WORCESTER POLYTECHNIC INSTITUTE

CS 539 Final Report

Pokémon for Machine Learning

SUBMITTED BY

Enbo Tian

Date Submitted : 4/29/2022
Date Completed : 4/29/2022
Course Instructor : Prof. Ali

Literature Review:

This project is considered the game of *Pokémon*, with developed by Game Freak and published by Nintendo, in 1996. Pokémon became more and more popular over time, with that Nintendo eventually produced animated TV shows, movies, trading card games, and various comics. The game *Pokémon Go* got a favorable reception. When Switch was put on sale in 2017, *Pokémon* on Switch and *Pokémon Go* got popular again, since the data on Switch and Phone can be connected.

In this project, I considering the data of Pokémon, which is focused on the stats and features of the Pokémon in the RPGs. Seven generations of Pokémon are given in this dataset. All in all, this dataset does not include the data corresponding to the last generation, since the databased when the seventh generation was not released yet. This database is a modification extension of the database "721 Pokémon with stats" by Alberto Barriadas (https://www.kaggle.com/abcsds/pokemon), which does not include in the latest generation either.

Data Source:

•Type_1. Primary type of the Pokémon. It is related the nature, with its lifestyle and with the movements it can learn for the fighting time. This categorical value can take 18 different values: Bug, Dark, Dragon, Electric, Fairy, Fighting, Fire, Flying, Ghost, Grass, Ground, Ice, Normal, Poison, Psychic ,Rock, Steel, and Water.

• Type_2. Pokémon can have two types, but not all of them do. The possible values this secondary type can take are the same than the variable Type_1.

•Total. The sum of all the base battle stats of a Pokémon. It should be a good indicator of the overall strength of a Pokémon. It is the sum of the next six variables. Each of them represents a base battle stat. All the

Battle stats are continuous yet integer variables, i.e. the number of values they can take is infinite in theory, or just very big in the practice.

• HP. Base health points of the Pokémon. The bigger it is, the longer the Pokémon will be able to stay in a fight before they faint and leave the combat.

• Attack. Base attack of the Pokémon. The bigger it is, the more damage its physical attacks will deal to the enemy Pokémon.

• Defense. Base defense of the Pokémon. The bigger it is, the less damage it will receive when being hit by a physical attack.

• Sp_Atk. Base special attack of the Pokémon. The bigger it is, the more damage its special attacks will deal to the enemy Pokémon.

• Sp_Def. Base special defense of the Pokémon. The bigger it is, the less damage it will receive when being hit by a special attack.

• Speed. Base speed of the Pokémon. The bigger it is, the more times the Pokémon will be able to attack to the enemy.

• Generation. The generation where the Pokémon was released. It is an integer between 1 and 6, so it is a

Numerical discrete variable. It could let us analyze the development or the growth of the game through the years.

• is_Legendary. Boolean indicating whether the Pokémon is legendary or not. Legendary Pokémon tend to be stronger, to have unique abilities, to be hard to find, and to be even harder to catch.

• Color. Color of the Pokémon according to the Pokédex. The Pokédex distinguishes between ten colors: Black, Blue, Brown, Green, Grey, Pink, Purple, Red, White, and Yellow.

• hasGender. Boolean indicating the Pokémon can be classified as male or female.

• Pr_Male. In case the Pokémon has Gender, the probability of its being male. The probability of being female is, of course, 1 minus this value. Like Generation, this variable is numerical and discrete, because although it is the probability of the Pokémon to appear as a female or male in the nature, it can only take 7 values: 0, 0.125, 0.25, 0.5, 0.75, 0.875, and 1.

• Egg_Group_1. Categorical value indicating the egg group of the Pokémon. It is related with the race of the Pokémon, and it is a determinant factor in the breeding of the Pokémon. Its 15 possible values are: morphous, Bug, Ditto, Dragon, Fairy, Field, Flying, Grass, Human-Like, Mineral, Monster, Undiscovered, water_1, Water_2, and Water_3.

• Egg_Group_2. Similarly, to the case of the Pokémon types, Pokémon can belong to two egg groups.

• hasMegaEvolution. Boolean indicating whether a Pokémon can mega-evolve or not. Mega-evolving is property that some Pokémon have and allows them to change their appearance, types, and stats during a combat into a much stronger form.

• Height_m. Height of the Pokémon according to the Pokédex, measured in meters. It is a numerical continuous variable.

• Weight_kg. Weight of the Pokémon according to the Pokédex, measured kilograms. It is also a numerical continuous variable.

• Catch_Rate. Numerical variable indicating how easy is to catch a Pokémon when trying to capture it to make it part of your team. It is bounded between 3 and 255. The number of different values it takes is not too high notwithstanding, we can consider it is a continuous variable.

• Body_Style. Body style of the Pokémon according to the Pokédex. 14 categories of body style are specified: bipedal_tailed, bipedal_tailless, four_wings, head_arms, head_base, head_legs, head_only, insectoid,

```
[30]: df= pd.read_csv("D:\CS 539\project\pokemon_alopez247.csv")
df.head()
```

[30]:

	Number	Name	Type_1	Type_2	Total	HP	Attack	Defense	Sp_Atk	Sp_Def	...	Color	hasGender	Pr_Male	Egg_Group_1	Egg_Group_2	hasMegaEvolution	Height_m
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	...	Green	True	0.875	Monster	Grass	False	0.71
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	...	Green	True	0.875	Monster	Grass	False	0.99
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	...	Green	True	0.875	Monster	Grass	True	2.01
3	4	Charmander	Fire	NaN	309	39	52	43	60	50	...	Red	True	0.875	Monster	Dragon	False	0.61
4	5	Charmeleon	Fire	NaN	405	58	64	58	80	65	...	Red	True	0.875	Monster	Dragon	False	1.09

5 rows × 23 columns

multiple_bodies, quadruped, serpentine_body, several_limbs, two_wings, and with_fins.

