# Importing Autos Data Set

```
### First, we read in the Autos.csv dataset from our Github Repository, then store it in a st
import pandas as pd
url = 'https://raw.githubusercontent.com/meintgl/ML_Portfolio/main/Auto.csv'
df = pd.read_csv(url)

### Using the .head function to show the first few columns.
print(df.head())

### Outputting the dimenions for the datashape. We notice that it is 392 rows by 9
print('\nDimensions of data frame:', 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 data frame: (392, 9)
```

# Data Exploration

```
### Describing the mpg, weight, and year columns.
df[["mpg","weight","year"]].describe(include="all")

# We are finding a lot of information from these three columns.
# The mean  miles per gallon (mpg) was 23.44. The range is about 35 (min-max).
# The mean weight is about 3000, and the range is about 3500. The range is really big and var
# The mean year is 76. This is in the format for years as in 19xx, so the mean year was 1976.
# so only a 12 year difference between the oldest and newest car.
```

| | 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 |

```
### Checking data types.
df.dtypes
```

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

```
### We want to convert the cylinders and origin column to categorical. We can do this in two
### First, we use cat.codes to convert cylinders into categorical.

df.cylinders = df.cylinders.astype('category').cat.codes

### Let's print to see of the cylinders column changed.
print(df.dtypes, "\n")
print(df.head())
```

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

    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
4  17.0          4         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
```

```
### We can convert the origins column to categorical as well with another similar method, wit


df.origin = df.origin.astype('category')

print(df.dtypes, "\n")
print(df.head())
```

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

    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
4  17.0          4         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
```

```
### In the next part of data exploration, it's important to check for NA values, and deal wit
### In this case, we will delete rows with NA's after checking for NAs.
df.isnull().sum()

### There is 1 NA in acceleration, and 1 in year. Let's drop it then check the overall dimens
df = df.dropna()
print('\nDimensions of data frame:', df.shape)
### This makes sense since 3 values are removed.
```

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

```
### Now let's create a new column named mpg_high. We need to make it categorical, where the c
```

```
### if the mpg is above average, else it is 0. We print this to see that this is achieved.

df['mpg_high'] = pd.cut(df['mpg'], bins=[0, df["mpg"].mean(), float('Inf')], labels=[0, 1])
print(df.loc[0:5,:])
```

```
    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

  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
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4: 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
  after removing the cwd from sys.path.
```

```
### Let's delete the mpg and name columns to make sure the algorithm doesn't predict mpg_high
df = df.drop(columns=['mpg', 'name'])
print(df.head())

## We see the mpg_high column remains, and the mpg and name columns are deleted.
```

```
   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
```

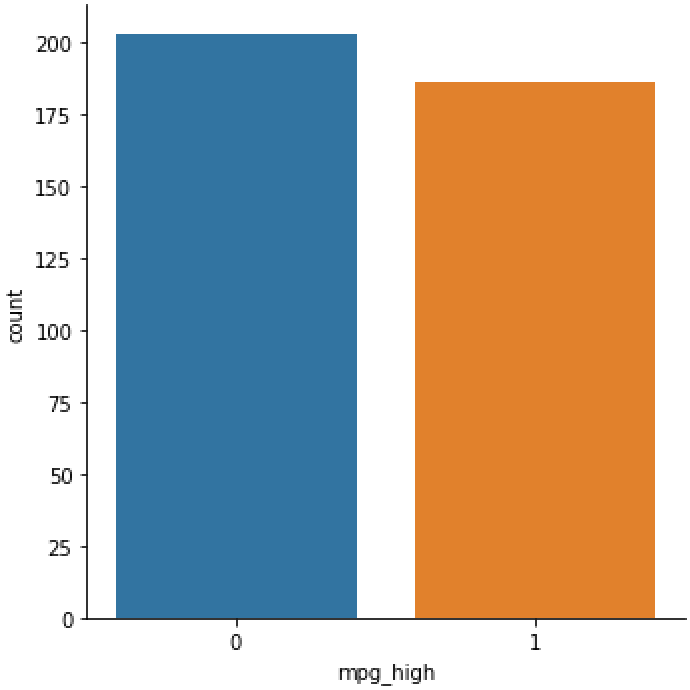
# Graphical Exploration with Seaborn

```
### First, let's important seaborn. Seaborn provides easy data visualization.
import seaborn as sb
```

```
### The first plot will be a categorical plot for mpg_high. First we create a y that will enc
```

```
sb.catplot(x="mpg_high", kind='count', data=df)
## We can see that there are more vehicles with miles per gallon below the average than above
```

```
<seaborn.axisgrid.FacetGrid at 0x7f977dc06a50>
```




```
### There are 203 instances of mpg_high = 0.
df['mpg_high'].value_counts()[0]
```

