u2053390

February 1, 2023

[2]:
```
%pip install numpy pandas matplotlib seaborn tensorflow keras xgboost
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: numpy in
/home/elshrimpo/.local/lib/python3.10/site-packages (1.23.4)
Requirement already satisfied: pandas in
/home/elshrimpo/.local/lib/python3.10/site-packages (1.5.1)
Requirement already satisfied: matplotlib in
/home/elshrimpo/.local/lib/python3.10/site-packages (3.6.0)
Requirement already satisfied: seaborn in
/home/elshrimpo/.local/lib/python3.10/site-packages (0.12.1)
Requirement already satisfied: tensorflow in
/home/elshrimpo/.local/lib/python3.10/site-packages (2.11.0)
Requirement already satisfied: keras in
/home/elshrimpo/.local/lib/python3.10/site-packages (2.11.0)
Requirement already satisfied: xgboost in
/home/elshrimpo/.local/lib/python3.10/site-packages (1.7.3)
Requirement already satisfied: python-dateutil>=2.8.1 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/lib/python3/dist-packages
(from pandas) (2022.1)
Requirement already satisfied: cycler>=0.10 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from matplotlib) (4.37.4)
Requirement already satisfied: pyparsing>=2.2.1 in /usr/lib/python3/dist-
packages (from matplotlib) (2.4.7)
Requirement already satisfied: contourpy>=1.0.1 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from matplotlib) (1.0.5)
Requirement already satisfied: pillow>=6.2.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from matplotlib) (9.2.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from matplotlib) (1.4.4)
Requirement already satisfied: packaging>=20.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from matplotlib) (21.3)
Requirement already satisfied: termcolor>=1.1.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (2.1.1)
Requirement already satisfied: tensorflow-estimator<2.12,>=2.11.0 in
```

```
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (2.11.0)
Requirement already satisfied: tensorboard<2.12,>=2.11 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (2.11.0)
Requirement already satisfied: libclang>=13.0.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (14.0.6)
Requirement already satisfied: typing-extensions>=3.6.6 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (4.4.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (1.50.0)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (0.28.0)
Requirement already satisfied: protobuf<3.20,>=3.9.2 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (3.19.6)
Requirement already satisfied: wrapt>=1.11.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (1.14.1)
Requirement already satisfied: six>=1.12.0 in /usr/lib/python3/dist-packages
(from tensorflow) (1.16.0)
Requirement already satisfied: absl-py>=1.0.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (1.3.0)
Requirement already satisfied: google-pasta>=0.1.1 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (0.2.0)
Requirement already satisfied: gast<=0.4.0,>=0.2.1 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (0.4.0)
Requirement already satisfied: astunparse>=1.6.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (1.6.3)
Requirement already satisfied: h5py>=2.9.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (3.7.0)
Requirement already satisfied: flatbuffers>=2.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (22.10.26)
Requirement already satisfied: opt-einsum>=2.3.2 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from tensorflow) (3.3.0)
Requirement already satisfied: setuptools in /usr/lib/python3/dist-packages
(from tensorflow) (59.6.0)
Requirement already satisfied: scipy in
/home/elshrimpo/.local/lib/python3.10/site-packages (from xgboost) (1.9.2)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/lib/python3/dist-
packages (from astunparse>=1.6.0->tensorflow) (0.37.1)
Requirement already satisfied: markdown>=2.6.8 in /usr/lib/python3/dist-packages
(from tensorboard<2.12,>=2.11->tensorflow) (3.3.6)
Requirement already satisfied: requests<3,>=2.21.0 in /usr/lib/python3/dist-
packages (from tensorboard<2.12,>=2.11->tensorflow) (2.25.1)
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from
tensorboard<2.12,>=2.11->tensorflow) (0.6.1)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from
tensorboard<2.12,>=2.11->tensorflow) (1.8.1)
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in
```

```
/home/elshrimpo/.local/lib/python3.10/site-packages (from
tensorboard<2.12,>=2.11->tensorflow) (0.4.6)
Requirement already satisfied: google-auth<3,>=1.6.3 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from
tensorboard<2.12,>=2.11->tensorflow) (2.14.1)
Requirement already satisfied: werkzeug>=1.0.1 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from
tensorboard<2.12,>=2.11->tensorflow) (2.2.2)
Requirement already satisfied: cachetools<6.0,>=2.0.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from google-
auth<3,>=1.6.3->tensorboard<2.12,>=2.11->tensorflow) (5.2.0)
Requirement already satisfied: rsa<5,>=3.1.4 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from google-
auth<3,>=1.6.3->tensorboard<2.12,>=2.11->tensorflow) (4.9)
Requirement already satisfied: pyasn1-modules>=0.2.1 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from google-
auth<3,>=1.6.3->tensorboard<2.12,>=2.11->tensorflow) (0.2.8)
Requirement already satisfied: requests-oauthlib>=0.7.0 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from google-auth-
oauthlib<0.5,>=0.4.1->tensorboard<2.12,>=2.11->tensorflow) (1.3.1)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from
werkzeug>=1.0.1->tensorboard<2.12,>=2.11->tensorflow) (2.1.1)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in
/home/elshrimpo/.local/lib/python3.10/site-packages (from
pyasn1-modules>=0.2.1->google-
auth<3,>=1.6.3->tensorboard<2.12,>=2.11->tensorflow) (0.4.8)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/lib/python3/dist-packages
(from requests-oauthlib>=0.7.0->google-auth-
oauthlib<0.5,>=0.4.1->tensorboard<2.12,>=2.11->tensorflow) (3.2.0)
Note: you may need to restart the kernel to use updated packages.
```

1 Auto-mpg dataset

We are predicting MPG based off of other features in the dataset: - displacement (countinous) - horsepower (countinous) - weight (countinous) - acceleration (countinous) - cylinders (discrete) - model year (discrete) - origin (discrete) - car name (unique string)

We can already see that some features are going to be more important than others, and that some features are going to be more difficult to work with than others. For example, the car name is a unique string, and we thus can't use that as a feature without some sort of encoding, we will evaluate if this feature is worth keeping or not. To understand the features better we can import the dataset and have a look at it.

[3]:
```
import pandas as pd
# Load dataset
```

```
# I've done some preprocessing on the dataset to convert it to a csv file, from␣
 ↪a tsb file (and replaced missing values (?) with nothing so they are treated␣
 ↪as NA values)
df = pd.read_csv("data/auto-mpg.csv")
print(df.head())
print(df.isna().sum())
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0  3504.0          12.0
1  15.0          8         350.0       165.0  3693.0          11.5
2  18.0          8         318.0       150.0  3436.0          11.0
3  16.0          8         304.0       150.0  3433.0          12.0
4  17.0          8         302.0       140.0  3449.0          10.5

   model year  origin                   car name
0          70       1  chevrolet chevelle malibu
1          70       1          buick skylark 320
2          70       1         plymouth satellite
3          70       1              amc rebel sst
4          70       1                ford torino
mpg             0
cylinders       0
displacement    0
horsepower      6
weight          0
acceleration    0
model year      0
origin          0
car name        0
dtype: int64
```

1.1 Data preparation:

1.1.1 NA Values

We can see that there are some missing values in the horsepower column, we can either drop these rows or replace them with the mean value. For now we will remove them, but if we find that horsepower is not a very important feature we might want to reintroduce them using the mean value. For now however we will remove them as they only account for ~1.5% of our dataset.

[4]:
```
df.dropna(inplace=True)
print(df.isna().sum())
```

```
mpg             0
cylinders       0
displacement    0
horsepower      0
weight          0
```

```
acceleration    0
model year      0
origin          0
car name        0
dtype: int64
```

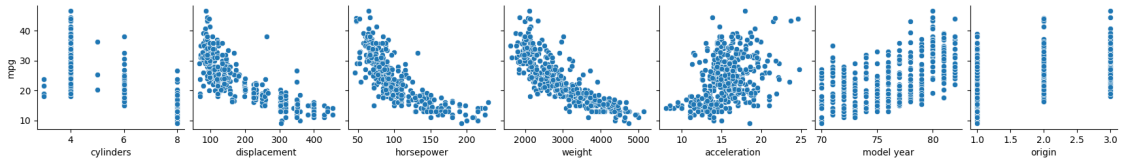
1.1.2 Feature engineering

Correlation We can plot the features of the dataset against the MPG to see if which features have a correlation with MPG.

We can see that displacement, horsepower, and weight all have a clear correlation with mpg. Acceleration however, is scattered and thus does not show much correlation to mpg. Looking at the discrete features we can see a that the different values are grouped at different ranges of mpg, this means they are correlated with mpg in some way, however we cannot use them in their current form, so we will have to encode them.

[5]:
```
# Plot features against mpg in pairplot
import seaborn as sns
sns.pairplot(df, x_vars=df.drop(["mpg", "car name"], axis=1, inplace=False).
 ↪columns, y_vars=["mpg"])
```

[5]:
```
<seaborn.axisgrid.PairGrid at 0x7f08a8693d30>
```



Car name feature The carname feature is a unique string, thus we cannot use it as a feature (at least not in its current form), since not only will the model only be trained on a single instance of each car name (since the car name is unique), but also the model will not be able to generalize to new car names. In addition to this encoding the car name would be very difficult, there are 301 unique car names, thus we would add 301 features to our model - then we are likely to suffer the consequences of the curse of dimensionality. We can however use it to create a new feature, the brand name is the first word in the car name, so we can use that as a feature. This will reduce the number of unique values to 37, which is still a lot, but we could use it as a feature.

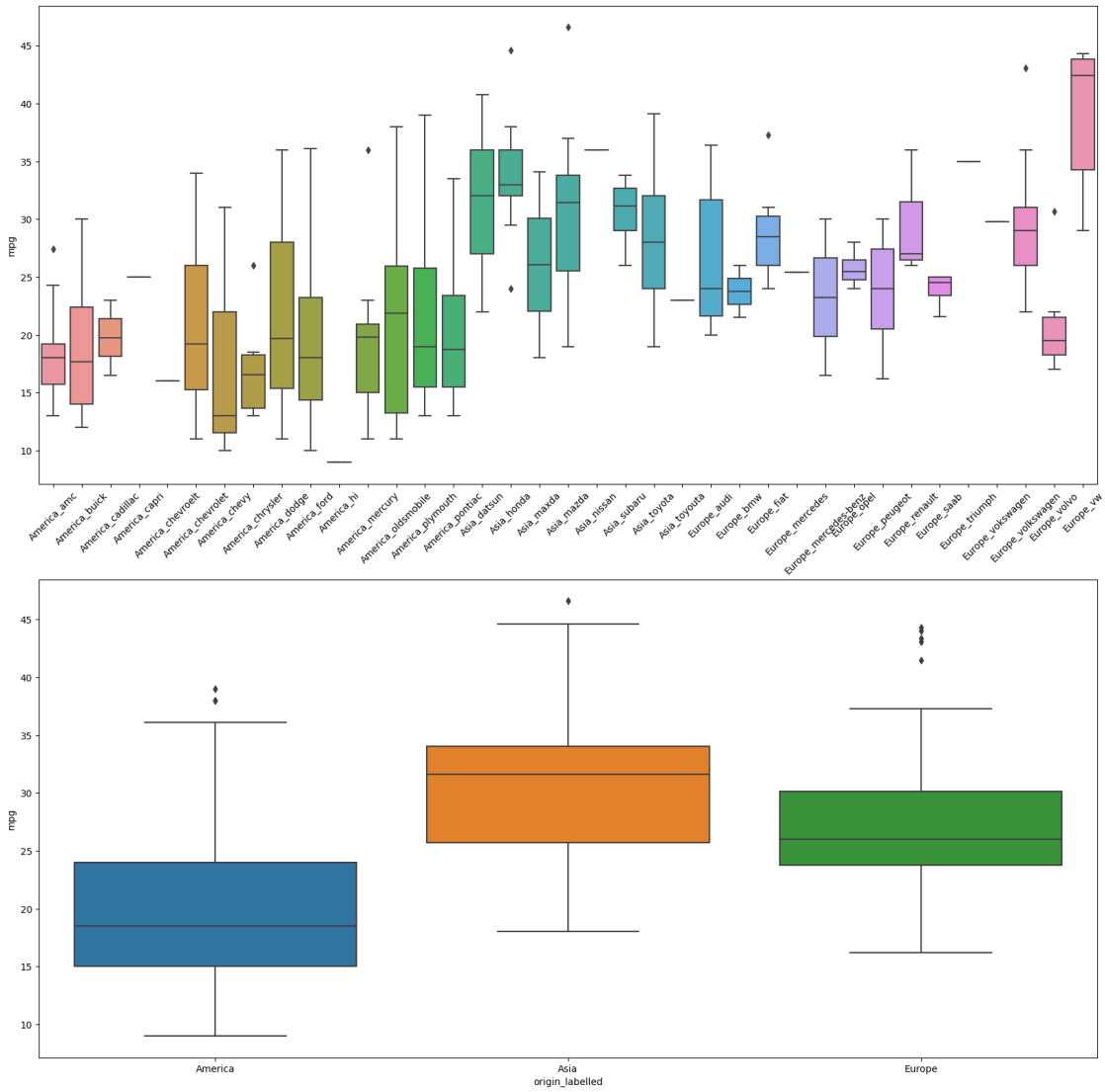
However, I think that we are better off just using the origin, as brands from a similar origin are likely to have similar mpg (MPG is very commonly marketed as a feature of the car, creating competition among brands to match / beat each other). So we will plot the brand_name and group them by origin to see if there is a correlation between brand, origin and mpg.

