Python ML with sklearn

Anurag Diwate

6 April 2023

In [79]:
```
# Import Statements
import sklearn
import io
import random
import tensorflow as tf
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_
from sklearn.metrics import classification_report
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree, preprocessing
from sklearn.pipeline import Pipeline
from sklearn.neural_network import MLPClassifier
```

In [80]:
```
# 1. Read the Auto data
from google.colab import files
uploaded = files.upload()
df = pd.read_csv(io.BytesIO(uploaded['Auto.csv']))
# Print the first few rows of the data
print(df.head())
print()
# Print the dimensions of the data
print(df.size)
print(df.shape)
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving Auto.csv to Auto.csv
    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

3528
(392, 9)
```

In [81]:
```
# 2. Data Exploration with code
print(df[["mpg", "weight", "year"]].describe())

# Range:
#     mpg: 37.0
#     weight: 3527.0
#     year: 12.0

# Average:
#     mpg: 23.445918
#     weight: 2977.584184
#     year: 76.010256
```

```
              mpg       weight        year
count  392.000000   392.000000  390.000000
mean    23.445918  2977.584184   76.010256
std      7.805007   849.402560    3.668093
min      9.000000  1613.000000   70.000000
25%     17.000000  2225.250000   73.000000
50%     22.750000  2803.500000   76.000000
75%     29.000000  3614.750000   79.000000
max     46.600000  5140.000000   82.000000
```

In [82]:
```
# 3. Explore data types
print(df.dtypes)
# cylinders to categorical using cat.codes
df.cylinders = df.cylinders.astype('category').cat.codes
# origin to categorical
df = df.astype({"origin": 'category'})
# verify type changes
print(df.dtypes)
df.head()
```

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

Out[82]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	4	307.0	130	3504	12.0	70.0	1	chevrolet chevelle malibu
1	15.0	4	350.0	165	3693	11.5	70.0	1	buick skylark 320
2	18.0	4	318.0	150	3436	11.0	70.0	1	plymouth satellite
3	16.0	4	304.0	150	3433	12.0	70.0	1	amc rebel sst
4	17.0	4	302.0	140	3449	NaN	70.0	1	ford torino

In [83]:

```
# 4. Dealing with NAs
df = df.dropna()
# New dimensions
print(df.size)
print(df.shape)
```

```
3501
(389, 9)
```

In [84]:

```
# 5. Modify columns
df['mpg_high'] = np.where(df.mpg > np.mean(df.mpg), 1, 0)
df = df.astype({"mpg_high": 'category'})
# deleting mpg and name columns
df = df.drop(columns = ['mpg', 'name'])
# First rows of the modified data frame
print(df.mpg_high.head())
print(df.dtypes)
```

