Pokémon - Gotta Predict 'Em All!

Colab link: https://colab.research.google.com/drive/19fjGJf8OXYKaJ8nDXcZuawbSNVvWd6Dq

Github link: https://github.com/Sasha91/SDS-M1-exam-datasets

In [0]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage, dendrogram
from scipy.cluster.hierarchy import fcluster
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import normalize
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score


import random
import math

sns.set(rc={"figure.figsize":(12, 10)})
plt.style.use("seaborn-colorblind")
```

In [0]:

```
# We create instances of standardscaler and the simpleImputer
imp = SimpleImputer(missing_values=np.nan, strategy="mean")
scaler = StandardScaler()
```

In [0]:

```
#We load in the pokemon dataset, and our generated combats dataset
pokedex = pd.read_csv("https://raw.githubusercontent.com/Sasha91/SDS-M1-exam-datasets/mas
ter/pokemon.csv")
combats = pd.read_csv("https://raw.githubusercontent.com/Sasha91/SDS-M1-exam-datasets/mas
ter/complete_combats.csv")
```

In [0]:

```
# We create a function to easier perform hyperparameter tunning
def MakeGridSearchCV(model, grid, x, y, scoring="accuracy", folds=5):
  cv = GridSearchCV(model, grid, cv=folds, scoring=scoring)
  cv.fit(x, y)
  return cv
```

```
# A nice little print of the scores
def PrintGridsearchCVScore(cv):
  print("Best parameter is: {}".format(cv.best_params_))
  print("Best score is: {}".format(cv.best_score_))
```

preprocessing

We drop unnessacary columns for this project. Missing values for height and weight are imputed

We do it up front instead of including it in a pipeline, because it will be used in none machine learning tasks as well

In [0]:

```
#Drop unnecessary columns
pokedex.drop(["classfication", "japanese_name", "capture_rate", "base_total"], axis= "co
lumns", inplace=True)
```

In [149]:

```
# imputing missing values
pokedex_temp = pokedex[["height_m", "weight_kg", "percentage_male"]]
pokedex_temp["height_m"] = pd.to_numeric(pokedex_temp["height_m"], errors="coerce")
pokedex_temp["weight_kg"] = pd.to_numeric(pokedex_temp["weight_kg"], errors="coerce")
pokedex_temp["percentage_male"] = pd.to_numeric(pokedex_temp["percentage_male"], errors=
"coerce")
pokedex_temp = pd.DataFrame(imp.fit_transform(pokedex_temp), columns=["height","weight",
"percentage_male"])
pokedex["weight_kg"] = pokedex_temp["weight"]
pokedex["height_m"] = pokedex_temp["height"]
pokedex["percentage_male"] = pokedex_temp["percentage_male"]

# We convert the is_legendary column from an int to a bool, so it's binary.
pokedex["is_legendary"] = pokedex["is_legendary"].astype(bool)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexin
g.html#indexing-view-versus-copy

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexin
g.html#indexing-view-versus-copy
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexin
g.html#indexing-view-versus-copy
  after removing the cwd from sys.path.
```

In [0]:

```
#Encoding values with the labelencoder from sklearn
type1enc = LabelEncoder()
type2enc = LabelEncoder()

pokedex_numeric = pokedex.drop(["abilities", "name"], axis=1)
pokedex_numeric["type1"] = type1enc.fit_transform(pokedex_numeric["type1"])
pokedex_numeric["type2"] = pokedex_numeric.type2.fillna("")
pokedex_numeric["type2"] = type2enc.fit_transform(pokedex_numeric["type2"])
pokedex_numeric["is_two_type"] = pokedex_numeric["type2"] != 0
```

EDA

In this section we dive into the data in various different ways to order to gather more insigt and find inspiration for questions.

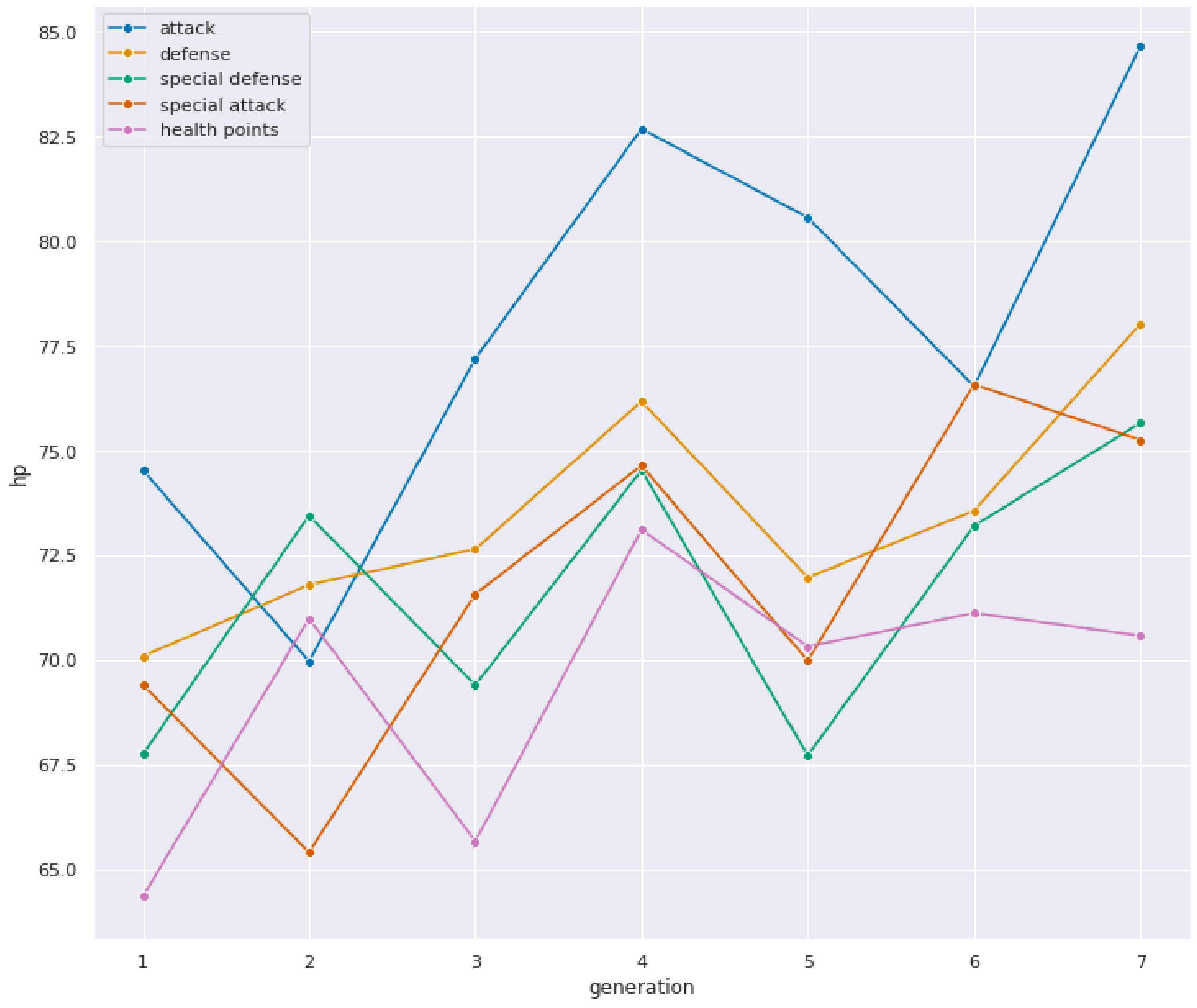
Stats distribution pr. generation

It actually seems like the average stats for the 4th and 7th generation pokemons are quite large compared to the rest.

Generation 2 actually saw a bit of a dip in attack power compared to generation 1, but an increase in defense.

In [151]:

```
generation_mean = pokedex.groupby(by="generation").mean()

sns.set(rc = {"figure.figsize": (12, 10.5)})
sns.set_palette("colorblind")
ax = sns.lineplot(x = generation_mean.index, y = generation_mean["attack"], marker="o", label="attack")
sns.lineplot(x = generation_mean.index, y = generation_mean["defense"], marker="o", label="defense", ax=ax)
sns.lineplot(x = generation_mean.index, y = generation_mean["sp_defense"], marker="o", label="special defense", ax=ax)
sns.lineplot(x = generation_mean.index, y = generation_mean["sp_attack"], marker="o", label="special attack", ax=ax)
sns.lineplot(x = generation_mean.index, y = generation_mean["hp"], marker="o", label="health points", ax=ax)
plt.margins(0.05)
plt.show()
```

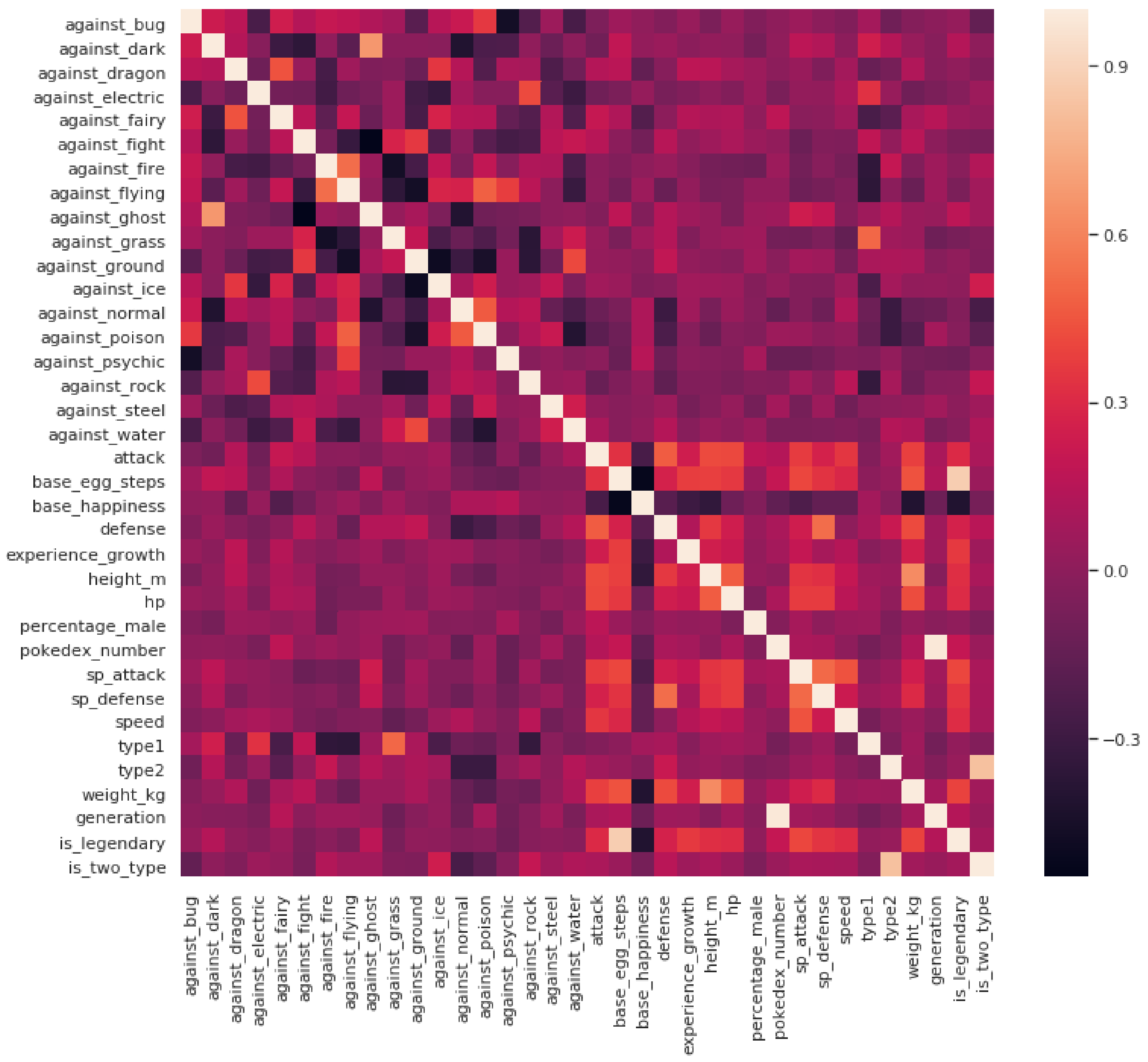


In [152]:

```
# exploring the correlation between the features
corr_vars =  pokedex_numeric.corr()

sns.heatmap(corr_vars)
```

Out[152]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa9975f5978>
```

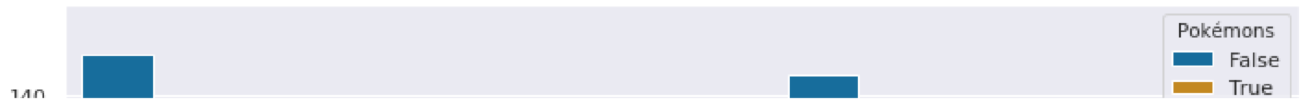


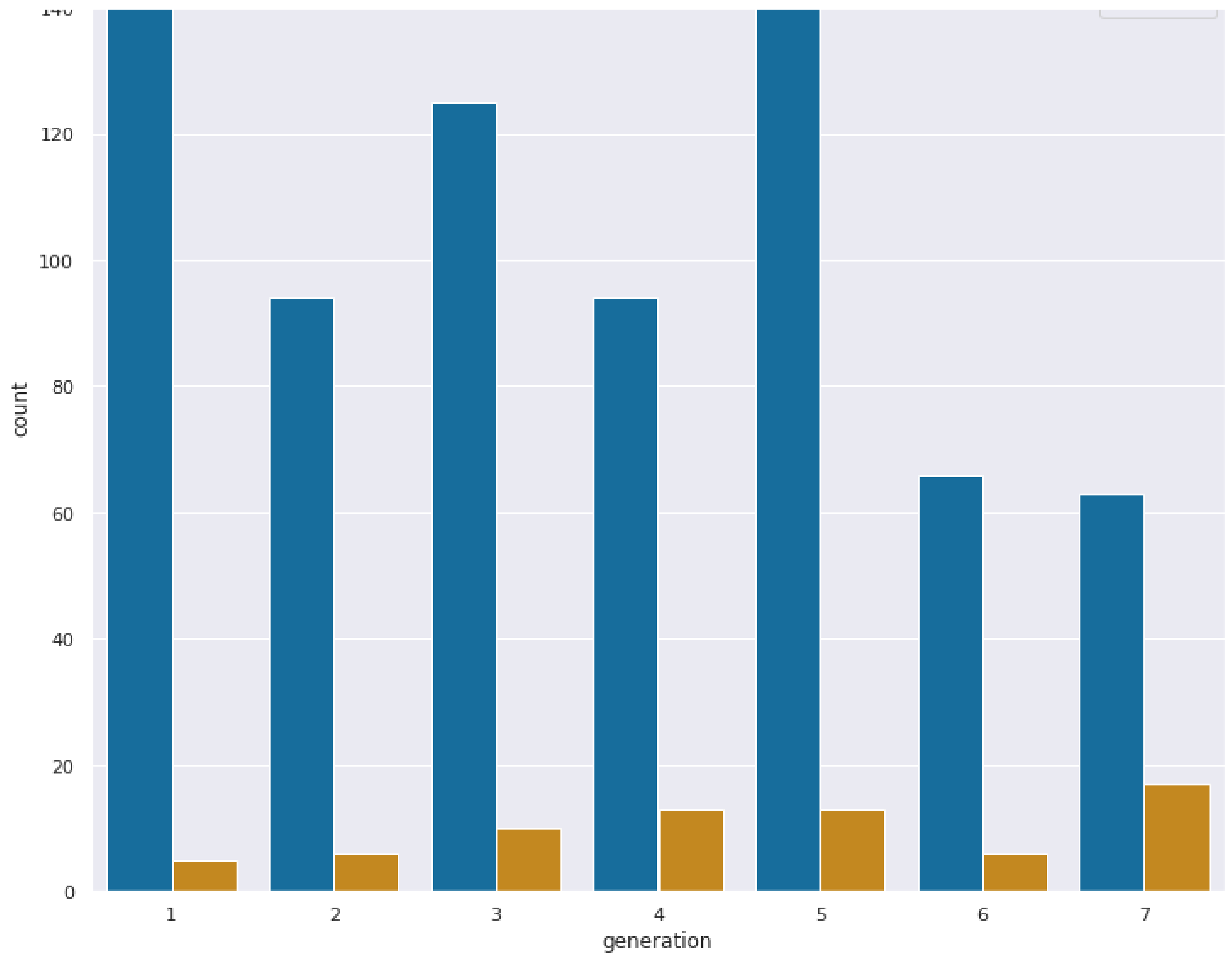
The heatmap shows a square of "more corelated features" and we see that base-stats such as hp, attack, defense, speed is related to eachother. Height and weight is also highly corelated, and even more interesting, is that is_legendary is extremely corelated to base_egg_steps.

number of normal and legendary pokemons pr. generation

In [153]:

```
pokemon_count = pokedex.copy()
ax = sns.countplot(x = "generation", hue="is_legendary", data=pokemon_count)
ax.legend_.set_title("Pokémons")
```





If we look at generation as a time-factor, we see that the amount of legendary pokemon created increased every generation except generation 6, where there in general were created less pokemon. In generation 7 the amount of legendary pokemon skyrocketed.

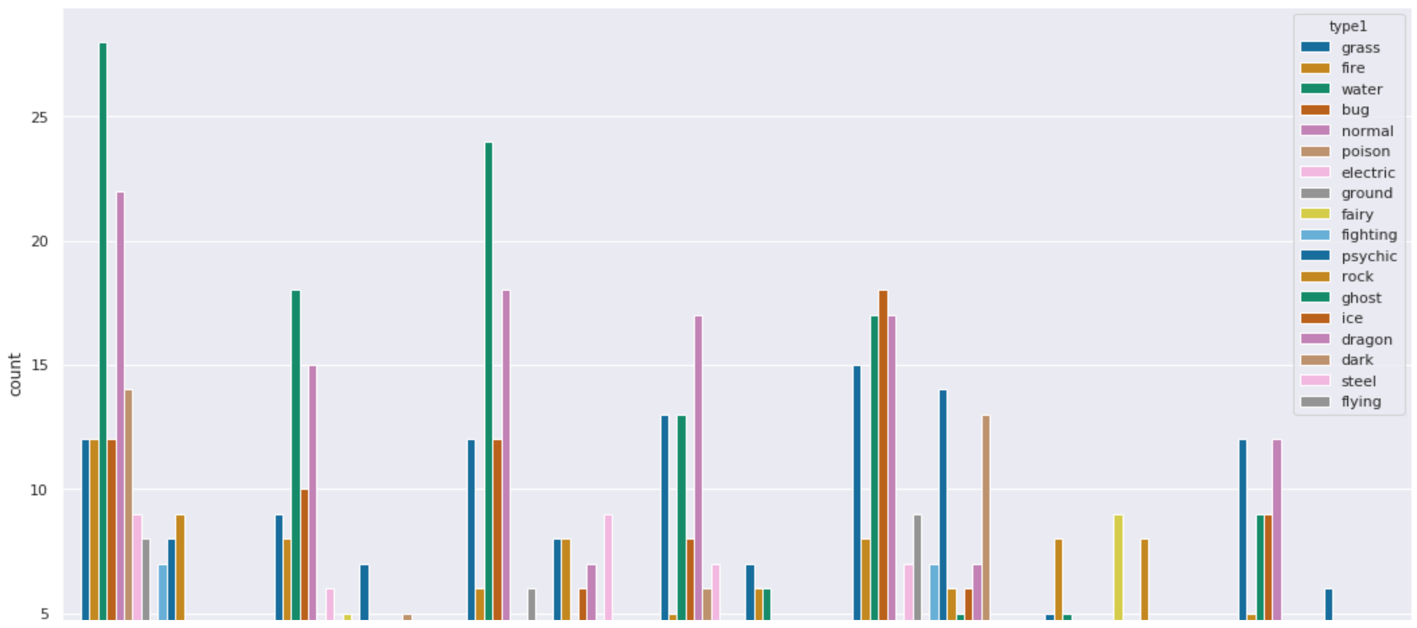
Number of Pokémon types pr. generation

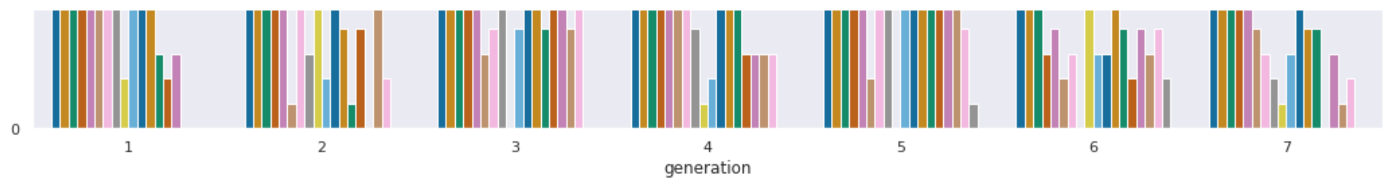
In [154]:

```
sns.set(rc={"figure.figsize":(18, 10)})
sns.countplot(x="generation",hue="type1", data=pokedex, palette="colorblind")
```

Out[154]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa998830630>
```





Previously we saw the amount of pokemon seperated by is_legendary and sorted by generation.

Here we showcase the amount of pokemons per type in each generation. We see that in the first five generation there is a special love for water and normal pokemon, with the sudden spiking interest in bugs, in the fifth generation. We can also deduct that there is a relatively high amount of the six first types, that are created every generation.

We can also see that there are certain types that were not created in specific generations. For example that dark, steel and flying weren't in the first generation, or that there were no fairies in generation three or five and no dragons in generation seven.

Median BMI pr. generation

We take the median, bacause some pokemons throw the scale off completely, such as the pokemon: Cosmoem from generation 7.

We can conclude that Pokémons have not gotten fatter with time, but they are still overweight, after human standards atleast.

In [155]:

```
pokedex_bmi = pokedex.copy()
pokedex_bmi["BMI"] =  round(pokedex["weight_kg"] / pokedex["height_m"] ** 2, 1)

bmi_pr_generation = pokedex_bmi.sort_values(by="BMI").groupby("generation")["BMI"].media
n()
bmi_pr_generation
```

Out[155]:

```
generation
1    30.60
2    26.40
3    28.40
4    32.50
5    31.90
6    29.75
7    27.10
Name: BMI, dtype: float64
```

Unsupervised machine learning

Splitting the dataframe in pokemons with two types, and those with just one

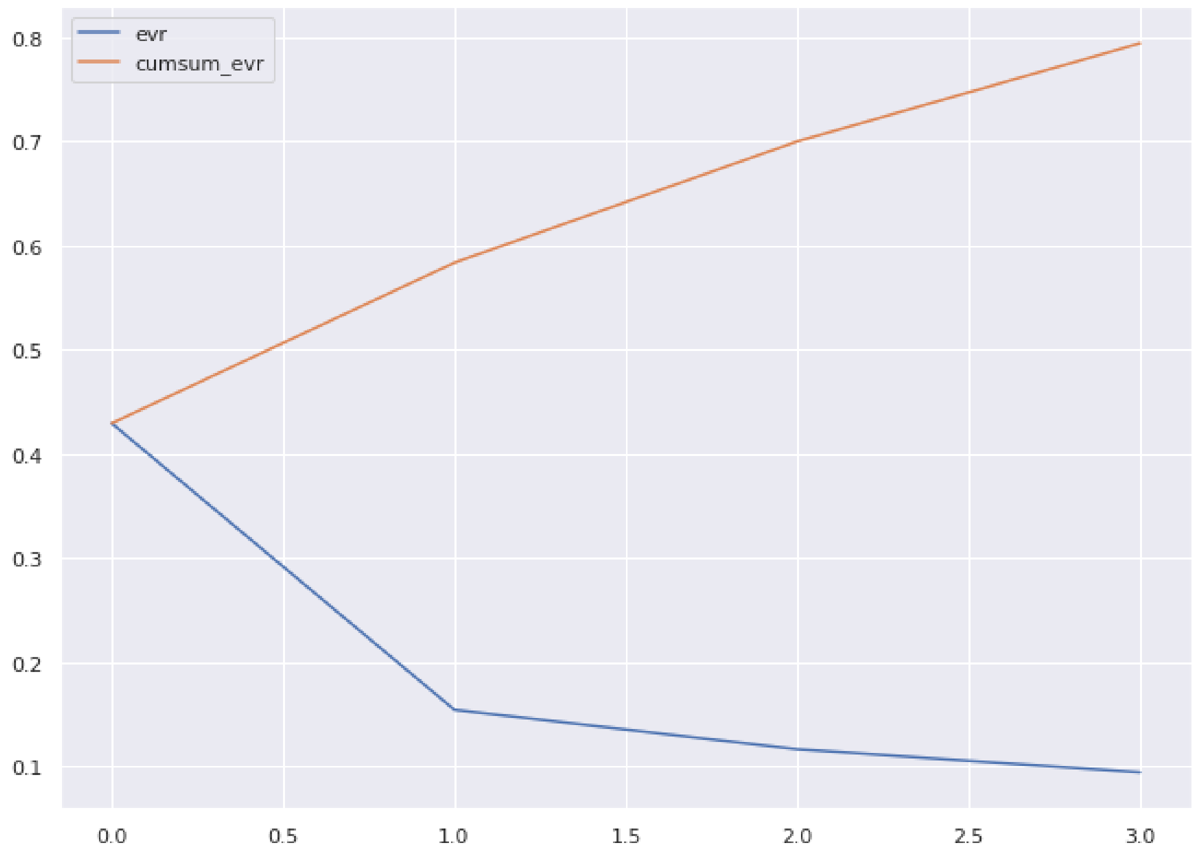
We want to examine if it matters that a pokemon has two types or not. We do this by selecting all the features that make up the stats of the pokemon, performing dimension reduction through PCA and then cluster the data to see if the clusters somewhat form around wether a pokemon has two types or not.

In [156]:

```
#Selecting the features we want to examine and scaling them
pokedex_numeric_temp = pokedex_numeric[["attack", "defense", "sp_attack", "sp_defense",
"hp", "speed", "height_m", "weight_kg"]]
pokedex_numeric_scaled = scaler.fit_transform(pokedex_numeric_temp)

pca = PCA(n_components=4)
```

```
pokedex_reduced = pca.fit_transform(pokedex_numeric_scaled)

pokedex_plot = pd.DataFrame({"evr": pca.explained_variance_ratio_, "cumsum_evr": np.cums
um(pca.explained_variance_ratio_)}).stack()
sns.set(rc={"figure.figsize": (11, 8)})
sns.lineplot(y = pokedex_plot.values, x = pokedex_plot.index.get_level_values(0), hue=po
kedex_plot.index.get_level_values(1))
print(pca.explained_variance_ratio_.sum())
```

```
0.7942597310194065
```



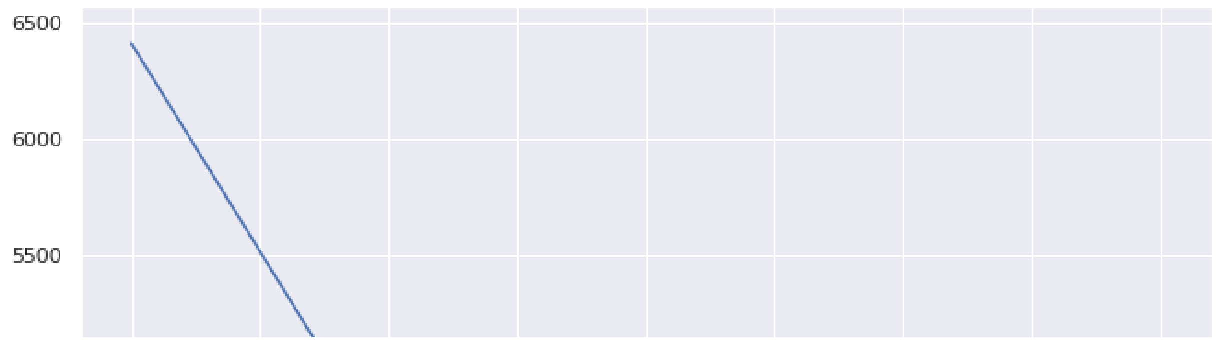
We chose the features we want to use, and then we scaled the data from 8 components to 4 and thereby reducing the dimension, using the PCA algorithm. We then plot the data to find the explained variance ratio, which in this case is 79 %, which means that 79 % of the data can be used to explain something. Our "explaining power".

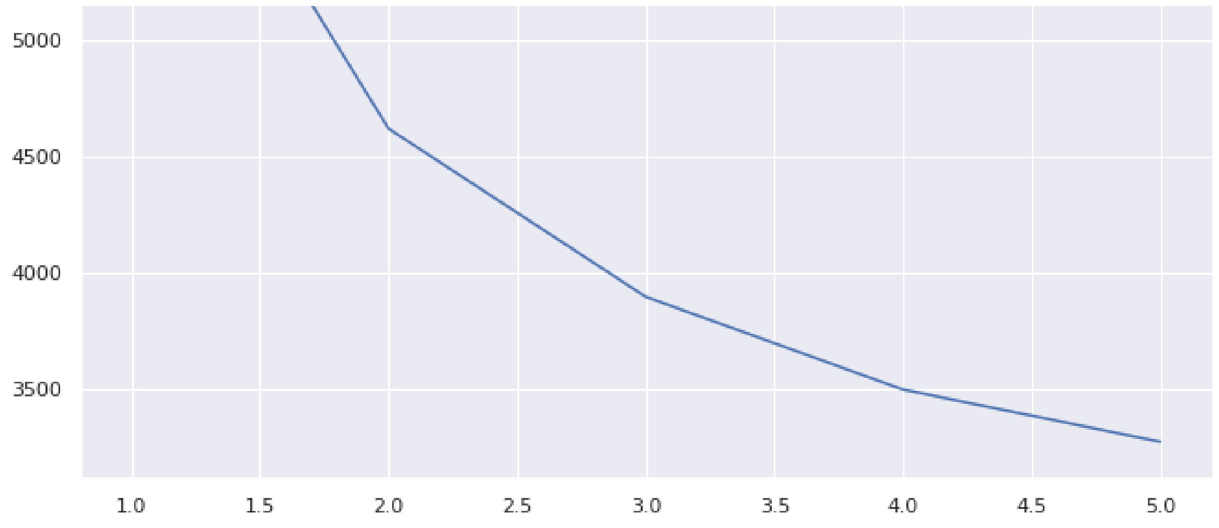
In [157]:

```
inertia = []

for i in range(1, 6):
  kmeans = KMeans(n_clusters = i)
  inertia.append(kmeans.fit(pokedex_numeric_scaled).inertia_)

sns.lineplot(y = inertia, x = range(1, 6))
```

Out[157]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa99749b240>
```





We would then like to cluster our data, but before we can do that, we would like to know how many clusters would be optimal for our dataset.

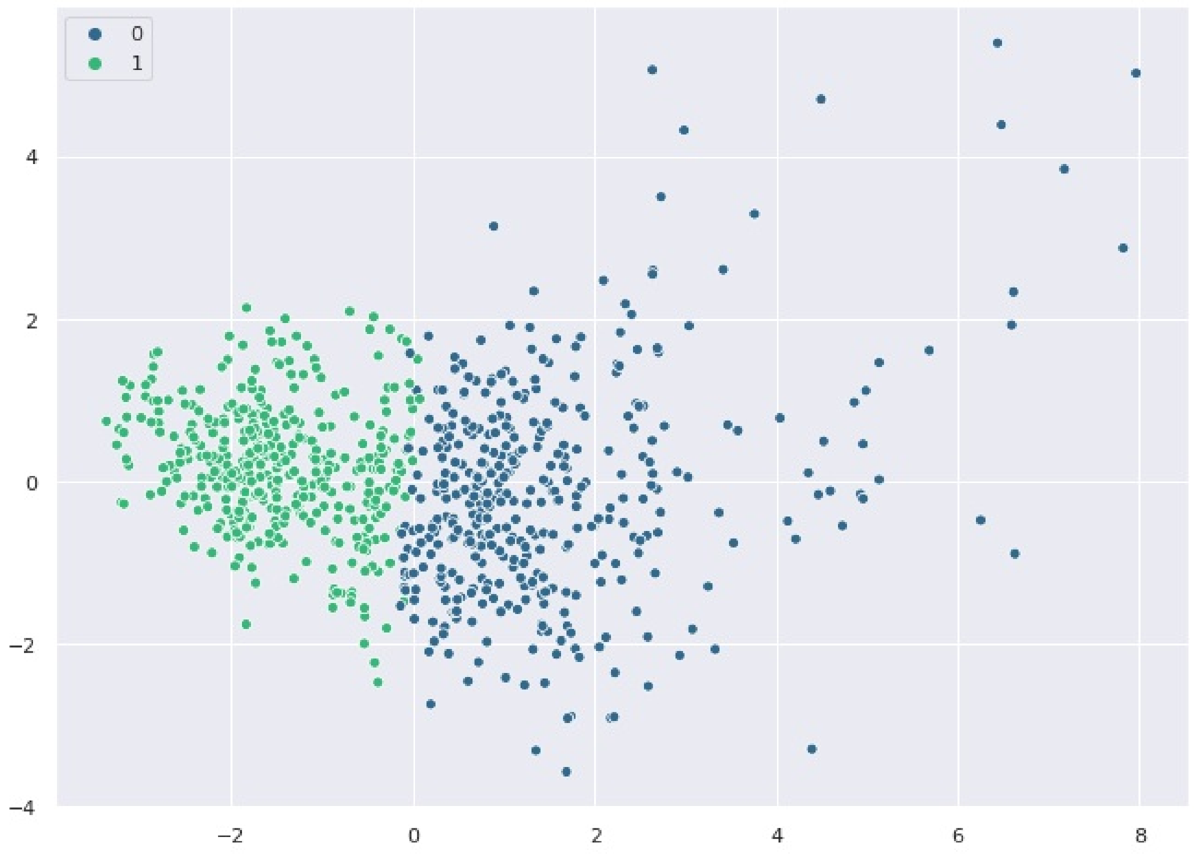
We did this using the KMeans algorithm to plot the inertia. We then use the elbow-method to choose the amount of clusters, which in this case would be two or three, and we choose two.

In [158]:

```
clusterer = KMeans(n_clusters = 2)
clusterer.fit(pokedex_numeric_scaled)

sns.scatterplot(pokedex_reduced[:, 0], pokedex_reduced[:,1], hue=clusterer.labels_, legen
d="full", palette="viridis")
```

Out[158]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa99750c978>
```



We tried to examine if there is a discernable clustering in the pokemons, based on their power, defense, height and weight.

As we can see in the crosstab, this does not appear to be the case, as the pokemons are more or less split 50-50, no matter if they have two types, or not.

In [159]:

```
pokedex_test_copy = pokedex_numeric.copy()
pokedex_test_copy["cluster"] = clusterer.labels_

pd.crosstab(pokedex_test_copy.is_two_type, pokedex_test_copy.cluster)
```

Out[159]:

cluster	0	1
is_two_type		
False	167	217
True	235	182

Hierarcial clustering of the primary types of pokemon, for each generation

Earlier we saw the amount of a certain type, per generation.

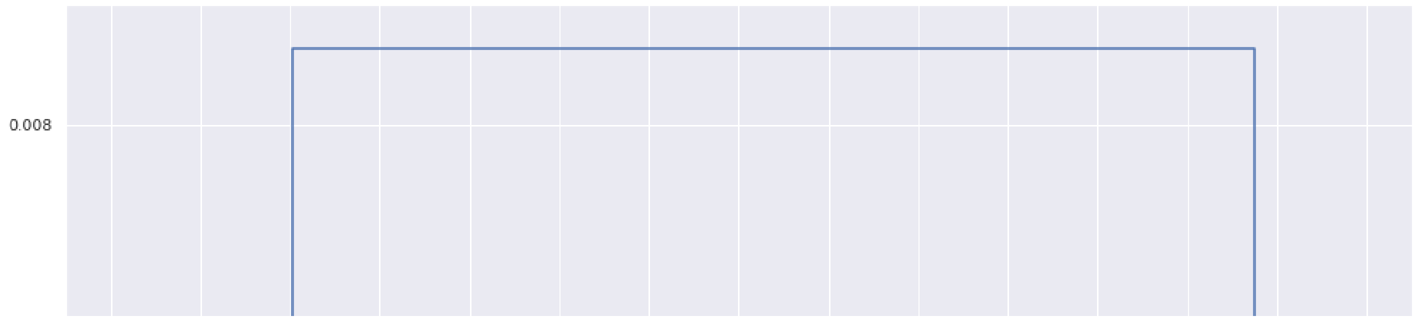
Using hierarchical clustering we would like to explore the correletion between each type for generation 1 and generation 7. With this we hope to identify some new correlations.

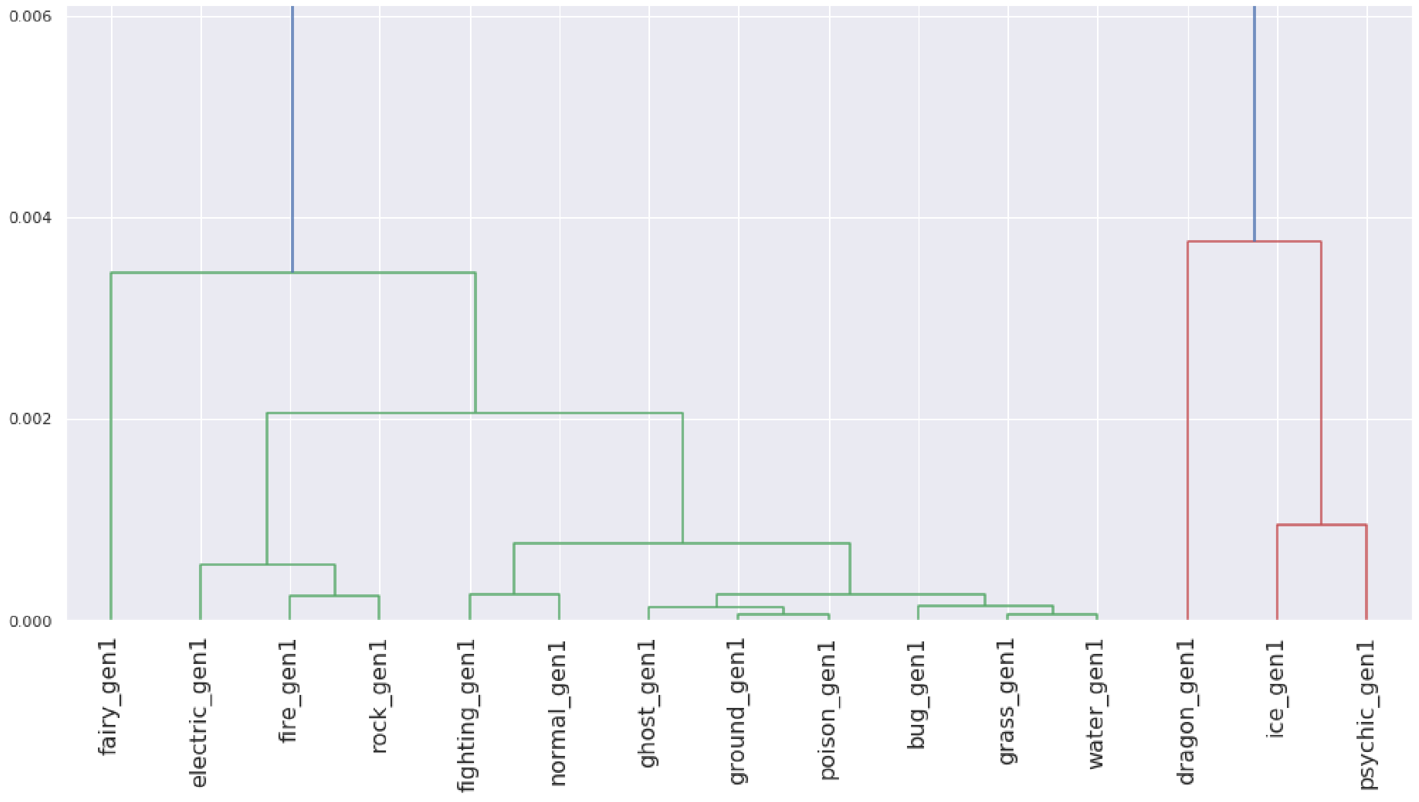
In [0]:

```
# We group by generation
generations = pokedex.groupby("generation")

#We get the type of generation 1 and add "_gen1"
gen1 = generations.get_group(1).groupby("type1").mean()
gen1 = gen1.set_index(gen1.index + "_gen1")

#We get the type of generation 7 and add "_gen7"
gen7 = generations.get_group(7).groupby("type1").mean()
gen7 = gen7.set_index(gen7.index + "_gen7")

#We concatinate the two dataframes
frames = [gen1, gen7]
concat = pd.concat(frames, ignore_index=False)
gen_combined = concat
```

In [161]:

```
pokemdex_normalized = normalize(gen1)

mergings = linkage(pokemdex_normalized, method="complete")

sns.set(rc={"figure.figsize": (17,12)})
dendrogram(mergings,
           labels=gen1.index,
           leaf_rotation=90,
           leaf_font_size=16,
)

plt.show()
```





We use the linkage function to calculate the hierarchical clustering, but for hierarchical clustering we need to scale the data using the normalize function, since hierarchical clustering doesn't fit in a pipeline.

The dendrogram shows when clusters are set together in a hierarcical order. I our case it means, the lower the distance, the bigger the relationship between the types.

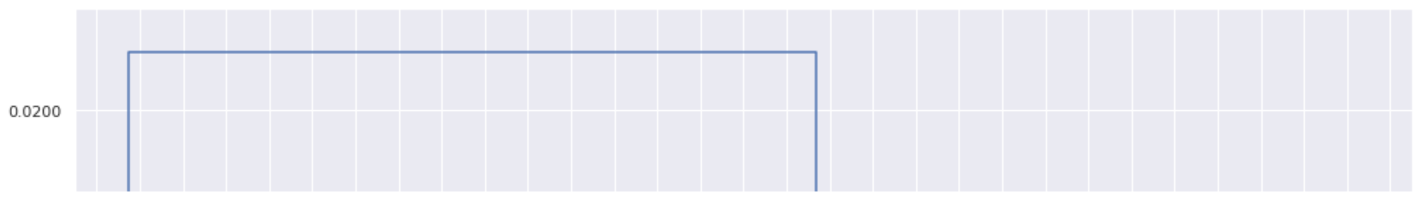
Just by glancing over the graph we see that the first clusters are created by ground/poison and grass/water. This might have something to do with their strength/weakness towards each other, since ground deals double damage to poison and grass deals double damage to water. The bug-type is added to the grass/water cluster afterwards, whereas bug deals double damage to grass.

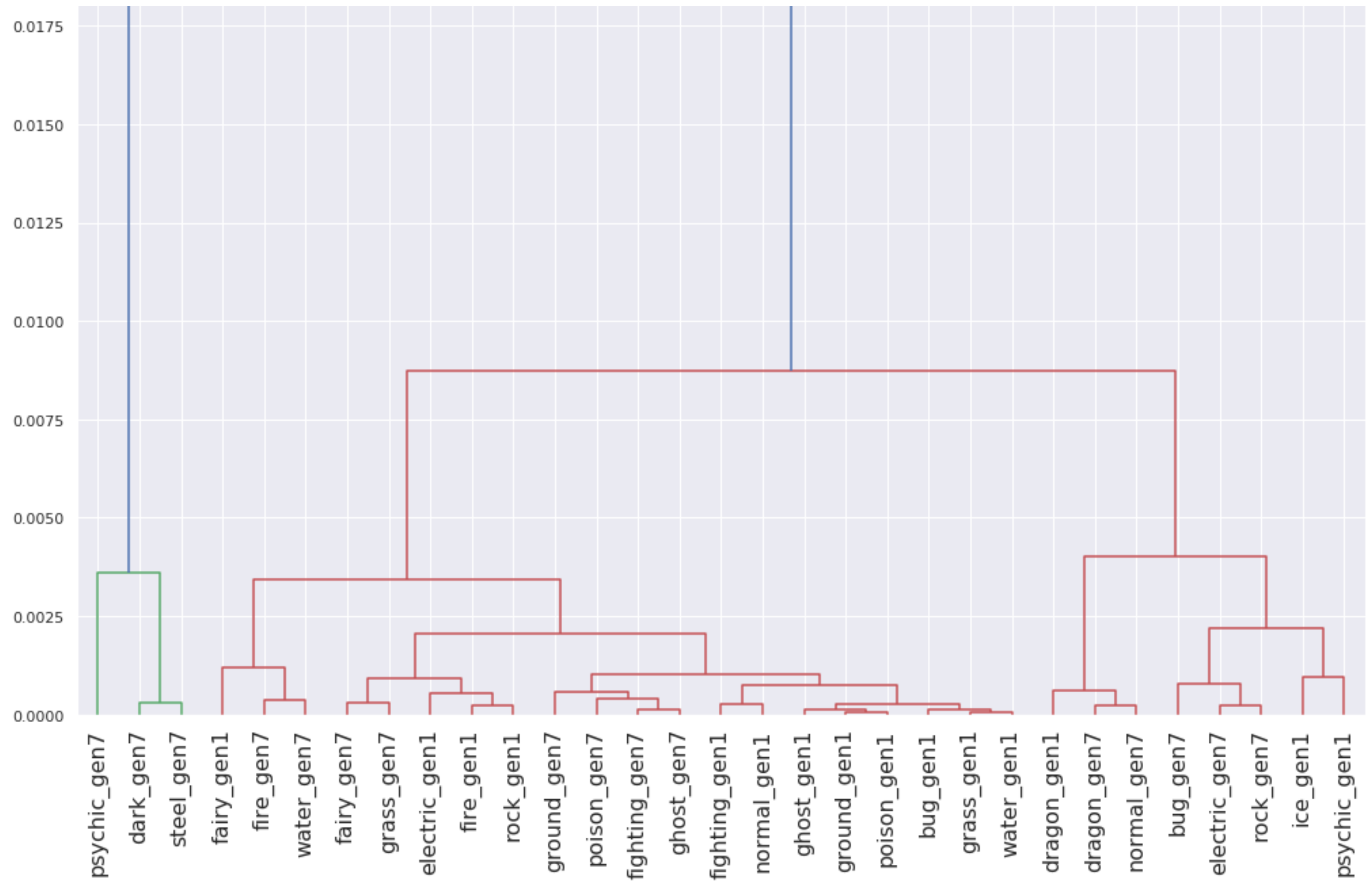
We then have two more clusters that merge, which is rock/fire and fighting/normal, which also seems to be grouped by their strength/weakness correlation. Rock deals double damage to fire and fighting double to normal. We then have fairy-type that complete stands alone, which is merged with most of the pokemon second last, only to dragons merge to ice/psychic, which also is weird, considering ice doesn't have a strength/weakness correlation with psychic, but rather deals double damage to dragon.

This shows that the strength/weakness correlation between the Pokemon is a big factor in the dendrogram, but not the only one. It also shows the mindset of the design of Pokemon, not only types or the number of Pokemon of each type, but the correlation between the types.

In [162]:

```
pokemdex_normalized = normalize(gen_combined)

mergings = linkage(pokemdex_normalized, method="complete")

sns.set(rc={"figure.figsize": (17,12)})
dendrogram(mergings,
           labels=gen_combined.index,
           leaf_rotation=90,
           leaf_font_size=16,
)

plt.show()
```





The dendrogram still separates the generations in the smaller clusters. The first generation 7 cluster is made from fighting/ghost, which is different from generation 1, where the correlation of fighting was with normal. This might be due to the newer type2, that made for a wider combination of pokemons, an example is the pokemon Marshadow, who is of type fighting/ghost in generation 7.

In generation 7 there were no ice-types, and therefore no correlation with psychic. The correlation for psychic is completely different and is now with two new types, that didn't exist in generation 1, steel/dark.

The bug type has a completely new correlation. Compared to before with grass/water, it now has a correlation with electric and rock. At the same time fairy is not left alone on the side, as in generation 1, but now has a relationship with grass.

In this graph we see the difference of thinking in the design of Pokémon and how new types and types left out, influence different correlations between Pokemon types.

Supervised machine learning

```
In [0]:
```

```
#We choose our target and our features and scale the data
y_leg = pokedex_numeric["is_legendary"]
x_leg = scaler.fit_transform(pokedex_numeric.drop("is_legendary", axis=1))

#We split the data in train and test-sets
x_train_leg, x_test_leg, y_train_leg, y_test_leg = train_test_split(x_leg, y_leg, test_s
ize=0.2, random_state=42)
```

Predicting legendary pokemons with Logistic regression

The first model we test is logistic regression, which seems to very good at predicting the legendary status of a pokemon.

We get an average roc_auc cross validation score of about 0.9, a high accuracy and our confusing matric validates that only few none legendary pokemon, was marked as being legendary.

We also compute a 5-fold cross validation f1 score in order to get a more harmonic / realistic mean accuracy score. Here we still see a high value.

We can conclude that our logistic regression model is pretty good at distinguishing normal pokemons from legendaries.

In [164]:

```
log_reg = LogisticRegression(multi_class = "auto", solver="liblinear")

c = np.arange(1, 5)
grid_params = {"C": c}
cv = MakeGridSearchCV(model = log_reg, grid = grid_params, x=x_train_leg, y=y_train_leg)
PrintGridsearchCVScore(cv)

#log_reg.fit(x_train_leg, y_train_leg)

y_pred = cv.best_estimator_.predict(x_test_leg)
#y_pred = log_reg.predict(x_test_leg)

print(confusion_matrix(y_test_leg, y_pred))
print(classification_report(y_test_leg, y_pred))

scores = cross_val_score(log_reg, x_leg, y_leg, cv=5, scoring="roc_auc")
print("Average roc_auc score:", scores.mean())

scores = cross_val_score(log_reg, x_leg, y_leg, cv=5, scoring="f1")
print("Average f1 score:", scores.mean())
```

```
Best parameter is: {'C': 1}
Best score is: 0.98125
[[143   0]
 [  3  15]]
              precision    recall  f1-score   support

       False       0.98      1.00      0.99       143
        True       1.00      0.83      0.91        18

    accuracy                           0.98       161
   macro avg       0.99      0.92      0.95       161
weighted avg       0.98      0.98      0.98       161

Average roc_auc score: 0.9923679060665362
Average f1 score: 0.8271395271395271
```

Predicting legendary pokemons with decision trees

Next we try to predict if a pokemon is legendary with a decision tree model. Once again we see good prediction result across all metrics.

In [165]:

```
cv.best_params_

tree = DecisionTreeClassifier()
tree.fit(x_train_leg, y_train_leg)
y_pred = tree.predict(x_test_leg)

print(confusion_matrix(y_test_leg, y_pred))
print(classification_report(y_test_leg, y_pred))

scores = cross_val_score(tree, x_leg, y_leg, cv=5, scoring="roc_auc")
print("Average roc_auc score:", scores.mean())

scores = cross_val_score(tree, x_leg, y_leg, cv=5, scoring="f1")
print("Average f1 score:", scores.mean())
```

```
[[143   0]
 [  0  18]]
              precision    recall  f1-score   support
```

```
       False       1.00      1.00      1.00       143
        True       1.00      1.00      1.00        18

    accuracy                           1.00       161
   macro avg       1.00      1.00      1.00       161
weighted avg       1.00      1.00      1.00       161

Average roc_auc score: 0.8834731152735067
Average f1 score: 0.7448185647385083
```

Predicting legendary pokemons with random forest

Like logistic regression, this model manages to predict the legendary pokemon quite well too, but not as good as decision tree.

It is to be noted that the random forrest average cross val score for f1 changes everytime it is run and we have experienced a span of 69% to roughly 91 %, which is quite a high range, whereas the decision tree only vary with 1 % and logistic regression doesn't vary at all.

Another factor is that we havn't used hyperparameter tunning on any of the algorithms, since it would signifacally increase the running time, if we had to apply it on every algorithm.

In [166]:

```
rf = RandomForestClassifier()

rf.fit(x_train_leg, y_train_leg)

y_pred = rf.predict(x_test_leg)
print(confusion_matrix(y_test_leg, y_pred))
print(classification_report(y_test_leg, y_pred))

scores = cross_val_score(rf, x_leg, y_leg, cv=5, scoring="roc_auc")
print("Average roc_auc score:", scores.mean())

scores = cross_val_score(rf, x_leg, y_leg, cv=5, scoring="f1")
print("Average f1 score:", scores.mean())
```

```
[[143   0]
 [  0  18]]
              precision    recall  f1-score   support

       False       1.00      1.00      1.00       143
        True       1.00      1.00      1.00        18

    accuracy                           1.00       161
   macro avg       1.00      1.00      1.00       161
weighted avg       1.00      1.00      1.00       161

Average roc_auc score: 0.9971135029354208
Average f1 score: 0.7477207977207978
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

Testing the importance of the `base_egg_steps` feature

This raises the question of how important the `base_egg_steps` feature is, as the correlation heatmap from the EDA showed that there was a high correlation between the `base_egg_steps` and `is_legendary` feature.

As we can see with on the roc_auc score and the f1 score, removing this metric have a large impact on the performance of our model

In [0]:

```
y_1 = pokedex_numeric["is_legendary"]
```

```
x_1 = scaler.fit_transform(pokedex_numeric.drop(["is_legendary", "base_egg_steps"], axis=
1)) # we try to drop the base_egg_Steps feature

x_train_1, x_test_1, y_train_1, y_test_1 = train_test_split(x_1, y_1, test_size=0.2, rand
om_state=42)
```

In [168]:

```
rf = RandomForestClassifier()
rf.fit(x_train_1, y_train_1)

y_pred = rf.predict(x_test_1)
scores = cross_val_score(rf, x_1, y_1, cv=5)
print(confusion_matrix(y_test_1, y_pred))
print(classification_report(y_test_1, y_pred))

scores = cross_val_score(rf, x_1, y_1, cv=5, scoring="roc_auc")
print("Average roc_auc score:", scores.mean())

scores = cross_val_score(rf, x_1, y_1, cv=5, scoring="f1")
print("Average f1 score:", scores.mean())
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

```
[[143   0]
 [  4  14]]
              precision    recall  f1-score   support

       False       0.97      1.00      0.99       143
        True       1.00      0.78      0.88        18

    accuracy                           0.98       161
   macro avg       0.99      0.89      0.93       161
weighted avg       0.98      0.98      0.97       161

Average roc_auc score: 0.9732989869137478
Average f1 score: 0.7177792757368087
```

Predicting pokemon battle victories

We have generated a dataset, based on the pokemon dataset, and a dataset with combats between two pokemons, denoted by their id, and who the winner was.

The generated datasets replaces the pokemon id with all the corrosponding features and the winner is denoted with a True/False value, where False means that the first pokemon won.

We split the combat test set and try to predict the outcome of a battle, based on the pokemons features. We do this with three models: Logistic regression, Decision tree and Random forest.

In [0]:

```
x_combats = scaler.fit_transform(combats.drop("winner", axis=1))
y_combats = combats["winner"]

x_train_combats, x_test_combats, y_train_combats, y_test_combats = train_test_split(x_com
bats, y_combats, test_size=0.2, random_state=42)
```

In [170]:

```
log_reg = LogisticRegression(solver = "liblinear")

log_reg.fit(x_train_combats, y_train_combats)

y_pred = log_reg.predict(x_test_combats)

print(confusion_matrix(y_test_combats, y_pred))
print(classification_report(y_test_combats, y_pred))
```

```
scores = cross_val_score(log_reg, x_combats, y_combats, cv=5, scoring="roc_auc")
print("Average roc_auc score:", scores.mean())

scores = cross_val_score(log_reg, x_combats, y_combats, cv=5, scoring="f1")
print("Average f1 score:", scores.mean())
```

```
[[2134 2619]
 [1646 3601]]
              precision    recall  f1-score   support

       False       0.56      0.45      0.50      4753
        True       0.58      0.69      0.63      5247

    accuracy                           0.57     10000
   macro avg       0.57      0.57      0.56     10000
weighted avg       0.57      0.57      0.57     10000

Average roc_auc score: 0.5994877391197229
Average f1 score: 0.624581749384002
```

In [171]:

```
tree = DecisionTreeClassifier()

tree.fit(x_train_combats, y_train_combats)

y_pred = tree.predict(x_test_combats)

print(confusion_matrix(y_test_combats, y_pred))
print(classification_report(y_test_combats, y_pred))

scores = cross_val_score(tree, x_combats, y_combats, cv=5, scoring="roc_auc")
print("Average roc_auc score:", scores.mean())

scores = cross_val_score(tree, x_combats, y_combats, cv=5, scoring="f1")
print("Average f1 score:", scores.mean())
```

```
[[3118 1635]
 [1664 3583]]
              precision    recall  f1-score   support

       False       0.65      0.66      0.65      4753
        True       0.69      0.68      0.68      5247

    accuracy                           0.67     10000
   macro avg       0.67      0.67      0.67     10000
weighted avg       0.67      0.67      0.67     10000

Average roc_auc score: 0.6445448872272452
Average f1 score: 0.6654180373161636
```

As we can see so far, neither the logistic regression nor the decision tree have performed particularly well. We do however see improvements when using the random forest model. Therefore we have performed hyperparameter tuning on this model

In [172]:

```
rf = RandomForestClassifier()

estimators = np.arange(50, 150, 10)
param_grid = {'n_estimators': estimators}
cv = MakeGridSearchCV(model=rf, grid=param_grid, x=x_train_combats, y=y_train_combats)
PrintGridsearchCVScore(cv)

y_pred = cv.best_estimator_.predict(x_test_combats)

print(confusion_matrix(y_test_combats, y_pred))
print(classification_report(y_test_combats, y_pred))
```

```
scores = cross_val_score(rf, x_combats, y_combats, cv=5, scoring="roc_auc")
print("Average roc_auc score:", scores.mean())

scores = cross_val_score(rf, x_combats, y_combats, cv=5, scoring="f1")
print("Average f1 score:", scores.mean())
```

```
Best parameter is: {'n_estimators': 140}
Best score is: 0.7908
[[3608 1145]
 [ 874 4373]]
              precision    recall  f1-score   support

       False       0.80      0.76      0.78      4753
        True       0.79      0.83      0.81      5247

    accuracy                           0.80     10000
   macro avg       0.80      0.80      0.80     10000
weighted avg       0.80      0.80      0.80     10000
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

```
Average roc_auc score: 0.8087550172441738
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

```
Average f1 score: 0.7142030704984579
```

Predicting pokemon types

In [0]:

```
y_type = pokedex_numeric["type1"]
x_type = scaler.fit_transform(pokedex_numeric.drop("type1", axis=1))

x_train_type, x_test_type, y_train_type, y_test_type = train_test_split(x_type, y_type,
test_size=0.2, random_state=42)
```

Predicting type with KNeighbours

As we can see from the cross validation score; Even with hyperparameter tuning, KNeighbours is not performing

very well when predicting the type of the pokemons. Atleast not when we look at the classifier report.

If we look at the cross validation score, with all our type data, we get an average accuracy score of about ~75% which isn't too bad.

This seems to indicate that our model has been overfitted to the training set, and is not performing well with other unseen datasets. even with hyperparameter tuning the accuracy remain low for the current split of the data.

In [174]:

```
neighbors = KNeighborsClassifier()

n = np.arange(5, 80)
param_grid = {'n_neighbors': n}
cv = MakeGridSearchCV(model=neighbors, grid=param_grid, x=x_train_type, y=y_train_type)
PrintGridsearchCVScore(cv)

y_pred = cv.best_estimator_.predict(x_test_type)

print(classification_report(y_test_type, y_pred))

scores = cross_val_score(neighbors, x_type, y_type, cv=5, scoring="accuracy")
print("Averace accuracy score:", scores.mean())
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:657: Warning: Th
e least populated class in y has only 3 members, which is too few. The minimum number of
members in any class cannot be less than n_splits=5.
  % (min_groups, self.n_splits)), Warning)
```

```
Best parameter is: {'n_neighbors': 9}
Best score is: 0.7578125
              precision    recall  f1-score   support

           0       0.88      0.64      0.74        11
           1       1.00      0.17      0.29         6
           2       0.62      1.00      0.77         5
           3       0.80      0.67      0.73         6
           4       0.67      1.00      0.80         4
           5       1.00      1.00      1.00        11
           6       0.79      1.00      0.88        11
           8       0.80      0.67      0.73         6
           9       1.00      0.79      0.88        14
          10       0.57      0.67      0.62         6
          11       1.00      1.00      1.00         3
          12       0.67      0.93      0.78        15
          13       0.71      0.50      0.59        10
          14       0.75      0.86      0.80        14
          15       0.71      0.50      0.59        10
          16       0.25      0.67      0.36         3
          17       0.87      0.77      0.82        26

    accuracy                           0.76       161
   macro avg       0.77      0.75      0.73       161
weighted avg       0.80      0.76      0.76       161

Averace accuracy score: 0.7606649999900285
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:814: Deprecatio
nWarning: The default of the `iid` parameter will change from True to False in version 0.
22 and will be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:657: Warning: Th
e least populated class in y has only 3 members, which is too few. The minimum number of
members in any class cannot be less than n_splits=5.
  % (min_groups, self.n_splits)), Warning)
```

Predicting type with Random forest

As we can see, the random forest model performs much better than KNeighbors that was tested before

As we can see, the random forest model performs much better than KNeighbors that was tested before.

Even though kneighbors showed signs of overfitted data, this model still performs well with the current data split. Hyper parameter tuning seems to help the accuracy further along.

In [175]:

```
rf = RandomForestClassifier()

estimators = np.arange(50, 200, 10)
param_grid = {'n_estimators': estimators}
cv = MakeGridSearchCV(model=rf, grid=param_grid, x=x_train_type, y=y_train_type)
PrintGridsearchCVScore(cv)

#rf.fit(x_train_type, y_train_type)
#y_pred = rf.predict(x_test_type)
y_pred = cv.best_estimator_.predict(x_test_type)

print(classification_report(y_test_type, y_pred))

scores = cross_val_score(rf, x_type, y_type, cv=5)
print("Accuracy:", round(scores.mean(), 2), "+/-", round(scores.std(), 2))
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:657: Warning: Th
e least populated class in y has only 3 members, which is too few. The minimum number of
members in any class cannot be less than n_splits=5.
  % (min_groups, self.n_splits)), Warning)
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:814: Deprecatio
nWarning: The default of the `iid` parameter will change from True to False in version 0.
22 and will be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:657: Warning: Th
e least populated class in y has only 3 members, which is too few. The minimum number of
members in any class cannot be less than n_splits=5.
  % (min_groups, self.n_splits)), Warning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

```
Best parameter is: {'n_estimators': 120}
Best score is: 0.934375
              precision    recall  f1-score   support

           0       0.73      1.00      0.85        11
           1       1.00      1.00      1.00         6
           2       0.83      1.00      0.91         5
           3       1.00      1.00      1.00         6
           4       1.00      1.00      1.00         4
           5       1.00      0.91      0.95        11
           6       1.00      0.91      0.95        11
           8       1.00      0.83      0.91         6
           9       0.92      0.79      0.85        14
          10       0.62      0.83      0.71         6
          11       1.00      1.00      1.00         3
          12       1.00      0.93      0.97        15
          13       1.00      0.80      0.89        10
          14       1.00      0.93      0.96        14
          15       0.82      0.90      0.86        10
          16       1.00      1.00      1.00         3
          17       0.96      1.00      0.98        26

    accuracy                           0.93       161
   macro avg       0.93      0.93      0.93       161
weighted avg       0.94      0.93      0.93       161

Accuracy: 0.88 +/- 0.03
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
```

```
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The
default value of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

predict the types of generation 7, based on the features from the previous 6 generations

Since the random forest model did such a good job at predicting types, we want to test if it can predict the types of the pokemons in genenration 7, by only training on the last 6 generations of pokemons.

Using the random forest model with hyperparameter tuning, we can predict the types of the pokemons in generation 7 with an accuracy of above 80%, based on the features from the pokemons in the previous generations. This is pretty good, but still indicate that some significant changes happened in the last generation.

In [0]:

```
generation_groups = pokedex_numeric.groupby("generation")

generation_1_6 = pd.concat([generation_groups.get_group(group) for i, group in enumerate
( generation_groups.groups) if i < 6 ])
generation_7 = generation_groups.get_group(7)
x_train = generation_1_6.drop("type1", axis=1)
y_train = generation_1_6["type1"]
x_test = generation_7.drop("type1", axis=1)
y_test = generation_7["type1"]
```

In [177]:

```
rf = RandomForestClassifier()

estimators = np.arange(10, 200, 10)
param_grid = {'n_estimators': estimators}
cv = MakeGridSearchCV(model=rf, grid=param_grid, x=x_train, y=y_train)
PrintGridsearchCVScore(cv)

#rf.fit(x_train, y_train)
y_pred = cv.best_estimator_.predict(x_test)

print(classification_report(y_test, y_pred))
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:657: Warning: Th
e least populated class in y has only 3 members, which is too few. The minimum number of
members in any class cannot be less than n_splits=5.
  % (min_groups, self.n_splits)), Warning)
```

```
Best parameter is: {'n_estimators': 80}
Best score is: 0.8904299583911235
              precision    recall  f1-score   support

           0       0.90      1.00      0.95         9
           1       0.50      1.00      0.67         1
           2       1.00      1.00      1.00         3
           3       1.00      1.00      1.00         3
           4       1.00      1.00      1.00         1
           5       1.00      1.00      1.00         3
           6       0.67      0.80      0.73         5
           8       0.67      0.50      0.57         4
           9       1.00      0.92      0.96        12
          10       0.50      1.00      0.67         2
          12       1.00      1.00      1.00        12
          13       1.00      0.50      0.67         4
          14       1.00      0.83      0.91         6
          15       1.00      1.00      1.00         4
          16       0.67      1.00      0.80         2
          17       1.00      0.89      0.94         9
```

```
    accuracy                           0.90        80
   macro avg       0.87      0.90      0.87        80
weighted avg       0.92      0.90      0.90        80
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:814: Deprecatio
nWarning: The default of the `iid` parameter will change from True to False in version 0.
22 and will be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
```