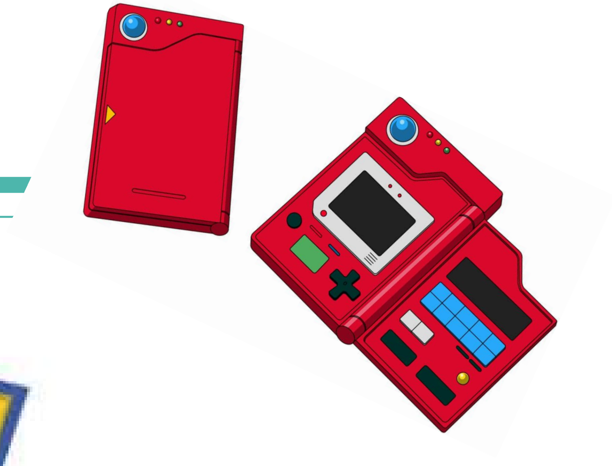
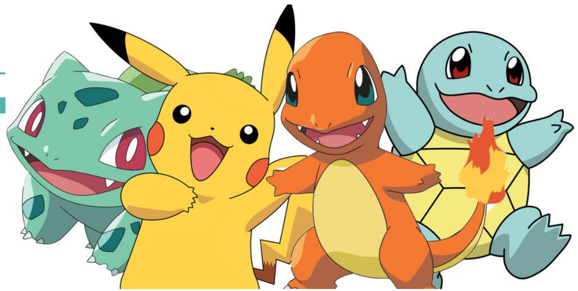


Pokémon

Predicting Pokemon types with stats from over 721 Pokémon



Background Info

- Nintendo released Pokemon Red & Green for Gameboy Japan in 1996
- Each Pokemon has a type and specific base stat that made it strong or weak against other Pokemon
- 150 Pokemon in Gen I, with 15 types
- 802 Pokemon in Gen VII with 18 types
- Pokemon can have a primary and secondary type
- For this project, we will be looking at primary type only because with the combination possibilities, there are 146 type combinations

Problem Statement and Hypothesis

By looking at and comparing the pokemon base stats (including HP, attack, special attack, defense, special defense, and speed) can we accurately predict the pokemon's primary type.   Some Pokemon only have 1 type, so Type_2 will show as NaN. We are going to ignore these for this project.

There are a lot of pokemon out there, and it'd be cool to predict its strengths and weaknesses by looking at base stats. I've been playing Pokemon since the original games came to USA in 1997, and still love it

- Note- we will only be looking at the game data. This will not include the trading cards, Pokemon Go, or the animated series

About the Data

- I found this data on Kaggle https://www.kaggle.com/abcsds/pokemon
- Most of the data was sourced from data found on Bublapedia, a Pokemon encyclopedia:http://bulbapedia.bulbagarden.net/wiki/Main_Page
- The data set contains information from Gen I to Gen VI, so I pulled my own testing data from Bulbapedia and used Pokemon from Gen VII as testing. I will split with Test/Train Split after

About the Pokemon Generations

Generation	Years	Games
Gen I	1996-1999	Red, Green, Blue, Yellow
Gen II	1999-2002	Gold, Silver, Crystal
Gen III	2002-2006	Ruby, Sapphire, Emerald, Remakes-FireRed, LeafGreen
Gen IV	2006-2010	Diamond, Pearl, Platinum Remakes- HeartGold, SoulSilver
Gen V	2010-2013	Black, White Remakes- Black2 White2
Gen VI	2013-2016	X, Y Remakes- Omega Ruby, Alpha Sapphire
Gen VII	2016-present	Sun, Moon

First Looks

There are 801 rows and 13 columns

```
In [18]: df.shape
Out[18]: (801, 13)
```

```
In [29]: df.head()
```

Out[29]:

	Pokemon_Number	Name	Type_1	Type_2	Total	HP	Attack	Defense	Sp.Atk	Sp.Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	1	False
3	3	VenusaurMega Venusaur	Grass	Poison	625	80	100	123	122	120	80	1	False
4	4	Charmander	Fire	NaN	309	39	52	43	60	50	65	1	False

Looking at Pokemon Types

```
In [30]: df.Type_1.value_counts()

Out[30]: Water       112
         Normal       98
         Grass        70
         Bug          69
         Psychic      57
         Fire         52
         Electric     44
         Rock         44
         Ground       32
         Ghost        32
         Dragon       32
         Dark         31
         Poison       28
         Fighting     27
         Steel        27
         Ice          24
         Fairy        17
         Flying        4
         Name: Type_1, dtype: int64
```

Graphing

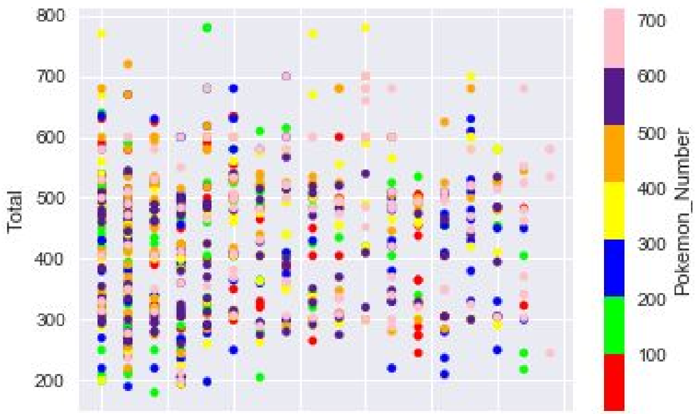
Renaming the types with a numeric value

```
In [34]: df['Type_1'] = df.Type_1.map({'Water':0, 'Normal':1, 'Grass':2, 'Bug':3, 'Psychic':4, 'Fire':5,
'Electric':6, 'Rock':7, 'Ground':8, 'Ghost':9, 'Dragon':10, 'Dark':11, 'Poison':12, 'Fighting':13,
'Steel':14, 'Ice':15, 'Fairy':16, 'Flying':17})
```