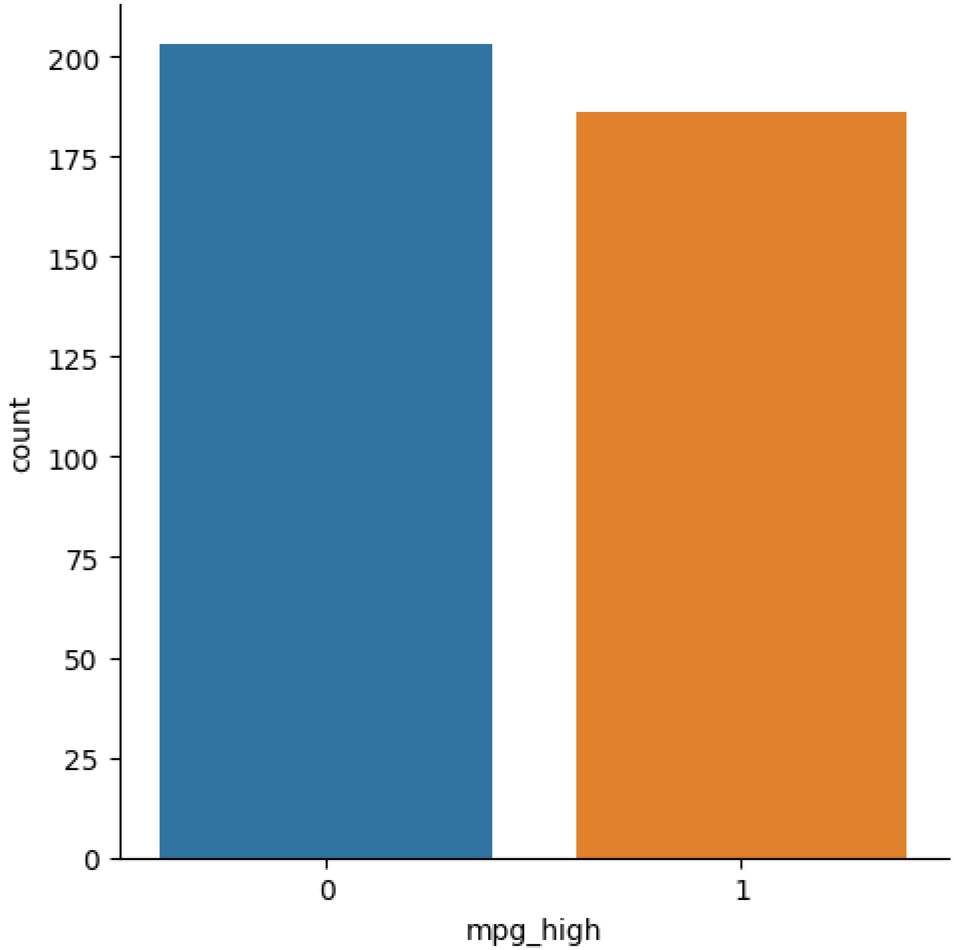
```
0    0
1    0
2    0
3    0
6    0
Name: mpg_high, dtype: category
Categories (2, int64): [0, 1]
cylinders          int8
displacement     float64
horsepower         int64
weight             int64
acceleration     float64
year             float64
origin          category
mpg_high        category
dtype: object
```

