Eighth Practice ML

In this practice, we will learn **Unsupervised Learning** like **KMeans** and **Hierarchical Clustering**.
We will also learn **Feature Compression** with **PCA**.
We will use **NBC (Naive Bayes)** and **LDA (Linear Discriminant Analysis)** as models.
We will also learn how to use **D-Tale** to show data reports and work with tabular data frames.

D-Tale

▶ 99 / 1000465	date	security_id	int_val	Col0	Col1	Col2	Col3	Col4	Col5	Col6	Col7	Col8	Col9	Col10	Col11
0	2018-09-13	100000	13700556066	1.65	1.68	-0.09	0.68	2.04	1.40	-0.74	-0.15	-0.40	-0.07	0.16	0.90
1	2018-09-13	100001	32370834703	0.81	-0.68	-2.29	-0.58	1.05	0.83	-0.01	0.40	-0.63	-0.34	-1.62	1.99
2	2018-09	42	74359132292	0.27	1.39			-1.1	27	-1.4	0.28	0.04			0.61
3	2018-09-13		3555690688	0.24			-0.31		.95		0.95	1.2		-1.74	-0.48
4	2018-09-13	00004	105740597	-0.64	-1.38	0	1.0		0.44		-1.81			2.02	1.05
5	2018-09-13	100005		.00	0.1	.73				6	-0.89		1.18	0.41	1.03
6	2018-09-		30414943406	0.09		0.9			0	0.39	-1.16			1.76	1.11
7	20	00007	94115021503	0.07		0.8	0.60						0.16	-1.00	-1.23
8	2018-09	100008	7207483836	-0.03	-0.68	-0.06	-0.58	-1		-1.13	0.3	0.09	-1.13	0.07	-0.07
9	2018-09-13	100009	24054692457	0.78	0.28	0.66	0.88	0.7	-0.57	-0.51	-0.29	-0.71	0.48	-1.07	0.41
10	2018-09-13	100010	55729553089	-0.24	-0.04	-0.29	1.63	-2.35	-0.18	0.35	-1.19	1.96	-1.02	0.43	1.40
11	2018-09-13	100011	79679700690	-0.41	0.34	-1.21	0.17	-0.94	-0.35	-1.54	-0.74	-0.81	-1.05	0.57	-0.20
12	2018-09-13	100012	35404502206	0.26	0.93	0.09	-0.11	0.77	-0.41	0.25	-1.92	1.05	-1.74	-0.58	-0.36
13	2018-09-13	100013	54366289347	1.60	0.55	-1.83	-0.74	1.90	-1.42	0.52	-0.86	1.19	0.47	1.69	-0.17
14	2018-09-13	100014	29512587692	0.74	-1.22	-0.79	0.40	0.20	-0.61	-1.58	1.87	1.11	-0.48	0.21	0.12
15	2018-09-13	100015	59921981555	1.70	0.39	-0.51	-0.77	0.99	-0.69	0.04	-0.87	1.60	0.31	1.39	-0.09

D-Tale (https://www.man.com/d-tale) is a lightweight tool that allows users to view all aspects of their data while maintaining the ability to perform simple operations such as sorting, filtering, and formatting.

D-Tale was born out of a pre-existing internal solution for viewing SAS datasets but meant for users of Python Pandas data structures, e.g. Series and DataFrames.

This tool is strongly geared towards anyone who used Pandas data structures (Series, DataFrame, etc.) in their day-to-day operations and needs an easier way of visualizing data without writing cumbersome Python in an editor (which is still the way to go if you need full control).

Downloads, Imports, and Definitions

We update packages that their Colab version is too old.

In [ ]:

```
# upgrade plotly and scipy and also install dtale
!pip install --upgrade plotly
!pip install --upgrade scipy
!pip install dtale
```

```
Requirement already up-to-date: plotly in /usr/local/lib/python3.6/dist-pa
ckages (4.14.1)
Requirement already satisfied, skipping upgrade: six in /usr/local/lib/pyt
hon3.6/dist-packages (from plotly) (1.15.0)
Requirement already satisfied, skipping upgrade: retrying>=1.3.3 in /usr/l
ocal/lib/python3.6/dist-packages (from plotly) (1.3.3)
Requirement already satisfied: dtale in /usr/local/lib/python3.6/dist-pack
ages (1.29.1)
Requirement already satisfied: xarray in /usr/local/lib/python3.6/dist-pac
kages (from dtale) (0.15.1)
Requirement already satisfied: Flask-Compress in /usr/local/lib/python3.6/
dist-packages (from dtale) (1.8.0)
Requirement already satisfied: scikit-learn>='0.21.0 in /usr/local/lib/pyt
hon3.6/dist-packages (from dtale) (0.22.2.post1)
Requirement already satisfied: squarify in /usr/local/lib/python3.6/dist-p
ackages (from dtale) (0.4.3)
Requirement already satisfied: lz4; python_version > "3.0" in /usr/local/l
ib/python3.6/dist-packages (from dtale) (3.1.1)
Requirement already satisfied: pandas in /usr/local/lib/python3.6/dist-pac
kages (from dtale) (1.1.5)
Requirement already satisfied: flask-ngrok; python_version > "3.0" in /us
r/local/lib/python3.6/dist-packages (from dtale) (0.0.25)
Requirement already satisfied: plotly>=4.9.0 in /usr/local/lib/python3.6/d
ist-packages (from dtale) (4.14.1)
Requirement already satisfied: dash-bootstrap-components; python_version >
"3.0" in /usr/local/lib/python3.6/dist-packages (from dtale) (0.11.1)
Requirement already satisfied: kaleido in /usr/local/lib/python3.6/dist-pa
ckages (from dtale) (0.1.0)
Requirement already satisfied: dash>=1.5.0 in /usr/local/lib/python3.6/dis
t-packages (from dtale) (1.18.1)
Requirement already satisfied: scipy in /usr/local/lib/python3.6/dist-pack
ages (from dtale) (1.4.1)
Requirement already satisfied: dash-daq in /usr/local/lib/python3.6/dist-p
ackages (from dtale) (0.5.0)
Requirement already satisfied: ppscore; python_version >= "3.6" in /usr/lo
cal/lib/python3.6/dist-packages (from dtale) (1.1.1)
Requirement already satisfied: Flask>=1.0 in /usr/local/lib/python3.6/dist
-packages (from dtale) (1.1.2)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packag
es (from dtale) (1.15.0)
Requirement already satisfied: itsdangerous in /usr/local/lib/python3.6/di
st-packages (from dtale) (1.1.0)
Requirement already satisfied: strsimpy in /usr/local/lib/python3.6/dist-p
ackages (from dtale) (0.2.0)
Requirement already satisfied: statsmodels in /usr/local/lib/python3.6/dis
t-packages (from dtale) (0.10.2)
Requirement already satisfied: future>=0.14.0 in /usr/local/lib/python3.6/
dist-packages (from dtale) (0.16.0)
Requirement already satisfied: requests in /usr/local/lib/python3.6/dist-p
ackages (from dtale) (2.23.0)
Requirement already satisfied: dash-colorscales in /usr/local/lib/python3.
6/dist-packages (from dtale) (0.0.4)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.6/dis
t-packages (from xarray->dtale) (1.19.4)
Requirement already satisfied: setuptools>=41.2 in /usr/local/lib/python3.
6/dist-packages (from xarray->dtale) (51.0.0)
Requirement already satisfied: brotli in /usr/local/lib/python3.6/dist-pac
kages (from Flask-Compress->dtale) (1.0.9)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.6/di
st-packages (from scikit-learn>='0.21.0->dtale) (1.0.0)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/py
```

```
thon3.6/dist-packages (from pandas->dtale) (2.8.1)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.6/di
st-packages (from pandas->dtale) (2018.9)
Requirement already satisfied: retrying>=1.3.3 in /usr/local/lib/python3.
6/dist-packages (from plotly>=4.9.0->dtale) (1.3.3)
Requirement already satisfied: dash-html-components==1.1.1 in /usr/local/l
ib/python3.6/dist-packages (from dash>=1.5.0->dtale) (1.1.1)
Requirement already satisfied: dash-core-components==1.14.1 in /usr/local/
lib/python3.6/dist-packages (from dash>=1.5.0->dtale) (1.14.1)
Requirement already satisfied: dash-table==4.11.1 in /usr/local/lib/python
3.6/dist-packages (from dash>=1.5.0->dtale) (4.11.1)
Requirement already satisfied: dash-renderer==1.8.3 in /usr/local/lib/pyth
on3.6/dist-packages (from dash>=1.5.0->dtale) (1.8.3)
Requirement already satisfied: Werkzeug>=0.15 in /usr/local/lib/python3.6/
dist-packages (from Flask>=1.0->dtale) (1.0.1)
Requirement already satisfied: click>=5.1 in /usr/local/lib/python3.6/dist
-packages (from Flask>=1.0->dtale) (7.1.2)
Requirement already satisfied: Jinja2>=2.10.1 in /usr/local/lib/python3.6/
dist-packages (from Flask>=1.0->dtale) (2.11.2)
Requirement already satisfied: patsy>=0.4.0 in /usr/local/lib/python3.6/di
st-packages (from statsmodels->dtale) (0.5.1)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/di
st-packages (from requests->dtale) (2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python
3.6/dist-packages (from requests->dtale) (3.0.4)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.6/dist-packages (from requests->dtale) (1.24.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python
3.6/dist-packages (from requests->dtale) (2020.12.5)
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.
6/dist-packages (from Jinja2>=2.10.1->Flask>=1.0->dtale) (1.1.1)
```

We import our regular packages.

In [ ]:

```
# import numpy, matplotlib, etc.
import numpy as np
import pandas as pd
import seaborn as sns
import plotly.express as px
import matplotlib.pyplot as plt
import plotly.graph_objects as go

# sklearn imports
from sklearn import metrics
from sklearn import pipeline
from sklearn import linear_model
from sklearn import preprocessing
from sklearn import neural_network
from sklearn import model_selection
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline
from sklearn.feature_selection import RFECV
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import SGDRegressor
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedKFold
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
```

Data Exploration

We use the updated Pokemon (https://www.kaggle.com/takamasakato/pokemon-all-status-data) dataset.

Overview

This dataset includes 1052 Pokemon (from 1st to 8th generation).
Each sample (Pokemon) has 32 features.

Description

1. **NUMBER**
   Each Pokemon has several Pokedex numbers. It's listed in this column.
   Some Pokemon have multiple appearances, status, evolution, or generation. In these cases, it is listed with the same numbers.
2. **CODE**
   Some Pokemon have multiple appearances, status, evolution, or generation.
   If the Pokemon has multiple appearances, status, or other, the code is distinguished between each Pokemon.
3. **SERIAL**
   The number to distinguish between each Pokemon, appearance, status, or other factors.
   The number is equal to `10*NUMBER + CODE` .
4. **NAME**
   Name for each Pokemon.
5. **TYPE1**
   Each Pokemon has a type. The type is listed in this column.
6. **TYPE2**
   Some Pokemon have two types (dual type). If the Pokemon is dual type, the second type is listed in this column.
7. **COLOR**
   Each Pokemon has a color. It's listed in this column.

8. **ABILITY1**
   Each Pokemon has an ability. The ability is listed in this column.
9. **ABILITY2**
   Some Pokemon has an alternative ability. If the Pokemon has an alternative ability, it is listed in this column.
10. **HIDDEN ABILITY**
    Some Pokemon has a special ability. it is listed in this column.
11. **GENERATION**
    The number of generations for each Pokemon.
    Mega evolved Pokemon's generation number is equivalent to origin (before evolution) Pokemon.
12. **LEGENDARY**
    The Pokemon that are part of a group of incredibly rare Pokemons and often has a very powerful status, called legendary Pokemon.
    If the Pokemon is a legendary Pokemon (includes mythical Pokemons), then filled 1, else filled 0.
13. **MEGA_EVOLUTION**
    Some Pokemon can Mega Evolve.
    If the Pokemon is Mega Evolved Pokemon, then filled 1, else filled 0.
14. **HEIGHT**
    Height for each Pokemon.
15. **WEIGHT**
    Weight for each Pokemon.
16. **HP (Hit Point)**
    One of Pokemon's base stats. It determines how much damage a Pokemon can receive before fainting.
17. **ATK (Physical Attack)**
    One of Pokemon's base stats. It determines how much damage the Pokemon deals when using a Physical Move.
18. **DEF (Physical Defense)**
    One of Pokemon's base stats. It determines how much damage the Pokemon receives when it's hit with a Physical Move.
19. **SP_ATK (Special Attack)**
    One of Pokemon's base stats. It determines how much damage the Pokemon deals when using a Special Move.
20. **SP_DEF (Special Defense)**
    One of Pokemon's base stats. It determines how much damage the Pokemon receives when it's hit with a Special Move.
21. **SPD (Speed)**
    One of Pokemon's base stats. It determines the order of Pokemon that can act in battle.
22. **TOTAL**
    Summed up each base stats.
23. **CAPTURE_RATE**
    It is that indicates the ease of catching. The higher the value, the easier it is to catch.
24. **BASEEGGCYCLE**
    It is that represents the number of steps required for a Pokemon egg to hatch.
25. **BASE_EXP**
    When the Pokémon is defeated in battle, it will give EXP values to the Pokémon that participated in the battle against it.
26. **EFFORT_HP**
    When the Pokémon is defeated in battle, it will give effort HP values to the Pokémon that participated in the battle against it.
27. **EFFORT_ATK**
    When the Pokémon is defeated in battle, it will give effort attack values to the Pokémon that participated in the battle against it.

28. **EFFORT_DEF**
    When the Pokémon is defeated in battle, it will give effort defense values to the Pokémon that participated in the battle against it.
29. **EFFORTSPATK**
    When the Pokémon is defeated in battle, it will give effort special attack values to the Pokémon that participated in the battle against it.
30. **EFFORTSPDEF**
    When the Pokémon is defeated in battle, it will give effort special defense values to the Pokémon that participated in the battle against it.
31. **EFFORT_SPD**
    When the Pokémon is defeated in battle, it will give effort speed values to the Pokémon that participated in the battle against it.
32. **EFFORT_TOTAL**
    - Summed up each effort values.

Quote

[bulbapedia (https://bulbapedia.bulbagarden.net/wiki/Main_Page)](https://bulbapedia.bulbagarden.net/wiki/Main_Page)
[pokemon database (https://pokemondb.net/)](https://pokemondb.net/)

Let's download the dataset from Github and explore it with Pandas tools.

In [ ]:

```
# download pokemon.csv file from Github
!wget https://gist.githubusercontent.com/aviasd/070b68d1068e92f059e143a1f4655e77/raw/2707de87c3f6ddc29246658bda5b1afc1058a4f7/Pokedex_Ver7.csv
```

```
--2021-01-01 00:00:12--  https://gist.githubusercontent.com/aviasd/070b68d1068e92f059e143a1f4655e77/raw/2707de87c3f6ddc29246658bda5b1afc1058a4f7/Pokedex_Ver7.csv
Resolving gist.githubusercontent.com (gist.githubusercontent.com)... 151.101.0.133, 151.101.64.133, 151.101.128.133, ...
Connecting to gist.githubusercontent.com (gist.githubusercontent.com)|151.101.0.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 128917 (126K) [text/plain]
Saving to: ‘Pokedex_Ver7.csv.3’

Pokedex_Ver7.csv.3  100%[===================>] 125.90K  --.-KB/s    in 0.02s

2021-01-01 00:00:12 (7.30 MB/s) - ‘Pokedex_Ver7.csv.3’ saved [128917/128917]
```

In [ ]:

```
# load the Pokedex_Ver7.csv file
pokemon_df = pd.read_csv('Pokedex_Ver7.csv')
pokemon_df
```

Out[ ]:

	NUMBER	CODE	SERIAL	NAME	TYPE1	TYPE2	COLOR	ABILITY1	ABILITY2
0	1	1	11	Bulbasaur	Grass	Poison	Green	Overgrow	NaN
1	2	1	21	Ivysaur	Grass	Poison	Green	Overgrow	NaN
2	3	1	31	Venusaur	Grass	Poison	Green	Overgrow	NaN
3	3	2	32	Mega Venusaur	Grass	Poison	Green	Thick Fat	NaN
4	4	1	41	Charmander	Fire	NaN	Red	Blaze	NaN
...	...	...	...	...	...	...	...	...	...
1047	896	1	8961	Glastrier	Ice	NaN	White	Chilling Neigh	NaN
1048	897	1	8971	Spectrier	Ghost	NaN	Black	Grim Neigh	NaN
1049	898	1	8981	Calyrex	Psychic	Grass	Green	Unnerve	NaN
1050	898	2	8982	Calyrex	Psychic	Ice	White	As One	NaN
1051	898	3	8983	Calyrex	Psychic	Ghost	Black	As One	NaN

1052 rows × 32 columns

In [ ]:

```
# show pokemon_df info
pokemon_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1052 entries, 0 to 1051
Data columns (total 32 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   NUMBER           1052 non-null   int64  
 1   CODE             1052 non-null   int64  
 2   SERIAL           1052 non-null   int64  
 3   NAME             1052 non-null   object 
 4   TYPE1            1052 non-null   object 
 5   TYPE2            558 non-null    object 
 6   COLOR            1052 non-null   object 
 7   ABILITY1         1052 non-null   object 
 8   ABILITY2         523 non-null    object 
 9   ABILITY HIDDEN   819 non-null    object 
 10  GENERATION       1052 non-null   int64  
 11  LEGENDARY        1052 non-null   int64  
 12  MEGA_EVOLUTION   1052 non-null   int64  
 13  HEIGHT           1052 non-null   float64
 14  WEIGHT           1052 non-null   float64
 15  HP               1052 non-null   int64  
 16  ATK              1052 non-null   int64  
 17  DEF              1052 non-null   int64  
 18  SP_ATK           1052 non-null   int64  
 19  SP_DEF           1052 non-null   int64  
 20  SPD              1052 non-null   int64  
 21  TOTAL            1052 non-null   int64  
 22  CAPTURE_RATE     1052 non-null   int64  
 23  BASE_EGG_CYCLES  1052 non-null   int64  
 24  BASE_EXP         1052 non-null   int64  
 25  EFFORT_HP        1052 non-null   int64  
 26  EFFORT_ATK       1052 non-null   int64  
 27  EFFORT_DEF       1052 non-null   int64  
 28  EFFORT_SP_ATK    1052 non-null   int64  
 29  EFFORT_SP_DEF    1052 non-null   int64  
 30  EFFORT_SPD       1052 non-null   int64  
 31  EFFORT_TOTAL     1052 non-null   int64  
dtypes: float64(2), int64(23), object(7)
memory usage: 263.1+ KB
```

In [ ]:

```
# show pokemon_df description
pokemon_df.describe()
```

Out[ ]:

	NUMBER	CODE	SERIAL	GENERATION	LEGENDARY	MEGA_EVOLUTIO
count	1052.000000	1052.000000	1052.000000	1052.000000	1052.000000	1052.0000(
mean	441.332700	1.191065	4414.518061	4.274715	0.120722	0.0475(
std	260.673105	0.529434	2606.720549	2.275842	0.325959	0.2128(
min	1.000000	1.000000	11.000000	1.000000	0.000000	0.0000(
25%	214.750000	1.000000	2148.750000	2.000000	0.000000	0.0000(
50%	436.500000	1.000000	4366.000000	4.000000	0.000000	0.0000(
75%	666.250000	1.000000	6663.500000	6.000000	0.000000	0.0000(
max	898.000000	6.000000	8983.000000	8.000000	1.000000	1.0000(

We will use D-Tale (https://github.com/man-group/dtale) EDA (Exploratory Data Analysis) tool.

In [ ]:

```
import dtale
import dtale.app as dtale_app

# dtale_app.USE_NGROK = True # Use this option if you have a NGROK account (It's free t
o create one), it's the way to use D-Tale on Kaggle
dtale_app.USE_COLAB = True # This is the way to use D-Tale on Google Colab (It's the be
tter option)
dtale.show(pokemon_df, ignore_duplicate=True)
```

```
2021-01-01 00:00:14,289 - INFO     - NumExpr defaulting to 2 threads.
```

Out[ ]:

```
http://bec52c30e420.ngrok.io/dtale/main/1
```

We can click on the link and go to the D-Tale client.
This client is running on a server on our machine.
We can show statistics and manipulate the dataset.
We can also generate code or export it as a CSV file.

Let's prepare our dataset (encoding and filling empty values).

In [ ]:

```
# encdoe and fill empty values
categorical_cols = pokemon_df.select_dtypes(include=['object', 'bool']).columns
numerical_cols = pokemon_df.select_dtypes(include=['int64', 'float64']).columns
ordinal_cols = pokemon_df.select_dtypes(include=['int64']).columns
all_cols = categorical_cols.tolist() + numerical_cols.tolist() + ordinal_cols.tolist()

pokemon_df_cp = pokemon_df.copy()
pokemon_df_cp.fillna('empty', inplace=True)

oe = OrdinalEncoder().fit(pokemon_df_cp[categorical_cols])

pokemon_df_cp[categorical_cols] = oe.transform(pokemon_df_cp[categorical_cols])
pokemon_df_cp
```

Out[ ]:

	NUMBER	CODE	SERIAL	NAME	TYPE1	TYPE2	COLOR	ABILITY1	ABILITY2	ABILI HIDD
0	1	1	11	82.0	9.0	13.0	3.0	117.0	126.0	1
1	2	1	21	372.0	9.0	13.0	3.0	117.0	126.0	1
2	3	1	31	887.0	9.0	13.0	3.0	117.0	126.0	1
3	3	2	32	525.0	9.0	13.0	3.0	193.0	126.0	15
4	4	1	41	107.0	6.0	18.0	7.0	15.0	126.0	12
...	...	...	...	...	...	...	...	...	...	
1047	896	1	8961	292.0	11.0	18.0	8.0	18.0	126.0	15
1048	897	1	8971	783.0	8.0	18.0	0.0	65.0	126.0	15
1049	898	1	8981	90.0	14.0	9.0	3.0	201.0	126.0	15
1050	898	2	8982	90.0	14.0	11.0	8.0	5.0	126.0	15
1051	898	3	8983	90.0	14.0	8.0	0.0	5.0	126.0	15

1052 rows × 32 columns

We can reconstruct the original df with the categories from the encoder.

In [ ]:

```
# show categories of encoder
oe.categories_
```

Out[ ]:

```
[array(['Abomasnow', 'Abra', 'Absol', 'Accelgor', 'Aegislash',
        'Aerodactyl', 'Aggron', 'Aipom', 'Alakazam', 'Alcremie',
        'Alomomola', 'Altaria', 'Amaura', 'Ambipom', 'Amoonguss',
        'Ampharos', 'Anorith', 'Appletun', 'Applin', 'Araquanid', 'Arbok',
        'Arcanine', 'Arceus', 'Archen', 'Archeops', 'Arctovish',
        'Arctozolt', 'Ariados', 'Armaldo', 'Aromatisse', 'Aron',
        'Arrokuda', 'Articuno', 'Audino', 'Aurorus', 'Avalugg', 'Axew',
        'Azelf', 'Azumarill', 'Azurill', 'Bagon', 'Baltoy', 'Banette',
        'Barbaracle', 'Barboach', 'Barraskewda', 'Basculin', 'Bastiodon',
        'Bayleef', 'Beartic', 'Beautifly', 'Beedrill', 'Beheeyem',
        'Beldum', 'Bellossom', 'Bellsprout', 'Bergmite', 'Bewear',
        'Bibarel', 'Bidoof', 'Binacle', 'Bisharp', 'Blacephalon',
        'Blastoise', 'Blaziken', 'Blipbug', 'Blissey', 'Blitzle',
        'Boldore', 'Boltund', 'Bonsly', 'Bouffalant', 'Bounsweet',
        'Braixen', 'Braviary', 'Breloom', 'Brionne', 'Bronzong', 'Bronzo
r',
        'Bruxish', 'Budew', 'Buizel', 'Bulbasaur', 'Buneary', 'Bunnelby',
        'Burmy', 'Butterfree', 'Buzzwole', 'Cacnea', 'Cacturne', 'Calyre
x',
        'Camerupt', 'Carbink', 'Carkol', 'Carnivine', 'Carracosta',
        'Carvanha', 'Cascoon', 'Castform', 'Caterpie', 'Celebi',
        'Celesteela', 'Centiskorch', 'Chandelure', 'Chansey', 'Charizard',
        'Charjabug', 'Charmander', 'Charmeleon', 'Chatot', 'Cherrim',
        'Cherubi', 'Chesnaught', 'Chespin', 'Chewtle', 'Chikorita',
        'Chimchar', 'Chimecho', 'Chinchou', 'Chingling', 'Cinccino',
        'Cinderace', 'Clamperl', 'Clauncher', 'Clawitzer', 'Claydol',
        'Clefable', 'Clefairy', 'Cleffa', 'Clobbopus', 'Cloyster',
        'Coalossal', 'Cobalion', 'Cofagrigus', 'Combee', 'Combusken',
        'Comfey', 'Conkeldurr', 'Conviknight', 'Convisquire', 'Copperaja
h',
        'Corphish', 'Corsola', 'Cosmoem', 'Cosmog', 'Cottonee',
        'Crabominable', 'Crabrawler', 'Cradily', 'Cramorant', 'Cranidos',
        'Crawdaunt', 'Cresselia', 'Croagunk', 'Crobat', 'Croconaw',
        'Crustle', 'Cryogonal', 'Cubchoo', 'Cubone', 'Cufant', 'Cursola',
        'Cutiefly', 'Cyndaquil', 'Darkrai', 'Darmanitan', 'Dartrix',
        'Darumaka', 'Decidueye', 'Dedenne', 'Deerling', 'Deino',
        'Delcatty', 'Delibird', 'Delphox', 'Deoxys', 'Dewgong', 'Dewott',
        'Dewpider', 'Dhelmise', 'Dialga', 'Diancie', 'Diggersby',
        'Diglett', 'Ditto', 'Dodrio', 'Doduo', 'Donphan', 'Dottler',
        'Doublade', 'Dracovish', 'Dracozolt', 'Dragalge', 'Dragapult',
        'Dragonair', 'Dragonite', 'Drakloak', 'Drampa', 'Drapion',
        'Dratini', 'Drednaw', 'Dreepy', 'Drifblim', 'Drifloon', 'Drilbur',
        'Drizzile', 'Drowzee', 'Druddigon', 'Dubwool', 'Ducklett',
        'Dugtrio', 'Dunsparce', 'Duosion', 'Duraludon', 'Durant',
        'Dusclops', 'Dusknoir', 'Duskull', 'Dustox', 'Dwebble',
        'Eelektrik', 'Eelektross', 'Eevee', 'Eiscue', 'Ekans', 'Eldegoss',
        'Electabuzz', 'Electivire', 'Electrike', 'Electrode', 'Elekid',
        'Elgyem', 'Emboar', 'Emolga', 'Empoleon', 'Entei', 'Escavalier',
        'Espeon', 'Espurr', 'Eternatus', 'Excadrill', 'Exeggcute',
        'Exeggutor', 'Exploud', 'Falinks', "Farfetch'd", 'Fearow',
        'Feebas', 'Fennekin', 'Feraligatr', 'Ferroseed', 'Ferrothorn',
        'Finneon', 'Flaaffy', 'Flabebe', 'Flapple', 'Flareon',
        'Fletchinder', 'Fletchling', 'Floatzel', 'Floette', 'Florges',
        'Flygon', 'Fomantis', 'Foongus', 'Forretress', 'Fraxure',
        'Frillish', 'Froakie', 'Frogadier', 'Froslass', 'Frosmoth',
        'Furfrou', 'Furret', 'Gabite', 'Gallade', 'Galvantula', 'Garbodo
r',
        'Garchomp', 'Gardevoir', 'Gastly', 'Gastrodon', 'Genesect',
        'Gengar', 'Geodude', 'Gible', 'Gigalith', 'Girafarig', 'Giratina',
```

```
        'Glaceon', 'Glalie', 'Glameow', 'Glastrier', 'Gligar', 'Gliscor',
        'Gloom', 'Gogoat', 'Golbat', 'Goldeen', 'Golduck', 'Golem',
        'Golett', 'Golisopod', 'Golurk', 'Goodra', 'Goomy', 'Gorebyss',
        'Gossifleur', 'Gothita', 'Gothitelle', 'Gothorita', 'Gourgeist',
        'Granbull', 'Grapploct', 'Graveler', 'Greedent', 'Greninja',
        'Grimer', 'Grimmsnarl', 'Grookey', 'Grotle', 'Groudon', 'Grovyle',
        'Growlithe', 'Grubbin', 'Grumpig', 'Gulpin', 'Gumshoos', 'Gurdur
r',
        'Guzzlord', 'Gyarados', 'Hakamo-o', 'Happiny', 'Hariyama',
        'Hatenna', 'Hatterene', 'Hattrem', 'Haunter', 'Hawlucha',
        'Haxorus', 'Heatmor', 'Heatran', 'Heliolisk', 'Helioptile',
        'Heracross', 'Herdier', 'Hippopotas', 'Hippowdon', 'Hitmonchan',
        'Hitmonlee', 'Hitmontop', 'Ho-Oh', 'Honchkrow', 'Honedge', 'Hoop
a',
        'Hoothoot', 'Hoppip', 'Horsea', 'Houndoom', 'Houndour', 'Huntail',
        'Hydreigon', 'Hypno', 'Igglybuff', 'Illumise', 'Impidimp',
        'Incineroar', 'Indeedee F', 'Indeedee M', 'Infernape', 'Inkay',
        'Inteleon', 'Ivysaur', 'Jangmo-o', 'Jellicent', 'Jigglypuff',
        'Jirachi', 'Jolteon', 'Joltik', 'Jumpluff', 'Jynx', 'Kabuto',
        'Kabutops', 'Kadabra', 'Kakuna', 'Kangaskhan', 'Karrablast',
        'Kartana', 'Kecleon', 'Keldeo', 'Kingdra', 'Kingler', 'Kirlia',
        'Klang', 'Klefki', 'Klink', 'Klinklang', 'Koffing', 'Komala',
        'Kommo-o', 'Krabby', 'Kricketot', 'Kricketune', 'Krokorok',
        'Krookodile', 'Kubfu', 'Kyogre', 'Kyurem', 'Lairon', 'Lampent',
        'Landorus', 'Lanturn', 'Lapras', 'Larvesta', 'Larvitar', 'Latias',
        'Latios', 'Leafeon', 'Leavanny', 'Ledian', 'Ledyba', 'Lickilicky',
        'Lickitung', 'Liepard', 'Lileep', 'Lilligant', 'Lillipup',
        'Linoone', 'Litleo', 'Litten', 'Litwick', 'Lombre', 'Lopunny',
        'Lotad', 'Loudred', 'Lucario', 'Ludicolo', 'Lugia', 'Lumineon',
        'Lunala', 'Lunatone', 'Lurantis', 'Luvdisc', 'Luxio', 'Luxray',
        'Lycanroc', 'Machamp', 'Machoke', 'Machop', 'Magby', 'Magcargo',
        'Magearna', 'Magikarp', 'Magmar', 'Magmortar', 'Magnemite',
        'Magneton', 'Magnezone', 'Makuhita', 'Malamar', 'Mamoswine',
        'Manaphy', 'Mandibuzz', 'Manectric', 'Mankey', 'Mantine',
        'Mantyke', 'Maractus', 'Mareanie', 'Mareep', 'Marill', 'Marowak',
        'Marshadow', 'Marshtomp', 'Masquerain', 'Mawile', 'Medicham',
        'Meditite', 'Mega Abomasnow', 'Mega Absol', 'Mega Aerodactyl',
        'Mega Aggron', 'Mega Alakazam', 'Mega Altaria', 'Mega Ampharos',
        'Mega Audino', 'Mega Banette', 'Mega Beedrill', 'Mega Blastoise',
        'Mega Blaziken', 'Mega Camerupt', 'Mega Charizard X',
        'Mega Charizard Y', 'Mega Diancie', 'Mega Gallade',
        'Mega Garchomp', 'Mega Gardevoir', 'Mega Gengar', 'Mega Glalie',
        'Mega Gyarados', 'Mega Heracross', 'Mega Houndoom',
        'Mega Kangaskhan', 'Mega Latias', 'Mega Latios', 'Mega Lopunny',
        'Mega Lucario', 'Mega Manectric', 'Mega Mawile', 'Mega Medicham',
        'Mega Metagross', 'Mega Mewtwo X', 'Mega Mewtwo Y', 'Mega Pidgeo
t',
        'Mega Pinsir', 'Mega Rayquaza', 'Mega Sableye', 'Mega Salamence',
        'Mega Sceptile', 'Mega Scizor', 'Mega Sharpedo', 'Mega Slowbro',
        'Mega Steelix', 'Mega Swampert', 'Mega Tyranitar', 'Mega Venusau
r',
        'Meganium', 'Melmetal', 'Meloetta', 'Meltan', 'Meowstic F',
        'Meowstic M', 'Meowth', 'Mesprit', 'Metagross', 'Metang',
        'Metapod', 'Mew', 'Mewtwo', 'Mienfoo', 'Mienshao', 'Mightyena',
        'Milcery', 'Milotic', 'Miltank', 'Mime Jr.', 'Mimikyu', 'Minccin
o',
        'Minior', 'Minun', 'Misdreavus', 'Mismagius', 'Moltres',
        'Monferno', 'Morelull', 'Morgrem', 'Morpeko', 'Mothim', 'Mr. Mim
e',
        'Mr. Rime', 'Mudbray', 'Mudkip', 'Mudsdale', 'Muk', 'Munchlax',
        'Munna', 'Murkrow', 'Musharna', 'Naganadel', 'Natu', 'Necrozma',
```

```
        'Nickit', 'Nidoking', 'Nidoqueen', 'Nidoran F', 'Nidoran M',
        'Nidorina', 'Nidorino', 'Nihilego', 'Nincada', 'Ninetales',
        'Ninjask', 'Noctowl', 'Noibat', 'Noivern', 'Nosepass', 'Numel',
        'Nuzleaf', 'Obstagoon', 'Octillery', 'Oddish', 'Omanyte',
        'Omastar', 'Onix', 'Oranguru', 'Orbeetle', 'Oricorio', 'Oshawott',
        'Pachirisu', 'Palkia', 'Palossand', 'Palpitoad', 'Pancham',
        'Pangoro', 'Panpour', 'Pansage', 'Pansear', 'Paras', 'Parasect',
        'Passimian', 'Patrat', 'Pawniard', 'Pelipper', 'Perrserker',
        'Persian', 'Petilil', 'Phanpy', 'Phantump', 'Pheromosa', 'Phione',
        'Pichu', 'Pidgeot', 'Pidgeotto', 'Pidgey', 'Pidove', 'Pignite',
        'Pikachu', 'Pikipek', 'Piloswine', 'Pincurchin', 'Pineco',
        'Pinsir', 'Piplup', 'Plusle', 'Poipole', 'Politoed', 'Poliwag',
        'Poliwhirl', 'Poliwrath', 'Polteageist', 'Ponyta', 'Poochyena',
        'Popplio', 'Porygon', 'Porygon-Z', 'Porygon2', 'Primal Groudon',
        'Primal Kyogre', 'Primarina', 'Primeape', 'Prinplup', 'Probopass',
        'Psyduck', 'Pumpkaboo', 'Pupitar', 'Purrloin', 'Purugly', 'Pyroa
r',
        'Pyukumuku', 'Quagsire', 'Quilava', 'Quilladin', 'Qwilfish',
        'Raboot', 'Raichu', 'Raikou', 'Ralts', 'Rampardos', 'Rapidash',
        'Raticate', 'Rattata', 'Rayquaza', 'Regice', 'Regidrago',
        'Regieleki', 'Regigigas', 'Regirock', 'Registeel', 'Relicanth',
        'Remoraid', 'Reshiram', 'Reuniclus', 'Rhydon', 'Rhyhorn',
        'Rhyperior', 'Ribombee', 'Rillaboom', 'Riolu', 'Rockruff',
        'Roggenrola', 'Rolycoly', 'Rookidee', 'Roselia', 'Roserade',
        'Rotom', 'Rowlet', 'Rufflet', 'Runerigus', 'Sableye', 'Salamence',
        'Salandit', 'Salazzle', 'Samurott', 'Sandaconda', 'Sandile',
        'Sandshrew', 'Sandslash', 'Sandygast', 'Sawk', 'Sawsbuck',
        'Scatterbug', 'Sceptile', 'Scizor', 'Scolipede', 'Scorbunny',
        'Scrafty', 'Scraggy', 'Scyther', 'Seadra', 'Seaking', 'Sealeo',
        'Seedot', 'Seel', 'Seismitoad', 'Sentret', 'Serperior', 'Servine',
        'Seviper', 'Sewaddle', 'Sharpedo', 'Shaymin', 'Shedinja',
        'Shelgon', 'Shellder', 'Shellos', 'Shelmet', 'Shieldon', 'Shiftr
y',
        'Shiinotic', 'Shinx', 'Shroomish', 'Shuckle', 'Shuppet',
        'Sigilyph', 'Silcoon', 'Silicobra', 'Silvally', 'Simipour',
        'Simisage', 'Simisear', 'Sinistea', "Sirfetch'd", 'Sizzlipede',
        'Skarmory', 'Skiddo', 'Skiploom', 'Skitty', 'Skorupi', 'Skrelp',
        'Skuntank', 'Skwovet', 'Slaking', 'Slakoth', 'Sliggoo', 'Slowbro',
        'Slowking', 'Slowpoke', 'Slugma', 'Slurpuff', 'Smeargle',
        'Smoochum', 'Sneasel', 'Snivy', 'Snom', 'Snorlax', 'Snorunt',
        'Snover', 'Snubbull', 'Sobble', 'Solgaleo', 'Solosis', 'Solrock',
        'Spearow', 'Spectrier', 'Spewpa', 'Spheal', 'Spinarak', 'Spinda',
        'Spiritomb', 'Spoink', 'Spritzee', 'Squirtle', 'Stakataka',
        'Stantler', 'Staraptor', 'Staravia', 'Starly', 'Starmie', 'Stary
u',
        'Steelix', 'Steenee', 'Stonjourner', 'Stoutland', 'Stufful',
        'Stunfisk', 'Stunky', 'Sudowoodo', 'Suicune', 'Sunflora',
        'Sunkern', 'Surskit', 'Swablu', 'Swadloon', 'Swalot', 'Swampert',
        'Swanna', 'Swellow', 'Swinub', 'Swirlix', 'Swoobat', 'Sylveon',
        'Taillow', 'Talonflame', 'Tangela', 'Tangrowth', 'Tapu Bulu',
        'Tapu Fini', 'Tapu Koko', 'Tapu Lele', 'Tauros', 'Teddiursa',
        'Tentacool', 'Tentacruel', 'Tepig', 'Terrakion', 'Thievul',
        'Throh', 'Thundurus', 'Thwackey', 'Timburr', 'Tirtouga',
        'Togedemaru', 'Togekiss', 'Togepi', 'Togetic', 'Torchic',
        'Torkoal', 'Tornadus', 'Torracat', 'Torterra', 'Totodile',
        'Toucannon', 'Toxapex', 'Toxel', 'Toxicroak', 'Toxtricity',
        'Tranquill', 'Trapinch', 'Treecko', 'Trevenant', 'Tropius',
        'Trubbish', 'Trumbeak', 'Tsareena', 'Turtonator', 'Turtwig',
        'Tympole', 'Tynamo', 'Type: Null', 'Typhlosion', 'Tyranitar',
        'Tyrantrum', 'Tyrogue', 'Tyrunt', 'Umbreon', 'Unfezant', 'Unown',
        'Ursaring', 'Urshifu', 'Uxie', 'Vanillish', 'Vanillite',
```

```
        'Vanilluxe', 'Vaporeon', 'Venipede', 'Venomoth', 'Venonat',
        'Venusaur', 'Vespiquen', 'Vibrava', 'Victini', 'Victreebel',
        'Vigoroth', 'Vikavolt', 'Vileplume', 'Virizion', 'Vivillon',
        'Volbeat', 'Volcanion', 'Volcarona', 'Voltorb', 'Vullaby',
        'Vulpix', 'Wailmer', 'Wailord', 'Walrein', 'Wartortle', 'Watchog',
        'Weavile', 'Weedle', 'Weepinbell', 'Weezing', 'Whimsicott',
        'Whirlipede', 'Whiscash', 'Whismur', 'Wigglytuff', 'Wimpod',
        'Wingull', 'Wishiwashi', 'Wobbuffet', 'Woobat', 'Wooloo', 'Woope
r',
        'Wormadam', 'Wurmple', 'Wynaut', 'Xatu', 'Xerneas', 'Xurkitree',
        'Yamask', 'Yamper', 'Yanma', 'Yanmega', 'Yungoos', 'Yveltal',
        'Zacian', 'Zamazanta', 'Zangoose', 'Zapdos', 'Zarude', 'Zebstrik
a',
        'Zekrom', 'Zeraora', 'Zigzagoon', 'Zoroark', 'Zorua', 'Zubat',
        'Zweilous', 'Zygarde'], dtype=object),
 array(['Bug', 'Dark', 'Dragon', 'Electric', 'Fairy', 'Fighting', 'Fire',
        'Flying', 'Ghost', 'Grass', 'Ground', 'Ice', 'Normal', 'Poison',
        'Psychic', 'Rock', 'Steel', 'Water'], dtype=object),
 array(['Bug', 'Dark', 'Dragon', 'Electric', 'Fairy', 'Fighting', 'Fire',
        'Flying', 'Ghost', 'Grass', 'Ground', 'Ice', 'Normal', 'Poison',
        'Psychic', 'Rock', 'Steel', 'Water', 'empty'], dtype=object),
 array(['Black', 'Blue', 'Brown', 'Green', 'Grey', 'Pink', 'Purple', 'Re
d',
        'White', 'Yellow'], dtype=object),
 array(['Adaptability', 'Aerilate', 'Aftermath', 'Air Lock',
        'Anticipation', 'As One', 'Aura Break', 'Bad Dreams', 'Battery',
        'Battle Armor', 'Battle Bond', 'Beast Boost', 'Bell Fetch',
        'Berserk', 'Big Pecks', 'Blaze', 'Bulletproof', 'Cheek Pouch',
        'Chilling Neigh', 'Chlorophyll', 'Clear Body', 'Color Change',
        'Comatose', 'Competitive', 'Compound Eyes', 'Contrary',
        'Corrosion', 'Cotton Down', 'Curious Medicine', 'Cursed Body',
        'Cute Charm', 'Damp', 'Dancer', 'Dark Aura', 'Dauntless Shield',
        'Dazzling', 'Defeatist', 'Defiant', 'Delta Stream',
        'Desolate Land', 'Disguise', 'Download', "Dragon's Maw", 'Drizzl
e',
        'Drought', 'Dry Skin', 'Early Bird', 'Effect Spore',
        'Electric Surge', 'Emergency Exit', 'Fairy Aura', 'Filter',
        'Flame Body', 'Flash Fire', 'Flower Gift', 'Flower Veil', 'Fluff
y',
        'Forecast', 'Forewarn', 'Frisk', 'Full Metal Body', 'Fur Coat',
        'Gluttony', 'Gorilla Tactics', 'Grassy Surge', 'Grim Neigh',
        'Gulp Missile', 'Guts', 'Healer', 'Honey Gather', 'Huge Power',
        'Hunger Switch', 'Hustle', 'Hydration', 'Hyper Cutter', 'Ice Bod
y',
        'Ice Face', 'Illuminate', 'Illusion', 'Immunity', 'Innards Out',
        'Inner Focus', 'Insomnia', 'Intimidate', 'Intrepid Sword',
        'Iron Barbs', 'Iron Fist', 'Justified', 'Keen Eye', 'Leaf Guard',
        'Leaf guard', 'Levitate', 'Light Metal', 'Lightning Rod', 'Limbe
r',
        'Limber ', 'Liquid Ooze', 'Magic Bounce', 'Magician',
        'Magma Armor', 'Magnet Pull', 'Marvel Scale', 'Mega Launcher',
        'Merciness', 'Mimicry', 'Minus', 'Misty Surge', 'Mold Breaker',
        'Motor Drive', 'Multitype', 'Mummy', 'Natural Cure',
        'Natural cure', 'Neuroforce', 'No Guard', 'Oblivious', 'Overcoat',
        'Overgrow', 'Own Tempo', 'Parental Bond', 'Pickup', 'Pixilate',
        'Plus', 'Poison Point', 'Poison Touch', 'Power Construct',
        'Power Spot', 'Prankster', 'Pressure', 'Primordial Sea',
        'Prism Armor', 'Psychic Surge', 'Punk Rock', 'Pure Power',
        'Quick Draw', 'RKS System', 'Rattled', 'Reciever', 'Reckless',
        'Refrigerate', 'Regenerator', 'Ripen', 'Rivalry', 'Rock Head',
        'Rough Skin', 'Run Away', 'Sand Force', 'Sand Rush', 'Sand Spit',
```

```
        'Sand Stream', 'Sand Veil', 'Sap Sipper', 'Schooling', 'Scrappy',
        'Serene Grace', 'Shadow Shield', 'Shadow Tag', 'Shed Skin',
        'Sheer Force', 'Shell Armor', 'Shield Dust', 'Shields Down',
        'Simple', 'Skill Link', 'Slow Start', 'Snow Cloak', 'Snow Warnin
g',
        'Solar Power', 'Solid Rock', 'Soul-Heart', 'Soundproof',
        'Speed Boost', 'Stakeout', 'Stance Change', 'Stanfisk', 'Static',
        'Steadfast', 'Steam Engine', 'Steelworker', 'Stench',
        'Sticky Hold', 'Strong Jaw', 'Sturdy', 'Suction Cups',
        'Surge Surfer', 'Swarm', 'Sweet Veil', 'Swift Swim', 'Synchroniz
e',
        'Tangled Feet', 'Technician', 'Telepathy', 'Terabolt', 'Thick Fa
t',
        'Torrent', 'Tough Claws', 'Trace', 'Transistor', 'Truant',
        'Turboblaze', 'Unaware', 'Unnerve', 'Unseen Fist', 'Victory Star',
        'Vital Spilit', 'Volt Absorb', 'Wandering Spilit', 'Water Absorb',
        'Water Bubble', 'Water Compaction', 'Water Veil', 'Watet Absorb',
        'Weak Armor', 'White Smoke', 'Wimp Out', 'Wonder Guard',
        'Wonder Skin', 'Zen Mode'], dtype=object),
 array(['Adaptability', 'Aftermath', 'Anger Point', 'Anticipation',
        'Arena Trap', 'Battle Armor', 'Big Pecks', 'Cheek Pouch',
        'Chlorophyll', 'Cloud Nine', 'Competitive', 'Compound Eyes',
        'Cursed Body', 'Damp', 'Download', 'Drizzle', 'Drought',
        'Dry Skin', 'Early Bird', 'Effect Spore', 'Filter', 'Flaash Fire',
        'Flame Body', 'Flash Fire', 'Forewarn', 'Frisk', 'Gluttony',
        'Guts', 'Heatproof', 'Heavy Metal', 'Huge Power', 'Hustle',
        'Hydration', 'Ice Body', 'Illuminate', 'Infiltrator',
        'Inner Focus', 'Insomnia', 'Intimidate', 'Iron Fist', 'Keen Eye',
        'Klutz', 'Leaf Guard', 'Lightning Rod', 'Limber', 'Liquid Ooze',
        'Magic Guard', 'Magnet Pull', 'Minus', 'Mold Breaker',
        'Motor Drive', 'Moxie', 'Natural Cure', 'Neutralizing Gas',
        'No Guard', 'Normalize', 'Oblivious', 'Overcoat', 'Own Tempo',
        'Pastel Veil', 'Pickup', 'Plus', 'Poison Heal', 'Poison Point',
        'Poison Touch', 'Power Construct', 'Pressure', 'Queenly Majesty',
        'Quick Feet', 'Rain Dish', 'Reckless', 'Regenerator', 'Rivalry',
        'Rock Head', 'Run Away', 'Sand Force', 'Sand Rush', 'Sand Stream',
        'Sand Veil', 'Sap Sipper', 'Scrappy', 'Screen Cleaner',
        'Serene Grace', 'Shed Skin', 'Sheer Force', 'Shell Armor',
        'Shield Dust', 'Simple', 'Skill Link', 'Slush Rush', 'Sniper',
        'Snow Cloak', 'Snow Warning', 'Solar Power', 'Solid Rock',
        'Soundproof', 'Stall', 'Stamina', 'Static', 'Steadfast',
        'Sticky Hold', 'Storm Drain', 'Strong Jaw', 'Sturdy',
        'Suction Cups', 'Super Luck', 'Swarm', 'Swift Swim', 'Synchroniz
e',
        'Tangled Feet', 'Tangling Hair', 'Technician', 'Telepathy',
        'Thick Fat', 'Tinted Lens', 'Tough Claws', 'Trace', 'Triage',
        'Unaware', 'Unburden', 'Unnerve', 'Vital Spilit', 'Water Absorb',
        'Water Veil', 'Weak Armor', 'White Smoke', 'empty'], dtype=objec
t),
 array(['Adaptability', 'Aftermath', 'Analytic', 'Anger Point',
        'Anticipation', 'Aroma Veil', 'Battle Armor', 'Big Pecks', 'Blaz
e',
        'Bulletproof', 'Chlorophyll', 'Chrolophyll', 'Clear Body',
        'Cloud Nine', 'Competitive', 'Compound Eyes', 'Contrary',
        'Cursed Body', 'Cute Charm', 'Damp', 'Defiant', 'Drizzle',
        'Drought', 'Dry Skin', 'Early Bird', 'Effect Spore',
        'Electric Surge', 'Flame Body', 'Flare Boost', 'Flash Fire',
        'Friend Guard', 'Frisk', 'Gale Wings', 'Galvanize', 'Gluttony',
        'Gooey', 'Grass Pelt', 'Grassy Surge', 'Guts', 'Harvest', 'Heale
r',
        'Heavy Metal', 'Honey Gather', 'Huge Power', 'Hustle', 'Hydratio
```

```
n',
        'Ice Body', 'Ice Scales', 'Immunity', 'Imposter', 'Infiltrator',
        'Inner Focus', 'Insomnia', 'Intimidate', 'Iron Fist', 'Justified',
        'Keen Eye', 'Klutz', 'Leaf Guard', 'Leaf guard', 'Libero',
        'Light Metal', 'Lightning Rod', 'Limber', 'Liquid Voice',
        'Long Reach', 'Magic Bounce', 'Magic guard', 'Magician',
        'Marvel Scale', 'Minus', 'Mirror Armor', 'Misty Surge',
        'Mold Breaker', 'Moody', 'Motor drive', 'Moxie', 'Multiscale',
        'Natural Cure', 'No Guard', 'Oblivious', 'Overcoat', 'Overgrow',
        'Own Tempo', 'Perish Body', 'Pickpocket', 'Pickup', 'Pixilate',
        'Plus', 'Poison Heal', 'Poison Touch', 'Power of Alchemy',
        'Prankster', 'Pressure', 'Propeller Tail', 'Protean',
        'Psychic Surge', 'Quick Feet', 'Rain Dish', 'Rain dish', 'Rattle
d',
        'Reckless', 'Regenerator', 'Rivalry', 'Rock Head', 'Rough Skin',
        'Run Away', 'Sand Force', 'Sand Rush', 'Sand Veil', 'Sap Sipper',
        'Scrappy', 'Serene Grace', 'Shadow Tag', 'Sheer Force',
        'Shell Armor', 'Simple', 'Skill Link', 'Slush Rush', 'Sniper',
        'Snow Cloak', 'Snow Warning', 'Solar Power', 'Soundproof',
        'Speed Boost', 'Stalwart', 'Static', 'Steadfast', 'Steely Spilit',
        'Stekeout', 'Stench', 'Storm Drain', 'Sturdy', 'Super Luck',
        'Swarm', 'Sweet Veil', 'Swift Swim', 'Symbiosis', 'Tangled Feet',
        'Technician', 'Telepathy', 'Thick Fat', 'Tinted Lens', 'Torrent',
        'Toxic Boost', 'Truant', 'Unaware', 'Unburden', 'Unnerve',
        'Vital Spilit', 'Volt Absorb', 'Water Absorb', 'Water Veil',
        'Weak Armor', 'White Smoke', 'Wonder Skin', 'Zen Mode', 'empty'],
       dtype=object)]
```

PCA

PCA (https://scikit-learn.org/stable/modules/decomposition.html#principal-component-analysis-pca) is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance.
In Scikit-learn, PCA is implemented as a transformer object that learns components in its fit method and can be used on new data to project it on these components.

Let's use Scikit-learn PCA (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html) to reduce the number of features in our dataset.
We will reduce to two features and show the samples on a graph.

In [ ]:

```
# convert data points to 2dim with pca
from sklearn.decomposition import PCA

pca = PCA(n_components=2).fit(pokemon_df_cp)
pca_pokemon_df = pd.DataFrame(pca.transform(pokemon_df_cp), columns=['pc1', 'pc2'])
pca_pokemon_df
```

Out[ ]:

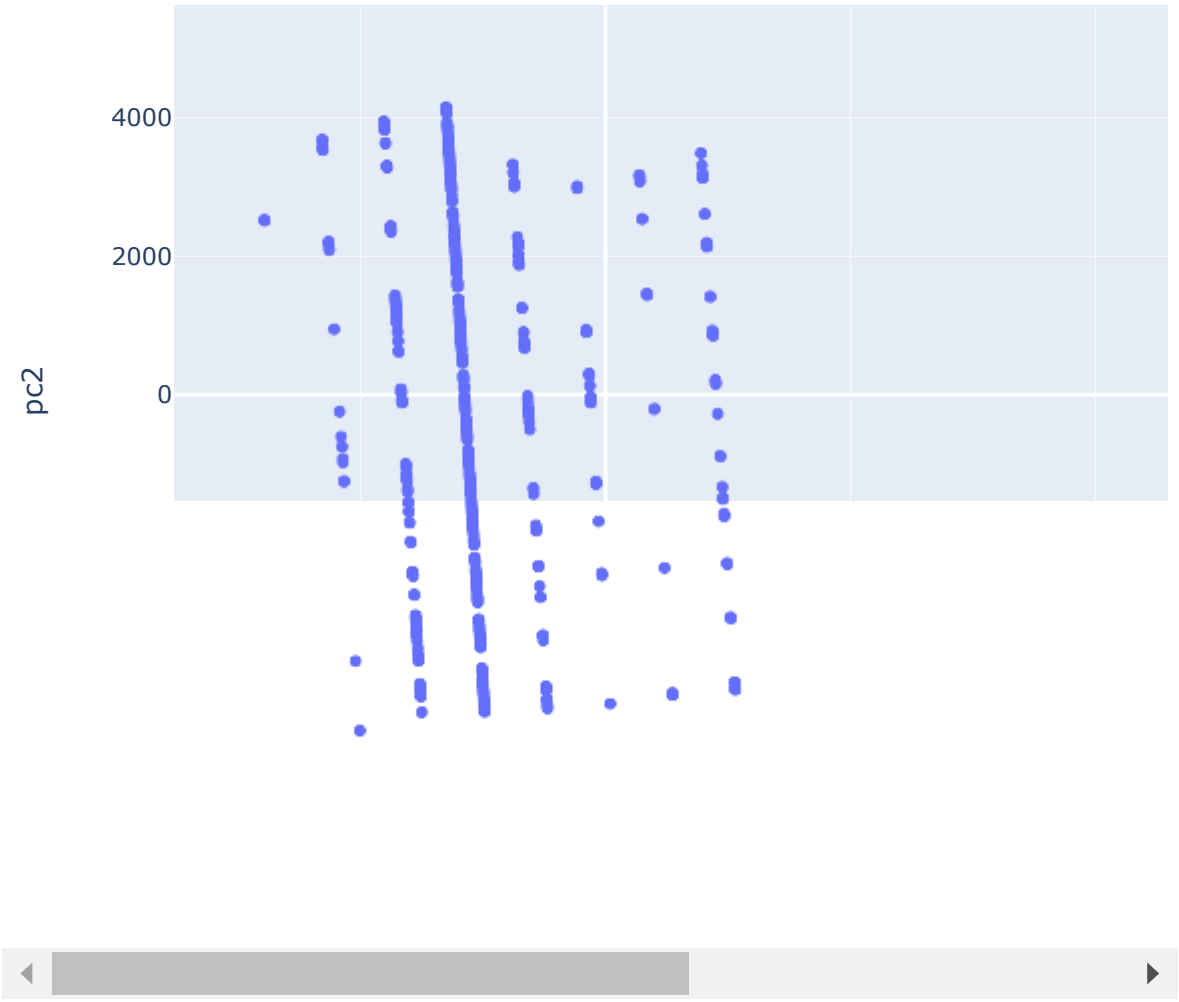
	pc1	pc2
0	-3252.467281	4147.886777
1	-3249.660580	4139.822527
2	-3245.247029	4133.207242
3	-3243.763135	4130.256263
4	-3249.501490	4117.844871
...	...	...
1047	23059.801772	-2511.606461
1048	23055.800987	-2518.285650
1049	23054.093658	-2532.469744
1050	23062.732488	-2533.739066
1051	23057.102914	-2534.498026

1052 rows × 2 columns

Let's see the samples on a graph.

In [ ]:

```
# show data points on a graph
fig = px.scatter(pca_pokemon_df, x='pc1', y='pc2')
fig.show()
```



Unsupervised Learning

We can use Unsupervised learning to research our dataset further.
We can use two algorithms that will help us cluster our dataset samples:

1. KMeans
2. Hierarchical Clustering

KMeans

The KMeans (https://scikit-learn.org/stable/modules/clustering.html#k-means) algorithm clusters data by trying to separate samples in `K` groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares.
This algorithm requires the number of clusters to be specified.
It scales well to a large number of samples and has been used across a large range of application areas in many different fields.

The KMeans algorithm divides a set of `N` samples `X` into `K` disjoint clusters `C`, each described by the mean `Uc` of the samples in the cluster.
The means are commonly called the cluster "centroids"; note that they are not, in general, points from `X`, although they live in the same space.

The K-means algorithm aims to choose centroids that minimize the inertia, or within-cluster sum-of-squares criterion:

$$\sum_{i=0}^{n} \min_{\mu_j \in C} (||x_i - \mu_j||^2)$$

Hierarchical Clustering

Hierarchical Clustering (https://scikit-learn.org/stable/modules/clustering.html#hierarchical-clustering) is a general family of clustering algorithms that build nested clusters by merging or splitting them successively.
This hierarchy of clusters is represented as a tree (or dendrogram).
The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample.
See the Wikipedia page (https://en.wikipedia.org/wiki/Hierarchical_clustering) for more details.

The AgglomerativeClustering (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html#sklearn.cluster.Agglomerativ object performs a hierarchical clustering using a bottom-up approach: each observation starts in its cluster, and clusters are successively merged.
The linkage criteria determine the metric used for the merge strategy:

- **Ward** minimizes the variance of the clusters being merged. In other words, it tries to create a new cluster that has the lowest WCV possible.
- **Maximum** or **Complete Linkage** linkage uses the maximum distances between all observations of the two sets.
- **Average Linkage** uses the average of the distances of each observation of the two sets.
- **Single Linkage** uses the minimum of the distances between all observations of the two sets.

Let's use Scikit-learn KMeans (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans) on our Pokemon dataset.

In [ ]:

```
# cluster with KMeans and show cluster centers
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=4, random_state=1).fit(pokemon_df_cp)
kmeans.cluster_centers_
```

Out[ ]:

```
array([[ 2.06313278e+02,  1.17634855e+00,  2.06430913e+03,
         4.86526971e+02,  9.83402490e+00,  1.35892116e+01,
         4.05601660e+00,  1.19929461e+02,  8.99813278e+01,
         9.37116183e+01,  2.50622407e+00,  7.21644966e-16,
         7.67634855e-02,  1.17572614e+00,  5.40375519e+01,
         6.55082988e+01,  7.44564315e+01,  7.01431535e+01,
         6.68443983e+01,  6.77074689e+01,  6.47842324e+01,
         4.09443983e+02,  1.05715768e+02,  5.29792531e+03,
         1.40844398e+02,  2.38589212e-01,  4.29460581e-01,
         2.61410788e-01,  2.94605809e-01,  2.05394191e-01,
         3.31950207e-01,  1.76141079e+00],
       [ 6.41886598e+02,  1.40206186e+00,  6.42026804e+03,
         5.32917526e+02,  8.89690722e+00,  1.19793814e+01,
         4.60824742e+00,  1.01876289e+02,  1.24742268e+02,
         1.51721649e+02,  5.44329897e+00,  1.00000000e+00,
         7.21649485e-02,  2.72577320e+00,  2.46844330e+02,
         9.70412371e+01,  1.16309278e+02,  9.57216495e+01,
         1.13525773e+02,  9.58350515e+01,  9.93711340e+01,
         6.17804124e+02,  2.59278351e+01,  3.07200000e+04,
         3.04824742e+02,  5.97938144e-01,  7.11340206e-01,
         1.03092784e-01,  8.76288660e-01,  2.16494845e-01,
         3.91752577e-01,  2.89690722e+00],
       [ 6.51624724e+02,  1.16114790e+00,  6.51740839e+03,
         4.52298013e+02,  8.99779249e+00,  1.26688742e+01,
         4.15894040e+00,  1.06291391e+02,  8.97373068e+01,
         9.18852097e+01,  5.92052980e+00,  2.20750552e-02,
         1.32450331e-02,  1.01964680e+00,  4.94679912e+01,
         6.79955850e+01,  7.82869757e+01,  7.31103753e+01,
         6.92185430e+01,  6.99757174e+01,  6.51589404e+01,
         4.23746137e+02,  9.98498896e+01,  5.42233996e+03,
         1.46163355e+02,  1.39072848e-01,  5.49668874e-01,
         3.00220751e-01,  3.26710817e-01,  1.87637969e-01,
         3.09050773e-01,  1.81236203e+00],
       [ 3.69500000e+02,  1.20000000e+00,  3.69620000e+03,
         5.43300000e+02,  1.12000000e+01,  1.14500000e+01,
         4.90000000e+00,  8.04500000e+01,  1.26000000e+02,
         1.19600000e+02,  3.95000000e+00,  1.00000000e+00,
        -1.38777878e-17,  1.58500000e+00,  1.16200000e+02,
         8.85000000e+01,  9.14000000e+01,  1.02050000e+02,
         9.95000000e+01,  1.06550000e+02,  9.20000000e+01,
         5.80000000e+02,  3.00000000e+00,  2.04800000e+04,
         2.90000000e+02,  5.00000000e-02,  5.50000000e-01,
         5.50000000e-01,  9.00000000e-01,  8.50000000e-01,
         1.00000000e-01,  3.00000000e+00]])
```

Let's show all points and cluster centers on a graph with different colors for different clusters.
We need to round the centroids' (cluster centers) values to make them look like real data points (real Pokemons).

In [ ]:

```
# roud a number to the closest integer (or min/max if it is not in range)
def round_number_to_closest_int(number, min, max):
    if number < min:
        return min
    elif number > max:
        return max
    else:
        return round(number)
```

In [ ]:

```
# round number in columns to closest integers
def round_columns_to_closest_int(df, origin_df, columns):
    for col in columns:
        min_in_col = origin_df[col].min()
        max_in_col = origin_df[col].max()
        df[col] = df[col].apply(lambda s: round_number_to_closest_int(s, min_in_col, ma
x_in_col))
    return df
```

In [ ]:

```
# show the means of the clusters as if they were real data points
mean_clusters = pd.DataFrame(kmeans.cluster_centers_, columns=pokemon_df_cp.columns)
mean_clusters = round_columns_to_closest_int(mean_clusters, pokemon_df_cp, ordinal_cols
)
mean_clusters = round_columns_to_closest_int(mean_clusters, pokemon_df_cp, categorical_
cols)

mean_clusters_cp = mean_clusters.copy()
mean_clusters_cp[categorical_cols] = oe.inverse_transform(mean_clusters_cp[categorical_
cols])
mean_clusters_cp.replace('empty', np.NaN, inplace=True)
mean_clusters_cp
```

Out[ ]:

	NUMBER	CODE	SERIAL	NAME	TYPE1	TYPE2	COLOR	ABILITY1	ABILITY2	ABIL HIDI
0	206	1	2064	Mega Beedrill	Ground	Psychic	Grey	Pickup	Sniper	Prop
1	642	1	6420	Mesprit	Grass	Normal	Pink	Mega Launcher	White Smoke	W
2	652	1	6517	Magikarp	Grass	Poison	Grey	Misty Surge	Sniper	Prank
3	369	1	3696	Milotic	Ice	Ice	Pink	Innards Out	NaN	S C

We can also show the clusters centers on a graph of two dimensions with the help of PCA for dimensionality reduction.

In [ ]:

```
# convert cluster centers to 2dim with pca
pca_mean_clusters = pd.DataFrame(pca.transform(mean_clusters), columns=['pc1', 'pc2'])
pca_mean_clusters
```

Out[ ]:

	pc1	pc2
0	-2887.673190	2111.671278
1	22827.156654	33.520135
2	-2362.415398	-2334.551516
3	12382.402367	1841.165139

In [ ]:

```
# add cluster column for each df
pca_mean_clusters_cp = pca_mean_clusters.copy()
pca_mean_clusters_cp['cluster'] = kmeans.predict(mean_clusters)
display(pca_mean_clusters_cp)

pca_pokemon_df_cp = pca_pokemon_df.copy()
pca_pokemon_df_cp['cluster'] = kmeans.predict(pokemon_df_cp)
display(pca_pokemon_df_cp)
```

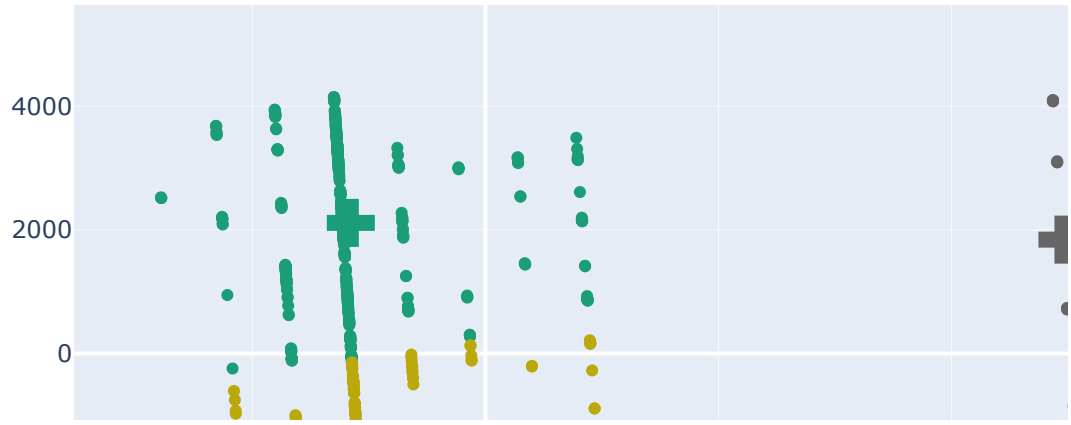
	pc1	pc2	cluster
0	-2887.673190	2111.671278	0
1	22827.156654	33.520135	1
2	-2362.415398	-2334.551516	2
3	12382.402367	1841.165139	3

	pc1	pc2	cluster
0	-3252.467281	4147.886777	0
1	-3249.660580	4139.822527	0
2	-3245.247029	4133.207242	0
3	-3243.763135	4130.256263	0
4	-3249.501490	4117.844871	0
...	...	...	...
1047	23059.801772	-2511.606461	1
1048	23055.800987	-2518.285650	1
1049	23054.093658	-2532.469744	1
1050	23062.732488	-2533.739066	1
1051	23057.102914	-2534.498026	1

1052 rows × 3 columns

In [ ]:

```
# show data points and cluster centers together (each color means a cluster)
fig = go.Figure()
fig.add_trace(go.Scatter(x=pca_mean_clusters_cp.pc1, y=pca_mean_clusters_cp.pc2, mode=
'markers', marker=dict(size=20, symbol='cross', color=pca_mean_clusters_cp.cluster, col
orscale=px.colors.qualitative.Dark2)))
fig.add_trace(go.Scatter(x=pca_pokemon_df_cp.pc1, y=pca_pokemon_df_cp.pc2, mode='marker
s' ,marker=dict(color=pca_pokemon_df_cp.cluster, colorscale=px.colors.qualitative.Dark2
)))
fig.show()
```



We can connect the two graphs and show a graph with the cluster centers and the data points together.

In [ ]:

```
# show data points and cluster centers together (each color means a cluster)
fig = go.Figure()
fig.add_trace(go.Scatter(x=pca_mean_clusters_cp.pc1, y=pca_mean_clusters_cp.pc2, mode=
'markers', marker=dict(size=20, symbol='cross', color=pca_mean_clusters_cp.cluster, col
orscale=px.colors.qualitative.Dark2)))
fig.add_trace(go.Scatter(x=pca_pokemon_df_cp.pc1, y=pca_pokemon_df_cp.pc2, mode='marker
s' ,marker=dict(color=pca_pokemon_df_cp.cluster, colorscale=px.colors.qualitative.Dark2
)))
fig.show()
```

We can use Scikit-learn AgglomerativeClustering (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html#sklearn.cluster.Agglomerativ as a Hierarchical Clustering.
We can plot the dendrogram with Scipy dendrogram (https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.dendrogram.html).

Simple Example

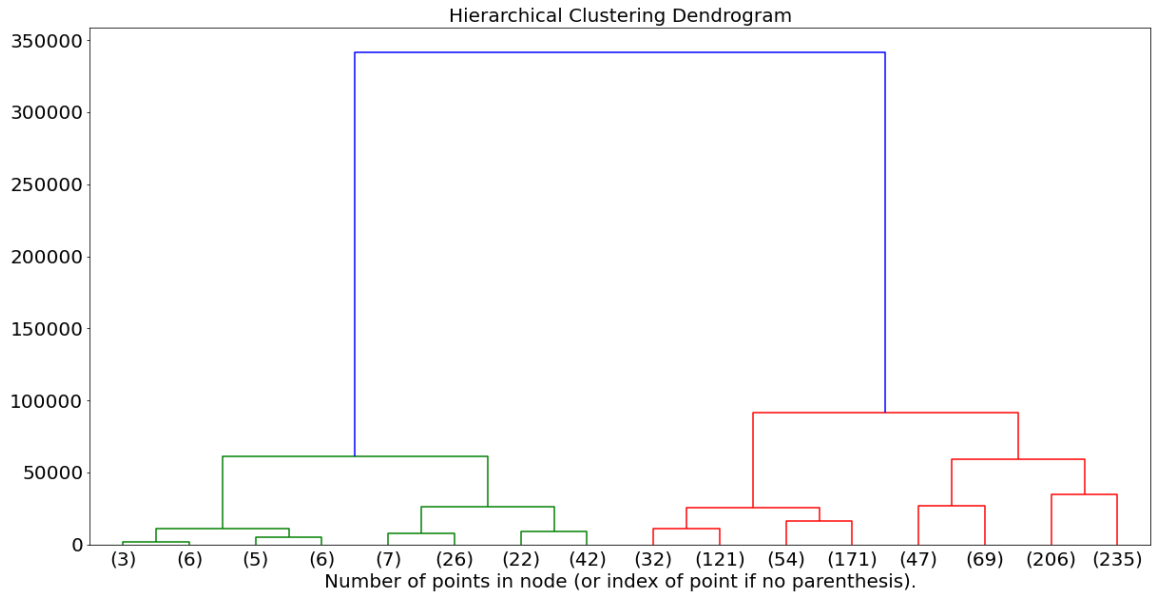
In [ ]:

```
from sklearn.cluster import AgglomerativeClustering
import numpy as np
X = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])
# Setting `n_clusters=None` in order to get the true number of clusters
clustering = AgglomerativeClustering(distance_threshold=0, n_clusters=None).fit(X)

print(clustering.labels_)
print('\n', clustering.children_)
```

```
[5 4 3 1 2 0]

 [[0 1]
 [3 5]
 [2 6]
 [4 7]
 [8 9]]
```

In [ ]:

```
# calculate Hierarchical Clustering and plot dendogram
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram

def plot_dendrogram(model, **kwargs):
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1  # leaf node
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count

    linkage_matrix = np.column_stack([model.children_, model.distances_, counts]).astyp
e(float)

    dendrogram(linkage_matrix, **kwargs)

agg = AgglomerativeClustering(distance_threshold=0, n_clusters=None)
agg = agg.fit(pokemon_df_cp)

plt.figure(figsize=(20, 10))
plot_dendrogram(agg, truncate_mode='level', p=3)

plt.title('Hierarchical Clustering Dendrogram', fontsize=20)
plt.xlabel("Number of points in node (or index of point if no parenthesis).", fontsize=
20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.show()
```



NBC (Naive Bayes)

Naive Bayes (https://scikit-learn.org/stable/modules/naive_bayes.html#naive-bayes) methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable.
Bayes' theorem states the following relationship, given class variable `y` and dependent feature vector `x1` through `xn` :

$$P(y \mid x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots, x_n \mid y)}{P(x_1, \dots, x_n)}$$

Using the naive conditional independence assumption that:

$$P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i|y),$$

for all `i` , this relationship is simplified to:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

Since `P(x1,...,xn)` is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

We can use Maximum A Posteriori (MAP) estimation (https://en.wikipedia.org/wiki/Maximum_a_posteriori_estimation) to estimate `P(y)` and `P(xi|y)` ; the former is then the relative frequency of class in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of `P(xi|y)` .

In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering.
They require a small amount of training data to estimate the necessary parameters (for theoretical reasons why naive Bayes works well, and on which types of data it does, see The Optimality of Naive Bayes (https://www.cs.unb.ca/~hzhang/publications/FLAIRS04ZhangH.pdf)).

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods.
The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one-dimensional distribution.
This in turn helps to alleviate problems stemming from the curse of dimensionality.

On the flip side, although naive Bayes is known as a decent classifier, it is known to be a bad estimator, so the probability outputs from `predict_proba` are not to be taken too seriously.

Let's use Scikit-learn GaussianNB (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn-naive-bayes-gaussiannb) to classify the Pokemons into types.
We will use **LEGENDARY** as a label.

In [ ]:

```
# divide the data to features and target
target_column = 'LEGENDARY'
t = pokemon_df_cp[target_column].copy()
X = pokemon_df_cp.drop([target_column], axis=1)
print('t')
display(t)
print()
print('X')
display(X)
```

t

```
0       0
1       0
2       0
3       0
4       0
       ..
1047    1
1048    1
1049    1
1050    1
1051    1
Name: LEGENDARY, Length: 1052, dtype: int64
```

X

	NUMBER	CODE	SERIAL	NAME	TYPE1	TYPE2	COLOR	ABILITY1	ABILITY2	ABILI HIDD
0	1	1	11	82.0	9.0	13.0	3.0	117.0	126.0	1
1	2	1	21	372.0	9.0	13.0	3.0	117.0	126.0	1
2	3	1	31	887.0	9.0	13.0	3.0	117.0	126.0	1
3	3	2	32	525.0	9.0	13.0	3.0	193.0	126.0	15
4	4	1	41	107.0	6.0	18.0	7.0	15.0	126.0	12
...	...	...	...	...	...	...	...	...	...	
1047	896	1	8961	292.0	11.0	18.0	8.0	18.0	126.0	15
1048	897	1	8971	783.0	8.0	18.0	0.0	65.0	126.0	15
1049	898	1	8981	90.0	14.0	9.0	3.0	201.0	126.0	15
1050	898	2	8982	90.0	14.0	11.0	8.0	5.0	126.0	15
1051	898	3	8983	90.0	14.0	8.0	0.0	5.0	126.0	15

1052 rows × 31 columns

Let's see how many different types we have in the target.

In [ ]:

```
# show unique values of target
t.unique()
```

Out[ ]:

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

Let's use Feature Selection and get the best features for our target.
We are using SGDClassifier for that (and not NBC), because we need an estimator that has a `coef_` or `feature_importances_` attributes.

In [ ]:

```
# use feature selection on the data
selector = RFECV(SGDClassifier(random_state=1), cv=RepeatedKFold(n_splits=5, n_repeats=
5, random_state=1)).fit(X, t)
display(X.loc[:, selector.support_])
```

	NAME	TYPE1	TYPE2	ABILITY1	ABILITY2	ABILITY HIDDEN	WEIGHT	HP	ATK	DEF	SP_A
0	82.0	9.0	13.0	117.0	126.0	11.0	6.9	45	49	49	
1	372.0	9.0	13.0	117.0	126.0	11.0	13.0	60	62	63	
2	887.0	9.0	13.0	117.0	126.0	11.0	100.0	80	82	83	1
3	525.0	9.0	13.0	193.0	126.0	157.0	155.5	80	100	123	1
4	107.0	6.0	18.0	15.0	126.0	122.0	8.5	39	52	43	
...	...	...	...	...	...	...	...	...	...	...	
1047	292.0	11.0	18.0	18.0	126.0	157.0	800.0	100	145	130	
1048	783.0	8.0	18.0	65.0	126.0	157.0	44.5	100	65	60	1
1049	90.0	14.0	9.0	201.0	126.0	157.0	7.7	100	80	80	
1050	90.0	14.0	11.0	5.0	126.0	157.0	809.1	100	165	150	
1051	90.0	14.0	8.0	5.0	126.0	157.0	53.6	100	85	80	1

1052 rows × 17 columns

Now, we can use Scikit-learn [MultinomialNB (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html#sklearn.naive_bayes.Multinomia](https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html#sklearn.naive_bayes.Multinomia) for classifying the data.

In [ ]:

```
# show score of nbc on the data
from sklearn.naive_bayes import MultinomialNB

print('MultinomialNB score:', cross_val_score(MultinomialNB(), X.loc[:, selector.suppor
t_], t, cv=15).mean())
```

```
MultinomialNB score: 0.9523809523809524
```

We can use PCA to convert the features to a smaller set of features.
We can use GridSearch to find the best number of features to be used as the target features for the PCA algorithm.
The PCA may return negative numbers, so, we need to use Scikit-learn GaussianNB (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB) as an NBC.
This is a different form of NBC that uses a different form of `P(xi|y)` estimation:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

This estimation is very similar to LDA (Linear Discriminant Analysis), and it can take non-positive values.

In [ ]:

```
# use grid search to find best pca component on the data for nbc
from sklearn.naive_bayes import GaussianNB

model_pipe = Pipeline([('pca', 'passthrough'),
                       ('nbc', GaussianNB())])

hyper_parameters = {'pca': [PCA()],
                    'pca__n_components': list(range(1, len(X.columns)-3))}

gs_model = GridSearchCV(model_pipe, hyper_parameters).fit(X, t)
print('Accuracy score for classification:')
print('gs_model', gs_model.best_score_)
print('best params', gs_model.best_params_)
```

```
Accuracy score for classification:
gs_model 0.9838230647709321
best params {'pca': PCA(copy=True, iterated_power='auto', n_components=1,
random_state=None,
    svd_solver='auto', tol=0.0, whiten=False), 'pca__n_components': 1}
```

The PCA algorithm with only one component (most important altered feature) scored better than the Feature Selection algorithm with 17 features.

LDA (Linear Discriminant Analysis) and QDA (Quadratic Discriminant Analysis)

When using NBC, we need to work with ordinal features.
In NBC we calculate the posterior `P(xi|y)` for each feature value with counting (number of samples with this feature value divided by the number of samples of the category).
In Linear and Quadratic Discriminant Analysis (https://scikit-learn.org/stable/modules/lda_qda.html#linear-and-quadratic-discriminant-analysis) we can use numerical features too.
We calculate the posterior `P(xi|y)` for numerical feature values with density function.

Both LDA and QDA can be derived from simple probabilistic models that model the class conditional distribution of the data `P(X|y=k)` for each class `k`.
Predictions can then be obtained by using Bayes' rule, for each training sample `x`.

$$P(y=k|x) = \frac{P(x|y=k)P(y=k)}{P(x)} = \frac{P(x|y=k)P(y=k)}{\sum_l P(x|y=l)\cdot P(y=l)}$$

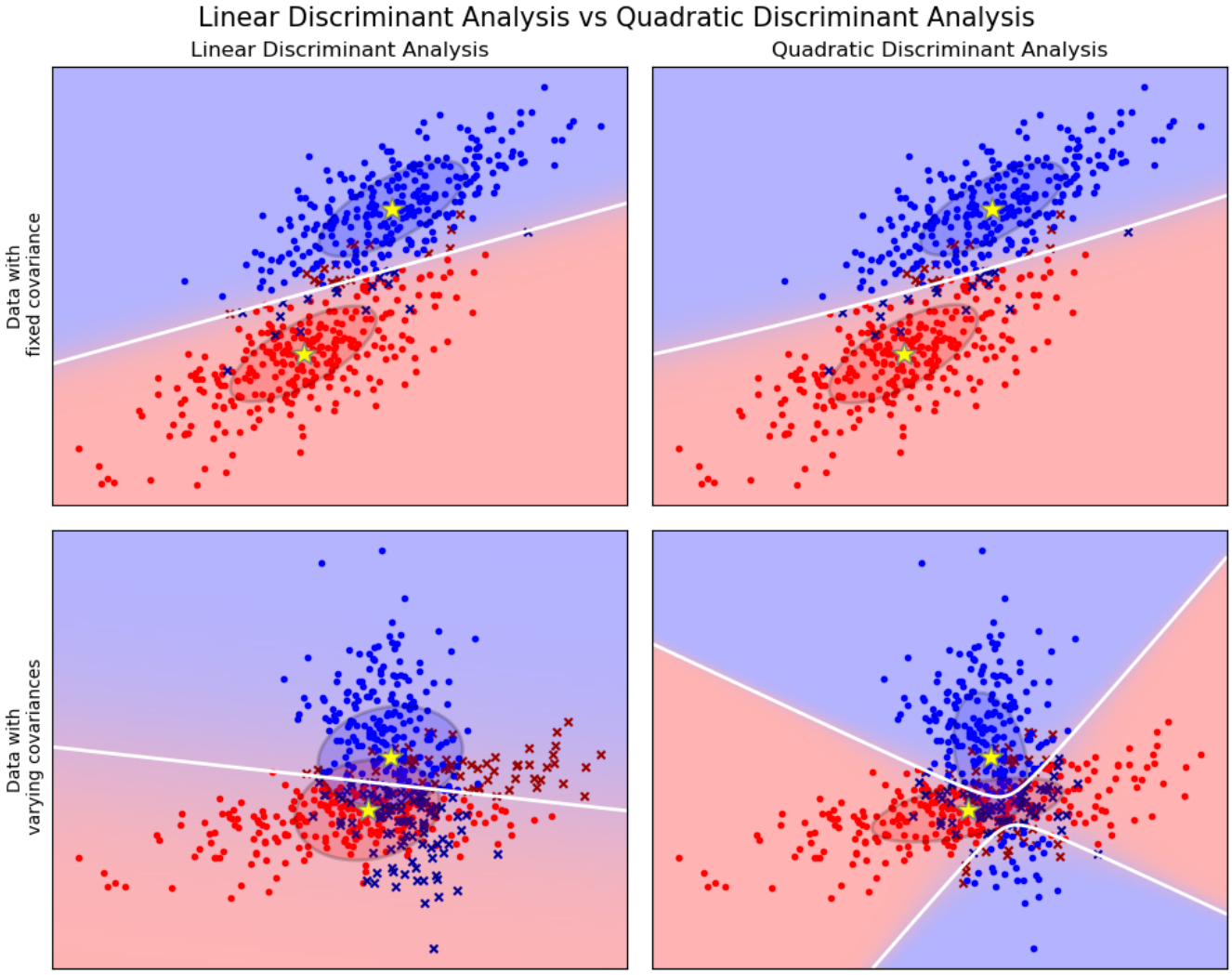
We select the class 'k' which maximizes this posterior probability.
More specifically, for linear and quadratic discriminant analysis `P(x|y)` is modeled as a multivariate Gaussian distribution with density:

$$P(x|y=k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^t \Sigma_k^{-1}(x-\mu_k)\right)$$

Where `d` is the number of features.

LDA is a special case of QDA, where the Gaussians for each class are assumed to share the same covariance matrix.

LDA can only learn linear boundaries, while QDA can learn quadratic boundaries and is, therefore, more flexible.



Let's use Scikit-learn LinearDiscriminantAnalysis (https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html#sklearn.dis on our data.

In [ ]:

```
# show score of lda on the data
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

print('LinearDiscriminantAnalysis score:', cross_val_score(LinearDiscriminantAnalysis
(), X.loc[:, selector.support_], t, cv=15).mean())
```

```
LinearDiscriminantAnalysis score: 0.9895238095238096
```

Here, we can also try to use PCA instead of Feature Selection.

In [ ]:

```
# use grid search to find best pca component on the data for lda
model_pipe = Pipeline([('pca', 'passthrough'),
                       ('nbc', LinearDiscriminantAnalysis())])

hyper_parameters = {'pca': [PCA()],
                    'pca__n_components': list(range(1, len(X.columns)-3))}

gs_model = GridSearchCV(model_pipe, hyper_parameters).fit(X, t)
print('Accuracy score for classification:')
print('gs_model', gs_model.best_score_)
print('best params', gs_model.best_params_)
```

```
Accuracy score for classification:
gs_model 0.9904897314375987
best params {'pca': PCA(copy=True, iterated_power='auto', n_components=1,
random_state=None,
    svd_solver='auto', tol=0.0, whiten=False), 'pca__n_components': 1}
```

The PCA algorithm with only one component (most important altered feature) scored better than the Feature Selection algorithm with 17 features on the LDA too.
The LDA model also scored better than the NBC model on this data.

More Information

How to use D-Tale EDA tool with Google Colab:
D-Tale (pandas data frame visualizer) now available in the cloud with Google Colab! (https://www.reddit.com/r/datascience/comments/f8uphl/dtale_pandas_dataframe_visualizer_now_available/)

A Description of D-Tale in PyPi:
D-Tale Project description (https://pypi.org/project/dtale/)

Example of how to model and plot with KMeans and PCA:
A demo of K-Means clustering on the handwritten digits data (https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_digits.html#a-demo-of-k-means-clustering-on-the-handwritten-digits-data)

Example of how to model and plot with Hierarchical Clustering:
Plot Hierarchical Clustering Dendrogram (https://scikit-learn.org/stable/auto_examples/cluster/plot_agglomerative_dendrogram.html#plot-hierarchical-clustering-dendrogram)

Method for stacking columns as a matrix:
numpy.column_stack (https://numpy.org/doc/stable/reference/generated/numpy.column_stack.html)

A special form of NBC that is similar to QDA:
sklearn.naive_bayes.GaussianNB (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html)

The differences between LDA and NBC:
Linear Discriminant Analysis vs Naive Bayes (https://stackoverflow.com/a/46607015)

Wikipedia on NBC classifier:
Naive Bayes classifier (https://en.wikipedia.org/wiki/Naive_Bayes_classifier)

Wikipedia on LDA:
Linear discriminant analysis (https://en.wikipedia.org/wiki/Linear_discriminant_analysis)

Wikipedia on QDA:
Quadratic discriminant analysis (https://en.wikipedia.org/wiki/Quadratic_classifier#Quadratic_discriminant_analysis)

Example of using Pipeline and GridSearch with PCA:
Selecting dimensionality reduction with Pipeline and GridSearchCV (https://scikit-learn.org/stable/auto_examples/compose/plot_compare_reduction.html)