Final Year Project Report

**Full Unit – Interim Report**

**Using Machine Learning Algorithms to Predict the Price of Pre-owned Cars**

Rizwan Bagdadi

A report submitted in part fulfilment of the degree of

**BSc (Hons) in Computer Science**

**Supervisor:** Li Zhang



Department of Computer Science

Royal Holloway, University of London

March 31, 2023

**Declaration**

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 13090

Student Name: Rizwan Bagdadi

Date of Submission: 31/03/2023

Signature: Rizwan Bagdadi

**Contents**

[**Abstract** 3](#_Toc131138892)

[**1.** **Introduction** 4](#_Toc131138893)

[**2.** **Methodology** 5](#_Toc131138894)

[**2.1.** **Outliers** 5](#_Toc131138895)

[**2.2.** **Data Wrangling** 7](#_Toc131138896)

[**2.3.** **Normalisation** 8](#_Toc131138897)

[**2.4.** **Data Visualisation** 9](#_Toc131138898)

[**3.** **Implementation** 12](#_Toc131138899)

[**3.1.** **Linear Regression** 12](#_Toc131138900)

[**3.2.** **K-Nearest Neighbour** 16](#_Toc131138901)

[**3.3.** **Decision Tree** 20](#_Toc131138902)

[**3.4.** **Random Forest** 25](#_Toc131138903)

[**3.5.** **User Interface** 28](#_Toc131138904)

[**4.** **Analysis and Evaluation** 30](#_Toc131138905)

[**4.1.** **Hold-Out Validation** 31](#_Toc131138906)

[**4.2.** **Cross-Validation** 32](#_Toc131138907)

[**4.3.** **User Input** 34](#_Toc131138908)

[**4.4.** **Overall Analysis and Evaluation** 34](#_Toc131138909)

[**5.** **Engineering tools** 35](#_Toc131138910)

[**6.** **Professional Issues** 38](#_Toc131138911)

[**7.** **Conclusion** 39](#_Toc131138912)

[**8.** **Bibliography** 40](#_Toc131138913)

[**9.** **Appendix** 42](#_Toc131138914)

[**9.1.** **Video links** 42](#_Toc131138915)

[**9.2.** **My diary** 42](#_Toc131138916)

# **Abstract**

In this paper, I investigate the application of supervised Machine Learning (ML) algorithms to predict the price of pre-owned cars in the UK. The goal of this project is to find the most suitable algorithm that can carry out the most accurate predictions, while providing the best performance and overall efficiency, by comparing their benchmark data. I use two techniques, K-nearest neighbour and decision trees, to make these predictions, using a dataset containing the features and prices of over 100,000 pre-owned cars to train the algorithms.

Throughout this paper, I first discuss why I chose to research this area, including the problems, the client, and the data. I then extensively describe and explain how each algorithm is effectively used, discussing the drawbacks and benefits of each, alongside extracts from my code with explanations on how they work. To give a more comprehensive view of the research, I also include a detailed explanation of the pre-processing techniques used to clean and prepare the data before training. Finally, I provide a full analysis and comparison of their outcomes, including a discussion on potential areas for future research in this field.

1. **Introduction**

More people are buying used cars than new. In a recent researched released by AA, 74% of the drivers reported that their most recent vehicle was used [1]. However, the absence of certainty and trust in the used car market makes it even more vital for both, the buyer and seller, to accurately predict the price of a pre-owned car. According to a study conducted by Lades in 2017, the automotive industry was one of the least trusted markets, ranking just behind banks and the telecommunications industry [1]. Individuals or dealerships often ask for an unreasonable price compared to the actual worth of the vehicle, taking advantage of those who are unsure of the approximate value of the car. Akerlof (1970) calls this phenomenon “information asymmetries” and highlights its negative impact on quality uncertainties [1]. Information asymmetries in markets occur due to lack of pre-purchase information that is trustworthy regarding a particular product. Therefore, creating uncertainties that result in higher costs and overall dissatisfaction for buyers on an individual level. However, the issue of quality uncertainty is not only caused by risk of deceitful behaviour of sellers, but also the inability of honest sellers to provide a comprehensive and reliable price of their product.

Additionally, face-to-face negotiations between the buyer and seller may create an opportunity for sellers to “treat buyers differently” based on age, gender, race, or religion, particularly if individual buyers have little or no means of knowing about prices paid by others [2]. Discrimination can impact negotiations as people often use visible characteristics to categorize others when they meet, which can serve as a cue for the stereotypes an individual may face in a society [3]. Thus, discrimination is another problem in the market that affects both the seller and the buyer. From a seller's perspective, they may be unsure of the worth of the vehicle they are selling, leaving them vulnerable, which commonly occurs with the elderly trying to sell their cars and facing age discrimination. From the buyer's perspective, they could be quoted an unfair price due to the seller's malicious intent or prejudice against certain groups. Hence, an accurate price predictor of pre-owned cars will help to reduce, not only information asymmetries, but also face-to-face discrimination that is faced by certain minorities in society.

When selling or buying a car, it's important to consider a variety of factors to ensure accurate and reliable prices. While the brand and model of the car are important, there are several other factors that can significantly affect the value of the car. For example, the mileage of the car is a key factor that can affect the price. A car with high mileage indicates it has been used more and may have more wear and tear on its components as compared to a car with low mileage. The engine size of the car is also important, as larger engines tend to be more powerful and may be more expensive to maintain. Other important factors to consider include the registration year, transmission type, fuel type, and fuel economy [1]. Calculating an estimate of all these features manually can not only be time-consuming and therefore inefficient but also costly if asked to be done by an expert.

Thus, the need for a system that utilizes a dataset containing hundreds of thousands of pre-owned car selling prices that can benefit both sellers and buyers by instantly predicting the selling price of a used vehicle. To ensure fair, reliable, and accurate results, I have opted to use machine learning (ML) to create this system. This approach eliminates all bias factors by removing user control from the system. The system uses supervised algorithms and requires training with a reliable dataset. Machine learning is useful in making accurate predictions, as it can learn from massive amounts of data whilst continuing to learn. For example, advanced machine learning can predict stock market trends, which would be extremely helpful in determining the selling prices of cars regardless of market conditions.

The aim of this project is to utilize machine learning algorithms to identify the most suitable algorithm for making accurate predictions while also providing the best overall efficiency and performance by comparing their benchmark data. To achieve this, I have utilised four different supervised machine learning algorithms, including Linear Regression, K-Nearest Neighbour, Decision Trees and Regression Forest, will be applied. Each algorithm used will undergo regression analysis and cross-validation to determine which provides the best performance. To train the algorithms, a dataset last updated in 2020 will be used, which contains the selling price and features of over 100,000 pre-owned cars [4].

1. **Methodology**

I sourced my data from Kaggle [4], which was uploaded by user Aditya and last updated in 2020. The data was collected in the same year from live listings. To ensure the dataset was appropriate, I verified that all the data was well-organized and included information on price, mileage, road tax, miles per gallon (mpg), and engine size. Additionally, I checked that there were no duplicate listings that could impact my results. Table 1 shows a sample of the data used from the Audi dataset.

Text

Description automatically generated

**Table 1.** *Sample Data Used of Audi Vehicles*

To use the data in my models, I had to first make it suitable. This involved performing data preprocessing, which includes the steps needed to parse the raw data easily by the algorithms and system [5]. Most real-world data tends to have missing values, noise, inconsistency, or unusable formats that cannot be directly used in machine learning algorithms. As shown in Table 1, there are a few implications pre-processing can eliminate. For example, certain features have string values, and the integer variables vary from 1.0 to 35000+ for different features. This process increases the accuracy and efficiency of the models.

## **Outliers**

Outliers are data points that are significantly different from the other values in a dataset. They can affect the accuracy of your results, and to inaccurate and unreliable results. Therefore, it is crucial to identify and handle outliers. The first step in preprocessing is to find any outliers in the dataset that could influence the results and remove them.

To identify outliers, I first calculated the interquartile range (IQR), which is done by finding the difference between the 75th percentile and the 25th percentile of the dataset and multiplying the result by a constant of 1.5. This provides the maximum and minimum values, which define the range of values that are not considered outliers. Values found outside this range are considered outliers. The sample of my code below shows the steps I took to find and eliminate the outliers in my dataset.

**for i in ['year']:**

q75,q25 = np.percentile(file.loc[:,i],[75,25])

      IQR = q75-q25

      max = q75+(1.5\*IQR)

    min = q25-(1.5\*IQR)

  file.loc[file[i] < min, i] = np.nan

    file.loc[file[i] > max, i] = np.nan

file = file.dropna(axis = 0)

While identifying outliers is a key step in cleaning my dataset, it is also necessary to understand the reasons behind their occurrence. In some cases, outliers may be valid data points that represent unique characteristics of the dataset. For example, the tax price of an expensive sports car, such as an Audi R8, may be an outlier, but it is a valid data point that should not be removed and must be considered.

Chart, box and whisker chart

Description automatically generated

**Figure 1.** *Box Plot demonstrating outliers in Audi dataset.*

Figure 1 displays the outliers identified in the Audi dataset. After analysing the data, I determined that the outliers in mileage, tax, mpg, and engine size are useful for feature analysis. However, the dataset includes only one data point for the years 1997, 1998, and 2002-2014. Since the year greatly impacts the algorithm's accuracy, I removed these outliers. While this reduces the available options for the user to select, it is necessary to ensure the accuracy of the algorithms.

Once the outliers have been identified, there are various ways to handle them. They can be removed from the dataset, treated separately in the analysis, or transformed using mathematical functions. As shown in the code sample above, I replaced all the outliers with Numpy's nan (Not a Number). I then handled these outside the for loop by removing all null values from the dataset.

Chart, box and whisker chart

Description automatically generated

**Figure 2.** *Box Plot demonstrating outliers in Audi dataset.*

After removing the outliers within the year feature, Figure 2 shows the Audi dataset. As a result, we can see that the extreme outliers in mileage have also been removed. By removing these outliers, we can ensure higher accuracy and reduce the chance of extremely inaccurate predictions. In addition, it is important to note that removing outliers may not always be the best solution. In some cases, it may be more appropriate to transform the data or analyse the outliers separately. Therefore, handling outliers requires careful consideration and a thorough understanding of the dataset being used.

## **Data Wrangling**

For the next step of preprocessing, I performed the data wrangling process, which involved a series of complex operations that transformed raw data into a usable and workable format. This was a crucial step in the data analysis process as it made it easier to process and obtain reliable output from the data. The dataset I was working with is split into car brands, each containing thousands of samples. To reduce the sample size and make it more manageable, I initially chose to use only one car brand, Audi. However, as I progressed with my analysis, I realized that limiting my analysis to just one car brand would not give me a complete picture of the data. As a result, I changed my code to work with all the car brands provided in the dataset.

However, within each car brand, the dataset was too large for my laptop to process, with a shape of (10668, 9) for the Audi dataset. To address this issue, I had to find a way to reduce the sample size without compromising the integrity of the data. After much consideration, I decided to use the Pandas head() function to select a sample size of 5000 for each car brand.

Text

Description automatically generated with low confidence

**Table 2.** *Data Types Used*

The first step I took in analysing the data was to examine the shape and type of variables included in the dataset, as shown in Table 2. This involved a careful examination of the data types found in the dataset, which were identified using the dtypes function from the Pandas library. I discovered that most of the variables were integers, with some float and object types. However, I encountered a problem with the variables that contained string values, such as the transmission, fuel type, and car model. These variables were critical in determining the selling price of the cars, but I could not use them in my algorithms because they were not numerical values.

To overcome this problem, I used Scikit-Learn's LabelEncoder() feature, which assigns numerical values to strings between 0 and the number of values – 1. This allowed me to include the variables with string values in my algorithms and obtain more accurate results in predicting the selling price of the cars. Overall, these adjustments helped me to obtain a more comprehensive understanding of the data and make more informed decisions in my analysis.

modelEncoder = LabelEncoder()

modelEncoder.fit( file [ "model" ])

file[ "model" ] = modelEncoder.transform( file[ "model" ])

file = file.head(5000)

The sample code above demonstrates how I used LabelEncoder to modify the variables of my model. For each feature (transmission, fuel type, and car model), I used a separate LabelEncoder to fit each feature, ensuring that there was no influence in transforming the data.

## **Normalisation**

Another thing I had to consider before using the dataset, is the spread of data. Table 1 provides an overview of the dataset and its wide range of values. For instance, the integer variables vary from 1.0 for engine size to over 35,000 for mileage, resulting in a significant difference in their ranges. To address this, I used the process of normalization, which transformed the variables in columns to have the same scale, making them more comparable. Note that not every dataset needs to be normalized for machine learning, only when the characteristics have different ranges [6].

Some machine learning algorithms, especially K-Nearest Neighbours and Support Vector Machines, benefit from normalization. This is because they use distance from data points to determine their similarities [7]. Figure 3 below shows the effects of scaling on a set of data points when comparing the Euclidean distance. We can see that scaling has brought both features closer to each other, making them more comparable than before.

A picture containing text

Description automatically generated

**Figure 3.** *Example of before and after of scaling [g]*

On the other hand, tree-based algorithms are not affected much by the scale of the features, as the decision process is not influenced by other features [7]. Therefore, I only normalised the data for the K-nearest neighbour algorithm. There are many different methods of normalisation, but for my project, I chose to use Min-Max scaling. Min-Max scaling works by setting the minimum of each feature to 0 and the maximum to 1, transforming the range into 0-1 [8].

When normalising the data, I had the option of normalising the data before or after splitting the data. However, I chose to normalise the data after splitting the dataset, as normalising it before can cause details of the test set to be leaked. This is because the mean and standard deviation that is used in the normalisation process would be used on the entire dataset, including the test data [8].

## **Data Visualisation**

After cleaning the data, we can visualize it much more clearly and observe the relationships and correlations between each aspect of the cars. Figure 4 below shows the distribution of each feature, excluding fuel type, transmission, and model.

Looking at the histogram graphs, we can already deduce a number of facts. Most used cars being sold are from 2019 and have a 2-litre engine with low mileage, but with moderately high fuel consumption. User-entered specifications that match these modes of features will have a higher accuracy prediction compared to cars with features like a 2012, 2.2-litre engine, 120,000+ miles, and extremely high fuel consumption. It should never be unexpected for a vehicle with such specifications, and it should be handled appropriately. Fortunately, with machine learning, all dimensions are considered and taken into account when formulating a prediction. This isn’t to say the prediction would be extremely accurate.

Chart, histogram, waterfall chart

Description automatically generated

**Figure 4.** *Distribution of Features in Audi Dataset*

Figure 5 depicts the relationships between each feature using Seaborn library's Heatmap feature. By examining the heatmap, we see that price has a positive correlation with engine size and year, indicating that people are willing to pay a premium price for newer cars with larger engines. However, it has a negative correlation with MPG and mileage, suggesting that people avoid cars with high fuel consumption and high mileage. The strongest correlation is between year and mileage, showing that older cars generally have higher mileage.

**Chart, bar chart

Description automatically generated**

**Figure 5.** *Relations Heatmap of Audi Dataset*

When purchasing a used car, mileage is often a key consideration as it can impact the car's reliability. Cars with higher mileage may have endured more stress, which can cause issues with the engine or other components. Figure 6 provides a more detailed view of how mileage impacts the selling price of a used car. As expected, cars with higher mileage tend to be older and sell for less. This supports the idea that buyers are willing to pay more for newer cars with lower mileage. However, it is important to note that other factors such as the car's make and model can also impact its selling price.

Chart, scatter chart

Description automatically generated

**Figure 6.** *Scatter Plot of Relation Between Mileage, Price, and Year of Audi Dataset*

Furthermore, the data reveals that most cars are sold within five years of ownership, with the mode year being 2019. This suggests that people tend to sell their cars after a few years of use, possibly to upgrade to newer models or to obtain a different type of vehicle. This trend may also be influenced by factors such as changes in lifestyle or financial circumstances. In summary, while mileage is an important factor when purchasing a pre-owned car, it is just one of many factors to consider. By taking into account the car's mpg, tax, transmission, fuel type, and other factors, buyers can make a more informed decision when buying a used car.

Figure 7 shown below further illustrates the correlation between price and year, highlighting the stark difference between the two factors. As we can see, the prices of newer cars tend to be higher than those of older cars. This could be attributed to a number of factors. Firstly, newer cars tend to have less mileage and therefore less wear and tear, which could lead to higher reliability and in turn, higher prices. Additionally, newer cars often come with more advanced features and technology, which could also contribute to their higher prices. It is also important to note that the demand for newer cars may also be a factor in the higher prices, with many consumers willing to pay a premium for the latest models.

Chart, bar chart

Description automatically generated

**Figure 7.** *Bar graph showing the relationship of price and year in the Audi Datatset.*

1. **Implementation**

In order to ensure that the outcomes of the project were reliable, I split the dataset used into two subsets: a training subset and a testing subset. The training subset consisted of 75% of the data, while the testing subset was made up of the remaining 25%. By using such a large training set, I was able to increase the reliability of the outcomes.

To create and analyse algorithms, I focused on three models: K-Nearest Neighbours, Decision Trees, and Random Forest. Each of these models has a unique approach and is likely to produce different results. For instance, K-Nearest Neighbours is a type of instance-based learning, while Decision Trees and Random Forest are more rule-based [9][10]. By using three different models, I was able to ensure that my analysis was thorough and that I considered all possible outcomes.

To begin my analysis, I started by testing the K-Nearest Neighbours algorithm. This algorithm works by classifying new data points based on the nearest neighbours in the training set [11]. By testing this algorithm first, I was able to establish a baseline for my analysis and gain an understanding of how the other algorithms compared.

* 1. **Linear Regression**

Linear Regression is a supervised machine learning algorithm used for predicting values. It is a statistical method for modelling the relationship between the dependent and independent variables. In this case, the independent variables are the predictors, being model, year, transmission, mileage, fuel type, tax, MPG and engine size. The main goal of linear regression is to find the line of best fit that describes the relationship between the variables. The algorithm works by fitting a linear equation to the training data, which can then be used to make predictions [12]. We use the following linear equation (1).

y = mx + c (1)

In equation 1, y in the independent variable, which can either be continuous or categorical, and x is a dependent variable, which is always continuous. We use probability distribution to analyse it, with a focus on conditional probability distribution with multivariate analysis [13].

* + 1. **Simple Linear Regression**

Simple linear regression, as represented in Figure 8, is a process where the dependent variable is predicted using a single independent variable. This is also known as univariate regression analysis. The goal is to measure the relationship between the two variables, similar to correlation, but with the added distinction between the dependent and independent variables.



**Figure 8.** *Single Linear Regression graph [14]*

* + 1. **Multiple Linear Regression**

While simple Linear regression is a useful approach for predicting a response based on single predictor variables, in reality there are often multiple predictors, as there when predicting the price of pre-Owned vehicles. One way to solve this is by running separate simple linear regression models for each predictor, but this approach has limitations [15]. A better approach to this is by extending the simple linear regression (1) to accommodate for multiple predictors by assigning each predictor a separate slope coefficient in a single model. This is called Multiple Linear Regression, it takes the form of equation (2) below. It also allows for more accurate predictions and a better understanding of the relationship between the predictors and the response.

Y = β0 + β1X1 + β2X2 + ··· + βpXp + ϵ (2) [15]

In the linear equation (2), Y represents the dependent variable and X1, X2… represents the independent variables, the predictors, that are used to predict the value of Y. β0 is the intercept term and β1, β2, ..., βp are the regression coefficients that represent the change in Y for a unit change in X1, X2, ..., Xp, respectively. The link between the dependent variable and the independent variables' random fluctuation or noise is represented by the error term ϵ.

 (3) [15]

The goal of multiple linear regression is to estimate the regression coefficients β0, β1, β2, ..., βp that minimize the sum of squared errors between the predicted values of Y and the actual values of Y. Given the estimates s βˆ0, βˆ1,..., βˆp, we can make predictions using the equation (3). Unlike the estimates for simple linear regression, the estimates of multiple regression coefficient estimates have more complex forms, which are best represented through matrix algebra.

Chart, scatter chart

Description automatically generated

**Figure 9.** *Multiple Linear Regression graph [15]*

For my linear regression model, I implemented the equation (2) into my code:

**def \_\_init\_\_(self, learning\_rate, iterations):**

self.learning\_rate = learning\_rate

self.iterations = iterations

self.weights = None

The ‘\_\_init\_\_’ method initializes the ‘learning\_rate’, ‘iterations’, and ‘weights’ of the model. The ‘learning\_rate’ determines the step size at each iteration of gradient descent, and the ‘iterations’ specify the number of times the gradient descent algorithm will run. The ‘weight’ is the array that stores the values of the coefficients β0, β1, β2, ..., βp. It is initially set to ‘None’ at this point, which means they will be initialized later in the ‘fit’ method.

**def fit(self, X\_train, Y\_train):**

X = np.insert(X\_train, 0, 1, axis=1)

self.weights = np.zeros(X.shape[1])

for i in range(self.iterations):

Y\_pred = np.dot(X, self.weights)

error = Y\_train - Y\_pred

gradient = - (2 \* (X.T).dot(error)) / X.shape[0]

self.weights = self.weights - self.learning\_rate \* gradient

return self

The ‘fit’ method trains a linear regression model using gradient descent optimization. The input parameter ‘X’ is a matrix of independent variables with dimensions (m x n), where ‘m’ is the number of training examples and ‘n’ is the number of features. The input variable ‘Y’ is a vector of size (m x 1) which contains the corresponding target values for each training example. We use this to calculate the error term.

The first step of the ‘fit’ method is to add a column of ones to the matrix ‘X’, to represent the intercept term from the equation (2), using the ‘np.insert’ function. For the next step, the weights vector ‘self.weights’ is initialized as a vector of zeros of size (n+1 x 1), where n+1 is the number of features including the intercept term.

Then, the gradient descent algorithm is performed for the specified number of iterations. At each iteration, the predicted values for the training data are calculated using the current weights. This is done by taking the dot product of ‘X’ and the weights vector. The predicted values are stored in ‘Y\_pred’. The error (ϵ) between the predicted values and the true values is then calculated.

gradient = - (2 \* (X.T).dot(error)) / X.shape[0] (4)

The gradient of the cost function with respect to the weights is then calculated, using the formula (4), where ‘X.T’ is the transpose of ‘X’. This formula (4) is the derivative of the cost function, which is the mean squared error (MSE) between the predicted and true values. The transpose of ‘X’ is multiplied by the error vector to get a vector of partial derivatives for each weight. The result is then divided by the number of training examples `m` to get the average gradient. The negative sign is added to the formula because the gradient descent algorithm seeks to minimize the cost function, and the direction of steepest descent is in the opposite direction of the gradient. Finally, the weights are updated using the formula ‘weights = weights - (learning\_rate \* gradient)’. The ‘fit’ method then returns the instance of the ‘linearRegression’ class.

**def predict(self, X\_test):**

X\_test = np.insert(X\_test, 0, 1, axis=1)

return np.dot(X\_test, self.weights)

The `predict` method is used to predict the values of the dependent variable for new observations. It takes in a matrix of independent variables with dimensions (m, n), where `m` is the number of new observations, and `n` is the number of features. The first step is to add a column of ones to the matrix `X`, using the `np.insert` function, which represents the intercept term in the linear regression equation. The predicted values of the dependent variables, `Y`, are then calculated as the dot product of `X` and the weights vector calculated in the `fit` method. This is equivalent to the linear combination of the independent variables with their corresponding weights plus the intercept term.

One advantage of using this algorithm is it simplicity to understand and implement with any dataset, and it does not require a lot of computational power to run. However, there are many downsides to using this algorithm to predict the price of used cars. Linear regression assumes there is a linear relationship between the variables, meaning that non-linear relations will not provide accurate predictions. Additionally, linear regression can be sensitive to outliers in data, which I have handled.

* 1. **K-Nearest Neighbour**

K-nearest neighbour (KNN) is a widely used and effective non-parametric method of supervised learning that uses proximity to predict the grouping of an individual data point [7]. It is a versatile algorithm that can be used for both classification and regression cases.



**Figure 10.** *Visual of KNN [16]*

KNN works by selecting the closest k data points to the test new point. This method is highly customizable, as the value of k can be adjusted based on the specific problem at hand. To do this, we calculate the Euclidean distance to each training data point and choose the k closest ones. There are other methods for calculating distance, such as Manhattan (for continuous data) and Hamming distance [17]. The Euclidean distance is calculated by taking the square root of the sum of the squared differences between the new point and the training points, as shown in Figure 11. Once the k closest points have been identified, we calculate the average of their labels to determine the prediction. In this case, the label is the price of the item being predicted.

Chart, line chart

Description automatically generated

**Figure 11.** *Euclidian distance*

One advantage of KNN is its simplicity and ease of interpretation. It is fairly easy to understand how it works and how it makes predictions. Additionally, it can handle both categorical and numerical data, as long as the data is transformed into dummy variables, making it a versatile approach. However, KNN can be sensitive to outliers and has extremely poor run time performance with large training data [19].

To implement the nearest neighbour algorithm, I needed to write five methods. These included a method to calculate the Euclidean distance, a method to sort lists, a method to calculate the root mean squared error (RMSE), a method to handle new predictions, and a KNN method to handle all the other methods.

*# K-Nearest Neighbour method*

**def KNN(train, testRow, yTrain, yTest, K):**

distance = list() #Stores distance of each point

for i in range(len(train)-1):

dist = eucDistance(train[i], testRow)

distance.append((train[i], dist, yTrain [i]))

distance = sort(distance) *# Gets sent to sort method – sorts in ascending order*

kNeighbours = list() #list to store K amount of neighbour results

for i in range(K):

kNeighbours.append((testRow, distance[i][1], distance[i][2]))

return kNeighbours

The KNN method works by iterating through all the rows in the training set and finding their Euclidean distance to the test row. However, this process can be quite costly as it must work out the distance for 750 points for just one of the 250 rows of test data. This means that the Euclidean distance must be calculated 187,500 times for only one value of K. Considering the size of big datasets, using KNN can be very time-consuming. If I hadn't reduced the Audi dataset to only 1,000 vehicles, the Euclidean distance would have had to be calculated an astonishing 18,750,000 times for one value of K.

After calculating the Euclidean distance to each training row, the KNN method sorts the list of distances and takes the K nearest points to the test row, returning these values and their labels to the prediction method. Finally, the prediction method takes the labels of the K nearest neighbours and calculates their mean, returning the prediction of that test row or user input.

*#Predictes the price of the given car*

**def predict(train, test, yTrain, num\_neighbors):**

predictions = list()

for i in range(len(test)):

neighbour = KNN(train, test[i], yTrain, num\_neighbors)

labels = [] #Stores yTrain for each test variable

for i in range (len(neighbour)):

t = labels.append(neighbour[i][2]) #Appends yTrain

predictions.append(mean(labels))

return predictions

During the evaluation process, the prediction method stores all given predictions in a list. This list is returned to the evaluation method after the completion of all predictions. Using the given predictions, the evaluation method calculates the RMSE value for each value of K. It then plots these values, providing a visualization of the accuracy of the algorithm.

*# Evaluating 100 nearest neighbours*

**def eva(train, test, yTrain, yTest):**

rmseValues = []

for k in range(1, 101):

y\_pred = predict(train, test, yTrain, k)

error = rmse(yTest, y\_pred)

rmseValues.append(error)

print('RMSE value for k =' , k , 'is:', error)

return

To evaluate and find the optimal value of K neighbours, for my datasets, I created a method, ‘eva’, that begins by creating a list of the RMSE values, then proceeds to loop through 100 values of K, in creating predictions. The prediction method takes all the training values and only the features of the test set, with this it creates a loop that iterates over each row (car) in the test set and finds the K nearest neighbours to that row using the KNN method.

Chart, line chart

Description automatically generated

**Figure 12.** *K Neighbours Evaluation*

Using my ‘eva’ method I worked out the RMSE value of each K value, from 0 to 100, to find the optimal value of K that enhances the accuracy of the predictions. In doing so, I was able to see that 4 nearest neighbours were the best value of K for my data set. RMSE is the root mean square error, which shows how far the predictions falls from the actual values, by calculating the average deviation between the actual price and predicting price, using the Euclidean distance between them [21]. The lower the RMSE value, the better the model fits the dataset. In figure 12, we see that the best values of K for this data set are between 3 and 6, with the best being 4 neighbours. After k = 6, there is a huge linear increase in the value of RMSE, showing that the larger the value of K, the less accurate the results.

Text, letter

Description automatically generated

**Figure 13.** *Demonstration of user input for my KNN algorithm*

After evaluating my algorithm against a test set and finding the optimal number of neighbours for my dataset, I created a temporary user interface, shown in figure 13, that allows users to provide details of their own vehicles, and receive a accurate prediction. Figure 13 shows an interaction where the details for a 2016 Audi RS6, with 49,050 miles, and an actual worth of £44,985 was inputted into my algorithm using 4 neighbours, outputting a prediction of £44,717. This result being -£268 from the actual worth and with a calculation time of 0.0938 seconds (3 sf) demonstrates promising results from my KNN algorithm. In order to calculate the prediction time, I used the time library to measure the time it took for the prediction method to run and return a prediction.

* 1. **Decision Tree**

A decision tree is a non-parametric form of supervised learning, similar to KNN, that uses labelled input and output datasets to train models. Decision trees are mainly used for classification problems, but can also be used in regression problems to predict outputs from unseen data. Essentially, a decision tree is a tree where “each node shows a feature (attribute), each link (branch) shows a decision (rule) and each leaf shows an outcome (categorical or continues value)” [20]. Decision trees are designed to emulate human-like thinking, resulting in a much more natural thought process, similar to how humans think. Figure 14 illustrates the basic layout of a decision tree, showing sub-trees that fine-tune the predictions from the previous node, starting from the root node, and ending with the leaf nodes.

Diagram

Description automatically generated

**Figure 14.** *Decision Tree Illustration [22]*

For my decision tree algorithm, I used the same dataset as in my KNN algorithm, which had a size of 1000. However, before splitting the data, I decided to perform some preprocessing steps to obtain better results. To start, I converted the features variable (X) using the to\_numpy() function from the Pandas library. This function provides a more efficient way to work with the data, converting a data frame into a NumPy array. This allowed me to handle the data more effectively and avoid unnecessary overhead.

Furthermore, I also reshaped the labels variable (Y) using values.reshape(-1, 1), shaping it into an array with 1 column. This step was necessary to ensure that the data was properly structured and processed by the algorithm, which required a specific format for the input data. Unlike the KNN data, I did not normalize the data, as the decision process isn't influenced by other features, and the results aren't affected by other features.

When building a decision tree using the training set, a greedy approach is used to make the best split at each step. This means that the algorithm does not look ahead to choose a split that may produce a more desirable tree later on. The desirability of a tree is measured by calculating the information gain, which determines how many features became more accurate than their parent nodes prior to being split.

In classification, we use a method called entropy to measure information gain on a new example's class [23]. This method measures the randomness within the information being processed [24]. In regression, we assume that we will produce an output given the input variables, rather than attempting to predict a class label. We examine whether a split would result in a decrease in the variance of the data. A split is considered useful if the collective weighted variance of its children has a smaller value than the parent's original variance.

The dataset is split recursively using binary splitting until the current node reaches a predetermined minimum number of splits required to split an internal node or a maximum depth allowed. This method follows a top-down, greedy approach to grow the tree with the training set.

Diagram

Description automatically generated with medium confidence

**Figure 15.** *Perspectives of recursive binary splitting [15]*

Figure 15 depicts perspectives of the recursive binary splitting process. The top right panel shows the splitting on a two-dimensional plane, which is then translated into the tree in the bottom left, and then into a 3D perspective of that [15]. This figure is a great tool to understand the depth of recursive binary splitting that occurs on many planes with the dataset of cars being used. By visualizing the different steps of the process, you can see how the algorithm works in a more comprehensive manner. It is important to understand this process, as it is a fundamental step in building a decision tree.

Timeline

Description automatically generated

**Figure 16.** *Illustration of my decision tree using the training set*

In Figure 16, we can see a visualization of the training set, from the Audi dataset, where the maximum depth is 3. This is an important aspect to consider when analysing the performance of the algorithm. By examining this visualization, we can observe how the algorithm splits at each node, based on the value of the current feature. This information is crucial for understanding how the algorithm makes decisions and for evaluating its effectiveness. In addition, we can also examine the distribution of the data points in the training set and how they are classified by the algorithm. This can provide further insights into the strengths and weaknesses of the algorithm and help identify areas for improvement.

Before writing code for the decision trees, I had to develop a Node class that was able to initialize instances of each node and store their values. This ensured that each node was properly equipped to handle the data it would be given, and that it could make informed decisions based on that data.

As shown in the extract of code below, by initializing instances of the class and storing their values, I was able to keep track of vital pieces of information necessary for making accurate predictions. For example, the "feature" variable stored the current features of the node, while the "limit" variable stored the local threshold of the node, where the threshold was calculated as the largest value from the training set below the limit.

*# Node class to initialise instances of each node*

**class Node():**

**def \_\_init\_\_(self, feature = None, limit = None, leftSide = None, rightSide = None, gain = None,**

**leaf = None):**

self.feature = feature

self.limit = limit

self.leftSide = leftSide

self.rightSide = rightSide

self.gain = gain

self.leaf = leaf

In addition to the "feature" and "limit" variables, I also made use of the "leftSide" and "rightSide" variables. These were used to store the subtrees on either side of the node, as given by the split method. This allowed me to create a tree structure that could be traversed in order to make predictions about new data. The "gain" variable was another important piece of information stored in each node. This variable kept track of the information gained on each node. By maximizing the information gain at each split, I was able to create a decision tree that accurately predicted the outcome of new data. Finally, the "leaf" variable was used to store the mean of the labels/prices of all the leaf nodes. This allowed me to quickly and easily make predictions about new data by simply traversing the decision tree and calculating the mean of the appropriate leaf nodes.

After completing the Node class, I wrote the main part of the code, the decision tree. This consists of several methods, including the initialization method, the method to fit the tree, the method to build the tree, the methods to split the tree, the method to find the information gain, and the methods to make predictions.

**class decisionTree():**

**def \_\_init\_\_ (self, minSamples, maxDepth):**

self.root = None

self.minSamples = minSamples

self.maxDepth = maxDepth

**def fit (self, X, Y):**

trainingSet = np.concatenate((X, Y), axis=1)

self.root = self.treeBuild(trainingSet)

The class begins by creating an instance of the class using ‘myTree = decisionTree(N, M)’, where the init method takes the N to define the minimum samples and M to define the max depth of the tree. This is then followed by a call to the fit method, with the training sets features and labels, which concatenates the training set into a whole data set, using the NumPy library, and then calls the treeBuild method with the training set.

**def treeBuild (self, trainingSet, currentDepth = 0):**

X = trainingSet [:,:-1] *# everything but the last value*

Y = trainingSet [:,-1] *# only the last value*

*# Iterates until this condition is met*

if X.shape[0] >= self.minSamples and currentDepth <= self.maxDepth:

bestSplit = self.bestSplit(trainingSet, X)

if bestSplit["gain"] > 0:

leftTree = self.treeBuild(bestSplit["leftSide"], currentDepth + 1)

rightTree = self.treeBuild(bestSplit["rightSide"], currentDepth + 1)

node = Node(bestSplit["feature"], bestSplit["limit"], leftTree, rightTree,

bestSplit["gain"])

return node

*# Calculates mean of leaf nodes*

leafValue = np.mean(Y)

val = Node (leaf = leafValue)

return val

The *treeBuild* method begin by splitting the dataset back into features and labels. The method then moves into a recursive binary loop that only stops once these conditions are met: when the shape of the tree/sub-tree reaches the minimal sample and the depth of the sub-tree reaches the max depth. As soon as the code enters this loop it calls the bestSplit method which finds the best splits to build the tree. After finding the best split of the current node, it then finds the best split of the left and right children, and then saves and returns the node along with its children/sub-tree values. This recursive binary loop keeps occurring until the conditions are met. Once met, the mean value for each leaf node is calculated. In classification, the mode is used instead of the mean, just like with nearest neighbour.

**def bestSplit (self, trainingSet, X):**

bestSplitt = {}

biggestGain = -1

for feature in range(X.shape[1]):

featureValues = trainingSet [:, feature]

thresholds = np.unique(featureValues)

for j in thresholds:

leftSide, rightSide = self.splitTree(trainingSet, feature, j) #splits node into 2 sub-trees

if (len(leftSide) > 0 and len(rightSide) > 0 ):

parent = trainingSet [:, -1]

leftNode = leftSide[:, -1]

rightNode = rightSide[:, -1]

currentGain = self.infoGain(parent, leftNode, rightNode)

if currentGain > biggestGain:

bestSplitt["feature"] = feature

bestSplitt["limit"] = j

bestSplitt["leftSide"] = leftSide

bestSplitt["rightSide"] = rightSide

bestSplitt["gain"] = currentGain

biggestGain = currentGain

return bestSplit

**def splitTree (self, trainingSet, feature, limit):**

left Branch = []

rightBranch = []

for i in trainingSet:

if i[feature] <= limit:

leftBranch.append(i)

else:

rightBranch.append(i)

rightBranch = np.array(rightBranch)

leftBranch = np.array(leftBranch)

return leftBranch, rightBranch

The best split method starts by creating a dictionary to store the best splits and a variable that stores the highest gain. The code then goes into a for loop that iterates over every feature storing and iterates over each unique features, finding the split of the left and right branch using the *splitTree* method. After splitting, I appended the current node labels to a parent list and the left/right nodes labels to a left/right node list. With these lists, I measured the info gain on them and then compared it to the biggestGain variable. If it was bigger than the biggest stored gain, that would then become the biggest gain variable and the best split.

**def infoGain (self, parent, leftNode, rightNode):**

leftWeight = len(leftNode) / len(parent)

rightWeight = len(rightNode) / len(parent)

information\_gain = np.var(parent) - (leftWeight \* np.var(leftNode) + rightWeight \*

np.var(rightNode))

return information\_gain

The infoGain method is used to determine the quality of a split in a decision tree. It takes in three parameters: the parent node and the left and right child nodes resulting from a split. The method first calculates the weights of the left and right child nodes based on the number of samples in each. It then uses these weights to calculate the information gain of the split. The information gain is calculated by subtracting the weighted sum of the variances of the left and right child nodes from the variance of the parent node. This calculation is based on the idea that a good split will result in a reduction in the variance of the variable being predicted. Therefore, the greater the reduction in variance, the higher the information gain, and the better the split.

**def prediction Loop(self, test Row, root):**

if root.leaf != None: #not empty

return root.leaf

featureVal = testRow[root.feature]

if featureVal <= root.limit:

return self.predictionLoop(testRow, root.leftSide)

else:

return self.predictionLoop(testRow, root.rightSide)

**def predict(self, xTest):**

predictions = [ ]

for row in xTest:

predictions.append(self.predictionLoop(row, self.root))

return predictions

The prediction method takes the testing variable and iterates over each row in the set given and calls the *predictionLoop* method for each row. The *predictionLoop* method also uses a recursive binary loop that repeats until it reaches a leaf. It searches through both the left and right branches of the already trained tree, by checking whether or not it feature value is less than or greater than the limit/threshold.

After initially testing my decision tree algorithm using a max depth and minimum sample of 3, I realised that my code was extremely and not performing to its full potential. To find the optimal parameters, I used Scikit Learn’s GridSearchCV method, setting the parameters to a range of 2 to 100. Through this process, I found the optimal variables to be a max depth of 34 and a minimum sample size of 3. This led to a massive increase in accuracy, but also increased the training time for algorithm.

* 1. **Random Forest**

Random forest (RF) regression is an ensemble machine learning algorithm that is constructed using multiple decision trees together to produce one final prediction, hence the reason it’s called a forest. The algorithm works by building a set number of decision trees, of which each tree predicts its output based on a subset of input variable. Each decision tree in the random forest is built using a different sample of rows, which is selected through the process of bagging. Bagging, also known as bootstrap aggregating, is a process that involves taking several random samples of the original dataset, with replacement. This minimises the variance of the model and prevents overfitting [25]. The algorithm then selects the best split at each node, based on a random sample of features selected for splitting. This technique is known as feature bagging and further minimises overfitting and helps the model’s generalisation to new data [25].

Overall, the utilization of bagging and feature bagging in random forests results in accurate predictions and reduces the risk of overfitting by reducing the impact of noise and outliers in the dataset. The final prediction is made by calculating the average of the predictions from all the trees [26]. This is an ensemble modelling technique that aggregates models to improve test accuracy, while reducing the costs associated with storing, training, and obtaining inferences from multiple models [27].

Diagram

Description automatically generated

**Figure 17.** Illustration of bagging

Random Forest Regression is a versatile algorithm that can handle a wide range of data types, including categorical and continuous variable. It also works well with noisy or incomplete data, making it a popular choice for real-world data [28]. Random Forest Regression has several other advantages, such as its robustness to outliers, where output outliers only affect the estimate of the leaf node it’s in and not the value of other leaf nodes; this is where the RF algorithm tends to stand out in comparison to other algorithms [29]. RF’s are also capable of handling non-linear relationships between variables, unlike other machine learning algorithms that are designed specifically for linear relationships, making it a versatile and powerful tool for data analysis [30].

In addition, the random forest algorithm is extremely useful for analysing large volumes of data, as it can efficiently analyse datasets with lots of data points, making it a popular choice for large data applications. Moreover, it is easily parallelizable, meaning it can benefit from multi-core processors and distributed computing power to further maximize performance on large datasets. However, this requires a computer capable of handling this workload and the right knowledge to do so. Keep in mind that the random forest algorithm can be computationally expensive, especially with large datasets and many trees.

Writing the code for the random forest algorithm was relatively straightforward, as the main idea utilizes decision trees. To use the decision tree algorithm, I had to convert the Jupyter Notebook file into a Python file so that the 'randomForest' class could access the 'decisionTree' class. Aside from the initialise method, I only needed to create three additional methods: the bootstrap sample method, the fit method, and the prediction method.

The ‘\_\_init\_\_’ method initializes the class and sets default values for the hyperparameters of the algorithm. The hyperparameters set in this method include the number of trees to be built (numTrees), the minimum number of samples per leaf node (minSamples), the maximum depth of each tree (maxDepth), and the seed value for the random number generator used in the algorithm (random\_state). Users can modify these parameters when initializing an instance of the ‘randomForest’ class.

**def bootstrapSample(self, X, y, state):**

        sampleNumb, featuresNumb = X.shape

        samples = np.random.RandomState(state).choice(a =

sampleNumb, size = sampleNumb, replace = True)

        return X[samples], y[samples]

The ‘bootstrapSample’ method is a function used for randomly sampling subsets of the training data during the construction of each decision tree. It takes in the input features X, target labels y, and the random state, and returns a random subset of the data. This ensures that each decision tree is trained on a different subset of the data, which helps reduce overfitting and improve the generalization performance of the model. The method generates a random sample of the data by selecting ‘sampleNumb’ samples from ‘X’ and ‘y’ with replacement. The size of the sample corresponds to the number of rows in the input feature matrix. The method then returns the selected samples, which are used to train the decision tree.

**def fit(self, X, y):**

if len(self.decisionTree) > 0:

self.decisionTree= []

num\_built = 0

for i in range(self.numTrees):

try:

DT = decisionTree(minSamples = self.minSamples,

maxDepth = self.maxDepth)

\_X, \_y = self.bootstrapSample(X, y,

self.random\_state + i)

DT.fit(\_X, \_y)

self.decisionTree.append(DT)

num\_built += 1

print("NUMBER BUILT: ", num\_built)

except Exception as e:

print("ERROR: ", e)

continue

The ‘fit’ method is the main method used to train the random forest, building ‘numTrees’ decision trees. It first checks if any decision trees have been previously built and clears the list if needed. The method then builds each tree by creating a decisionTree object and training it on a random sample of the input features ’X’ and target labels ‘y’, using the ‘bootstrapSample’ method. In case of an error during the construction of a tree, such as a failure to split a node due to too few samples, the method catches the exception and continues building the remaining trees. Once the trees are built, they are used to make predictions on new input features in the predict method.

**def predict(self, X):**

y = []

for tree in self.decisionTree:

y.append(tree.predict(X))

y = np.swapaxes(a = y, axis1 = 0, axis2 = 1)

predictions = []

for preds in y:

predictions.append(np.mean(preds))

return predictions

The ‘predict’ method is used to predict the output labels for new input features ‘X’ using the trained random forest model. It first predicts the output labels for each decision tree in the forest and stores the predictions in a list. It then calculates the mean of the predictions for each input feature and returns the result as the final prediction.

Similar to my decision tree algorithm, I did not initially find and use the optimal values for my algorithm. As a result, the algorithm gave much poorer results than the decision tree algorithm. Once again, I used Scikit Learn's GridSearchCV method and found that the optimal values were 31 trees, a max depth of 90, and a minimum sample of 6.

Even after finding the optimal values, my algorithm still gave extremely poor results, worse than the decision tree algorithm. After a lot of debugging and tracing, I discovered that the problem was in the 'bootstrapSample' method. Originally, I was using Numpy's 'random.choice', but later switched to Numpy's 'random.RandomState(state)', where 'state' was a fixed variable equal to the number of the tree being used. Using this allowed me to use the same internal state of the random number generator each time I used the algorithm, making it useful for generating reproducible results.

* 1. **User Interface**

To make my program accessible to a wider audience, I created a simple and intuitive user interface using PyQt. PyQt is a toolkit that allows developers to create graphical user interfaces for both desktop and mobile applications. It is made up of Python bindings for Qt, a set of C++ libraries and development tools that offer platform-independent abstractions. With PyQt, developers can use a wide range of libraries, tools, and widgets to build modern and sophisticated user interfaces.

To use PyQt with my algorithms, I first had to convert all the Jupyter Notebook files into Python files. Once I had all my algorithms in .py files, I created four classes: mainMenuUI, InputUI, and predPage, each for a separate page. The mainMenuUI class provides a graphical user interface (GUI) for the main menu, where users can select the algorithm they want to use for prediction. The InputUI class provides a GUI for users to input the required data to make a prediction.

In addition to these classes, I created a path() function to handle the paths to files required by the application. It checks if the application is running as an executable, and if so, it returns the path to the resource files required by the application. Otherwise, it returns the absolute path to the files.

A screenshot of a computer

Description automatically generated with medium confidence.

**Figure 18.** *Screen snippet of mainMenuUI*

The mainMenuUI class loads a UI file using the uic.loadUi() method. The method sets the title of the window to "Pre-Owned Car Price Predictor". The goToPage() method is called when the user clicks the "Select Algorithm" button on the main menu. It checks which algorithm radio button is selected and sets the algorithm attribute of the InputUI class accordingly. The method then sets the label text, the title of the window, and the current index of the stacked widget to the appropriate values.

Graphical user interface

Description automatically generated

**Figure 19.** *Screen snippet of inputUI*

The InputUI class also loads a UI file using the uic.loadUi() method. It initializes several variables, such as the modelEncoder, transmissionEncoder, fuelTypeEncoder, and scaler. These variables are used to pre-process the input data before making a prediction. The addModel() method populates the modelCombo combobox with the available car models for the selected brand. The runUI() method is called when the user clicks the "Predict" button. It retrieves the user input and preprocesses it using the variables initialized earlier. It then calls the appropriate machine learning algorithm to make a prediction.

The machine learning algorithms are implemented in their respective classes. Each class has its own fit() method to train the algorithm on the input data, a scalar function to normalise the data if needed, and a predict() method to make a prediction given new data.

To run these classes, I created a main method that creates an instance of the QApplication class. Then, I created instances of the mainMenuUI, InputUI, and predPage classes. I used PyQt's stacked widget functionality to switch between the main menu, the input page, and the prediction page. Finally, I executed the application using the exec\_() method of the QApplication class.

After testing that the GUI worked with all the algorithms and fixing all bugs, I converted the 'UI\_Interface.py' file into an executable file using a program called 'auto-py-to-exe'. This allowed anyone to use the program without any knowledge on running a python file, along with all the external files. Compiling the source code also reduces the file size and prevents people from stealing my code.

1. **Analysis and Evaluation**

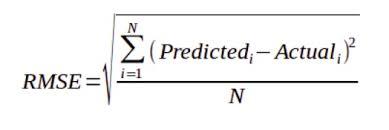
To analyse my algorithms, I held two forms of evaluation. Firstly, I used the Hold-out validation technique, which involves splitting the available data into two sets: a training set and a validation set. Secondly, I used the cross-validation technique, which divides the data into *K* subsets/folds.

Before testing my algorithms against each dataset, I set a limit of 10,000 rows of data for each dataset. This was to ensure that this was to ensure that the tests were equal and fair. Table 3 shows that each dataset had varying amounts of data, where Ford had more than 7,000 additional rows of data as compared to the Audi dataset.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Audi | BMW | Ford | Mercedes | Vauxhall | Volkswagen |
| Rows of Data | 10,668 | 10,781 | 17,965 | 13,119 | 13,632 | 15,157 |

**Table 3.** *Size of Datasets*

Once I had my data sets limited to 10,000 rows, I created an evaluation method that takes two arguments: the predictions, ‘X\_test’, and the original labels/prices, Y\_test. First, predictions are made using the predicted methods in the algorithms’ code, and stored to the variable ‘Y\_pred’. Then the RMSE value is calculated between ‘Y\_pred’ and ‘Y\_test’, and is printed/outputted for the user to see.

 (4) [31]

The RMSE value measures the difference between the predicted and actual values, and is expressed in the same units as the target values. The lower the RMSE, the greater the accuracy of the algorithm. It is calculated by taking the square root of the average of the squared differences between the predicted and actual values. To calculate the RMSE for my algorithms, I used equation (4), where N is the number of rows of test variables.

* 1. **Hold-Out Validation**

To conduct hold-out validation I split the data into a training subset that consisted of 75% of the data, and a validation subset that was made up of the remaining 25%. An advantage of the hold-out validation is that it is easy to implement and provides a quick estimation of model performance. To perform hold-out validation, my datasets were already split using Sci Kit Learn’s ‘train\_test\_split’ function, I then called my evaluation method using the split data.

However, there are some disadvantages to hold-out validation. The results obtained from a single split of the data can be sensitive to the particular way the data is split. Additionally, there is a risk of overfitting the model to the training set, which can result in poor performance on the validation set.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| RMSE Values Hold-out Validation (£) | **Audi** | **BMW** | **Ford** | **Mercedes** | **Vauxhall** | **Volkswagen** | **Average RMSE** |
| **Linear Regression** | 7764.08 | 6988.61 | 2931.35 | 7969.01 | 1928.56 | 3368.18 | **4991.16** |
| **Nearest Neighbour** | 2883.45 | 3216.30 | 1329.26 | 3304.55 | 919.40 | 1475.10 | **2188.06** |
| **Decision Tree** | 2928.44 | 3353.41 | 1526.27 | 3154.13 | 1152.03 | 1672.16 | **2297.74** |
| **Random Forest** | 2383.46 | 2781.36 | 1213.16 | 2484.42 | 939.84 | 1327.03 | **1854.88** |

**Table 4.** *RMSE values for Hold-Out Validation*

Table 4 shows the RMSE values calculated for each algorithm along with the average value. We can see that the Random Forest model still has the lowest average RMSE value of 1854.88, indicating that it performs the best across all car brands. The Nearest Neighbour model comes in second with an average RMSE of 2188.06, followed by the Decision Tree model with an average RMSE of 2297.74, and finally the Linear Regression model with an average RMSE of 4991.16.

When looking at the car brands, it is evident that the models performed the best with the ‘Vauxhall’ dataset, highlighted in green. This could be due to Vauxhall having a greater quality and spread of data. Whereas other brands like Audi, BMW, Ford and Mercedes have sports cars that can sometimes cause the data to spread.

In terms of the poorest performance, the models performed differently for each brand. For instance, the Linear Regression and Nearest Neighbour model performs poorly for Mercedes with RMSE values of 7969.01 and 3304.55, respectively. The Decision Tree and Random Forest model performs the worst for BMW with an RMSE value of 3353.41 and 2781.36, highlighted in red. Decision Tree and Random Forest models tend to perform poorly if the underlying patterns in the data are too complex and noisy. With the BMW dataset, these models may overfit the data and create overly complex decision trees that do not generalize well to new data. Whereas, the Linear Regression and Nearest Neighbour models are more effective in handling noisy data, as they rely on local data points to make predictions and can smooth out noisy data points.

It appears that the linear regression model performs well for some car brands and poorly for others. This may be due to some datasets having a linear relationship while others have a nonlinear relationship. As such, Linear Regression may not be a reliable algorithm for real-world scenarios, where the relationship between input variables and target variables is often nonlinear. On average, the linear regression model had an average RMSE value of 4991.16, and that is a huge amount of money that could be lost when purchasing a new car, making Linear regression a questionable algorithm to use in the process of buying a pre-owned car.

* 1. **Cross-Validation**

Cross-validation works by splitting the dataset into multiple subsets or folds, where each fold is used as a training and testing set. This process is critical because it allows the model to be tested on different data points and helps prevent overfitting.

The data is split into K folds, and the model is trained on (K-1) of the folds. This process is repeated K times, where each fold is used as a testing set once. By doing this, it ensures that each data point is used as both a training and testing data point. After each iteration, the results are recorded, and the mean is calculated. This provides an evaluation of the algorithms’ performance, which is not biased by the specific split of the data.

To perform the cross-validation test, I used Scikit Learn's KFold function and set the number of folds to 5.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| RMSE Values Cross-Validation (£) | **Audi** | **BMW** | **Ford** | **Mercedes** | **Vauxhall** | **Volkswagen** | **Average RMSE** |
| **Linear Regression** | 7160.07 | 7069.23 | 2870.60 | 8417.46 | 1990.11 | 3468.45 | **5162.65** |
| **Nearest Neighbour** | 8360.66 | 7623.08 | 3694.52 | 8597.36 | 2238.44 | 4052.86 | **5761.15** |
| **Decision Tree** | 2917.79 | 3453.76 | 1514.96 | 3280.74 | 1287.42 | 1664.44 | **2353.19** |
| **Random Forest** | 2364.66 | 2710.43 | 1240.68 | 2620.54 | 1048.89 | 1324.14 | **1884.89** |

**Table 5.** *RMSE values for Cross Validation*

Looking at table 5, we can see that it is quite similar to the results from the hold-out validation in table 5. The Random Forest model has the lowest RMSE value for all car brands, followed by the Decision Tree model, both of which outperforming the Linear Regression and Nearest Neighbour models.

In general, the RMSE values obtained from cross-validation are considered more reliable than those from hold-out validation. This is because cross-validation uses multiple train-test splits, whereas hold-out validation uses only one. As a result, cross-validation provides a better estimate of the model's performance on unseen data.

The average RMSE value for the nearest neighbour algorithm was affected the most in the cross-validation test, going from 2188.06 in the hold-out validation test to 5761.15 in the cross-validation test. This could have happened because the hold-out validation could have selected a more favourable training set, and in hold-out, the model is only tested on a fixed subset, leading to a lower RMSE. Overall, I believe the results from the cross-validation to produce a more accurate representation of the capabilities of each algorithm, and will be taking these results into high consideration.

* 1. **User Input**

To test how long each algorithm takes to present an answer, I used a controlled variable to measure the time taken to train and return a prediction, as well as the price it gave in return. The Vauxhall car model provided the most consistent results and worked well with all algorithms, so I used it for this test. As a random sample, I chose a 2017 Vauxhall Zafira with manual transmission, petrol fuel type, 30879 miles, £200 road tax, 41.5 mpg, a 1.4-liter engine, and a selling price of £10,230 from the test set.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Linear Regression** | **Nearest Neighbour** | **Decision Tree** | **Random Forest** |
| **Time (s)** | **0.1855** | **1.3193** | **56.6752** | **1158.8646**  **(19 mins 19 secs)** |
| **Predicted Price (£)** | **11,099.14** | **9,912.00** | **10,830.50** | **10,033.78** |

**Table 6.** *Time taken to run controlled variable.*

Looking at Table 6, we can see that the time it takes to train and predict the data varies among different algorithms. Linear regression is the quickest with a total time of only 0.1855 seconds, followed by nearest neighbour, decision tree, and random forest, which takes 19 minutes and 19 seconds.

Despite this, it is important to note that each algorithm has its own strengths and weaknesses, and the amount of time it takes to complete the process is only one factor to consider when choosing the most appropriate machine learning model for a particular task.

* 1. **Overall Analysis and Evaluation**

For my algorithm analysis, I used two forms of evaluation: hold-out validation and cross-validation. I tested them against different datasets of pre-owned cars. Random Forest had the lowest RMSE value, indicating that it performed the best across all car brands. The nearest neighbour algorithm followed, according to the hold-out validation, but the decision tree came second according to the cross-validation. Linear regression gave terribly low RMSE values of 4991.16 and 5162.65, but performed well in terms of time. This could be due to the datasets not having a linear relation.

Nearest neighbour had amazing results with the hold-out validation but poor results for the cross-validation. It is hard to recommend this algorithm when it sometimes produces great results and sometimes doesn’t. The nearest algorithm has to calculate many metric distances, so it takes an extremely long time to process all the calculations for a large dataset. A positive aspect of the nearest neighbour algorithm is that it’s instance-based learning, meaning it does not have to learn anything during the training period. Instead, it learns from the training set while making predictions. This also means that new data can be added to the dataset whenever, without the problem of having to train the algorithm again.

The decision tree and random forest algorithms both took an extremely long time. Building a tree requires data to be recursively partitioned into smaller and smaller subsets, which requires evaluating every possible split of each node in a tree to determine the best split that maximizes information gain. As the maximum depth of the tree increases, the number of possible splits exponentially increases, resulting in extremely long training times, especially with large datasets like the ones I used.

The random forest algorithm took much longer than the decision tree algorithm because it is an ensemble of decision trees. This improves the accuracy of the predictions greatly and reduces the risk of overfitting. However, considering the accuracy of it, it is definitely the most reliable. If I were to recreate this program, I would have the decision tree and random forest algorithms pre-trained on all datasets, so that users would not have to wait for huge amounts of time for their outcomes.

Overall, these results suggest that different algorithms may perform better or worse depending on the specific dataset being analyzed. The Random Forest and Decision Tree algorithms appear to perform well across all car brands, while Linear Regression may be less reliable for real-world scenarios where the relationship between input and target variables is often nonlinear. The nearest neighbor algorithm would be suited best when new variables are added, but not for large datasets because the calculation times would be ridiculously long.

One of my original ideas was to combine all the algorithms to create one overall prediction together, but seeing how differently they varied in RMSE values and seeing the continuous reliability of random forests, I realized that this would be extremely time costing and would not be beneficial.

For the executable file, I reduced the size of the dataset to 5000. Otherwise, it would take too long for the decision tree and random forests to train based on the dataset selected and output a result. This reduces the accuracy of the results greatly, and this is something I would like to improve on. Another thing I would like to improve on is the way I reduced the size of the datasets to 10,000 and split the data. I would improve this by randomly selecting 10,000 rows of data, rather than selecting the first 10,000. I would also increase the training set size to 90%, rather than 75%. This would largely increase the accuracy of data and reduce the bias.

As far as following my proposed timeline, proposed in my project plan, I didn’t follow it very well. I had underestimated how long it would take to understand and implement. In terms of completing research, I was able to do that in the proposed time. I had initially struggled to grasp the concept of how decision trees worked, with the concept of splitting with multiple features.

Additionally, following my proposed project timeline was difficult, as I had underestimated how long it would take to comprehend, improve, and implement each algorithm. Despite that, I managed to create four working algorithms, with each improving from the one before. The decision tree algorithm was a difficult concept to grasp and required slowing down to understand the how the trees were split and what determined their best split.

1. **Engineering tools**

For coding and testing my algorithms, I used Python and Jupyter Notebook. I chose Python because of its access to great libraries for machine learning, specifically SciKit-Learn and Pandas. SciKit-Learn is useful for handling basic ML tasks, such as normalizing and splitting my datasets into training and test sets. It also contains functions that calculate regression predictions for many machine learning algorithms, making it useful for comparing my algorithms. Another useful library was Pandas, which helped me handle and present my dataset it in a clear, viewable format. I used Python as it is very readable and understandable, which is important when handling complex code such as machine learning algorithms.

My main programming software was Jupyter Notebook, a web-based computing platform that clearly lays out code. It allowed me to illustrate all of my code and my analysis step-by-step all in one place, and also allows others to view and understand all of my code. Jupyter Notebook also contains all the libraries I needed, without me having to install each one on my personal computer.

Graphical user interface, website

Description automatically generated

**Figure 20.** *Screenshot of Py-Qt Designer*

Once I began creating my user interface, I switched to Visual Studio Code (VSCode), so that I could comfortably write my algorithms on Python files. I chose to use VSCode because it includes plenty of useful extensions for coding Python, including a library that allows me to access my Jupyter Notebooks on VSCode. To design my UI, I used Py-Qt Designer, which provided a helpful UI to design my own GUI, as shown in figure 20.

Graphical user interface, text, application, email

Description automatically generated

**Figure 21.** *Screenshot of auto-py-to-exe*

To convert my GUI into an executable file, I used a program called auto-py-to-exe, as shown in figure 21. It is a .py to .exe converter that uses a simple GUI and PyInstaller in Python. Not only did this make it much easier for me to make my file executable, it also saved me a lot of time by allowing me to avoid the tedious task of manually writing all the command lines needed to convert my code. This also allowed me to ensure that all the necessary features and files were included.

Graphical user interface, application

Description automatically generated

**Figure 22.** *Screenshot of Trello.*

In order to keep track of all the tasks I had to do during the project, I used Trello, a visual tool that allowed me to keep track of my to-do list, tasks in progress, and completed tasks. As shown in figure 22, I had three categories: "To Do", "Doing", and "Done", each with its own to-do list within them and a colour label to represent its importance and difficulty. I found that using Trello not only helped me stay organized and on track, but it also made it easier for me to see my progress and accomplishments.

When it came to committing and pushing my updates, I used GitHub Desktop. GitHub Desktop allowed me to interact with GitHub using a GUI instead of a command line. This tool was extremely useful as it shows all the changes I made to any specific file, making it easier for me to track my progress and see what I still needed to work on. Additionally, I found that using GitHub Desktop made the process of committing and pushing my updates much smoother and more efficient, as it eliminated the need to remember complex command lines and navigate through different branches manually.

1. **Professional Issues**

Machine learning algorithms have become increasingly popular across many industries, including the automobile industry, where it is used to predict the price of pre-owned cars. Although my algorithms have proven to be useful in predicting the prices of pre-owned cars, there are several professional issues that must be considered to ensure the accuracy and reliability of these predictions.

One of the main concerns is the quality of the data used to train the algorithm. To ensure the accuracy of the predictions, the data used to predict used car prices needs to be reliable and accurate. If the data is Incomplete or inaccurate, it can affect the accuracy of the predictions made. Therefore, it is crucial that the datasets used is reliable and represent the market of pre-owned cars extremely well, and includes a wide variety of different cars and models.

In the case of predicting the price of pre-owned cars, the selecting relevant features such as mileage, age, brand, model, transmission, MPG, and fuel type can have a significant impact on the accuracy of the predictions. Selecting irrelevant features can lead to overfitting or underfitting of the model. Therefore, it is necessary to select appropriate features.

Another concern is the selection of the appropriate algorithm for the task. There are various machine learning algorithms available for predicting used car prices, such as linear regression, decision trees, random forests, and nearest neighbour, etc. Some algorithms may perform better on certain types of data or features than others. Which is why choosing the right algorithm is crucial in achieving accurate and reliable results.

Bias can also be a significant issue when training machine learning algorithms. If the data used for training the algorithm is biased towards a particular type of car or model, it can lead to biased predictions. Which is why, it is necessary to ensure that the data used for training is represents the entire market of pre-owned cars.

Machine learning algorithms can often be opaque, meaning that you have no sense of how or why a result has been reached, making it difficult to understand how they make their predictions [32]. This lack of transparency and interpretability can be a significant issue in industries where decisions need to be explained or justified, such as in financial situations.

Finally, even though machine learning algorithms can automate tasks such as pricing pre-owned cars, this can lead to the loss of jobs and skills among people who were previously involved in the pricing process. There poses the risk of dehumanizing the decision-making process [33]. Therefore, it is essential to ensure that the use of machine learning algorithms does not have a negative impact on human employment or skills.

In conclusion, although machine learning algorithms can be extremely useful when it comes to predicting prices of pre-owned cars, there are several concerns that need to be considered. Addressing these concerns can lead to more accurate predictions and a better understanding of how the algorithm arrives at its predictions, whilst ensuring it sticks to a high ethical standard.

1. **Conclusion**

Predicting the price of used cars can be a challenging task due to several factors that affect it. My aim for this project was to find the most suitable machine learning algorithm capable of providing accurate and reliable predictions. I successfully produced four different ML algorithms, linear regression, nearest neighbour, decision tree, and random forest, from scratch and analysed them accurately using hold-out validation and cross-validation. From my cross validation - All the algorithms produced promising results, with my random forest algorithm producing the most accurate results with an RMSE of £1884.89, followed by my decision tree algorithm with an RMSE of £2353.19, linear regression with an RMSE of £5162.44 and nearest neighbour with an RMSE of £5761.15. From my evaluation, the results highlighted that factors like non-normalized data could have an adverse impact on the results of linear regression and nearest neighbor algorithms. On the other hand, decision trees and random forests do not need to consider scaling since they rely on a ranking system of a data point within a feature.

The main limiting factor during this project was the inability to use the datasets to their full extent. Using the full dataset for each brand would have allowed me to produce more accurate results, but it would also increase processing and training times. Unfortunately, I was limited by computational power, preventing me from processing large amounts of data. For example, some data had taken over 12 hours to train the random forest. Another method to increase the accuracy of my results would be to increase the size of the training set beyond the current 75% of the dataset used.

Although each algorithm produced promising results, there is still room for improvement. Firstly, I would make all my algorithms find their unique optimal parameters for each dataset, resulting in the most accurate results for each individual dataset. For my nearest neighbour code, I would adapt my code to run an evaluation process that automatically finds the optimal value of K, for the data set being used. This may induce a time-consuming process, as it would have to calculate millions of Euclidean distances. I would also like to integrate the ability to have my decision tree and random forest pre-trained so that the user does not have to wait an absurd amount of time for a single prediction.

In conclusion, I believe that this project successfully compares different types of machine learning algorithms in the context of pre-owned car prices. The results show that the random forest algorithm was the most accurate option, however, requiring long training times. Additionally, I learned that the linear regression model only works well with linear data patterns, and that nearest neighbour is the fastest option for adding new data to the dataset and receiving results.

1. **Bibliography**

[1] Zavolokina, L., Miscione, G. and Schwabe, G. (2019) “Buyers of ‘lemons’: How can a blockchain platform address buyers’ needs in the market for ‘lemons’?,” *Electronic Markets*, 30(2), pp. 227–239. Available at: https://doi.org/10.1007/s12525-019-00380-9

[2] Ayres, I. and Siegelman, P. (1995) *Race and Gender Discrimination in Bargaining for a New Car*, 85(4), pp. 304–321.

[3] Toosi, N.R. *et al.* (2020) “How culture and race shape gender dynamics in negotiation,” *Research Handbook on Gender and Negotiation*, pp. 260–280. Available at: https://doi.org/10.4337/9781788976763.00024.

[4] Kaggle.com. 2022. *100,000 UK Used Car Data set*. [online] Available at: *<https://www.kaggle.com/datasets/adityadesai13/used-car-dataset-ford-and-mercedes>* [Accessed 28 September 2022].

[5] Baheti, P. (2022) *Data preprocessing in machine learning [steps & techniques]*, *Data Preprocessing in Machine Learning [Steps & Techniques]*. Available at: https://www.v7labs.com/blog/data-preprocessing-guide#h1 (Accessed: October 2, 2022).

[6] *What is normalization in Machine Learning* (2021) *Deepchecks*. Available at: https://deepchecks.com/glossary/normalization-in-machine-learning/ (Accessed: October 4, 2022).

[7] Bhandari, A. (2022) *Feature scaling: Standardization vs normalization*, *Analytics Vidhya*. Available at: https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/ (Accessed: October 20, 2022).

[8] Tingle, M. (2021) *Preventing data leakage in your machine learning model*, *Medium*. Towards Data Science. Available at: https://towardsdatascience.com/preventing-data-leakage-in-your-machine-learning-model-9ae54b3cd1fb (Accessed: October 16, 2022).

[9] IBM (2022) *k-Nearest Neighbors* , *K-Nearest Neighbors (KNN)*. IBM. Available at: https://www.ibm.com/docs/en/ias?topic=procedures-k-nearest-neighbors-knn (Accessed: March 24, 2023).

[10] IBM (2021) *Decision Tree Model Rules*, *Decision tree model rules*. IBM. Available at: https://www.ibm.com/docs/en/spss-modeler/18.0.0?topic=nuggets-decision-tree-model-rules (Accessed: March 31, 2023).

[11] *What is the K-nearest neighbors algorithm?* (no date) *IBM*. Available at: https://www.ibm.com/uk-en/topics/knn#:~:text=The%20k%2Dnearest%20neighbors%20algorithm%2C%20also%20known%20as%20KNN%20or,of%20an%20individual%20data%20point. (Accessed: October 17, 2022).

[12] Seber, G.A.F. and Lee, A.J. (2003) *Linear regression analysis, 2nd edition*. 2nd edn. John Wiley & Sons.

[13] Montgomery, D.C., Peck, E.A. and Vining, G.G. (2013) *Introduction to linear regression analysis*. 5th edn. Hoboken: John Wiley & Sons, Inc.

[14] Javatpoint (2021) *Linear regression in machine learning - javatpoint*, *www.javatpoint.com*. Javatpoint. Available at: https://www.javatpoint.com/linear-regression-in-machine-learning (Accessed: March 22, 2023).

[15] James, G. *et al.* (2021) “An Introduction to Statistical Learning with Applications in R,” *Springer Texts in Statistics*, 2, pp. 303–332. Available at: https://doi.org/10.1007/978-1-0716-1418-1\_1.

[16] Christophe , J. (no date) *K-Nearest Neighbors (KNN) in Python*, *JC Chouinard*. Available at: https://www.jcchouinard.com/k-nearest-neighbors/ (Accessed: November 23 2022).

[17] Singh, A. (2022) *K-Nearest Neighbors Algorithm: KNN Regression Python*, *Analytics Vidhya*. Available at: https://www.analyticsvidhya.com/blog/2018/08/k-nearest-neighbor-introduction-regression-python/ (Accessed: November 7, 2022).

[18] *Euclidean distance formula - derivation, examples* (no date) *Cuemath*. Available at: https://www.cuemath.com/euclidean-distance-formula/ (Accessed: October 19, 2022).

[19] Bolandraftar, M. (2013) “Application of K-nearest neighbor (KNN) approach for predicting economic events theoretical backgroun,” *Journal of Engineering Research and Applications* , 3(5), pp. 605–610.

[20] Patel, H.H. and Prajapati, P. (2018) “Study and analysis of decision tree based classification algorithms,” *International Journal of Computer Sciences and Engineering*, 6(10), pp. 74–78. Available at: https://doi.org/10.26438/ijcse/v6i10.7478.

[21] Zach (2021) *RMSE vs. R-squared: Which metric should you use?*, *Statology*. Available at: https://www.statology.org/rmse-vs-r-squared/ (Accessed: December 3, 2022).

[22] Jauregui, A.F. (2022) *How to code decision tree in python from scratch*, *Ander Fernández*. Available at: https://anderfernandez.com/en/blog/code-decision-tree-python-from-scratch/ (Accessed: November 28, 2022).

[23] Jordan, J. (2022) *Decision trees.*, *Jeremy Jordan*. Jeremy Jordan. Available at: https://www.jeremyjordan.me/decision-trees/ (Accessed: December 2, 2022).

[24] *Decision trees in python - step-by-step implementation* (2022) *AskPython*. Available at: https://www.askpython.com/python/examples/decision-trees (Accessed: December 3, 2022).

[25] Prasad, A.M., Iverson, L.R. and Liaw, A. (2006) “Newer classification and Regression Tree Techniques: Bagging and random forests for ecological prediction,” *Ecosystems*, 9(2), pp. 181–199. Available at: https://doi.org/10.1007/s10021-005-0054-1.

[26] Mwiti, D. (2023) *Random Forest regression: When does it fail and why?*, *neptune.ai*. neptune.ai. Available at: https://neptune.ai/blog/random-forest-regression-when-does-it-fail-and-why#:~:text=Each%20tree%20is%20created%20from,to%20produce%20a%20single%20result. (Accessed: March 16, 2023).

[27] Shanmugasundar, G. *et al.* (2021) “A comparative study of linear, random forest and AdaBoost regressions for modeling non-traditional machining,” *Processes*, 9(11), pp. 1–14. Available at: https://doi.org/10.3390/pr9112015.

[28] Manjuhariashok (2020) *All about random forests and handling missing values in them.*, *Numpy Ninja*. Numpy Ninja. Available at: https://www.numpyninja.com/post/all-about-random-forests-and-handling-missing-values-in-them#:~:text=Random%20Forests%20and%20Missing%20Data%20%3A&text=This%20forces%20researchers%20who%20want,lost%20and%20inferential%20power%20compromised. (Accessed: March 18, 2023).

[29] Elg, E. (2020) *Jump start your modeling with random forests*, *Elder Research*. Available at: https://www.elderresearch.com/blog/jump-start-your-modeling-with-random-forests/ (Accessed: March 15, 2023).

[30] Auret, L. and Aldrich, C. (2012) “Interpretation of nonlinear relationships between process variables by use of random forests,” *Minerals Engineering*, 35, pp. 27–42. Available at: https://doi.org/10.1016/j.mineng.2012.05.008.

[P] Aakash, M. (2020) *Root-mean-square error in R programming*, *GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/root-mean-square-error-in-r-programming/ (Accessed: March 23, 2023).

[32] Burrell, J. (2016) “How the machine ‘thinks’: Understanding opacity in machine learning algorithms,” *Big Data & Society*, 3(1), pp. 1–12. Available at: https://doi.org/10.1177/2053951715622512.

[32] Grodzinsky, F.S., Miller, K.W. and Wolf, M.J. (2010) “Developing artificial agents worthy of trust: ‘would you buy a used car from this artificial agent?,’” *Ethics and Information Technology*, 13(1), pp. 17–27. Available at: https://doi.org/10.1007/s10676-010-9255-1.

1. **Appendix**
   1. **Video links**

ML Pre-Owned Car Predictor GUI: <https://youtu.be/kc0Oxqno0Ys>

* 1. **My diary**

|  |  |
| --- | --- |
| Date | Tasks |
| 27/10/2022 | - Begun to research into the different types of Machine Learning Algorithms  - looked in K-Nearest Neighbour, how it works and how to implement it |
| 28/10/2022 | - Installed scikit-learn  - Added dataset to repository |
| 01/11/2022 | - Started coding to Jupyter Notebook - so that all data can be visualised/compiled in one place  - Researched the seaborn library to potentially visualise data on informative statistical graphic  - Also Researched Pandas library - to potentially analyse datasets  - split the data set using scikit-Learn  - Created a method to work out the Euclidean distance between two points  - Created a method to work out the KNN |
| 03/11/2022 | - Trying to convert .csv file into a .txt file - because reading the data set file by row has become difficult  - Created a sort method to sort out the distances in ascending order  - Used numpy's genfromtxt instead, but dataset is too large and takes too long to process |
| 04/11/2022 | - Cleaned the data: getting rid of model, transmission and fuel type.  - Broke down dataset into first 200 entries, because dataset is too large to process  - Created method for 1NN to test  - Numpy's genfromtxt doesn't like the engineSize and tax variables - so I have left them out for the time being |
| 05/11/2022 | - Created a prediction method to predict the price of the cars  - Results from prediction method are returning in an unwanted format  - Using Pandas library to read from library - for efficiency and better readings from the dataset  - Created a seaborn heatmap to visualise dataset  - Looked into Seaborn's distplots graph |
| 17/11/2022 | - Began writing the evaluation method for KNN - is currently suited for classification, not regression  - Began work writing interim report. |
| 18/11/2022 | - Began writing the methodology in the interim report |
| 21/11/2022 | - Begun writing description of KNN algorithm to Interim report |
| 23/11/2022 | - Begun coding the decision tree algorithm: imported & split data, method using scikit-learn, entropy method  - Begun writing description of decision algorithm to Interim report  - Attempted to visualise the decision tree |
| 25/11/2022 | - Normalised data for KNN  - Added preprocessing and normalization to Interim report  - Removed unnecessary .txt files |
| 26/11/2022 | - Fixed issued with reading normalised data  - Added code examples to interim report |
| 29/11/2022 | - Completed the evaluation method for KNN, using RMSE |
| 30/11/2022 | - Changed how I reduced data set to increase accuracy by 0.3 at peak  - Used Pandas feature "get\_dummies()" to generate dummy variables for model, transmission and fuel type  - Created a user input to get prediction of their own car |
| 01/12/2022 | - Removed "get\_dummies()" and replaced with "LabelEncoder()"  - Completed user prediction input - fully working |
| 02/12/2022 | - Fixed an issue with prediction, where predictions were completely wrong - fixed by normalising user input |
| 05/12/2022 | - Worked on decision tree algorithm - understanding what each method does  - Calculated how many times Euclidean distance is calculated and how long a single prediction takes  - Described my KNN algorithm code in the interim report |
| 06/12/2022 | - Created Decision tree class  - Initialised values  - Created fit method to put features and label together  - Created start of build tree method  - Created method to find the best split |
| 07/12/2022 | - Created a method to find split tree into branches - extension of best split method  - Using template entropy method to calculate entropy of a node - temporary to test later methods  - Created method to calculate information gain  - Completed best split and tree build methods  - Created method to predict the price of a car - currently not working fully |
| 08/12/2022 | - Fixed predict and best split methods  - Created method to evaluate and calculate the RMSE of the decision tree  - Completed Interim report |
| 16/12/2022 | - KNN and decision tree algorithm work with all car models on dataset  - Begun Final report  - Created a Trello board to keep track of tasks |
| 17/12/2022 | - Installed PyQt5 to build GUI |
| 19/12/2022 | - Created a PyQt file and tested UI design  - Created NN.py file for the K-Nearest Neighbour algorithm  - Modified code of nearestNeighbour.ipynb that allows other files to access functions from the file  - created "runUI.py" to run the GUI |
| 20/12/2022 | - Created an appealing UI design and a main page  - Main page with with radio buttons to choose which ML algorithm to use  - Created a page for the KNN algorithm  - Created a page to clearly state the predicted price |
| 21/12/2022 | - Created DT.py file for the decision tree algorithm  - Modified code of decision\_tree.ipynb that allows other files to access functions from the file  - Created a separate UI page for the decision tree algorithm  - Then made the decision tree and KNN algorithms use the same UI page - to reduce code |
| 29/12/2022 | - Converted GUI into a single executable file, so that it can be run on any Windows system  - Faced problem with executable file not detecting the .ui files - fixed by putting UI files in a folder |
| 23/01/2023 | - Begun research into Random Forest |
| 25/01/2023 | - Continuted research into Random Forest - taking notes  - Created Jupyter Notebook page for Random Forest Algorithm  - Perfromed GridSearchCV for Decision Tree to find the optimal parameters  - Applied GridSearchCV parameters to my decision tree |
| 29/01/2023 | - Created Random Forest class with \_\_init\_\_ method |
| 30/01/2023 | - Imported, cleaned and split the dataset for the Random Forest code  - Working on my Random Forest algorithm  - testing bagging technique |
| 01/02/2023 | - Created a fit method for the Random Forest algorithm |
| 03/02/2023 | - Created a predict method for the Random Forest algorithm  - Completed first draft of the Random Forest algorithm - it prints a different price each time |
| 04/02/2023 | - GridSearchCV performed on Random Forest algorithm  - Further testing to solve problem with Random Forest algorithm, where the max depth can't be increased above 5  - Re-implemented the entropy method into the Decision tree algorithm to test whether it improves accuracy |
| 05/02/2023 | - Removed entropy method from Decision Tree algorithm  - Found optimal parameters for Random Forest algorithm - still faced problems where the max depth can't be increased above 5 |
| 16/02/2023 | - Changed range of suitable parameters to limit the max depth to 6 - due to unknown error  - change range of Decision trees parameters to the same range of the random forest parameters - to keep controlled varriables similar |
| 17/02/2023 | - Solved problem with random forest algorithm - the max depth can now be increased above 5  - Created a method to evaluate the accuracy of the Random Forest algorithm  - Fixed a problem with outputting the predicted price of the Random Forest algorithm - where only the output from one tree was Used  - Removed random state from the sample method - as it was causing the same sample to be used each time |
| 22/02/202 | - Worked on optimising the random forest algorithm  - Created a method to calculate the RMSE of the Random Forest algorithm |
| 05/03/2023 | - Wrote up random forest on the final report |
| 06/03/2023 | - Solved issue and improved Random Forest code accuracy, by changing random state value to None. |
| 07/03/2023 | - Created more data visualisations  - Begun working on and completed first draft of Linear Regrrssion algorithm |
| 08/03/2023 | - Begun working in Suport Vector Machine algorithm |
| 12/03/2023 | - Gotten rid out outliers in dataset  - Created more data visualisations - of outliers |
| 13/03/2023 | - Added function to remove outliers in all algorithms  - Begun write up on outliers  - Wrote up more on data visualisation  - Begun write up on User Interface |
| 15/03/2023 | - Compacted and cleaned the .py code for all algorithms  - Moved common functions to the UI\_Interface.py file  - Cleaned the code on the jupyter notebook files  - Organised Folders  - Created a .py file for Linear Regression algorithm |
| 16/03/2023 | - Improved and completed UI files  - Completed Linear Regression .py files  - Added Linear Regression algorithm to the UI |
| 18/03/2023 | - Attempted to combine all the algorithms into one |
| 25/03/2023 | - Started write up on Linear Regression algorithm  - Completely re-coded Linear Regression algorithm |
| 26/03/2023 | - Completed write up on Linear Regression |
| 27/03/2023 | - Completed write up on Random Forests  - Changed write up on KNN |
| 28/03/2023 | - Completed write up on user interface  - Completed write up on professional issues  - Completed write up on Engineering Tools |
| 29/03/2023 | - Completed Hold-out validation analysis on all algorithms  - KNN took extremely long to run. I had to run on multiple devices  - Completed write up on hold-out validation |
| 30/03/2023 | - Completed K-Fold validation analysis on all algorithms  - Got rid of combined algorithm file  - Completed Analsis write up  - Completed evaluation  - Completed conclusion  - Commented all code  - Completed final report |