Regression Walkthrough

The purpose of this section is to go through modeling data using python. This will involve two major sections, traditional regressions and machine learning.

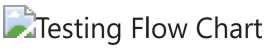
In [81]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
import random as rd
from sklearn import metrics, linear_model, model_selection

# This is just to handle for a warning of copies of df's as opposed to working with the
# Read more here https://stackoverflow.com/questions/42105859/pandas-map-to-a-new-colum
pd.options.mode.chained_assignment = None
```

In [2]:
```
pokemon_df=pd.read_csv('./data/pokemon_data.csv')
```

Regressions

Regressions are a traditional statistical tool that is incredibly useful for take quantitative inputs and outputting a quantitative output. Most often this comes in the form of predicting behavior, which of course is why we're discussing them in modeling.

As a quick reminder from the stats notebook, here is a useful flowchart for determining which statistical tools to use.

 Testing Flow Chart

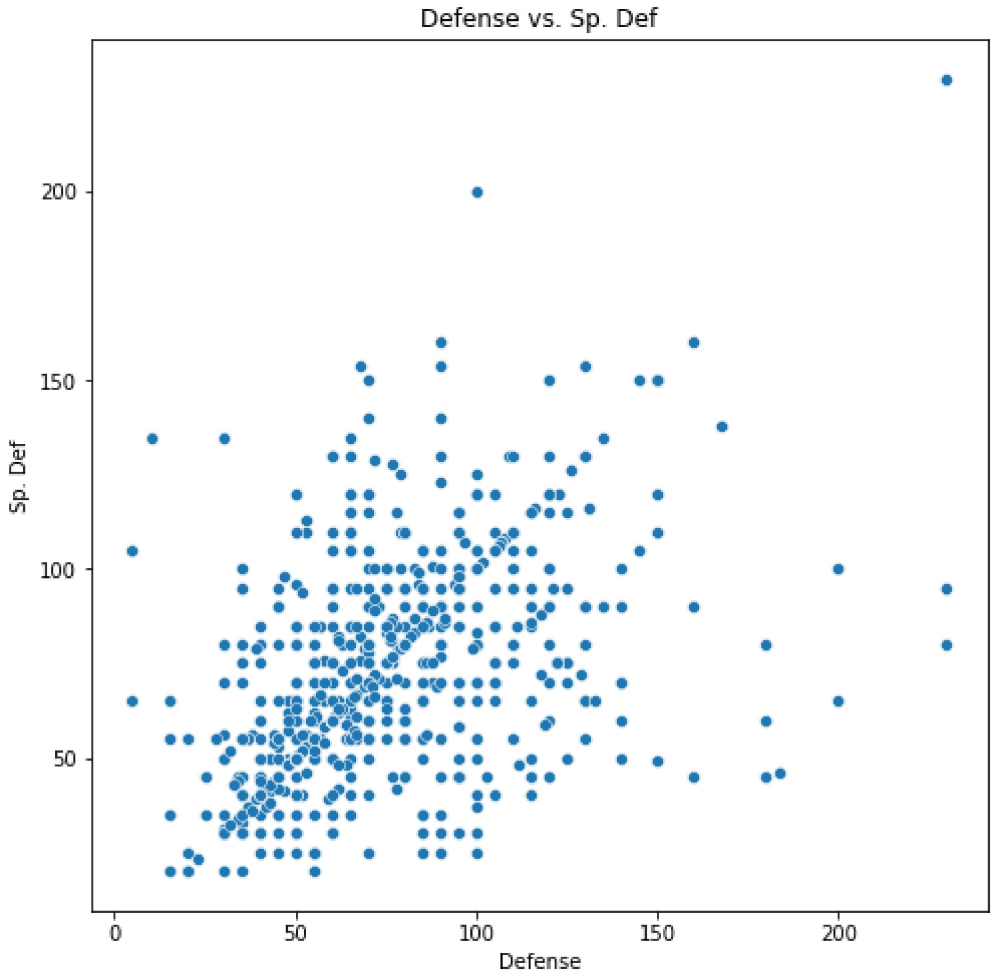
Simple Regression

Simple linear regressions are the most well-known regression type. It simply attempts to understand the relationship between two variables, x (the input) and y (the output). As you can imagine, the simplicity of such a regression lends the regression to working with simple problems.

In our context of pokemon, let's plot the association between defense and special defense and generate a regression. In other words, given the normal defense stat of a particular pokemon, what is the special defense stat?

In [3]:
```
defense_df = pokemon_df[['Defense','Sp. Def']]

