title: "ML_with_sklearn" author: Aarushi Pandey date: 11/4/2022

# ML with sklearn

## 1. Read the Auto data

In [44]:
```
### a. use pandas to read the data
import pandas as pd
df = pd.read_csv('/data/Auto.csv') # use all columns

### b. output the first few rows
print(df.head())

### c. output the dimensions of the data
print('\nDimensions of data frame:', df.shape)  # 392 rows and 9 cols
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
0  18.0          8         307.0         130    3504          12.0  70.0   
1  15.0          8         350.0         165    3693          11.5  70.0   
2  18.0          8         318.0         150    3436          11.0  70.0   
3  16.0          8         304.0         150    3433          12.0  70.0   
4  17.0          8         302.0         140    3449           NaN  70.0   

   origin                       name  
0       1  chevrolet chevelle malibu  
1       1          buick skylark 320  
2       1         plymouth satellite  
3       1              amc rebel sst  
4       1                ford torino  

Dimensions of data frame: (392, 9)
```

#### 2. Data exploration with code

In [45]:
```python
### a. use describe() on the mpg, weight, and year columns
print(df.mpg.describe())
print(df.weight.describe())
print(df.year.describe())

### b. write comments indicating the range and average of each column
print(df.cylinders.describe())
print(df.displacement.describe())
print(df.horsepower.describe())
print(df.acceleration.describe())
print(df.origin.describe())
```

```
count    392.000000
mean      23.445918
std        7.805007
min        9.000000
25%       17.000000
50%       22.750000
75%       29.000000
max       46.600000
Name: mpg, dtype: float64
count     392.000000
mean     2977.584184
std       849.402560
min      1613.000000
25%      2225.250000
50%      2803.500000
75%      3614.750000
max      5140.000000
Name: weight, dtype: float64
count    390.000000
mean      76.010256
std        3.668093
min       70.000000
25%       73.000000
50%       76.000000
75%       79.000000
max       82.000000
Name: year, dtype: float64
count    392.000000
mean       5.471939
std        1.705783
min        3.000000
25%        4.000000
50%        4.000000
75%        8.000000
max        8.000000
Name: cylinders, dtype: float64
count    392.000000
mean     194.411990
std      104.644004
min       68.000000
25%      105.000000
50%      151.000000
75%      275.750000
max      455.000000
Name: displacement, dtype: float64
count    392.000000
mean     104.469388
std       38.491160
min       46.000000
25%       75.000000
50%       93.500000
75%      126.000000
max      230.000000
Name: horsepower, dtype: float64
count    391.000000
mean      15.554220
std        2.750548
min        8.000000
25%       13.800000
50%       15.500000
75%       17.050000
```

```
max        24.800000
Name: acceleration, dtype: float64
count    392.000000
mean       1.576531
std        0.805518
min        1.000000
25%        1.000000
50%        1.000000
75%        2.000000
max        3.000000
Name: origin, dtype: float64
```

mpg column:- Range= 46.600000 - 9.000000 = 37.600000, Average= 23.445918

weight column:- Range= 5140.000000 - 1613.000000 = 3527.000000, Average= 2977.584184

year column:- Range= 82.000000 - 70.000000 = 12.000000 , Average= 76.010256

cylinders column: Range= 8.000000 - 3.000000 = 5.000000 , Average = 5.471939

displacement column: Range= 455.000000 - 68.000000 = 387.000000, Average = 194.411990

horsepower column: Range= 230.000000 - 46.000000 = 184.000000 , Average = 104.469388

acceleration column: Range= 24.800000 - 8.000000 = 16.000000, Average = 15.554220

origin column: Range= 3.000000 - 1.000000 = 2.000000, Average = 1.576531

## 3. Explore data types

In [46]:
```
### a. check the data types of all columns
df.dtypes
```

Out[46]:
```
mpg             float64
cylinders         int64
displacement    float64
horsepower        int64
weight            int64
acceleration    float64
year            float64
origin            int64
name             object
dtype: object
```

Originally, cylinders, horsepower, weight, and origin columns contained values of int64 type, and all other columns except name column (mpg, displacement, acceleration, year) contained values of float64 type. The name column contained values of object type.

```
In [47]: ### b. change the cylinders column to categorical (use cat.codes)
df2 = df.copy()   # copied df in case of conversion errors
df2.cylinders = df.cylinders.astype('category').cat.codes

### c. change the origin column to categorical (don’t use cat.codes)
df2.origin = df.origin.astype('category')

### d. verify the changes with the dtypes attribute
df2.dtypes
df = df2.copy()
df.dtypes
```

```
Out[47]: mpg              float64
cylinders           int8
displacement     float64
horsepower         int64
weight             int64
acceleration     float64
year             float64
origin          category
name              object
dtype: object
```