2	hasMegaEvolution	Height_m	Weight_kg	Catch_Rate	Body_Style
s	False	0.71	6.9	45	quadruped
s	False	0.99	13.0	45	quadruped
s	True	2.01	100.0	45	quadruped
n	False	0.61	8.5	45	bipedal_tailed
n	False	1.09	19.0	45	bipedal_tailed

```
df.info()
```

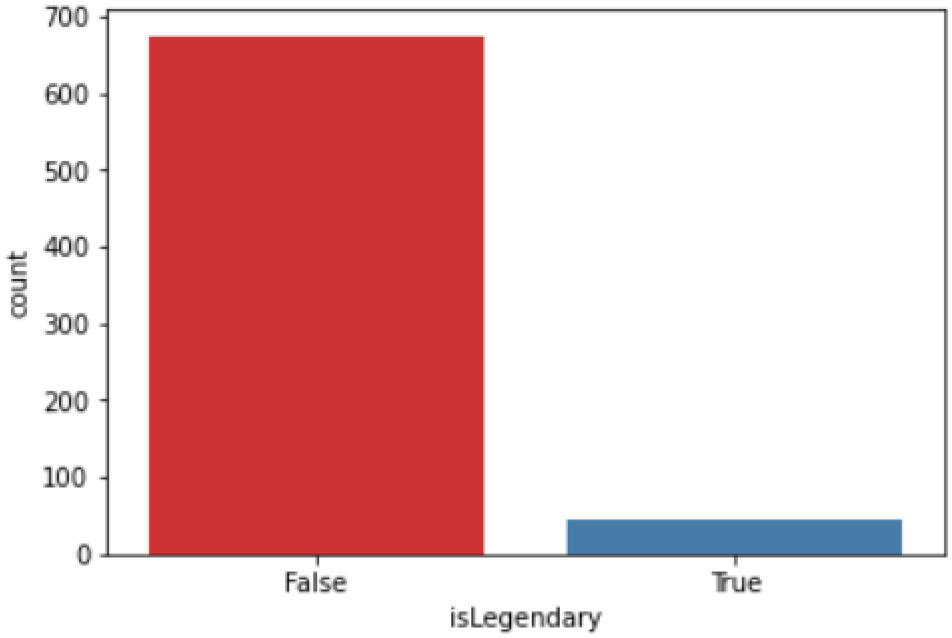
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 721 entries, 0 to 720
Data columns (total 23 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Number             721 non-null    int64  
 1   Name               721 non-null    object 
 2   Type_1             721 non-null    object 
 3   Type_2             350 non-null    object 
 4   Total              721 non-null    int64  
 5   HP                 721 non-null    int64  
 6   Attack             721 non-null    int64  
 7   Defense            721 non-null    int64  
 8   Sp_Atk             721 non-null    int64  
 9   Sp_Def             721 non-null    int64  
 10  Speed              721 non-null    int64  
 11  Generation         721 non-null    int64  
