Pokemon

September 20, 2022

0.1 Introduction

Pokémon is a series of video games and animated TV shows that first aired in 1996, created by Nintendo and Game Freak. The show and games have been highly successful and have received many Game Of The Year awards. The first game of the year award came in 2000, with the release of a their first ever game named 'Pokémon Yellow'. The Pokémon games have since been highly rated amongst the gaming community.

The world of Pokémon is a fantasy world based on monsters that people, known as ***Pokémon Trainers***, catch with special devices called ***Pokeballs***. Till this date they have been eight generation of Pokémon games and a total of 905 unique Pokémon's across all generations. The first generation introduced 150 Pokémon's with 4 special Pokémon's called **Legendary Pokémon**. Now we have a total of 82 Legendary Pokémon.

0.2 What makes a Pokemon Legendary?

A Legendary Pokémon is a special type of Pokémon that is very rare and extremely powerful. In the world of Pokémon, they are considered as myths or legends. The stats of a Legendary Pokémon differ vastly from a normal Pokémon, as they have Higher Attack, Defense, Speed, Special Attack and Special Defense then normal Pokémon.

So, in this pipeline we will looking at all these stats and using them to predict if a Pokémon is Legendary or Not.

The Legendary Dataset This dataset contains information on all 802 Pokémon from all Seven Generations of Pokémon. The information contained in this dataset include Base Stats, Performance against Other Types, Height, Weight, Classification, Egg Steps, Experience Points, Abilities, etc. The information was scraped from http://serebii.net/

Contents of the Dataset

1 - Name: The English name of the Pokemon

2 - Japanese Name: The Original Japanese name of the Pokemon

3 - Pokedex Number: The entry number of the Pokemon in the National Pokedex

4 - Percentage male: The percentage of the species that are male. Blank if the Pokemon is genderless.

5 - Type1: The Primary Type of the Pokemon

6 - Type2: The Secondary Type of the Pokemon

7 - Classification: The Classification of the Pokemon as described by the Sun and Moon Pokedex

8 - Height (m): Height of the Pokemon in metres

9 - Weight (kg): The Weight of the Pokemon in kilograms

10 - Capture Rate: Capture Rate of the Pokemon

11 - Base Egg Steps: The number of steps required to hatch an egg of the Pokemon

12 - Abilities: A stringified list of abilities that the Pokemon is capable of having

13 - Experience Growth: The Experience Growth of the Pokemon

14 - Base Happiness: Base Happiness of the Pokemon

15 - Against: Eighteen features that denote the amount of damage taken against an attack of a particular type

16 - HP: The Base HP (Health) of the Pokemon

17 - Attack: The Base Attack of the Pokemon

18 - Defense: The Base Defense of the Pokemon

19 - SP Attack: The Base Special Attack of the Pokemon

20 - SP Defense: The Base Special Defense of the Pokemon

21 - Speed: The Base Speed of the Pokemon

22 - Generation: The numbered generation which the Pokemon was first introduced

21 - Is Legendary: Denotes if the Pokemon is legendary.

0.2.1 Overview of the Pipeline

In this pipeline we will be going through many different Machine Learning stages, starting from Data Collection, then we will move on to the Data Exploration stage. After Data Exploration we will seeing the Data Preprocessing stage, where we will clean our datasets of any missing values, outliers or any other value that might alter the efficiency of our Machine Learning Model. Then we will move onto the Feature Engineering and then Training and Testing the model, for this pipeline we are using three Machine Learning Models, namely, ***Support Vector Machine, Decision Tree Classifier and K Nearest Neighbors***. Then we will conclude this pipeline by analyzing the results by these three models and pick out the best model based on metric score.

But the most crucial stage for any pipeline is the importing libraries stage, as without them there is no code. So, for this pipeline we will be working with, ***Pandas, Scikit-Learn, Matplotlib and Seaborn*** libraries. So, let's start by,

0.3 Importing libraries

[1]:
```
import pandas                   as pd

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing   import OneHotEncoder, StandardScaler
from sklearn.decomposition   import PCA
from imblearn.over_sampling  import SMOTE
from sklearn.tree            import DecisionTreeClassifier, plot_tree
from sklearn.neighbors       import KNeighborsClassifier
from sklearn.svm             import SVC
from sklearn.metrics         import accuracy_score, f1_score, recall_score,␣
 ↪precision_score
from sklearn.metrics         import confusion_matrix, ConfusionMatrixDisplay

import seaborn                  as sns
import matplotlib.pyplot        as plt
```

Now that libraries have been imported, now we can use Panda's library to import the Dataset we will be working on,

0.4 Loading Dataset

[2]:
```
df = pd.read_csv('pokemon.csv')
dfpop = df.pop('name')
df.insert(0,'name', value = dfpop)
df = df.set_index('pokedex_number')
df.head()
```

[2]:
```
                     name                     abilities  against_bug  \
pokedex_number
1               Bulbasaur  ['Overgrow', 'Chlorophyll']          1.0
2                 Ivysaur  ['Overgrow', 'Chlorophyll']          1.0
3                Venusaur  ['Overgrow', 'Chlorophyll']          1.0
4              Charmander     ['Blaze', 'Solar Power']          0.5
5              Charmeleon     ['Blaze', 'Solar Power']          0.5

                against_dark  against_dragon  against_electric  against_fairy  \
pokedex_number
1                        1.0             1.0               0.5            0.5
2                        1.0             1.0               0.5            0.5
3                        1.0             1.0               0.5            0.5
4                        1.0             1.0               1.0            0.5
5                        1.0             1.0               1.0            0.5

                against_fight  against_fire  against_flying  …  \
pokedex_number                                               …
1                         0.5           2.0             2.0  …
2                         0.5           2.0             2.0  …
3                         0.5           2.0             2.0  …
4                         1.0           0.5             1.0  …
5                         1.0           0.5             1.0  …

                   japanese_name  percentage_male  sp_attack  sp_defense  \
pokedex_number
1               Fushigidane                  88.1         65          65
2                Fushigisou                  88.1         80          80
3               Fushigibana                  88.1        122         120
4                   Hitokage                 88.1         60          50
5                    Lizardo                 88.1         80          65

                speed  type1   type2  weight_kg  generation  is_legendary
pokedex_number
1                  45  grass  poison        6.9           1             0
2                  60  grass  poison       13.0           1             0
3                  80  grass  poison      100.0           1             0
4                  65   fire     NaN        8.5           1             0
5                  80   fire     NaN       19.0           1             0
```

```
[5 rows x 40 columns]
```

1 Data Exploration

Now that we have uploaded the Dataset, let's have some fun with it. Let's use it to find some interesting insights.

[3]:
```
sns.set_style('darkgrid');
sns.set_context(context='paper', font_scale=1.2);
```

Listing the names of all Legendary Pokemon and their stats:

[4]:
```
pd.options.display.max_rows = 70
df[(df['is_legendary'] == 1)]
```

[4]:
```
                          name                          abilities  \
pokedex_number
144                   Articuno         ['Pressure', 'Snow Cloak']
145                     Zapdos             ['Pressure', 'Static']
146                    Moltres         ['Pressure', 'Flame Body']
150                     Mewtwo            ['Pressure', 'Unnerve']
151                        Mew                    ['Synchronize']
243                     Raikou        ['Pressure', 'Inner Focus']
244                      Entei        ['Pressure', 'Inner Focus']
245                    Suicune        ['Pressure', 'Inner Focus']
249                      Lugia         ['Pressure', 'Multiscale']
250                      Ho-Oh        ['Pressure', 'Regenerator']
251                     Celebi                   ['Natural Cure']
377                   Regirock          ['Clear Body', 'Sturdy']
378                     Regice        ['Clear Body', 'Ice Body']
379                  Registeel     ['Clear Body', 'Light Metal']
380                     Latias                       ['Levitate']
381                     Latios                       ['Levitate']
382                     Kyogre                        ['Drizzle']
383                    Groudon                        ['Drought']
384                   Rayquaza                       ['Air Lock']
385                    Jirachi                   ['Serene Grace']
386                     Deoxys                       ['Pressure']
480                       Uxie                       ['Levitate']
481                    Mesprit                       ['Levitate']
482                      Azelf                       ['Levitate']
483                     Dialga          ['Pressure', 'Telepathy']
484                     Palkia          ['Pressure', 'Telepathy']
485                    Heatran       ['Flash Fire', 'Flame Body']
```

```
486              Regigigas                                  ['Slow Start']
487               Giratina          ['Pressure', 'Telepathy', 'Levitate']
488              Cresselia                                    ['Levitate']
490                Manaphy                                   ['Hydration']
491                Darkrai                                  ['Bad Dreams']
492                Shaymin               ['Natural Cure', 'Serene Grace']
493                 Arceus                                   ['Multitype']
494                Victini                                ['Victory Star']
638               Cobalion                                   ['Justified']
639              Terrakion                                   ['Justified']
640               Virizion                                   ['Justified']
641               Tornadus        ['Prankster', 'Defiant', 'Regenerator']
642              Thundurus        ['Prankster', 'Defiant', 'Volt Absorb']
643               Reshiram                                  ['Turboblaze']
644                 Zekrom                                    ['Teravolt']
645               Landorus    ['Sand Force', 'Sheer Force', 'Intimidate']
646                 Kyurem         ['Pressure', 'Teravolt', 'Turboblaze']
647                 Keldeo                                   ['Justified']
648               Meloetta                                ['Serene Grace']
649               Genesect                                    ['Download']
716                Xerneas                                  ['Fairy Aura']
717                Yveltal                                   ['Dark Aura']
718                Zygarde              ['Aura Break', 'Power Construct']
719                Diancie                                  ['Clear Body']
720                  Hoopa                                    ['Magician']
721              Volcanion                                ['Water Absorb']
785              Tapu Koko                ['Electric Surge', 'Telepathy']
786              Tapu Lele                 ['Psychic Surge', 'Telepathy']
787              Tapu Bulu                  ['Grassy Surge', 'Telepathy']
788              Tapu Fini                   ['Misty Surge', 'Telepathy']
789                 Cosmog                                     ['Unaware']
790                Cosmoem                                      ['Sturdy']
791               Solgaleo                             ['Full Metal Body']
792                 Lunala                               ['Shadow Shield']
793               Nihilego                                 ['Beast Boost']
794               Buzzwole                                 ['Beast Boost']
795              Pheromosa                                 ['Beast Boost']
796              Xurkitree                                 ['Beast Boost']
797             Celesteela                                 ['Beast Boost']
798                Kartana                                 ['Beast Boost']
799               Guzzlord                                 ['Beast Boost']
800               Necrozma                                 ['Prism Armor']
801               Magearna                                  ['Soul-Heart']

                against_bug  against_dark  against_dragon  against_electric  \
pokedex_number
144                    0.50           1.0             1.0              2.00
```

145	0.50	1.0	1.0	1.00
146	0.25	1.0	1.0	2.00
150	2.00	2.0	1.0	1.00
151	2.00	2.0	1.0	1.00
243	1.00	1.0	1.0	0.50
244	0.50	1.0	1.0	1.00
245	1.00	1.0	1.0	2.00
249	1.00	2.0	1.0	2.00
250	0.25	1.0	1.0	2.00
251	4.00	2.0	1.0	0.50
377	1.00	1.0	1.0	1.00
378	1.00	1.0	1.0	1.00
379	0.50	1.0	0.5	1.00
380	2.00	2.0	2.0	0.50
381	2.00	2.0	2.0	0.50
382	1.00	1.0	1.0	2.00
383	1.00	1.0	1.0	0.00
384	0.50	1.0	2.0	1.00
385	1.00	2.0	0.5	1.00
386	2.00	2.0	1.0	1.00
480	2.00	2.0	1.0	1.00
481	2.00	2.0	1.0	1.00
482	2.00	2.0	1.0	1.00
483	0.50	1.0	1.0	0.50
484	1.00	1.0	2.0	1.00
485	0.25	1.0	0.5	1.00
486	1.00	1.0	1.0	1.00
487	0.50	2.0	2.0	0.50
488	2.00	2.0	1.0	1.00
490	1.00	1.0	1.0	2.00
491	2.00	0.5	1.0	1.00
492	2.00	1.0	1.0	0.50
493	1.00	1.0	1.0	1.00
494	1.00	2.0	1.0	1.00
638	0.25	0.5	0.5	1.00
639	0.50	0.5	1.0	1.00
640	1.00	0.5	1.0	0.50
641	0.50	1.0	1.0	2.00
642	0.50	1.0	1.0	1.00
643	0.50	1.0	2.0	0.50
644	1.00	1.0	2.0	0.25
645	0.50	1.0	1.0	0.00
646	1.00	1.0	2.0	0.50
647	0.50	0.5	1.0	2.00
648	2.00	2.0	1.0	1.00
649	0.50	1.0	0.5	1.00
716	0.50	0.5	0.0	1.00

717	1.00	0.5	1.0	2.00
718	1.00	1.0	2.0	0.00
719	0.50	0.5	0.0	1.00
720	1.00	4.0	1.0	1.00
721	0.50	1.0	1.0	2.00
785	0.50	0.5	0.0	0.50
786	1.00	1.0	0.0	1.00
787	1.00	0.5	0.0	0.50
788	0.50	0.5	0.0	2.00
789	2.00	2.0	1.0	1.00
790	2.00	2.0	1.0	1.00
791	1.00	2.0	0.5	1.00
792	1.00	4.0	1.0	1.00
793	0.50	1.0	1.0	1.00
794	0.50	0.5	1.0	1.00
795	0.50	0.5	1.0	1.00
796	1.00	1.0	1.0	0.50
797	0.25	1.0	0.5	2.00
798	1.00	1.0	0.5	0.50
799	2.00	0.5	2.0	0.50
800	2.00	2.0	1.0	1.00
801	0.25	0.5	0.0	1.00

	against_fairy	against_fight	against_fire	against_flying \
pokedex_number				
144	1.00	1.00	2.00	1.0
145	1.00	0.50	1.00	0.5
146	0.50	0.50	0.50	1.0
150	1.00	0.50	1.00	1.0
151	1.00	0.50	1.00	1.0
243	1.00	1.00	1.00	0.5
244	0.50	1.00	0.50	1.0
245	1.00	1.00	0.50	1.0
249	1.00	0.25	1.00	1.0
250	0.50	0.50	0.50	1.0
251	1.00	0.50	2.00	2.0
377	1.00	2.00	0.50	0.5
378	1.00	2.00	2.00	1.0
379	0.50	2.00	2.00	0.5
380	2.00	0.50	0.50	1.0
381	2.00	0.50	0.50	1.0
382	1.00	1.00	0.50	1.0
383	1.00	1.00	1.00	1.0
384	2.00	0.50	0.50	1.0
385	0.50	1.00	2.00	0.5
386	1.00	0.50	1.00	1.0
480	1.00	0.50	1.00	1.0

481	1.00	0.50	1.00	1.0
482	1.00	0.50	1.00	1.0
483	1.00	2.00	1.00	0.5
484	2.00	1.00	0.25	1.0
485	0.25	2.00	1.00	0.5
486	1.00	2.00	1.00	1.0
487	2.00	0.00	0.50	1.0
488	1.00	0.50	1.00	1.0
490	1.00	1.00	0.50	1.0
491	2.00	2.00	1.00	1.0
492	1.00	1.00	2.00	2.0
493	1.00	2.00	1.00	1.0
494	0.50	0.50	0.50	1.0
638	1.00	2.00	2.00	1.0
639	2.00	2.00	0.50	1.0
640	2.00	1.00	2.00	4.0
641	1.00	0.50	1.00	1.0
642	1.00	0.50	1.00	0.5
643	1.00	1.00	0.25	1.0
644	2.00	1.00	0.50	0.5
645	1.00	0.50	1.00	1.0
646	2.00	2.00	1.00	1.0
647	2.00	1.00	0.50	2.0
648	1.00	1.00	1.00	1.0
649	0.50	1.00	4.00	1.0
716	1.00	0.50	1.00	1.0
717	2.00	1.00	1.00	1.0
718	2.00	1.00	0.50	1.0
719	1.00	1.00	0.50	0.5
720	1.00	0.00	1.00	1.0
721	0.50	1.00	0.25	1.0
785	1.00	0.50	1.00	0.5
786	1.00	0.25	1.00	1.0
787	1.00	0.50	2.00	2.0
788	1.00	0.50	0.50	1.0
789	1.00	0.50	1.00	1.0
790	1.00	0.50	1.00	1.0
791	0.50	1.00	2.00	0.5
792	1.00	0.00	1.00	1.0
793	0.50	1.00	0.50	0.5
794	2.00	0.50	2.00	4.0
795	2.00	0.50	2.00	4.0
796	1.00	1.00	1.00	0.5
797	0.50	1.00	2.00	0.5
798	0.50	2.00	4.00	1.0
799	4.00	2.00	0.50	1.0
800	1.00	0.50	1.00	1.0

```
801                    0.50           1.00          2.00             0.5

                ...                  japanese_name  percentage_male  \
pokedex_number  ...
144             ...                        Freezer              NaN
145             ...                        Thunder              NaN
146             ...                           Fire              NaN
150             ...                         Mewtwo              NaN
151             ...                            Mew              NaN
243             ...                         Raikou              NaN
244             ...                          Entei              NaN
245             ...                        Suicune              NaN
249             ...                          Lugia              NaN
250             ...                          Houou              NaN
251             ...                         Celebi              NaN
377             ...                       Regirock              NaN
378             ...                         Regice              NaN
379             ...                      Registeel              NaN
380             ...                         Latias              0.0
381             ...                         Latios            100.0
382             ...                         Kyogre              NaN
383             ...                        Groudon              NaN
384             ...                       Rayquaza              NaN
385             ...                        Jirachi              NaN
386             ...                         Deoxys              NaN
480             ...                          Yuxie              NaN
481             ...                          Emrit              NaN
482             ...                         Agnome              NaN
483             ...                         Dialga              NaN
484             ...                         Palkia              NaN
485             ...                        Heatran             50.0
486             ...                      Regigigas              NaN
487             ...      Giratina (another Forme)               NaN
488             ...                      Cresselia              0.0
490             ...                        Manaphy              NaN
491             ...                        Darkrai              NaN
492             ...            Shaymin (sky Forme)              NaN
493             ...                         Arceus              NaN
494             ...                        Victini              NaN
638             ...                        Cobalon              NaN
639             ...                      Terrakion              NaN
640             ...                       Virizion              NaN
641             ...       Tornelos (keshin Forme)             100.0
642             ...       Voltolos (keshin Forme)             100.0
643             ...                       Reshiram              NaN
644             ...                         Zekrom              NaN
645             ...        Landlos (keshin Forme)             100.0
```

```
646             …                      Kyurem                NaN
647             …    Keldeo (itsumo No Sugata)               NaN
648             …        Meloetta (step Forme)               NaN
649             …                    Genesect                NaN
716             …                     Xerneas                NaN
717             …                     Yveltal                NaN
718             …           Zygarde (10% Forme)              NaN
719             …                     Diancie                NaN
720             …  Hoopa (imashimerareshi Hoopa)             NaN
721             …                   Volcanion                NaN
785             …                  Kapu-kokeko               NaN
786             …                  Kapu-tetefu               NaN
787             …                   Kapu-bulul               NaN
788             …                  Kapu-rehire               NaN
789             …                      Cosmog                NaN
790             …                    Cosmovum                NaN
791             …                    Solgaleo                NaN
792             …                      Lunala                NaN
793             …                     Uturoid                NaN
794             …                   Massivoon                NaN
795             …                   Pheroache                NaN
796             …                  Denjyumoku                NaN
797             …                   Tekkaguya                NaN
798             …                  Kamiturugi                NaN
799             …                   Akuziking                NaN
800             …                     Necrozma               NaN
801             …                     Magearna               NaN

                sp_attack  sp_defense  speed     type1    type2  weight_kg  \
pokedex_number
144                    95         125     85       ice   flying       55.4
145                   125          90    100  electric   flying       52.6
146                   125          85     90      fire   flying       60.0
150                   194         120    140   psychic      NaN      122.0
151                   100         100    100   psychic      NaN        4.0
243                   115         100    115  electric      NaN      178.0
244                    90          75    100      fire      NaN      198.0
245                    90         115     85     water      NaN      187.0
249                    90         154    110   psychic   flying      216.0
250                   110         154     90      fire   flying      199.0
251                   100         100    100   psychic    grass        5.0
377                    50         100     50      rock      NaN      230.0
378                   100         200     50       ice      NaN      175.0
379                    75         150     50     steel      NaN      205.0
380                   140         150    110    dragon  psychic       40.0
381                   160         120    110    dragon  psychic       60.0
382                   180         160     90     water      NaN      352.0
```

383	150	90	90	ground	NaN	950.0
384	180	100	115	dragon	flying	206.5
385	100	100	100	steel	psychic	1.1
386	95	90	180	psychic	NaN	60.8
480	75	130	95	psychic	NaN	0.3
481	105	105	80	psychic	NaN	0.3
482	125	70	115	psychic	NaN	0.3
483	150	100	90	steel	dragon	683.0
484	150	120	100	water	dragon	336.0
485	130	106	77	fire	steel	430.0
486	80	110	100	normal	NaN	420.0
487	120	100	90	ghost	dragon	750.0
488	75	130	85	psychic	NaN	85.6
490	100	100	100	water	NaN	1.4
491	135	90	125	dark	NaN	50.5
492	120	75	127	grass	grass	2.1
493	120	120	120	normal	NaN	320.0
494	100	100	100	psychic	fire	4.0
638	90	72	108	steel	fighting	250.0
639	72	90	108	rock	fighting	260.0
640	90	129	108	grass	fighting	200.0
641	110	90	121	flying	NaN	63.0
642	145	80	101	electric	flying	61.0
643	150	120	90	dragon	fire	330.0
644	120	100	90	dragon	electric	345.0
645	105	80	91	ground	flying	68.0
646	170	100	95	dragon	ice	325.0
647	129	90	108	water	fighting	48.5
648	77	77	128	normal	psychic	6.5
649	120	95	99	bug	steel	82.5
716	131	98	99	fairy	NaN	215.0
717	131	98	99	dark	flying	203.0
718	91	95	85	dragon	ground	284.6
719	160	110	110	rock	fairy	8.8
720	170	130	80	psychic	ghost	NaN
721	130	90	70	fire	water	195.0
785	95	75	130	electric	fairy	20.5
786	130	115	95	psychic	fairy	18.6
787	85	95	75	grass	fairy	45.5
788	95	130	85	water	fairy	21.2
789	29	31	37	psychic	NaN	0.1
790	29	131	37	psychic	NaN	999.9
791	113	89	97	psychic	steel	230.0
792	137	107	97	psychic	ghost	120.0
793	127	131	103	rock	poison	55.5
794	53	53	79	bug	fighting	333.6
795	137	37	151	bug	fighting	25.0

796	173	71	83	electric	NaN	100.0
797	107	101	61	steel	flying	999.9
798	59	31	109	grass	steel	0.1
799	97	53	43	dark	dragon	888.0
800	127	89	79	psychic	NaN	230.0
801	130	115	65	steel	fairy	80.5

	generation	is_legendary
pokedex_number		
144	1	1
145	1	1
146	1	1
150	1	1
151	1	1
243	2	1
244	2	1
245	2	1
249	2	1
250	2	1
251	2	1
377	3	1
378	3	1
379	3	1
380	3	1
381	3	1
382	3	1
383	3	1
384	3	1
385	3	1
386	3	1
480	4	1
481	4	1
482	4	1
483	4	1
484	4	1
485	4	1
486	4	1
487	4	1
488	4	1
490	4	1
491	4	1
492	4	1
493	4	1
494	5	1
638	5	1
639	5	1
640	5	1

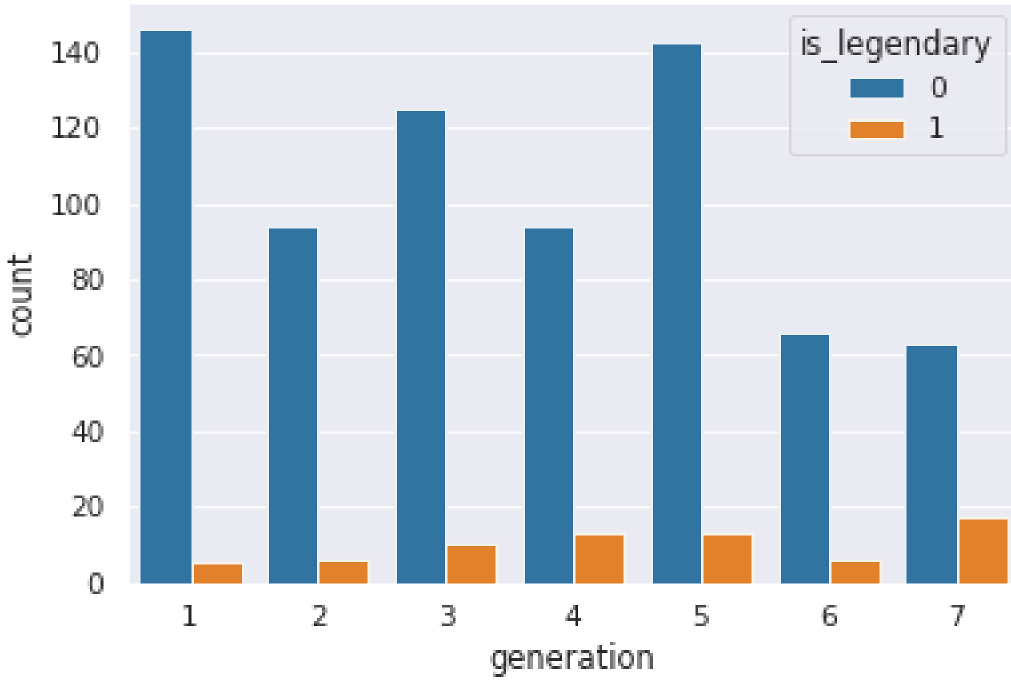
```
641                       5              1
642                       5              1
643                       5              1
644                       5              1
645                       5              1
646                       5              1
647                       5              1
648                       5              1
649                       5              1
716                       6              1
717                       6              1
718                       6              1
719                       6              1
720                       6              1
721                       6              1
785                       7              1
786                       7              1
787                       7              1
788                       7              1
789                       7              1
790                       7              1
791                       7              1
792                       7              1
793                       7              1
794                       7              1
795                       7              1
796                       7              1
797                       7              1
798                       7              1
799                       7              1
800                       7              1
801                       7              1

[70 rows x 40 columns]
```

Proving the rarity of Legendary Pokemon

[5]:
```
sns.countplot(x= df.generation, hue= df.is_legendary)
```

[5]:
```
<AxesSubplot:xlabel='generation', ylabel='count'>
```



The plot shows the generation on x-axis and the number of Pokemon in each Generation. From this we can see for ourselves the rarity of Legendary Pokémon.

Comparision of Stats of Legendary and Non-Legendary Pokemon Let's start the comparsion by seperating Legendary and Non-Legendary Pokemon,

[6]:
```
legend = pd.DataFrame(df[(df['is_legendary']==1)])
nonleg = pd.DataFrame(df[(df['is_legendary']==0)])
```

Once seperated, we can now select the stats columns of both dataframes, take each of their mean values and save it in a varible.

[7]:
```
legmean = pd.DataFrame(legend[['hp', 'attack', 'defense', 'sp_attack',␣
 ↪'sp_defense', 'speed']].mean(axis=0))
nlegmean = pd.DataFrame(nonleg[['hp', 'attack', 'defense', 'sp_attack',␣
 ↪'sp_defense', 'speed']].mean(axis=0))
legmean = legmean.T
nlegmean = nlegmean.T
```

Now we concatenate both the dataframes into one dataframe, with a new column that denotes which row is legendary,

[8]:
```
df4 = pd.concat([legmean, nlegmean], axis=0, ignore_index=False)
df4['col'] = (len(legmean)*(0,) + len(nlegmean)*(1,))
species = ['Legendary', 'Non-Legendary']
```

```
df4['species'] = species
```

Now we can start plotting the stats,

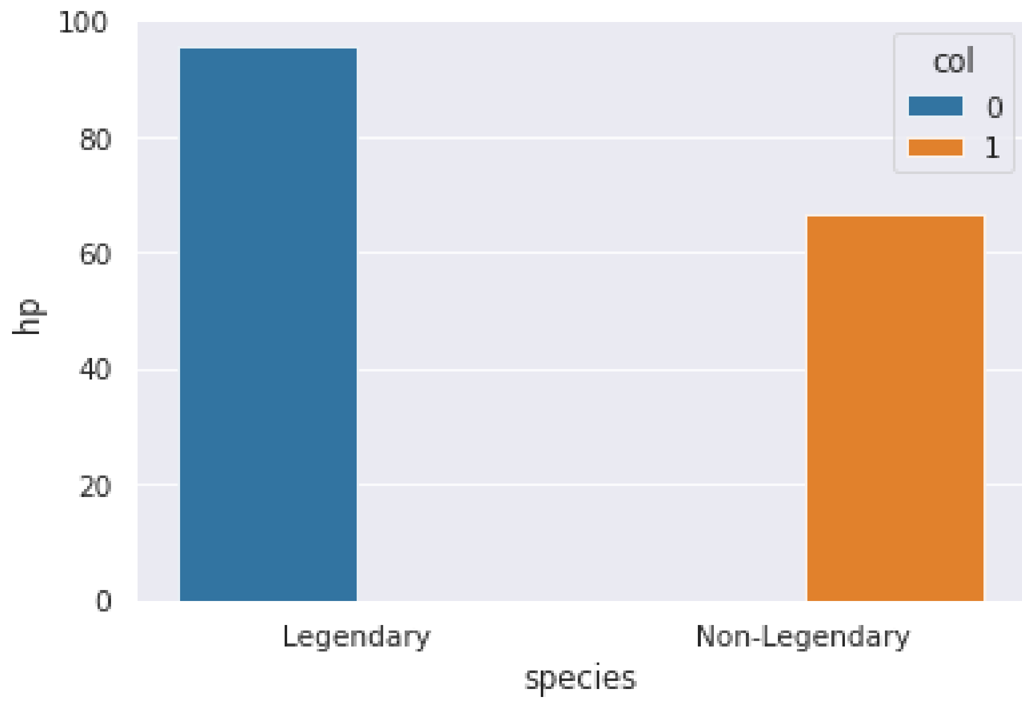
Difference in Health stats of both species,

[ ]:
```

```

[9]:
```
sns.barplot(x='species', y='hp', hue='col', data=df4)
```

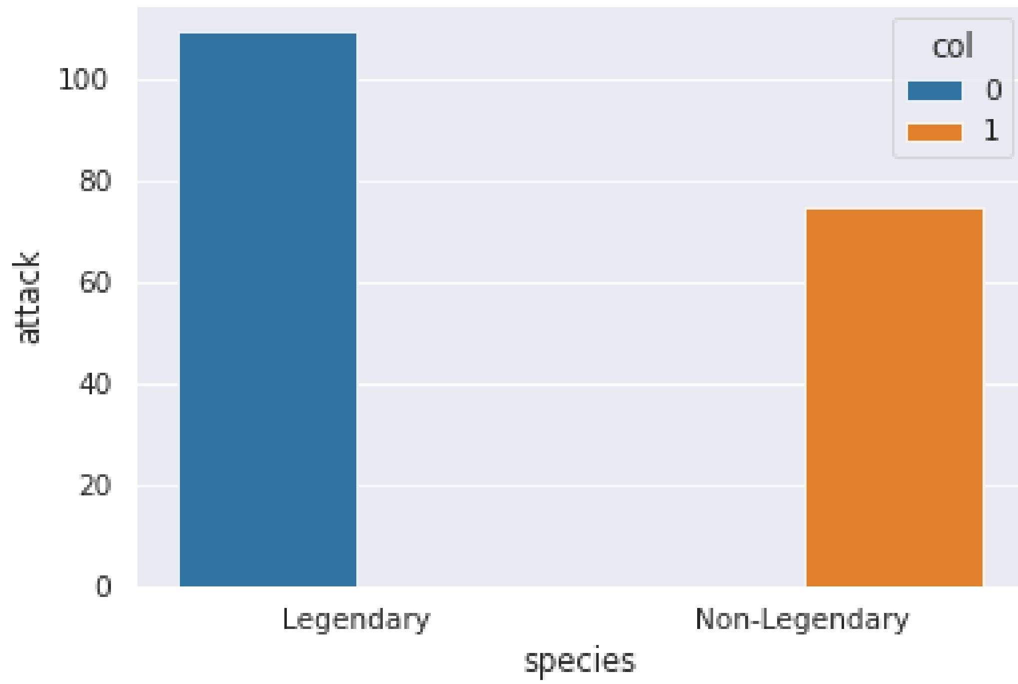
[9]: <AxesSubplot:xlabel='species', ylabel='hp'>



Difference in Attack stats of both species,

[10]:
```
sns.barplot(x='species', y='attack', hue='col', data=df4)
```

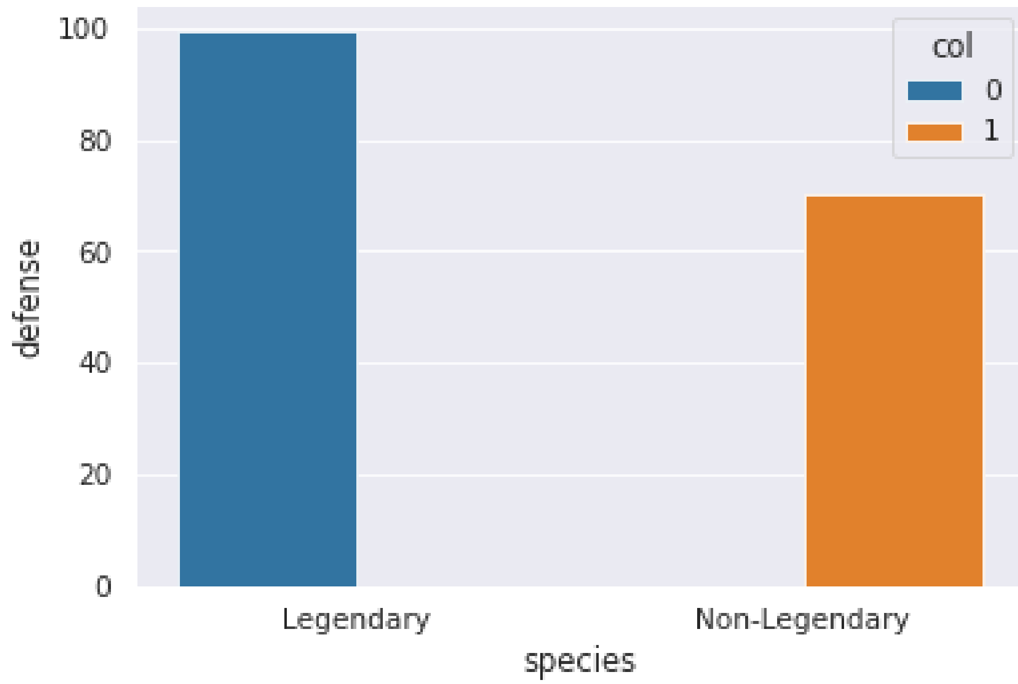
[10]: <AxesSubplot:xlabel='species', ylabel='attack'>



Difference in Defense stats of both species,

[11]:
```
sns.barplot(x='species', y='defense', hue='col', data=df4)
```

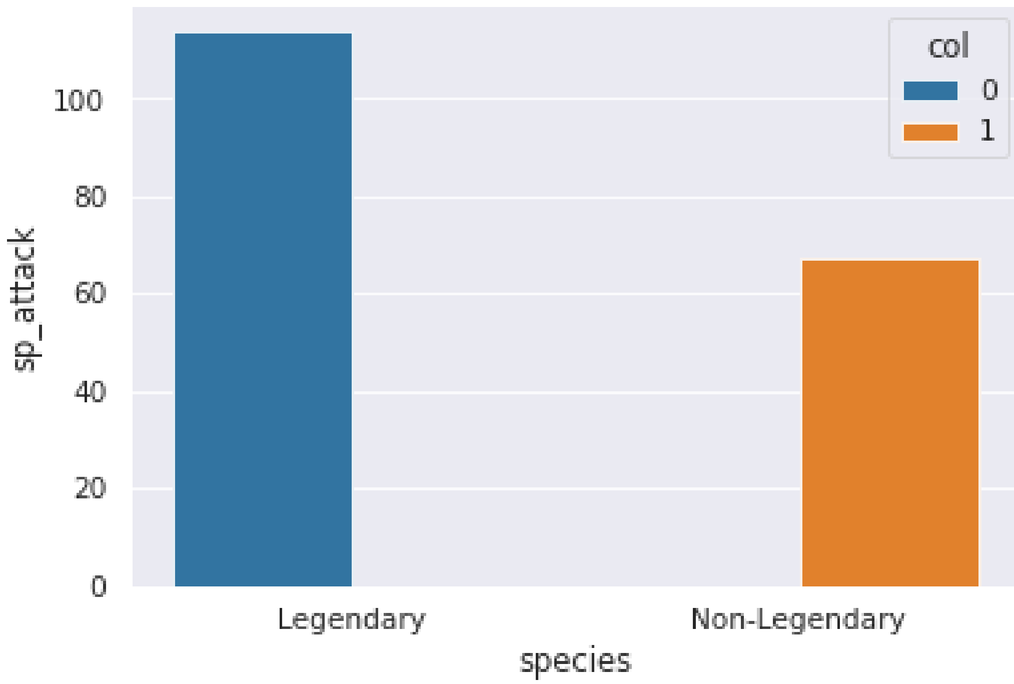
[11]: <AxesSubplot:xlabel='species', ylabel='defense'>



Difference in Special Attack stats of both species,

[12]:
```
sns.barplot(x='species', y='sp_attack', hue='col', data=df4)
```

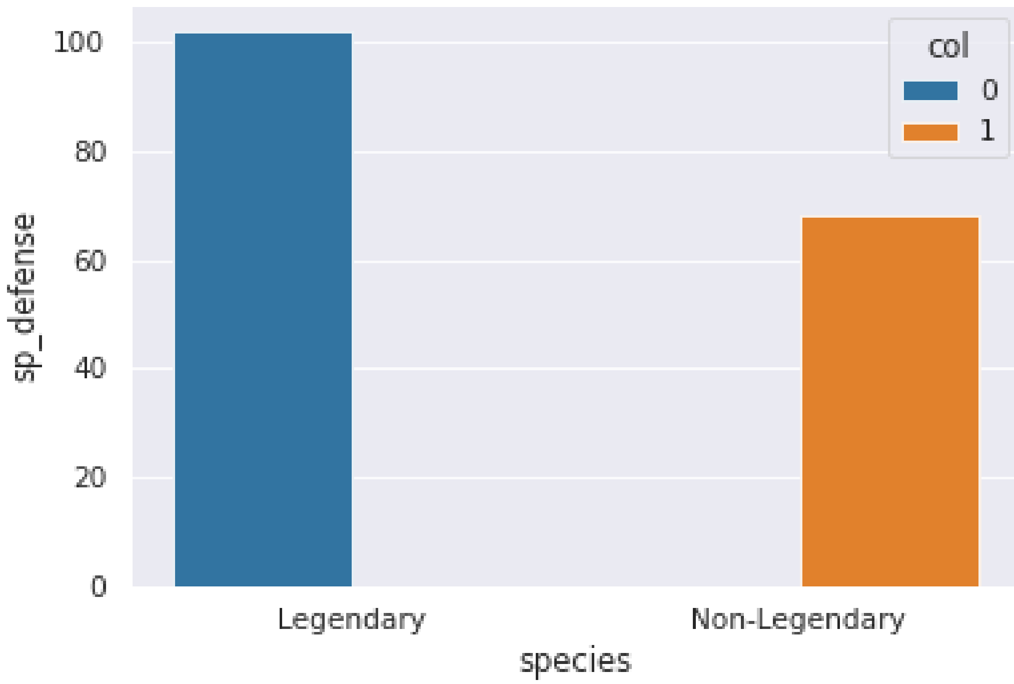
[12]: <AxesSubplot:xlabel='species', ylabel='sp_attack'>



Difference in Special Defense stat of both species,

[13]:
```
sns.barplot(x='species', y='sp_defense', hue='col', data=df4)
```

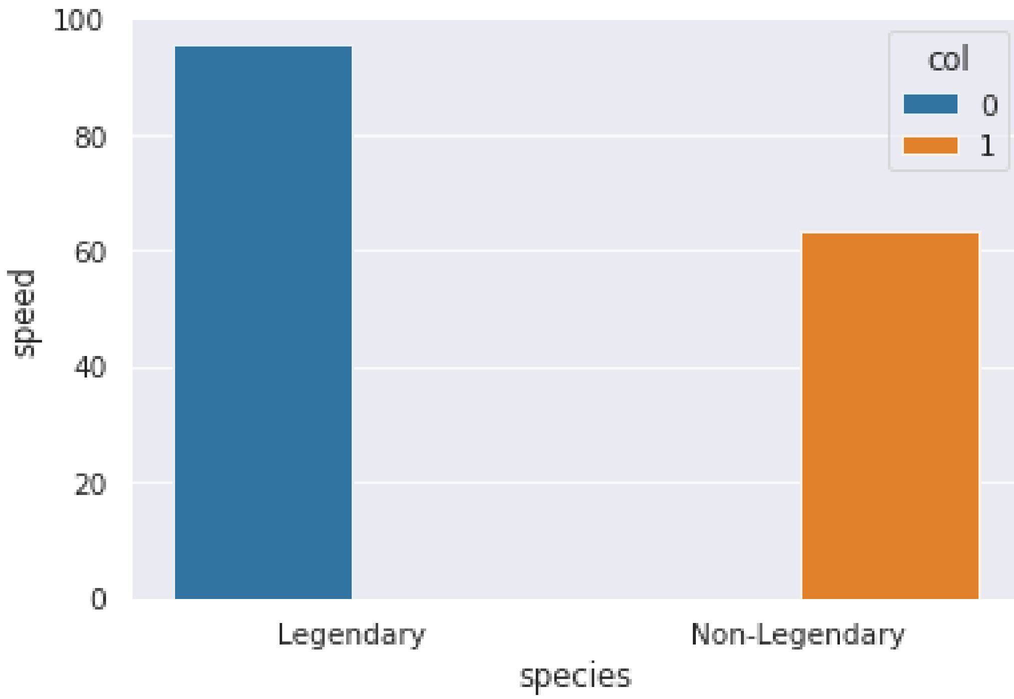
[13]: <AxesSubplot:xlabel='species', ylabel='sp_defense'>



Difference in Speed stat of both species,

[14]:
```
sns.barplot(x='species', y='speed', hue='col', data=df4)
```

[14]: <AxesSubplot:xlabel='species', ylabel='speed'>



Looking at all these stats, we can prove the vast difference in Offense, Defense and Speed statistics of both types of Pokemon. With the Legendary Pokemon being in the league of their own.

2 Data Preprocessing

2.1 Data Cleaning

To remove unwanted column, viewing all column names:

[15]:
```
df.columns
```

[15]:
```
Index(['name', 'abilities', 'against_bug', 'against_dark', 'against_dragon',
       'against_electric', 'against_fairy', 'against_fight', 'against_fire',
       'against_flying', 'against_ghost', 'against_grass', 'against_ground',
       'against_ice', 'against_normal', 'against_poison', 'against_psychic',
       'against_rock', 'against_steel', 'against_water', 'attack',
       'base_egg_steps', 'base_happiness', 'base_total', 'capture_rate',
       'classfication', 'defense', 'experience_growth', 'height_m', 'hp',
       'japanese_name', 'percentage_male', 'sp_attack', 'sp_defense', 'speed',
       'type1', 'type2', 'weight_kg', 'generation', 'is_legendary'],
      dtype='object')
```

Dropping the unwanted columns such as all the *against columns, height, weight and percentage* *male columns* as they do not contribute towards a Pokémon being **Legendary**.

[16]:
```
df = df.drop(['against_bug', 'against_dark', 'against_dragon','height_m',
       'against_electric', 'against_fairy', 'against_fight', 'against_fire',
       'against_flying', 'against_ghost', 'against_grass', 'against_ground',
       'against_ice', 'against_normal', 'against_poison', 'against_psychic',
       'against_rock', 'against_steel', 'against_water',␣
 ↪'percentage_male','type2', 'weight_kg', ], axis = 1)
```

Now we will split the dataset into two sub datasets, namely, **df_train** and **df_test**. This done before the data cleaning stage to make sure that during the cleaning phase, the test data is not exposed to the training set, to avoid over fitting of the model.

[17]:
```
df_train, df_test = train_test_split(df)
print(df_train.shape, df_test.shape)
```

```
(600, 18) (201, 18)
```

Now we will look for the missing data in the **df_train** and **df_test** and see if we can make do with removing them or should we apply a different approach to clean data.

[18]:
```
print('Null values in Training Set: \n', df_train.isna().sum())
print('\n')
print('Null values in the Testing Set: \n', df_test.isna().sum())
```

```
Null values in Training Set:
 name                 0
abilities            0
attack               0
base_egg_steps       0
base_happiness       0
base_total           0
capture_rate         0
classfication        0
defense              0
experience_growth    0
hp                   0
japanese_name        0
sp_attack            0
sp_defense           0
speed                0
type1                0
generation           0
is_legendary         0
dtype: int64


Null values in the Testing Set:
 name                 0
```

```
abilities            0
attack               0
base_egg_steps       0
base_happiness       0
base_total           0
capture_rate         0
classfication        0
defense              0
experience_growth    0
hp                   0
japanese_name        0
sp_attack            0
sp_defense           0
speed                0
type1                0
generation           0
is_legendary         0
dtype: int64
```

From above we can see that there are no null values in both the sub datasets. So now we will move on to look at the data types of the features.

[19]:
```
df_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 600 entries, 255 to 643
Data columns (total 18 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   name               600 non-null    object
 1   abilities          600 non-null    object
 2   attack             600 non-null    int64
 3   base_egg_steps     600 non-null    int64
 4   base_happiness     600 non-null    int64
 5   base_total         600 non-null    int64
 6   capture_rate       600 non-null    object
 7   classfication      600 non-null    object
 8   defense            600 non-null    int64
 9   experience_growth  600 non-null    int64
 10  hp                 600 non-null    int64
 11  japanese_name      600 non-null    object
 12  sp_attack          600 non-null    int64
 13  sp_defense         600 non-null    int64
 14  speed              600 non-null    int64
 15  type1              600 non-null    object
 16  generation         600 non-null    int64
 17  is_legendary       600 non-null    int64
dtypes: int64(12), object(6)
```

```
memory usage: 89.1+ KB
```

Looking at the data types we can see that we have two types, **int64** and **object**, we can work with int64 but we need to tranform the object because the Machine Learning model can not read strings as attributes. So we will enocde categorical attributes, to make it readable, in the feature engineering phase of this pipeline.

3 Feature Engineering

The first step into feature engineering is to select your features and target label. Since, we are wroking to find the Legendary status of Pokemon's based on their stats, we will select the column **is_Legendary** as our target variable and the other columns will the features that will be used to predict that.

3.0.1 Seperating Features and Target Variable

[20]:
```
x_train = df_train.drop(['is_legendary'], axis = 1)
y_train = df_train['is_legendary']

x_test = df_test.drop(['is_legendary'], axis = 1)
y_test = df_test['is_legendary']

print('x_test:', x_test.shape)
print('y_test:', y_test.shape)
print('x_train:', x_train.shape)
print('y_train:', y_train.shape)
```

```
x_test: (201, 17)
y_test: (201,)
x_train: (600, 17)
y_train: (600,)
```

Now that we are done with seperating Target Variable from the Features, we can move on to assigning categorical attributes to the features. For this we will use **OneHotEncoder()** from the ***preprocessing*** library of Scikit-Learn and transfrom both the sub datasets.

3.0.2 Encoding categorical attributes

[21]:
```
ohe = OneHotEncoder(handle_unknown = 'ignore')
ohe.fit(x_train)
x_train = ohe.transform(x_train)
x_test = ohe.transform(x_test)
print('x_test:', x_test.shape)
print('x_train:', x_train.shape)
```

```
x_test: (201, 2919)
x_train: (600, 2919)
```

The next step in feature engineering is to standardize the sub datasets. For this we will use **StandardScaler()** from the ***preprocesing*** library of Scikit-Learn.

3.0.3 Standardizing the sub datasets

[22]:
```
stand = StandardScaler(with_mean = False)
stand.fit(x_train)
x_train = stand.transform(x_train)
x_test = stand.transform(x_test)
print('x_train:', x_train.shape)
print('x_test:', x_test.shape)
```

```
x_train: (600, 2919)
x_test: (201, 2919)
```

The third step in feature engineering phase of this pipeline we will look at the domensionality reduction to remove the less important variable from the data, this will reduce the complexity of the model and also curb any overfitting of the model.

For this we will use **PCA** from the ***decomposition*** library of Scikit-Learn.

3.0.4 Dimensionality Reduction

[23]:
```
dimred = PCA(n_components = 100)
dimred.fit(x_train.toarray())
x_train = dimred.transform(x_train.toarray())
x_test = dimred.transform(x_test.toarray())
print('x_train:', x_train.shape)
print('x_test:', x_test.shape)
```

```
x_train: (600, 100)
x_test: (201, 100)
```

The last step in the feature engineering phase is to balance the datasets. Balancing is done to make sure that we do not have any imbalance classes that can lead to underfitting or over fitting of the model.

Balancing the data

[24]:
```
osam = SMOTE()
x_train, y_train = osam.fit_resample(x_train, y_train)
print(x_train.shape, y_train.shape)
```

```
(1092, 100) (1092,)
```

Once we are done with balancing the data, we can move on from feature engineering to training Machine Learning Models.

4 Model Training & Testing

Looking at the dataset, we can see that it consists of discrete variable, so we will be going with classification models. The 3 models we have choosen are for training are ***Support Vector Machine, Decision Tree Classifier and KN Neighbors Models***.

Through out this pipeline we will be using **GridSearchCV()** library from Scikit-Learn to find the best barameters for the said model and then find the ***F1 Score, Confusion Matrix, Precision, Recall and Accuracy*** to interpret the efficiency of the models.

4.0.1 Support Vector Machine Model

Training an Support Vector Machine Model and fiding the best hyper-parameters using GridSearchCV from Scikit-Learn. Here, we will be defining a dictionary of parameters for the SVM Model, the two main hyper-parameters for svm are ***C and Kernel***.

[25]:
```
svm = SVC()
param = {
    'C': [1, 10, 100, 1000],
    'kernel': ['linear', 'rbf']
}
gs_svm = GridSearchCV(svm, param_grid = param, cv =5)
gs_svm.fit(x_train, y_train)
gs_svm.cv_results_
gsdf_svm = pd.DataFrame(gs_svm.cv_results_)
gsdf_svm
```

[25]:
```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time param_C  \
0       0.014823      0.008004         0.004363        0.004858       1
1       0.016098      0.000378         0.005195        0.000200       1
2       0.007009      0.000702         0.000812        0.000036      10
3       0.014485      0.000495         0.004360        0.000220      10
4       0.007004      0.000709         0.000804        0.000044     100
5       0.014463      0.000643         0.004252        0.000127     100
6       0.007054      0.000702         0.000804        0.000041    1000
7       0.014465      0.000652         0.004282        0.000179    1000

  param_kernel                          params  split0_test_score  \
0       linear     {'C': 1, 'kernel': 'linear'}          0.990868
1          rbf        {'C': 1, 'kernel': 'rbf'}          0.990868
2       linear    {'C': 10, 'kernel': 'linear'}          0.990868
3          rbf       {'C': 10, 'kernel': 'rbf'}          0.990868
4       linear   {'C': 100, 'kernel': 'linear'}          0.990868
```

```
5          rbf      {'C': 100, 'kernel': 'rbf'}           0.990868
6       linear  {'C': 1000, 'kernel': 'linear'}           0.990868
7          rbf     {'C': 1000, 'kernel': 'rbf'}           0.990868

   split1_test_score  split2_test_score  split3_test_score  split4_test_score  \
0           0.986301           0.995413           0.958716           0.986239
1           0.995434           0.995413           0.990826           1.000000
2           0.986301           0.995413           0.958716           0.986239
3           0.995434           1.000000           0.990826           1.000000
4           0.986301           0.995413           0.958716           0.986239
5           0.990868           1.000000           0.990826           1.000000
6           0.986301           0.995413           0.958716           0.986239
7           0.990868           1.000000           0.990826           1.000000

   mean_test_score  std_test_score  rank_test_score
0         0.983507        0.012852                5
1         0.994508        0.003425                4
2         0.983507        0.012852                5
3         0.995425        0.004094                1
4         0.983507        0.012852                5
5         0.994512        0.004481                2
6         0.983507        0.012852                5
7         0.994512        0.004481                2
```

After training the model, we find the best parameters and the score of that parameters. In this case,

[26]:
```
print("\n The best score across ALL searched params:\n",gs_svm.best_score_)
print("\n The best parameters across ALL searched params:\n",gs_svm.
 ↪best_params_)
```

```
 The best score across ALL searched params:
 0.9954254115872816

 The best parameters across ALL searched params:
 {'C': 10, 'kernel': 'rbf'}
```

Now that we have the best parameters for SVM Model, we can test it find the scores.

Testing the model and checking metric score:

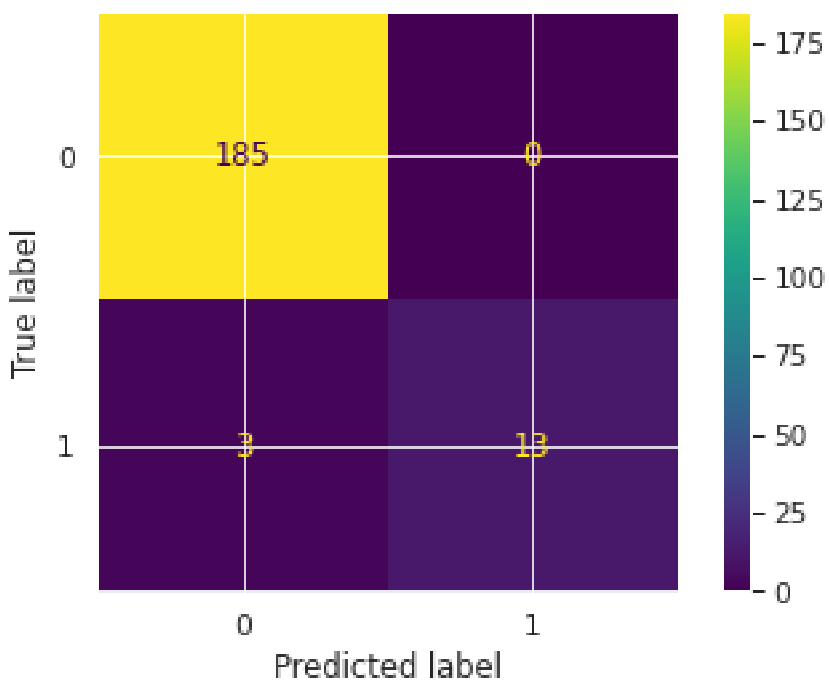
[27]:
```
y_pred_svm = gs_svm.best_estimator_.predict(x_test)
cfx_svm = confusion_matrix(y_test, y_pred_svm)
f1_svm =  f1_score(y_test, y_pred_svm)
accu_svm = accuracy_score(y_test, y_pred_svm)
prec_svm = precision_score(y_test, y_pred_svm)
rec_svm = recall_score(y_test, y_pred_svm)
```

```python
s_svm = pd.Series({'Model': 'Support Vector Machines',
                'F1 Score': f1_svm,
                'Accuracy': accu_svm,
                'Precision': prec_svm,
                'Recall' : rec_svm})
s_svm = pd.DataFrame(s_svm)
print(s_svm)

ConfusionMatrixDisplay(cfx_svm).plot()
```

```
                                 0
Model      Support Vector Machines
F1 Score                  0.896552
Accuracy                  0.985075
Precision                      1.0
Recall                      0.8125
```

[27]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7fb5cf9f6dc0>



From the above values we can tell that our model has been trained with a ***F1 score of 89.65%, Accuracy of 98.51%, Precision of 100% and Recall of 81.13%***. Now we will move to another model:

4.0.2 Decision Tree Classifier

Training a Decision Tree Classifier Model and fiding the best hyper-parameters using GridSearchCV from Scikit-Learn. Here, we will be defining a dictionary of parameters for the DTC Model, the hyper-parameters for DTC are ***Criterion, splitter and max_depth***.

