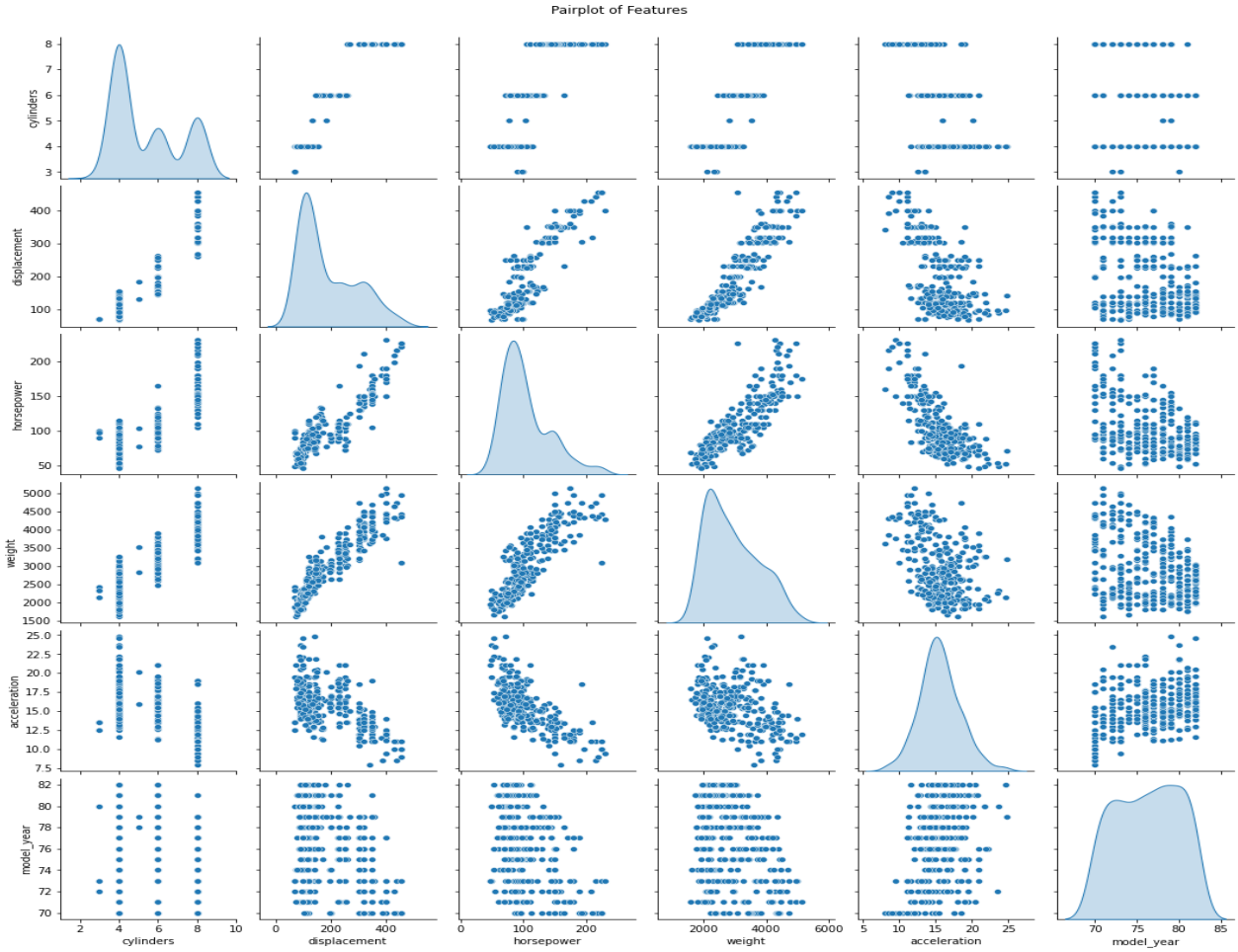
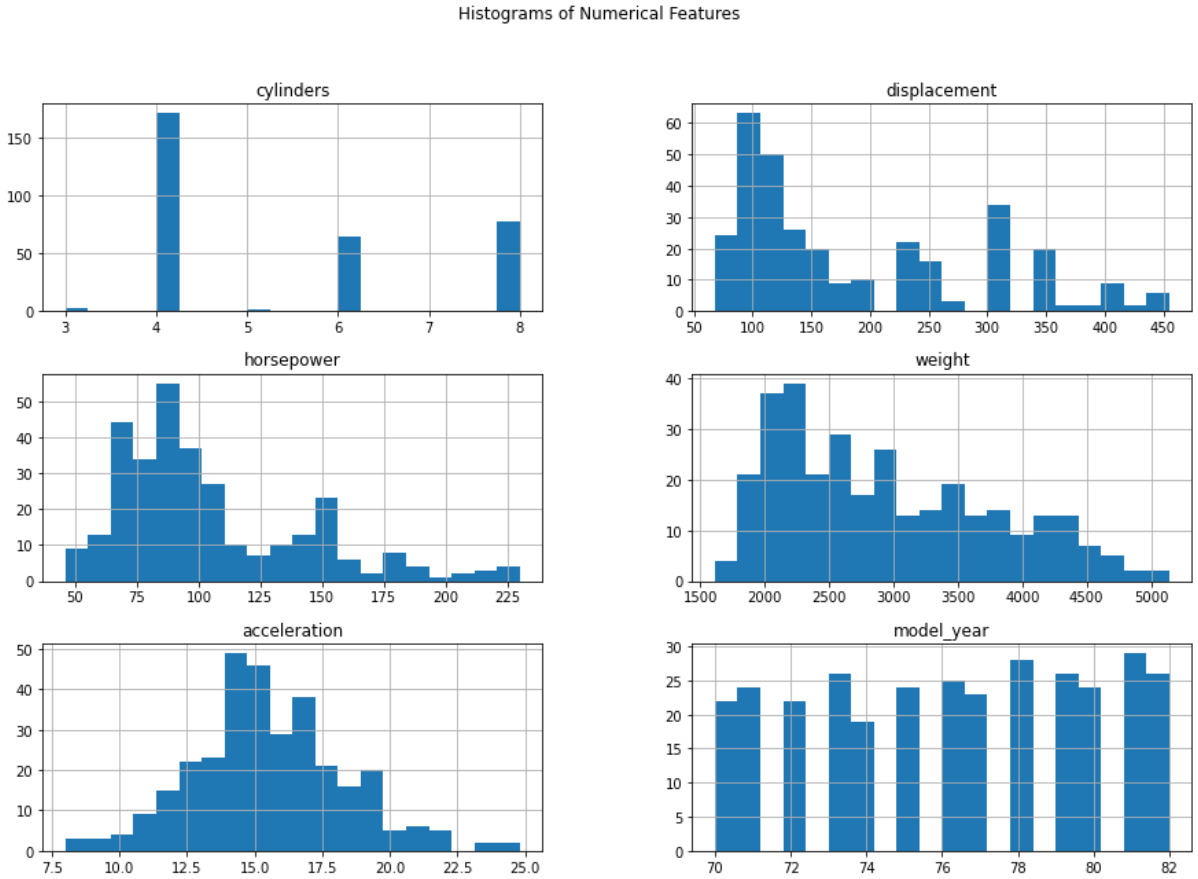
Using the method df.describe() we can isolate the total count of values in each given feature. In this case we have a total of 318 rows. However, we have 6 missing values from the feature ‘horsepower’. This means that we will need some sort of preprocessing done to impute these missing values. We have a ‘model_year’ feature which tells us the year of make for our cars, because our linear regression model doesn’t have a time series component to it this feature although represented as a value should be treated as a categorical variable, the ‘model_year’ variable doesn’t show any correlation with any continuous variables I will encode it. The cylinders variable is numeric but has a meaningful order and very few values, in addition model_year could be treated as an ordinal feature because of the older cars possibly correlating to lower value I Looking at our features, there is one which stands out as not providing much information towards our target value and contains a ton of unique categorical values. I’m referring to the ‘name’ feature, this feature may be useful in an SQL generated database as a key identifier but in the case for our linear regression model’s training set, I am going to remove this redundant class in favour of less dimensions for our model to deal with. Because there are only 6 missing values in our entire training set that makes up less than 2% of a column’s data so we could justify simply deleting values. However, since the specified task asks to implement an imputation method, I believe the best approach is to use simple imputer with the mean as the replacement parameter because