[6]:
```
import matplotlib.pyplot as plt
# show count of unique car names
print(f"Unique car names: {len(df['car name'].unique())}")
```

```
# car names are in the format "brand name model name"
# Show count of unique brand names
print(f"Unique brand names: {len(df['car name'].apply(lambda x: x.split()[0]).
 ↪unique())}")

# Add brands as a column prefixed with orgin_old
df["origin_labelled"] = df["origin"].map({1: "America", 2: "Europe", 3: "Asia"})
df["brand"] = df["origin_labelled"] + "_" + df["car name"].apply(lambda x: x.
 ↪split()[0])
# Plot correlation of brand to MPG, sorting brands by alphabetical order, with␣
 ↪a large figure size and the correlation of mpg to origin_labelled below
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(20, 20))
# Rotate x-axis labels 45 degrees
plt.setp(ax1.get_xticklabels(), rotation=45)
sns.boxplot(x="brand", y="mpg", data=df, order=sorted(df["brand"].unique()),␣
 ↪ax=ax1)
sns.boxplot(x="origin_labelled", y="mpg", data=df,␣
 ↪order=sorted(df["origin_labelled"].unique()), ax=ax2)
plt.show()
```

```
Unique car names: 301
Unique brand names: 37
```



We can observe that there is a relationship between the brand name and MPG, however it is not very clear, and we can see that the brands are quite fairly represented by their origin, so we will drop the brand name feature.

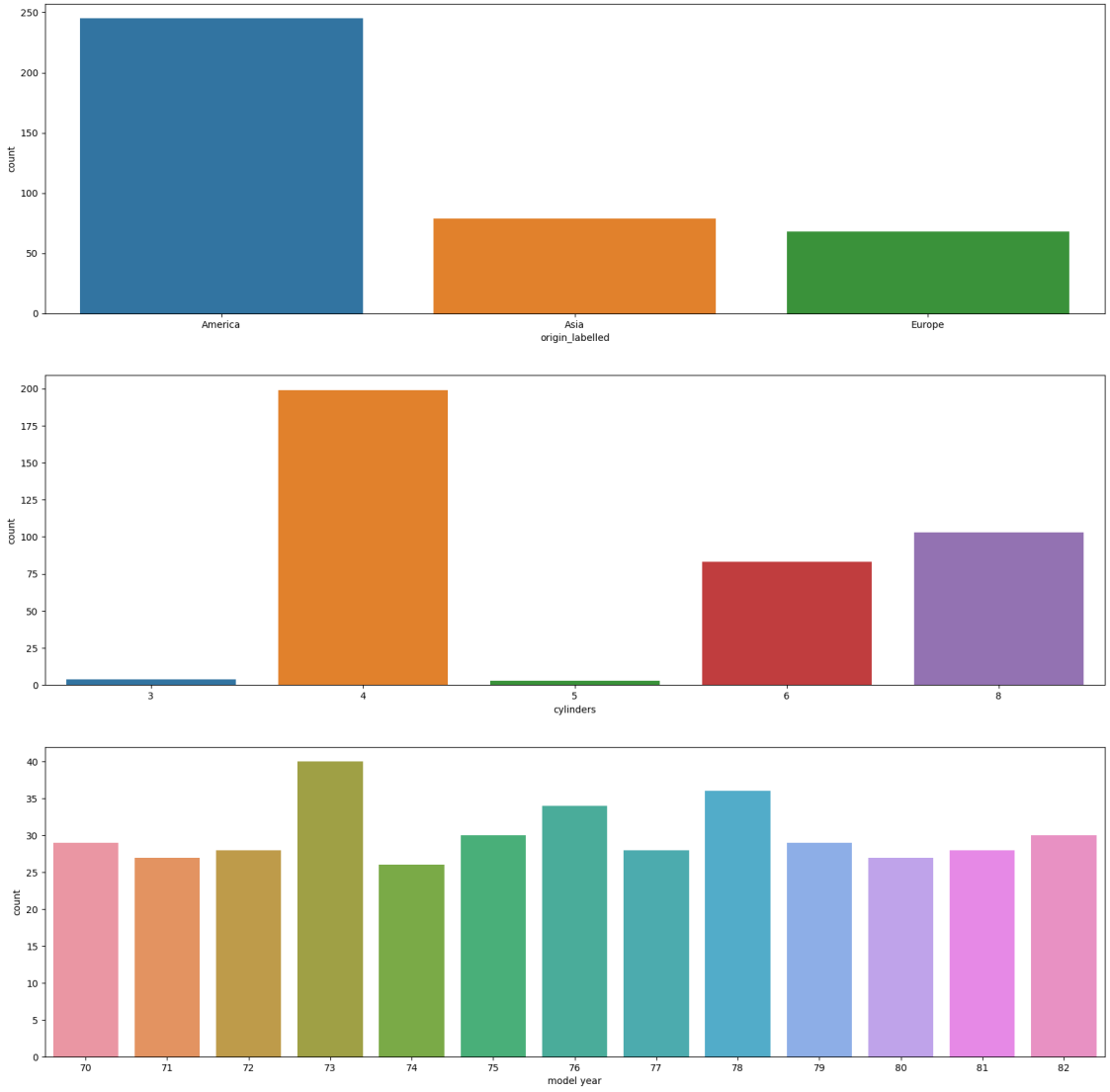
Another benefit to this is that the brand data is quite dirty, there are some brands that are misspelled ('toyouta') and some that are listed twice ('volkswagen' and 'vw'). This would be quite difficult to clean up, and we would have to add a lot of features to our model, so dropping it is the best option.

This also has the added benefit that our model will perform better on brands that aren't in the dataset.

Categorical features We can see that the cylinders, model year and origin features are categorical, we would like these to be distributed evenly (uniformly), but we can see that they are not. America makes up the majority of the cars in the dataset, this can cause the model to not generalise

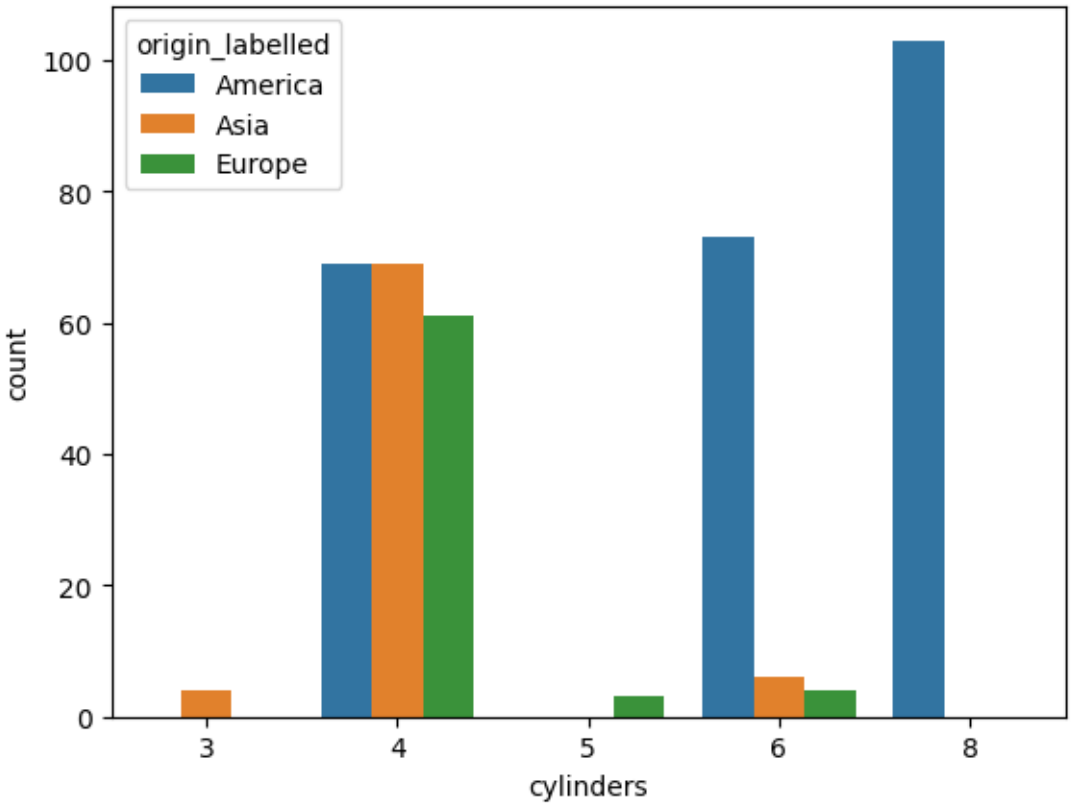
well to other origins. Looking at our cylinders we can see that we have very few cars with 3 or 5 cylinders, this is due to the fact most engines work best when balanced (half of the cylinders firing as the other half are on the compression stroke), 3 and 5 cylinder engines are generally only seen in rotary or inline engines. The fact that we do not have many values for these cylinder counts, and that they engines that do have these cylinder counts are of a different design, so we should probably drop these values. First we will look at the distribution of cylinders among the different origins.

[7]:
```
# Three sns.countplot, each one is plot of value counts of a column, the␣
 ↪columns are the categorical features (origin, cylinders, model year)
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(20, 20))
sns.countplot(x="origin_labelled", data=df, ax=ax1)
sns.countplot(x="cylinders", data=df, ax=ax2)
sns.countplot(x="model year", data=df, ax=ax3)
plt.show()
```



[8]:
```
# Create a count plot of cylinders, with the hue set to orgin_labelled
sns.countplot(x="cylinders", data=df, hue="origin_labelled")
```

[8]:
```
<AxesSubplot: xlabel='cylinders', ylabel='count'>
```

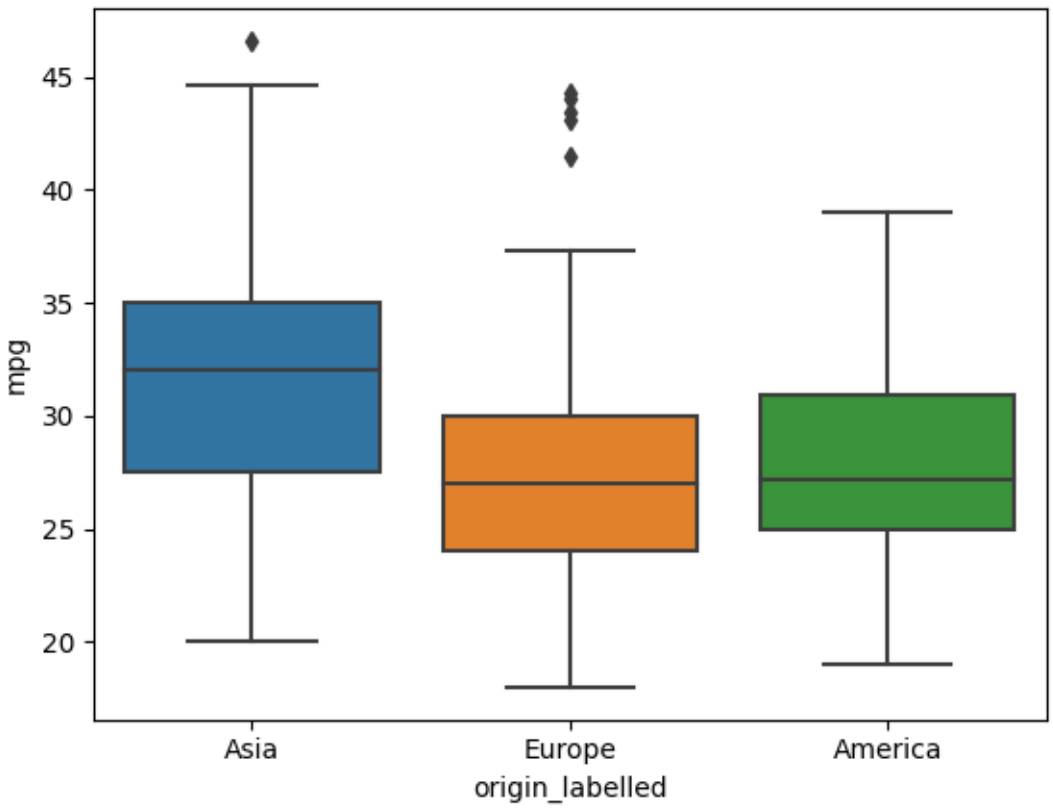


we can see that the cars with 3 cylinders are all Asian, and the cars with 5 cylinders are all European. We have many 8 cylinder cars, all American, since this is clearly a large proportion of the dataset we will leave them in, but we could drop the 3 and 5 cylinder cars. The consequence of leaving in the 8 cylinders is that the model could learn a bias, but we will have to accept this.

