Lessons In Python: Machine Learning

Gray Simpson

In this assignment we use data about cars for classification as to whether it has good gas mileage. We will be performing assorted data reading and learning as we have previously done in R. However, now we will add neural networks and practice it here in Python.

This is my first time using Python to any significant extent, as well, so it doubles as a good way to ensure knowledge and really become familiar.

```
import numpy as np
import pandas as pd
import seaborn as sb
import sklearn as sk
```

Okay, with things imported, we can move on to data exploration and cleaning. As we go, we may have to import more libraries or even specific functions, but for now this should do just fine.

1: Reading the Data

Now, let's genuinely read the data we have to work with here, and start trying to understand it a bit more.

```
df = pd.read_csv('drive/MyDrive/data/Auto.csv')
print(df.head())
print('\nDimensions of df: ', df.shape)
```

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

Okay, things are looking good. Lets move on to more extensive data exploration.

2: Data Exploration

We'll pull up the details of all the different attributes so that we can see how they lie, common averages, range, and more.

```
df.describe(percentiles=None)
```

	mpg	cylinders	displacement	horsepower	weight	acceleration
count	392.000000	392.000000	392.000000	392.000000	392.000000	391.000000
mean	23.445918	5.471939	194.411990	104.469388	2977.584184	15.554220
std	7.805007	1.705783	104.644004	38.491160	849.402560	2.750548
min	9.000000	3.000000	68.000000	46.000000	1613.000000	8.000000
25%	17.000000	4.000000	105.000000	75.000000	2225.250000	13.800000
50%	22.750000	4.000000	151.000000	93.500000	2803.500000	15.500000

Let's write out the exact ranges we can see from the given max and min values. We'll focus on MPG, weight of the car, and the year.

MPG: avg: 23.446 range: 37.6 (9 to 46.6)

Weight: avg: 2977.584 range: 3527 (1613 to 5140)

Year: avg: 76.01 range: 12 (70 to 82)

We can also see that 'origin' and 'cylinders' might be better suited as categories. They are all stagnant integers. Going back up to double check from when we looked at the head of the dataframe, we see that it is true, and they are all specific, smaller whole numbers without a ton of variation, meaning that these may have been intended as codes used to represent categories.

```
print(df.cylinders.unique())
print(df.origin.unique())
```

```
[8 4 6 3 5]
[1 3 2]
```

Yes, with that it seems fairly clear that we are dealing with categorical values. We'll move on to convert them into something cleaner and easier to deal with, in that case.

3: Explore Data Types

Let's see what sorts of data we have, and fix the aspects we noticed should be categorical above into actual categorical-type values, whether it be smaller intengers, or specific category values.

```
df.dtypes
df.cylinders = df.cylinders.astype('category').cat.codes
df.origin = df.origin.astype('category')
```

Now let's see what data types we have now:

```
df.dtypes
```

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

4: Deal With NAs

Now that we finished that, lets handle a bit more cleaning of the data. Lets see if we have any null or nonavailable values we'll need to sweep out.

```
df.isnull().sum()
```

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

Okay, so we can see there is a handful of NAs we'll need to remove. Just in two attributes, though, so it should be quick and easy. Let's do this.

```
df = df.dropna(axis=0,how='any')
df.isnull().sum()
```

```
mpg             0
cylinders       0
displacement    0
horsepower      0
weight          0
acceleration    0
year            0
origin          0
name            0
dtype: int64
```

