Portfolio 5: ML with SKLearn

Bushra Rahman

Using SKLearn on Auto.csv

Auto.csv is a small dataset that describes characteristics about vehicles. The dataset contains 9 features for 392 observations.

Read the Auto Data

In [1]:
```
import pathlib
import pandas as pd

path_string = pathlib.Path.cwd().joinpath('Auto.csv')

# Use pandas to read the data
df = pd.read_csv(path_string, header=0, encoding='latin-1')
# Output the dimensions of the data
print('Dimensions of df:', df.shape)
# Output the first few rows
print(df.head())
```

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

Data Exploration

In [2]:
```
# Use describe() on the mpg, weight, and year columns
print('Describe mpg, weight, and year:\n',
df.loc[:, ['mpg', 'weight', 'year']].describe())

# Extract range and average of each column
mpg_mean = df.describe()['mpg']['mean']
print('\nMean of mpg:', mpg_mean)
mpg_range = float(df.describe()['mpg']['max']) - float(df.describe()['mpg']['min'])
print('Range of mpg:', mpg_range)
# Mean of mpg = 23.490488
# Range of mpg = 37.6
```

```
weight_mean = df.describe()['weight']['mean']
print('\nMean of weight:', weight_mean)
weight_range = float(df.describe()['weight']['max']) - float(df.describe()['weight']['
print('Range of weight:', weight_range)
# Mean of weight = 2973.871465
# Range of weight = 3527.0

year_mean = df.describe()['year']['mean']
print('\nMean of year:', year_mean)
year_range = float(df.describe()['year']['max']) - float(df.describe()['year']['min'])
print('Range of year:', year_range)
# Mean of year = 76.025707
# Range of year = 12.0
```

```
Describe mpg, weight, and year:
              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

Mean of mpg: 23.44591836734694
Range of mpg: 37.6

Mean of weight: 2977.5841836734694
Range of weight: 3527.0

Mean of year: 76.01025641025642
Range of year: 12.0
```

Explore Data Types

In [3]:
```
# Check the data types of all columns
df.dtypes
```

Out[3]:
```
mpg             float64
cylinders         int64
displacement    float64
horsepower        int64
weight            int64
acceleration    float64
year            float64
origin            int64
name             object
dtype: object
```

In [4]:
```
# Change the cylinders column to categorical with numeric factor codes
df.cylinders = df.cylinders.astype('category').cat.codes
print(df.cylinders.head())  # cat.codes makes dtype =int8 instead of =category
```

```
0    4
1    4
2    4
3    4
4    4
Name: cylinders, dtype: int8
```

In [5]:

```
# Change the origin column to categorical (don’t use cat.codes)
df.origin = df.origin.astype('category')
print(df.origin.head())
```

```
0    1
1    1
2    1
3    1
4    1
Name: origin, dtype: category
Categories (3, int64): [1, 2, 3]
```

In [6]:

```
# Verify the changes with the dtypes attribute
print(df.dtypes)
```

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

Deal With NAs

In [7]:

```
# Delete rows with NAs
df = df.dropna()
# Output the new dimensions
print('Dimensions of df after removing NAs:', df.shape)
```

```
Dimensions of df after removing NAs: (389, 9)
```

Modify Columns

In [8]:

```
mpg_high_list = []  # Create new list for mpg_high column

# column = 1 if mpg > average mpg, else == 0
for mpg_value in df['mpg']:
    if mpg_value > mpg_mean:
        mpg_high_list.append(1)
    else:
        mpg_high_list.append(0)

# Use .insert() to append new column mpg_high as the 2nd col (index=1) in df
df.insert(1, 'mpg_high', mpg_high_list)

# Make mpg_high a categorical column
df.mpg_high = df.mpg_high.astype('category')
```

In [9]:
```
# Delete the mpg and name columns
del df['mpg']
del df['name']

# Output the first few rows of the modified data frame
print('Dimensions of df after column modifications:', df.shape)
print(df.head())
df.dtypes
```

```
Dimensions of df after column modifications: (389, 8)
  mpg_high  cylinders  displacement  horsepower  weight  acceleration  year  \
0        0          4         307.0         130    3504          12.0  70.0
1        0          4         350.0         165    3693          11.5  70.0
2        0          4         318.0         150    3436          11.0  70.0
3        0          4         304.0         150    3433          12.0  70.0
6        0          4         454.0         220    4354           9.0  70.0

  origin
0      1
1      1
2      1
3      1
6      1
```

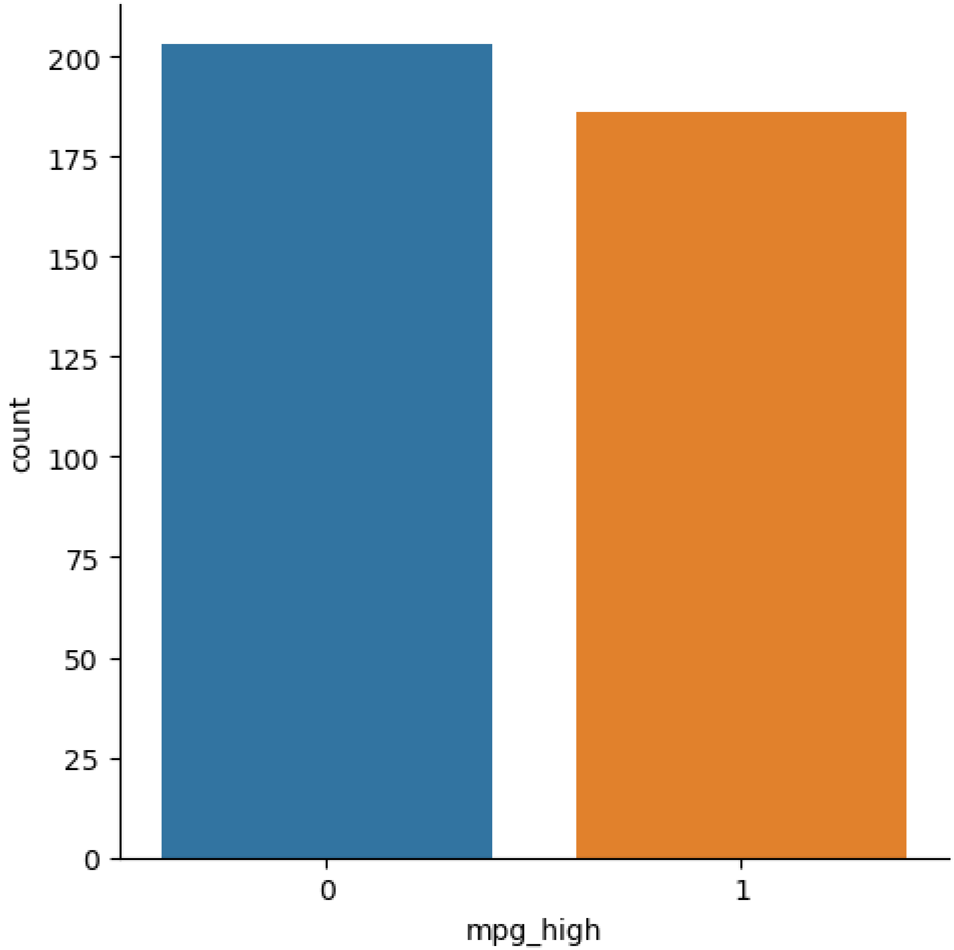
Out[9]:
```
mpg_high        category
cylinders           int8
displacement     float64
horsepower         int64
weight             int64
acceleration     float64
year             float64
origin          category
dtype: object
```

Data Exploration With Graphs

The following are 3 graphs that depict various features of the data against each other. The first graph is a **categorical plot** showing the frequencies of the two levels of mpg_high. The two levels have almost the same height, but level 0 (which corresponds to an mpg value less than the mpg mean) is slightly higher. This indicates that the distribution of high and low mpg values is almost entirely even, but slightly skewed towards lower mpg values.

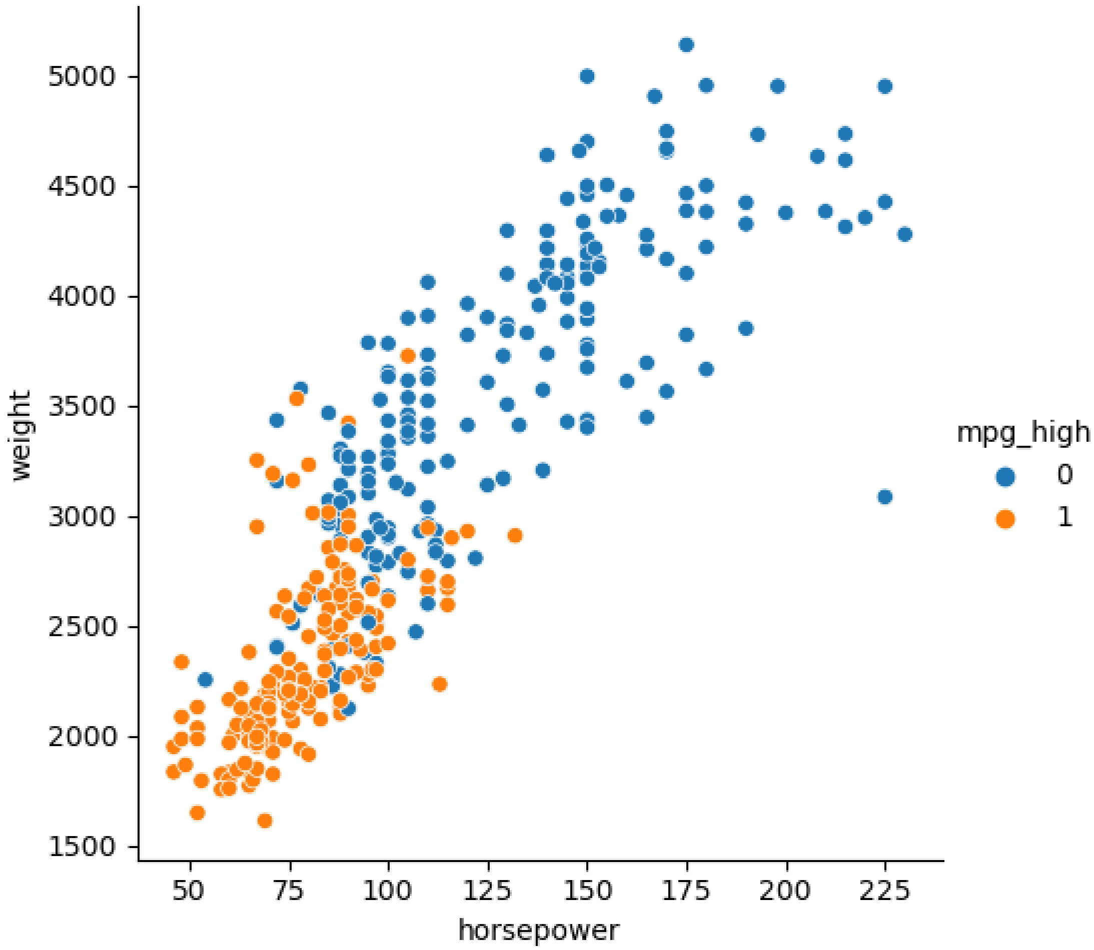
In [10]:
```
import seaborn as sb
import matplotlib.pyplot as plt  # Plots created using seaborn need to be displayed li

# Seaborn catplot on the mpg_high column
sb.catplot(x='mpg_high', kind='count', data=df)
plt.show()
```



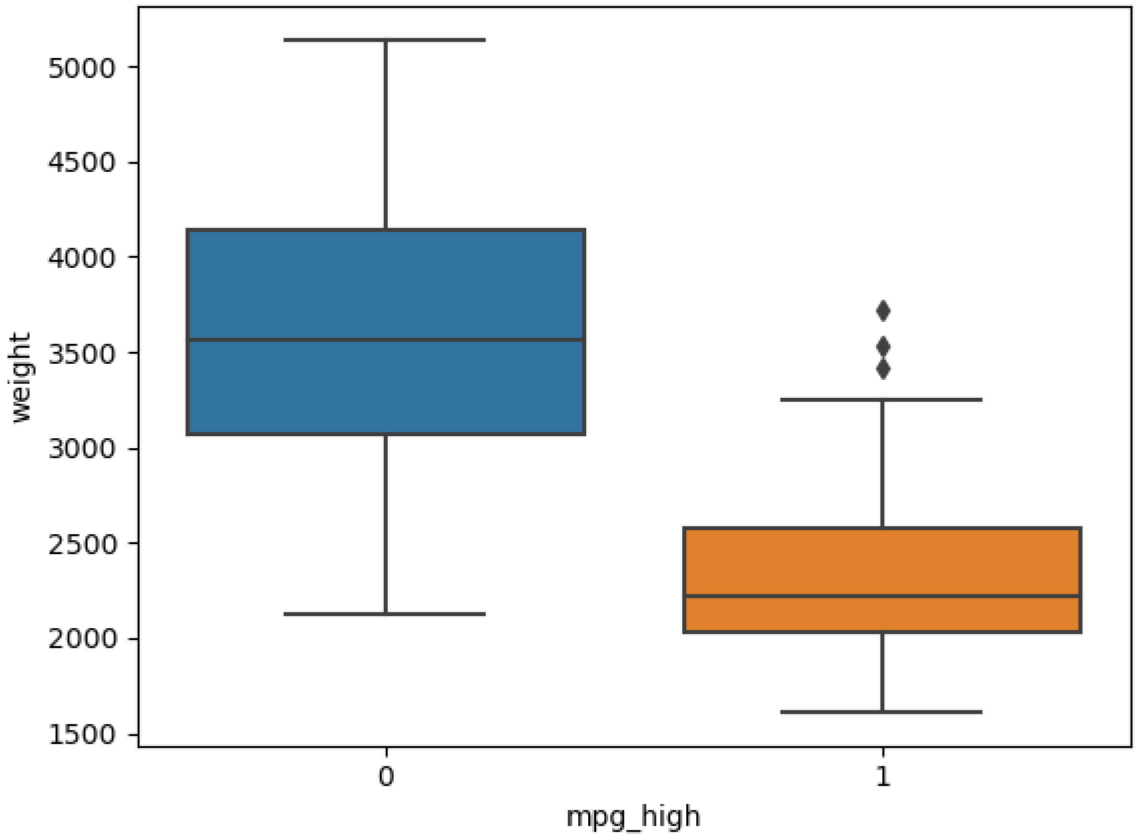
The second graph is a **relational plot** of horsepower against weight, with blue dots representing 0 for mpg_high and orange dots representing 1 for mpg_high. This graph shows a roughly linear relationship between horsepower and weight, with low horsepower corresponding to low weight and high horsepower corresponding to high weight. Using our domain knowledge, we know that horsepower measures the power of the engine output, so it makes sense that a heavy vehicle would have more horsepower and vice versa. Additionally, the different-colored points show us that there appears to be a roughly linear boundary between low mpg and high mpg on this graph: heavy vehicles with high horsepower tend to have an mpg_high value of 0, while lighter vehicles with low horsepower tend to have an mpg_high value of 1. Again, using our domain knowledge, we know that mpg indicates speed, so it makes sense that heavy vehicles are slower than lighter vehicles.

In [11]:
```
# Seaborn relplot with horsepower on the x axis, weight on the y axis, setting hue or
sb.relplot(x='horsepower', y='weight', data=df, hue=df.mpg_high)
plt.show()
```



The third graph is a **boxplot** of mpg_high against weight. This boxplot shows that the "box" (the 2nd and 3rd quartiles) for low-mpg observations corresponds to higher weights (between about 3500-4000), while the box for high-mpg observations corresponds to lower weights (between about 2000-2500). This matches what was learned from the graph above. Additionally, the long "whiskers" (the 1st and 4th quartiles) on mpg_high=0 indicate a wide range of observations with varying weights, which matches how sparsely distributed the blue dots were in the relplot above. Conversely, the whiskers on mpg_high=1 appear to be shorter, thus matching how densely-packed the orange dots were in the relplot above. Finally, a couple outliers can be noticed beyond the top whisker of mpg_high=1, which probably corresponds to a few orange dots in the relplot above that are located quite far out from the densely packed locale where most of the orange dots reside.

In [12]:
```
# Seaborn boxplot with mpg_high on the x axis and weight on the y axis
sb.boxplot(x='mpg_high', y='weight', data=df)
plt.show()
```



Train/Test Split

In [13]:
```
from sklearn.model_selection import train_test_split

# Set up X and y (X consists of all columns except mpg_high, which is the 1st col)
X = df.iloc[:, 1:7]  # 7 predictors
y = df.mpg_high      # 1 target

# Divide into train/test sets on an 80/20 split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.

# Output the dimensions of train and test
print('Train size:', X_train.shape)
print('Test size:', X_test.shape)
```

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

Logistic Regression

In [14]:
```
from sklearn.linear_model import LogisticRegression

# Train logistic regression model using solver lbfgs
classifier = LogisticRegression(solver='lbfgs', class_weight='balanced')
classifier.fit(X_train, y_train)
```

Out[14]:
```
LogisticRegression(class_weight='balanced')
```

In [15]:
```
# Predict on the test data
pred = classifier.predict(X_test)
```

In [16]:
```
from sklearn.metrics import confusion_matrix

# Print confusion matrix
print(confusion_matrix(y_test, pred))
# Form of confusion matrix:
# [[TP   FP
#    FN   TN]]
```

```
[[40 10]
 [ 1 27]]
```

In [17]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import classification_report

# Evaluation metrics for LogReg
print(classification_report(y_test, pred))
print('Precision score: ', precision_score(y_test, pred))
print('Recall score: ', recall_score(y_test, pred))
print('Accuracy score: ', accuracy_score(y_test, pred))
print('F1 score: ', f1_score(y_test, pred))
```

```
              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

Precision score:  0.7297297297297297
Recall score:  0.9642857142857143
Accuracy score:  0.8589743589743589
F1 score:  0.8307692307692307
```

Decision Tree

In [18]:
```
from sklearn.tree import DecisionTreeClassifier

# Train decision tree model
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
```

Out[18]:
```
DecisionTreeClassifier()
```

In [19]:
```
# Predict on the test data
pred = clf.predict(X_test)
```

In [20]:
```
# Print confusion matrix
confusion_matrix(y_test, pred)
```

Out[20]:
```
array([[46,  4],
       [ 2, 26]], dtype=int64)
```

In [21]:
```
# Evaluation metrics for DT
print(classification_report(y_test, pred))
print('Precision score: ', precision_score(y_test, pred))
```

```
print('Recall score: ', recall_score(y_test, pred))
print('Accuracy score: ', accuracy_score(y_test, pred))
print('F1 score: ', f1_score(y_test, pred))
```

```
              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

Precision score:  0.8666666666666667
Recall score:  0.9285714285714286
Accuracy score:  0.9230769230769231
F1 score:  0.896551724137931
```

At a glance, these metrics for the DT model are better overall than the metrics for the logistic regression model. Precision, accuracy, and F1 are all better for DT, although recall is the same.

Neural Networks

In [22]:
```
# First normalize the data b/c NN performs better on scaled 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)
```

Criteria for the number of hidden nodes:\ • between 1 and the number of predictors: **b/w 1 and 7**\ • two-thirds of the input layer size plus the size of the output layer: **(2/3)(7+1) = 5**\ • <twice the input layer size: **<(7)(2) = <14**

Based on these criteria, **5 hidden nodes** seems like an appropriate choice, as it is between 1 and 7, and less than 14.

How many layers should these 5 hidden nodes be spread on? Multiple layers can capture more complex relationships, but may overfit on data of a small size. First we will try using **2 layers**. The chosen topology is **(3,2)**, ie. 3 nodes in the first layer and 2 in the second layer.

In [23]:
```
from sklearn.neural_network import MLPClassifier

# Train a neural network (network topology = 3,2)
clf = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(3, 2), max_iter=500, random_st
clf.fit(X_train_scaled, y_train)
```

Out[23]:
```
MLPClassifier(hidden_layer_sizes=(3, 2), max_iter=500, random_state=1234,
              solver='lbfgs')
```

In [24]:
```
# Predict on the test data
pred = clf.predict(X_test_scaled)
```

In [25]:
```
# Print confusion matrix
confusion_matrix(y_test, pred)
```

Out[25]:
```
array([[43,  7],
       [ 2, 26]], dtype=int64)
```

In [26]:
```
# Evaluation metrics for NN (1st try)
print(classification_report(y_test, pred))
print('Precision score: ', precision_score(y_test, pred))
print('Recall score: ', recall_score(y_test, pred))
print('Accuracy score: ', accuracy_score(y_test, pred))
print('F1 score: ', f1_score(y_test, pred))
```

```
              precision    recall  f1-score   support

           0       0.96      0.86      0.91        50
           1       0.79      0.93      0.85        28

    accuracy                           0.88        78
   macro avg       0.87      0.89      0.88        78
weighted avg       0.90      0.88      0.89        78

Precision score:  0.7878787878787878
Recall score:  0.9285714285714286
Accuracy score:  0.8846153846153846
F1 score:  0.8524590163934426
```

These scores are all less than the scores for the DT model, which is concerning. Next, we will try using **1 layer** with more hidden nodes. **7 hidden nodes** are chosen since 7 is the number of predictors.

In [27]:
```
# Train a neural network (network topology = 7)
clf = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(7), max_iter=500, random_state
clf.fit(X_train_scaled, y_train)

# Predict on the test data
pred = clf.predict(X_test_scaled)

# Print confusion matrix
confusion_matrix(y_test, pred)
```

Out[27]:
```
array([[45,  5],
       [ 1, 27]], dtype=int64)
```

In [28]:
```
# Evaluation metrics for NN (2nd try)
print(classification_report(y_test, pred))
print('Precision score: ', precision_score(y_test, pred))
print('Recall score: ', recall_score(y_test, pred))
print('Accuracy score: ', accuracy_score(y_test, pred))
print('F1 score: ', f1_score(y_test, pred))
```

```
              precision    recall  f1-score   support

           0       0.98      0.90      0.94        50
           1       0.84      0.96      0.90        28

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

Precision score:  0.84375
Recall score:  0.9642857142857143
Accuracy score:  0.9230769230769231
F1 score:  0.8999999999999999
```

These metrics show significant improvement from those of the DT model and the previous NN model. All of the scores for this NN model are higher than the 1st NN model. Although the precision of this model is less than the the DT model, and its accuracy is the same as the DT model, its recall and F1 are better than the DT model.

Analysis

The models with the highest accuracy are the DT model and the 2nd NN model, which came to the same accuracy score. Based on the confusion matrices, the DT model performed best overall with 46 true positives and 26 true negatives. Comparatively, the logistic regression model and the 2nd NN model both had 27 true negatives, 1 more than the DT model, but both had less true positives. However, the 2nd NN model had only 1 less true positive than the DT model, making it come quite close in terms of accuracy. The DT model also had the least false positives out of all the models.

In regards to accuracy, recall, and precision, the DT model and 2nd NN model had the best overall scores. Both models had the exact same accuracy score of 0.92. However, neither model totally outperforms the other in terms of precision and recall combined: the DT model has a higher precision, while the 2nd NN model has a higher recall. However, both models' precision and recall scores are within the same range — 0.86 and 0.92 for the DT model, and 0.84 and 0.96 for the 2nd NN model — that neither one ultimately outperforms the other.

The issue with the 1st NN model was that it appeared to be overfitting the data, since its metrics did not indicate that it performed well. So, the 2nd NN model was adjusted to have 1 less layer, while also having more nodes. The result was that the 2nd NN model performed better than the 1st one, but that it still did not do as well as the DT model. According to [1], decision trees are deterministic as opposed to being probabilistic, making them generally better at modeling rules-based scenarios rather than probabilistic scenarios. Neural networks, by comparison, are better-suited for the latter. The better performance of the DT model as opposed to the NN models in this particular scenario may be because Auto.csv is a structured dataset that does not model probabilistic nuances, thus making decision trees a more applicable algorithm.

R and Python's SKLearn library are both very powerful tools for modeling complex machine learning algorithms. Personally, I find Python more accessible to learn and easier to grasp, so my instinctive inclination is towards SKLearn. However, I am also partial to R's many built-in tools that can easily handle complex functionalities in a few simple lines of code. SKLearn *is* also quite clean and simple, since Python itself tends to produce clean and simple code. But R having so much functionality built into it gives programmers the advantage of having to handle less of the programming burden themselves. Ultimately I find both to be equally usable and powerful tools.

Sources

[1] https://towardsdatascience.com/when-and-why-tree-based-models-often-outperform-neural-networks-ceba9ecd0fd8