The large number of 8 cylinder engines from america is likely the reason for its low mpg compared to the other origins. If this is true then the origin column will not be an important feature, due to its high collinearity with the cylinders feature.

[9]:
```
# Plot the mpg of cars with 4 cylinders, with the hue set to origin_labelled
sns.boxplot(x="origin_labelled", y="mpg", data=df[df["cylinders"] == 4])
```

[9]:
```
<AxesSubplot: xlabel='origin_labelled', ylabel='mpg'>
```



Observing the above we can see that there is still a significant difference in the distribution of mpg between the different origins (for 4 cylinders only), so we will keep the origin feature.

Feature correlation We can plot the correlation matrix to see if there are any features that are highly correlated with each other, we are interested in features that are highly correlated to mpg specifically, however if two features are highly correlated to each other we can drop one of them (colinearity).

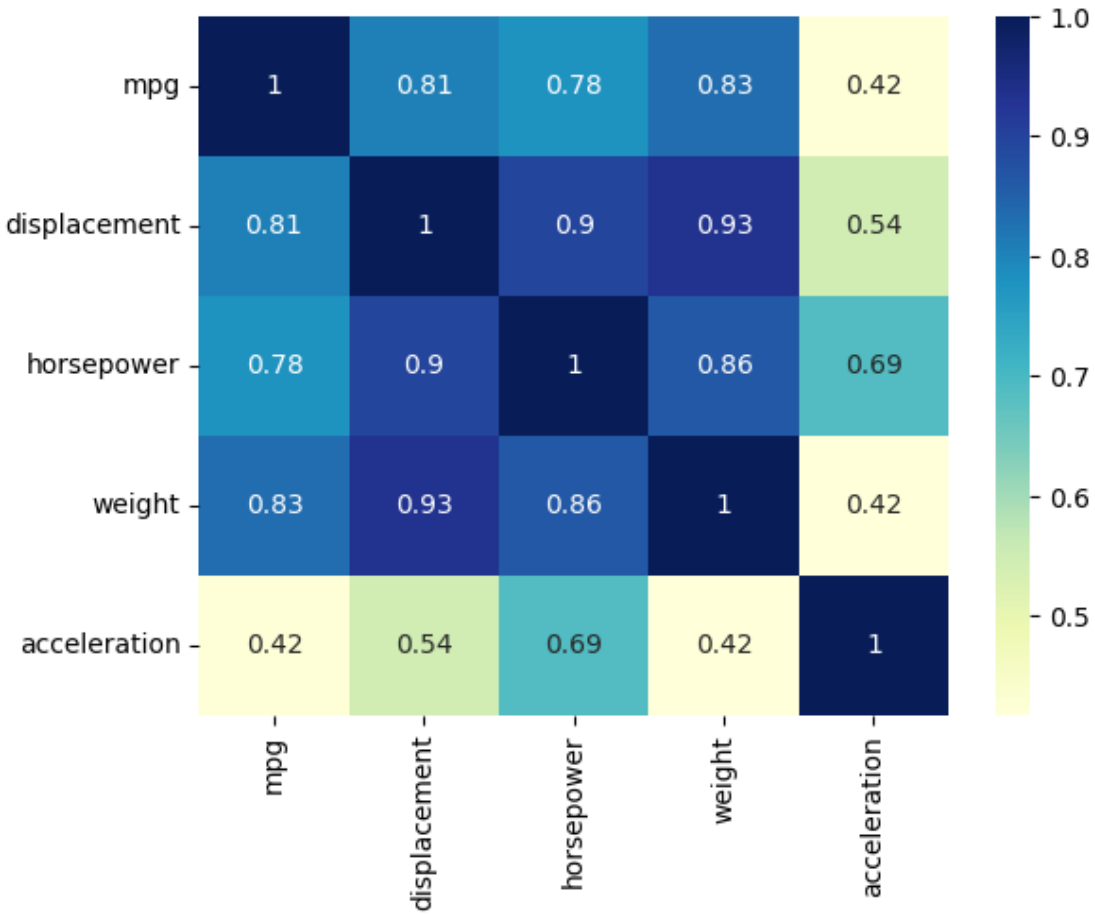
[10]:
```
continous = ["mpg", "displacement", "horsepower", "weight", "acceleration"]
categorical = ["cylinders", "origin_labelled", "model year"]

# heatmap of absolute correlation of continous features
sns.heatmap(df[continous].corr().abs(), annot=True, cmap="YlGnBu")

# Correlation of each feature to MPG
print(f"{'Feature':12} : correlation coefficent.")
correlations = dict()
for feature in continous:
    if not feature == "mpg":
        correlations[feature] = abs(df[feature].corr(df['mpg']))
```

```
# Sort correlations by value (highest to lowest) then print
for key, value in sorted(correlations.items(), key=lambda item: item[1],␣
 ↪reverse=True):
    print(f"{key:12} : {value:0.2f}")
```

```
Feature      : correlation coefficent.
weight       : 0.83
displacement : 0.81
horsepower   : 0.78
acceleration : 0.42
```



We can see that there is a weak correlation between mpg and acceleration, and a strong correlation between mpg and displacement, horsepower and weight. The correlation between weight, horsepower, and displacement are all very strong, so depending on our model we might want to drop one or two of these features.

Feature pruning Based on the above we will drop some features from the dataset: - Car name

Which means dropping the following columns: - Car name - origin (replaced with origin_labelled)

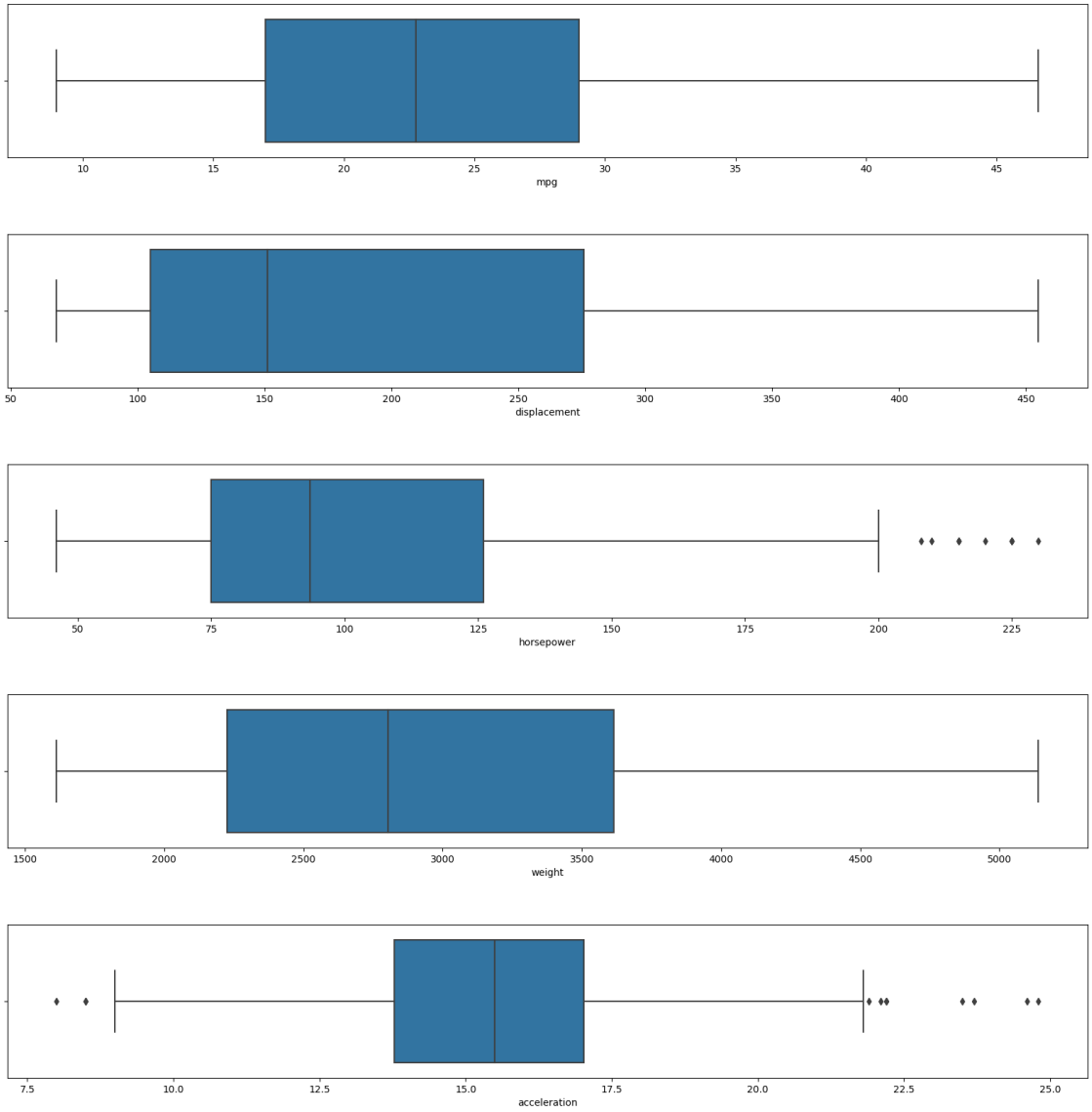
- brand

[11]:
```
df.drop("car name", axis=1, inplace=True)
df.drop("origin", axis=1, inplace=True)
df.drop("brand", axis=1, inplace=True)
```

Outliers We can look for outliers in the data using boxplots. Outliers in our training data can impact our model, so we should remove them. This is especially important if we choose a model that does not handle outliers well, such as linear regression.

[12]:
```
def plot_outliers(df, columns=continous): # Default to all continuous features
    fig, axes = plt.subplots(len(columns), 1, figsize=(20, 20))
    for i, column in enumerate(columns):
        sns.boxplot(x=column, data=df, ax=axes[i])
    plt.subplots_adjust(hspace=0.5)
    plt.show()

plot_outliers(df)
```



Here we see that there are some outliers in the horsepower and acceleration features, we will remove the horsepower outliers, we will drop all of these rows, as they could be a result of a mistake in data collection, and we don't want to introduce bias into our model.

We will remove the outliers using tukey fences, which uses the interquartile range to determine the outliers.