Scatter Plot: X = Type_1, Y = Total (all stats added together) and c = Pokemon Number

```
Out[44]: <matplotlib.axes._subplots.AxesSubplot at 0x1187394d0>
```

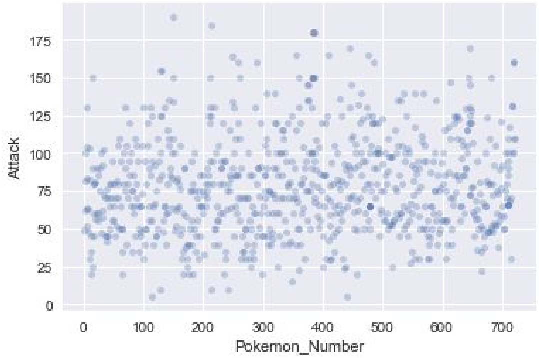
You can see the 18 columns for the different types. You can also see that there is a high range with the TOTAL metric



Scatter Plots of Pokemon Stats

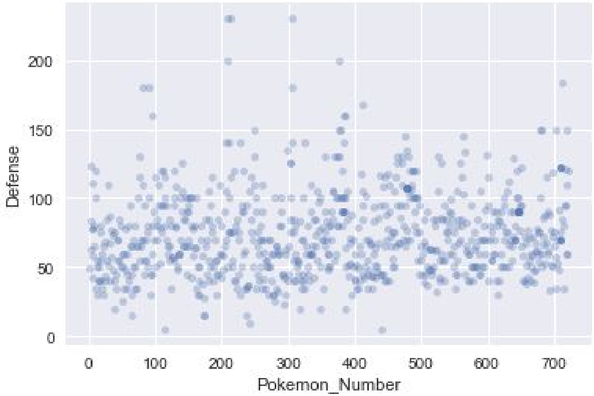
Attack

```
In [50]: # add transparency
         df.plot(kind='scatter', x='Pokemon_Number', y='Attack', alpha=0.3)

Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x11874b990>
```



Defense

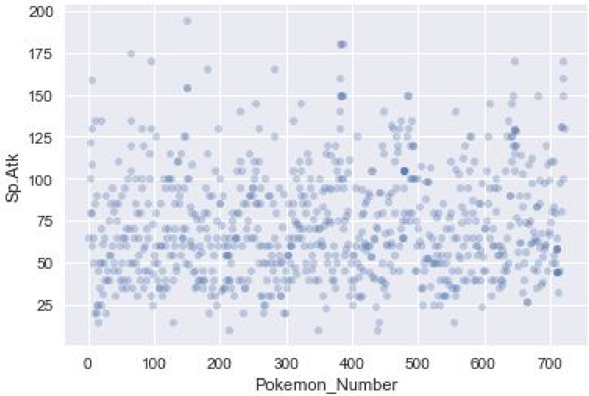
```
In [51]: df.plot(kind='scatter', x='Pokemon_Number', y='Defense', alpha=0.3)

Out[51]: <matplotlib.axes._subplots.AxesSubplot at 0x118dab490>
```



Continued

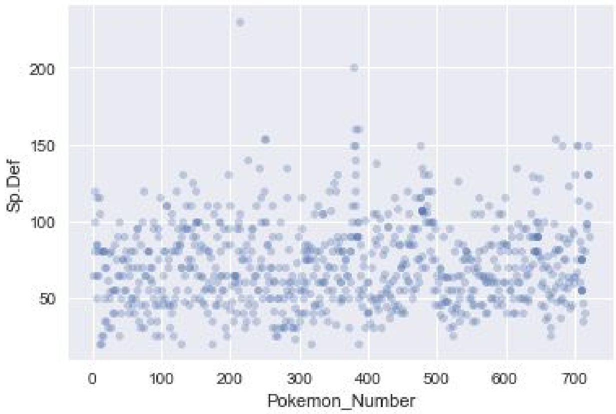
Special Attack

```
In [54]: df.plot(kind='scatter', x='Pokemon_Number', y='Sp.Atk', alpha=0.3)
Out[54]: <matplotlib.axes._subplots.AxesSubplot at 0x1191dba90>
```