‘horsepower’ is a feature with continuous variables in it. I would argue it is superior to KNN imputation in this scenario because we don’t need to add more computational complexity for the sake of imputing only 6 values. We can identify from this pair plot a few variables which may contain a linear relationship such as weight and displacement as well as horsepower and weight show a relatively strong positive correlation. We can also see the points form a somewhat negative correlation line in the horsepower and acceleration relationship.



Looking at our histograms of numerical features, the cylinders and model_year which don’t contain continuous values don’t show useful information regarding distribution. However, we see the acceleration values are somewhat normally distributed, where as horsepower, displacement and weight have a right skewed or positive skewed distribution. For data preprocessing I have chosen to use the simple mean method of Simple Imputer for my continuous variables, majority of the distributions are not exactly normal so I will use standard scaler to standardise the continuous variables in this case. I have defined cylinders and model_year as ordinal data with the relevant inherent ordering as they don’t fit with the other continuous variables but hold a meaningful intrinsic order. Both Ordinal and Categorical variables are regularly encoded as there are no more missing values I have identified in the dataset. I have chosen to use onehotencoder to handle the categorical variables without innate order and convert nominal data to numeric. For similar reasons my two ordinal variables will be encoded with the ordinal encoder to maintain their intrinsic order but will be encoded more appropriately as ordinal variables.

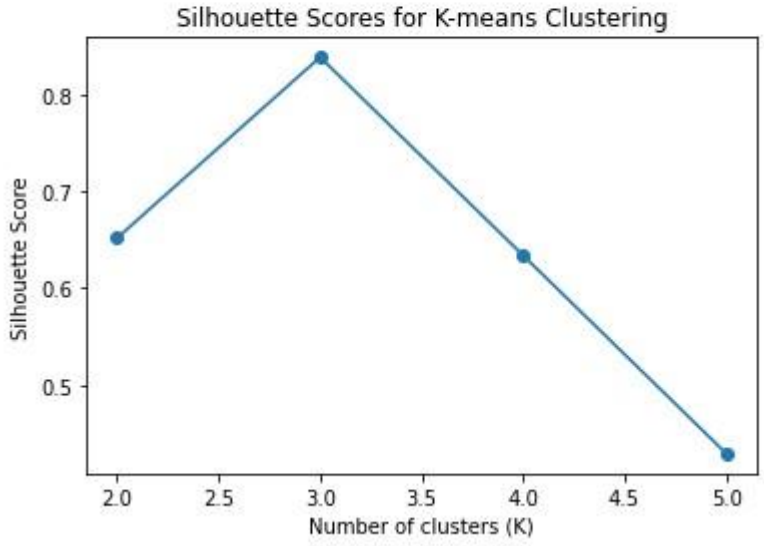
```
Coefficients:
 [ 4.86471469  2.55434423 -0.94902181 -5.71298884  0.39433463  2.77768148
  0.62283369  1.00918318 -1.63201687  0.         -4.2196476 ]
Mean squared error: 9.46
R squared: 0.85
```

We have a good R squared total for our linear regression model as our value is less than one. We have a mean squared error of 9.46 and Coefficients as presented above.