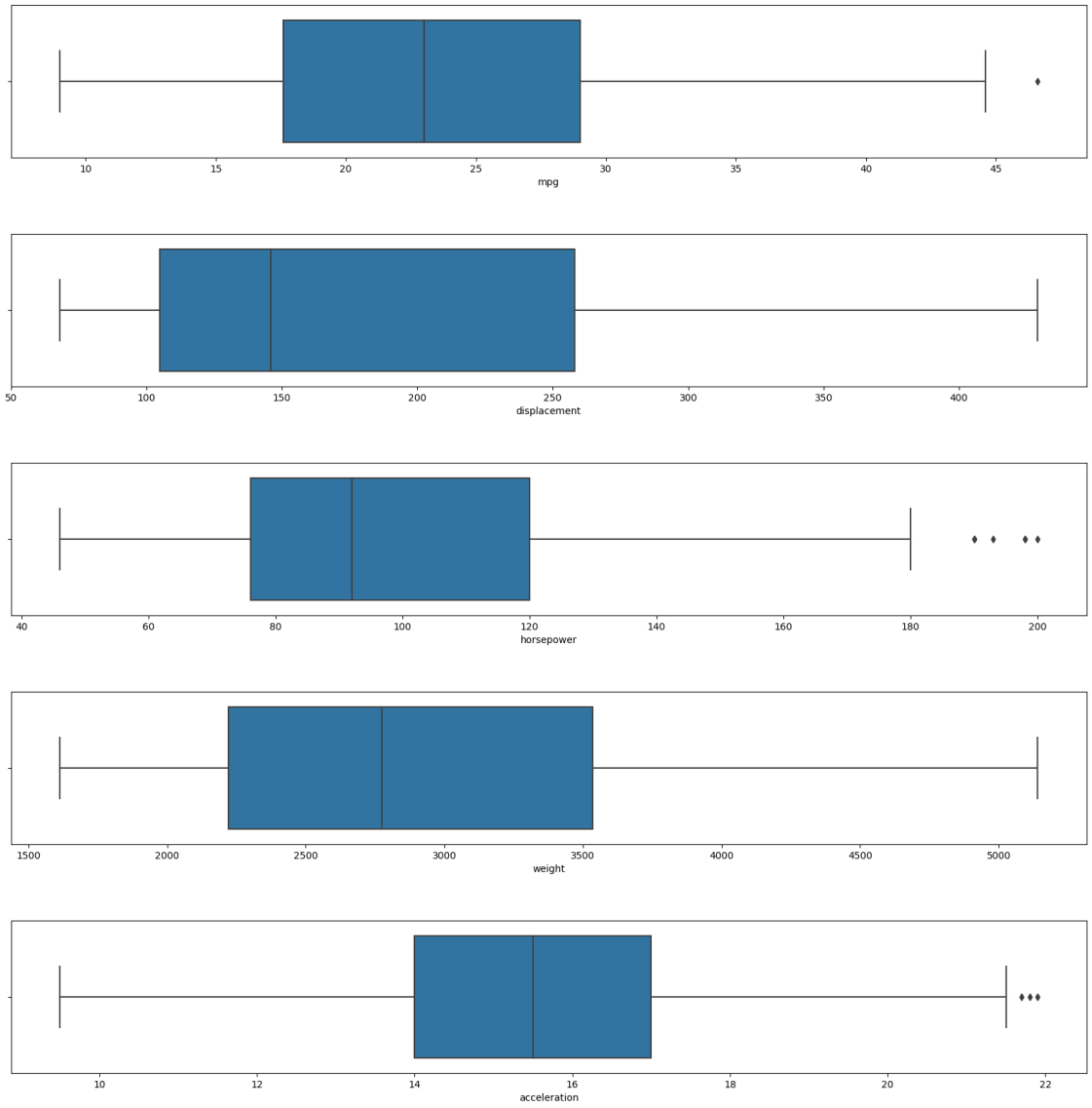
[13]:
```
# Function that removes outliers based on tukey fences, and prints how many␣
  ↪rows were removed for each column
def remove_outliers(df, column):
    q1 = df[column].quantile(0.25)
    q3 = df[column].quantile(0.75)
    iqr = q3 - q1
```

```
    fence_low = q1 - 1.5*iqr
    fence_high = q3 + 1.5*iqr
    print(f"Removed {len(df.loc[(df[column] < fence_low) | (df[column] >␣
 ↪fence_high)]):3} rows from {column}")
    return df.loc[(df[column] > fence_low) & (df[column] < fence_high)]

# Remove outliers from all columns except for Asia, Europe, America
for column in continous:
    df = remove_outliers(df, column)

# Plot outliers again to see if they have been removed
plot_outliers(df)
```

```
Removed   0 rows from mpg
Removed   0 rows from displacement
Removed  10 rows from horsepower
Removed   0 rows from weight
Removed   9 rows from acceleration
```



Feature preperation

Categorical features We have two options for categorical features, label encoding and one-hot encoding. Label encoding is a simple way to encode categorical features, it assigns each unique value in the feature a number, however this can cause the model to think that there is a relationship between the different values, when there is not. One-hot encoding is a more complex way to encode categorical features, it creates a new column for each unique value in the feature, and assigns a 1 or 0 to each row depending on if the value in the row matches the value in the column. This is a better way to encode categorical features, as it does not imply a relationship between the different values.

We will use one-hot encoding for the origin feature since there is no relationship between the different origins, and we will use label encoding for the model year (newer models are more likely to

be more efficient than older models) and cylinders because there is a ranking between the variables.

[14]:
```
# Encoding features
# Get dummies will create a new column for each unique non numerical value in␣
 ↪the column
df = pd.get_dummies(df, prefix="", prefix_sep="")

print(df.head()) # We can observe the presence of the new columns␣
 ↪"origin_labelled_Asia", "origin_labelled_America", "origin_labelled_Europe"
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0  3504.0          12.0
1  15.0          8         350.0       165.0  3693.0          11.5
2  18.0          8         318.0       150.0  3436.0          11.0
3  16.0          8         304.0       150.0  3433.0          12.0
4  17.0          8         302.0       140.0  3449.0          10.5

   model year  America  Asia  Europe
0          70        1     0       0
1          70        1     0       0
2          70        1     0       0
3          70        1     0       0
4          70        1     0       0
```

Splitting the data We will split the data into a training set and a test set. We will use the training set to train our model, and the test set to evaluate the performance of our model. We will use an 80/20 split.

[15]:
```
# Split to train and test datasets
train = df.sample(frac=0.8, random_state=0)
test = df.drop(train.index)

# Split into features and labels
train_features = train.copy()
train_labels = train_features.pop("mpg")
test_features = test.copy()
test_labels = test_features.pop("mpg")
```

Scaling We want to scale our features so that they are all between 0 and 1, this will improve the performance and stability of our model. It is important we scale after splitting the data, otherwise we will introduce bias into our model.

[16]:
```
# Scaling

# Function to normalise a column to a range of 0-1, returning the adjusted␣
 ↪dataframe and the min and max values
def normalise_column(df, column, cmin=None, cmax=None):
```

```
    cmin = cmin or df[column].min()
    cmax = cmax or df[column].max()
    df[column] = (df[column] - cmin) / (cmax - cmin)
    return df, cmin, cmax

def normalise_y(y, cmin=None, cmax=None):
    cmin = cmin or y.min()
    cmax = cmax or y.max()
    y = (y - cmin) / (cmax - cmin)
    return y, cmin, cmax

def denormalise_y(y, min, max):
    return (y * (max - min)) + min

def denormalise_column(df, column, cmin, cmax):
    df[column] = (df[column] * (cmax - cmin)) + cmin
    return df

# Normalise the training features
normalisation_bounds = dict()
for column in train_features.columns:
    train_features, cmin, cmax = normalise_column(train_features, column)
    test_features, _, _ = normalise_column(test_features, column, cmin, cmax)
    normalisation_bounds[column] = (cmin, cmax) # Store min and max values for␣
 ↪later (scaling back to original values)

test_labels, cmin, cmax = normalise_y(test_labels)
train_labels, _, _ = normalise_y(train_labels, cmin, cmax)
normalisation_bounds["mpg"] = (cmin, cmax)
```

We are scaling all the features, even the categorical ones so that our inputs are all between 0 and 1, the categories still exist in the scaled data.

1.2 Model selection

The two models I will analyse are XGBoost regression and a Multi-Layer Perceptron.

1.2.1 Multi-layer perceptron

A mutli-layer perceptron is a feed-forward neural network. MLPs can model complex, non-linear relationships between the features and the target, if any of these are present in the dataset, then an MLP will be able to model them. This is advtantagous in that we do not need to know what the relationship between the features and the target is, we can just feed the data into the model and it will learn the relationship, however this can be a disadvantage as it turns the model into a black box, we cannot see how the model is making its predictions, and we cannot explain the decisions of the model to others.

MLPs have a large number of hyper parameters, and take a long time to train, so it takes a long time to train and tune the model. In addition to this MLPs generally require more data preperation

than other models becuase it is sensitive to scale, distribution and quality of the input data.

1.2.2 XGBoost

XGBoost is a tree-based gradient, boosting model. XGBoost is a computationally efficient and accurate model, it is fast to train and tune for a multitude of reasons. XGBoost uses tree pruning to reduce the nodes on a tree, and thus reduce the complexity of the model and its chance of overfitting, It also uses regularisation techniques such as L1 (Lasso) & L2 (Ridge) to further prevent overfitting. XGBoost is also capable of parallel training, so it can create mutliple decision trees at the the same time, significantly reducing the training time.

Being tree-based it also means that XGBoost can be easily visualised, and the model can be explained to others.

1.2.3 Application to the problem

Both models are capable of modelling the relationship between the features and the target. However, with the relatively small dataset of ~400 instances, the underlying patterns in the data are likely to be simple, or have not been adequately captured. This means MLP is likely to overfit the data, or not generalise well to new data. XGBoost is less likely to overfit the data, and is more likely to generalise well to new data, so it is more likely to be a better model for this problem.

However to be certain we will train both models and compare their performance.

1.3 XGBoost

Implementation of the XGBoost model, training, tuning, and evaluation.

[38]:
```
from xgboost import XGBRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_absolute_error,␣
 ↪mean_absolute_percentage_error, mean_squared_error

paramter_grid = {
    "learning_rate": [0.05, 0.1, 0.13],
    "max_depth": [2, 3, 4], # Maximum tree depth, higher values can lead to␣
 ↪overfitting
    "n_estimators": [100, 250, 500], # Number of trees to train
    "subsample": [0.75], # Fraction of training data to use for each tree
    "colsample_bytree": [0.5], # Fraction of features to use for each tree
    "min_child_weight": [4, 5, 6], # Minimum sum of instance weight needed in a␣
 ↪child
}

xgb = XGBRegressor()

grid_search = GridSearchCV(xgb, paramter_grid, cv=5, n_jobs=-1, verbose=1)

grid_search.fit(train_features, train_labels)
```

```
print(f"Best parameters: {grid_search.best_params_}\n")
xgb = grid_search.best_estimator_

# Predict on test data
predictions = xgb.predict(test_features)

# Unscale predictions and labels
xgb_predictions = denormalise_y(predictions, *normalisation_bounds["mpg"])
unnormalised_test_labels = denormalise_y(test_labels,␣
 ↪*normalisation_bounds["mpg"])

print(f"Mean absolute error: {mean_absolute_error(unnormalised_test_labels,␣
 ↪xgb_predictions):0.2f} MPG")
print(f"Mean absolute percentage error:␣
 ↪{mean_absolute_percentage_error(unnormalised_test_labels, xgb_predictions):0.
 ↪3f}%")
print(f"Root mean squared error: {mean_squared_error(unnormalised_test_labels,␣
 ↪xgb_predictions, squared=False):0.2f} MPG")
```

```
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Best parameters: {'colsample_bytree': 0.5, 'learning_rate': 0.05, 'max_depth':
4, 'min_child_weight': 6, 'n_estimators': 100, 'subsample': 0.75}
Mean absolute error: 2.08 MPG
Mean absolute percentage error: 0.091%
Root mean squared error: 2.64 MPG
```

1.4 Multi-layer perceptron

There are so many hyper parameters for the MLP, that it is difficult to tune the model. In this instance I have used a genetic algorithm

1.4.1 Genetic algorithm

Genetic algorithms are a type of evolutionary algorithm, inspired by the process of natural selection. The hyperparameters are 'genes' (A full set of genes are called a 'Chromosome'), and the fitness of the model is its ability to predict the mpg of the cars in the test set. The algorithm works by creating a population of models, then it selects the fittest models and breeds them (creating new models with some mix of the 'parents' hyper parameters), there is also a chance of mutation, where a random hyper parameter is changed, this is done to prevent the algorithm from converging to early on a local minima. This process is repeated until either the fitness of the population has converged, or the maximum number of generations has been reached. This is most certainly not the most efficient way to tune the hyper parameters but it is better than trying all combinations (like in a grid search) and is still quite exhaustive. One problem I encountered was that my algorithm was encouraging the model to overfit the data,