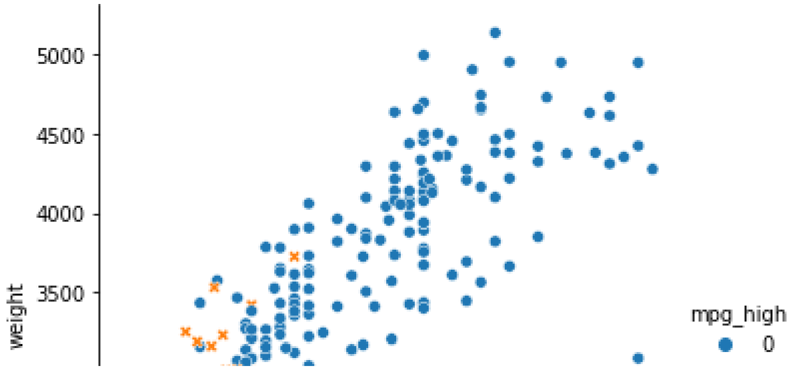
```
203
```

```
### There are 203 instances of mpg_high = 1.
df['mpg_high'].value_counts()[1]
```

```
186
```

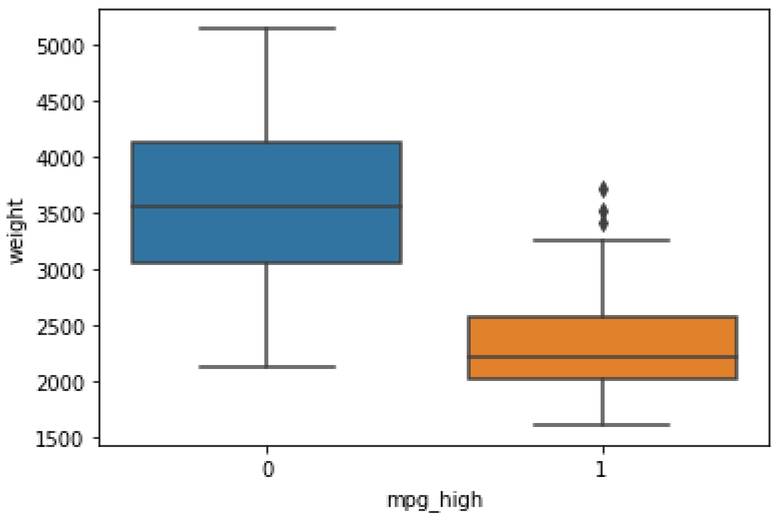
```
### The second plot will be a Seaborn Relational Plot for horsepower vs weight.
sb.relplot(x='horsepower', y='weight', data=df, hue=df.mpg_high, style=df.mpg_high)
### We can see that automotives with lower than average of miles per gallon (mpg_high = 0) ha
```

```
<seaborn.axisgrid.FacetGrid at 0x7f977927dd10>
```




```
### The third plot will be a boxplot with mpg_high on the x-axis, and weight on the y-axis.
sb.boxplot('mpg_high', y='weight', data=df)
### We can see that automotives with mpg_high = 1 have on average, a lower weight. However, t
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass t
  FutureWarning
<matplotlib.axes._subplots.AxesSubplot at 0x7f97790f5c50>
```




## Train / Test Split

```
### Let's print the dataframe just to be sure everything looks good before doing the train/te
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
```

```
## We will split our data into 80% train and 20% test. We will use random_state = 1234.
from sklearn.model_selection import train_test_split

x = df.loc[:, ['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'year', '
y = df.mpg_high

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

## Logistic Regression

```
## Let's important logistic regression from the LogisticRegression package.
from sklearn.linear_model import LogisticRegression

## Now we see the fit for our x_train and y_train data.
clf = LogisticRegression(max_iter=390)
clf.fit(x_train, y_train)
clf.score(x_train, y_train)
```

```
0.9035369774919614
```

```
### We can create a prediction using clf.predict
pred = clf.predict(x_test)
```

```
### Let's print the accuracy score and metrics. Our model had an accuracy of 89.7%, which is
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

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))
```

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

```
### Let's create a confusion matrix. We can see that our model only predicted 8 wrong.
```

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, pred)
```

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

# Decision Tree

```
## Let's important decision trees from the DecisionTreeClassifier.

from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier()
dt.fit(x_train, y_train)
```

```
DecisionTreeClassifier()
```

```
### We can create a prediction using dt.predict
pred2 = dt.predict(x_test)
```

```
### Let's print the accuracy score and metrics. Our model had an accuracy of 92.3%. It is imp
### results will vary across test runs.
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))
```