Special Defense

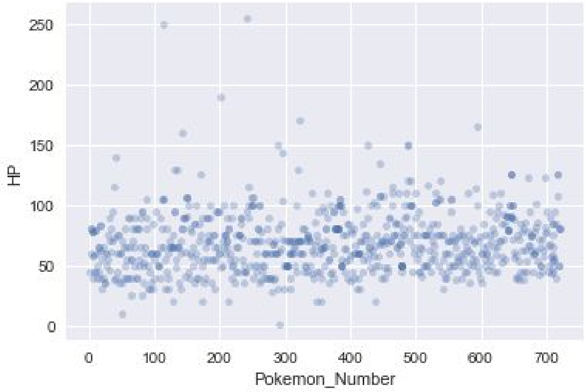
```
df.plot(kind='scatter', x='Pokemon_Number', y='Sp.Def', alpha=0.3)
<matplotlib.axes._subplots.AxesSubplot at 0x110b886d0>
```



Continued

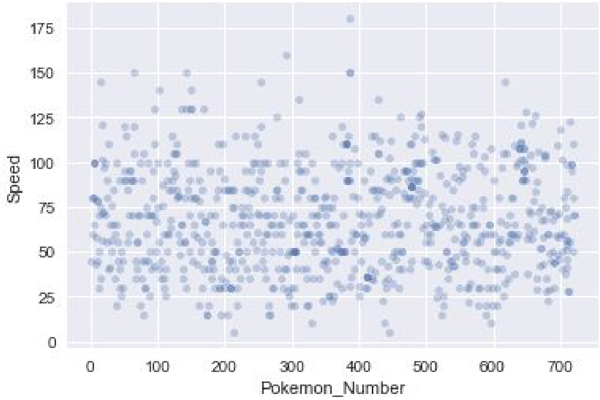
HP

```
In [52]: df.plot(kind='scatter', x='Pokemon_Number', y='HP', alpha=0.3)
Out[52]: <matplotlib.axes._subplots.AxesSubplot at 0x1191dba10>
```

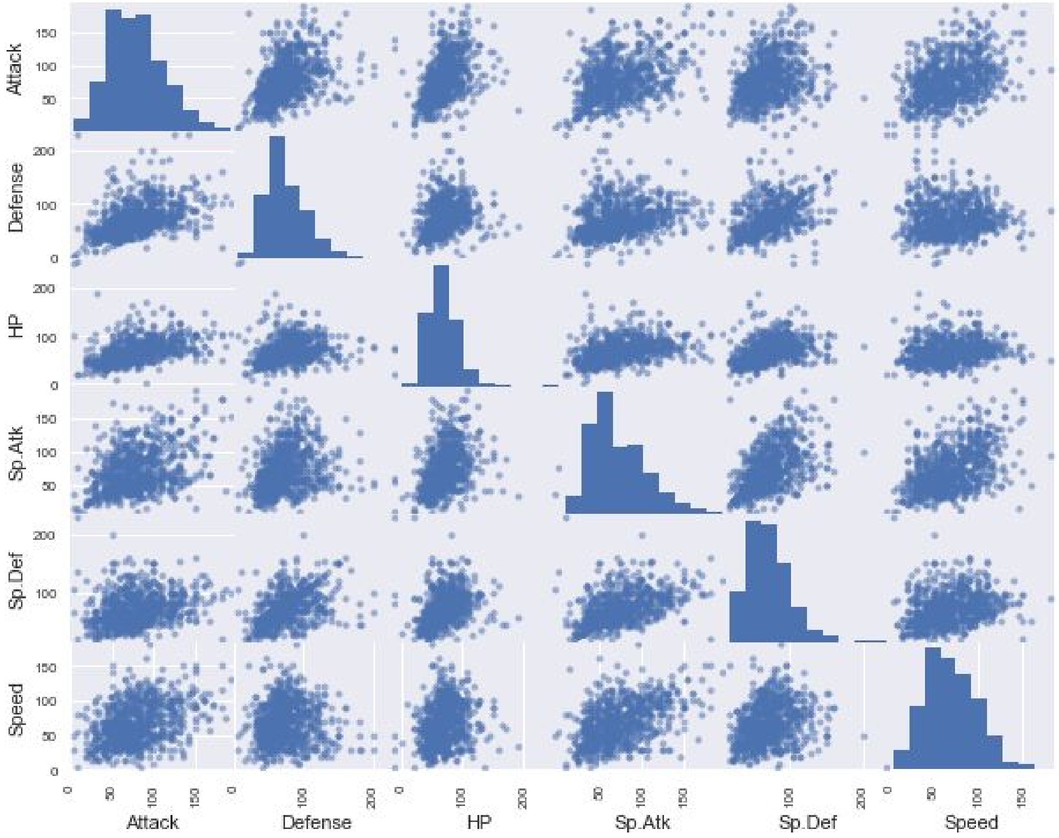


Speed

```
df.plot(kind='scatter', x='Pokemon_Number', y='Speed', alpha=0.3)
<matplotlib.axes._subplots.AxesSubplot at 0x110b926d0>
```



Scatter Matrix of Pokemon Stats



KNN to predict Type based on Stats- Step 1

Basic KNN Model, without test/train split

```
#Starting a KNN to see features versus type to determine strength in type1
# store feature matrix in "X"
feature_cols = ['Attack', 'Defense', 'HP', 'Sp.Atk', 'Sp.Def', 'Speed']
X = df[feature_cols]
print (X)
```

	Attack	Defense	HP	Sp.Atk	Sp.Def	Speed
0	49	49	45	65	65	45
1	62	63	60	80	80	60
2	82	83	80	100	100	80
3	100	123	80	122	120	80
4	52	43	39	60	50	65
5	64	58	58	80	65	80
6	84	78	78	109	85	100
7	130	111	78	130	85	100
8	104	78	78	159	115	100
9	48	65	44	50	64	43
10	63	80	59	65	80	58
11	83	100	79	85	105	78
12	103	120	79	135	115	78
13	30	35	45	20	20	45

Creating our features (X)

Step 1 Continued...