[36]:
```
import os
import time
```

```
import pickle
import random
import numpy as np
import pandas as pd
from numba import jit # For speeding up the code

# Supress Tensorflow warnings
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

from sklearn.metrics import mean_absolute_error

import tensorflow as tf
from tensorflow import keras

# Tell linter to ignore this imports, it doesn't like them
from tensorflow.keras import layers # type: ignore
from tensorflow.keras import optimizers # type: ignore
from tensorflow.keras.callbacks import EarlyStopping # type: ignore

"""
Script constants:
    - Parameters for the algorithm
        - GENERATION_COUNT: Maximum number of generations to run for
        - MUTATION_RATE: The probability of a mutation occuring
        - ELITISM: The number of top performing individuals to keep for the␣
 ↪next generation
        - CYCLES: The number of times to run each model to get an average score
        - TRAIN_TEST_SPLIT: The percentage of the data to use for training

    - Parameters for the model
        - HIDDEN_LAYER_COUNTS: The number of hidden layers in the model
        - HIDDEN_LAYER_SIZES: The number of neurons in each hidden layer
        - EPOCHS: The number of epochs to train for
        - BATCH_SIZES: The batch size to use when training
        - LOSS: The loss function to use when training
        - OPTIMISERS: The optimisers to use when training
"""

# Parameters for the algorithm
GENERATION_COUNT = 1
MUTATION_RATE = 1/3
ELITISM = 10
CYCLES = 1
TRAIN_TEST_SPLIT = 0.8
VALIDATION_SPLIT = 0.2
```

```
INPUT_DIMENSIONS = len(train_features.columns)
Y = "mpg"

# Parameters for the model
HIDDEN_LAYER_COUNTS = [1, 2, 3]
HIDDEN_LAYER_SIZES = range(2, INPUT_DIMENSIONS) # The number of neurons in each␣
 ↪hidden layer
EPOCHS = 300
LEARNING_RATE = [0.001, 0.01, 0.05, 0.1]
BATCH_SIZES = [8, 16, 32, 48]
LOSS = "mean_squared_error"
OPTIMIZERS = [
    optimizers.Adam,
    optimizers.Adadelta,
    optimizers.Adagrad,
    optimizers.Adamax,
    optimizers.Ftrl,
    optimizers.Nadam,
    optimizers.RMSprop,
    optimizers.SGD
]
ACTIVATIONS = [
    "relu",
    "sigmoid",
    "tanh",
    "softmax",
    "softplus",
    "softsign",
    "selu",
    "elu",
    "exponential"
]
MAX_LAYER_COUNT = max(HIDDEN_LAYER_COUNTS)
LEARNING_RATE_IDX = 0
BATCH_SIZE_IDX = 1
OPTIMIZER_IDX = 2
LAYERS_IDX = 3
HIDDEN_LAYER_SIZE_IDX = 0
HIDDEN_LAYER_ACTIVATION_IDX = 1

"""
Paramter (CHROMOSOME) structure:
    [
        learning_rate,
        batch_size,
        optimizer,
```

```
        [
            [hidden_layer_size_1, activation_1],
            [hidden_layer_size_2, activation_2],
            [hidden_layer_size_3, activation_3]
        ]
    ]
"""

CALLBACKS = [EarlyStopping(monitor="val_loss", mode="min", verbose=0,␣
 ↪patience=25)]
```