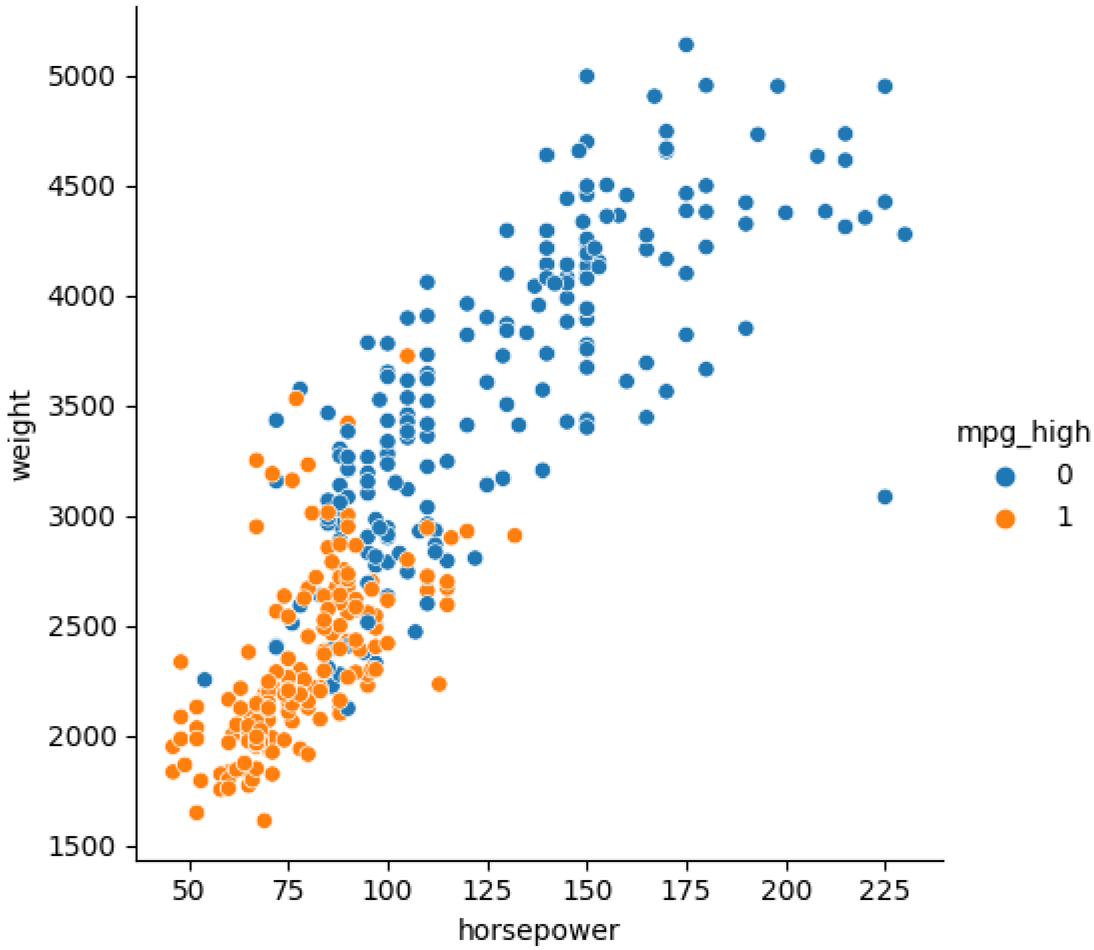
In [85]:

```
# 6. Data exploration with graphs
sns.catplot(data = df, x = "mpg_high", kind = 'count')
# seaborn relplot with horsepower on the x axis, weight on the y axis, hue or style tc
sns.relplot(data = df, x = "horsepower", y = "weight", hue = "mpg_high")
```