Creating our response vector, y and checking types

```
# store response vector in "y"
y = df.Type_1
print(y)
```

```
0       2
1       2
2       2
3       2
4       5
5       5
6       5
7       5
8       5
9       0
10      0
```

```
# check X's type
print type(X)
print type(X.values)
```

```
<class 'pandas.core.frame.DataFrame'>
<type 'numpy.ndarray'>
```

```
# check X's shape (n = number of observations, p = number of features)
print X.shape
```

```
(800, 6)
```

```
# check y's shape (single dimension with length n)
print y.shape
```

```
(800,)
```

Step 2. Import the Estimator

In this step, we're going to import KNN Classifier from Sklearn

```
#Step 2: Decide on the estimator you want to to use and import that class
from sklearn.neighbors import KNeighborsClassifier
```

Step 3. Instantiate the estimator

```
#Step 3: "Instantiate" the "estimator"
knn = KNeighborsClassifier(n_neighbors=1)
type(knn)
```

```
sklearn.neighbors.classification.KNeighborsClassifier
```

```
print knn
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')
```

Step 4. Fit the Model

```
#Step 4: Fit the model with data (aka "model training")
knn.fit(X, y)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')
```

We're using stats from Generation I through Generation VI to fit the KNN model. For our predictions, we'll use Pokemon stats from Gen VII

Step 5. Use the model to predict the response for a new observation

```
#Step 5: Use the model to predict the response for a new observation
#this data only goes through Gen VI, using pkmn from VII to have knn predict type_1
#Grass: Fomantis num 753,
#Attack55,Defense35,HP40,Sp.Atk50,Sp.Def35,Speed35....looking for type #2 grass
#Electric: Xurkitree num 796
#Attack89,Defense71,HP 83,Sp.Atk173,Sp.Def71,Speed83....looking for type #6 electric
#Psychic: Necrozma num 800
#Attack107,Defense101,HP97,Sp.Atk127,Sp.Def89,Speed79....looking for type #4 psychic
new_observation = [[55, 35, 40, 50, 35, 35], [89, 71, 83, 173, 71, 83], [107, 101, 97, 127, 89, 79]]
knn.predict(new_observation)
```

Adding a new observation to see how it is classified

Step 5 continued...

```
#adding a new pokemon- dragon type Jangmo-o num 782 - should be type 10
#adding water, Pyukumuku, num 771 should be type 0
X_new = [[55, 35, 40, 50, 35, 35], [89, 71, 83, 173, 71, 83], [107, 101, 97, 127, 89, 79], [55, 65, 45,
45, 45, 45], [60, 130, 55, 30, 130, 5]]
knn.predict(X_new)
```

```
array([3, 6, 5, 0, 9])
```

```
#predict probability
knn.predict_proba
```

```
<bound method KNeighborsClassifier.predict_proba of KNeighborsClassifier(algorithm='auto', leaf_size=30, m
etric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')>
```

Step 6. Evaluate Accuracy

```
#Step 6: Evaluate the error or accuracy of the model--measure accuracy/cross validation
# instantiate the model (using the value K=5)
knn = KNeighborsClassifier(n_neighbors=5)

# fit the model with data
knn.fit(X, y)

# predict the response for new observations
knn.predict(X_new)
```

```
array([1, 5, 5, 0, 7])
```

```
# calculate predicted probabilities of class membership
knn.predict_proba(X_new)
```

```
array([[ 0. ,  0.2,  0. ,  0.2,  0. ,  0. ,  0.2,  0. ,  0. ,  0. ,  0. ,
         0.2,  0.2,  0. ,  0. ,  0. ,  0. ,  0. ],
       [ 0. ,  0.2,  0. ,  0. ,  0. ,  0.4,  0.2,  0. ,  0. ,  0. ,  0. ,
         0.2,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ],
       [ 0. ,  0. ,  0. ,  0. ,  0. ,  0.6,  0.2,  0. ,  0. ,  0. ,  0. ,
         0.2,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ],
       [ 0.2,  0. ,  0.2,  0.2,  0. ,  0. ,  0. ,  0.2,  0. ,  0. ,  0. ,
         0.2,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ],
       [ 0. ,  0. ,  0.2,  0. ,  0. ,  0. ,  0. ,  0.6,  0. ,  0.2,  0. ,
         0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ]])
```

KNN Model pt 1 Conclusions

- It predicts sometimes, but it's not very accurate.
- We can use different models to increase the accuracy or find a more precise accuracy measurement
- I will start working with decisions trees, clustering, and Random Forests to see if the accuracy will improve
- I will also use test/train split to see how that changes the model

Adding Test/Train Split to KNN

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
#before splitting
print X.shape
```