 12  isLegendary        721 non-null    bool   
 13  Color              721 non-null    object 
 14  hasGender          721 non-null    bool   
 15  Pr_Male            644 non-null    float64
 16  Egg_Group_1        721 non-null    object 
 17  Egg_Group_2        191 non-null    object 
 18  hasMegaEvolution   721 non-null    bool   
 19  Height_m           721 non-null    float64
 20  Weight_kg          721 non-null    float64
 21  Catch_Rate         721 non-null    int64  
 22  Body_Style         721 non-null    object 
dtypes: bool(3), float64(3), int64(10), object(7)
memory usage: 114.9+ KB
```

Data Explore:

To figure out the number Legendary Pokémon:

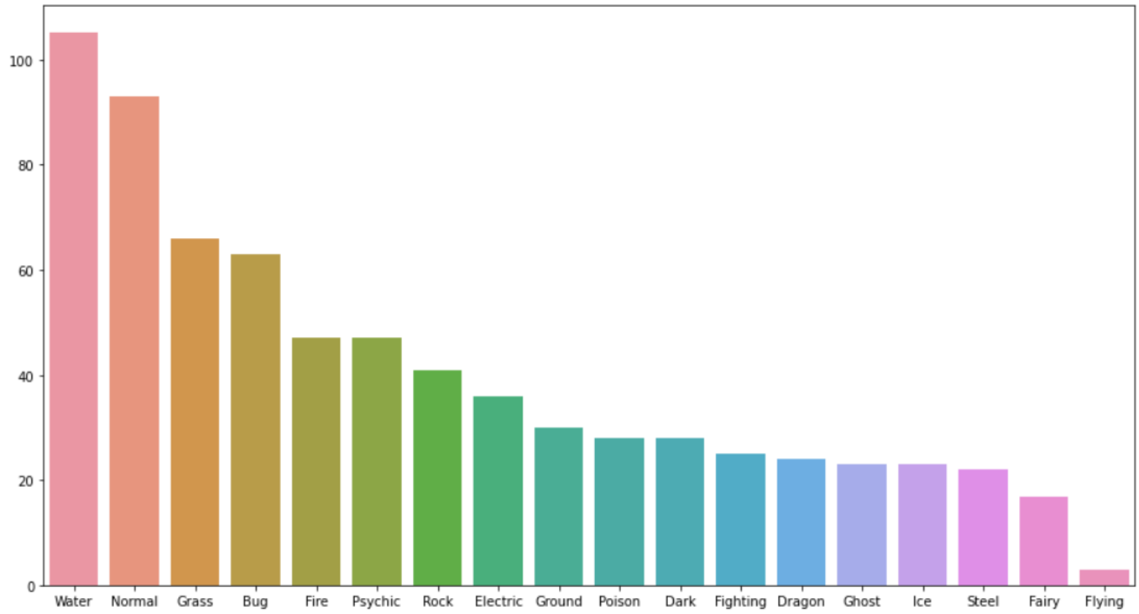
```
print(df['isLegendary'].value_counts())
sns.countplot(x='isLegendary', data=df, palette='Set1')
plt.show()
```

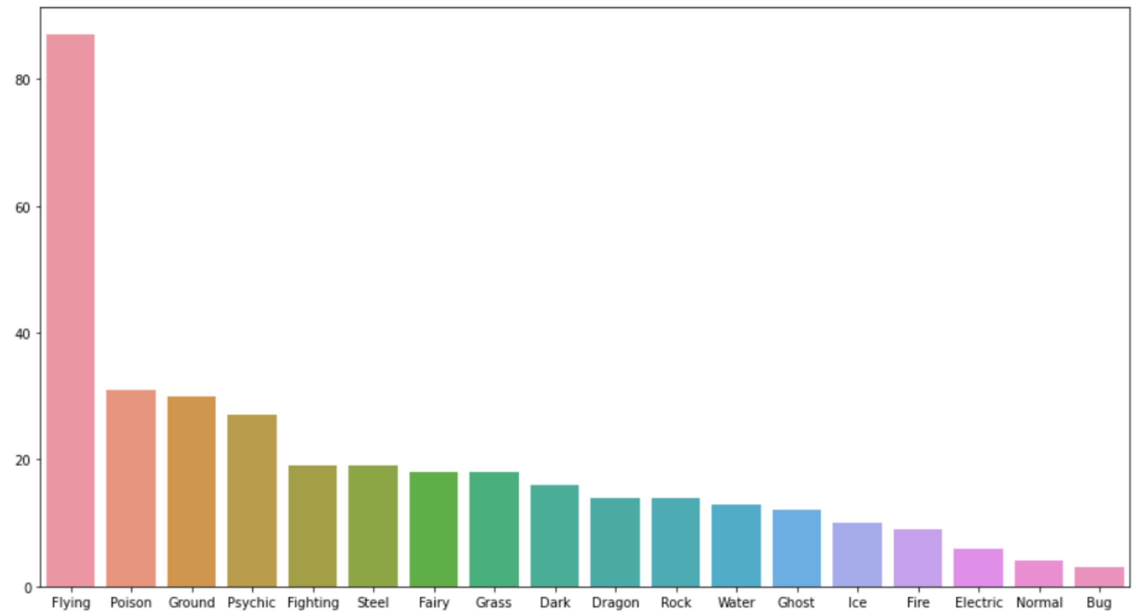
```
False    675
True      46
Name: isLegendary, dtype: int64
```



Type1 and type 2 Pokémon:

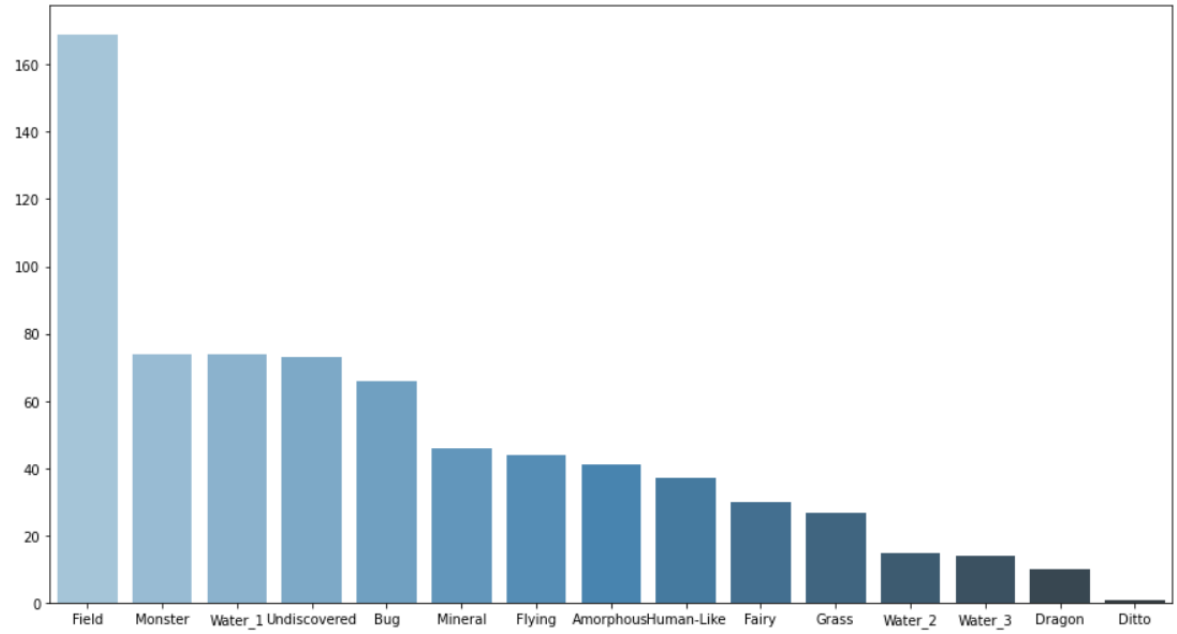
```
type1 = df['Type_1'].value_counts()