Out[85]:

```
<seaborn.axisgrid.FacetGrid at 0x7ff369d6dac0>
```


200
175
150
125
100
75
50
25
0
count
0
1
mpg_high

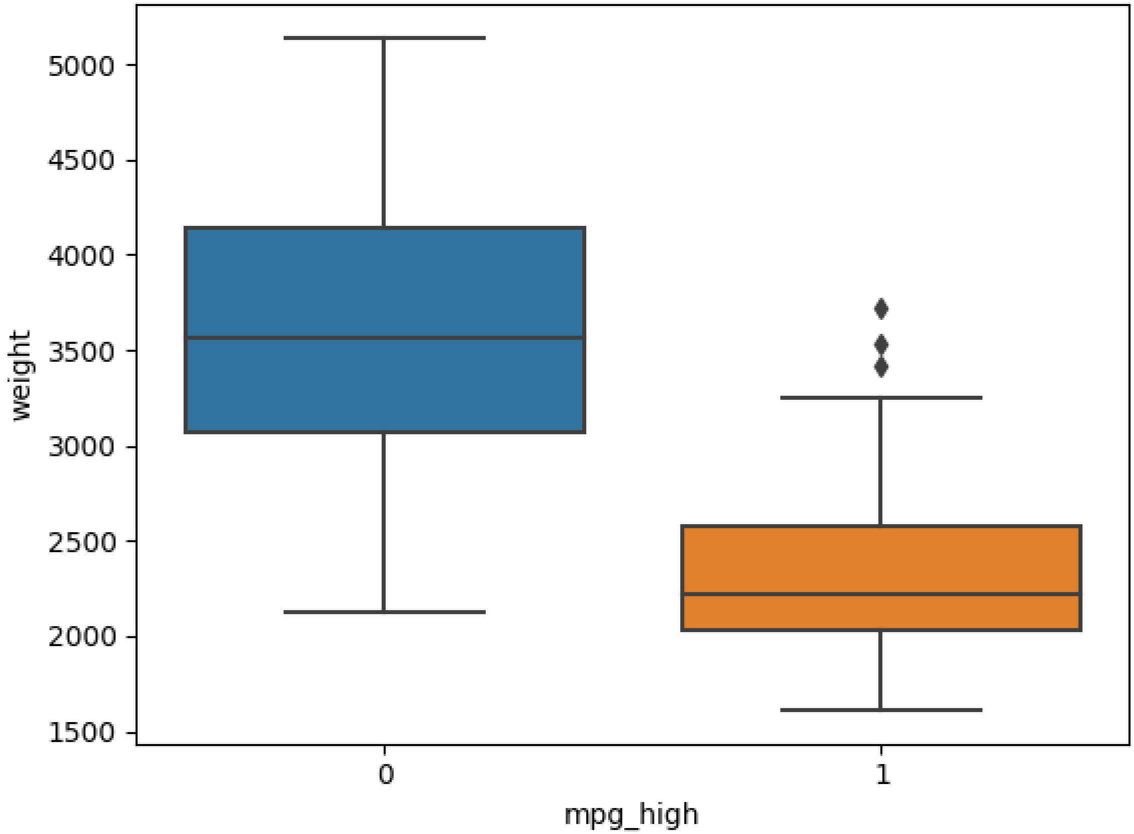


In [86]:
```
# seaborn boxplot with mph_high on the x axis, weight on the y axis
sns.boxplot(data = df, x = "mpg_high", y = "weight")
# 1. The interesting thing about the first graph is that more values are below average
#    That means there are big outliers on the higher end of the data values.

# 2. The second graph shows that with a heigher weight and a heigher horsepower, we ge

# 3. The third graph shows that with a higher weight, automobiles tend to have a lower
#    Even the automobiles that have a higher than average mpg ratio, the heavier vehic
```

Out[86]:
```
<Axes: xlabel='mpg_high', ylabel='weight'>
```



In [87]:
```
# 7. Train/Test split
# 80/20, with seed 1234
# X is all columns except mpg_high
X = df.drop(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_stat
# dimensions of train and test
print(X_train.size)
print(X_train.shape)
print(y_train.size)
print(y_train.shape)
```

```
2177
(311, 7)
311
(311,)
```

In [88]:
```
# 8. Logistic Regression
clf = LogisticRegression(solver = 'lbfgs', max_iter = 1000)
clf.fit(X_train, y_train)
clf.score(X_train, y_train)
```

Out[88]:
```
0.9067524115755627
```

In [89]:
```
# testing
pred = clf.predict(X_test)
# evaluation
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
```

```
# classification report
print(classification_report(y_test,pred))
```

```
accuracy score:  0.8589743589743589
precision score:  0.7297297297297297
recall score:  0.9642857142857143
f1 score:  0.8307692307692307
              precision    recall  f1-score   support

           0       0.98      0.80      0.88        50
           1       0.73      0.96      0.83        28

    accuracy                           0.86        78
   macro avg       0.85      0.88      0.85        78
weighted avg       0.89      0.86      0.86        78
```

In [90]:

```
# 9. Decision Tree
# train
dtc = DecisionTreeClassifier()
dtc.fit(X_train, y_train)
# test
pred = dtc.predict(X_test)
# evaluate
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
# classification report
print(classification_report(y_test,pred))
# plot the tree
tree.plot_tree(dtc)
```