## 4. Deal with NAs

```
In [48]: ### a. delete rows with NAs
df = df.dropna()

### b. output the new dimensions
print('\nDimensions of data frame:', df.shape) # from 392 to 389 rows
```

```
Dimensions of data frame: (389, 9)
```

## 5. Modify columns

In [49]:
```
### a. make a new column, mpg_high, and make it categorical:
###   i. the column == 1 if mpg > average mpg, else == 0
df3 = df.copy()

import numpy as np
mpg_mean = np.mean(df.mpg)

l = []
for i, mpg in enumerate(df.mpg):
  if mpg > mpg_mean:
    l.append(1)
  else:
    l.append(0)

df3['mpg_high'] = l
df3.mpg_high = df3.mpg_high.astype('category')

### b. delete the mpg and name columns (delete mpg so the algorithm doesn’t just learn t
o predict mpg_high from mpg)
df3 = df3.drop(columns=['mpg', 'name'])
df = df3.copy()
#print(df3.mpg_high)

### c. output the first few rows of the modified data frame
print(df.head())
```

```
   cylinders  displacement  horsepower  weight  acceleration  year origin  \
0          4         307.0         130    3504          12.0  70.0      1
1          4         350.0         165    3693          11.5  70.0      1
2          4         318.0         150    3436          11.0  70.0      1
3          4         304.0         150    3433          12.0  70.0      1
6          4         454.0         220    4354           9.0  70.0      1

  mpg_high
0        0
1        0
2        0
3        0
6        0
```

