ML with sklearn

Sanjana Jadhav
CS 4375.004

Sources Used: Textbook, Class Slides + other sources listed below

Read the CSV

- import the Pandas library
- use **pd.read_csv()** to read the **Auto.csv** file into a Pandas dataframe named **df**

df.head() function

- print the first few rows

In [ ]:
```
import pandas as pd

df = pd.read_csv('Auto.csv')

df.head()
```

Out[ ]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130	3504	12.0	70.0	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70.0	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70.0	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70.0	1	amc rebel sst
4	17.0	8	302.0	140	3449	NaN	70.0	1	ford torino

df.shape

- print the dimesnions of the dataframe

In [ ]:
```
print("Dimensions:", df.shape)
```

```
Dimensions: (392, 9)
```

Data Exploration

df.describe() function

- use **.loc** to subset the **mpg, weight, and year** columns
- call the **describe()** function to generate statistics for the columns such as mean, min, max, etc.

In [ ]:
```
df.loc[:, ['mpg', 'weight', 'year']].describe()
```

Out[ ]:

	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

Range

- find the range of the **mpg, weight, and year** columns by subtracting the min from the max
- the range of the **weight** column is the highest (3527) and the **year** column has the smallest range (12.0)

  I used this Youtube video to guide me (as I am not familiar with Python):

In [ ]:
```
print("range of mpg:", df['mpg'].max()-df['mpg'].min())
print("\nrange of weight:", df['weight'].max()-df['weight'].min())
print("\nrange of year:", df['year'].max()-df['year'].min())
```

```
range of mpg: 37.6

range of weight: 3527

range of year: 12.0
```

Mean

- find the mean of the **mpg, weight, and year** columns by calling the mean() function
- the mean of the **weight** column is the highest (2977.5841836734694) and the **year** column has the lowest mean (23.445918367346938)

I used this link to guide me (as I am not familiar with Python)

In [ ]:
```
print("mean of mpg:", df['mpg'].mean())
print("\nmean of weight:", df['weight'].mean())
print("\nmean of year:", df['year'].mean())
```

```
mean of mpg: 23.445918367346938

mean of weight: 2977.5841836734694

mean of year: 76.01025641025642
```

Explore Data Types

df.dtypes

- return the **data types** of all the columns in df

In [ ]:
```
df.dtypes
```

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

Convert to Categorical Variables

- **cylinders** is being converted to a categorical variable with **numeric codes** associated with each category
- **origin** is being converted to a categorical variable without converting the categories into numeric codes
- then, we call **df.dtypes** to see the updated changes

In [ ]:
```
df.cylinders = df.cylinders.astype('category').cat.codes
df.origin = df.origin.astype('category')
df.dtypes
```

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

Deal with NAs

df.isnull().sum() function

- returns the total number of missing values in each column

In [ ]:
```
df.isnull().sum()
```

Out[ ]:
```
mpg             0
cylinders       0
displacement    0
horsepower      0
weight          0
acceleration    1
year            2
origin          0
name            0
dtype: int64
```

df.dropna() function

- delete the rows containing missing values
- print the new dimensions

In [ ]:
```
df = df.dropna()
print('Dimensions of data frame after dropping NA:', df.shape)
```

```
Dimensions of data frame after dropping NA: (389, 9)
```

Modify Columns

- find the mean of the **mpg** column and assign it to **mpg_mean**
- create a new column named **mpg_high** and assign 1 if the value in **mpg** is greater than **mpg_mean**, else assign 0 (used a conditional if/else statement to implement this)
- use the **df.drop()** function to delete the **name** and **mpg** columns
- output the first few rows of the dataframe

I used this link to guide me (as I am not familiar with Python)

In [ ]:
```
mpg_mean = df['mpg'].mean()
df['mpg_high'] = [1 if mpg > mpg_mean else 0 for mpg in df.mpg]

df = df.drop(columns=['name','mpg'])

df.head()
```

```
<ipython-input-384-cc0a80903517>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy
  df['mpg_high'] = [1 if mpg > mpg_mean else 0 for mpg in df.mpg]
```

Out[ ]:

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

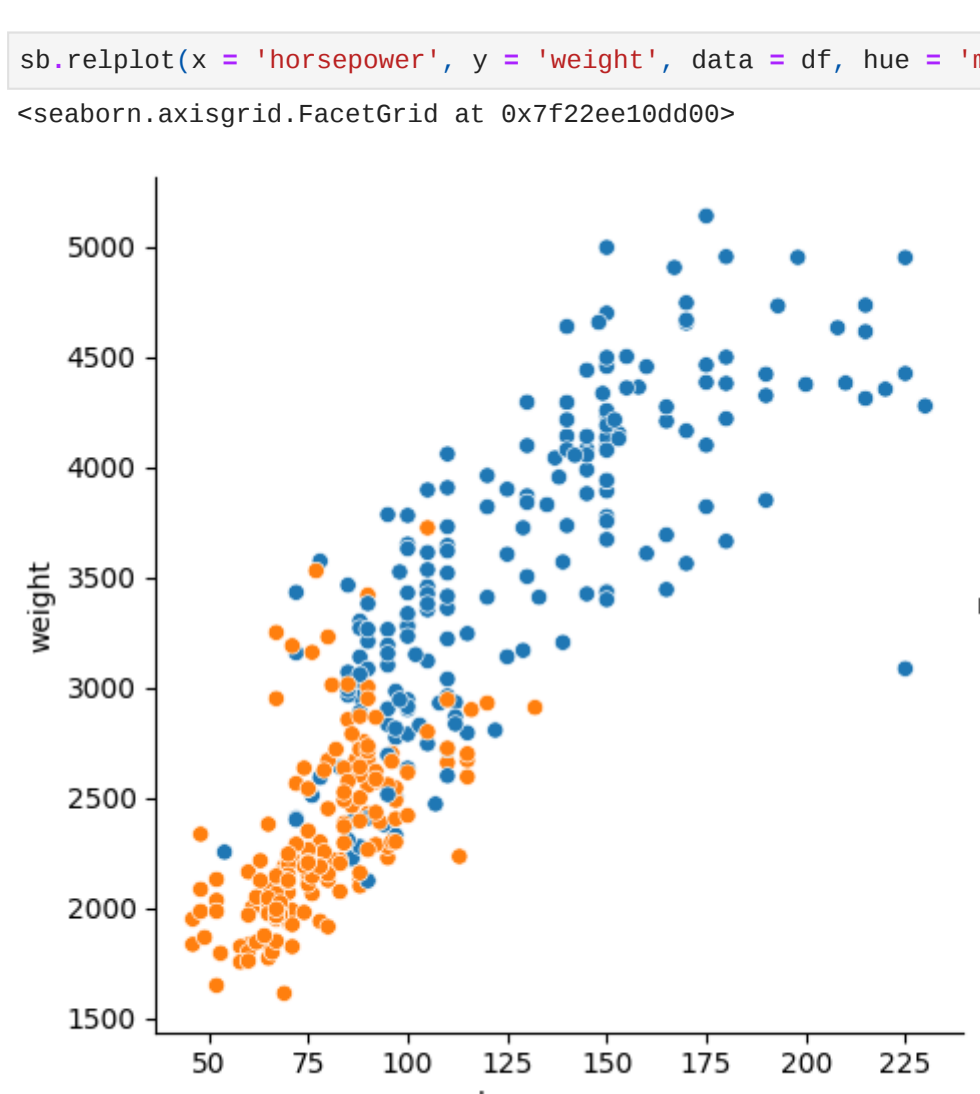
Data Exploration with Graphs

seaborn catplot()

- create a **catplot** (count plot) to tell us the number of observations in each category for **mpg_high** (0/1)
- in our dataset, there are more cars whose **mpg** is higher than hte average

In [ ]:
```
import seaborn as sb
sb.catplot(x="mpg_high", kind='count', data=df)
```