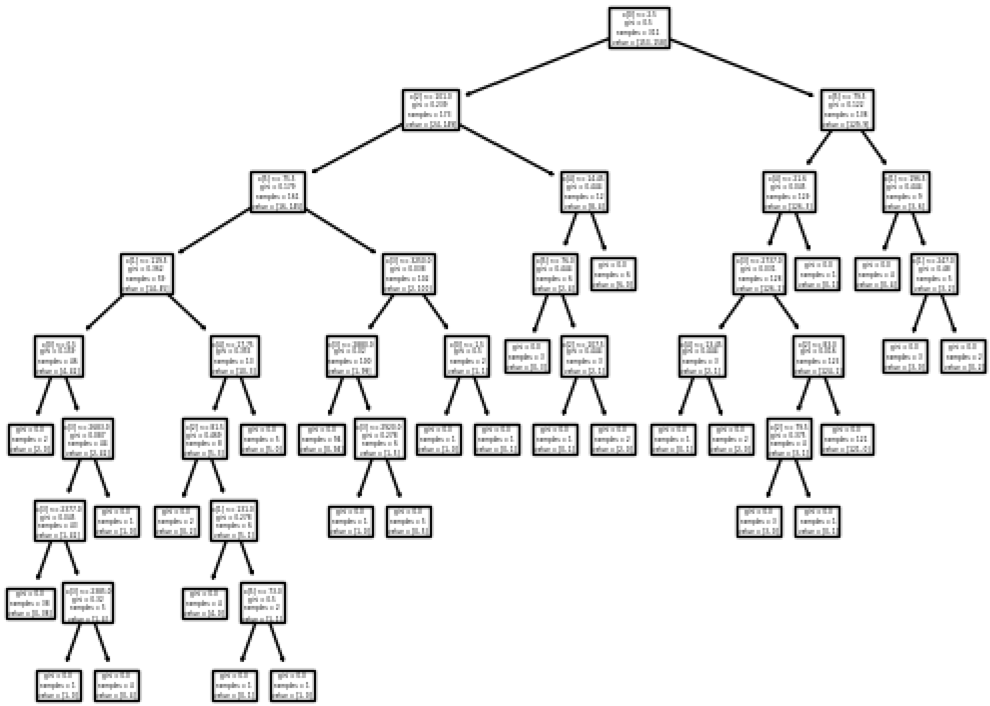
```
accuracy score:  0.9102564102564102
precision score:  0.8387096774193549
recall score:  0.9285714285714286
f1 score:  0.8813559322033899
              precision    recall  f1-score   support

           0       0.96      0.90      0.93        50
           1       0.84      0.93      0.88        28

    accuracy                           0.91        78
   macro avg       0.90      0.91      0.90        78
weighted avg       0.91      0.91      0.91        78
```

Out[90]:
```
[Text(0.6433823529411765, 0.9444444444444444, 'x[0] <= 2.5\ngini = 0.5\nsamples = 311
\nvalue = [153, 158]'),
 Text(0.4338235294117647, 0.8333333333333334, 'x[2] <= 101.0\ngini = 0.239\nsamples =
173\nvalue = [24, 149]'),
 Text(0.27941176470588236, 0.7222222222222222, 'x[5] <= 75.5\ngini = 0.179\nsamples =
161\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[0] <= 0.5\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\nsample
s = 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 =
[10, 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[1] <= 131.0\ngini = 0.278\nsamples
= 6\nvalue = [5, 1]'),
 Text(0.20588235294117646, 0.16666666666666666, 'gini = 0.0\nsamples = 4\nvalue = [4,
0]'),
 Text(0.2647058823529412, 0.16666666666666666, 'x[5] <= 73.0\ngini = 0.5\nsamples = 2
\nvalue = [1, 1]'),
 Text(0.23529411764705882, 0.05555555555555555, 'gini = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.29411764705882354, 0.05555555555555555, 'gini = 0.0\nsamples = 1\nvalue = [1,
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,
94]'),
 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[0] <= 1.5\ngini = 0.5\nsamples = 2\nvalue = [1,
1]'),
 Text(0.4411764705882353, 0.3888888888888889, 'gini = 0.0\nsamples = 1\nvalue = [1,
0]'),
```

```
 Text(0.5, 0.3888888888888889, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.5882352941176471, 0.7222222222222222, 'x[4] <= 14.45\ngini = 0.444\nsamples =
12\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[2] <= 107.5\ngini = 0.444\nsamples = 3\nvalue = [2,
1]'),
 Text(0.5588235294117647, 0.3888888888888889, 'gini = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.6176470588235294, 0.3888888888888889, 'gini = 0.0\nsamples = 2\nvalue = [2,
0]'),
 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 =
138\nvalue = [129, 9]'),
 Text(0.7941176470588235, 0.7222222222222222, 'x[4] <= 21.6\ngini = 0.045\nsamples =
129\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[4] <= 13.45\ngini = 0.444\nsamples = 3\nvalue = [2,
1]'),
 Text(0.6764705882352942, 0.3888888888888889, 'gini = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.7352941176470589, 0.3888888888888889, 'gini = 0.0\nsamples = 2\nvalue = [2,
0]'),
 Text(0.8235294117647058, 0.5, 'x[2] <= 83.0\ngini = 0.016\nsamples = 125\nvalue = [1
24, 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 = [12
1, 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]')]
```