type2 = df['Type_2'].value_counts()

fig, (ax1,ax2) = plt.subplots(nrows=2)

fig.set_size_inches(15,18)
# using seaborn barplot to visualize Type_1 and Type_2
sns.barplot(x=type1.index,y=type1.values, ax= ax1)
sns.barplot(x=type2.index,y=type2.values, ax= ax2)
```

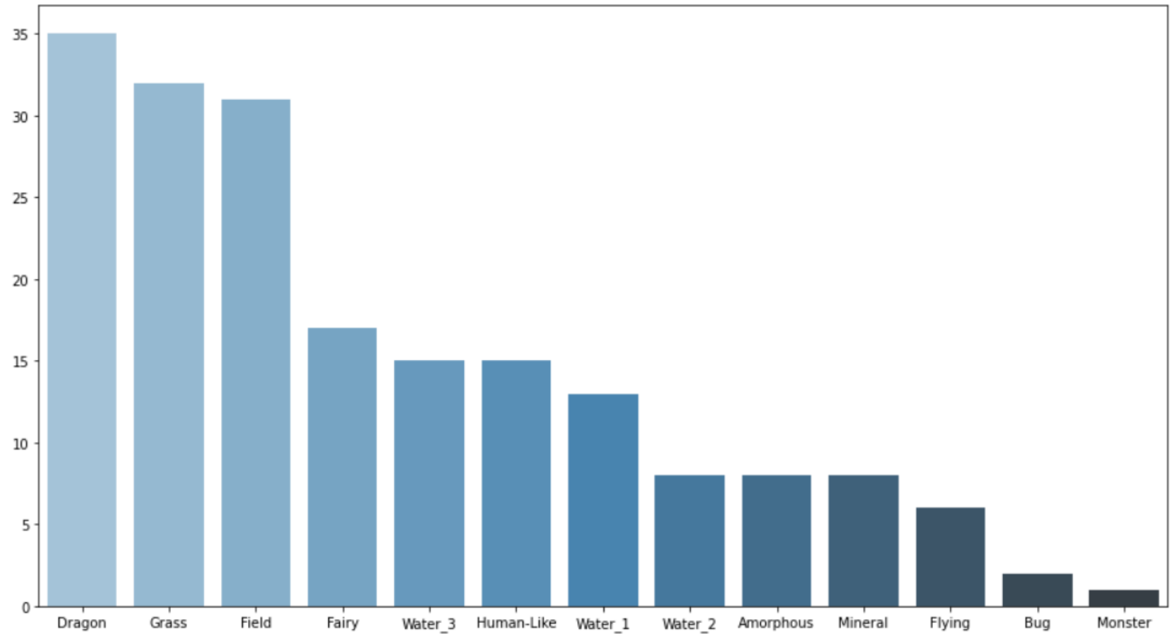




Eggs in each type:

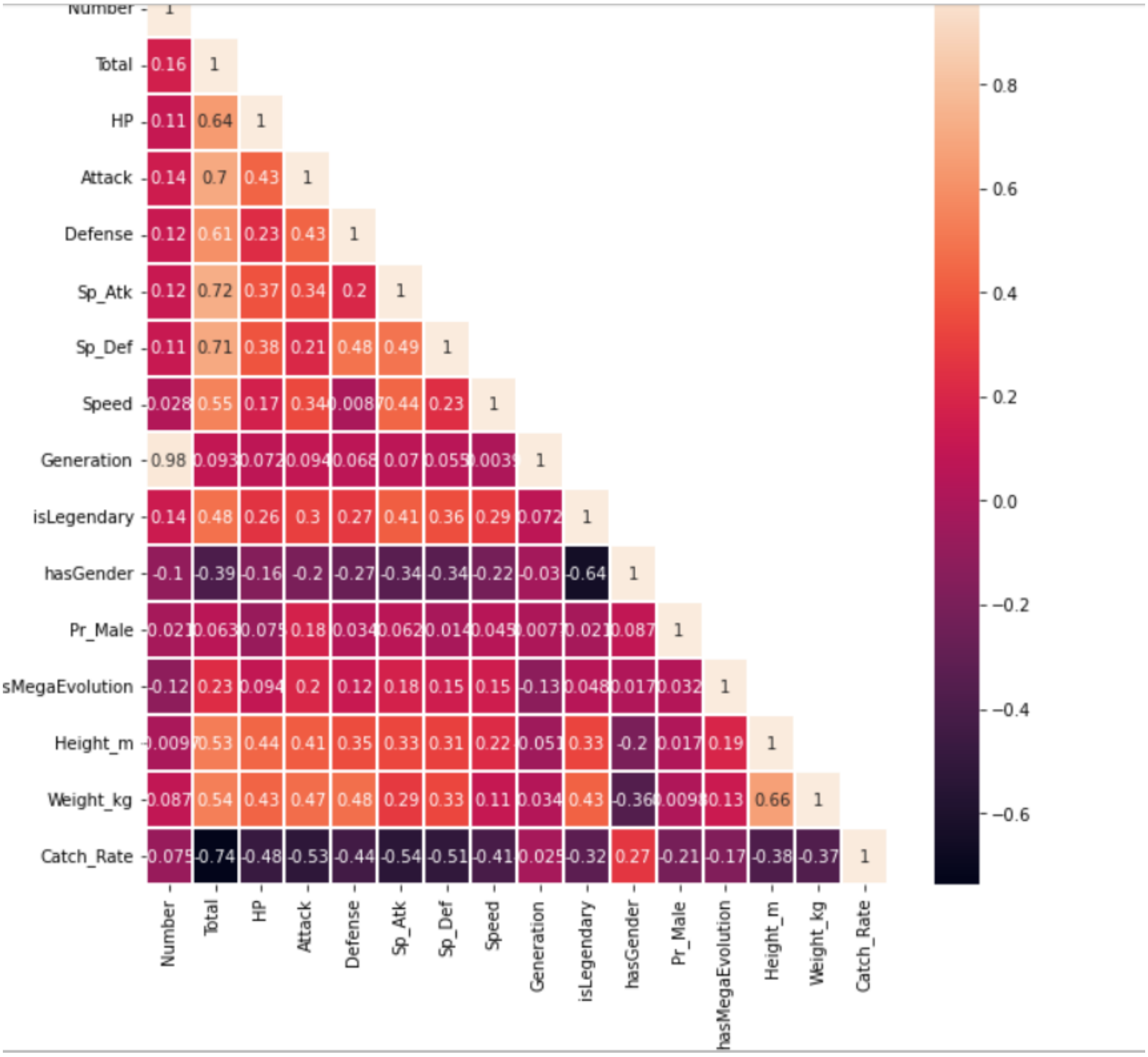
```
egg1 = df['Egg_Group_1'].value_counts()