5: Modify Columns

While we've looked at plenty of interesting data, we still don't have anything that will let us predict a classification well. We could look at the origin, or the number of cylinders, but it would be more helpful for a user to be able to have a method to determine whether the car is considered to have good gas mileage or poor gas mileage. We'll fit this into a rating as to whether or not it is above average or below average.

```
mpg_high = ['0'] * df.shape[0]
i=0

for mpg in df['mpg']:
  if mpg > df['mpg'].mean():
    mpg_high[i] = 1
  else:
    mpg_high[i] = 0
  i+=1


df['mpg_high'] = mpg_high
print(df.head())
df['mpg_high'].describe(percentiles=None)
```

```
         mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
```

```
0  18.0          4         307.0         130    3504          12.0  70.0
1  15.0          4         350.0         165    3693          11.5  70.0
2  18.0          4         318.0         150    3436          11.0  70.0
3  16.0          4         304.0         150    3433          12.0  70.0
6  14.0          4         454.0         220    4354           9.0  70.0

  origin                       name  mpg_high
0      1  chevrolet chevelle malibu         0
1      1          buick skylark 320         0
2      1         plymouth satellite         0
3      1              amc rebel sst         0
6      1            chevrolet impala         0
count    389.000000
mean       0.478149
std        0.500166
min        0.000000
25%        0.000000
50%        0.000000
75%        1.000000
max        1.000000
Name: mpg_high, dtype: float64
```

Okay, so we can see that our data modification worked out well. MPG high seems to be working just fine. Let's move on to delete the initial MPG and name columns (so they don't interfere with our predictions and make it too accurate) and look at the gist of the dataframe again.

```
df = df.drop(columns=['mpg','name'])
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 Explorations With Graphs (seaborn)

Now that we've looked through the data and cleaned it up, let's take a closer look at what we can understand in the data that will help us realize what would be most helpful in creating a system to help us predict the highs of the gas mileage. Let's look at some graphs and see what it teaches us!

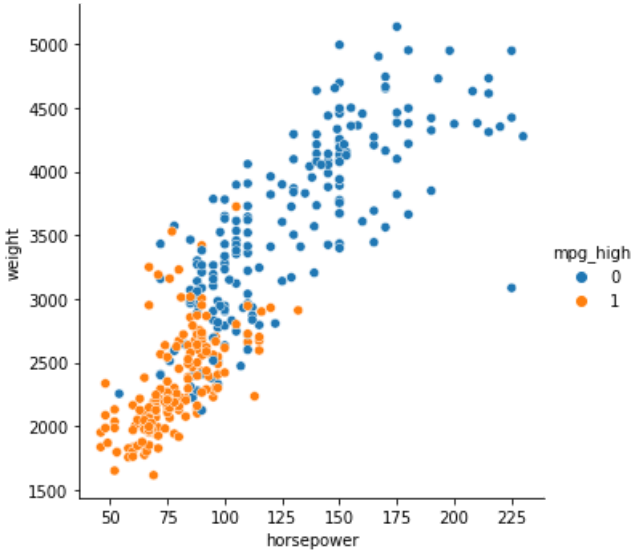
```
sb.catplot('mpg_high', kind='count',data=df)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as
  FutureWarning
<seaborn.axisgrid.FacetGrid at 0x7f939a092fd0>
```

Okay, we can see that both are mostly split up the same. This will be a good advantage in the learning and will help encourage it to learn more effectively. It also helps us feel more confident in the accuracy of train data, and that it is spread out well enough.


```
sb.relplot(x="horsepower", y="weight",hue="mpg_high",data=df)
```

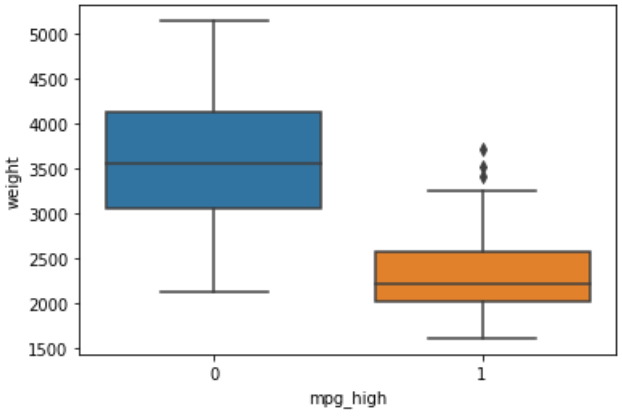
```
<seaborn.axisgrid.FacetGrid at 0x7f939a00b310>
```



Oh wow, we can see a very strong correlation here between the horsepower of a vehicle, the weight, and its gas mileage. As both get higher, the more likely that a car has worse gas mileage. Lighter cars can get more out of their gas in most scenarios, though we can see a few outliers. That's where other attributes will be more helpful to fill in that knowledge gap.