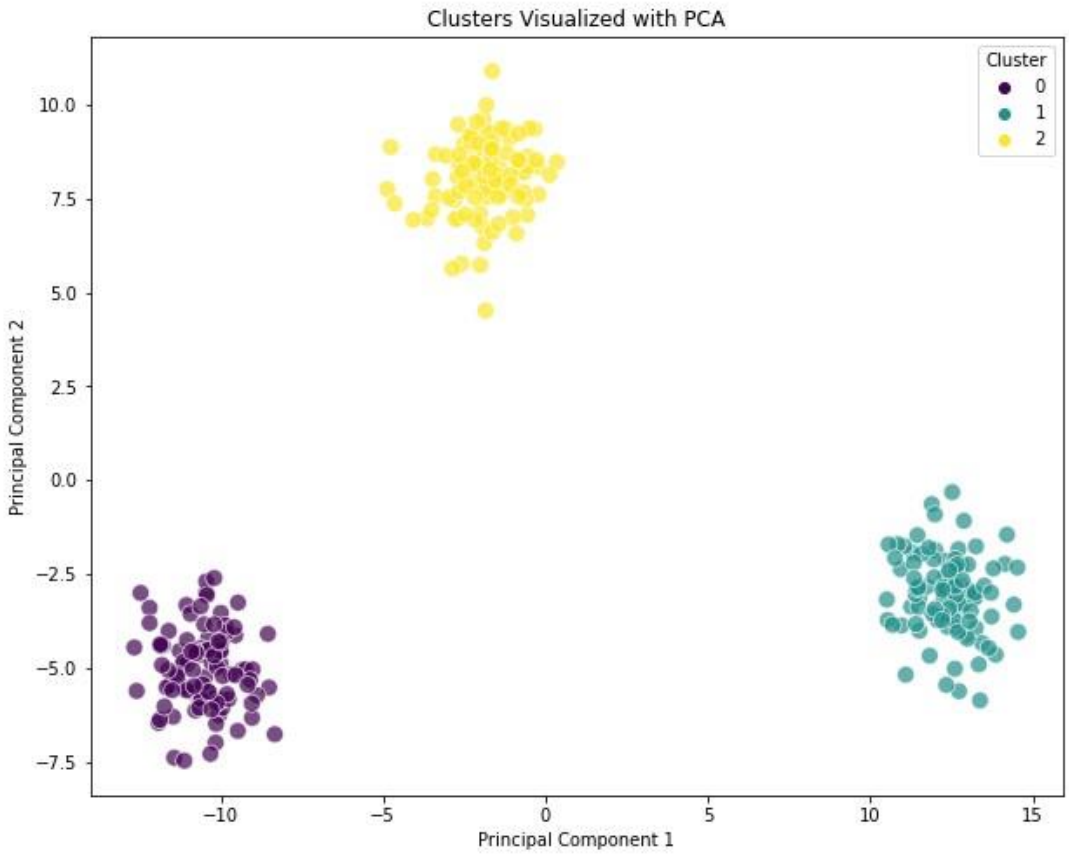
Clustering

Now we are going to be constructing a machine learning model focused on unsupervised learning where we no longer have target labels in our dataset. The first algorithm we use is the K means algorithm. This algorithm randomly selects points depending on the parameter selected centres (referred to as k) (in our case 3). With this 3 centroids are selected and iteratively our centroid is updated to be at the mean of all objects closest to that centroid until no more changes the centroids needs to be made. We can evaluate the performance of k means with the silhouette score metric.