[37]:
```
def anti_overfitting(chromosome):
    # Prevent overfitting by ensuring that models are a pyramid shape (i.e. the␣
 ↪number of neurons in each layer decreases or atleast stays the same)
    chromosome[LAYERS_IDX] = sorted(chromosome[LAYERS_IDX], key=lambda x:␣
 ↪x[HIDDEN_LAYER_SIZE_IDX], reverse=True)
    return chromosome

def init_population():
    population = list()
    for _ in range(ELITISM):
        layers = list()
        lcount = random.choice(HIDDEN_LAYER_COUNTS)
        for i in range(MAX_LAYER_COUNT):
            if i < lcount:
                layers.append([
                    random.choice(HIDDEN_LAYER_SIZES),
                    random.choice(ACTIVATIONS)
                ])
            else:
                layers.append([0, "none"])

        chromosome = [
            random.choice(LEARNING_RATE),
            random.choice(BATCH_SIZES),
            random.choice(OPTIMIZERS),
            layers
        ]
        population.append(anti_overfitting(chromosome))
    return population

MUTATIONS = {
    LEARNING_RATE_IDX: lambda: random.choice(LEARNING_RATE),
    BATCH_SIZE_IDX: lambda: random.choice(BATCH_SIZES),
    OPTIMIZER_IDX: lambda: random.choice(OPTIMIZERS),
    LAYERS_IDX: lambda: random.choice(HIDDEN_LAYER_COUNTS),
```

```
    LAYERS_IDX + 1 + HIDDEN_LAYER_SIZE_IDX: lambda: random.
 ↪choice(HIDDEN_LAYER_SIZES),
    LAYERS_IDX + 1 + HIDDEN_LAYER_ACTIVATION_IDX: lambda: random.
 ↪choice(ACTIVATIONS)
}

PARAM_COUNT = LAYERS_IDX + MAX_LAYER_COUNT * 2

def mutate(chromosome):
    mutate_idxs = random.sample(range(PARAM_COUNT), int(PARAM_COUNT / 2))
    for idx in mutate_idxs:
        if idx < LAYERS_IDX:
            chromosome[idx] = MUTATIONS[idx]()
        else:
            layer_idx = (idx - LAYERS_IDX) // 2 # 0 - 5 hopefully
            target_idx = (idx - LAYERS_IDX) % 2 # 0 or 1
            chromosome[LAYERS_IDX][layer_idx][target_idx] =␣
 ↪MUTATIONS[LAYERS_IDX + 1 + idx]()
            non_target = int(not target_idx)
            if chromosome[LAYERS_IDX][layer_idx][non_target] in [0, "none"]:
                chromosome[LAYERS_IDX][layer_idx][non_target] =␣
 ↪MUTATIONS[LAYERS_IDX + 1 + non_target]()
    return anti_overfitting(chromosome)

def possible_mutation(chromosome):
    return mutate(chromosome) if random.random() < MUTATION_RATE else chromosome

def crossover_index(idx, p1, p2, c1, c2):
    if idx < LAYERS_IDX:
        c1[idx] = p1[idx]
        c2[idx] = p2[idx]
        return c1, c2

    layer_idx = (idx - LAYERS_IDX) // 2 # 0 - 2
    target_idx = (idx - LAYERS_IDX) % 2 # 0 or 1

    c1[LAYERS_IDX][layer_idx][target_idx] =␣
 ↪p1[LAYERS_IDX][layer_idx][target_idx]
    c2[LAYERS_IDX][layer_idx][target_idx] =␣
 ↪p2[LAYERS_IDX][layer_idx][target_idx]
    non_target_idx = int(not target_idx)
    for c in [c1, c2]:
        if c[LAYERS_IDX][layer_idx][non_target_idx] in [0, "none"]:
            c[LAYERS_IDX][layer_idx][non_target_idx] = MUTATIONS[LAYERS_IDX + 1␣
 ↪+ non_target_idx]()
    return anti_overfitting(c1), anti_overfitting(c2)
```

```
def crossover(chromosome1, chromosome2):
    crossover_idx = random.sample(range(PARAM_COUNT), int(PARAM_COUNT/2))
    # Init children with empty values
    c1 = c2 = [None] * LAYERS_IDX + [[[0, "none"] for _ in␣
 ↪range(MAX_LAYER_COUNT)]]
    for idx in range(PARAM_COUNT):
        if idx in crossover_idx:
            pa, pb = chromosome1, chromosome2
        else:
            pb, pa = chromosome1, chromosome2
        c1, c2 = crossover_index(idx, pa, pb, c1, c2)
    return c1, c2

def get_model(chromosome, X_train, y_train):
    model = keras.Sequential()
    # ensure no non-empty layers without activation
    for layer in chromosome[LAYERS_IDX]:
        if layer[HIDDEN_LAYER_SIZE_IDX] > 0 and␣
 ↪layer[HIDDEN_LAYER_ACTIVATION_IDX] == "none":
            layer[HIDDEN_LAYER_ACTIVATION_IDX] = MUTATIONS[LAYERS_IDX + 1 +␣
 ↪HIDDEN_LAYER_ACTIVATION_IDX]()
    model.add(layers.Dense(chromosome[LAYERS_IDX][0][HIDDEN_LAYER_SIZE_IDX],␣
 ↪activation=chromosome[LAYERS_IDX][0][HIDDEN_LAYER_ACTIVATION_IDX],␣
 ↪input_shape=(X_train.shape[1],)))
    for i in range(1, MAX_LAYER_COUNT):
        if chromosome[LAYERS_IDX][i][HIDDEN_LAYER_SIZE_IDX] > 0:
            model.add(layers.
 ↪Dense(chromosome[LAYERS_IDX][i][HIDDEN_LAYER_SIZE_IDX],␣
 ↪activation=chromosome[LAYERS_IDX][i][HIDDEN_LAYER_ACTIVATION_IDX]))
    model.add(layers.Dense(1))
    model.compile(loss=LOSS,␣
 ↪optimizer=chromosome[OPTIMIZER_IDX](chromosome[LEARNING_RATE_IDX]),␣
 ↪metrics=["mae", "mse"])
    model.fit(X_train, y_train, epochs=EPOCHS,␣
 ↪batch_size=chromosome[BATCH_SIZE_IDX], verbose=0, callbacks=CALLBACKS,␣
 ↪validation_split=VALIDATION_SPLIT)
    return model

def unnormalise_y(y):
    min, max = normalisation_bounds[Y]
    return (y * (max - min) + min)

def pretty_print(chromosome):
    print(f"Learning rate: {chromosome[LEARNING_RATE_IDX]}")
    print(f"Batch size: {chromosome[BATCH_SIZE_IDX]}")
```

```
    print(f"Optimizer: {chromosome[OPTIMIZER_IDX]}")
    print(f"L1: {chromosome[LAYERS_IDX][0][HIDDEN_LAYER_SIZE_IDX]}␣
 ↪{chromosome[LAYERS_IDX][0][HIDDEN_LAYER_ACTIVATION_IDX]}")
    print(f"L2: {chromosome[LAYERS_IDX][1][HIDDEN_LAYER_SIZE_IDX]}␣
 ↪{chromosome[LAYERS_IDX][1][HIDDEN_LAYER_ACTIVATION_IDX]}")
    print(f"L3: {chromosome[LAYERS_IDX][2][HIDDEN_LAYER_SIZE_IDX]}␣
 ↪{chromosome[LAYERS_IDX][2][HIDDEN_LAYER_ACTIVATION_IDX]}")
    print("\n")

unylabel = unnormalise_y(test_labels)

def evaluate_chromosome(chromosome):
    pretty_print(chromosome)
    scores = []
    for _c in range(CYCLES):
        model = get_model(chromosome, train_features, train_labels)
        epochs = len(model.history.history["loss"])
        yhat = model.predict(test_features).flatten()
        if np.isnan(yhat).any():
            print("NaN in prediction")
            mae = None
        else:
            unyhat = unnormalise_y(yhat)
            mae = mean_absolute_error(unylabel, unyhat)
        print(f"Cycle {_c} -> MAE: {mae} (epochs: {epochs})")
        scores.append(mae)
    score = np.mean(scores)
    print(f"Chromosome score: {score} | finshed @ {time.ctime()}")
    return score

"""
Evaluated chromosome structure:
[
    fitness,
    chromosome...
]
"""

def evaluate_population(population):
    evaluated_population = []
    for e_chromosome in population:
        score, chromosome = e_chromosome[0], e_chromosome[1]
        if score is None:
            score = evaluate_chromosome(chromosome)
        evaluated_population.append([score, chromosome])
    return evaluated_population
```

```
def crossover_elites(evaluated_population):
    # We take ELITISM number of chromosomes, sorting by fitness (10000 is for␣
 ↪None values)
    elites = sorted(evaluated_population, key=lambda x: x[0] or 10000)[:ELITISM]
    minimum, maximum = elites[0][0], elites[-1][0]
    print(f"Elites | Minimum: {minimum} <-> Maximum: {maximum}")
    random.shuffle(elites)
    population = elites.copy() # We keep the elites in the population
    for i in range(0, ELITISM, 2):
        c1, c2 = crossover(elites[i][1], elites[i+1][1])
        population.append([None, c1])
        population.append([None, c2])
    return population

spike = [
    [
        0.001,
        16,
        optimizers.Nadam,
        [
            [4, "softsign"],
            [4, "softmax"],
            [0, "none"]
        ]
    ],
    [
        0.1,
        8,
        optimizers.SGD,
        [
            [8, "sigmoid"],
            [7, "elu"],
            [6, "exponential"]
        ]
    ],
    [
        0.001,
        16,
        optimizers.Nadam,
        [
            [7, "sigmoid"],
            [7, "sigmoid"],
            [3, "exponential"]
        ]
    ]
]
```

```
inital_pop = spike + init_population()[len(spike):]

def get_best_chromosome(inital_pop):
    population = [[None, x] for x in inital_pop]
    current_generation = 0
    while current_generation <= GENERATION_COUNT:
        print(f"\nGeneration {current_generation} started @ {time.ctime()}\n")
        elites = evaluate_population(population)
        pickle.dump(elites, open(f"GA/Generation_{current_generation}-elites.
 ↪pkl", "wb"))
        population = crossover_elites(elites)
        current_generation += 1
    return sorted(elites, key=lambda x: x[0] or 10000)[0]

fittest_model = [2.10772694905599, [0.05, 32, optimizers.Nadam, [[2, "selu"],␣
 ↪[2, "selu"], [2, "tanh"]]]]
fittest_model = get_best_chromosome(inital_pop) # Takes ~ 1 hour
print(f"\n\n\nBest model: (score: {fittest_model[0]:2f})")
pretty_print(fittest_model[1])
```

```
Generation 0 started @ Wed Feb  1 16:30:31 2023

Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 4 softsign
L2: 4 softmax
L3: 0 none


3/3 [==============================] - 0s 3ms/step
Cycle 0 -> MAE: 2.2112920761108397 (epochs: 300)
Chromosome score: 2.2112920761108397 | finshed @ Wed Feb  1 16:31:20 2023
Learning rate: 0.1
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 8 sigmoid
L2: 7 elu
L3: 6 exponential


3/3 [==============================] - 0s 5ms/step
Cycle 0 -> MAE: 2.2676317545572915 (epochs: 93)
Chromosome score: 2.2676317545572915 | finshed @ Wed Feb  1 16:31:44 2023
Learning rate: 0.05
Batch size: 48
```

```
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
L1: 3 sigmoid
L2: 0 none
L3: 0 none


3/3 [==============================] - 0s 3ms/step
Cycle 0 -> MAE: 2.168891184488932 (epochs: 300)
Chromosome score: 2.168891184488932 | finshed @ Wed Feb  1 16:32:09 2023
Learning rate: 0.01
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adadelta.Adadelta'>
L1: 7 exponential
L2: 6 selu
L3: 0 none


3/3 [==============================] - 0s 5ms/step
Cycle 0 -> MAE: 4.040552045186361 (epochs: 300)
Chromosome score: 4.040552045186361 | finshed @ Wed Feb  1 16:32:55 2023
Learning rate: 0.05
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 4 elu
L2: 2 tanh
L3: 0 none


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.34808344523112 (epochs: 55)
Chromosome score: 2.34808344523112 | finshed @ Wed Feb  1 16:33:12 2023
Learning rate: 0.1
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 tanh
L2: 7 softmax
L3: 5 softplus


3/3 [==============================] - 0s 6ms/step
Cycle 0 -> MAE: 2.1801394220987955 (epochs: 157)
Chromosome score: 2.1801394220987955 | finshed @ Wed Feb  1 16:33:54 2023
Learning rate: 0.01
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.ftrl.Ftrl'>
L1: 8 selu
L2: 8 sigmoid
L3: 0 none
```

```
3/3 [==============================] - 0s 5ms/step
Cycle 0 -> MAE: 6.319318008422852 (epochs: 37)
Chromosome score: 6.319318008422852 | finshed @ Wed Feb  1 16:34:02 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 2 tanh
L2: 0 none
L3: 0 none


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.3456044565836587 (epochs: 300)
Chromosome score: 2.3456044565836587 | finshed @ Wed Feb  1 16:34:49 2023
Learning rate: 0.1
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adadelta.Adadelta'>
L1: 6 selu
L2: 0 none
L3: 0 none


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.495306945800781 (epochs: 300)
Chromosome score: 2.495306945800781 | finshed @ Wed Feb  1 16:35:33 2023
Learning rate: 0.1
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adadelta.Adadelta'>
L1: 6 elu
L2: 0 none
L3: 0 none


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.7280927810668945 (epochs: 300)
Chromosome score: 2.7280927810668945 | finshed @ Wed Feb  1 16:35:52 2023
Elites | Minimum: 2.168891184488932 <-> Maximum: 6.319318008422852

Generation 1 started @ Wed Feb  1 16:35:52 2023

Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softsign
L2: 5 none
L3: 2 none
```

```
3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 6.360866038004557 (epochs: 300)
Chromosome score: 6.360866038004557 | finshed @ Wed Feb  1 16:36:19 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softsign
L2: 5 softsign
L3: 2 relu


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 3.53099085744222 (epochs: 300)
Chromosome score: 3.53099085744222 | finshed @ Wed Feb  1 16:36:44 2023
Learning rate: 0.01
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adadelta.Adadelta'>
L1: 8 selu
L2: 5 none
L3: 2 none


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 4.782213038126628 (epochs: 300)
Chromosome score: 4.782213038126628 | finshed @ Wed Feb  1 16:37:01 2023
Learning rate: 0.01
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adadelta.Adadelta'>
L1: 8 selu
L2: 5 tanh
L3: 2 softsign


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 3.799497075398763 (epochs: 300)
Chromosome score: 3.799497075398763 | finshed @ Wed Feb  1 16:37:17 2023
Learning rate: 0.1
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adadelta.Adadelta'>
L1: 6 exponential
L2: 6 selu
L3: 4 none


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 3.117684145609538 (epochs: 300)
```

```
Chromosome score: 3.117684145609538 | finshed @ Wed Feb  1 16:37:34 2023
Learning rate: 0.1
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adadelta.Adadelta'>
L1: 6 exponential
L2: 6 selu
L3: 4 softmax


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 3.3676149215698246 (epochs: 300)
Chromosome score: 3.3676149215698246 | finshed @ Wed Feb  1 16:37:51 2023
Learning rate: 0.1
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 8 elu
L2: 7 tanh
L3: 6 exponential


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.6749965159098306 (epochs: 43)
Chromosome score: 2.6749965159098306 | finshed @ Wed Feb  1 16:37:57 2023
Learning rate: 0.1
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 8 elu
L2: 7 tanh
L3: 6 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.4463042068481444 (epochs: 42)
Chromosome score: 2.4463042068481444 | finshed @ Wed Feb  1 16:38:02 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
L1: 7 none
L2: 6 none
L3: 3 softsign


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2380360972086586 (epochs: 300)
Chromosome score: 2.2380360972086586 | finshed @ Wed Feb  1 16:38:19 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
```

```
L1: 7 sigmoid
L2: 6 sigmoid
L3: 3 softsign


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.330777379353841 (epochs: 300)
Chromosome score: 2.330777379353841 | finshed @ Wed Feb  1 16:38:38 2023
Elites | Minimum: 2.168891184488932 <-> Maximum: 2.495306945800781

Generation 2 started @ Wed Feb  1 16:38:38 2023

Learning rate: 0.001
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
L1: 4 sigmoid
L2: 4 softmax
L3: 2 none


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 4.843574620564779 (epochs: 300)
Chromosome score: 4.843574620564779 | finshed @ Wed Feb  1 16:38:49 2023
Learning rate: 0.001
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
L1: 4 sigmoid
L2: 4 softmax
L3: 2 softmax


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 6.309478861490885 (epochs: 171)
Chromosome score: 6.309478861490885 | finshed @ Wed Feb  1 16:38:56 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 sigmoid
L2: 7 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.4586310119628902 (epochs: 300)
Chromosome score: 2.4586310119628902 | finshed @ Wed Feb  1 16:39:15 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
```

```
L1: 7 sigmoid
L2: 7 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.1439667409261065 (epochs: 300)
Chromosome score: 2.1439667409261065 | finshed @ Wed Feb  1 16:39:33 2023
Learning rate: 0.1
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 tanh
L2: 7 elu
L3: 6 softplus


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.2640676829020183 (epochs: 124)
Chromosome score: 2.2640676829020183 | finshed @ Wed Feb  1 16:39:44 2023
Learning rate: 0.1
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 tanh
L2: 7 elu
L3: 6 softplus


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.287487650553385 (epochs: 118)
Chromosome score: 2.287487650553385 | finshed @ Wed Feb  1 16:39:58 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 8 softsign
L2: 7 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.1878557128906246 (epochs: 300)
Chromosome score: 2.1878557128906246 | finshed @ Wed Feb  1 16:40:18 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 8 softsign
L2: 7 sigmoid
L3: 6 sigmoid
```

```
3/3 [==============================] - 0s 4ms/step
Cycle 0 -> MAE: 2.1684851989746092 (epochs: 210)
Chromosome score: 2.1684851989746092 | finshed @ Wed Feb  1 16:40:33 2023
Learning rate: 0.05
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 6 selu
L2: 6 none
L3: 2 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.4927768376668293 (epochs: 43)
Chromosome score: 2.4927768376668293 | finshed @ Wed Feb  1 16:40:38 2023
Learning rate: 0.05
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 6 selu
L2: 6 elu
L3: 2 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.282387730916341 (epochs: 35)
Chromosome score: 2.282387730916341 | finshed @ Wed Feb  1 16:40:43 2023
Elites | Minimum: 2.1439667409261065 <-> Maximum: 2.282387730916341

Generation 3 started @ Wed Feb  1 16:40:43 2023

Learning rate: 0.001
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 8 none
L2: 7 exponential
L3: 3 none


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2323735275268555 (epochs: 300)
Chromosome score: 2.2323735275268555 | finshed @ Wed Feb  1 16:40:57 2023
Learning rate: 0.001
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 8 softmax
L2: 7 exponential
L3: 3 sigmoid
```

```
3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 5.693855509440104 (epochs: 300)
Chromosome score: 5.693855509440104 | finshed @ Wed Feb  1 16:41:10 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 6 sigmoid
L3: 3 none


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.2353574396769202 (epochs: 234)
Chromosome score: 2.2353574396769202 | finshed @ Wed Feb  1 16:41:30 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 6 sigmoid
L3: 3 softsign


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.2001007486979165 (epochs: 255)
Chromosome score: 2.2001007486979165 | finshed @ Wed Feb  1 16:41:53 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 softplus


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 6.5583855946858725 (epochs: 300)
Chromosome score: 6.5583855946858725 | finshed @ Wed Feb  1 16:42:14 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 softplus


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 6.541834869384766 (epochs: 300)
Chromosome score: 6.541834869384766 | finshed @ Wed Feb  1 16:42:34 2023
```

```
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 sigmoid