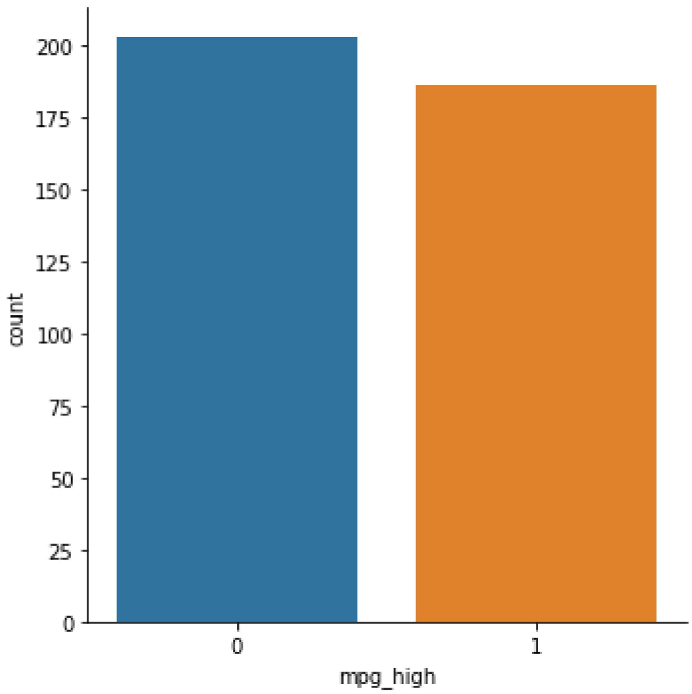
## 6. Data exploration with graphs

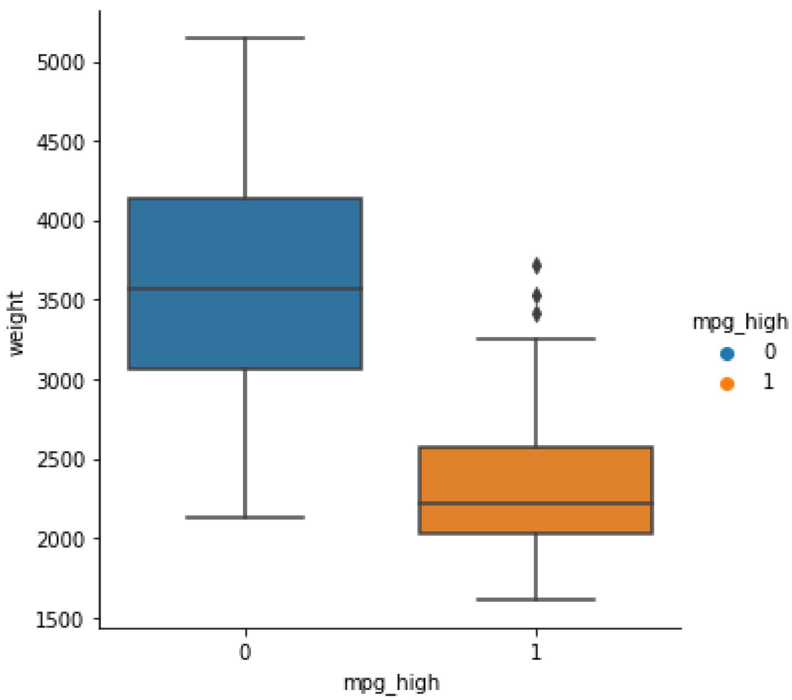
In [50]:

```
### a. seaborn catplot on the mpg_high column
import seaborn as sb
sb.catplot(x="mpg_high", kind='count', data=df)  # about equal amount of 0s and 1s, with
~25 more 0s.

### b. seaborn relplot with horsepower on the x axis, weight on the y axis, setting hue
 or
### style to mpg_high
sb.relplot(x='horsepower', y='weight', data=df, hue=df.mpg_high)  # if mpg_high=1, horse
power is less (between 50 and 125)

### c. seaborn boxplot with mpg_high on the x axis and weight on the y axis
sb.boxplot('mpg_high', y='weight', data=df) # if mpg_high=0, median weight is ~3500 and
 if mpg_high=1, median weight is ~2200 (huge difference in values depending on mpg_weigh
t)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass t
he following variable as a keyword arg: x. From version 0.12, the only valid positional
argument will be `data`, and passing other arguments without an explicit keyword will r
esult in an error or misinterpretation.
  FutureWarning
```

Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x7fc3058ba6d0>




In the first graph, there are about equal amount of 0s (around 200) and 1s (about 185), with ~15 more 0s. In the second graph, the horsepower is less if mpg_high=1 (between 50 and 125) and a lot more if mpg_high=0. In the third graph, if mpg_high=0, median weight is ~3500 and if mpg_high=1, median weight is ~2200 (huge difference in values depending on mpg_weight).

## 7. Train/test split

In [51]:
```
### a. 80/20
### b. use seed 1234 so we all get the same results
### c. train /test X data frames consists of all remaining columns except mpg_high
import numpy as np
np.random.seed(1234)

from sklearn.model_selection import train_test_split
X = df.loc[:, df.columns!='mpg_high']
y = df.mpg_high

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=12
34)

### d. output the dimensions of train and test
print('train size:', X_train.shape)
print('test size:', X_test.shape)
```

```
train size: (311, 7)
test size: (78, 7)
```

## 8. Logistic Regression

In [52]:
```
### a. train a logistic regression model using solver lbfgs
from sklearn.linear_model import LogisticRegression
clf1 = LogisticRegression(solver='lbfgs', max_iter=2000)
clf1.fit(X_train, y_train)

### b. test and evaluate
clf1.score(X_train, y_train)
pred1 = clf1.predict(X_test)

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
print('accuracy score: ', accuracy_score(y_test, pred1))
print('precision score: ', precision_score(y_test, pred1))
print('recall score: ', recall_score(y_test, pred1))
print('f1 score: ', f1_score(y_test, pred1))

### c. print metrics using the classification report
from sklearn.metrics import classification_report
print(classification_report(y_test, pred1))
```

```
accuracy score:  0.8974358974358975
precision score:  0.7777777777777778
recall score:  1.0
f1 score:  0.8750000000000001
              precision    recall  f1-score   support

           0       1.00      0.84      0.91        50
           1       0.78      1.00      0.88        28

    accuracy                           0.90        78
   macro avg       0.89      0.92      0.89        78
weighted avg       0.92      0.90      0.90        78
```