```
sb.boxplot(x="mpg_high", y="weight",data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9399f2ae90>
```



This one lets us see how much tighter the parameters are for cars with good gas mileage. They tend to be quite light comparatively, but there are a few outliers we see. Cars with poor mileage vary a lot more. We saw this on the prior graph, but here it makes it much easier to read. The averages between these two values are very distinct. There is enough overlap to cause some confusion potentially in the learning, but it does not seem to be enough to siginificantly impact the accuracy. It won't be perfect, but we can work on getting there.

7: Train/Test/Split

Now that we've really gotten into understanding the data, let's divvy it up and see how it performs with Logistic Regression, Decision Trees, and a Neural Network.

```
from sklearn.model_selection import train_test_split
x = df.iloc[:, 0:6] #we are learning from 7 different columns to predict the 8th, and Python starts counting from 0
y = df.iloc[:, 7]
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size=.20, random_state=1234)

print("Train shape:", xtrain.shape)
print("Test shape:", xtest.shape)
```

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

8: Logistic Regression

We'll start the process with a simple bit, and try out logistic regression. By default, it uses a LBFGS solver here.

```
from sklearn.linear_model import LogisticRegression

lbfgs = LogisticRegression(random_state=1234)
lbfgs.fit(xtrain, ytrain)
lbfgs.score(xtrain, ytrain)
```

```
0.9035369774919614
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

pred = lbfgs.predict(xtest)
print("Accuracy: ", accuracy_score(ytest, pred))
print("Precision: ", precision_score(ytest, pred))
print("Recall: ", recall_score(ytest,pred))
print("F1: ", f1_score(ytest, pred))
```

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

Looks like it did pretty alright on this data. Let's pull up a confdusion matrix and the sklearn report on this data to try and understand it a bit deeper:

```
from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(ytest,pred))
print(classification_report(ytest,pred,labels=[0,1]))
```

```
[[40 10]
```

```
 [ 1 27]]
              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
```

We can see that it was pretty accurate on the test data. Definitely not perfect, but it wasn't half bad. However, there's some room for improvement, and we can see if others can manage that. Some of our info from before is duplicated with the report, just with a little less precision.

9: Decision Tree

Now, we'll try a decision tree. We do have some obvious lines in between the data, but with the outliers, there is a good chance the accuracy will be similar to logistic regression. We'll see for sure shortly, however.

```
from sklearn.tree import DecisionTreeClassifier

dtc = DecisionTreeClassifier(random_state=1234)
dtc.fit(xtrain, ytrain)
```

```
DecisionTreeClassifier(random_state=1234)
```

Now let's do the predictions, overwriting the value from before for ease:

```
pred = dtc.predict(xtest)
```

Now that we've done that, time to look again at our statistics for it:

```
print("Accuracy: ", accuracy_score(ytest, pred))
print(confusion_matrix(ytest,pred))
print(classification_report(ytest,pred,labels=[0,1]))
```

```
Accuracy:  0.9102564102564102
[[45  5]
 [ 2 26]]
              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
```

Woah! It's improved a bit from before with logistic regression. It isn't a huge jump, but its enough to help. Let's see how the decision tree ended up looking:

```
from matplotlib import pyplot as pplot
from sklearn import tree
tree.plot_tree(dtc, filled=True)
```

```
 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 = [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[3] <= 2329.5\ngini = 0.278\nsamples = 6\nvalue = [5, 1]'),
 Text(0.20588235294117646, 0.16666666666666666, 'X[4] <= 14.75\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, 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[5] <= 77.5\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[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 = [124, 1]'),
 Text(0.7941176470588235, 0.3888888888888889, 'X[1] <= 225.0\ngini = 0.375\nsamples = 4\nvalue = [3, 1]'),
 Text(0.7647058823529411, 0.2777777777777778, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.8235294117647058, 0.2777777777777778, 'gini = 0.0\nsamples = 3\nvalue = [3, 0]'),
 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]')]
```

How interesting! There is a lot going into the tree here, but the accuracy is quite high, and it seems to understand the nuances well, even if this graph is not the most readable.

10: Neural Network

Time to bring out the heavy duty work. Since this is such a small data set, we can't be certain that it will work super well, but we can hope and see what the power of a neural net can do to understand the gas efficiency of cars. To begin, however, we have to fit it down a bit so it can be parsed more effectively in training.

```
from sklearn import preprocessing
scaler = preprocessing.StandardScaler().fit(xtrain)
xtrainscaled = scaler.transform(xtrain)
xtestscaled = scaler.transform(xtest)
```

Now we can move to training and predicting with the neural network.

```
from sklearn.neural_network import MLPClassifier
nnc = MLPClassifier(solver='lbfgs',hidden_layer_sizes=(6,2),max_iter=500,random_state=1234)
nnc.fit(xtrainscaled,ytrain)
pred = nnc.predict(xtestscaled)
```

```
print("Accuracy: ", accuracy_score(ytest, pred))
print(confusion_matrix(ytest,pred))
print(classification_report(ytest,pred,labels=[0,1]))
```

```
Accuracy:  0.8846153846153846
[[43  7]
 [ 2 26]]
              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
```

In the above, a few different attempt were tried to get more accurate results. The 'sgd' solver was used with lessened accuracy, and changing the layer sizes any further away caused lessened accuracy as well. (5,2) hidden layers and (6,2) hidden layers resulted in nearly the same accuracy, however, on the 'sgd' solver. (6,2) was chosen for having the fault more evenly distributed (relatively, based on values in each category) between categories.

11: Analysis

On Results

And here are the results, based on accuracy:

1. Decision Tree (.91)
2. Neural Network (.88)
3. Logistic Regression (.86)

The precision for the low miles per gallon stayed largely the same, but the high mpg varied more. But, if we wanted to rank it, based on weighted average of precision, we would have this hierarchy:

1. Decision Tree (.91 -> .96 low, .84 high)
2. Neural Network (.90 -> .96 low, .79 high)
3. Logistic Regression (.89 -> .98 low, .73 high)

And on weighted recall:

1. Decision Tree (.91 -> .90 low, .93 high)
2. Neural Network (.88 -> .86 low, .93 high)
3. Logistic Regression (.86 -> .80 low, .96 high)

We see overall that Decision Trees seem to be the best in this situation. Across the board, it sits at a stable .91 accuracy, precision, and recall. It gets the best results most assuredly, and looking back on the graphs of the data, we can see clear lines in many situations that it could manage well between different attribute types. Considering the size of the data set, this also makes sense.

We were also able to look at the Decision Tree's logic, and see how it managed to determine the outliers. On one level, it looks like overfitting with the specifics, but it was able to actually generalize quite well. While other algorithms look for trends more so, the Decision Tree looked for divisive lines, which worked better for this data set. As cars, they are subject to their own times' standards, their engineers, and age as time continues. This creates specifics that a Decision Tree sees better.

Looking at how some algorithms are better at different aspects of learning, the accuracy of this could well be improved by ensemble learning methods. Where Logistic Regression is the most precise for understanding low accuracy, we could use the higher aspects of Decision Trees to further it. With the variance of Neural Networks, it could be arranged so that it would excel in one area more than the other, and therefore make one more accurate model to understand this. It likely won't ever be perfect, but it doesn't have to be. It understands the data quite well in its current state. The Decision Tree's accuracy is high enough to be effective.

On Python

As for my own foray into Python, this has certainly been interesting. R was very finicky in my experience, and I had more trouble debugging it than I did Python. SKlearn was a very powerful set that allowed me to do much in a few lines of code, and with its Python roots, was not specific about how I needed to present data. There are pros and cons to that, as I find df.attribute and df['attribute'] effectively being the same somewhat confusing to read and understand. I found myself switching between them when coding, uncertain of which was 'better'. I'm not much of a fan of the ambiguity.

In the end, I have reasons to like them both. Python with sklearn was more versatile, but R was more specialized it felt towards Machine Learning and offered a lot of power. I like R a lot, but Python and sklearn is much kinder.