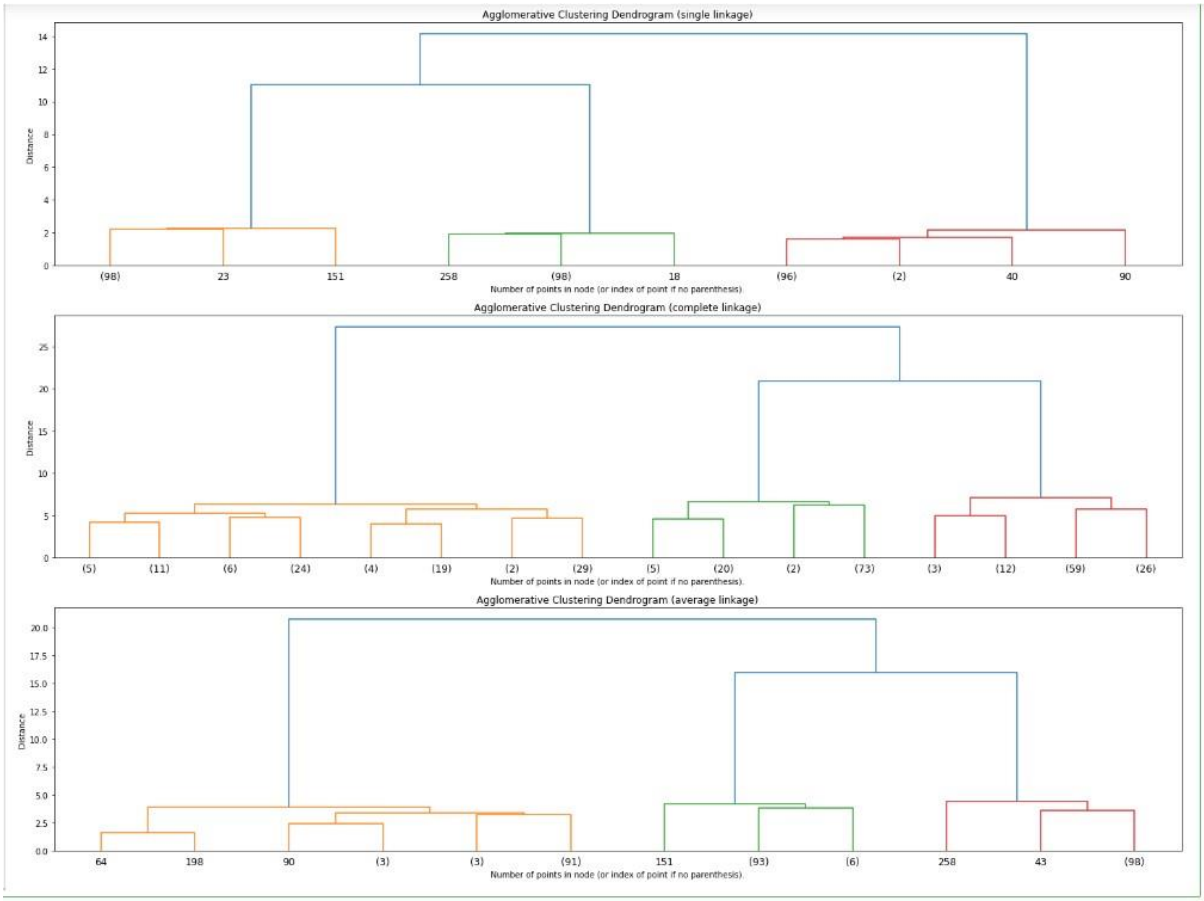
```
For n_clusters = 2, the silhouette score is 0.652
For n_clusters = 3, the silhouette score is 0.838
For n_clusters = 4, the silhouette score is 0.634
For n_clusters = 5, the silhouette score is 0.429
```



Our performance is by far at its best when k = 3 with a silhouette score of 0.84.