## 9. Decision Tree

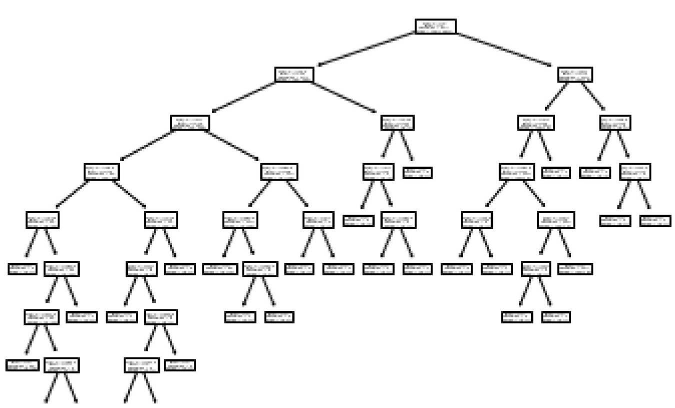
In [53]:

```
### a. train a decision tree
from sklearn.tree import DecisionTreeClassifier
clf2 = DecisionTreeClassifier()
clf2.fit(X_train, y_train)

### b. test and evaluate
pred2 = clf2.predict(X_test)

#from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
print('accuracy score: ', accuracy_score(y_test, pred2))
print('precision score: ', precision_score(y_test, pred2))
print('recall score: ', recall_score(y_test, pred2))
print('f1 score: ', f1_score(y_test, pred2))

### c. print the classification report metrics
#from sklearn.metrics import classification_report
print(classification_report(y_test, pred2))

### d. plot the tree (optional)
from sklearn import tree
tree.plot_tree(clf2)
```

```
accuracy score:  0.9230769230769231
precision score:  0.8666666666666667
recall score:  0.9285714285714286
f1 score:  0.896551724137931
              precision    recall  f1-score   support

           0       0.96      0.92      0.94        50
           1       0.87      0.93      0.90        28

    accuracy                           0.92        78
   macro avg       0.91      0.92      0.92        78
weighted avg       0.93      0.92      0.92        78
```