egg2 = df['Egg_Group_2'].value_counts()

fig, (ax1,ax2) = plt.subplots(nrows=2)

fig.set_size_inches(15,18)
# using seaborn barplot to visualize Type_1 and Type_2
sns.barplot(x=egg1.index,y=egg1.values, palette="Blues_d", ax= ax1)
sns.barplot(x=egg2.index,y=egg2.values, palette="Blues_d",ax= ax2)
```





Heatmap:

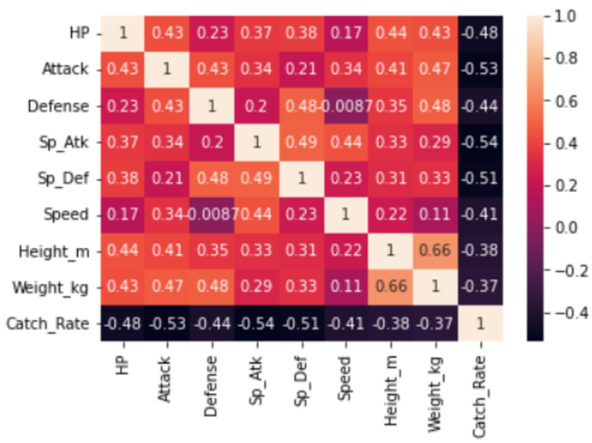
```
37]: mask = np.array(df.corr())
     mask[np.tril_indices_from(mask)] = False

     plt.figure(figsize=(10,10))
     sns.heatmap(df.corr(), mask=mask, annot=True,linewidths=.3)
```



Reduce no related variable:

```
]: pokemon = df[["HP","Attack","Defense","Sp_Atk","Sp_Def","Speed","Height_m","Weight_kg","Catch_Rate"]]
   corr = pokemon.corr()
   sns.heatmap(corr,annot = True)
```

```
]: <AxesSubplot:>
```



```
Type1 = list(df['Type_1'].unique())
for type1 in Type1:
  print (type1,":", np.mean(df[df['Type_1'] == type1].Total))
```

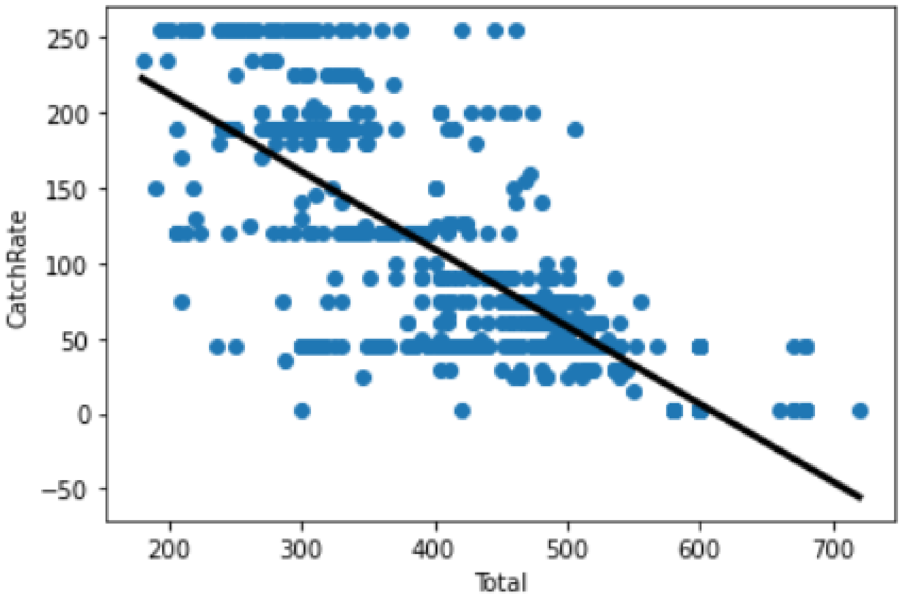
```
Grass : 409.56060606060606
Fire : 443.02127659574467
Water : 417.2
Bug : 365.12698412698415
Normal : 392.16129032258067
Poison : 399.14285714285717
Electric : 420.69444444444446
Ground : 421.0
Fairy : 413.1764705882353
Fighting : 404.36
Psychic : 442.48936170212767
Rock : 437.8048780487805
Ghost : 423.6521739130435
Ice : 427.0869565217391
Dragon : 501.9583333333333
Dark : 434.75
Steel : 464.90909090909093
Flying : 453.3333333333333
```

Methodology and Results

In Pokémon Go, the main game is to catch Pokémon with using different balls. I considered that the catch rate should be related to the different data and type for Pokémon. Thus, the first thing to do is to fit a Linear regression for Pokémon where the response is catch rate, and other factors are my x variables.