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

```
### Lastly, we can visualize the tree using the tree package. However, we see that there are
from sklearn.datasets import load_iris
from sklearn import tree
iris = load_iris()
x, y = iris.data, iris.target
tree.plot_tree(dt)
```

```
[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[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 =
[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[5] <= 71.5\ngini = 0.278\nsamples
= 6\nvalue = [5, 1]'),
 Text(0.20588235294117646, 0.16666666666666666, 'X[2] <= 88.0\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,
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
```

## Neural Networks

```
### First, let's the data and import preprocessing.
from sklearn import preprocessing

scaler = preprocessing.StandardScaler().fit(x_train)

x_train_scaled = scaler.transform(x_train)
x_test_scaled = scaler.transform(x_test)
```

```
### Training our neural network. Our network topology is lbfgs, which is in the family of qua
from sklearn.neural_network import MLPClassifier

neural = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(7, 4), max_iter=500, random_state=
neural.fit(x_train_scaled, y_train)
```

```
MLPClassifier(hidden_layer_sizes=(7, 4), max_iter=500, random_state=1234,
              solver='lbfgs')
```

```
###  We can create a prediction using clf.predict
pred3 = neural.predict(x_test_scaled)
```

```
### Let's output results for the accuracy score and a confusion matrix.

print('accuracy = ', accuracy_score(y_test, pred3))
confusion_matrix(y_test, pred3)
```

```
accuracy =  0.8846153846153846
array([[43,  7],
       [ 2, 26]])
```

```
### For this model, let's print out the classification report.

from sklearn.metrics import classification_report
print(classification_report(y_test, pred3))
```

```
              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
```

```
### Let's train another model with a different network topology. This time I will use sgd, wh
### layer sizes, and increase the max_iterations.
neural2 = MLPClassifier(solver='sgd', hidden_layer_sizes=(4, 2), max_iter=500, random_state=1
neural2.fit(x_train_scaled, y_train)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/neural_network/_multilayer_perceptron.py
  ConvergenceWarning,
MLPClassifier(hidden_layer_sizes=(4, 2), max_iter=500, random_state=1234,
              solver='sgd')
```

```
###  We can create a prediction using clf.predict
pred4 = neural2.predict(x_test_scaled)
```

```
### Let's output results for the accuracy score and a confusion matrix.

print('accuracy = ', accuracy_score(y_test, pred4))
confusion_matrix(y_test, pred4)
```

```
accuracy =  0.8589743589743589
array([[42,  8],
       [ 3, 25]])
```

```
### For this model, let's print out the classification report.

from sklearn.metrics import classification_report
print(classification_report(y_test, pred4))
```

```
              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
```

We see that the first network was better. It had more iterations and a different solver. The predictions were about 3% higher. After research online, sgd is faster and simpler to implement, but lbfgs creates results closer to what is optimal at a higher cost. For hidden layer sizes, for data with higher dimensions such as this one, it was better to have 3-5 hidden layers. I experimented a good amount and 7-4 gave good results. Overall, although both models were solid, the first one with lbfgs performed better.

## Analysis

```
### Let's print the classification metrics: accuracy, recall, an d precision for each of the
print('Logistic Regression Classification Report')
print(classification_report(y_test, pred))

print('Decision Tree Classification Report')
print(classification_report(y_test, pred2))

print('Neural Network with lbfgs Classification Report')
print(classification_report(y_test, pred3))
```

```
Logistic Regression Classification Report
              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

Decision Tree Classification Report
              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 Network with lbfgs Classification Report
              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
```

### Algorithm Analysis

We can see that decision tree performed the best, neural networks the second best, and logistic regression slightly behind. However, all of the algorithms predicted the accuracy pretty well. Decision trees performed the best. This can be attributed to decision trees performing better than neural networks with categorical variables with multiple classes, such as cylinders. In neural networks, it is fine to handle binary categorical variables, but some columns have more than that. Also, neural networks are more complex and generally need more tuning, while decision trees are easier to interpet and even visually see where data is split when we plotted the tree. Logistic regression worked fine, but is not as well-suited for columns with categorical variables (binary type or multi-class). We had two, so it maybe struggled on those columns.

**R vs sklearn**

I prefer sklearn to Rr heavily. Although R feels easier initially, once I read the documentation and practiced writing the code and models with sklearn and Python, everything felt more intuitive, and there was less errors. I felt like python is overall smoother to use with the syntax and the way it was constructed, and finding documentation for each of the parameters and functions was easy.

The best thing for me in learning sklearn was the packages. The packages are designed to be intuitive, with seaborn handling the plots, pandas handling data analysis and manipulation, and packages for each of the algorithms that were easy to implement (eg. LogisticRegression, DecisionTreeClassifier, tree).