Out[53]:
```
[Text(0.6433823529411765, 0.9444444444444444, 'X[0] <= 2.5\ngini = 0.5\nsamples = 311\n
value = [153, 158]'),
 Text(0.4338235294117647, 0.8333333333333334, 'X[2] <= 101.0\ngini = 0.239\nsamples = 1
73\nvalue = [24, 149]'),
 Text(0.27941176470588236, 0.7222222222222222, 'X[5] <= 75.5\ngini = 0.179\nsamples = 1
61\nvalue = [16, 145]'),
 Text(0.14705882352941177, 0.6111111111111112, 'X[1] <= 119.5\ngini = 0.362\nsamples =
59\nvalue = [14, 45]'),
 Text(0.058823529411764705, 0.5, 'X[4] <= 13.75\ngini = 0.159\nsamples = 46\nvalue =
[4, 42]'),
 Text(0.029411764705882353, 0.3888888888888889, 'gini = 0.0\nsamples = 2\nvalue = [2,
0]'),
 Text(0.08823529411764706, 0.3888888888888889, 'X[3] <= 2683.0\ngini = 0.087\nsamples =
44\nvalue = [2, 42]'),
 Text(0.058823529411764705, 0.2777777777777778, 'X[3] <= 2377.0\ngini = 0.045\nsamples
= 43\nvalue = [1, 42]'),
 Text(0.029411764705882353, 0.16666666666666666, 'gini = 0.0\nsamples = 38\nvalue = [0,
38]'),
 Text(0.08823529411764706, 0.16666666666666666, 'X[3] <= 2385.0\ngini = 0.32\nsamples =
5\nvalue = [1, 4]'),
 Text(0.058823529411764705, 0.05555555555555555, 'gini = 0.0\nsamples = 1\nvalue = [1,
0]'),
 Text(0.11764705882352941, 0.05555555555555555, 'gini = 0.0\nsamples = 4\nvalue = [0,
4]'),
 Text(0.11764705882352941, 0.2777777777777778, 'gini = 0.0\nsamples = 1\nvalue = [1,
0]'),
 Text(0.23529411764705882, 0.5, 'X[4] <= 17.75\ngini = 0.355\nsamples = 13\nvalue = [1
0, 3]'),
 Text(0.20588235294117646, 0.3888888888888889, 'X[2] <= 81.5\ngini = 0.469\nsamples = 8
\nvalue = [5, 3]'),
 Text(0.17647058823529413, 0.2777777777777778, 'gini = 0.0\nsamples = 2\nvalue = [0,
2]'),
 Text(0.23529411764705882, 0.2777777777777778, 'X[3] <= 2329.5\ngini = 0.278\nsamples =
6\nvalue = [5, 1]'),
 Text(0.20588235294117646, 0.16666666666666666, 'X[3] <= 2242.0\ngini = 0.5\nsamples =
2\nvalue = [1, 1]'),
 Text(0.17647058823529413, 0.05555555555555555, 'gini = 0.0\nsamples = 1\nvalue = [1,
0]'),
 Text(0.23529411764705882, 0.05555555555555555, 'gini = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.2647058823529412, 0.16666666666666666, 'gini = 0.0\nsamples = 4\nvalue = [4,
0]'),
 Text(0.2647058823529412, 0.3888888888888889, 'gini = 0.0\nsamples = 5\nvalue = [5,
0]'),
 Text(0.4117647058823529, 0.6111111111111112, 'X[3] <= 3250.0\ngini = 0.038\nsamples =
102\nvalue = [2, 100]'),
 Text(0.35294117647058826, 0.5, 'X[3] <= 2880.0\ngini = 0.02\nsamples = 100\nvalue =
[1, 99]'),
 Text(0.3235294117647059, 0.3888888888888889, 'gini = 0.0\nsamples = 94\nvalue = [0, 9
4]'),
 Text(0.38235294117647056, 0.3888888888888889, 'X[3] <= 2920.0\ngini = 0.278\nsamples =
6\nvalue = [1, 5]'),
 Text(0.35294117647058826, 0.2777777777777778, 'gini = 0.0\nsamples = 1\nvalue = [1,
0]'),
 Text(0.4117647058823529, 0.2777777777777778, 'gini = 0.0\nsamples = 5\nvalue = [0,
5]'),
 Text(0.47058823529411764, 0.5, 'X[4] <= 21.0\ngini = 0.5\nsamples = 2\nvalue = [1,
1]'),
 Text(0.4411764705882353, 0.3888888888888889, 'gini = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.5, 0.3888888888888889, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
```

```
 Text(0.5882352941176471, 0.7222222222222222, 'X[4] <= 14.45\ngini = 0.444\nsamples = 1
2\nvalue = [8, 4]'),
 Text(0.5588235294117647, 0.6111111111111112, 'X[5] <= 76.0\ngini = 0.444\nsamples = 6
\nvalue = [2, 4]'),
 Text(0.5294117647058824, 0.5, 'gini = 0.0\nsamples = 3\nvalue = [0, 3]'),
 Text(0.5882352941176471, 0.5, 'X[3] <= 2760.0\ngini = 0.444\nsamples = 3\nvalue = [2,
1]'),
 Text(0.5588235294117647, 0.3888888888888889, 'gini = 0.0\nsamples = 2\nvalue = [2,
0]'),
 Text(0.6176470588235294, 0.3888888888888889, 'gini = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.6176470588235294, 0.6111111111111112, 'gini = 0.0\nsamples = 6\nvalue = [6,
0]'),
 Text(0.8529411764705882, 0.8333333333333334, 'X[5] <= 79.5\ngini = 0.122\nsamples = 13
8\nvalue = [129, 9]'),
 Text(0.7941176470588235, 0.7222222222222222, 'X[4] <= 21.6\ngini = 0.045\nsamples = 12
9\nvalue = [126, 3]'),
 Text(0.7647058823529411, 0.6111111111111112, 'X[3] <= 2737.0\ngini = 0.031\nsamples =
128\nvalue = [126, 2]'),
 Text(0.7058823529411765, 0.5, 'X[2] <= 111.0\ngini = 0.444\nsamples = 3\nvalue = [2,
1]'),
 Text(0.6764705882352942, 0.3888888888888889, 'gini = 0.0\nsamples = 2\nvalue = [2,
0]'),
 Text(0.7352941176470589, 0.3888888888888889, 'gini = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.8235294117647058, 0.5, 'X[2] <= 83.0\ngini = 0.016\nsamples = 125\nvalue = [12
4, 1]'),
 Text(0.7941176470588235, 0.3888888888888889, 'X[2] <= 79.5\ngini = 0.375\nsamples = 4
\nvalue = [3, 1]'),
 Text(0.7647058823529411, 0.2777777777777778, 'gini = 0.0\nsamples = 3\nvalue = [3,
0]'),
 Text(0.8235294117647058, 0.2777777777777778, 'gini = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.8529411764705882, 0.3888888888888889, 'gini = 0.0\nsamples = 121\nvalue = [121,
0]'),
 Text(0.8235294117647058, 0.6111111111111112, 'gini = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.9117647058823529, 0.7222222222222222, 'X[1] <= 196.5\ngini = 0.444\nsamples = 9
\nvalue = [3, 6]'),
 Text(0.8823529411764706, 0.6111111111111112, 'gini = 0.0\nsamples = 4\nvalue = [0,
4]'),
 Text(0.9411764705882353, 0.6111111111111112, 'X[1] <= 247.0\ngini = 0.48\nsamples = 5
\nvalue = [3, 2]'),
 Text(0.9117647058823529, 0.5, 'gini = 0.0\nsamples = 3\nvalue = [3, 0]'),
 Text(0.9705882352941176, 0.5, 'gini = 0.0\nsamples = 2\nvalue = [0, 2]')]
```