plt.figure(figsize=(8,8))
sns.scatterplot(data=defense_df, x='Defense', y='Sp. Def')
plt.title('Defense vs. Sp. Def')
```

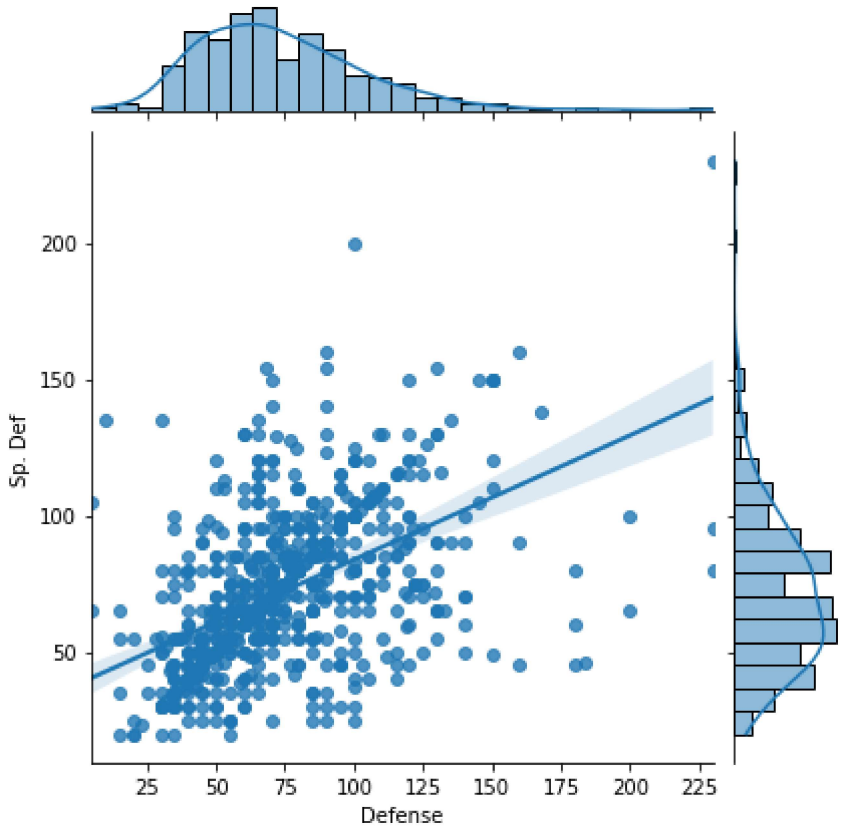
Out[3]:
```
Text(0.5, 1.0, 'Defense vs. Sp. Def')
```



As you can probably make out, it looks as if there is some straight line relationship to the Defense and Special Defense. However generate a regression and a few measures to evaluate how accurate the regression is.

In [4]:
```
# fitting our model
simple_reg=stats.linregress(defense_df.loc[:,'Defense'], defense_df.loc[:,'Sp. Def'])

defense_df.loc[:,'Sp. Def Predicted']=defense_df['Sp. Def'].apply(lambda x : x*simple_r
defense_df.loc[:,'Squared Error']=np.power((defense_df.loc[:,'Sp. Def']-defense_df.loc[

plt.figure(figsize=(8,8))
sns.jointplot(data=defense_df, x='Defense', y='Sp. Def', kind='reg')

print('\nThe formula of our regression is y={:.2f}x+{:.2f}'.format(simple_reg.slope, si
print('The R Squared value of our regression is {:.2f}.'.format(simple_reg.rvalue))
print('The P-value of our regression is {:.2f}.'.format(simple_reg.pvalue))
print('The Root Mean Squared Error of our regression is {:.2f}'.format(np.power(np.mean
print('The Mean Absolute Error of our regression is {:.2f}.\n'.format(metrics.mean_abso
```

```
The formula of our regression is y=0.46x+38.24
The R Squared value of our regression is 0.51.
The P-value of our regression is 0.00.
The Root Mean Squared Error of our regression is 15.16
The Mean Absolute Error of our regression is 11.91.

<Figure size 576x576 with 0 Axes>
```



An R Squared value of 0.51 is fairly interesting. Remember that an R squared value should be interpreted as "R percent of the variability in y can be explained by the variability in x". Additionally, the p-value can be of note. The null hypothesis in this instance is that the coefficient of our model is zero, as in there is no relationship. Additionally, we have a root mean squared error here of about 15, meaning that approximately we can expect a plus or minus 15 points for each value we predict. Mean absolute error is another common metric for evaluating the accuracy of a model, and generally will be fairly close to what our model shows.

As you can imagine, there is a lot of room for improvement in terms of sophistication in modeling.

Multiple Linear Regressions

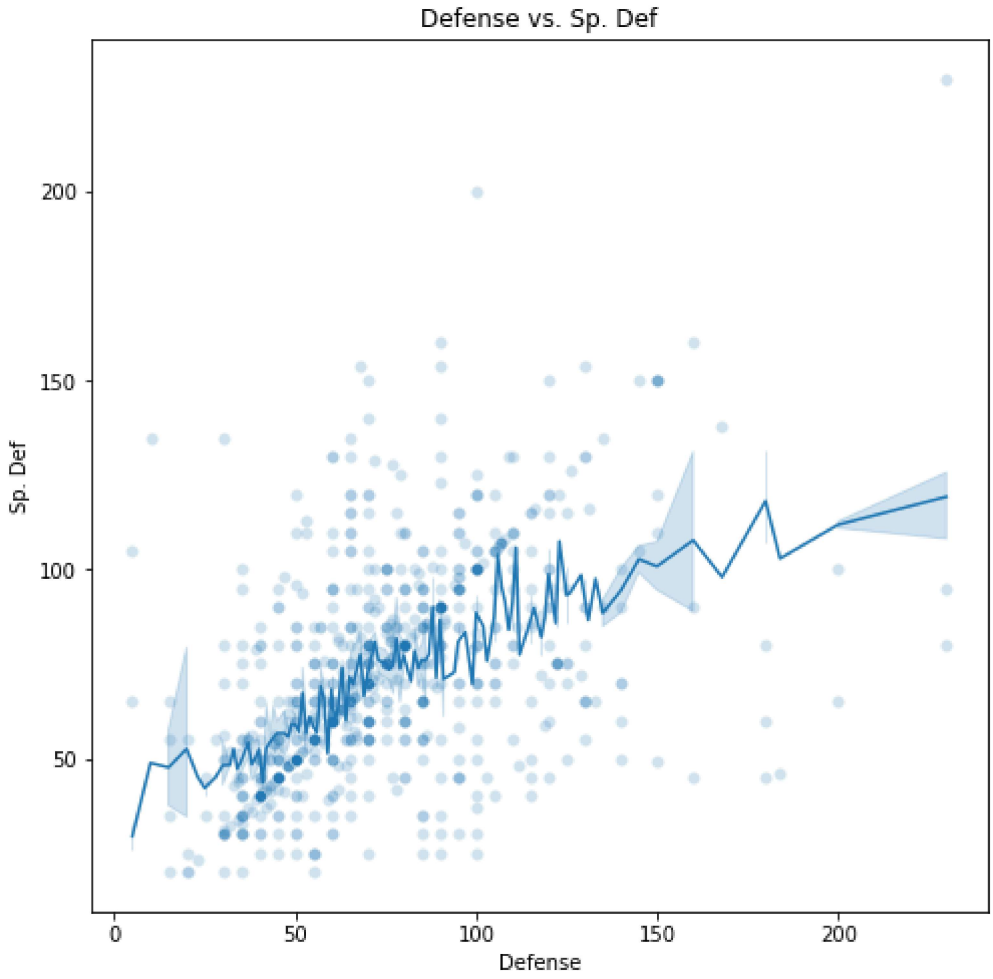
We can expand on the simple linear regression by doing what's called a multiple linear regression. This is functionally the same process as above, however we are now allowing for mutliple independent variables. This still results in a linear model.

In this context, we're now predicting special defemse by taking in to account the pokemon's defense and special attack stats.

In [7]:
```
defense_df = pokemon_df[['Defense','Sp. Atk','Sp. Def']]

X=defense_df[['Defense','Sp. Atk']].to_numpy()
y=defense_df['Sp. Def'].to_numpy()

multi_reg=linear_model.LinearRegression()
multi_reg.fit(X,y)

y_pred=multi_reg.predict(X)
```

```
defense_df.loc[:,'Sp. Def Predicted']=y_pred
defense_df.loc[:,'Squared Error']=np.power((defense_df.loc[:,'Sp. Def']-defense_df.loc[

plt.figure(figsize=(8,8))
sns.scatterplot(data=defense_df, x='Defense', y='Sp. Def', alpha=.2)
sns.lineplot(data=defense_df, x='Defense', y='Sp. Def Predicted')
plt.title('Defense vs. Sp. Def')

print('\nThe formula for our regression is Sp. Defence = {:.2f}*Defence + {:.2f}*Sp. At


print('The R Squared value of our regression is {:.2f}.'.format(multi_reg.score(X,y)))
print('The Root Mean Squared Error of our regression is {:.2f}.\n'.format(np.power(np.m
```

```
The formula for our regression is Sp. Defence = 0.37*Defence + 0.35*Sp. Attack + 18.77.
The R Squared value of our regression is 0.42.
The Root Mean Squared Error of our regression is 21.13.
```



As you can see, the variability of the regression is all over the place, as obviously we're not seeing the third dimension of the special attack stat. This is what is driving the jagged trend line.

Additionally, note that the R Squared value of this model and the root mean squared error are actually worse than our simple linear model. This goes to show that increasing model complexity is not always a gaurantee of increasing model accuracy. Playing with which features should go into a model is a necessary part of designing an accurate model.

In [9]:

```
# sklearn has a function for qucickly evaluating mean square error
```

```
print(np.power(metrics.mean_squared_error(defense_df['Sp. Def'], defense_df['Sp. Def Pr
```

```
21.133759161206385
```

Training and Test Datasets

I'm gonna take a second to talk about splitting up your datasets. When creating models, it is very intuitive to think that we should pass through all of our data to create a good model. However, we must be wary of a concept known as overfitting. Overfitting is when you fit your model too well to the data we have presently, but the model fails to predict the underlying trend when introduced to new data. Therefore, we should always take a subset of data to train our model on.

So what should we do with the other half the model isn't trained on? This will be used as our test dataset. This is the dataset that will be used to evaluate the efficacy of our model. That way, we can truly have an understanding of how well our model is picking up the underlying trends. The concepts of overfitting and splitting your data out are not that relevant with simple linear regressions. However, the become increasingly important as model complexity increases.

In [11]:
```
# Sklearn also has defined function for doing this for us
# train_size is a float to tell how much should be reserved
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, train_size=.8

# Let's reevaluate the multiple linear analysis with the split up data

multi_reg.fit(X_train, y_train)

# Predicting y_values
y_pred=multi_reg.predict(X_test)

# Evaluating RMSE
print('\nRMSE of the Test Data Set is {:.2f}.\n'.format(np.power(metrics.mean_squared_e
```

```
RMSE of the Test Data Set is 23.14.
```

Notice how the RMSE has changed slightly. Obviously, some of this is due to random noise. However, once again, as more complexity is introduced in to the model. The potential for overfitting is always present.

Another concept I'll briefly mention is the concept of cross-validating. Cross validating is simply rotating out which data is used as the training data set and which is the testing data, such that every data point is a training point once and a testing point once. You would then evaluate the mean average error and across all the rotations.

Non-Linear Regressions

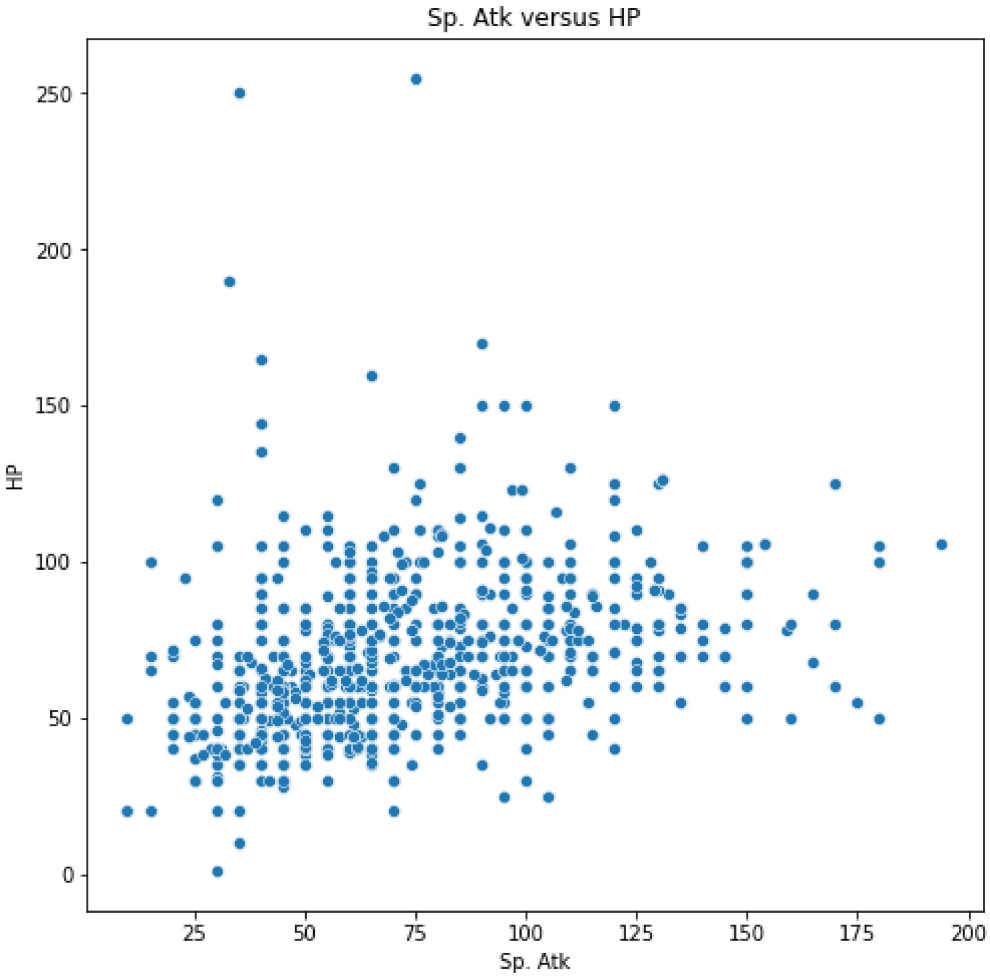
Polynomial Regressions

Polynomial regressions as you likely guessed, involve polynomials to minimize the error. While useful for picking up more unique trends, non-linear regressions can be dangerous, as they can definitely be used to identify patterns that aren't there (i.e. overfitting).

In [25]:
```
```

```
# Let's look at the relationship between Sp. Attack and HP

attack_df = pokemon_df[['Sp. Atk', 'HP']]

plt.figure(figsize=(8,8))
sns.scatterplot(data=attack_df, x='Sp. Atk', y='HP')
plt.title('Sp. Atk versus HP')
```

Out[25]: Text(0.5, 1.0, 'Sp. Atk versus HP')



In [84]:

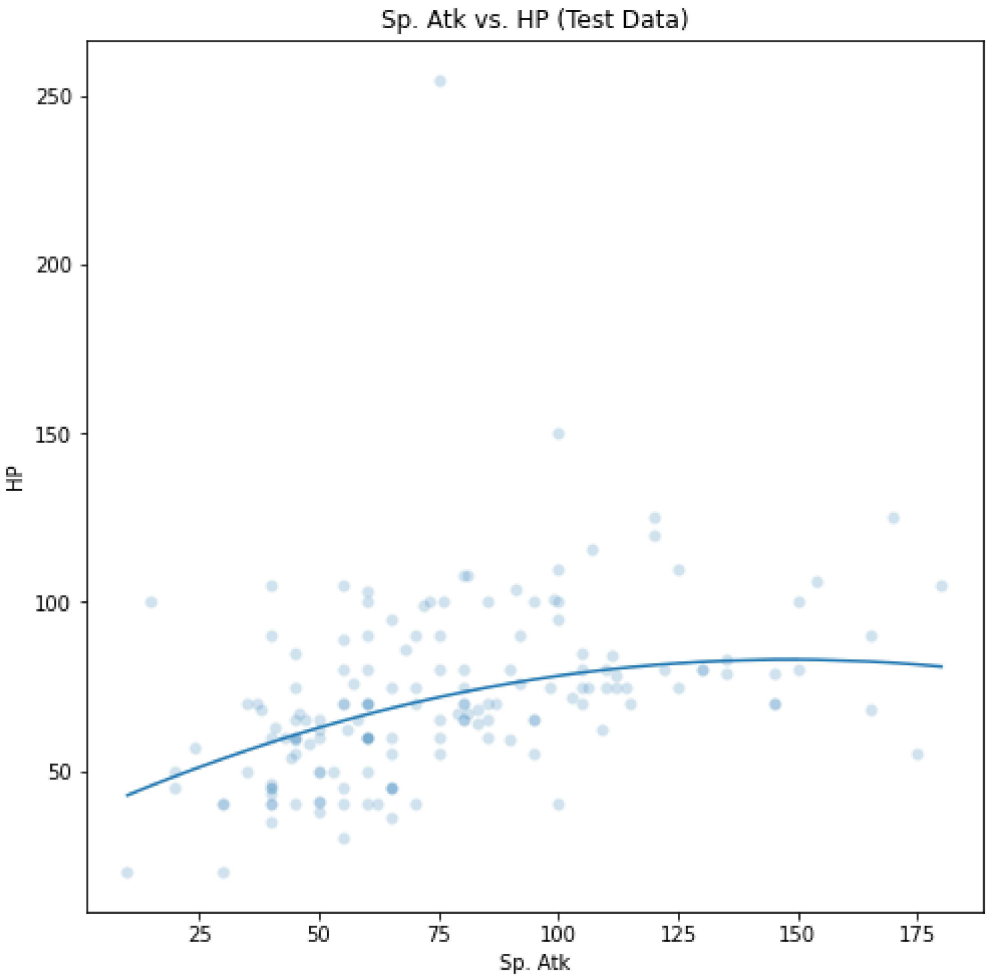
```
# Training our model
X=attack_df['Sp. Atk']
y=attack_df['HP']

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, train_size=.8

# The third arg is how many degrees the polynomial will have
# The higher this is, generally the more accurate however the more susceptable to overf
# you can also replace the second Polynomial with things like Chebyshev regressions
poly_reg = np.polynomial.Polynomial.fit(X_train, y_train, 3)
```

In [96]:

```
y_pred=poly_reg(X_test)
data={'Sp. Atk': X_test, 'HP': y_test, 'HP Predicted':y_pred }

attack_test_df=pd.DataFrame(data=data)

attack_test_df['Absolute Error']=np.absolute(attack_test_df['HP']-attack_test_df['HP Pr
```

```
plt.figure(figsize=(8,8))
sns.scatterplot(data=attack_test_df, x='Sp. Atk', y='HP', alpha=.2)
sns.lineplot(data=attack_test_df, x='Sp. Atk', y='HP Predicted')
plt.title('Sp. Atk vs. HP (Test Data)')
```

Out[96]: Text(0.5, 1.0, 'Sp. Atk vs. HP (Test Data)')



In [95]:
```
print('\nThe mean squared error of our polynomial regression is {:.2f}.'.format(np.mean
print('Our polynomial regression equation is y = {:.2f}x^3 + {:.2f}x^2 + {:.2f}x + {:.2
                                                                    poly_reg.coe
                                                                    poly_reg.coe
                                                                    poly_reg.coe
```

```
The mean squared error of our polynomial regression is 15.72.
Our polynomial regression equation is y = 78.53x^3 + 17.79x^2 + -17.85x + 0.18
```

Spline Functions

Splines are also non-linear functions that are useful building regressions. Effectively, these splines take a piece-wise approach, building out a regression for different segments of the domain. The piece-wise boundaries are called "knots", as the terms "spline" and "knots" are borrowed from ship building.

Let's reuse the context above, deteriming HP from Sp. Atk. Just eye-balling it, it looks the trend line changes direction after 125 on the x-axis, so we'll put a "knot" there.

In [ ]:

```
```

Logistic Regressions

Logistic regressions are intended for use of binary classification. That is to say you feed in quantitative values into the regression and then you spit out a binary yes or no. So for our pokemon status, let's try to predict whether or not a pokemon is a legendary by feeding in it's base stats.

In [98]:

```
log_reg= linear_model.LogisticRegression(random_state=10)

leg_df = pokemon_df[['HP','Attack','Defense','Sp. Atk','Sp. Def', 'Speed','Legendary']]

X=leg_df[['HP','Attack','Defense','Sp. Atk','Sp. Def', 'Speed']].to_numpy()
y=leg_df['Legendary'].to_numpy()

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, train_size=.8

log_reg.fit(X_train, y_train)
```

Out[98]:

```
LogisticRegression(random_state=10)
```

In [99]:

```
# Evaluating our model intuitively

test_df=pd.DataFrame(X_test, columns=['HP','Attack','Defense','Sp. Atk','Sp. Def', 'Spe

test_df['Legendary Actual']=y_test

y_pred=log_reg.predict(X_test)

test_df['Legendary Predicted']=y_pred

test_df.loc[(test_df['Legendary Actual']==True)].head()
```

Out[99]:

	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Legendary Actual	Legendary Predicted
33	80	90	80	130	110	110	True	True
44	106	190	100	154	100	130	True	True
51	125	120	90	170	100	95	True	True
75	80	75	150	75	150	50	True	False
93	125	170	100	120	90	95	True	True

In [100…

```
# Now lets look at truuly how accurate the model was with a confusion matrix

conf=metrics.confusion_matrix(y_test, y_pred, labels=[True, False])

conf_df=pd.DataFrame(conf, columns=['True Predicted','False Predicted'], index=['True A
```

```
conf_df
```

Out[100...

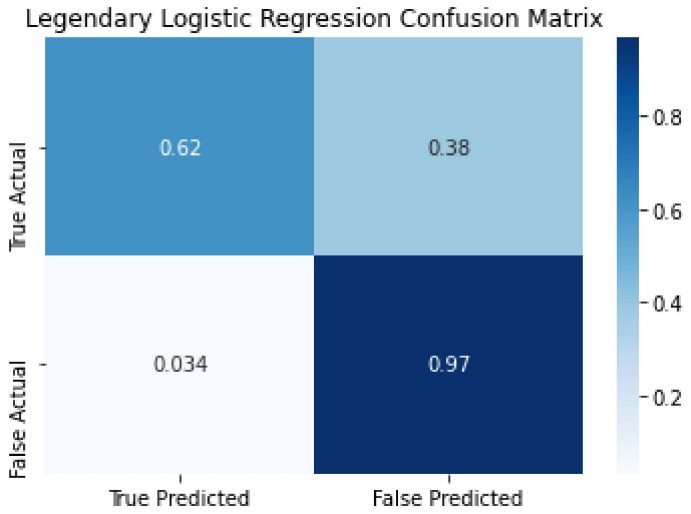
	True Predicted	False Predicted
True Actual	8	5
False Actual	5	142

As you can see, we're acually pretty decent a correctly identifying non-legendary pokemon. Let's go ahead and normalize this to make it a bit more readable.

In [101...

```
conf_norm=metrics.confusion_matrix(y_test, y_pred, labels=[True, False], normalize='tru

conf_norm_df=pd.DataFrame(conf_norm, columns=['True Predicted','False Predicted'], inde

sns.heatmap(conf_norm_df, annot = True, cmap='Blues')
plt.title('Legendary Logistic Regression Confusion Matrix')
```

Out[101... `Text(0.5, 1.0, 'Legendary Logistic Regression Confusion Matrix')`



This confusion matrix represents a normalized view which presents the "Actual" rows as a percentage of total. So in this context, when we have an actual non-legendary pokemon, we are 97% accurate in identifying it as such. However, on the other hand, if we have an actual Legnedary, we are only 62%. Now it's important to keeps two things in mind; we have a relatively small sample size of legendarys (as obviously they are rare) and we need to think about our context.

Taking the second point, we need to think about which we care about more. Do we care about correctly identifying Legendary pokemon? If so, then our model could use some work. Are we interested in the converse? Then we're on the right track.

Additionally, lgistic regression can take in weightings to influence predictions that are rare in these circumstances. So for context, we could effectively force our model to predict True more often, thus likely increasing our correct legendary predictions at the expense of our accuracy of false predictions. In other words, our True Actual X True Predicted value will go up in addition to our False Actual X True Predicted going up as well.