Here is a visualisation of our two principle components in the form of a scatterplot where the legend of each cluster corresponds to each colour. The intra cluster and inter cluster distances seem to be quite optimal making it unsurprising we have quite good performance in this case. Hierarchical clustering has a major benefit in not being any where near as computational costing as the K means algorithm. This is due to the iterative nature of k means must rerun each time it needs to update the centroid and find the best point to update as the centroid. Whereas with hierarchical clustering we simply update the threshold and we cut our corresponding dendrogram to change our algorithm making it more efficient. Hierarchical clustering can also be effective with categorical variables whereas it is more difficult to define the distance between categorical objects in k means clustering. However, hierarchical clustering can become nested within clustering done from cutting the dendrogram from a larger height. For dendrogram plot some code used from https://scikitlearn.org/stable/auto_examples/cluster/plot_agglomerative_dendrogram.html . K means is also a very simple algorithm that works well with large sample sizes (given they are numerical). Below we can see a visual representation of our 3 different linkage method.



Neural Networks

To scale the points from 0 to 1, I've standardised the data using sklearn's minmaxscaler function. Data split into 80-20 split and put into tensor variables. For my DataLoader objects I decided to set shuffle to false to keep reproducibility (shuffle parameter shuffles values after every epoch, iteration over entire dataset). Utilizing Pytorch's nn.linear package I can declare my constructor with my input, hidden and output layers and initialize them without having to explicitly declare the weights and biases (these are done by PyTorch). Our hidden layers have been predefined as 180 neurons, intuitively with this problem because we are dealing with images made up of 8x8 pixels we naturally will have an input size of 64 for our input layer. Our output layer will be 10 because we are expecting a digit from 0 – 9 hence 10 possible outputs. For my optimizer I have chosen to go with the adam algorithm, which has the benefits of both AdaGrad and RMSProp. Maintaining two moving averages for each parameter; one for gradients and one for the square of gradients, which allows the algorithm to adjust the learning rate of each parameter. A learning rate is extremely important in a neural network as it decides how big of a step in updating our bias and weights are when backward passing. However, a larger learning rate is not always a good thing as we can jump over our optimal performance whereas when it is too small we can fixate on what we think is our optimal performance when we may have been able to achieve greater performance.

```
Epoch [1/15], Loss: 1.4713, Accuracy: 99.10%
Epoch [2/15], Loss: 1.4702, Accuracy: 99.37%
Epoch [3/15], Loss: 1.4688, Accuracy: 99.37%
Epoch [4/15], Loss: 1.4689, Accuracy: 99.37%
Epoch [5/15], Loss: 1.4685, Accuracy: 99.37%
Epoch [6/15], Loss: 1.4683, Accuracy: 99.37%
Epoch [7/15], Loss: 1.4682, Accuracy: 99.37%
Epoch [8/15], Loss: 1.4683, Accuracy: 99.37%
Epoch [9/15], Loss: 1.4687, Accuracy: 99.37%
Epoch [10/15], Loss: 1.4681, Accuracy: 99.37%
Epoch [11/15], Loss: 1.4680, Accuracy: 99.37%
Epoch [12/15], Loss: 1.4698, Accuracy: 99.37%
Epoch [13/15], Loss: 1.4691, Accuracy: 99.30%
Epoch [14/15], Loss: 1.4679, Accuracy: 99.37%
Epoch [15/15], Loss: 1.4682, Accuracy: 99.37%
```

```
Test Accuracy: 0.9500
Example Predictions:
Predicted: 6, Actual: 6
Predicted: 1, Actual: 1
Predicted: 9, Actual: 9
Predicted: 7, Actual: 7
Predicted: 9, Actual: 9
```

The predictions and training accuracy is seemingly very high, meaning that our simple multi-layer perceptron neural network is well suited to this image prediction task. Changing the number of neurons or hidden layers in a neural network can have significant implications on the performance of the neural network. This is because too many neurons lead to overfitting, even in the case of our 128 seemingly our training accuracy is perfectly fitting the data each epoch suggesting it may be overfitting this dataset. However more neurons also means the capacity of the model is increased as more neurons means the model can handle more features etc. It is important to find a balance between the right capacity of the model to solve the desired problem but not increasing computation time too significantly.