L2: 7 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 6.1658630879720056 (epochs: 300)
Chromosome score: 6.1658630879720056 | finshed @ Wed Feb  1 16:42:52 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 sigmoid
L2: 7 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 4.3697117309570315 (epochs: 300)
Chromosome score: 4.3697117309570315 | finshed @ Wed Feb  1 16:43:10 2023
Learning rate: 0.1
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 6 selu
L2: 6 elu
L3: 6 tanh


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.4781957321166987 (epochs: 67)
Chromosome score: 2.4781957321166987 | finshed @ Wed Feb  1 16:43:17 2023
Learning rate: 0.1
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 6 selu
L2: 6 elu
L3: 6 tanh


3/3 [==============================] - 0s 4ms/step
Cycle 0 -> MAE: 2.2343948974609376 (epochs: 179)
Chromosome score: 2.2343948974609376 | finshed @ Wed Feb  1 16:43:39 2023
Elites | Minimum: 2.1439667409261065 <-> Maximum: 2.2353574396769202

Generation 4 started @ Wed Feb  1 16:43:39 2023
```

```
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 5 sigmoid
L3: 4 none


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.1708086242675777 (epochs: 152)
Chromosome score: 2.1708086242675777 | finshed @ Wed Feb  1 16:43:58 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 5 sigmoid
L3: 4 relu


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.1843406524658207 (epochs: 166)
Chromosome score: 2.1843406524658207 | finshed @ Wed Feb  1 16:44:20 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 6 selu
L2: 6 sigmoid
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.1327101694742834 (epochs: 300)
Chromosome score: 2.1327101694742834 | finshed @ Wed Feb  1 16:44:45 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 6 selu
L2: 6 sigmoid
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2939813537597655 (epochs: 300)
Chromosome score: 2.2939813537597655 | finshed @ Wed Feb  1 16:45:06 2023
Learning rate: 0.001
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
L1: 8 sigmoid
```

```
L2: 5 exponential
L3: 3 sigmoid


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.8105642420450847 (epochs: 300)
Chromosome score: 2.8105642420450847 | finshed @ Wed Feb  1 16:45:20 2023
Learning rate: 0.001
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
L1: 8 sigmoid
L2: 5 exponential
L3: 3 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 6.284897994995117 (epochs: 300)
Chromosome score: 6.284897994995117 | finshed @ Wed Feb  1 16:45:32 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2305252075195314 (epochs: 300)
Chromosome score: 2.2305252075195314 | finshed @ Wed Feb  1 16:45:50 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.1615822906494135 (epochs: 300)
Chromosome score: 2.1615822906494135 | finshed @ Wed Feb  1 16:46:07 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 sigmoid
L2: 7 sigmoid
L3: 3 exponential
```

```
3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.300753189086914 (epochs: 300)
Chromosome score: 2.300753189086914 | finshed @ Wed Feb  1 16:46:25 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 sigmoid
L2: 7 sigmoid
L3: 3 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.122076235453288 (epochs: 300)
Chromosome score: 2.122076235453288 | finshed @ Wed Feb  1 16:46:44 2023
Elites | Minimum: 2.122076235453288 <-> Maximum: 2.1878557128906246

Generation 5 started @ Wed Feb  1 16:46:44 2023

Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 sigmoid
L2: 7 softmax
L3: 5 softplus


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 6.3711297353108725 (epochs: 122)
Chromosome score: 6.3711297353108725 | finshed @ Wed Feb  1 16:46:55 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 sigmoid
L2: 7 softmax
L3: 5 softplus


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 6.3509347534179685 (epochs: 40)
Chromosome score: 6.3509347534179685 | finshed @ Wed Feb  1 16:46:59 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 6 softsign
L2: 6 softplus
L3: 5 tanh
```

```
3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.1706433868408204 (epochs: 186)
Chromosome score: 2.1706433868408204 | finshed @ Wed Feb  1 16:47:21 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 6 softsign
L2: 6 softplus
L3: 5 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.3526327743530273 (epochs: 189)
Chromosome score: 2.3526327743530273 | finshed @ Wed Feb  1 16:47:42 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 4 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.275214655558268 (epochs: 147)
Chromosome score: 2.275214655558268 | finshed @ Wed Feb  1 16:47:58 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 4 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2959266179402666 (epochs: 226)
Chromosome score: 2.2959266179402666 | finshed @ Wed Feb  1 16:48:22 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 8 softsign
L2: 7 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.206781560262044 (epochs: 263)
Chromosome score: 2.206781560262044 | finshed @ Wed Feb  1 16:48:39 2023
Learning rate: 0.001
```

```
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 8 softsign
L2: 7 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.213000559488932 (epochs: 272)
Chromosome score: 2.213000559488932 | finshed @ Wed Feb  1 16:48:55 2023
Learning rate: 0.05
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 sigmoid
L2: 6 none
L3: 6 none


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.854808525085449 (epochs: 70)
Chromosome score: 2.854808525085449 | finshed @ Wed Feb  1 16:48:59 2023
Learning rate: 0.05
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 sigmoid
L2: 6 sigmoid
L3: 6 selu


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.5525127334594724 (epochs: 80)
Chromosome score: 2.5525127334594724 | finshed @ Wed Feb  1 16:49:03 2023
Elites | Minimum: 2.122076235453288 <-> Maximum: 2.1843406524658207

Generation 6 started @ Wed Feb  1 16:49:03 2023

Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 6 none
L2: 5 sigmoid
L3: 3 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.280247047424316 (epochs: 300)
Chromosome score: 2.280247047424316 | finshed @ Wed Feb  1 16:49:20 2023
Learning rate: 0.001
```

```
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 6 softsign
L2: 5 sigmoid
L3: 3 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.253310015360514 (epochs: 300)
Chromosome score: 2.253310015360514 | finshed @ Wed Feb  1 16:49:37 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 6 softsign
L2: 5 sigmoid
L3: 4 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.3446258773803708 (epochs: 119)
Chromosome score: 2.3446258773803708 | finshed @ Wed Feb  1 16:49:50 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 6 softsign
L2: 5 sigmoid
L3: 4 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.199229611714681 (epochs: 197)
Chromosome score: 2.199229611714681 | finshed @ Wed Feb  1 16:50:09 2023
Learning rate: 0.1
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softsign
L2: 7 softmax
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.294916119893392 (epochs: 300)
Chromosome score: 2.294916119893392 | finshed @ Wed Feb  1 16:50:26 2023
Learning rate: 0.1
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softsign
L2: 7 softmax
```

```
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.133342267354329 (epochs: 300)
Chromosome score: 2.133342267354329 | finshed @ Wed Feb  1 16:50:43 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 6 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.224811454772949 (epochs: 111)
Chromosome score: 2.224811454772949 | finshed @ Wed Feb  1 16:50:55 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 6 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.218035120646159 (epochs: 215)
Chromosome score: 2.218035120646159 | finshed @ Wed Feb  1 16:51:16 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 3 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.4421271107991536 (epochs: 193)
Chromosome score: 2.4421271107991536 | finshed @ Wed Feb  1 16:51:34 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 3 exponential


3/3 [==============================] - 0s 1ms/step
```

```
Cycle 0 -> MAE: 2.20582139078776 (epochs: 300)
Chromosome score: 2.20582139078776 | finshed @ Wed Feb  1 16:52:04 2023
Elites | Minimum: 2.122076235453288 <-> Maximum: 2.1801394220987955

Generation 7 started @ Wed Feb  1 16:52:04 2023

Learning rate: 0.1
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 tanh
L2: 7 softmax
L3: 5 softplus


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2868609339396158 (epochs: 43)
Chromosome score: 2.2868609339396158 | finshed @ Wed Feb  1 16:52:08 2023
Learning rate: 0.1
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 tanh
L2: 7 softmax
L3: 5 softplus


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.57879495493571 (epochs: 54)
Chromosome score: 2.57879495493571 | finshed @ Wed Feb  1 16:52:12 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 8 softsign
L2: 5 sigmoid
L3: 4 relu


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.197681312561035 (epochs: 258)
Chromosome score: 2.197681312561035 | finshed @ Wed Feb  1 16:52:29 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 8 softsign
L2: 5 sigmoid
L3: 4 relu


3/3 [==============================] - 0s 1ms/step
```

```
Cycle 0 -> MAE: 2.2238697611490883 (epochs: 300)
Chromosome score: 2.2238697611490883 | finshed @ Wed Feb  1 16:52:48 2023
Learning rate: 0.05
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softmax
L2: 6 tanh
L3: 3 none


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 3.1082034403483076 (epochs: 300)
Chromosome score: 3.1082034403483076 | finshed @ Wed Feb  1 16:52:59 2023
Learning rate: 0.05
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softmax
L2: 6 tanh
L3: 3 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.5913340581258137 (epochs: 300)
Chromosome score: 2.5913340581258137 | finshed @ Wed Feb  1 16:53:09 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 7 softsign
L2: 6 sigmoid
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2499262720743816 (epochs: 266)
Chromosome score: 2.2499262720743816 | finshed @ Wed Feb  1 16:53:25 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 7 softsign
L2: 6 sigmoid
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.134596817016601 (epochs: 264)
Chromosome score: 2.134596817016601 | finshed @ Wed Feb  1 16:53:40 2023
Learning rate: 0.001
Batch size: 16
```

```
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 6 softplus
L3: 5 exponential


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.2751739451090494 (epochs: 300)
Chromosome score: 2.2751739451090494 | finshed @ Wed Feb  1 16:53:59 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 6 softplus
L3: 5 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.204590881347656 (epochs: 164)
Chromosome score: 2.204590881347656 | finshed @ Wed Feb  1 16:54:10 2023
Elites | Minimum: 2.122076235453288 <-> Maximum: 2.1708086242675777

Generation 8 started @ Wed Feb  1 16:54:10 2023

Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 6.221409708658854 (epochs: 300)
Chromosome score: 6.221409708658854 | finshed @ Wed Feb  1 16:54:26 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 6.34370491027832 (epochs: 61)
Chromosome score: 6.34370491027832 | finshed @ Wed Feb  1 16:54:30 2023
Learning rate: 0.001
Batch size: 8
```

```
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 3 relu


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.136921234130859 (epochs: 146)
Chromosome score: 2.136921234130859 | finshed @ Wed Feb  1 16:54:44 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 3 relu


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.1927151997884113 (epochs: 117)
Chromosome score: 2.1927151997884113 | finshed @ Wed Feb  1 16:54:58 2023
Learning rate: 0.05
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
L1: 6 sigmoid
L2: 5 softmax
L3: 2 tanh


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 3.0898737869262694 (epochs: 53)
Chromosome score: 3.0898737869262694 | finshed @ Wed Feb  1 16:55:04 2023
Learning rate: 0.05
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
L1: 6 sigmoid
L2: 5 softmax
L3: 2 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.3287090021769203 (epochs: 49)
Chromosome score: 2.3287090021769203 | finshed @ Wed Feb  1 16:55:09 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 8 softsign
L2: 7 sigmoid
L3: 6 tanh
```

```
3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.3635601018269856 (epochs: 300)
Chromosome score: 2.3635601018269856 | finshed @ Wed Feb  1 16:55:27 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 8 softsign
L2: 7 sigmoid
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.20164422861735 (epochs: 184)
Chromosome score: 2.20164422861735 | finshed @ Wed Feb  1 16:55:38 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 7 sigmoid
L2: 6 sigmoid
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.153603441874186 (epochs: 300)
Chromosome score: 2.153603441874186 | finshed @ Wed Feb  1 16:55:55 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 7 sigmoid
L2: 6 sigmoid
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2913433100382488 (epochs: 300)
Chromosome score: 2.2913433100382488 | finshed @ Wed Feb  1 16:56:13 2023
Elites | Minimum: 2.122076235453288 <-> Maximum: 2.168891184488932

Generation 9 started @ Wed Feb  1 16:56:13 2023

Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 sigmoid
L2: 7 sigmoid
L3: 6 exponential
```

```
3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.1888298009236653 (epochs: 300)
Chromosome score: 2.1888298009236653 | finshed @ Wed Feb  1 16:56:31 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 sigmoid
L2: 7 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.1590010732014973 (epochs: 300)
Chromosome score: 2.1590010732014973 | finshed @ Wed Feb  1 16:56:48 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 7 sigmoid
L2: 6 sigmoid
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.1864074732462564 (epochs: 300)
Chromosome score: 2.1864074732462564 | finshed @ Wed Feb  1 16:57:05 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 7 sigmoid
L2: 6 sigmoid
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.1806534347534177 (epochs: 300)
Chromosome score: 2.1806534347534177 | finshed @ Wed Feb  1 16:57:22 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2262689870198566 (epochs: 147)
```

```
Chromosome score: 2.2262689870198566 | finshed @ Wed Feb  1 16:57:31 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2100700505574546 (epochs: 300)
Chromosome score: 2.2100700505574546 | finshed @ Wed Feb  1 16:57:49 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softsign
L2: 7 softmax
L3: 6 sigmoid


3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 6.21809076944987 (epochs: 300)
Chromosome score: 6.21809076944987 | finshed @ Wed Feb  1 16:58:16 2023
Learning rate: 0.001
Batch size: 8
Optimizer: <class 'keras.optimizers.optimizer_experimental.sgd.SGD'>
L1: 7 softsign
L2: 7 softmax
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 6.379955190022787 (epochs: 300)
Chromosome score: 6.379955190022787 | finshed @ Wed Feb  1 16:58:43 2023
Learning rate: 0.05
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
L1: 6 selu
L2: 6 none
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.360387776692708 (epochs: 206)
Chromosome score: 2.360387776692708 | finshed @ Wed Feb  1 16:58:51 2023
Learning rate: 0.05
Batch size: 48
Optimizer: <class 'keras.optimizers.optimizer_experimental.adamax.Adamax'>
```

```
L1: 6 selu
L2: 6 tanh
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2248425674438477 (epochs: 176)
Chromosome score: 2.2248425674438477 | finshed @ Wed Feb  1 16:58:58 2023
Elites | Minimum: 2.122076235453288 <-> Maximum: 2.1684851989746092

Generation 10 started @ Wed Feb  1 16:58:58 2023

Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 softmax
L3: 3 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2245327529907226 (epochs: 300)
Chromosome score: 2.2245327529907226 | finshed @ Wed Feb  1 16:59:17 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 softmax
L3: 3 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.3027952473958333 (epochs: 300)
Chromosome score: 2.3027952473958333 | finshed @ Wed Feb  1 16:59:35 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 sigmoid
L2: 6 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.16990385945638 (epochs: 300)
Chromosome score: 2.16990385945638 | finshed @ Wed Feb  1 16:59:53 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
```

```
L1: 7 sigmoid
L2: 6 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2882749964396156 (epochs: 300)
Chromosome score: 2.2882749964396156 | finshed @ Wed Feb  1 17:00:11 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 exponential
L2: 6 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2636570231119792 (epochs: 203)
Chromosome score: 2.2636570231119792 | finshed @ Wed Feb  1 17:00:24 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 exponential
L2: 6 sigmoid
L3: 6 sigmoid


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2491722386678057 (epochs: 242)
Chromosome score: 2.2491722386678057 | finshed @ Wed Feb  1 17:00:41 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 tanh


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2021655069986976 (epochs: 231)
Chromosome score: 2.2021655069986976 | finshed @ Wed Feb  1 17:00:55 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.adam.Adam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 tanh
```

```
3/3 [==============================] - 0s 2ms/step
Cycle 0 -> MAE: 2.245358184814453 (epochs: 233)
Chromosome score: 2.245358184814453 | finshed @ Wed Feb  1 17:01:09 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.139854988098145 (epochs: 267)
Chromosome score: 2.139854988098145 | finshed @ Wed Feb  1 17:01:25 2023
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 softsign
L2: 7 sigmoid
L3: 6 exponential


3/3 [==============================] - 0s 1ms/step
Cycle 0 -> MAE: 2.2779237543741857 (epochs: 300)
Chromosome score: 2.2779237543741857 | finshed @ Wed Feb  1 17:01:44 2023
Elites | Minimum: 2.122076235453288 <-> Maximum: 2.1615822906494135



Best model: (score: 2.122076)
Learning rate: 0.001
Batch size: 16
Optimizer: <class 'keras.optimizers.optimizer_experimental.nadam.Nadam'>
L1: 7 sigmoid
L2: 7 sigmoid
L3: 3 exponential
```