Out[ ]:
```
<seaborn.axisgrid.FacetGrid at 0x7f22ed8fa4c0>
```

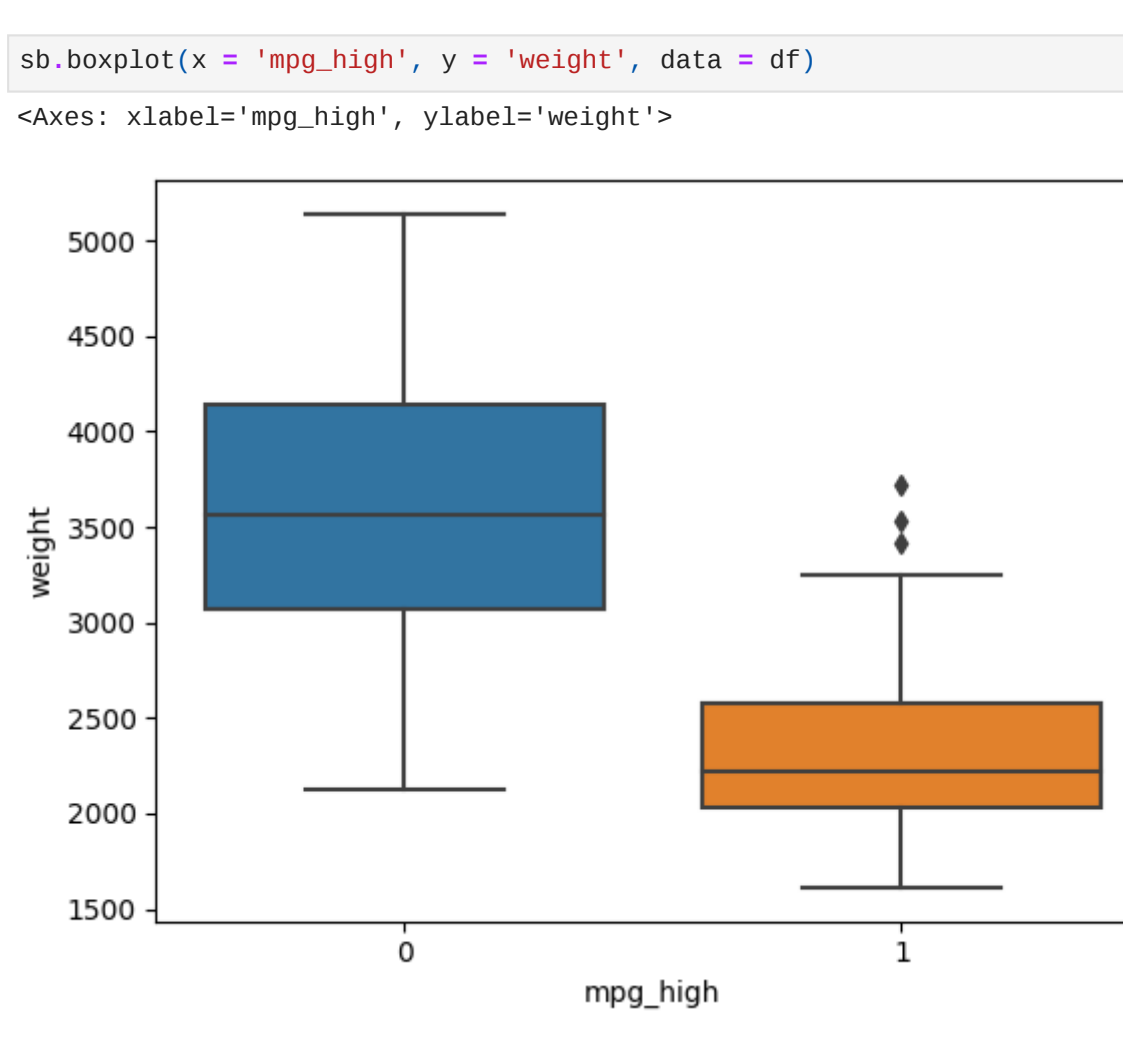


seaborn relplot()

- create a **relplot** (relationship plot) to model the relationship between horsepower and weight
- the **hue** parameter tells us "orange stands for 0" in the the **mpg_high** column and "blue stands for 1" in the the **mpg_high** column
- we can see how horsepower and weight affect **mpg_high**, telling us that low horsepower and weight results in higher **mpg**

In [ ]:
```
sb.relplot(x = 'horsepower', y = 'weight', data = df, hue = 'mpg_high')
```

Out[ ]:
```
<seaborn.axisgrid.FacetGrid at 0x7f22ee10dd00>
```

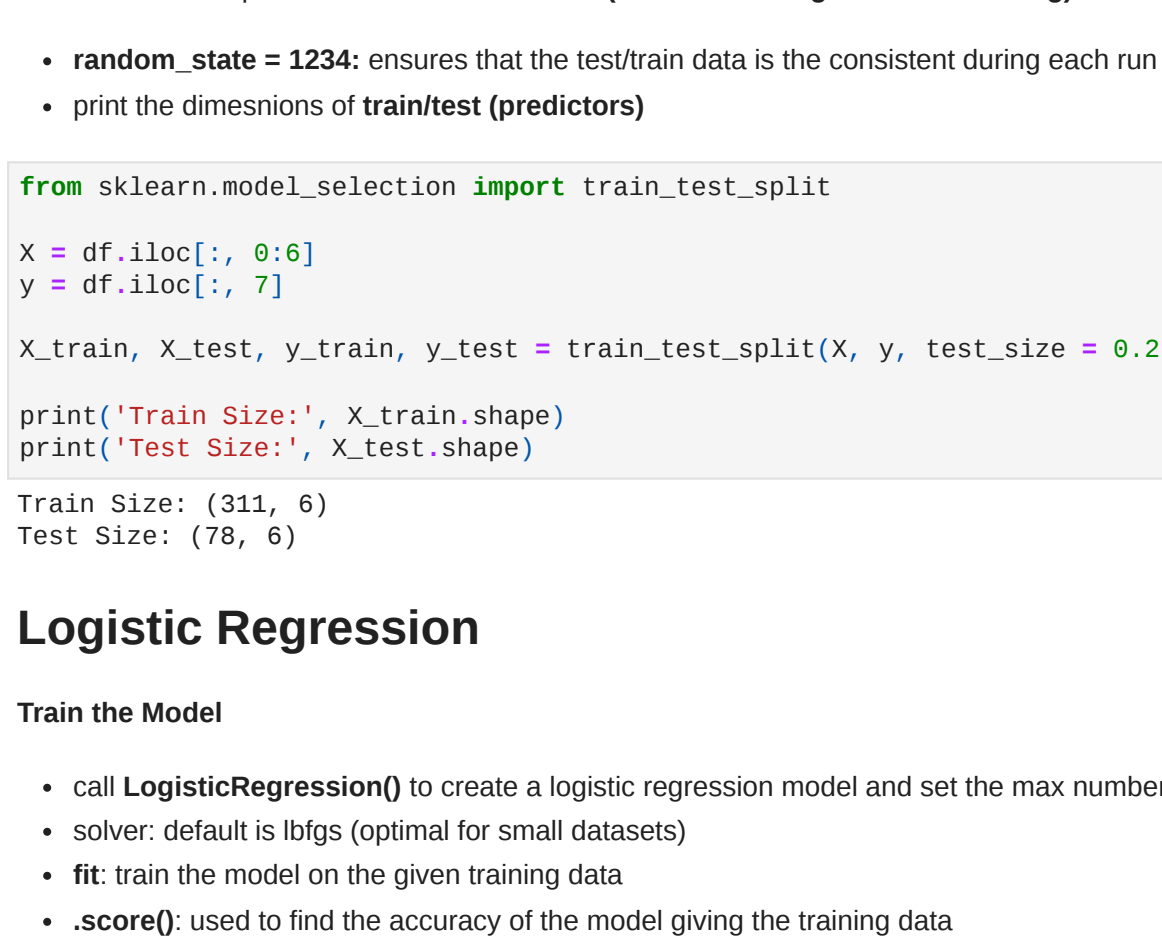


seaborn boxplot()

- create a **boxplot** to distribution of **weight** for the two values in the **mpg_high** column
- we can see that low weight results in higher **mpg**

In [ ]:
```
sb.boxplot(x = 'mpg_high', y = 'weight', data = df)
```

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



Train/Test Split

- X: contains all the columns that are predictors
- Y: contains the target (mpg_high)

- X and Y are split in train/test ***italicized text*****(80% for training & 20% for testing)**

- **random_state = 1234:** ensures that the test/train data is the consistent during each run
- print the dimesnions of **train/test (predictors)**

In [ ]:
```
from sklearn.model_selection import train_test_split

X = df.iloc[:, 0:6]
y = df.iloc[:, 7]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 1234)

print('Train Size:', X_train.shape)
print('Test Size:', X_test.shape)
```

```
Train Size: (311, 6)
Test Size: (78, 6)
```

Logistic Regression

Train the Model

- call **LogisticRegression()** to create a logistic regression model and set the max number of iterations to 1000
- solver: default is lbfgs (optimal for small datasets)
- **fit**: train the model on the given training data
- **.score()**: used to find the accuracy of the model giving the training data
  - **0.9035369774919614** (which is high)

I used this link to guide me (as I am not familiar with Python)

In [ ]:
```
from sklearn.linear_model import LogisticRegression

LR = LogisticRegression(max_iter=1000)
LR.fit(X_train, y_train)
LR.score(X_train, y_train)
```

Out[ ]:
```
0.9035369774919614
```

Make Predictions on Test Data

In [ ]:
```
pred1 = LR.predict(X_test)
```

Confusion Matrix

- The model correctly predicted 40 samples as positive (TP)
- The model incorrectly predicted 10 samples as negative that were actually positive (FN)
- The model correctly predicted 27 samples as negative (TN)
- The model incorrectly predicted 1 sample as positive that were actually negative (FP)

We want most of our values to be **TP** or **TN**, as these values were accurately predicted. This is the case in our confusion matrix.

In [ ]:
```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_test, pred1)
```

Out[ ]:
```
array([[40, 10],
       [ 1, 27]])
```

Classification Report

- **precision**: tells us the amount of true positives over all values that were predicted positive (0.98 for class 0 and 0.73 for class 1)
- **recall**: tells us the amount of true positives over all values that were actually positive (0.80 for class 0 and 0.96 for class 1)
- **f1 score**: mean of precision & higher values are preferred (0.88 for class 0 and 0.83 for class 1)
- **support**: number of samples (50 for class 0 and 28 for class 1)
- **accuracy**: percent of accurately predicted values for both classes (0.86)
- **macro avg**: mean of precision, recall, and f1 score of both classes (0.85, 0.88, 0.85 respectively)
- **weighted avg**: mean of precision, recall, and f1 score of both classes while factoring in the number of samples (0.89, 0.86, 0.86 respectively)

Used these links for more clarification!

1. link 1
2. link 2

In [ ]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred1))
```

```
              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
```

Decision Trees

Train the Model

- call **DecisionTreeClassifier()** to create a decision tree classifier model
- **fit**: train the model on the given training data

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier

DT = DecisionTreeClassifier()
DT.fit(X_train, y_train)
```

Out[ ]:
```
▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

Make Predictions on Test Data

In [ ]:
```
pred2 = DT.predict(X_test)
```

Confusion Matrix

- The model correctly predicted 43 samples as positive (TP)
- The model incorrectly predicted 7 samples as negative that were actually positive (FN)
- The model correctly predicted 27 samples as negative (TN)
- The model incorrectly predicted 1 sample as positive that were actually negative (FP)

We want most of our values to be **TP** or **TN**, as these values were accurately predicted. This is the case in our confusion matrix.

In [ ]:
```
confusion_matrix(y_test, pred2)
```

Out[ ]:
```
array([[46,  4],
       [ 2, 26]])
```

Classification Report

- **precision**: tells us the amount of true positives over all values that were predicted positive (0.98 for class 0 and 0.79 for class 1)
- **recall**: tells us the amount of true positives over all values that were actually positive (0.86 for class 0 and 0.96 for class 1)
- **f1 score**: mean of precision & higher values are preferred (0.91 for class 0 and 0.87 for class 1)
- **support**: number of samples (50 for class 0 and 28 for class 1)
- **accuracy**: percent of accurately predicted values for both classes (0.90)
- **macro avg**: mean of precision, recall, and f1 score of both classes (0.89, 0.81, 0.89 respectively)
- **weighted avg**: mean of precision, recall, and f1 score of both classes while factoring in the number of samples (0.91, 0.90, 0.90 respectively)

In [ ]:
```
print(classification_report(y_test, pred2))
```

```
              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
```

Neural Networks

Scale & Transform

- call **StandardScaler()** to fit the training data: compute mean and standard deviation
- scale the train/test data

In [ ]:
```
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 the 1st Model

- call **MLPClassifier()** to create a neural networks model and set the max number of iterations to 1000
- solver: lbfgs (optimal for small datasets)
- **fit**: train the model on the given training data

In [ ]:
```
from sklearn.neural_network import MLPClassifier

nn_1 = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(5, 2), max_iter=1000, random_state=1234)
nn_1.fit(X_train_scaled, y_train)
```

Out[ ]:
```
▾ MLPClassifier
MLPClassifier(hidden_layer_sizes=(5, 2), max_iter=1000, random_state=1234,
              solver='lbfgs')
```

Make Predictions on Test Data

In [ ]:
```
pred3 = nn_1.predict(X_test_scaled)
```

Confusion Matrix

- The model correctly predicted 44 samples as positive (TP)
- The model incorrectly predicted 6 samples as negative that were actually positive (FN)
- The model correctly predicted 25 samples as negative (TN)
- The model incorrectly predicted 3 sample as positive that were actually negative (FP)

We want most of our values to be **TP** or **TN**, as these values were accurately predicted. This is the case in our confusion matrix.

In [ ]:
```
confusion_matrix(y_test, pred3)
```

Out[ ]:
```
array([[44,  6],
       [ 3, 25]])
```

Classification Report

- **precision**: tells us the amount of true positives over all values that were predicted positive (0.94 for class 0 and 0.81 for class 1)
- **recall**: tells us the amount of true positives over all values that were actually positive (0.88 for class 0 and 0.89 for class 1)
- **f1 score**: mean of precision & higher values are preferred (0.91 for class 0 and 0.85 for class 1)
- **support**: number of samples (50 for class 0 and 28 for class 1)
- **accuracy**: percent of accurately predicted values for both classes (0.88)
- **macro avg**: mean of precision, recall, and f1 score of both classes (0.87, 0.89, 0.88 respectively)
- **weighted avg**: mean of precision, recall, and f1 score of both classes while factoring in the number of samples (0.89, 0.88, 0.89 respectively)

In [ ]:
```
print(classification_report(y_test, pred3))
```

```
              precision    recall  f1-score   support

           0       0.94      0.88      0.91        50
           1       0.81      0.89      0.85        28

    accuracy                           0.88        78
   macro avg       0.87      0.89      0.88        78
weighted avg       0.89      0.88      0.89        78
```

Train the 2nd Model

- call **MLPClassifier()** to create a neural networks model and set the max number of iterations to 1500
- solver: sgd (optimal for large datasets)
- **fit**: train the model on the given training data

In [ ]:
```
nn_2 = MLPClassifier(solver='sgd', hidden_layer_sizes=(6,2), max_iter=1500, random_state=1234)
nn_2.fit(X_train_scaled, y_train)
```

Out[ ]:
```
▾ MLPClassifier
MLPClassifier(hidden_layer_sizes=(6, 2), max_iter=1500, random_state=1234,
              solver='sgd')
```

Make Predictions on Test Data

In [ ]:
```
pred4 = nn_2.predict(X_test_scaled)
```

Confusion Matrix

- The model correctly predicted 42 samples as positive (TP)
- The model incorrectly predicted 8 samples as negative that were actually positive (FN)
- The model correctly predicted 28 samples as negative (TN)
- The model incorrectly predicted 0 sample as positive that were actually negative (FP)

We want most of our values to be **TP** or **TN**, as these values were accurately predicted. This is the case in our confusion matrix.

In [ ]:
```
confusion_matrix(y_test, pred4)
```

Out[ ]:
```
array([[42,  8],
       [ 0, 28]])
```

Classification Report

- **precision**: tells us the amount of true positives over all values that were predicted positive (1.00 for class 0 and 0.78 for class 1)
- **recall**: tells us the amount of true positives over all values that were actually positive (0.84 for class 0 and 1.00 for class 1)
- **f1 score**: mean of precision & higher values are preferred (0.91 for class 0 and 0.88 for class 1)
- **support**: number of samples (50 for class 0 and 28 for class 1)
- **accuracy**: percent of accurately predicted values for both classes (0.90)
- **macro avg**: mean of precision, recall, and f1 score of both classes (0.89, 0.91, 0.89 respectively)
- **weighted avg**: mean of precision, recall, and f1 score of both classes while factoring in the number of samples (0.92, 0.90, 0.90 respectively)

In [ ]:
```
print(classification_report(y_test, pred4))
```

```
              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
```

The second model, **nn_2**, performed better than the first model as it used the **"sgd"** solver. Moreover, the number of hidden layers was increased to **(6, 2)** and the **max_iter** was also increased to 1500, allowing the algorithm to perform more iterations. As a result, the **true positive (TP) and true negative (TN) values** in the confusion matrix increased, indicating better performance in correctly classifying both positive and negative instances.

Analysis

In [ ]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

print('Accuracy Score for LR: ', accuracy_score(y_test, pred1))
print('Precision Score for LR: ', precision_score(y_test, pred1))
print('Recall Score for LR: ', recall_score(y_test, pred1))
print('f1 Score for LR: ', f1_score(y_test, pred1))
```

```
Accuracy Score for LR:  0.8589743589743589
Precision Score for LR:  0.7297297297297297
Recall Score for LR:  0.9642857142857143
f1 Score for LR:  0.8307692307692307
```

In [ ]:
```
print('Accuracy Score for DT: ', accuracy_score(y_test, pred2))
print('Precision Score for DT: ', precision_score(y_test, pred2))
print('Recall Score for DT: ', recall_score(y_test, pred2))
print('f1 Score for DT: ', f1_score(y_test, pred2))
```

```
Accuracy Score for DT:  0.9230769230769231
Precision Score for DT:  0.8666666666666667
Recall Score for DT:  0.9285714285714286
f1 Score for DT:  0.896551724137931
```

In [ ]:
```
print('Accuracy Score for the 1st NN: ', accuracy_score(y_test, pred3))
print('Precision Score for the  1st NN: ', precision_score(y_test, pred3))
print('Recall Score for the 1st NN: ', recall_score(y_test, pred3))
print('f1 Score for the 1st NN: ', f1_score(y_test, pred3))
```

```
Accuracy Score for the 1st NN:  0.8846153846153846
Precision Score for the  1st NN:  0.8064516129032258
Recall Score for the 1st NN:  0.8928571428571429
f1 Score for the 1st NN:  0.8474576271186439
```

In [ ]:
```
print('Accuracy Score for the 2nd NN: ', accuracy_score(y_test, pred4))
print('Precision Score for the 2nd NN: ', precision_score(y_test, pred4))
print('Recall Score for the 2nd NN: ', recall_score(y_test, pred4))
print('f1 Score for the 2nd NN: ', f1_score(y_test, pred4))
```

```
Accuracy Score for the 2nd NN:  0.8974358974358975
Precision Score for the 2nd NN:  0.7777777777777778
Recall Score for the 2nd NN:  1.0
f1 Score for the 2nd NN:  0.8750000000000001
```

Analyzing the Algorithms

Using **f1 score** (mean)

- Decision Trees can outperform Logistic Regression by capturing non-linear relationships between features and the target. They recursively split data based on features, capturing complex interactions. Decision Trees are also more robust to outliers.
- Logistic Regression models assume a linear relationship between input and output variables, while Neural Networks can capture more complex relationships due to their ability to learn through multiple hidden layers. This is why the 2nd model performed better, as it had more hidden layers and a higher amount of iterations.

The Decision Tree Model performed best for our data!

R vs. sklearn

Initially, I found coding in Python a bit challenging as it was my first time. However, I took the initiative to research and utilize other resources to guide me through the assignment. Although I am more accustomed to coding in R, which has excellent built-in libraries, I appreciate the faster execution speed of Python. I am excited to explore and become proficient in Python!