```
(800, 6)
```

```
# after splitting
print X_train.shape
print X_test.shape
```

```
(600, 6)
(200, 6)
```

```
# before splitting
print y.shape
```

```
(800,)
```

```
# after splitting
print y_train.shape
print y_test.shape
```

```
(600,)
(200,)
```

```
# WITHOUT a random_state parameter
X_train, X_test, y_train, y_test = train_test_split(X, y)
# print the first element of each object
print X_train.head(1)
print X_test.head(1)
print("")
print y_train.head(1)
print y_test.head(1)
```

```
   Attack  Defense  HP  Sp.Atk  Sp.Def  Speed
4      52       43  39      60      50     65
     Attack  Defense  HP  Sp.Atk  Sp.Def  Speed
534      65      107  50     105     107     86

4    5
Name: Type_1, dtype: int64
534    6
Name: Type_1, dtype: int64
```

```
# WITH a random_state parameter
#set random state to 1 to match earlier models
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)

# print the first element of each object
print X_train.head(1)
print X_test.head(1)
print("")
print y_train.head(1)
print y_test.head(1)
```

```
     Attack  Defense  HP  Sp.Atk  Sp.Def  Speed
132     110       80  70      55      80    105
   Attack  Defense  HP  Sp.Atk  Sp.Def  Speed
8     104       78  78     159     115    100

132    3
Name: Type_1, dtype: int64
8    5
Name: Type_1, dtype: int64
```

Checking our KNN post Test/Train Split

```
#going to re-check our KNN with test_train split data
#also for classification random forests
#STEP 1: split X and y into training and testing sets (using random_state for reproducibility)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
```

```
# STEP 2: train the model on the training set (using K=1)
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')
```

```
# STEP 3: test the model on the testing set, and check the accuracy
y_pred_class = knn.predict(X_test)
print metrics.accuracy_score(y_test, y_pred_class)
```

```
0.145
```

```
#testing with neighbors=5
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_class = knn.predict(X_test)
print metrics.accuracy_score(y_test, y_pred_class)
#this one works better :)
```

```
0.21
```

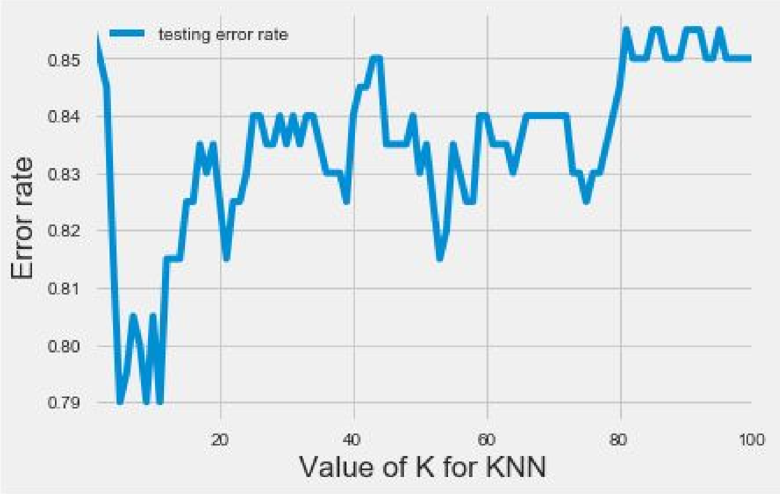
Continued...(finding error values)

```python
# calculate TRAINING Accuracy and TESTING accuracy for K=1 through 100
#
k_range = range(1, 101)
training_error_rate = []
testing_error_rate = []

for k in k_range:

    # instantiate the model with the current K value
    knn = KNeighborsClassifier(n_neighbors=k)

# calculate training error
    knn.fit(X, y)
    y_pred_class = knn.predict(X)
    training_accuracy = metrics.accuracy_score(y, y_pred_class)
    training_error_rate.append(1 - training_accuracy)

    # calculate testing error
    knn.fit(X_train, y_train)
    y_pred_class = knn.predict(X_test)
    testing_accuracy = metrics.accuracy_score(y_test, y_pred_class)
    testing_error_rate.append(1 - testing_accuracy)
```

```python
# allow plots to appear in the notebook
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

```python
# plot the relationship between K (HIGH TO LOW) and TESTING Accuracy
df.plot(y='testing error rate')
plt.xlabel('Value of K for KNN')
plt.ylabel('Error rate') #lower is better for this
```

```
<matplotlib.text.Text at 0x1257f41d0>
```



View our K Values and Error Rates

```
# find the minimum testing error and the associated K value
df.sort_values(by='testing error rate').head()
```

	testing error rate	training error rate
K		
9	0.790	0.60250
5	0.790	0.56375
11	0.790	0.63250
6	0.795	0.56875
8	0.800	0.59500

```
# alternative method
min(zip(testing_error_rate, k_range))
```