```
#compare with linear regression
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
x = np.array(df.loc[:, 'Total']).reshape(-1,1)
y = np.array(df.loc[:,'Catch_Rate']).reshape(-1,1)
reg = LinearRegression()
predict_space = np.linspace(min(x), max(x)).reshape(-1,1)
reg.fit(x,y)
predicted = reg.predict(predict_space)
print('R^2 score: ',reg.score(x, y))
plt.plot(predict_space, predicted, color='black', linewidth=3)
plt.scatter(x=x,y=y)
plt.xlabel('Total')
plt.ylabel('CatchRate')
plt.show()
#linear regression is not significant
```

```
R^2 score:  0.5450568400784921
```



However, from the result of Linear Regression, we can see that the R^2 score is only about 0.54, and the plot is not following a linear line. Thus, Linear Regression is not a good fitted here.

Thus, we would like to make a classifier for the variables of “HP”, “Attack”, “Defense”, “Speed”, “Sp_Atk” and “Sp_Def”. These variables are shown the effectiveness of Pokémon. Also, they are the important evidence to see if the Pokémon is legendary or not.

I would like to try KNN method first with K =3 neighbors since most of the Pokémon have 3 kinds of form. Each time of the evolvement of Pokémon, the property of Pokémon would increase a lot.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
knn = KNeighborsClassifier(n_neighbors = 3)
x1 = df.loc[:,['HP','Attack','Defense','Speed','Sp_Atk','Sp_Def']]
y1 = df.loc[:,'Type_1']
x_train, x_test, y_train, y_test = train_test_split(x1, y1, test_size = 0.3)
knn.fit(x_train,y_train)
prediction = knn.predict(x_test)
print('With KNN (K=3) accuracy is: ',knn.score(x_test,y_test))
#not siginificant
```

```
With KNN (K=3) accuracy is:  0.17511520737327188
```

Confusion matrix:

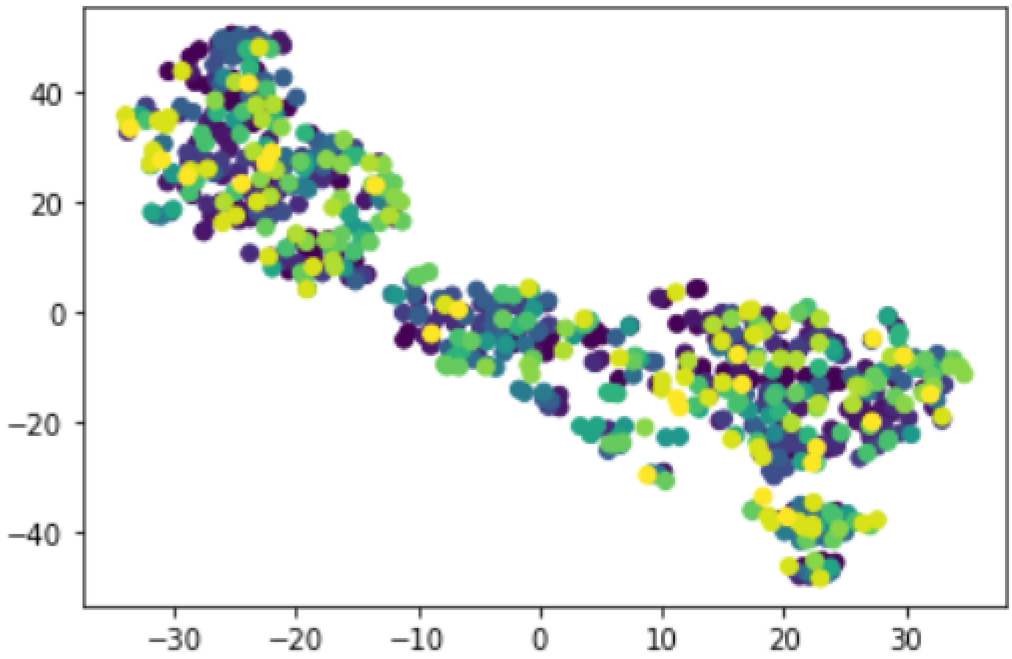
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_train, knn.predict(x_train))
```

```
[50]: array([[42,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  2,
         0,  0],
       [ 3, 12,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  1],
       [ 0,  4, 14,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0],
       [ 4,  1,  0, 19,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0],
       [ 1,  0,  1,  0, 10,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,
         0,  0],
       [ 3,  1,  0,  1,  1,  9,  1,  0,  0,  0,  0,  0,  1,  0,  0,  0,
         0,  0],
       [ 1,  1,  2,  6,  0,  3, 18,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0],
       [ 1,  0,  0,  1,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0],
       [ 2,  2,  1,  2,  1,  1,  0,  0,  7,  0,  0,  0,  0,  0,  0,  0,
         0,  0],
       [ 7,  2,  0,  6,  2,  0,  4,  0,  1, 23,  0,  0,  0,  0,  2,  0,
         0,  1],
       [ 3,  2,  3,  1,  1,  3,  1,  0,  1,  0,  4,  0,  0,  0,  0,  1,
         2,  0],
       [ 1,  0,  0,  2,  3,  1,  3,  0,  1,  4,  1,  1,  1,  0,  0,  0,
         1,  0],
       [ 5,  2,  2,  5,  0,  3,  0,  0,  1,  4,  1,  3, 32,  0,  0,  0,
         0,  1],
       [ 1,  0,  1,  0,  0,  1,  1,  0,  2,  3,  1,  1,  3,  1,  0,  0,
         0,  1],
       [ 5,  1,  1,  5,  2,  0,  3,  0,  1,  2,  1,  1,  1,  0, 10,  0,
         0,  0],
       [ 1,  2,  4,  2,  1,  3,  0,  0,  1,  2,  0,  3,  1,  1,  0, 12,
         2,  0],
       [ 0,  0,  2,  0,  0,  0,  0,  0,  3,  1,  0,  0,  0,  0,  1,  1,
         9,  0],
       [ 9,  3,  2,  3,  7,  1,  4,  0,  1, 10,  1,  2,  2,  0,  2,  1,
         0, 22]], dtype=int64)
```

As a result, the accuracy of KNN is 0.175. This accuracy is not high enough. I would like to try some other method to compare the accuracy. So I make a TSNE for the status of Pokémon.

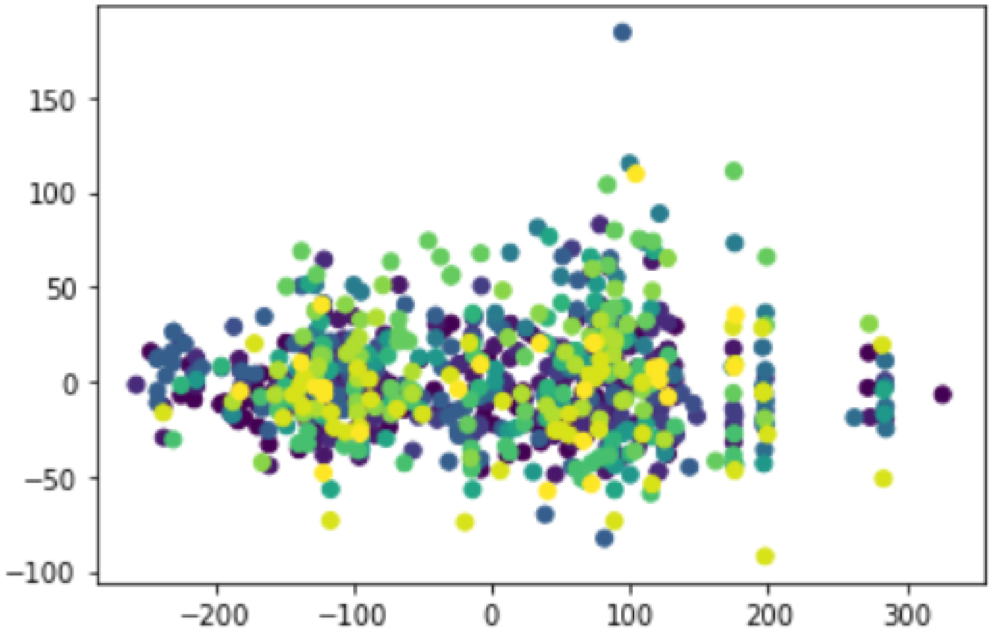
```python
#TSNE on the ordered_vals dataframe scatterplot
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

tsne = TSNE(n_components = 2)
tsne_vals = tsne.fit_transform(ordered_stats)
x = []
y = []
for i in range(720):
    x.append(tsne_vals[i][0])
    y.append(tsne_vals[i][1])

fixed_ordered_labels = []
for i in range(720) :
    fixed_ordered_labels.append(ordered_labels[i][0])
plt.scatter(x,y,c=fixed_ordered_labels,cmap='viridis')
```