[40]:
```python
# Create a new model with the best chromosome
model = get_model(fittest_model[1], train_features, train_labels)

# Evaluate the model
predictions = model.predict(test_features).flatten()

mlp_predictions = denormalise_y(predictions, *normalisation_bounds[Y])
```

```
print(f"Mean absolute error: {mean_absolute_error(unnormalised_test_labels,␣
 ↪mlp_predictions):0.2f} MPG")
print(f"Mean absolute percentage error:␣
 ↪{mean_absolute_percentage_error(unnormalised_test_labels, mlp_predictions):0.
 ↪3f}%")
print(f"Root mean squared error: {mean_squared_error(unnormalised_test_labels,␣
 ↪mlp_predictions, squared=False):0.2f} MPG")
```

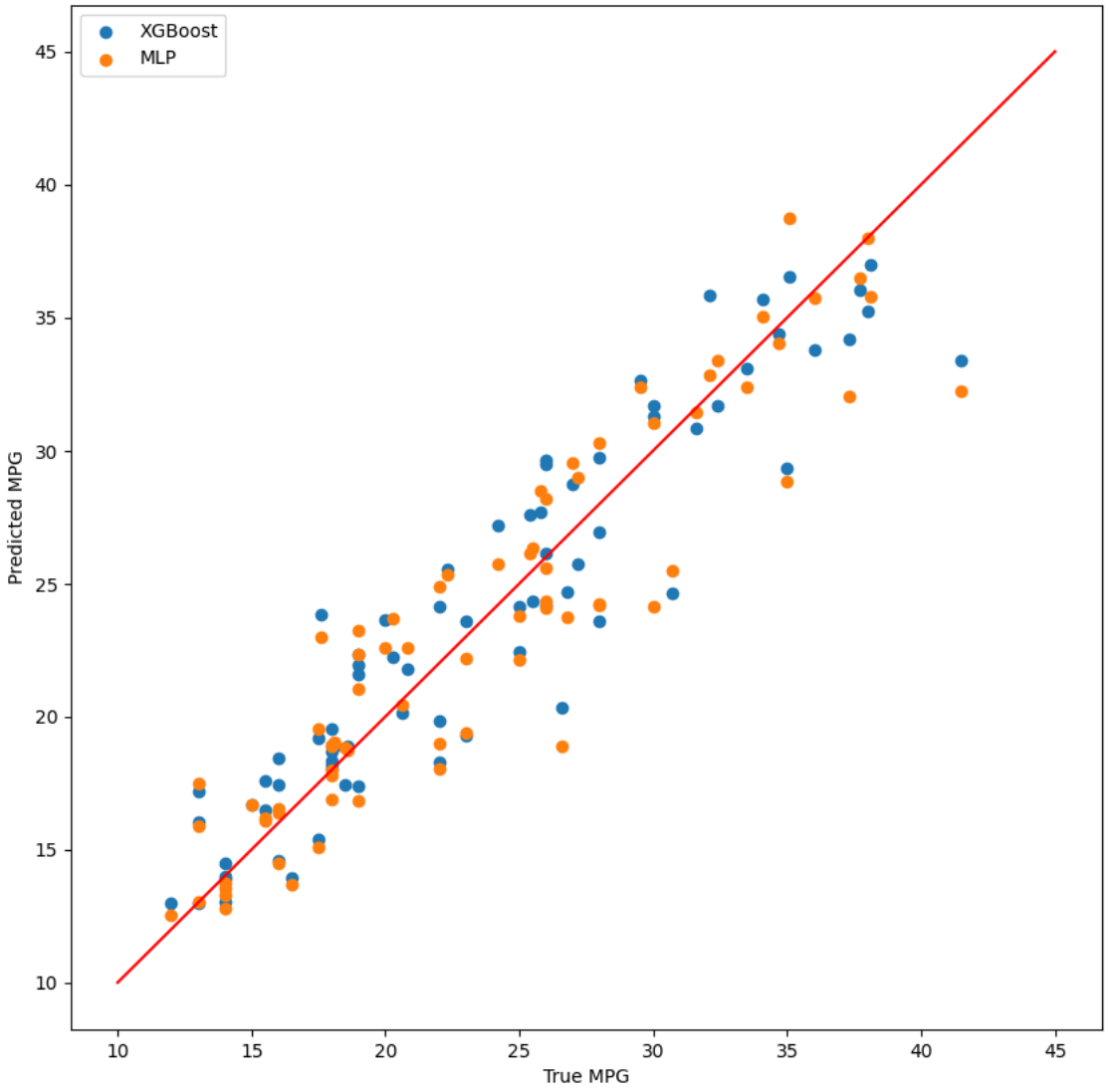
```
3/3 [==============================] - 0s 2ms/step
Mean absolute error: 2.12 MPG
Mean absolute percentage error: 0.091%
Root mean squared error: 2.82 MPG
```

1.5 Results analysis

Plotting the results of the models, and the residuals of both models.

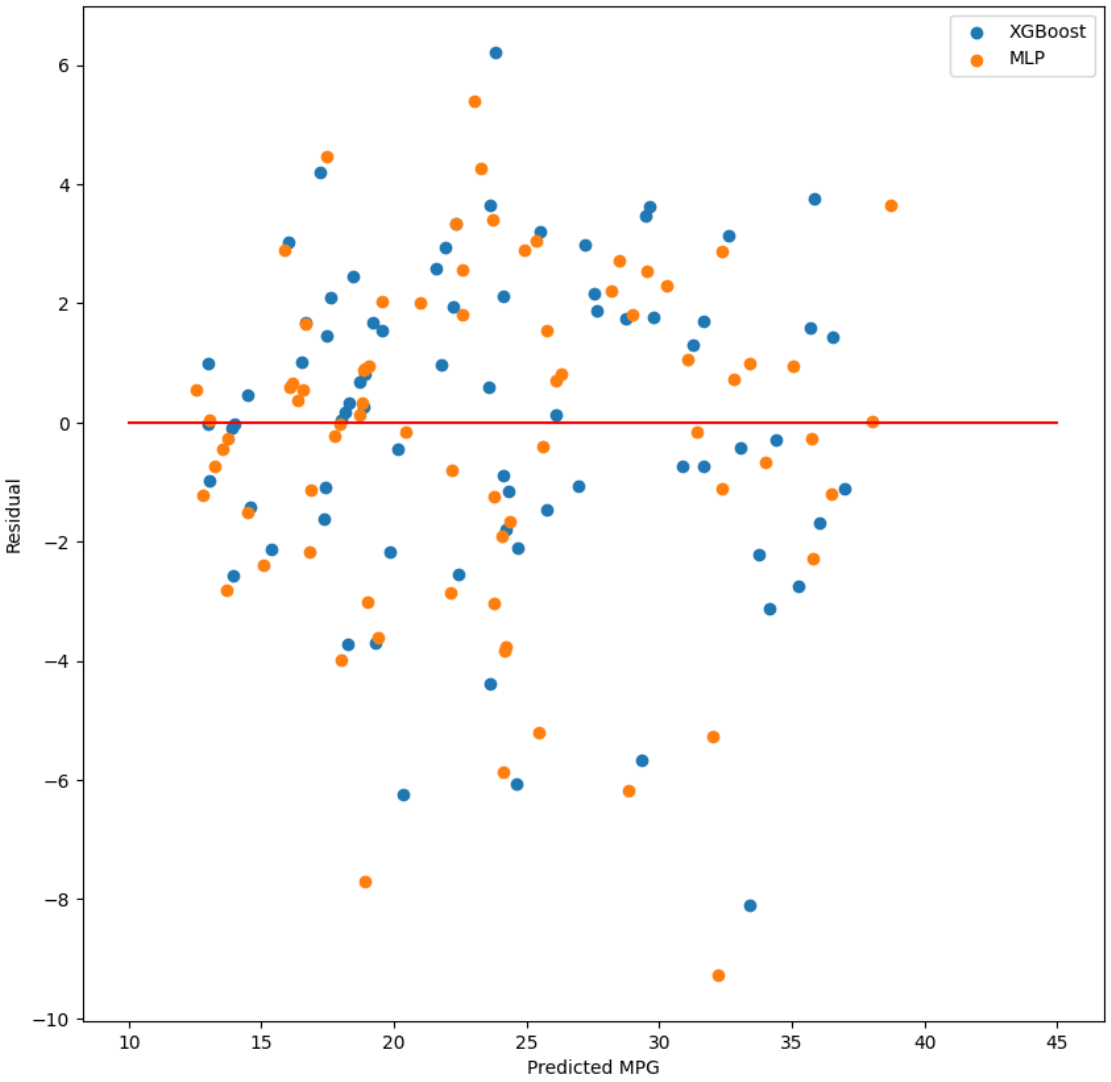
[41]:
```
# Plot both xgb_predictions and mlp_predictions on the same graph
# With a red line for the perfect prediction

plt.figure(figsize=(10, 10))
plt.scatter(unnormalised_test_labels, xgb_predictions, label="XGBoost")
plt.scatter(unnormalised_test_labels, mlp_predictions, label="MLP")
plt.plot([10, 45], [10, 45], color="red")
plt.xlabel("True MPG")
plt.ylabel("Predicted MPG")
plt.legend()
plt.show()
```



[42]:
```
# Plot the both of the residuals on the same graph

plt.figure(figsize=(10, 10))
plt.scatter(xgb_predictions, xgb_predictions - unnormalised_test_labels,␣
 ↪label="XGBoost")
plt.scatter(mlp_predictions, mlp_predictions - unnormalised_test_labels,␣
 ↪label="MLP")
plt.plot([10, 45], [0, 0], color="red")
plt.xlabel("Predicted MPG")
plt.ylabel("Residual")
plt.legend()
plt.show()
```



Both models have a similar MAE, however the XGBoost has a lower RMSE, this indicates that the XGBoost is has a smaller variance in errors Source, and is more likely to generalise well to new data, the MLP on the other hand is more likely to be overfitted to the data, it also has a few large errors, which is not ideal. In addition to this the XGBoost is much easier to explain to others, as it is a tree-based model, and the decision trees can be visualised which is not possible with the MLP.

1.6 Conclusion

Both models have an error of ~2 MPG which is not perfect, however the variance in MPG of a car can be affected by many factors that aren't included in this dataset, such as the driver, the road conditions, and the weather. If more data was available then the MLP could be a better model, however with the relatively small dataset of ~400 instances, the underlying patterns in the data are likely to be simple, or not adequately captured. However on this particular dataset I'd recommend the XGBoost model, as it has a lower variance in errors, and thus is more reliable.