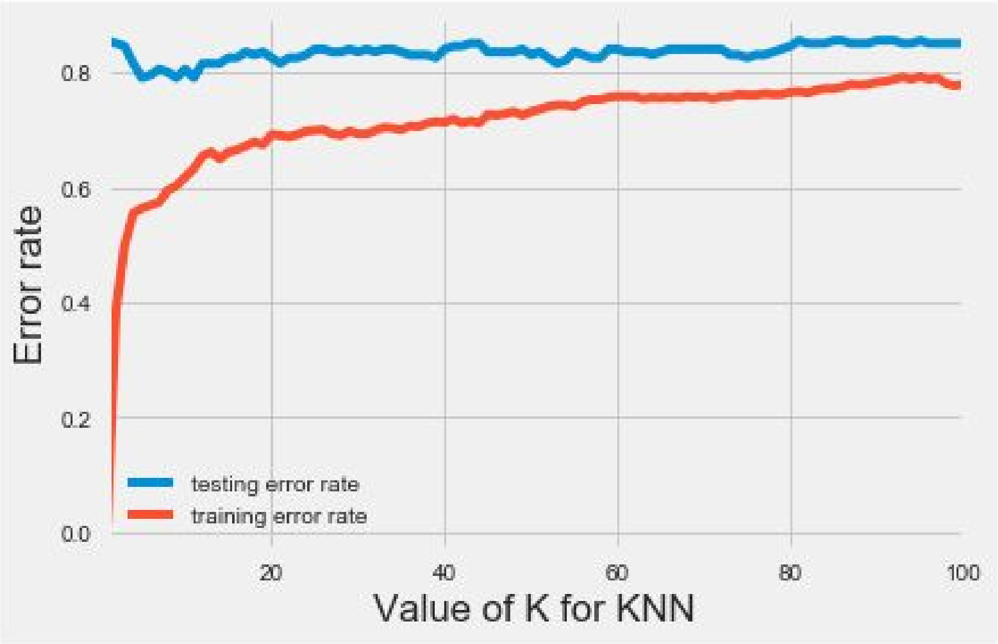
```
(0.7900000000000004, 5)
```

- We want the lowest error rates, and we find that at K = 5
- K=9 isn't far off, but the training error is a bit higher
- The zip at the bottom reassures that we want to pick K=5
- We can also see that we are left with a 21 % accuracy with our testing data

Plotting K for KNN

This model is comparing the testing data to the training data.

The testing data has higher error rates than the training data



Decision Trees

Import DecisionTreeRegressor and Cross_validation from sklearn

```python
# use cross-validation to estimate the RMSE for this model
from sklearn.cross_validation import cross_val_score
scores = cross_val_score(treereg, X, y, cv=10, scoring='neg_mean_squared_error')
np.mean(np.sqrt(-scores))
```

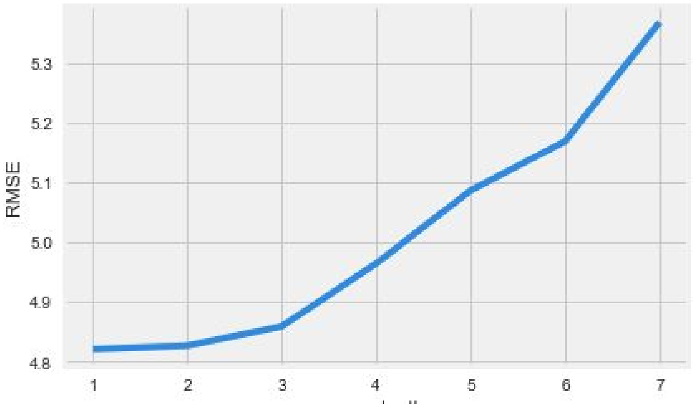
```
6.9837090723415445
```

```python
treereg = DecisionTreeRegressor(max_depth=4, random_state=1)
scores = cross_val_score(treereg, X, y, cv=10, scoring='neg_mean_squared_error')
np.mean(np.sqrt(-scores))
```

```
5.0199406277102891
```

```python
# plot max_depth (x-axis) versus RMSE (y-axis)
plt.plot(max_depth_range, RMSE_scores)
plt.xlabel('max_depth')
plt.ylabel('RMSE')
```

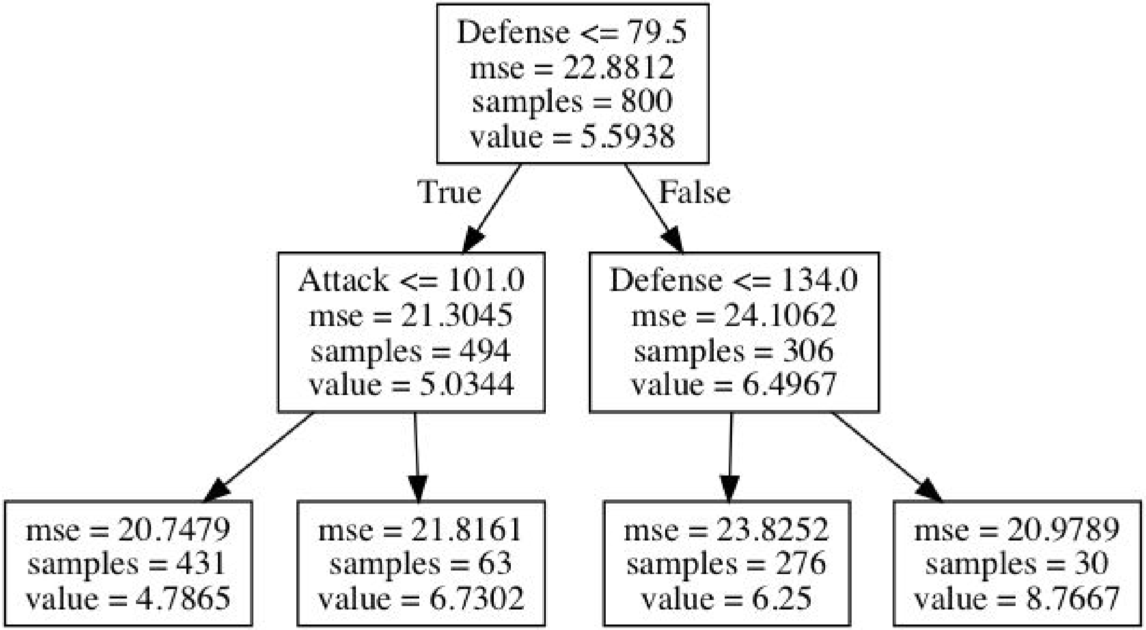
```
<matplotlib.text.Text at 0x1255e0bd0>
```



From the graph, we can see that we'll have the best results with 1 or 2 levels

Decision Tree Results

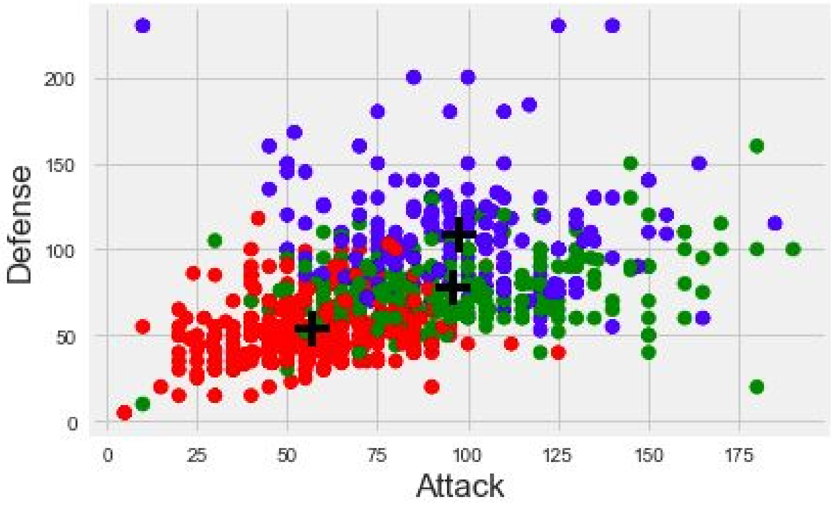
	feature	importance
0	Attack	0.265159
1	Defense	0.734841
2	HP	0.000000
3	Sp.Atk	0.000000
4	Sp.Def	0.000000
5	Speed	0.000000

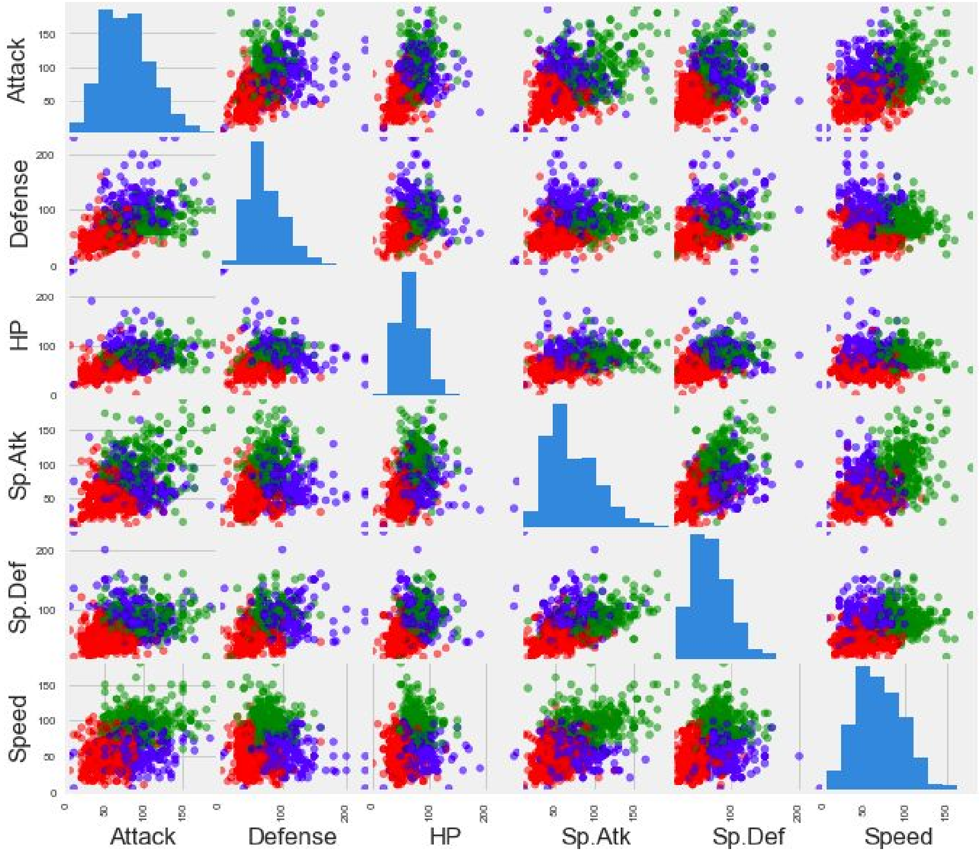


3 Clusters based on Attack and Defense

The + marks the center of the cluster

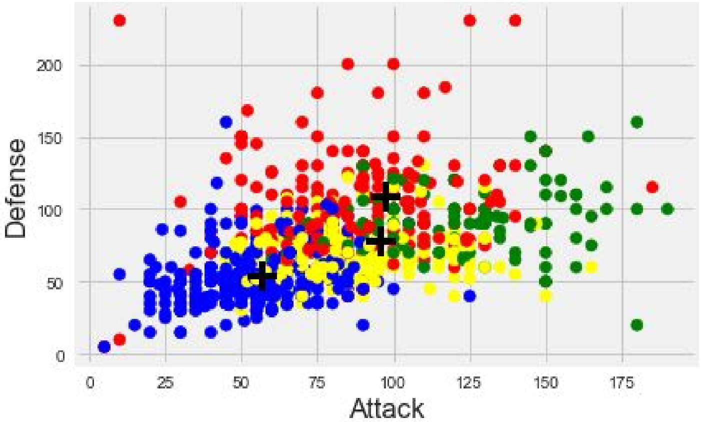
<matplotlib.text.Text at 0x134103390>

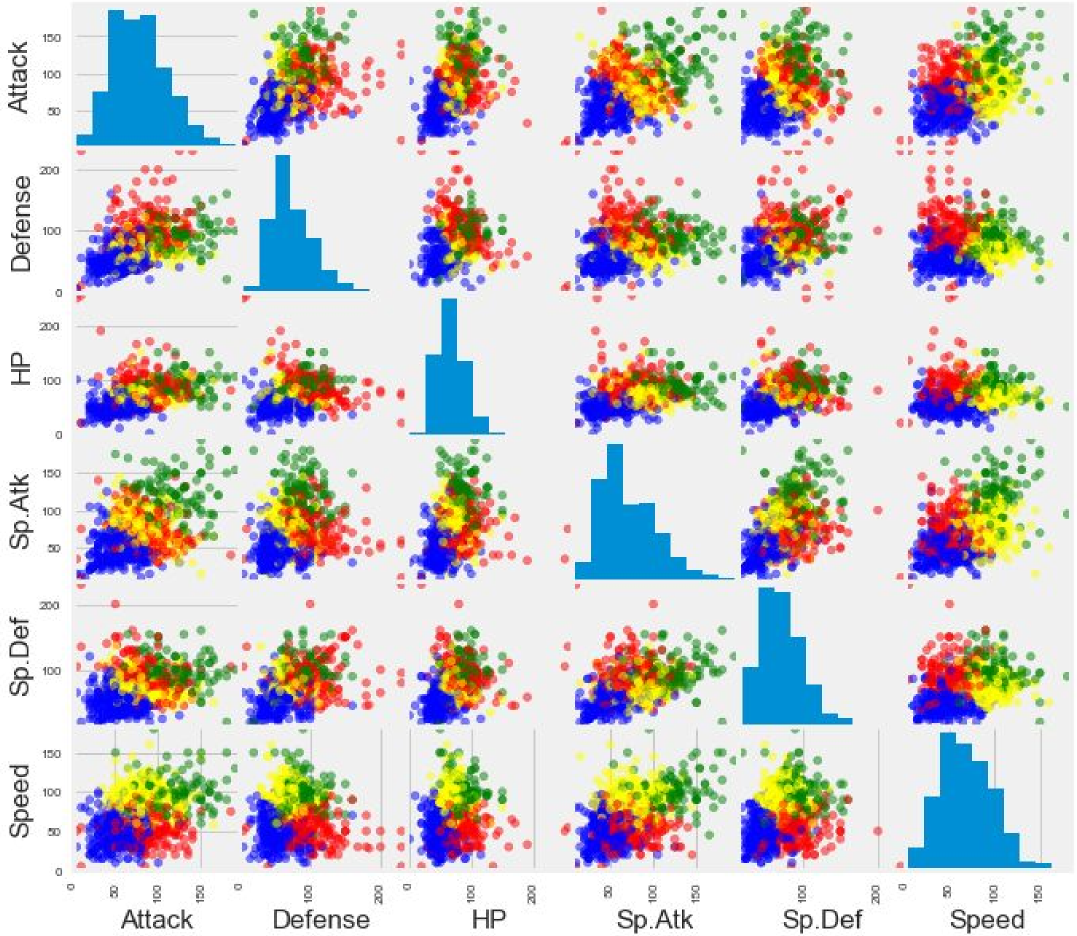




4 Clusters

<matplotlib.text.Text at 0x13a5960d0>





Random Forest Classifier

Random Forest gives us a more accurate feature importance than the decision trees. Using this, we can also see that we have a 79% error rate at our best K value. The best K value is 5, and we have 21% accuracy

```
#split into testing and training data for random forst using classifier

treereg = RandomForestClassifier(n_estimators=21, max_depth=14, random_state=1)
treereg.fit(X_train, y_train)

#treereg will be a model
treereg.predict(X_test)

y_pred = treereg.predict(X_test)
#check accuracy, zip and make a
min(zip(testing_error_rate, k_range))
```

```
(0.7900000000000004, 5)
```

	feature	importance
0	Attack	0.174248
3	Sp.Atk	0.173892
5	Speed	0.168631
2	HP	0.165213
1	Defense	0.163291
4	Sp.Def	0.154725