## 10. Neural Network

In [54]:

```
### a. train a neural network, choosing a network topology of your choice
# normalize the data
from sklearn import preprocessing
scaler = preprocessing.StandardScaler().fit(X_train)

X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# train
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(5, 2), max_iter=500, random_stat
e=1234)
clf.fit(X_train_scaled, y_train)

### b. test and evaluate
pred3 = clf.predict(X_test_scaled)
#from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
print('accuracy score: ', accuracy_score(y_test, pred3))
print('precision score: ', precision_score(y_test, pred3))
print('recall score: ', recall_score(y_test, pred3))
print('f1 score: ', f1_score(y_test, pred3))

print(classification_report(y_test, pred3))

### c. train a second network with a different topology and different settings
clf1 = MLPClassifier(solver='sgd', hidden_layer_sizes=(3,), max_iter=1500, random_state=
1234)
clf1.fit(X_train_scaled, y_train)

### d. test and evaluate
pred4 = clf.predict(X_test_scaled)
#from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
print('accuracy score: ', accuracy_score(y_test, pred4))
print('precision score: ', precision_score(y_test, pred4))
print('recall score: ', recall_score(y_test, pred4))
print('f1 score: ', f1_score(y_test, pred4))

print(classification_report(y_test, pred4))
```

```
accuracy score:  0.8717948717948718
precision score:  0.78125
recall score:  0.8928571428571429
f1 score:  0.8333333333333334
              precision    recall  f1-score   support

           0       0.93      0.86      0.90        50
           1       0.78      0.89      0.83        28

    accuracy                           0.87        78
   macro avg       0.86      0.88      0.86        78
weighted avg       0.88      0.87      0.87        78

accuracy score:  0.8717948717948718
precision score:  0.78125
recall score:  0.8928571428571429
f1 score:  0.8333333333333334
              precision    recall  f1-score   support

           0       0.93      0.86      0.90        50
           1       0.78      0.89      0.83        28

    accuracy                           0.87        78
   macro avg       0.86      0.88      0.86        78
weighted avg       0.88      0.87      0.87        78
```

The two models had the same accuracies (0.8717948717948718), precision (0.78125), recall (0.8928571428571429), and f1 scores (0.8333333333333334). I think that the performance was the same because the target variable (mpg_high) could be accurately predicted in 3 nodes in one layer, and the other nodes are not necessary. This is probably a case of the relationship between the target variable and predictors being simple.

## 11. Analysis

a. which algorithm performed better? Decision tree algorithm performed the best, followed by logistic regression and neural network.

b. compare accuracy, recall and precision metrics by class. The accuracy is the most in decision tree (0.923), followed by logistic regression (0.897) and neural network (0.872). On the other hand, the recall is most for logistic regression (1.0), followed by neural network (0.893) and decision tree (0.923). The precision is the most in decision tree (0.866), followed by neural network (0.781) and logistic regression (0.778).

c. give your analysis of why the better-performing algorithm might have outperformed the other. The accuracy is the most for decision tree because it might be a simple relationship between a few predictors and target variable. Judging from the graphs, it is easy to make linear boundaries to divide up data in an accurate way. The precision was also the most for this algorithm, and while its recall was the lowest it was still more than 90%. Logistic regression also performed well, having the perfect recall. This might be because a few predictors' values may be enough to predict the target variable. The neural network algorithm works well with complex relationships, which this one is not. It might have led to overfitting with this algorithm.

d. write a couple of sentences comparing your experiences using R versus sklearn. Feel free to express strong preferences. Using R feels more natural than using sklearn at this point of the course. Most commands are pretty intuitive and can be found after some research, but the same can be said for sklearn commands too. Models in R could be summarized (to display important predictors etc.) and were more interpretable than in sklearn, which was really helpful to me. (I also don't like how the runtime disconnects after a few minutes of inactivity in Colab, but I suppose that is not a disadvantage for sklearn in particular.)