```
<matplotlib.collections.PathCollection at 0x21e22f81fd0>
```



```python
from sklearn.decomposition import PCA
pca = PCA(2)
pca_vals = pca.fit_transform(ordered_stats)
x = []
y = []
for i in range(720):
    x.append(pca_vals[i][0])
    y.append(pca_vals[i][1])
plt.scatter(x,y,c=fixed_ordered_labels,cmap='viridis')
```

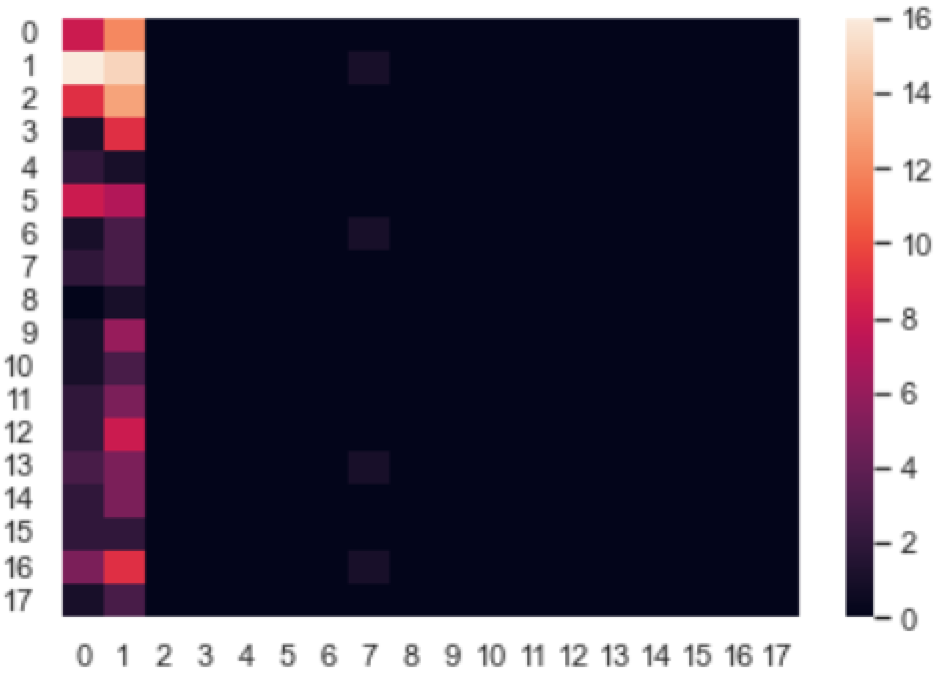
```
<matplotlib.collections.PathCollection at 0x21e22fed640>
```



It is hard to see the difference between each type. And if we test the accuracy, the accuracy would be very low, just like the KNN method. Here, I tried a logistic method.

```python
from sklearn.decomposition   import PCA
from sklearn.linear_model    import LogisticRegression
from sklearn.metrics         import confusion_matrix
from sklearn.model_selection import train_test_split
import seaborn as sns; sns.set_theme()

tsne_data = pd.DataFrame(tsne_vals)
y_data = fixed_ordered_labels
# Split the data.
x_train, x_test, y_train, y_test = train_test_split(
    tsne_data,
    y_data,
    test_size=.25
)

# Fit the logistic regression model.
lr_model = LogisticRegression(solver='liblinear', random_state=0).fit(x_train,y_train)
# Get predictions and their confusion matrix.
y_predict = lr_model.predict(x_test)
matrix = confusion_matrix(y_test, y_predict)
M = confusion_matrix(y_test, y_predict)
n_samples = len(y_test)
print(M)
print('Accuracy:  %.2f' % ((M[0][0] + M[1][1] + M[2][2]+ M[3][3] + M[4][4] + M[5][5] +M[6][6] + M[7][7] + M[8][8]
sns.heatmap(M)
```

```
[[ 8 12  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [16 15  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0]
 [ 9 13  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 1  9  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 8  7  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 1  3  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0]
 [ 2  3  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 1  6  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 1  3  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 2  5  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 2  8  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 3  5  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0]
 [ 2  5  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 2  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  9  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0]
 [ 1  3  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
Accuracy:  0.13
```



The accuracy is still low.

I think a better is to just consider the "is legendary" and "catch rate", since the status of Pokémon is somehow included in the status of "is legendary".

We started with logistic regression again to predict a legendary Pokémon.

```
clf = LogisticRegression()
clf.fit(X_train, y_train)
y_pred_log_reg = clf.predict(X_test)
acc_log_reg = round( clf.score(X_train, y_train) * 100, 2)
print(str(acc_log_reg) + ' percent')
```

```
94.45 percent
```

We now have the accuracy with 94.45, which looks more normal than what we got before.

We can now try some more method with prediction and check the accuracy now. Here are the methods:

k-Nearest Neighbors

```
clf = KNeighborsClassifier(n_neighbors = 3)
clf.fit(X_train, y_train)
y_pred_knn = clf.predict(X_test)
acc_knn = round(clf.score(X_train, y_train) * 100, 2)
print (acc_knn)
```

```
95.67
```

Support Vector Machine (SVM)

```
clf = SVC()
clf.fit(X_train, y_train)
y_pred_svc = clf.predict(X_test)
acc_svc = round(clf.score(X_train, y_train) * 100, 2)
print (acc_svc)
```

```
92.89
```

Decision Tree

```
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
y_pred_decision_tree = clf.predict(X_test)
acc_decision_tree = round(clf.score(X_train, y_train) * 100, 2)
print (acc_decision_tree)
```

```
98.79
```

Random Forest

```
clf = RandomForestClassifier(n_estimators=100)
clf.fit(X_train, y_train)
y_pred_random_forest = clf.predict(X_test)
acc_random_forest = round(clf.score(X_train, y_train) * 100, 2)
print (acc_random_forest)
```

```
98.79
```

Gaussian Naive Bayes

```
clf = GaussianNB()
clf.fit(X_train, y_train)
y_pred_gnb = clf.predict(X_test)
acc_gnb = round(clf.score(X_train, y_train) * 100, 2)
print (acc_gnb)
```

```
64.12
```

Stochastic Gradient Descent (SGD)

```
clf = SGDClassifier(max_iter=5, tol=None)
clf.fit(X_train, y_train)
y_pred_sgd = clf.predict(X_test)
acc_sgd = round(clf.score(X_train, y_train) * 100, 2)
print (acc_sgd)
```

```
92.03
```

Conclusion

In conclusion, most of these methods is good, with more than 90 percent. However, Gaussian Naïve Bayes is not good with a 0.6412. Random Forest and Decision Tree have the highest accuracy score with 0.9879. Between the two, we choose Random Forest Classifier as it has the ability to limit overfitting as compared to Decision Tree classifier.