MPE) : -1.0285
Mean Absolute Percentage Error (MAPE) : 8.6670
```

```
print(linear_model2.pvalues.round(4))
```

```
const              0.0000
LOT_SQFT           0.0000
YR_BUILT           0.4425
LIVING_AREA        0.0000
FLOORS             0.0000
ROOMS              0.2066
BEDROOMS           0.6328
FULL_BATH          0.0000
HALF_BATH          0.0000
KITCHEN            0.0000
FIREPLACE          0.0000
REMODEL_Old        0.0167
REMODEL_Recent     0.0000
dtype: float64
```

```
#drop the variables that are not significant (i.e., p>0.05)
train_X = train_X.drop(["YR_BUILT","ROOMS","BEDROOMS"], axis =
test_X = test_X.drop(["YR_BUILT","ROOMS","BEDROOMS"], axis = 1
```

```
linear_model3 = sm.OLS(list(train_y), train_X).fit()
predicted_y_training3 = linear_model3.predict(train_X)
regressionSummary(train_y, predicted_y_training3)
```

```
Regression statistics

                    Mean Error (ME) : 0.0000
     Root Mean Squared Error (RMSE) : 44.1958
          Mean Absolute Error (MAE) : 33.6046
        Mean Percentage Error (MPE) : -1.1717
Mean Absolute Percentage Error (MAPE) : 8.7989
```

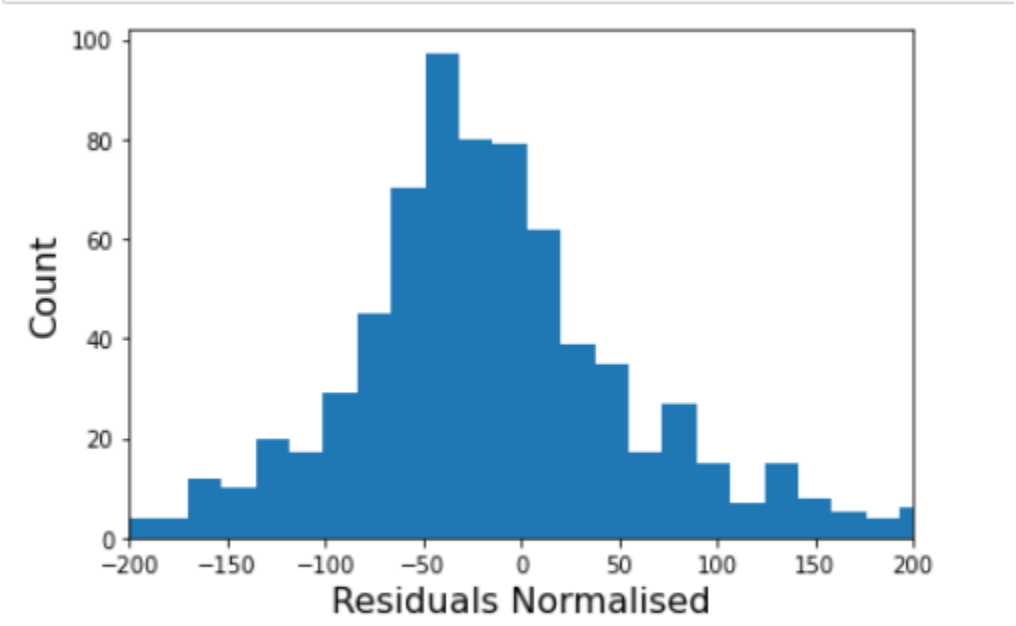
```
predicted_y_test3 = linear_model3.predict(test_X)
regressionSummary(test_y, predicted_y_test3)
```

```
Regression statistics

                    Mean Error (ME) : 0.3043
     Root Mean Squared Error (RMSE) : 44.2694
          Mean Absolute Error (MAE) : 33.1459
        Mean Percentage Error (MPE) : -1.0353
Mean Absolute Percentage Error (MAPE) : 8.6718
```

```
# Checking for accuracy
accuracy = linear_model3.rsquared
accuracy
```

```
0.7997408905769834
```

We use the Statsmodel library to compare the metrics. For those features with p value greater than 0.05 we drop them and compare the metrics. We see a similar trend when used without Statsmodel. And accuracy is approximately the same ~80%, which is what we got before applying Statsmodel. And this is expected.



The residual distribution is also similar.

Therefore, we can say that Linear Regression model is one appropriate model for the WestRoxbury dataset.

2.Decision tree

One of the most understandable and logical techniques for data regression is the decision tree. Using a set of decision rules that are based on the feature values, the main concept is to recursively divide the data into smaller and smaller subsets.. At each split, the goal is to find the feature and threshold that result in the largest reduction in the variance of the target variable. Usually, the mean squared error (MSE) or another metric is used to calculate this.Following training, the tree is traversed from root to leaf node to provide a prediction for a new input. Each node's decision rules are used to choose which branch to take.

Justification: Since Decision tree is another effective approach for predicting continuous numerical values, it is frequently used to solve regression issues. Decision trees can handle nonlinear relationships between the target variable (in this case, the total value) and the predictor variables.Decision trees are able to record interactions between predictor variables, such as the relationship between the age of the property and the number of rooms. Understanding how many factors affect the cost of homes in the West Roxbury neighborhood can be aided by knowing this.Decision trees provide a transparent and interpretable model that allows us to understand how different predictors contribute to the final predictions. Therefore, using Decision trees on the WestRoxbury dataset is a valid decision.

Tune/train: They can be divided into a few steps as below.

1.Since pre-processing is already done in phase 1 we now divide the data into training and test sets.

2. Train the Decision model.

3. Then we evaluate the model with certain evaluation metrics like mean squared error, root mean squared error. Also we then find the accuracy to determine how well our model is performed.

These steps are shown in the code below

```
# partition data into train and test sets
X_prediction = predictors_df1
y_prediction = response_df
train_X_prediction, test_X_prediction, train_y_prediction, test_y_prediction = train_test_split(X_prediction,
                                                                                                 y_prediction, test_size=0.3, random_state=616)
```

```
# normalise using Z-score
z_score_norm2 = preprocessing.StandardScaler()
z_score_norm2.fit(predictors_df1)

train_X_prediction = pd.DataFrame(z_score_norm2.transform(train_X_prediction),
                                  columns = predictors_df1.columns)
test_X_prediction = pd.DataFrame(z_score_norm2.transform(test_X_prediction),
                                  columns = predictors_df1.columns)
```

```
# train the Decision tree model
WestRoxbury = DecisionTreeRegressor(max_depth=5, random_state=13, splitter="best").fit(train_X_prediction, train_y_prediction) #r
predicted_y_training_Rox = WestRoxbury.predict(train_X_prediction)
```

Effectiveness: Based on the output for training and testing the statistics such as Root Mean Squared Error, we can infer that there is only a small difference in values. Hence, we can say that the model is not overfit. The root mean squared error is more compared to Linear Regression. When it comes to checking for accuracy it shows it's 75% accurate in the prediction task, which corresponds to the increase in error.

```
print("Root Mean Squared Error (RMSE): ", round(mean_squared_error(train_y_prediction, predicted_y_training_Rox) ** 0.5, 4))
```

```
Root Mean Squared Error (RMSE):  46.7916
```

```
predicted_y_test_Rox = WestRoxbury.predict(test_X_prediction)
```

```
print("Root Mean Squared Error (RMSE): ", round(mean_squared_error(test_y_prediction, predicted_y_test_Rox) ** 0.5, 4))
```

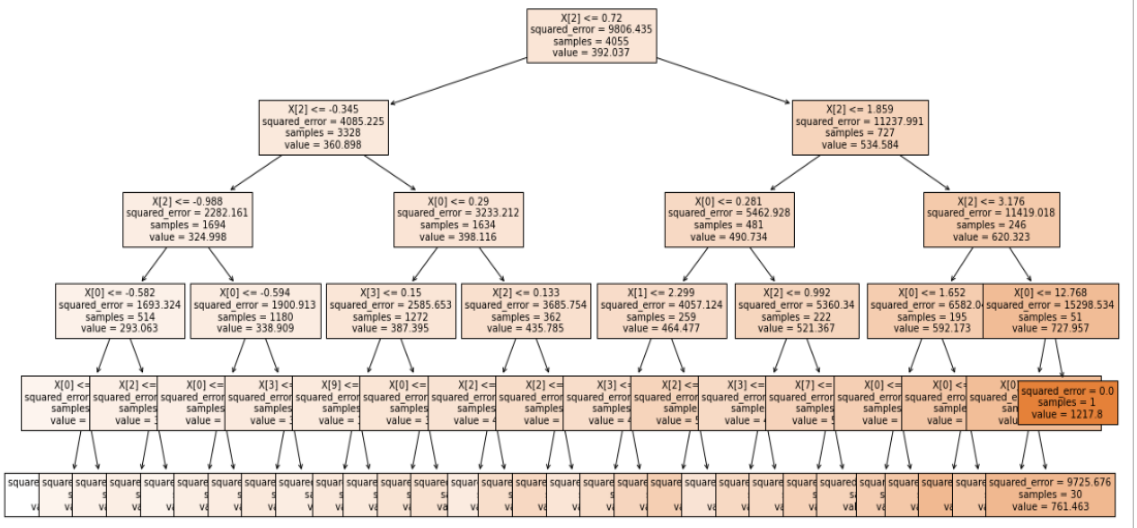
```
Root Mean Squared Error (RMSE):  49.7434
```

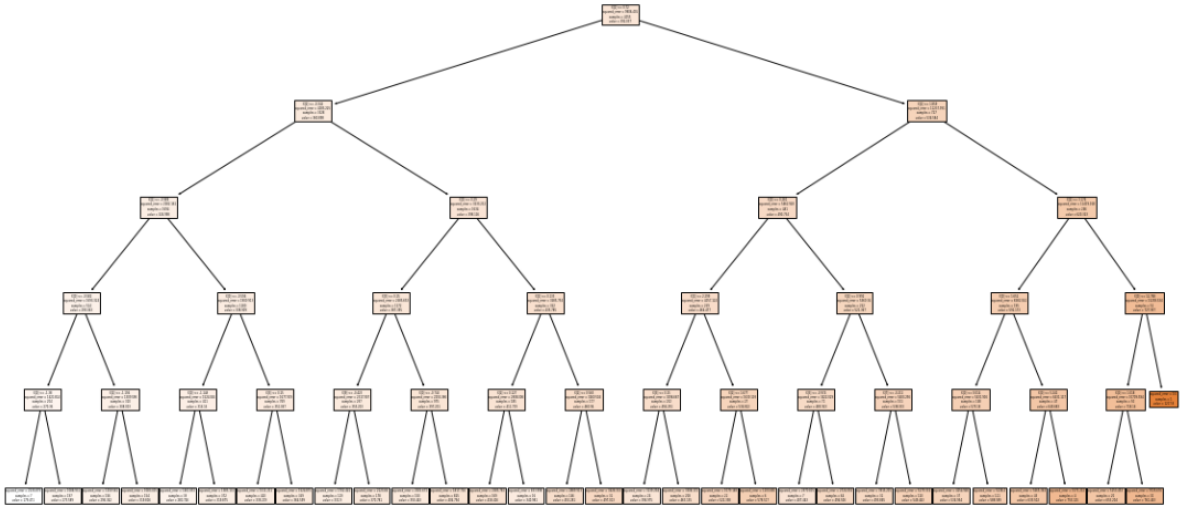
```
r2 = r2_score(test_y_prediction ,predicted_y_test_Rox)
r2
```

```
0.7443605064265439
```

Some of the and their visualizations along with their python code and their explanation are below

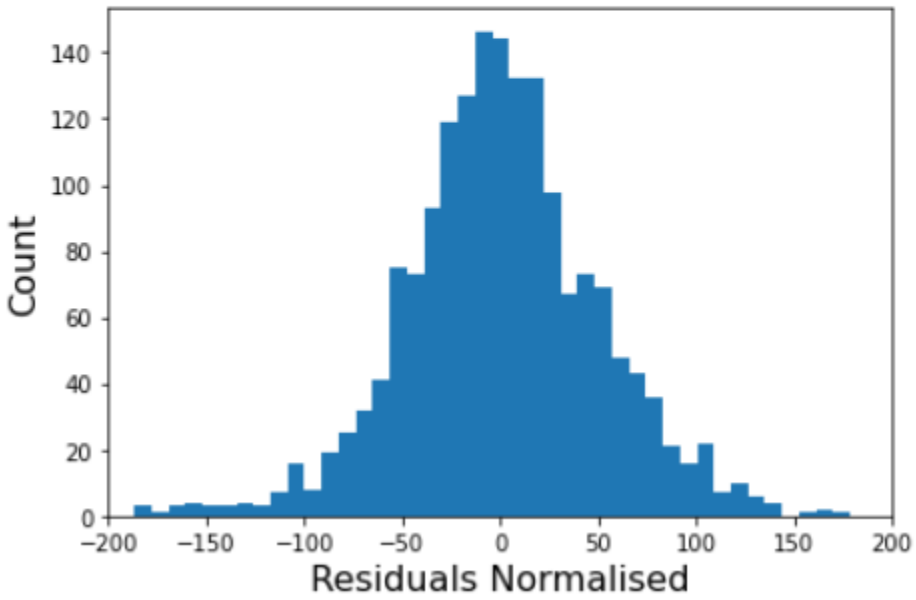
```
In [108]: # Decision tree for our dataset
          plt.figure(figsize=(20,10))
          plot_tree(WestRoxbury, filled=True,fontsize=10)
          plt.savefig('Decision_tree.png', dpi=300)
          plt.show()
```





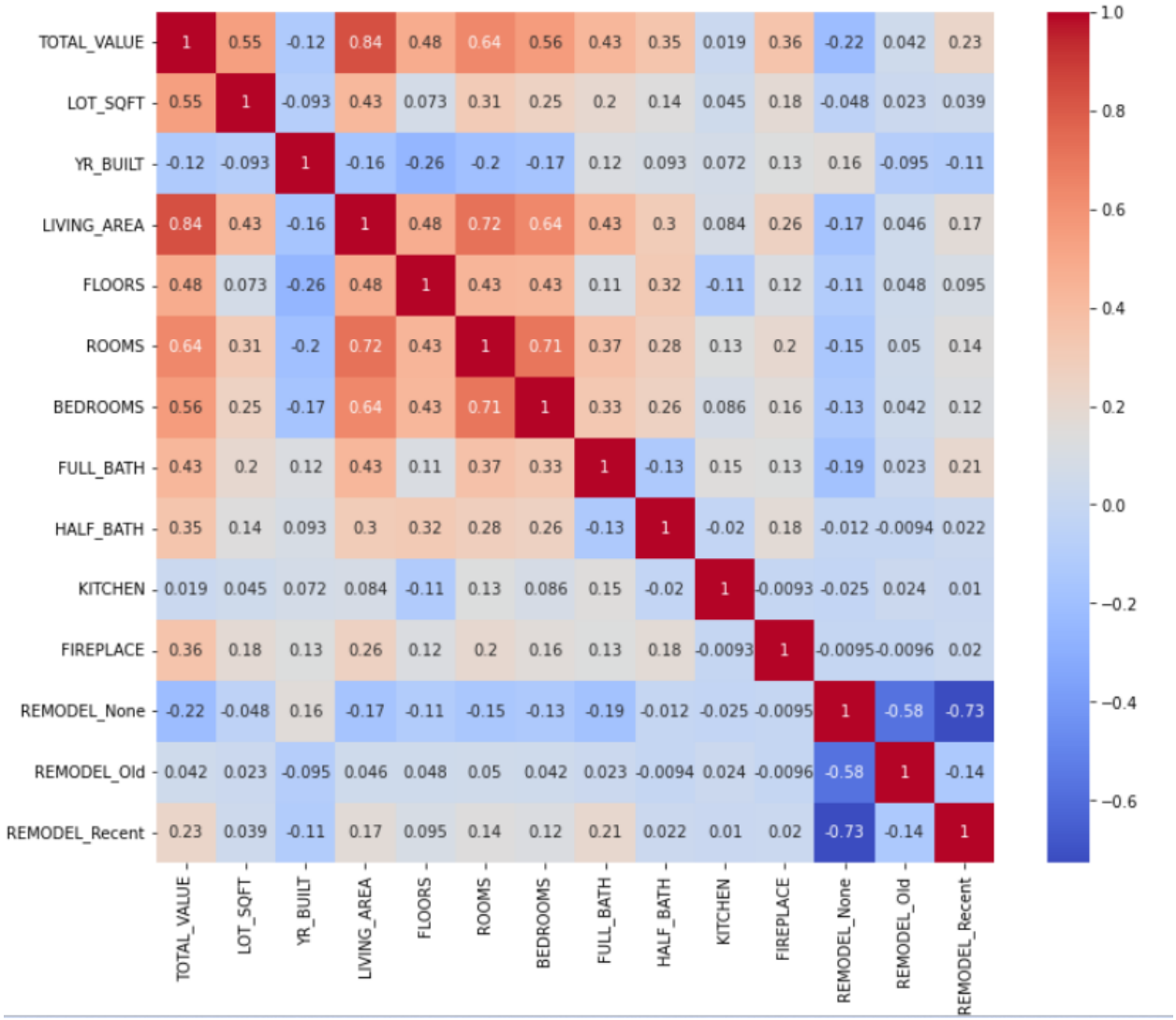
This code is using the plot_tree() function from the sklearn.tree module to plot a decision tree for the WestRoxbury dataset.With the help of this code, you can visualize the decision tree that was created using the data, complete with leaf node predictions for the target variable and splitting criteria at each node. The png file is attached in the folder along with this report to look through in more detail.

```
# Checking if our residuals are normally distributed
residuals = test_y_prediction - predicted_y_test_Rox
plt.hist(residuals, bins = 50)
plt.xlim([-200,200])
plt.xlabel('Residuals Normalised',fontsize=16)
plt.ylabel('Count',fontsize=16)
plt.tight_layout()
```



This snippet of Python code plots a histogram of the residuals from a Decision tree model, which are the discrepancies between the test set target values test_y and the projected target values predicted_y_test. Residuals calculate the difference between the actual test set target values and the predicted target values from the regression model. And then a histogram is created for these residuals. Based on the above histogram we can say that the residuals are normally distributed. This is true because the histogram is evenly distributed around zero. It is not perfectly bell shaped, but it is close to being symmetric. So we consider it to be normally distributed.

```
# Confusion matrix for our dataset
import seaborn as sns
plt.figure(figsize=(12,10))
sns.heatmap(ds.corr(),cmap='coolwarm',annot=True)
```



The values show that the model fits well on our data and gives balanced results. The correlation between each pair of numerical features in the dataset is shown on a heatmap. Overall the decision tree regressor is performing less compared to the Linear Regression model.

3. Random Forest Regressor

Random Forest Regressor is a machine learning algorithm that builds a collection of decision trees to perform regression analysis. It works by creating a forest of decision trees, where each tree is trained on a randomly selected subset of the training data and a random subset of the features. During prediction, the output of each tree is averaged to obtain a final prediction. This model has several advantages over other methods, such as high accuracy, low overfitting, and the ability to handle large datasets with many features.

Justification: Random Forest Regressor can automatically detect the most important features for predicting the target variable, which can help to understand the factors that influence the total price of the properties.
Also, this model is less sensitive to highly correlated features as it uses a collection of decision trees, each of which is trained on a random subset of features and training data.This allows the model to capture the complex relationships between the features and the target variable while avoiding overfitting or relying too heavily on any single feature. Lastly, Random Forest Regressor can automatically detect the most important features for predicting the target variable. This is useful for understanding the factors that influence the total price of the properties in West Roxbury. By analyzing the feature importance scores generated by the model, we can identify the most significant factors (e.g., square footage, number of bedrooms) and potentially use this information to guide decision-making or further analysis.

Tune/train: They can be divided into a few steps as below.

1.Since pre-processing is already done in phase 1 we now divide the data into training and test sets.
2. Train the Random Forest Regressor model.
3. Extract the features of importance and sort the features in their order of importance.
4. Then we evaluate the model with certain evaluation metrics like mean squared error, root mean squared error. Also we then find the accuracy to determine how well our model is performed.

These steps are shown in the code below

In [112]:
```
# Import libraries
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
```

In [115]:
```
# Loading dataset and split it into training and testing sets
X_prediction = predictors_df1
y_prediction = response_df
train_X_prediction, test_X_prediction, train_y_prediction, test_y_prediction = train_test_split(X_prediction,
                                                                                                 y_prediction, test_size=0.3, random_state=42)
```

In [116]:
```
# Train the model
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(train_X_prediction, train_y_prediction)
```

Out[116]:
```
RandomForestRegressor(random_state=42)
```

In [117]:
```
# Extract feature of importances
importances = model.feature_importances_
```

In [118]:
```
# Sort feature importances in descending order
indices = np.argsort(importances)[::-1]
```

Effectiveness:
Based on the output for training and testing the statistics such as Root Mean Squared Error, we can infer that there is only a small difference in values. Hence, we can say that the model is not overfit. The root mean squared error is less compared to Linear Regression. When it comes to

checking for accuracy it shows it's 81.5% accurate in the prediction task, which corresponds to the decrease in error.

```
In [121]: predicted_y_training_corollas = model.predict(train_X_prediction)

In [122]: print("Root Mean Squared Error (RMSE): ", round(mean_squared_error(train_y_prediction, predicted_y_training_corollas) ** 0.5, 4))

          Root Mean Squared Error (RMSE):  15.7331

In [123]: # Check R^2 of our test data
          r2 = r2_score(test_y_prediction, y_pred_rf)
          r2

Out[123]: 0.8153383929101298
```

```
# Evaluate the model
mse = mean_squared_error(test_y_prediction, y_pred_rf)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 1931.1310401432556
```

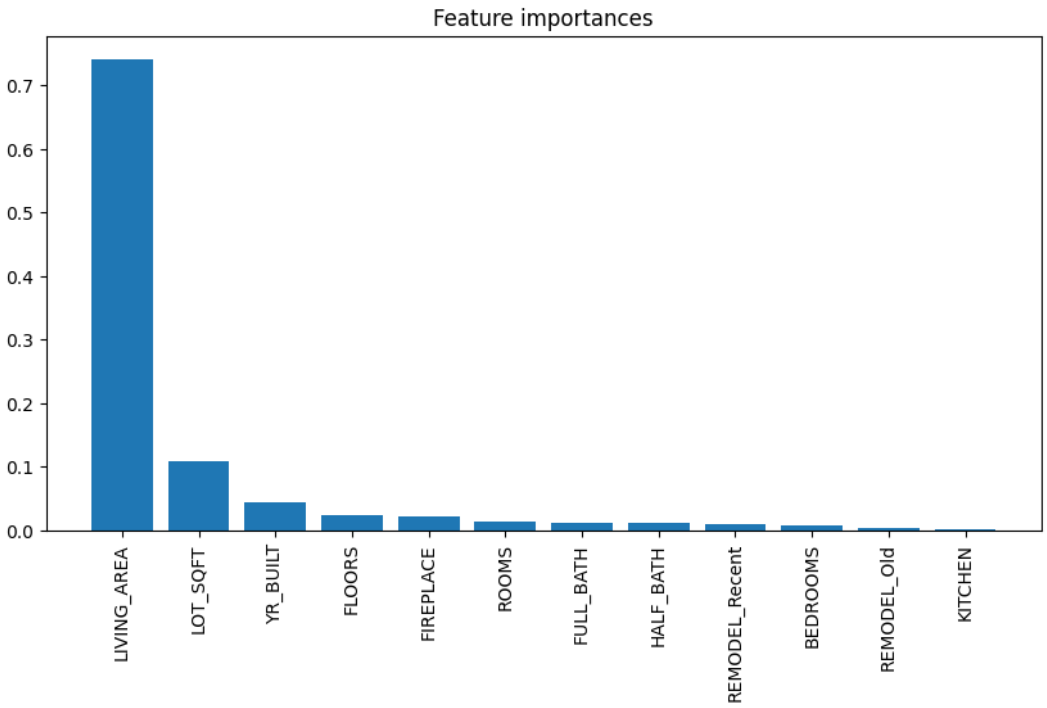
```
print("Root Mean Squared Error (RMSE): ", round(math.sqrt(mse), 4))
```

```
Root Mean Squared Error (RMSE):  43.9446
```

Some of the and their visualizations along with their python code and their explanation are below

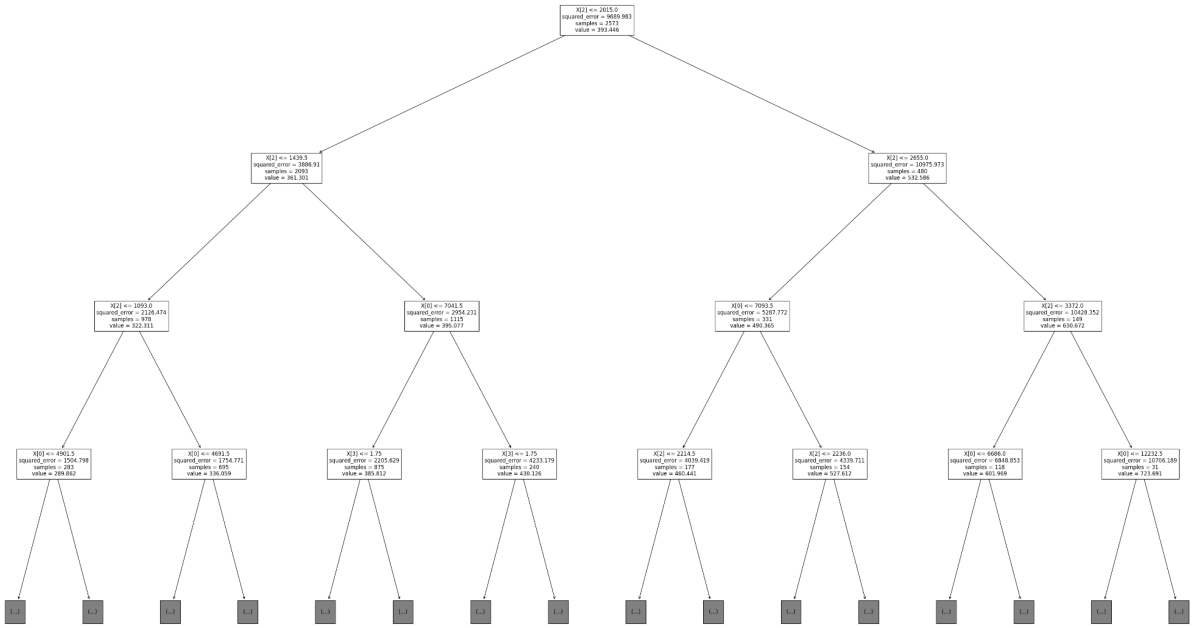
In a Random Forest regressor model, feature importance is usually calculated using the Gini impurity or Mean Decrease Impurity method. These methods measure the total reduction in impurity (or variance) achieved by splitting on a given feature over all trees in the forest.
The resulting feature importance scores can be plotted as a bar chart, where the height of each bar represents the importance of the corresponding feature. The bars are typically sorted in descending order, so that the most important feature is plotted first.
The feature importance bar plot provides a quick and easy way to identify which features are the most important predictors in the model. So in the figure, we can see that Living_Area is the most important predictor identified by the model.

```
In [126]: # Plot the feature importances using a bar plot
          plt.figure(figsize=(10,5))
          plt.title("Feature importances")
          plt.bar(range(train_X_prediction.shape[1]), importances[indices])
          plt.xticks(range(train_X_prediction.shape[1]), train_X_prediction.columns[indices], rotation=90)
          plt.show()
```



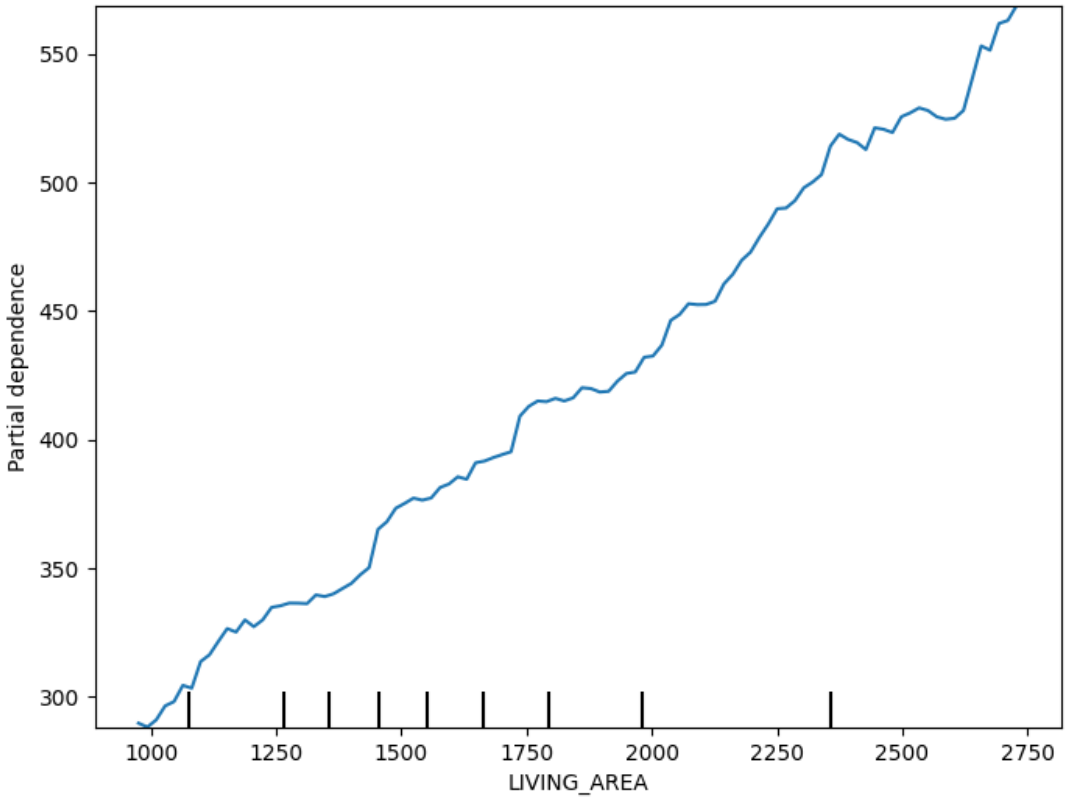
The decision tree plot shows the individual trees in the forest and how they are connected. Each node in the tree represents a decision based on the value of a feature, and the branches represent the possible outcomes of that decision. Additionally, the decision tree plot can be useful for communicating the model's predictions to non-technical audiences, as it provides a visual representation of the decision-making process that can be easily understood.
Since our dataset involves many features, we have plotted the tree until depth of 3.

In [127]:
```
# Plot the first decision tree
fig, ax = plt.subplots(figsize=(50, 30))
plot_tree(model.estimators_[0], max_depth=3, ax=ax)
plt.show()
```



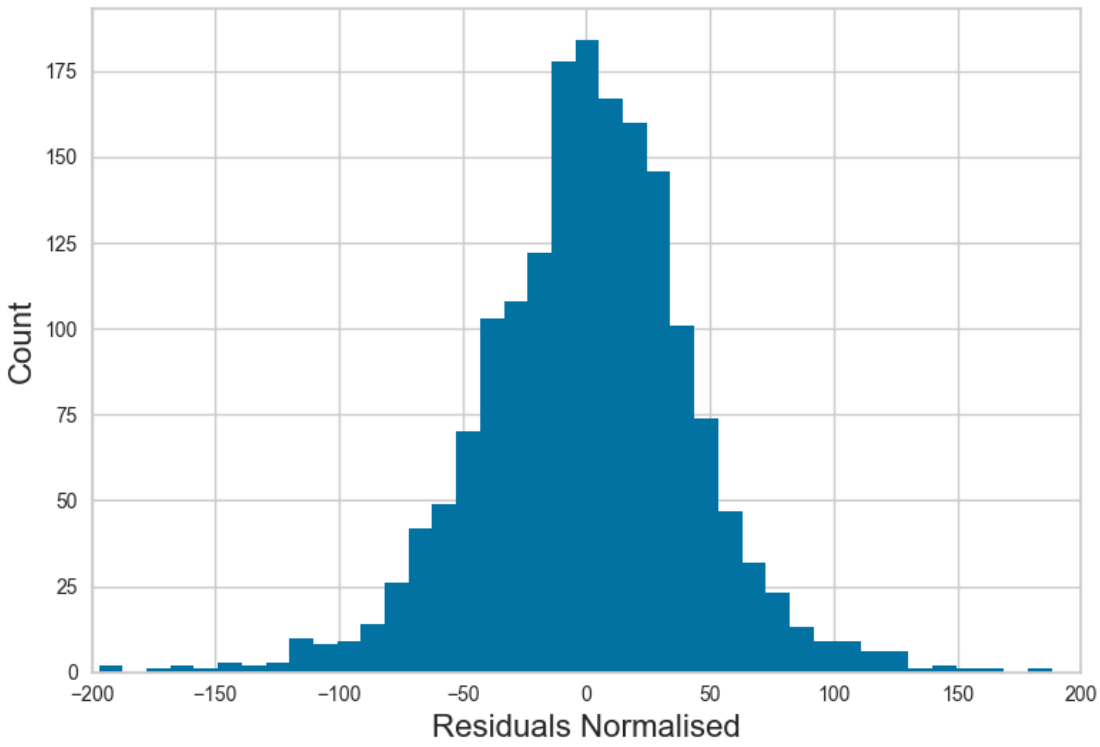
Next, we see a plot of partial dependence for a specific feature in a Random Forest regressor which shows the relationship between that feature and the predicted target variable, while holding all other features constant at their average or median values. The target feature - Living_Area, is on the x-axis and the predicted target variable on the y-axis. The partial dependence plot allows you to see how the target feature affects the predicted target variable, independent of the other features in the model. It can be useful for identifying non-linear relationships between features and the target variable, and for understanding how changes in the target feature value affect the predicted target variable.
In the plot obtained, we see how changes in the Living_Area affect the total price, while holding other features as constant. It can be identified that the change is linear, almost, more the living area, more is the price of the property.

```
from sklearn.inspection import plot_partial_dependence
# Plot partial dependence for a specific feature
fig, ax = plt.subplots(figsize=(8, 6))
plot_partial_dependence(model, train_X_prediction, ['LIVING_AREA'], ax=ax)
plt.show()
```



Lastly, for this model we plot a histogram of the residuals, which are the discrepancies between the test set target values test_y and the projected target values predicted_y_test. Residuals calculate the difference between the actual test set target values and the predicted target values from the regressor model. And then a histogram is created for these residuals. Based on the above histogram we can say that the residuals are normally distributed. This is true because the histogram is evenly distributed around zero. It is not perfectly bell shaped, but it is close to being symmetric. So we consider it to be normally distributed.

In [192]:
```
residuals = test_y - predicted_y_test
plt.hist(residuals, bins = 50)
plt.xlim([-200,200])
plt.xlabel('Residuals Normalised',fontsize=16)
plt.ylabel('Count',fontsize=16)
plt.tight_layout()
plt.show()
```



4. Support Vector Regression

Support Vector Regression (SVR) with a linear kernel is a type of regression analysis based on the Support Vector Machines (SVM) algorithm that uses a linear kernel function to transform the input data into a higher-dimensional space. The goal is to find a hyperplane that fits the data well and separates the data into two classes: those that are within the margin (support vectors) and those that are outside the margin. The hyperplane is defined by a set of weights (w) and a bias term (b) and is given by the equation: $y(x) = w^T x + b$
where $y(x)$ is the predicted output for the input vector x.
SVR with a linear kernel is a powerful technique for regression analysis that can handle linear data distributions efficiently. Its ability to handle linear data makes it particularly useful in cases where traditional linear regression models would not be effective.

Justification: Since the West Roxbury dataset includes several variables that may affect the total price of the property, SVR with a linear kernel can effectively handle this high-dimensional input space.
This model can capture linear relationships between the input variables and the target variable (i.e., the total price of the property). This is important because the total price of a property is often influenced by factors such as the number of rooms and the number of kitchens, etc, which may have a linear relationship with the sale price. Additionally, SVR with a linear kernel is a computationally efficient algorithm, which makes it a good choice for large datasets. The West Roxbury dataset contains information about over 1,000 properties, so using a computationally efficient algorithm can significantly reduce the training time required to fit the model.

Tune/train: They can be divided into a few steps as below.

1.Since pre-processing is already done in phase 1 we now divide the data into training and test sets.

2. Train the SVR model with a linear kernel.

3. Then we evaluate the model with certain evaluation metrics like mean squared error, root mean squared error. Also we then find the accuracy to determine how well our model is performed.

These steps are shown in the code below

In [188]:
```
# Import libraries
import pandas as pd
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error
```

In [131]:
```
# Load your dataset and split it into training and testing sets
X_prediction = predictors_df1
y_prediction = response_df
train_X_prediction, test_X_prediction, train_y_prediction, test_y_prediction = train_test_split(X_prediction,
                                                                y_prediction, test_size=0.3, random_state=42)
```

In [132]:
```
# Train the model
model = SVR(kernel='linear')
model.fit(train_X_prediction, train_y_prediction)
```

Out[132]:
```
SVR
SVR(kernel='linear')
```

In [133]:
```
# Make predictions
y_pred_svr = model.predict(test_X_prediction)
```

Effectiveness:

Based on the output for training and testing the statistics such as Root Mean Squared Error, we can infer that there is only a small difference in values. Hence, we can say that the model is not overfit. The root mean squared error is more compared to Random Forest Regressor. When it comes to checking for accuracy it shows it's 77.7% accurate in the prediction task, which corresponds to the increase in error.

In [135]:
```
predicted_y_training_corollas = model.predict(train_X_prediction)
```

In [136]:
```
print("Root Mean Squared Error (RMSE): ", round(mean_squared_error(train_y_prediction, predicted_y_training_corollas) ** 0.5, 4))
```

```
Root Mean Squared Error (RMSE):  46.345
```

In [137]:
```
# Check R^2 of our test data
model.score(test_X_prediction,test_y_prediction)
```

Out[137]:
```
0.7771814699541739
```

```
: # Evaluate the model
mse = mean_squared_error(test_y_prediction, y_pred_svr)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 2330.163732849866
```

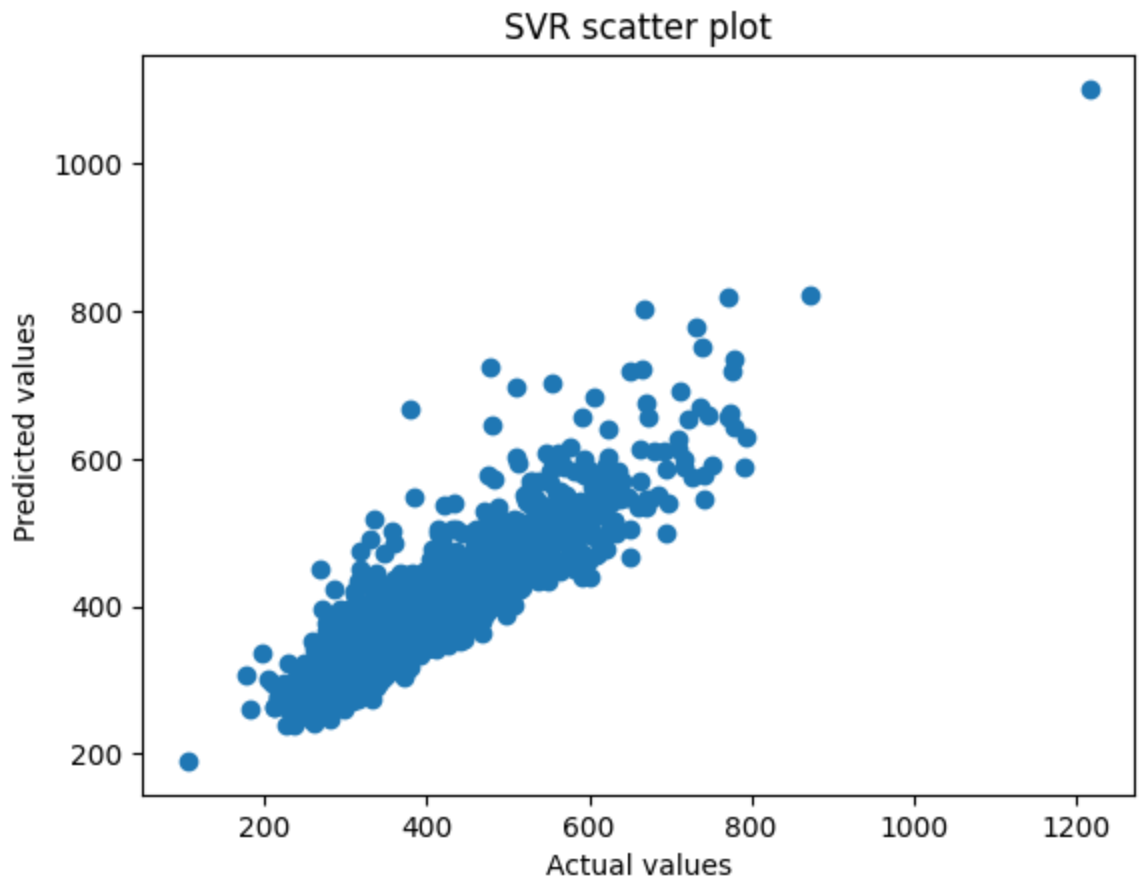
```
: print("Root Mean Squared Error (RMSE): ", round(math.sqrt(mse), 4))
```

```
Root Mean Squared Error (RMSE):  48.2718
```

Some of the and their visualizations along with their python code and their explanation are below

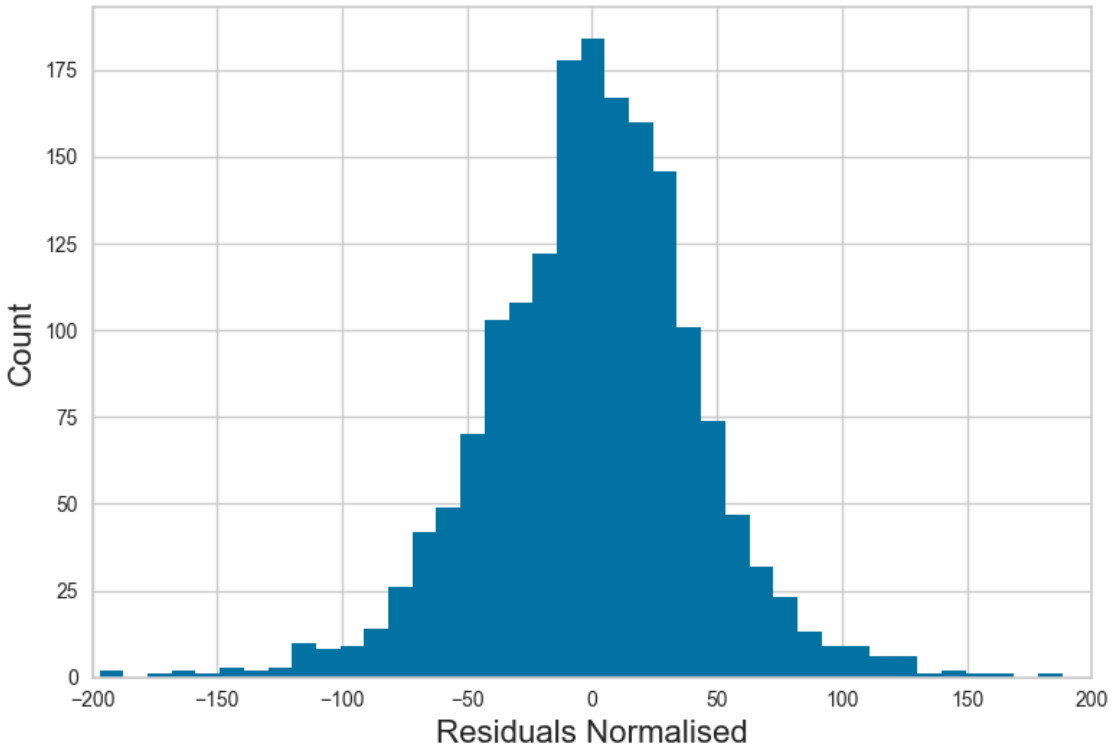
A scatter plot of SVR (Support Vector Regression) is a visualization tool that can be used to understand the relationship between the input features and the target variable in a regression problem. In a scatter plot of SVR, the x-axis is representing actual values, and the y-axis represents the predicted values.
We see that the points are tightly clustered around the diagonal line, which means that the model is performing well.

```
In [138]: import matplotlib.pyplot as plt

plt.scatter(test_y_prediction, y_pred_svr)
plt.xlabel('Actual values')
plt.ylabel('Predicted values')
plt.title('SVR scatter plot')
plt.show()
```



Lastly, for this model we plot a histogram of the residuals, which are the discrepancies between the test set target values test_y and the projected target values predicted_y_test. Residuals calculate the difference between the actual test set target values and the predicted target values from the SVR model. And then a histogram is created for these residuals. Based on the above histogram we can say that the residuals are normally distributed. This is true because the histogram is evenly distributed around zero. It is not perfectly bell shaped, but it is close to being symmetric. So we consider it to be normally distributed.

In [208]:
```
residuals = test_y - predicted_y_test
plt.hist(residuals, bins = 50)
plt.xlim([-200,200])
plt.xlabel('Residuals Normalised',fontsize=16)
plt.ylabel('Count',fontsize=16)
plt.tight_layout()
plt.show()
```



5. K-nearest Neighbors

K-Nearest Neighbors (KNN) is a simple yet powerful machine learning algorithm that can be used for both classification and regression tasks. The basic idea behind the KNN algorithm is to classify or predict new data points based on their proximity to other known data points in the training set. The choice of K would be an important hyperparameter to tune, as it would affect the bias-variance trade-off of the model. A larger K would result in a smoother decision boundary, but it may also lead to misclassifications due to the inclusion of irrelevant data points. On the other hand, a smaller K may lead to a more complex decision boundary that better fits the training data, but may also be more sensitive to noise and outliers.

Justification:
KNN is a suitable algorithm to use for this task because it can learn complex nonlinear relationships between the features and the target variable, and it does not make any assumptions about the distribution of the data. We have chosen the value of K as 10. We experimented with values from 1 to 50 and we obtained more accuracy when k = 10.

Tune/train: They can be divided into a few steps as below.

1.Since pre-processing is already done in phase 1 we now divide the data into training and test sets.

2. Train the KNN model.

3. Then we evaluate the model with certain evaluation metrics like mean squared error, root mean squared error. Also we then find the accuracy to determine how well our model is performed.

These steps are shown in the code below

```
In [245]: # Import libraries
          import pandas as pd
          from sklearn.neighbors import KNeighborsRegressor
          from sklearn.metrics import mean_squared_error

In [246]: # Load your dataset and split it into training and testing sets
          X_prediction = predictors_df1
          y_prediction = response_df
          train_X_prediction, test_X_prediction, train_y_prediction, test_y_prediction = train_test_split(X_prediction,
                                                                                                           y_prediction, test_size=0.3, random_state=42)

In [247]: # Train the model
          model = KNeighborsRegressor(n_neighbors=10)
          model.fit(train_X_prediction, train_y_prediction)

Out[247]: KNeighborsRegressor
          KNeighborsRegressor(n_neighbors=10)

In [248]: # Make predictions
          y_pred_knn = model.predict(test_X_prediction)
```

Effectiveness:

Based on the output for training and testing the statistics such as Root Mean Squared Error, we can infer that there is only a small difference in values. Hence, we can say that the model is not overfit. The root mean squared error is more compared to Random Forest Regressor. When it comes to checking for accuracy it shows it's 72.17% accurate in the prediction task, which corresponds to the increase in error.

```
In [250]: predicted_y_training_corollas = model.predict(train_X_prediction)

In [251]: print("Root Mean Squared Error (RMSE): ", round(mean_squared_error(train_y_prediction, predicted_y_training_corollas) ** 0.5, 4))

          Root Mean Squared Error (RMSE):  45.9932

In [252]: # Check R^2 of our test data
          model.score(test_X_prediction,test_y_prediction)

Out[252]: 0.7217184636583985
```

```
# Evaluate the model
mse = mean_squared_error(test_y_prediction, y_pred_knn)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 2910.1778176688435
```

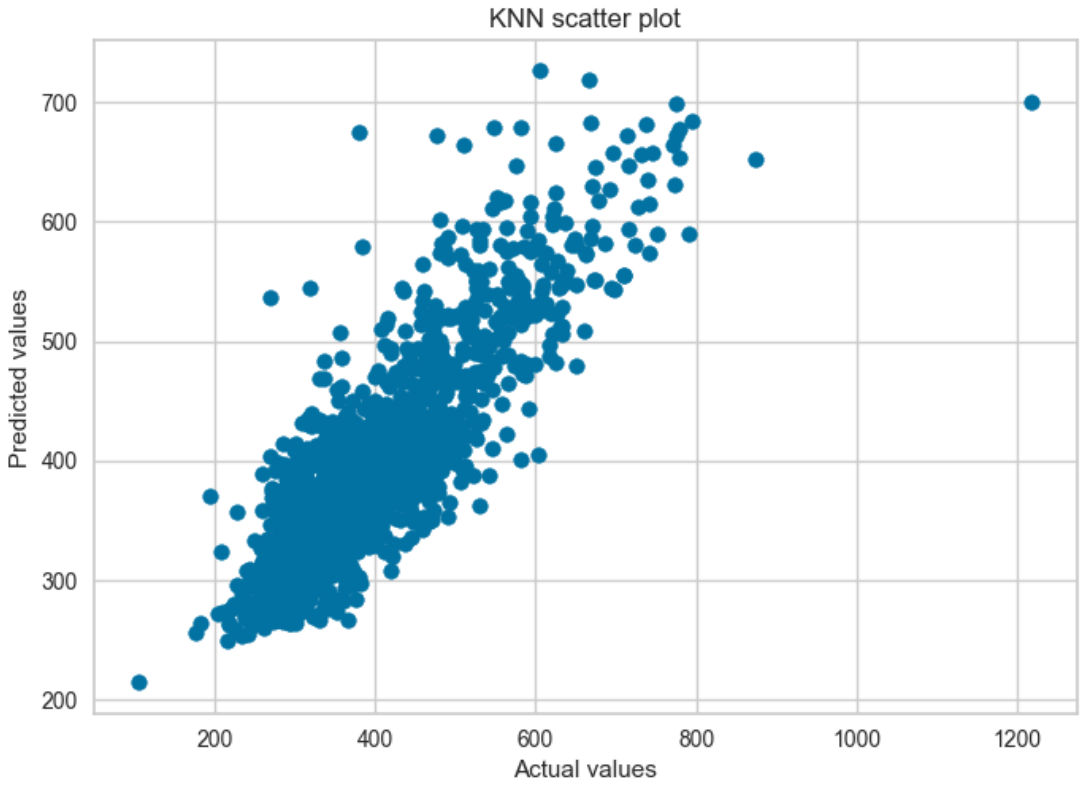
```
print("Root Mean Squared Error (RMSE): ", round(math.sqrt(mse), 4))
```

```
Root Mean Squared Error (RMSE):  53.9461
```

Some of the and their visualizations along with their python code and their explanation are below

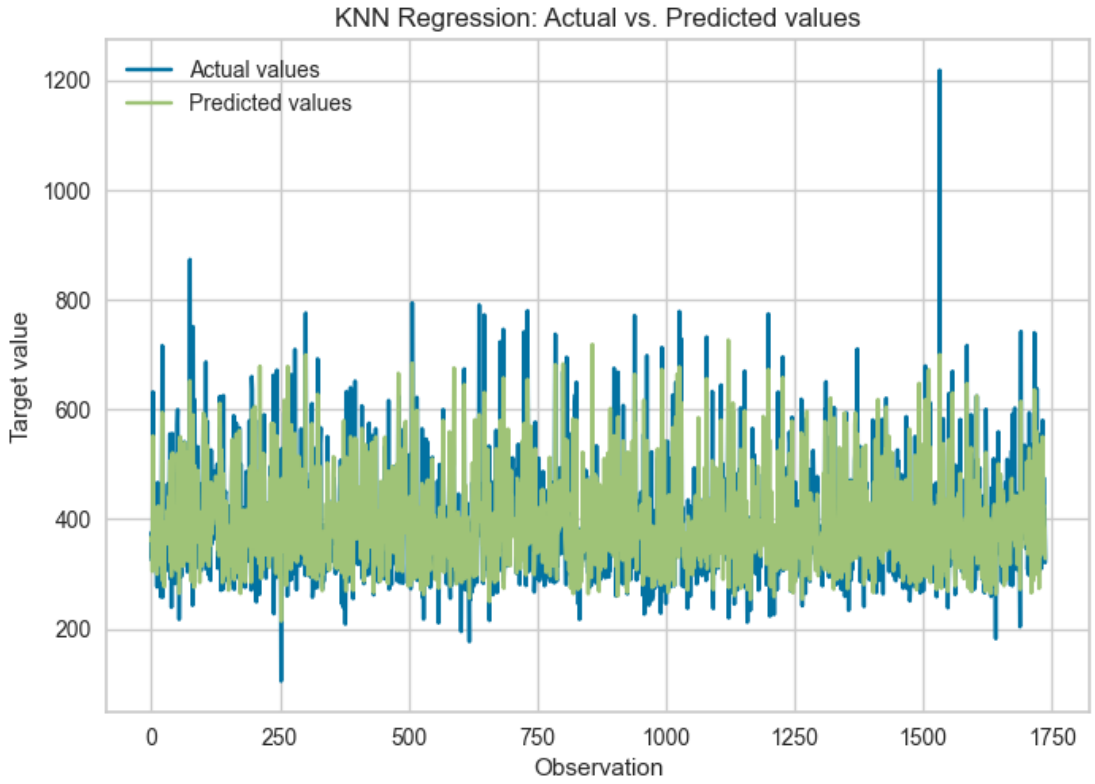
The scatter plot for KNN can help to visualize how the algorithm is making its classifications. In the plot, each data point is plotted on a 2-D graph where x-axis represents actual values and y-axis represents predicted values.We see that the data points are well-separated in scatter plot, meaning that the algorithm is likely to make accurate classifications.

```
In [148]: import matplotlib.pyplot as plt

          plt.scatter(test_y_prediction, y_pred_knn)
          plt.xlabel('Actual values')
          plt.ylabel('Predicted values')
          plt.title('KNN scatter plot')
          plt.show()
```



In the next plot, we visualize the performance of the regression model.The blue line in the plot represents the actual target values, while the orange line represents the predicted target values. By comparing the two lines, we can see how well the regression model is performing. We see that the predicted values are very close to the actual values, resulting in overlapping with each other.

```
In [149]: import matplotlib.pyplot as plt

plt.plot(test_y_prediction.values.ravel(), label='Actual values')
plt.plot(y_pred_knn.ravel(), label='Predicted values')
plt.title('KNN Regression: Actual vs. Predicted values')
plt.xlabel('Observation')
plt.ylabel('Target value')
plt.legend()
plt.show()
```



6. Gradient Boosting Regression

Gradient boosting is one of the most popular machine learning algorithms for tabular datasets. One of the ensemble technique variations that uses numerous weak models combined for greater overall performance is known as gradient boosting.

Justification: Gradient boosting works by iteratively updating the model parameters in the direction of the steepest descent of the cost function, which measures the difference between the model predictions and the actual target values.Other regression procedures that rely on matrix inversion, like the normal equation or Ridge regression, can be computationally expensive to utilize because of the WestRoxbury dataset's huge number of samples and attributes.Gradient descent, on the other hand, can scale well to large datasets by processing the data in mini-batches or using stochastic gradient descent.The mean squared error (MSE) or other cost functions that are unaffected by a few extreme values are minimized through gradient

descent regression.This can be particularly useful in the WestRoxbury dataset, which may contain some extreme values due to factors such as high property values or demographic characteristics.Therefore, using Gradient Boosting on the WestRoxbury dataset is a valid decision.

Tune/train: They can be divided into a few steps as below.

1.Since pre-processing is already done in phase 1 we now divide the data into training and test sets.

2. Train the Gradient boost model.

3. Then we evaluate the model with certain evaluation metrics like root mean squared error. Also we then find the accuracy to determine how well our model is performed.

These steps are shown in the code below

```
# Dividing the data
X = predictor_df_normalized
y = response_df
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.3, random_state=1)
```

```
GB = GradientBoostingRegressor(random_state=616).fit(train_X, train_y)
predicted_y_training = GB.predict(train_X)
```

```
predicted_y_test = GB.predict(test_X)
```

Effectiveness: The root mean squared error is decreased to a larger extent when compared with other models. This in turn shows the accuracy. It is the highest with 83% accuracy. The same details are shown below.There is not much difference in the root mean squared error when compared to training and testing, hence we can say it is definitely not overfit. It is a best fit model.Again since all the pre-processing steps along with normalization being applied to the data we have an effective model as a result. Overall, the WestRoxbury dataset may be modeled using gradient descent regression, which is a powerful and adaptable approach.

```
print("Root Mean Squared Error (RMSE): ", round(mean_squared_error(train_y, predicted_y_training) ** 0.5, 4))
```

```
Root Mean Squared Error (RMSE):  35.9822
```

```
print("Root Mean Squared Error (RMSE): ", round(mean_squared_error(test_y, predicted_y_test) ** 0.5, 4))
```

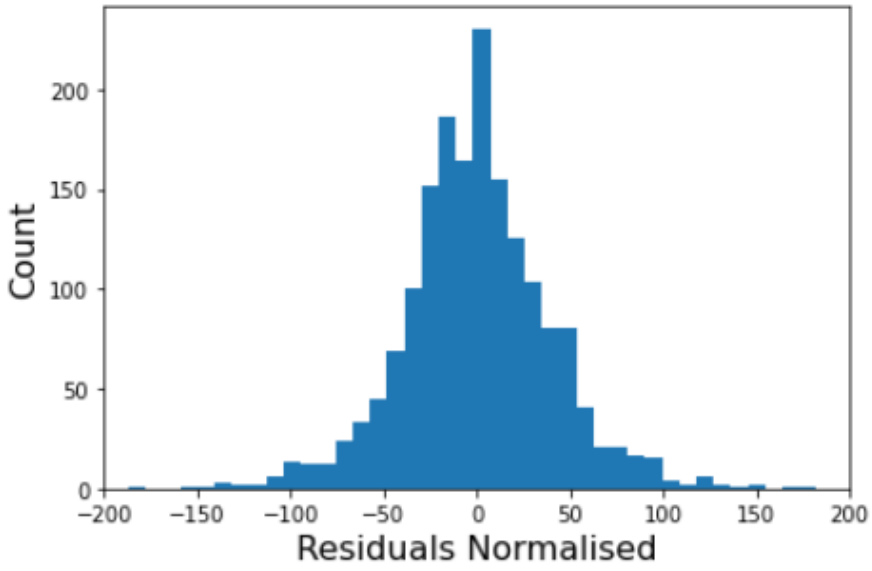
```
Root Mean Squared Error (RMSE):  40.607
```

```
r2 = r2_score(test_y, predicted_y_test)
r2
```

```
0.8317678207798673
```

Some of the and their visualizations along with their python code and their explanation are below

```
# # Checking if our residuals are normally distributed
residuals = test_y - predicted_y_test
plt.hist(residuals, bins = 50)
plt.xlim([-200,200])
plt.xlabel('Residuals Normalised',fontsize=16)
plt.ylabel('Count',fontsize=16)
plt.tight_layout()
```

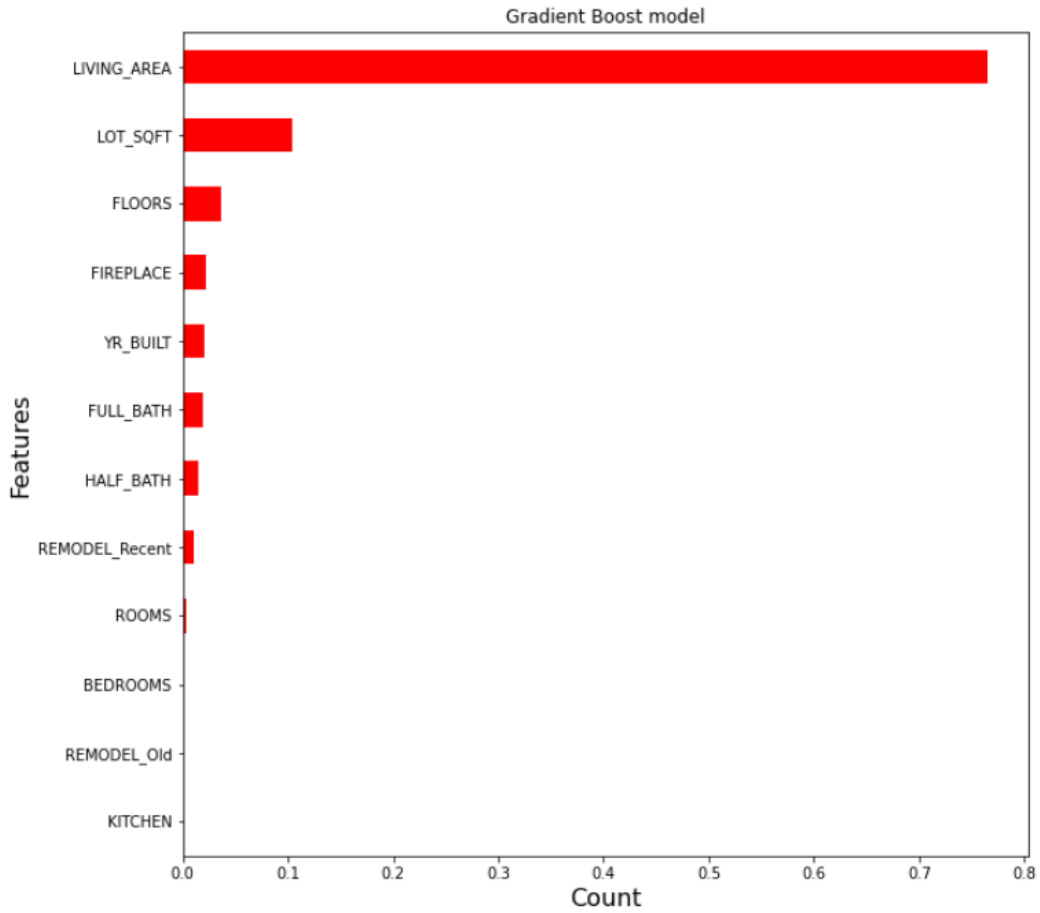


This snippet of Python code plots a histogram of the residuals from a Gradient boosting model, which are the discrepancies between the test set target values test_y and the projected target values predicted_y_test. Residuals calculate the difference between the actual test set target values and the predicted target values from the regression model.This enables us to see how the model's prediction errors are distributed.The model is functioning properly and the errors are random if the residuals have a normal distribution . Based on the above histogram we can say that the residuals are normally distributed. This is true because the histogram is evenly distributed around zero. Compared to any other models with a histogram, in the Gradient descent algorithm we see a perfect distribution or a bell curve formed. This indicates normality. It is close to being symmetric. So we consider it to be normally distributed.

```
feature_imp_GB = pd.Series(GB.feature_importances_, index = predictors_df1.columns)
feature_imp_GB
```

```
LOT_SQFT          0.104937
YR_BUILT          0.021359
LIVING_AREA       0.764878
FLOORS            0.036418
ROOMS             0.004134
BEDROOMS          0.001021
FULL_BATH         0.018908
HALF_BATH         0.015056
KITCHEN           0.000109
FIREPLACE         0.022143
REMODEL_Old       0.000346
REMODEL_Recent    0.010692
dtype: float64
```

```
# Checking for feature importance with Gradient boost model
figure(figsize=(10,10))
feature_imp_GB.sort_values().plot.barh(color='red')
plt.xlabel('Count',fontsize=16)
plt.ylabel('Features',fontsize=16)
plt.title('Gradient Boost model')
```



With the help of Gradient descent we can find features which are most important for the prediction task. With the code and output above we can see the value for Living area is the

highest. Which means living area plays a significant role in predicting total value. The same is seen in the visualization as well.

7. Lasso regression

Lasso regression is a form of regression that performs feature selection and causes the coefficients of less significant features to converge to zero using L1 regularization.Lasso stands for Least Absolute Shrinkage and Selection Operator.

Justification: Given the high degree of dimension in this dataset, Lasso regression may be an appropriate modeling method to use. As mentioned above it causes the coefficients of less significant features to coverage to zero. This can help to identify which features are the most relevant for predicting the target variable and remove irrelevant or redundant features from the model. Overall, using Lasso regression on the WestRoxbury dataset can help to simplify the model, improve its performance, and provide insights into impact on total value of the property.

Tune/train: They can be divided into a few steps as below.

1.Since pre-processing is already done in phase 1 we now divide the data into training and test sets.
2. Train the Lasso Regression model.
3. Then we evaluate the model with certain evaluation metrics like root mean squared error. Also we then find the accuracy to determine how well our model is performed.

These steps are shown in the code below

```
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.3, random_state=1)
```

```
# Model to use
lasso = Lasso(alpha=0.1)
```

```
lasso.fit(train_X, train_y)
```

```
Lasso(alpha=0.1)
```

Effectiveness: The root mean squared error has similar values as Linear Regression. Followed by accuracy having 80%. There is not much difference in the root mean squared error when compared to training and testing, hence we can say it is not overfit. The details are shown in the code below.

```
print("Root Mean Squared Error (RMSE): ", round(mean_squared_error(train_y, predicted_y_training) ** 0.5, 4))
```

```
Root Mean Squared Error (RMSE):  44.1868
```

```
predicted_y_test = lasso.predict(test_X)
```

```
print("Root Mean Squared Error (RMSE): ", round(mean_squared_error(test_y, predicted_y_test) ** 0.5, 4))
```

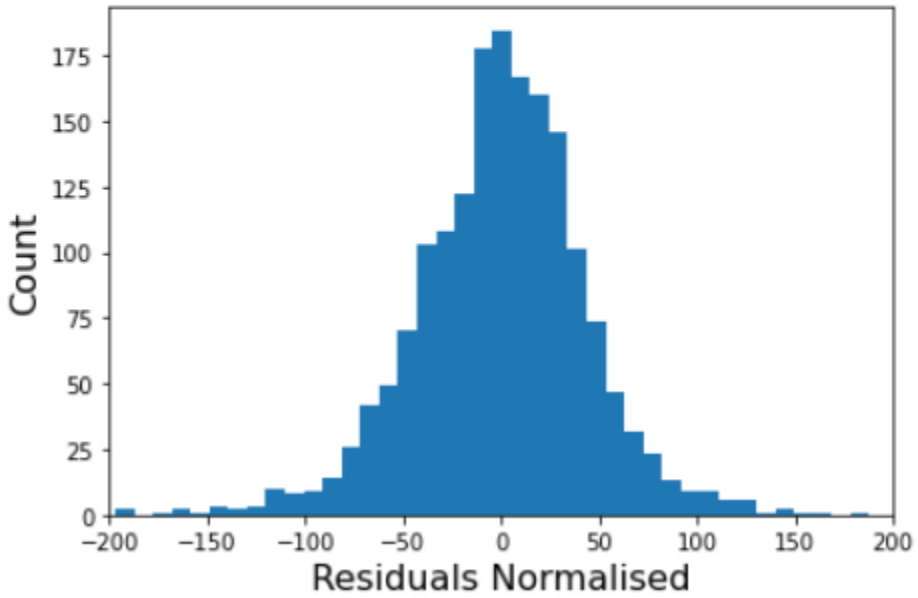
```
Root Mean Squared Error (RMSE):  44.2358
```

```
# Checking for accuracy
r2 = r2_score(test_y, predicted_y_test)
r2
```

```
0.8003560717136704
```

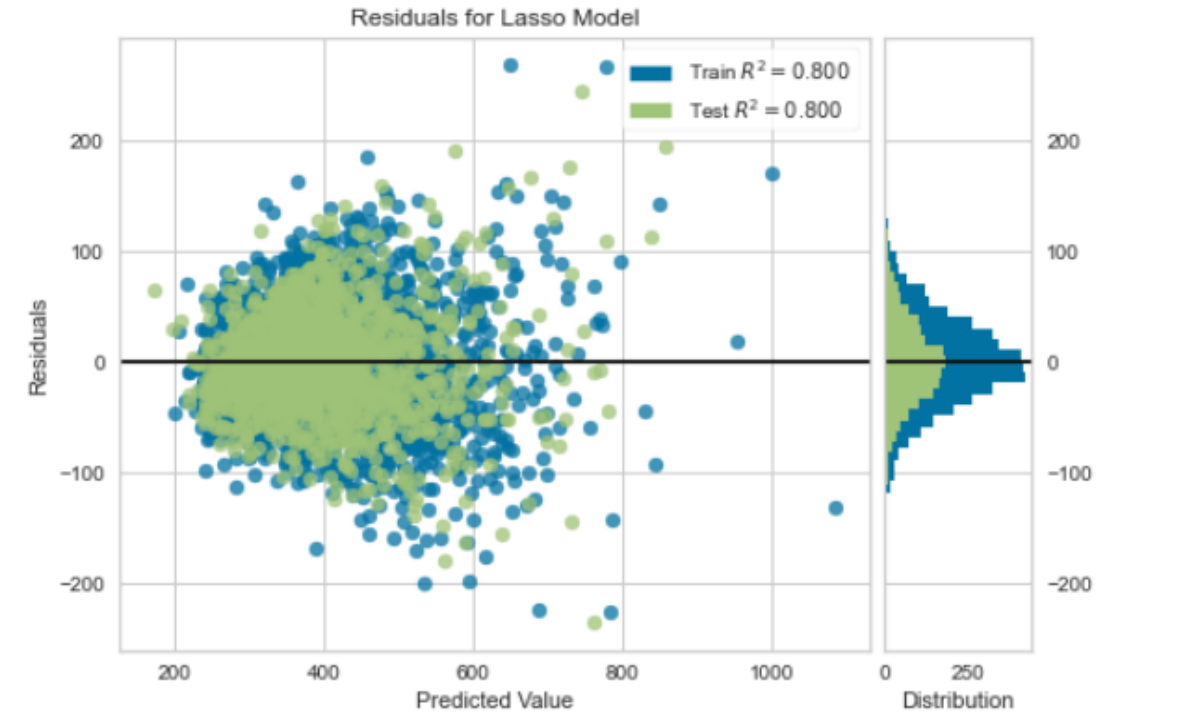
Some of the and their visualizations along with their python code and their explanation are below

```
# # Checking if our residuals are normally distributed
residuals = test_y - predicted_y_test
plt.hist(residuals, bins = 50)
plt.xlim([-200,200])
plt.xlabel('Residuals Normalised',fontsize=16)
plt.ylabel('Count',fontsize=16)
plt.tight_layout()
```



This snippet of Python code plots a histogram of the residuals from a Gradient boosting model, which are the discrepancies between the test set target values test_y and the projected target values predicted_y_test.This enables us to see how the model's prediction errors are distributed. Based on the above histogram we can say that the residuals are normally distributed. This is true because the histogram is evenly distributed around zero. It is close to being symmetric. So we consider it to be normally distributed.

```
# Plot for residuals in Lasso model
visualizer = ResidualsPlot(lasso)
visualizer.fit(train_X, train_y)
visualizer.score(test_X, test_y)
visualizer.poof()
```

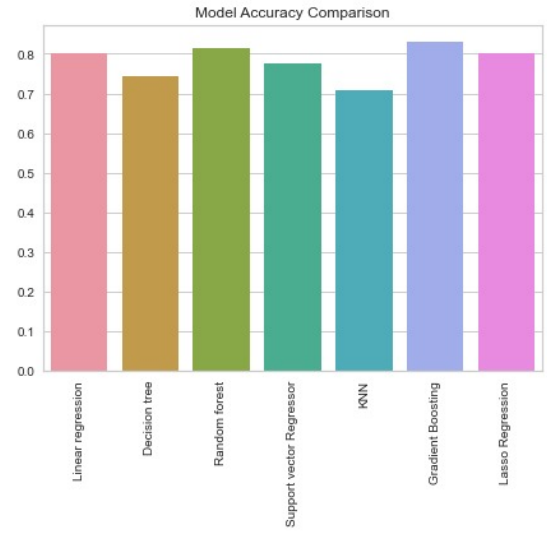


The offered code snippet plots the residuals of a Lasso regression model trained on a training set and tested on a test set using the ResidualsPlot visualization tool from the yellowbrick library.The ResidualsPlot shows the difference between the predicted values and the true values of the target variable. The second line in the code visualizer.fit produces the residuals for the training set and trains the Lasso regression model using the training set. Higher R-squared scores indicate greater performance since they indicate how well the model fits the test set data.

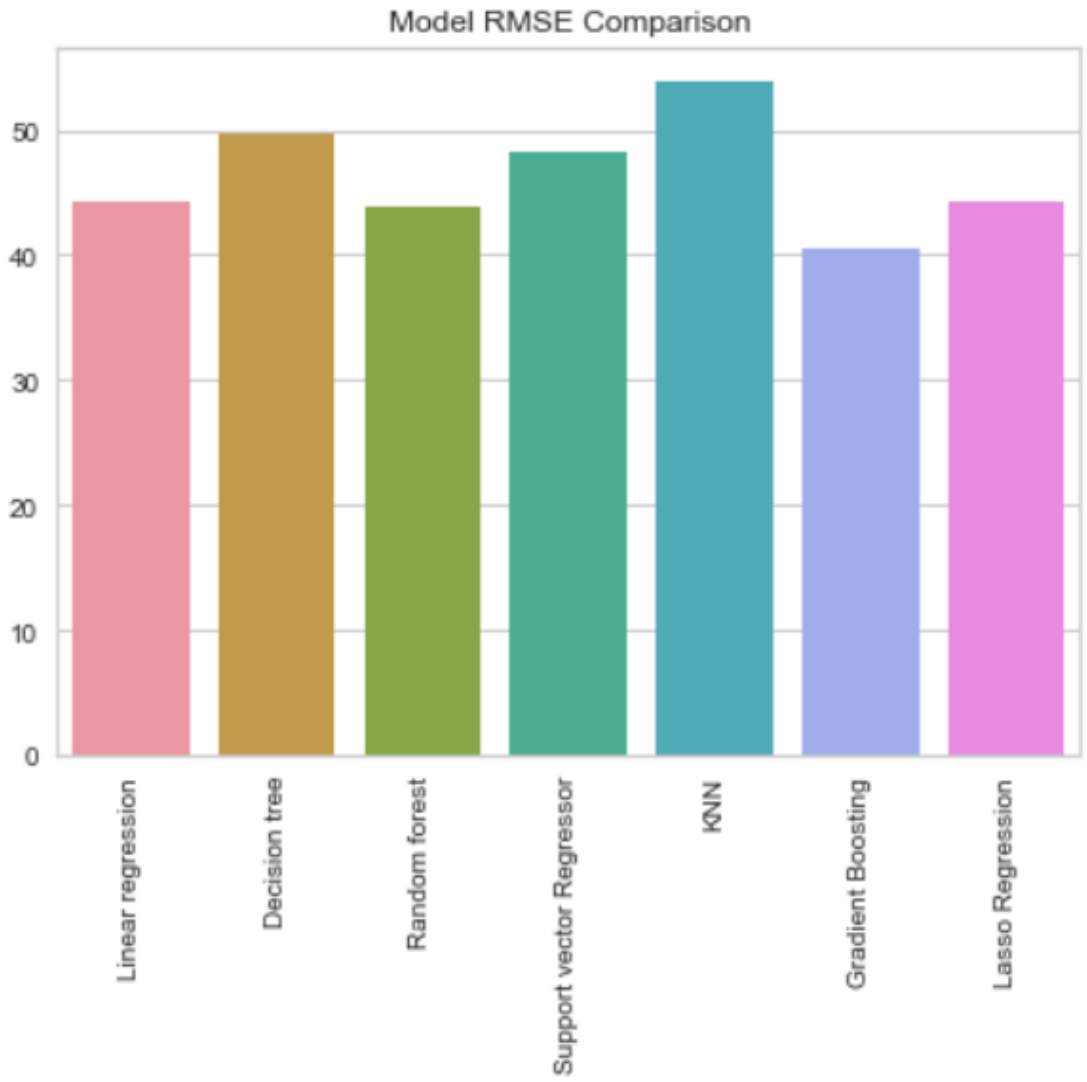
Conclusion: Though we see close values for Root mean square when compared to Training and testing in all the models, we have seen a huge decrease in error for Gradient descent. Also accuracy being the highest among all. That is 83%. Hence, we choose the Gradient descent model for our prediction task.

The same visualization and code is shown below.

```
plt.subplots(figsize=(7,5))
sns.barplot(x= list(model_accuracies.keys()), y = [float(model_accuracies[k]) for k in list(model_accuracies.keys())])
plt.xticks(rotation=90)
plt.title('Model Accuracy Comparison')
plt.show()
```




```
plt.subplots(figsize=(7,5))
sns.barplot(x= list(model_rmse.keys()), y = [float(model_rmse[k]) for k in list(model_rmse.keys())])
plt.xticks(rotation=90)
plt.title('Model RMSE Comparison')
plt.show()
```



References

[1] https://en.wikipedia.org/wiki/Gradient_descent

[2] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm.

[3]https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

[4]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

[5]https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

[6]https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

[7] Galit Shmueli,Peter C. Bruce,Inbal Yahav,Nitin R. Patel,Kenneth C. Lichtendahl, Jr.DATA MINING FOR BUSINESS ANALYTICS,2018

[8] Rachel Schutt and Cathy O'Neil. Doing data science, 2014.

[9]Joel Grus.Data Science from Scratch,2015

[10] https://seaborn.pydata.org/tutorial.html

[11] https://scikit-learn.org/stable/

[12]https://plotly.com/python/