In [91]:
```
# 10. Neural Network
# scaling the data
scaler = preprocessing.StandardScaler().fit(X_train)

X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# first neural network
mlp = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(5, 2), max_iter=1000, random_s
mlp.fit(X_train_scaled, y_train)
# test
pred = mlp.predict(X_test_scaled)
# evaluate
print('accuracy = ', accuracy_score(y_test, pred))

confusion_matrix(y_test, pred)
print(classification_report(y_test, pred))
```

```
accuracy =  0.8589743589743589
              precision    recall  f1-score   support

           0       0.93      0.84      0.88        50
           1       0.76      0.89      0.82        28

    accuracy                           0.86        78
   macro avg       0.85      0.87      0.85        78
weighted avg       0.87      0.86      0.86        78
```

In [92]:
```
# second neural network
mlp2 = MLPClassifier(solver='sgd', hidden_layer_sizes=(3,), max_iter=1500, random_stat
mlp2.fit(X_train_scaled, y_train)
# test
pred2 = mlp2.predict(X_test_scaled)
# evaluate
print('accuracy = ', accuracy_score(y_test, pred2))
```

```
confusion_matrix(y_test, pred2)
print(classification_report(y_test, pred2))
# comparison:
# The first neural network performed better, with a marginally better overall accuracy
# I think the performance was different due to the difference
# in the solver, max iterations, and the hidden layer mesh
```

```
accuracy =  0.8333333333333334
              precision    recall  f1-score   support

           0       0.93      0.80      0.86        50
           1       0.71      0.89      0.79        28

    accuracy                           0.83        78
   macro avg       0.82      0.85      0.83        78
weighted avg       0.85      0.83      0.84        78
```

11. Analysis

a. Which algorithm performed better?

The first algorithm was the best performing algorithm, with a higher overall accuracy score.

b. Compare accuracy, recall and precision metrics by class

Across both of the algorithms, lower than average mpg data predictions were more accurate and precise. On the other hand the recall scores were marginally higher for higher than average mpg data.

c. Give your analysis of why the better-performing algorithm might have outperformed the other

I think the first algorithm performed better promarily due to the difference in the hidden layer sizes. With there being more in the first algorithm, the accuracy as well as the efficiency was benefited. This is why, even with a lower number of iterations, the first algorithm yielded better results.

d. Write a couple of sentences comparing your experiences using R versus sklearn. Feel free to express strong preferences.

Personally, I vastly prefer sklearn over R. But I think this is because of the same reason that I prefer Python over Java or C. sklearn feels like a higher-level language to work in, compared to

R. That being said, I do see the need to have started in R, so that a grassroots level of learning for the concepts can be achieved.

In [94]:
```
%%shell
jupyter nbconvert --to html ///content/sklearnML.ipynb
```

```
[NbConvertApp] Converting notebook ///content/sklearnML.ipynb to html
[NbConvertApp] Writing 796705 bytes to /content/sklearnML.html
```

Out[94]: