Pokemon Project work report

Exam of Lab of Information Systems and Analytics June 15th 2021

Leonardo Provenzano - 879514

If you want a better view of the report go to the following link:

https://www.notion.so/Pokemon-Project-work-report-7e6a89c2bcb64e2e9df2e3c35f1eb69c

Table of contents:

0. Disclaimer

Disclaimer for very hard core pokemon fans, do not read if you are not:

the majority of the assumptions I made are based on my experience. The objective of this work was also to learn how to use Machine Learning algorithms, so some characteristics of the pokemon were removed, not considered, or made easier to understand for the sake of simplicity. The pokemon world creates too many complications.

I'm not a game freak Nintendo developer, I'm just a student who's looking to have some fun :)

Introduction

1.1 Data Presentation

I have worked with 2 datasets.
The first one is the Complete Pokemon Dataset that provided me with the main data that I have used to build my classifier.
The second one is the Pokedex Dataset, which is an auxiliary dataset that owns a lot of the same information already present in the former dataset, yet it gave me a significant feature that I require to create my evolution stages feature: evolution_chain_id.

THE COMPLETE POKEMON DATASET

https://www.kaggle.com/rounakbanik/pokemon

(pokemon.csv)

- **name:** The English name of the pokemon
- **japanese_name:** The Original Japanese name of the pokemon
- **pokedex_number:** The entry number of the pokemon in the National Pokedex
- **percentage_male:** The percentage of the species that are male. Blank if the pokemon is genderless.
- **type1:** The Primary Type of the pokemon
- **type2:** The Secondary Type of the Pokemon
- **classification:** The Classification of the pokemon as described by the Sun and Moon Pokedex
- **height_m:** Height of the pokemon in meters
- **weight_kg:** The Weight of the pokemon in kilograms
- **capture_rate:** Capture Rate of the pokemon

POKEDEX

https://www.kaggle.com/davidrgp/pokec

(pokemon_evolution.csv)

- **id:** The entry number of the pokemon in the National Pokedex
- **pokemon:** The English name of the pokemon
- **species_id**
- **height:** Height of the pokemon
- **weight:** The Weight of the pokemon
- **base_experience:** The base experience that a pokemon has at Lv. 1
- **type_1:** The Primary Type of the pokemon
- **type_2:** The Secondary Type of the pokemon
- **attack:** The Base Attack of the pokemon
- **defense:** The Base Defense of the pokemon
- **hp:** The Base HP of the pokemon
- **special_attack:** The Base Special Attack of the pokemon
- **special_defense:** The Base Special Attack of the pokemon

- **base*egg*steps:** The number of steps required to hatch an egg of the pokemon
- **abilities:** A stringified list of abilities that the pokemon is capable of having
- **experience_growth:** The Experience Growth of the pokemon (the total points of experience that a pokemon can have at Lv. 100)
- **base_happiness:** Base Happiness of the pokemon
- **against_?:** Eighteen features that denote the amount of damage taken against an attack of a particular type
- **hp:** The Base HP of the pokemon
- **attack:** The Base Attack of the pokemon
- **defense:** The Base Defense of the pokemon
- **sp_attack:** The Base Special Attack of the pokemon
- **sp_defense:** The Base Special Defense of the pokemon
- **speed:** The Base Speed of the pokemon
- **generation:** The numbered generation which the pokemon was first introduced
- **is_legendary:** Denotes if the pokemon is legendary.
- **speed:** The Base Speed of the pokemon
- **ability_1:** The first ability that a pokemon can assume
- **ability_2:** The second ability that a pokemon can have
- **ability_hidden:** The hidden ability of a pokemon
- **color_1:** The first main color of a pokemon
- **color_2:** The second main color of a pokemon
- **color_f:** Variant color if a pokemon is Female
- **egg_group_1:** First egg group of a pokemon
- **egg_group_2:** Second egg group of a pokemon
- **generation_id:** The numbered generation which the pokemon was first introduced
- **evolves_from_species_id:** The pokemon id from which a pokemon evolves
- **evolution_chain_id:** The id of the evolution chain
- **shape:** The name of the shape of a pokemon
- **shape_id:** The id of a shape of a pokemon

1.2 Goal of the Project

Based on the data and the statistics of pokemon, I have decided to:

1. Build a classifier that predicts if a given pokemon is of the type water;
2. Create two new features that take into account additional important characteristics of pokemon: sex and evolution stage.

I wanted to check what model would provide the best accuracy by using the following classifiers:

- Decision Tree
- Random Forest
- Logistic Regression
- KNN
- Support Vector Machine

2. Data Loading and Cleaning

I started by loading the two datasets and cleaning the first one.

I immediately deleted 3 columns that I'm sure are not useful for Machine Learning: japanese_name, classification, and abilities.

There are also other columns, like pokedex_number and generation, that I needed to handle data and create other features, so I have deleted them later on.

While looking at the other features of my data, I had to deal with the following issues:

1. height_m and weight_m have both 20 NaN values:

   I noticed that 20 pokemon were missing height and weight values, so I checked if they were the same 20, they were**.** I substitute the NaN values with the mean weight and height of all the pokemon.

2. type_2 NaN values:

   pokemon have either one type (type_1) or two types, all the pokemon that are monotypes have NaN values for the type_1 feature. For this reason, I substituted NaN values with strings 'None'; so if a Pokemon has one type it will be like it has a second type: "type None".

3. capture_rate expressed as strings and the value for one object is incorrectly labeled:

   The numbers of capture rate were not integers but strings, and for Pokemon n. 774 (Minior) the capture rate was the following string: "30 (Meteorite)255 (Core)". (This is probably caused by the fact that Minior can assume 2 forms during battle, Meteorite and Core - when it is in Meteorite form is harder to catch.)

The first thing that I have done was the conversion from string numbers into integers, and then I have set the capture rate for Minior at 255.

3. Feature Creation

1.3 Sex Feature

I have already had a feature in my main dataset that provided me with information about the sex of a pokemon: percentage_male. This feature tells us the probability of a pokemon being male. Considering that there also exists pokemon that do not have sex, the percentage_male column reports NaN values for them.

I wanted to use this information for Machine Learning but the main problem was that I could not express the probability of a pokemon being male if It does not have sex at all.

As a result, I opted for a simpler solution: creating a feature that reveals if a pokemon has or has not sex: This information is not completely useless as to my personal knowledge the types steel, poison, and psychic have substantial enough numbers of pokemon without sex (I obviously had to confirm my theory by data).

2.3 Evolution Feature

I also wanted to create a feature that considered the evolution stages of each pokemon.

2.3.1 Context

Evolution
Charmander
Charmeleon
Charizard

When some pokemon reach a certain level or meet some specific conditions, they can evolve and become stronger, acquire new abilities and change their characteristics. Pokemon have 3 evolution stages, but not all pokemon evolve, and not all pokemon have 3 evolution stages evolving just one time.

My goal was to create a feature that would have told me if a Pokemon is in its first, second, or third stage (i.e. Charmander → stage 1, Charmeleon → stage 2, Charizard → stage 3).

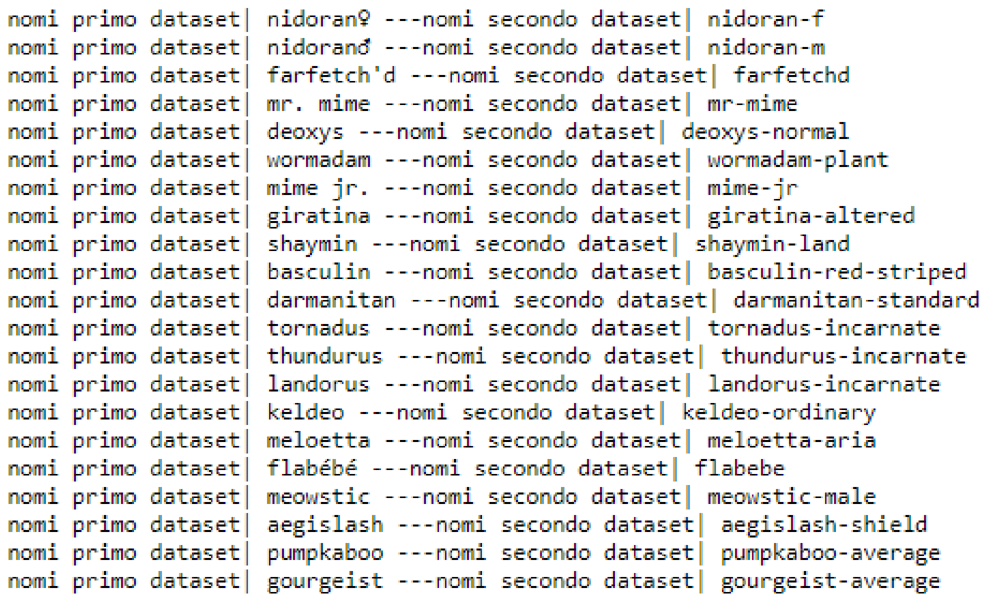
2.3.2 Cleaning of the second Dataset

First, I had to manage the second dataset "pokemon_evolution.csv" and look at the differences with the first one: "pokemon.csv":

1. In the second dataset, pokemon go up to Generation 6, while in the first one they reach Generation 7;
2. From the last pokemon of Generation 6 - Volcanion - up to the last object, the second dataset had accounted for a lot of alternative forms (mainly mega-evolutions);
3. Some pokemon had differences in the strings that expressed their names.

To deal with the first two differences, I simply cut my first dataset at the index position 720 - Volcanion - so that I could start working with pokemon up to Generation 6.

Through a simple code that I created, I have found some differences in the name of some pokemon:



```
nomi primo dataset| nidoran♀ ---nomi secondo dataset| nidoran-f
nomi primo dataset| nidoran♂ ---nomi secondo dataset| nidoran-m
nomi primo dataset| farfetch'd ---nomi secondo dataset| farfetchd
nomi primo dataset| mr. mime ---nomi secondo dataset| mr-mime
nomi primo dataset| deoxys ---nomi secondo dataset| deoxys-normal
nomi primo dataset| wormadam ---nomi secondo dataset| wormadam-plant
nomi primo dataset| mime jr. ---nomi secondo dataset| mime-jr
nomi primo dataset| giratina ---nomi secondo dataset| giratina-altered
nomi primo dataset| shaymin ---nomi secondo dataset| shaymin-land
nomi primo dataset| basculin ---nomi secondo dataset| basculin-red-striped
nomi primo dataset| darmanitan ---nomi secondo dataset| darmanitan-standard
nomi primo dataset| tornadus ---nomi secondo dataset| tornadus-incarnate
nomi primo dataset| thundurus ---nomi secondo dataset| thundurus-incarnate
nomi primo dataset| landorus ---nomi secondo dataset| landorus-incarnate
nomi primo dataset| keldeo ---nomi secondo dataset| keldeo-ordinary
nomi primo dataset| meloetta ---nomi secondo dataset| meloetta-aria
nomi primo dataset| flabébé ---nomi secondo dataset| flabebe
nomi primo dataset| meowstic ---nomi secondo dataset| meowstic-male
nomi primo dataset| aegislash ---nomi secondo dataset| aegislash-shield
nomi primo dataset| pumpkaboo ---nomi secondo dataset| pumpkaboo-average
nomi primo dataset| gourgeist ---nomi secondo dataset| gourgeist-average
```

Manually checking, we can see that the pokemon are the same so I did not make any changes.

The last thing that I have done was dropping all the columns that I didn't need, keeping only the following features: id, pokemon, evolution_chain_id.

2.3.3 Feature Creation

The evolution_chain_id feature shows that if a pokemon evolves, they will have associated the same id number (i.e. Charmander→3, Charmeleon→3, Charizard→3).

Consequently, if two pokemon have the same evolution chain id, they are one the evolution of the other. To understand at which evolution stage a pokemon is, I looked at the order in which pokemon of the same evolution chain were placed in the Pokedex, as the first evolutions are placed before the subsequent ones (i.e. Charmander→4, Charmeleon→5, Charizard→6).

Through a code, I built a list that had three values (1, 2, 3) that correspond to the evolution stages of the pokemon. I added the list to my main data frame so that my feature evolution_stages was created.

4. Data preparation for Machine Learning

For my Machine Learning algorithm, I wanted to use 2 categorical variables. So I had to create dummy variables for each of the classes of these categorical variables, as the algorithms do not understand texts but just numbers.

I created the dummies for the evolution_stages feature through the get_dummies() method.

I created the dummies also for type2 feature to take into account also type None. Instead, to consider type1 I built a for that, if the pokemon has a certain type, it will change at his position the respective dummy of that type into 1.

I also dropped names, generation, and pokedex_number, given that they are not useful to the algorithm, and I do not need them anymore.

At the end of all the data cleaning and data preparation, these are the features that my classification models will work with:

- attack
- base_egg_steps
- base_happiness
- base_total
- capture_rate
- defense
- experience_growth
- height_m
- hp
- sp_attack
- sp_defense
- speed
- weight_kg
- is_legendary
- it_has_sex
- stage_1
- stage_2
- stage_3
- water
- None
- bug
- dark
- dragon
- electric
- fairy
- fighting
- fire
- flying
- ghost
- grass
- ground
- ice
- normal
- poison
- psychic
- rock
- steel

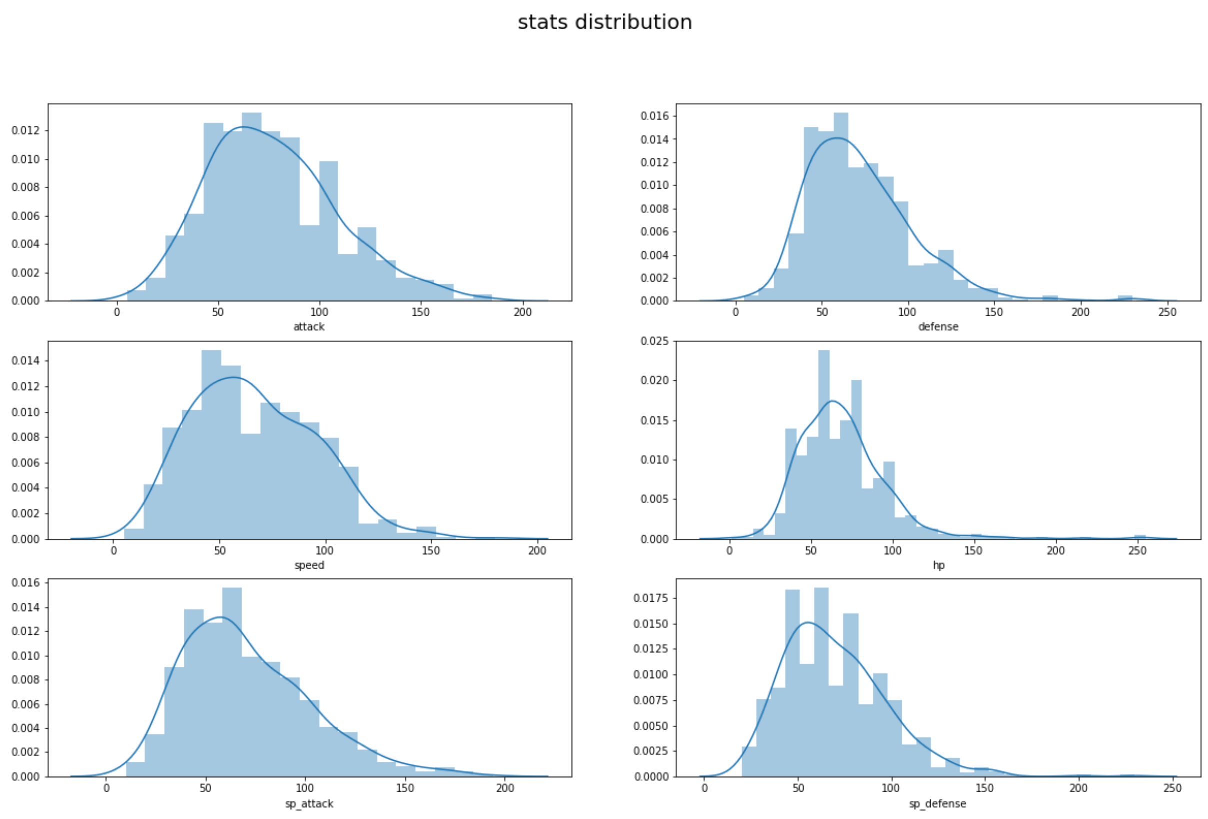
5. Exploratory Data Analysis

Based on my knowledge, water-type pokemons are a well-balanced typing that tends to have higher and special defense statistics. Hence, my goal with this EDA was to check if the hypothesis was correct and understand if there were too highly correlated features that would have caused problems to the various classifiers implemented.

5.1 Distribution of all the statistics

I looked at the distribution of the statistics of all pokemon to have a clear picture of the entire data.

This was the result:



	Variable	N	Mean	SD	SE	95% Conf.	Interval
0	sp_attack	709.0	70.7419	32.1317	1.2067	68.3727	73.1111

	Variable	N	Mean	SD	SE	95% Conf.	Interval
0	defense	709.0	72.2412	30.906	1.1607	69.9624	74.52

	Variable	N	Mean	SD	SE	95% Conf.	Interval
0	speed	709.0	66.5317	28.9866	1.0886	64.3944	68.669

	Variable	N	Mean	SD	SE	95% Conf.	Interval
0	hp	709.0	68.7546	26.545	0.9969	66.7973	70.7119

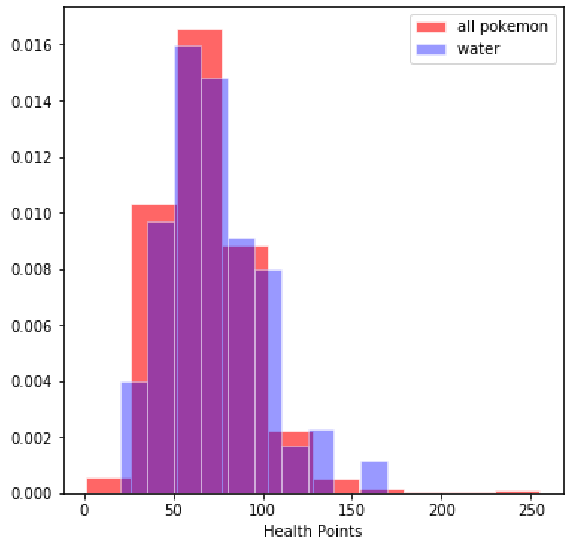
	Variable	N	Mean	SD	SE	95% Conf.	Interval
0	sp_attack	709.0	70.7419	32.1317	1.2067	68.3727	73.1111

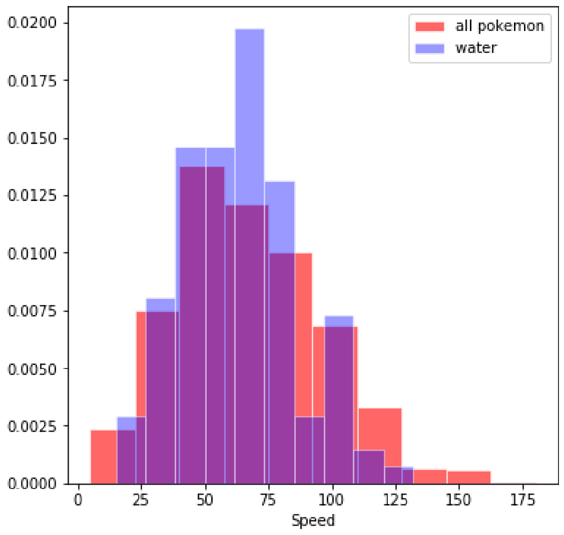
	Variable	N	Mean	SD	SE	95% Conf.	Interval
0	sp_defense	709.0	69.9337	27.5049	1.033	67.9057	71.9618

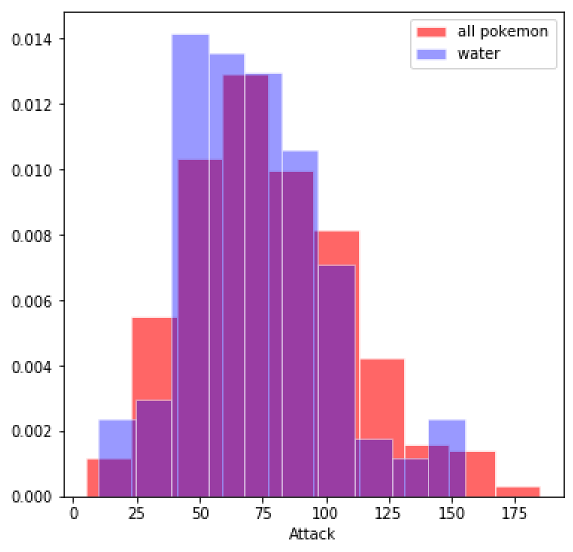
All the data are distributed almost equally with some small differences and the mean is always around 70.

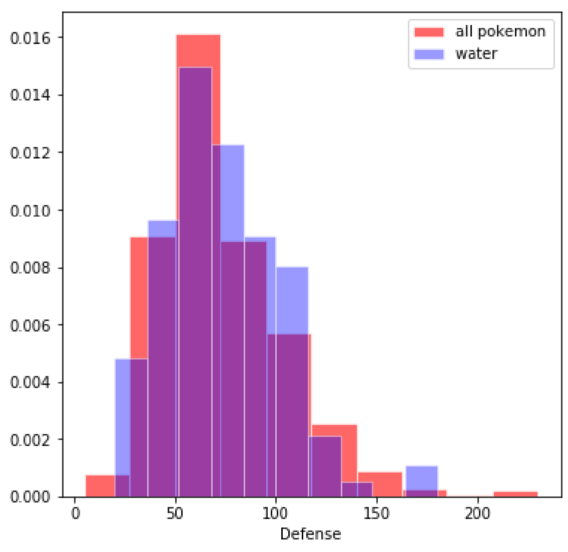
5.2 Distribution of the statistics for water type pokemon

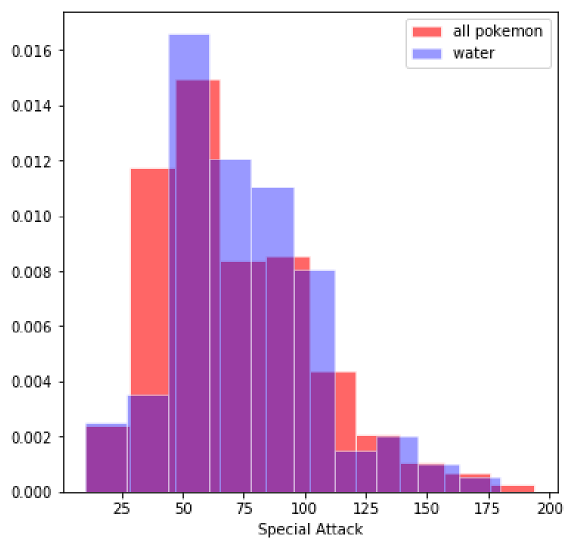
Here, I wanted to compare the distribution of the statistics of all the pokemons with water-type pokemons, to see if my assumptions were right:

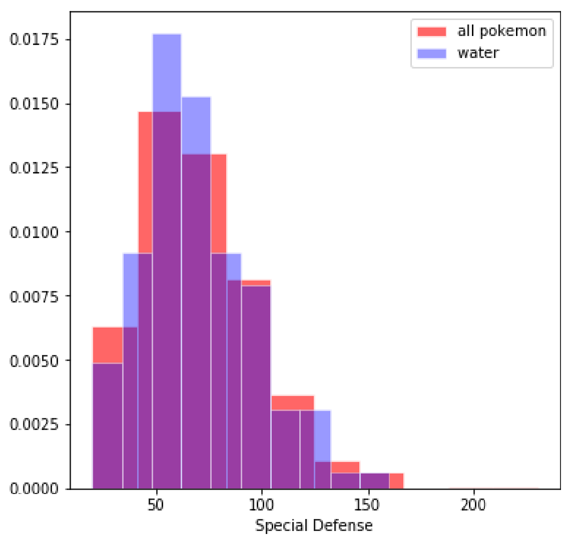










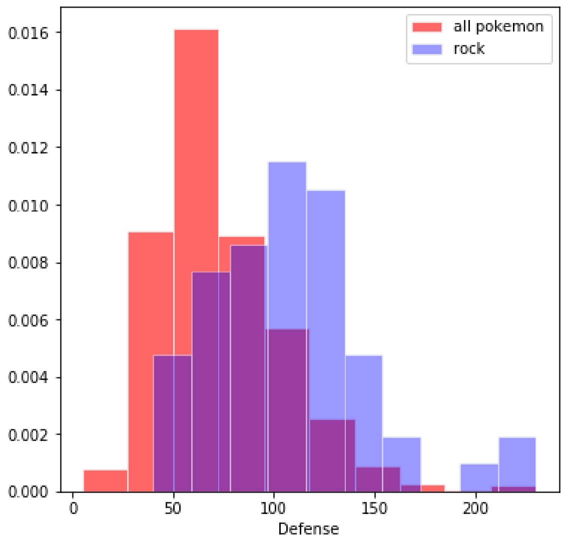


By looking at the graphs we can clearly understand that there are no big differences between all the pokemons and the water-type ones.

I can conclude that my assumption about water pokemons to be high balanced typing was correct; at the same time, my theory about higher defense statistics was not.

Given that the results of the data were not always coherent with my knowledge, I started to question also the most basic common assumptions about the pokemon game. So, to verify if my way of reasoning was correct I also plotted the distribution of the defense statistics for rock-type pokemon, as it is a type that tends to have a high defense.

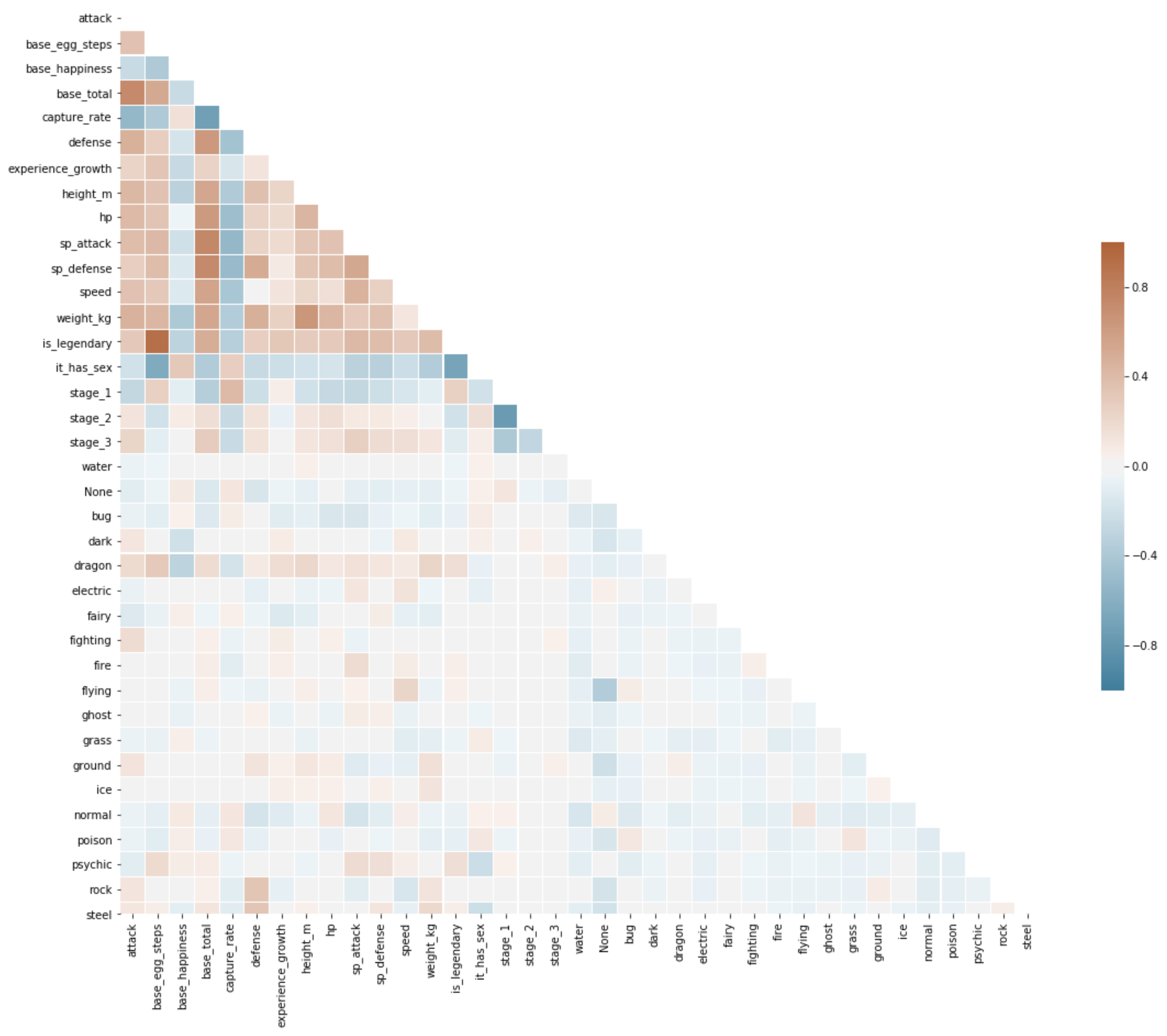
Thankfully, we can see that this assumption was right.



5.3 Correlation Matrix

To further verify the previous consideration, I plotted a correlation Matrix. This tool enabled me to have a bigger picture of the statistics and provide important information about the correlation coefficient of all the features.

Features that are too much correlated with the categorical variable that we want to predict will increase too much the accuracy of our model, leading to overfittingness.



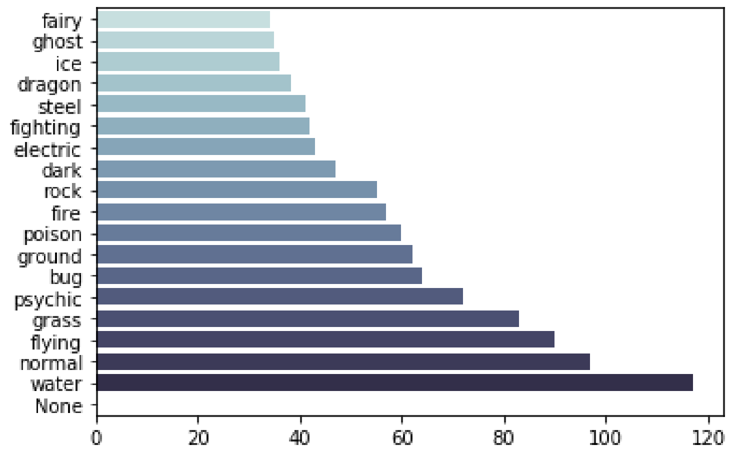
The matrix provided with very goods results, as it shows that for type water we do not have a too high correlation with other statics or the other feature, this is so consistent with the similarity of the distribution of the statistics of all pokemon compared to water type pokemon, further confirming the fact that water is a well-balanced typing.

Given the fact that the correlation between water and the other features is low I will not need to delete some features and the classificators should show good results for predicting the class water.

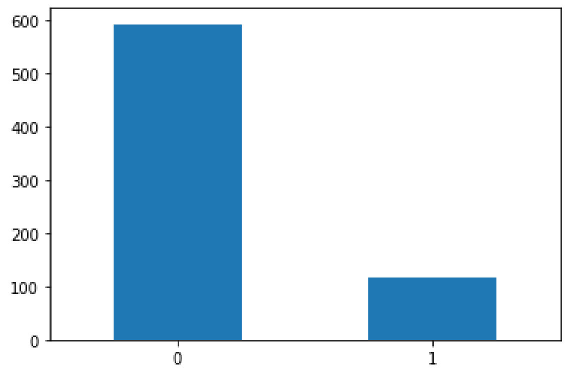
5.4 How many water type pokemons we have

I also needed to understand how many water-type pokemons there exist amongst the data that I have because I have to check if my data is imbalanced.

Imbalanced data can lead to overfittingness, low precision, and sensitivity values. This can compromise our model. I will deepen this topic later in the report.



Looking at the entire sets of pokemon, we see that the type water is the most copious compared to the others, let's see precisely how many of them we have:



```
0    592
1    117
```

As I suspected, the dataset is imbalanced before applying any machine learning algorithm, so we will have to balance it.

6. Machine Learning intro

This is the most important and interesting part of the report.

Here, I have applied the following classifiers to predict the class water:

- Decision Tree
- Random Forest
- Logistic Regression
- KNN
- Support Vector Machine

We will see that the balancing of the data was a crucial part for the building of these models.

I have divided the application of the various classifiers into two parts: one in which I apply an under-sampling technique - Near Miss - and another one in which I apply an over-sampling technique - SMOTE.

We will see that the results drastically change by using one technique or the other.

6.1 Balancing

Machine Learning classifiers fail to cope with imbalanced training datasets as they are sensitive to the proportions of the different classes. As a consequence, these algorithms tend to favor the class with the largest proportion of observations (majority class, in our, is 0, that is to say, a pokemon not being water), which may lead to misleading accuracies. This may be particularly problematic when we are interested in the correct classification of a “rare” class (minority class, 1 a pokemon is water) but we find high accuracies which are actually the product of the correct classification of the majority class.

Because these algorithms aim to minimize the overall error rate, instead of paying special attention to the minority class, they may fail to make an accurate prediction for this class if they don’t get the necessary amount of information about it.

There are 2 main balancing approaches:

1. Undersampling:

   Undersampling works by taking randomly picked samples from the majority class equal in number to the minority class and discard the remaining samples. Then, it combines them with the minority class to balance the data.

2. Oversampling:

   Oversampling works by generating a new sample from the minority class which is duplicated and balanced with the majority class.

7. Machine Learning with NearMiss

7.1 Balancing with NearMiss

NearMiss is an under-sampling technique. It aims to balance class distribution by randomly eliminating majority class examples.

After I split the dataset into X (data that includes the variable that the model will use to predict the typing) and Y (data containing the classes to be predicted) I have applied the NearMiss method which reduced the number of non-water type pokemon, bringing the total amount of object that the models will work with at 234.

Through the train_test_split function I have splitted the data into an 80% of training set, and a 20% of test set.

Once that has been done we can start implementing our classificators

(For all the models instead of splitting the dataset into train and validate, I used the cross-validation function cross_val_score in order to compute an even more accurate validation score, applying 5 different validations to the dataset)

7.2 Decision Tree - first try

For the Decision tree, I ran a for that ranged from 2 to 50 with a step of 2 to, at each iteration, train the decision tree classifier with a different number of leaves in order to find the best number of leaves based on the cross-validation accuracy.

The test accuracy was: 0.809, with best maximum number of Leaves: 18.

I then plotted the graphic representation of the tree to see if it looked deeper enough or just used very few features to make the prediction. The tree looked fine.

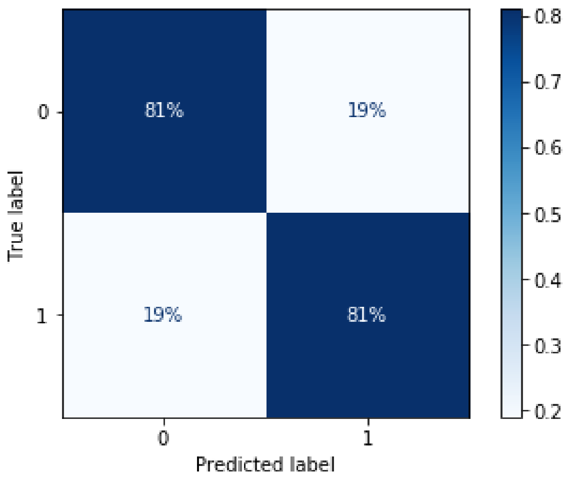
This were are the results of the most important measures of classification:

```
              precision    recall  f1-score   support

           0       0.77      0.81      0.79        21
           1       0.84      0.81      0.82        26

    accuracy                           0.81        47
   macro avg       0.81      0.81      0.81        47
weighted avg       0.81      0.81      0.81        47
```

This are the results of the confusion matrix:



We have high values for Precision, Sensitivity and both for positive and negative predictions the model performed very well most of the time with an accuracy of 81%.

Based on these results I can conclude that the decision tree performed really well and it built a strong classifier.

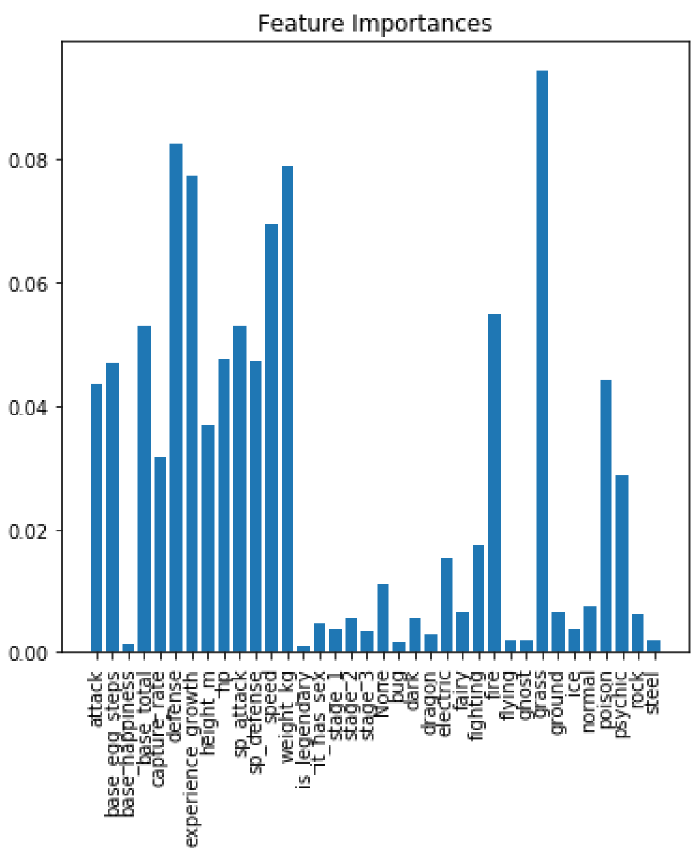
But the result that I had with the decision tree were not consistent with Random Forest, I so had some suspects about how they handled the features of the training set. In the next paragraph, we will see what I have done.

7.3 Random Forest

I deployed random forest, being an augmented version of decision tree I assumed it would have given better results than the previous classifier but this was not the case:

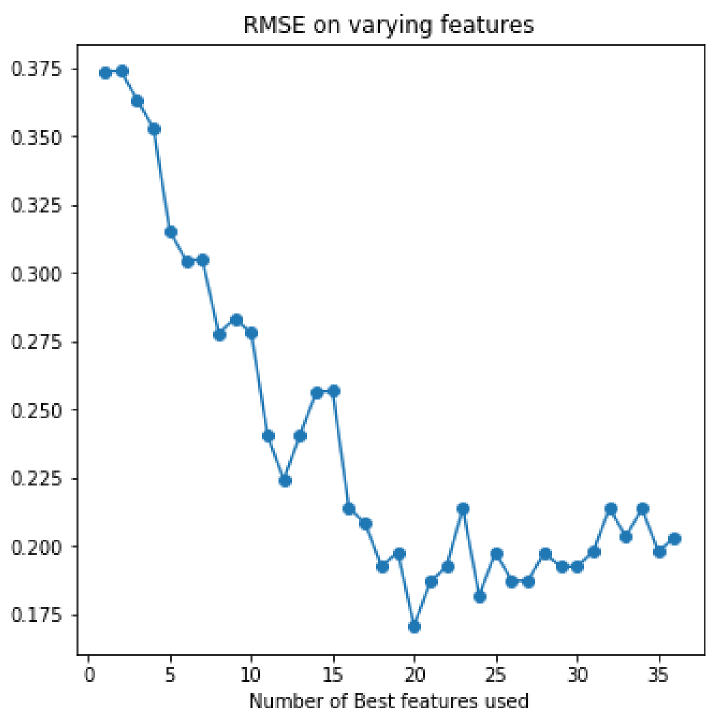
```
Cross-Validation Score:  0.7862019914651494
Test Accuracy: 0.745
```

Given this score I wanted to visualize the importance that the model gave to the various feature, through feature_importance_ I was able to plot the following graph:



The first thing I noticed was that the features that I have created were not used a lot from the algorithm, I so needed to understand which feature to delete to increase the performance of the model.

By computing the mean squared error I plotted the following:



The graph tells that the number of feature that minimize the error should be around 20

To be precise and to not make mistake I decided to use the Recursive Feature Elimination Cross Validation method to let him decide how many features were the optimum and which feature to delete.

7.4 Recursive Feature Elimination Cross Validation

Recursive feature elimination cross validation (RFECV) is basically a backward selection of the predictors, this technique begins by building a model on the entire set of predictors and computing an importance score for each predictor, the least important predictors are then removed, the model is re-built and importance scores are computed again

In practice RFECV provided the optimal number of feature to use and deleted the "useless" ones.

The model left me with 22 features:

```
'attack', 'base_egg_steps', 'base_total', 'capture_rate', 'defense',
       'experience_growth', 'height_m', 'hp', 'sp_attack', 'sp_defense',
       'speed', 'weight_kg', 'it_has_sex', 'stage_1', 'None', 'electric',
       'fighting', 'fire', 'grass', 'ground', 'poison', 'psychic'
```

I so then proceeded at computing the test score, but the accuracy lowered from the initial 0.745 to 0.723

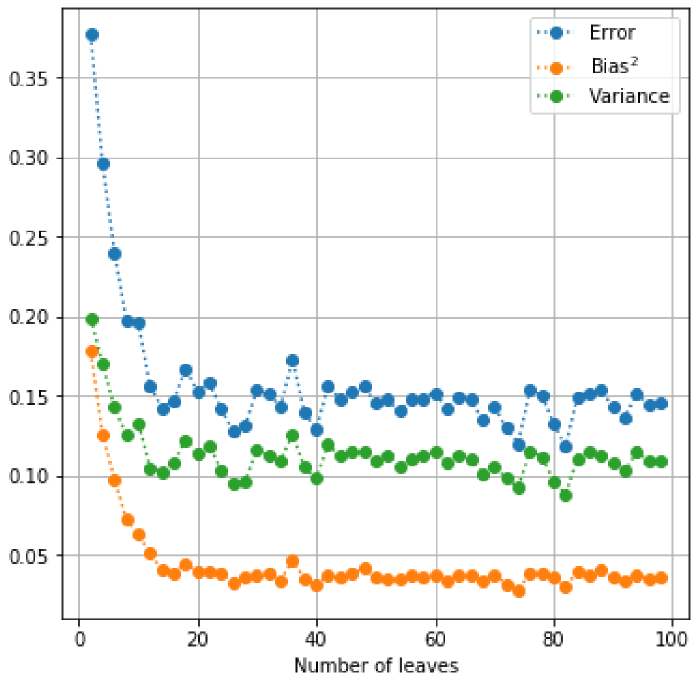
I tried RFECV for a second round to see if it will give me other features that might not actually be helpful but the results were the same, 22 features with 0.723 accuracy

7.5 Decision tree - second try after feature selection

After I applied RFECV I checked the performances of the decision tree with the new number of features, the result did not change a lot the accuracy remained at around 81%. I plotted the trees to see if something changed and this time it used way less feature to make its predictions.

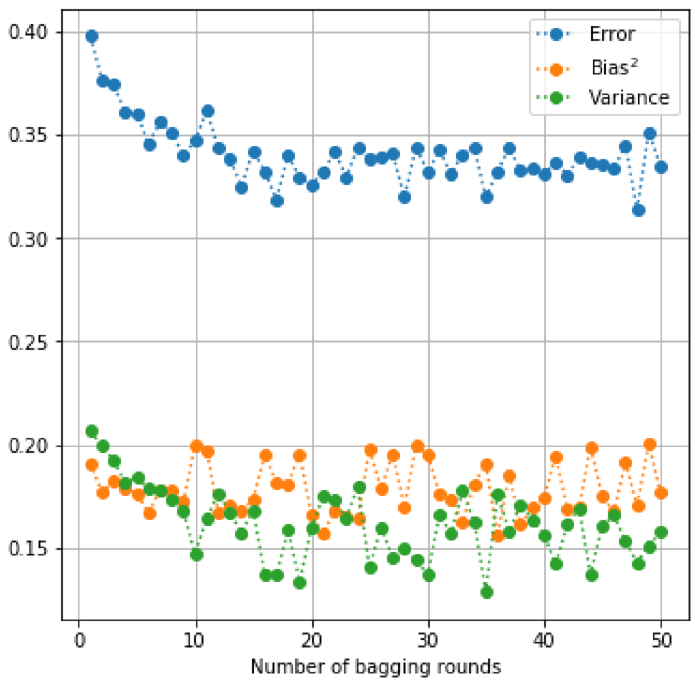
7.6 Bias - Variance analysis

I then so tried to look at the Bias and the Variance.



From the graph, we see that the bias and variance are not too high but the variance is higher than the bias I, therefore, decided to apply the Bagging technique to reduce the variance and see If I could improve the model

The result was the following:



7.7 Decision tree- final try after Bagging

Once I applied the decision tree for the last time the accuracy was still around 81% with no major changes. I so decided to use others classifier to compare the different performances

7.8 Scaling and KNN, Logistic Regression, SVM results

Since the next models that I will implement are based on the distance between the different elements of the data, I so have to scale my data since I have too many features with too many different magnitudes of values.

I have decided to apply a MinMaxScaler to my train and test set.

7.8.1 Logistic regression

I used the Logistic regression by setting solver as liblinear, because it fits better for smaller datasets, as the Scikit-Learn documentation suggests.

This were the results:

```
Cross-validation accuracy score: 0.679
Test Accuracy Score:  0.8085106382978723
```

7.8.2 KNN

When building the KNN classifier I decided to use a for ranging from 1 to 100 with a step of 3 (odd number so to avoid unwanted errors) in order to find the best number of k neighbors

With 8 as best number o k neighbours the algorithm had a 0.809 accuracy

7.8.3 SVM

I deployed SVM that gave me the following results:

```
Accuracy: 0.7872340425531915
Best C: 10
```

7.9 Final Results with NearMiss balancing

```
These are the following results of the project:

Decision Tree accuracy after getting rid of underperforming features:  0.80851063829787
Random Forest:  0.723404255319149
Bagging:  0.7872340425531915
Logistic Regression:  0.8085106382978723
KNN  0.8085106382978723
Support Vector Machine:  0.7872340425531915
```

After implementing all the desired classifier, and after the process of feature selection and Bagging, I can conclude that:

- The best models built to predict the class water are: Decision tree, Logistic Regression, and KNN
- The feature that I created were not useful to the machine learnings algorithms

8. Machine Learning with SMOTE

After my first supervised learnings, I re-implemented again all the classificators, and went through the same steps as before, but this time I balanced the data with SMOTE.

In this section we will compare all the results

8.1 Balancing with SMOTE

SMOTE (Synthetic Minority Over-sampling) is an oversampling technique that generates synthetic samples using k nearest neighbors algorithm from the minority class (pokemon who are water type) to balance the data.

The advantage of using SMOTE over simple oversampling or undersampling techniques like NearMiss is that we do not have the Loss of information, caused by undersampling, and it mitigates the overfitting caused by normal oversampling.

8.2 SMOTE VS NearMiss result comparisons.

After the implementation of SMOTE all the models outperformed their previous result, increasing the accuracy by substantial amounts.

8.2.1 Decision tree

SMOTE

The test accuracy was: 0.869, with the best maximum number of Leaves: 28.

NearMiss

The test accuracy was: 0.809, with the best maximum number of Leaves: 18.

NearMiss

```
              precision    recall  f1-score   support

           0       0.77      0.81      0.79        21
           1       0.84      0.81      0.82        26

    accuracy                           0.81        47
   macro avg       0.81      0.81      0.81        47
weighted avg       0.81      0.81      0.81        47
```

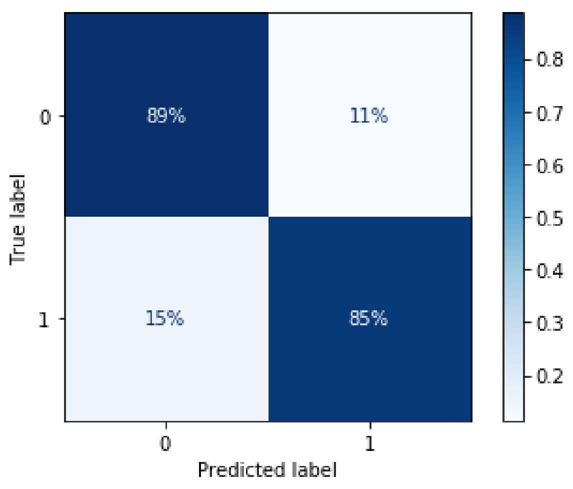
SMOTE

```
             precision    recall  f1-score   support

           0       0.85      0.89      0.87       114
           1       0.89      0.85      0.87       123

    accuracy                           0.87       237
   macro avg       0.87      0.87      0.87       237
weighted avg       0.87      0.87      0.87       237
```

Accuracy and Sensitivity both increase for negative and positive predictions, this led to an increase in the prediction accuracy:



8.2.2 Random Forest

The accuracy with random forest dramatically increased and performs far better than the Decision tree

SMOTE

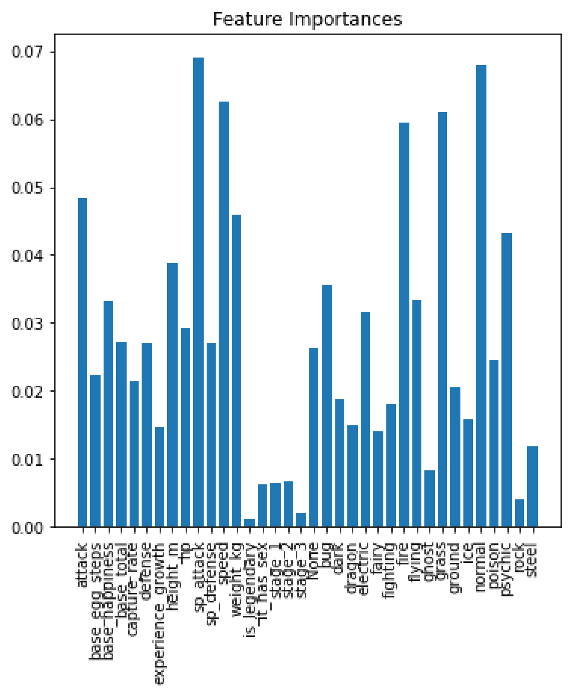
```
Cross-Validation Score:  0.9503870788
Test Accuracy: 0.949
```

NearMiss

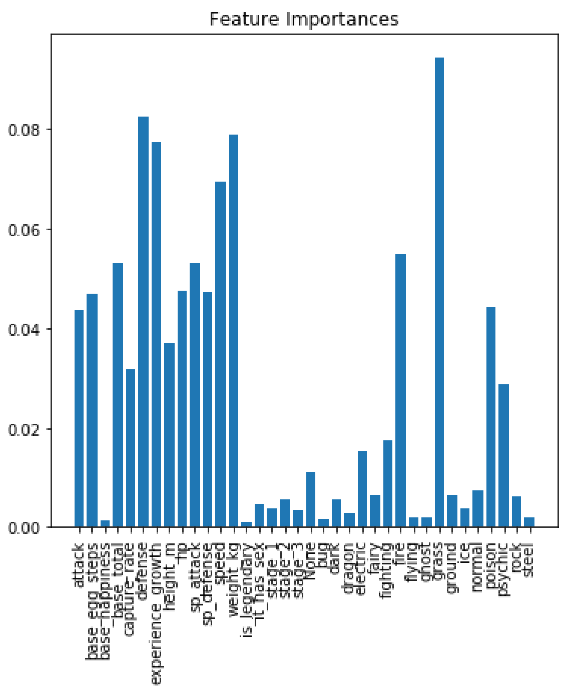
```
Cross-Validation Score:  0.7862019914
Test Accuracy: 0.745
```

Feature Importance:

SMOTE



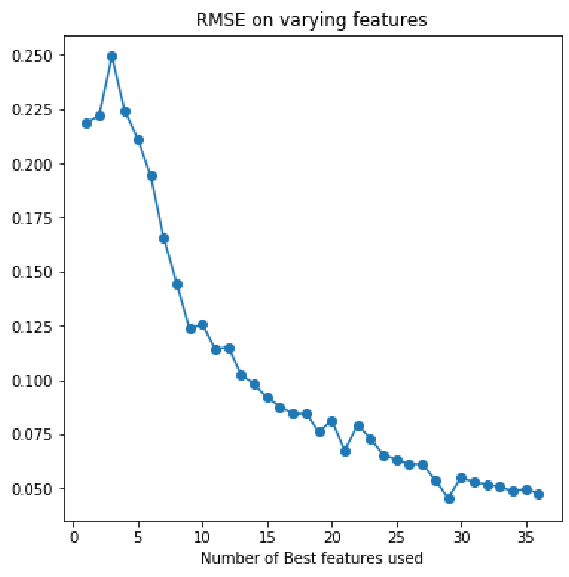
NearMIss



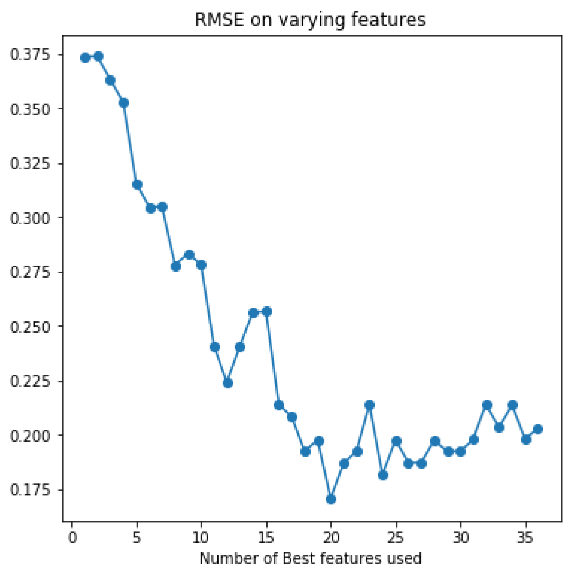
applying again feature_importances_ we can clearly see that this time the majority of the features describing the typing have increased their importance and have played an important part in training the model for the prediction, they have become useful.

I cannot say the same for the feature that I have personally created that still remain under used.

SMOTE



NearMiss



This time the mean squared error indicates an higher optimal number of feature to use.

Let's see it this is consistent with RFECV

RFECV left me with 33 features out of the intial 36:

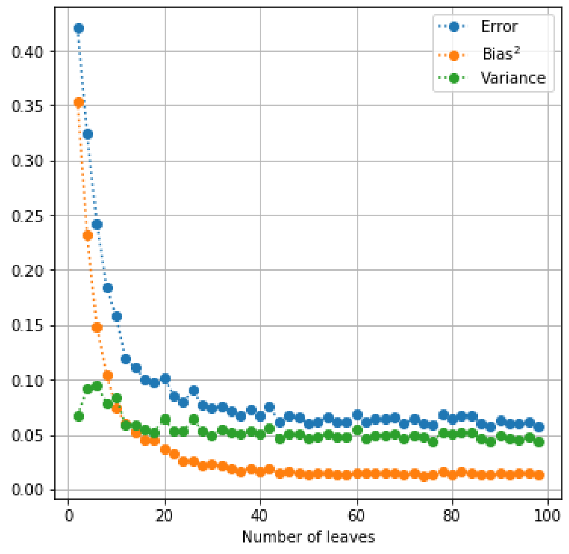
```
'attack', 'base_egg_steps', 'base_happiness', 'base_total',
       'capture_rate', 'defense', 'experience_growth', 'height_m', 'hp',
       'sp_attack', 'sp_defense', 'speed', 'weight_kg', 'it_has_sex',
       'stage_1', 'stage_2', 'None', 'bug', 'dark', 'dragon', 'electric',
       'fairy', 'fighting', 'fire', 'flying', 'ghost', 'grass', 'ground',
       'ice', 'normal', 'poison', 'psychic', 'steel'
```

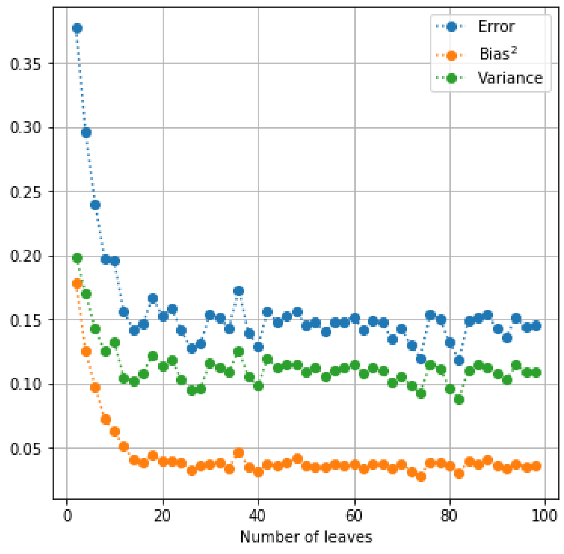
8.2.3 Bias-Variance Comparison

SMOTE

NearMiss

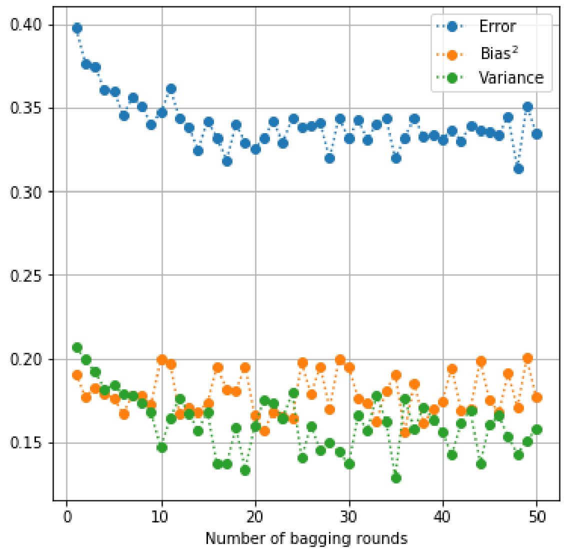
Before Bagging





SMOTE enabled to minimize a lot bias and variance, they assume values below 0.05 a very good result

After Bagging



9. Final Results - comparison of all model performance

```
These are the following results of the project after the implementatio of SMOTE:

Decision Tree accuracy:  0.869198312236287
Random Forest:  0.9324894514767933
Logistic Regression:  0.9915611814345991
KNN  0.8607594936708861
Support Vector Machine:  0.9915611814345991
```

```
These are the following results of the with NearMiss:

Decision Tree accuracy after getting rid of underperforming features:  0.80851063829787
Random Forest:  0.723404255319149
Bagging:  0.7872340425531915
Logistic Regression:  0.8085106382978723
KNN  0.8085106382978723
Support Vector Machine:  0.7872340425531915
```

We can clearly see that the use of SMOTE gave an incredible incrementation in all performances for all the models. The Performance for all clasifficators dramatically increased.

I can so conclude:

1. The balancing of data is a crucial part of Machine Learning, using the proper balancing technique can dramatically change the results.
2. Overall simple undersampling and oversampling balancing approach may not suffice due to the loss of relevant information.
3. The feature that I have created were not useful for the training of the algorithms
4. The best classificators built to predict if a pokemon is of the water type are: Support Vector Machine, and Linear Regression