[28]:
```
dtc = DecisionTreeClassifier()
params = {
    'criterion': ['gini', 'entropy'],
    'splitter': ['best', 'random'],
    'max_depth': [10, 100, 1000, 2000, 5000, 10000, 20000]
}
gs_dtc = GridSearchCV(dtc, param_grid = params ,cv = 5)
fit = gs_dtc.fit(x_train, y_train)
gsdf_dtc = pd.DataFrame(gs_dtc.cv_results_)
gsdf_dtc
```

[28]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	\
0	0.033703	0.004935	0.000352	0.000023	
1	0.002894	0.000363	0.000341	0.000003	
2	0.033522	0.004951	0.000343	0.000004	
3	0.003265	0.000485	0.000335	0.000002	
4	0.033519	0.004968	0.000336	0.000002	
5	0.002947	0.000464	0.000335	0.000003	
6	0.033551	0.004952	0.000338	0.000002	
7	0.002466	0.000285	0.000332	0.000002	
8	0.033539	0.004942	0.000338	0.000003	
9	0.002988	0.000289	0.000335	0.000001	
10	0.033522	0.004960	0.000337	0.000004	
11	0.003156	0.000126	0.000335	0.000001	
12	0.033509	0.004926	0.000336	0.000003	
13	0.002901	0.000320	0.000335	0.000005	
14	0.036927	0.002831	0.000332	0.000003	
15	0.003001	0.000496	0.000332	0.000007	
16	0.036927	0.002852	0.000335	0.000007	
17	0.002936	0.000581	0.000336	0.000004	
18	0.036908	0.002820	0.000335	0.000005	
19	0.002786	0.000452	0.000330	0.000004	
20	0.036887	0.002824	0.000330	0.000004	
21	0.002840	0.000229	0.000332	0.000005	
22	0.036872	0.002819	0.000331	0.000004	
23	0.002805	0.000381	0.000327	0.000002	
24	0.036867	0.002821	0.000329	0.000003	
25	0.003500	0.000362	0.000330	0.000002	
26	0.036881	0.002799	0.000331	0.000005	
27	0.003100	0.000386	0.000331	0.000002	

	param_criterion	param_max_depth	param_splitter	\
0	gini	10	best	
1	gini	10	random	
2	gini	100	best	
3	gini	100	random	
4	gini	1000	best	
5	gini	1000	random	
6	gini	2000	best	
7	gini	2000	random	
8	gini	5000	best	
9	gini	5000	random	
10	gini	10000	best	
11	gini	10000	random	
12	gini	20000	best	
13	gini	20000	random	
14	entropy	10	best	
15	entropy	10	random	
16	entropy	100	best	
17	entropy	100	random	
18	entropy	1000	best	
19	entropy	1000	random	
20	entropy	2000	best	
21	entropy	2000	random	
22	entropy	5000	best	
23	entropy	5000	random	
24	entropy	10000	best	
25	entropy	10000	random	
26	entropy	20000	best	
27	entropy	20000	random	

	params	split0_test_score	\
0	{'criterion': 'gini', 'max_depth': 10, 'splitt…	0.972603	
1	{'criterion': 'gini', 'max_depth': 10, 'splitt…	0.990868	
2	{'criterion': 'gini', 'max_depth': 100, 'split…	0.968037	
3	{'criterion': 'gini', 'max_depth': 100, 'split…	0.977169	
4	{'criterion': 'gini', 'max_depth': 1000, 'spli…	0.968037	
5	{'criterion': 'gini', 'max_depth': 1000, 'spli…	0.990868	
6	{'criterion': 'gini', 'max_depth': 2000, 'spli…	0.963470	
7	{'criterion': 'gini', 'max_depth': 2000, 'spli…	0.990868	
8	{'criterion': 'gini', 'max_depth': 5000, 'spli…	0.977169	
9	{'criterion': 'gini', 'max_depth': 5000, 'spli…	0.958904	
10	{'criterion': 'gini', 'max_depth': 10000, 'spl…	0.968037	
11	{'criterion': 'gini', 'max_depth': 10000, 'spl…	0.986301	
12	{'criterion': 'gini', 'max_depth': 20000, 'spl…	0.972603	
13	{'criterion': 'gini', 'max_depth': 20000, 'spl…	0.977169	
14	{'criterion': 'entropy', 'max_depth': 10, 'spl…	0.977169	
15	{'criterion': 'entropy', 'max_depth': 10, 'spl…	0.995434	

```
16  {'criterion': 'entropy', 'max_depth': 100, 'sp…          0.981735
17  {'criterion': 'entropy', 'max_depth': 100, 'sp…          0.990868
18  {'criterion': 'entropy', 'max_depth': 1000, 's…          0.981735
19  {'criterion': 'entropy', 'max_depth': 1000, 's…          0.977169
20  {'criterion': 'entropy', 'max_depth': 2000, 's…          0.977169
21  {'criterion': 'entropy', 'max_depth': 2000, 's…          0.986301
22  {'criterion': 'entropy', 'max_depth': 5000, 's…          0.977169
23  {'criterion': 'entropy', 'max_depth': 5000, 's…          0.986301
24  {'criterion': 'entropy', 'max_depth': 10000, '…          0.977169
25  {'criterion': 'entropy', 'max_depth': 10000, '…          0.977169
26  {'criterion': 'entropy', 'max_depth': 20000, '…          0.977169
27  {'criterion': 'entropy', 'max_depth': 20000, '…          0.954338

    split1_test_score  split2_test_score  split3_test_score  \
0            0.995434           0.986239           0.977064
1            0.986301           0.986239           0.972477
2            1.000000           0.986239           0.972477
3            0.990868           0.977064           0.990826
4            1.000000           0.986239           0.977064
5            0.990868           0.981651           0.981651
6            0.995434           0.986239           0.972477
7            0.981735           0.986239           0.963303
8            0.995434           0.986239           0.972477
9            0.986301           1.000000           0.986239
10           1.000000           0.986239           0.977064
11           0.986301           0.986239           0.986239
12           0.995434           0.986239           0.977064
13           0.995434           1.000000           0.967890
14           1.000000           0.986239           0.986239
15           0.986301           0.995413           0.977064
16           1.000000           0.990826           0.986239
17           0.995434           0.958716           0.977064
18           1.000000           0.981651           0.995413
19           0.977169           0.990826           0.977064
20           1.000000           0.986239           0.995413
21           0.986301           0.995413           0.981651
22           1.000000           0.995413           0.995413
23           0.995434           0.990826           0.977064
24           1.000000           0.990826           0.995413
25           0.990868           0.995413           0.977064
26           1.000000           0.995413           0.986239
27           0.995434           0.977064           0.977064

    split4_test_score  mean_test_score  std_test_score  rank_test_score
0            0.986239         0.983516        0.007970               23
1            0.986239         0.984425        0.006235               20
2            0.995413         0.984433        0.012473               17
```

3	0.995413	0.986268	0.007656	13
4	1.000000	0.986268	0.012603	13
5	0.995413	0.988090	0.005513	11
6	0.995413	0.982607	0.012732	25
7	0.986239	0.981677	0.009630	27
8	1.000000	0.986264	0.010446	15
9	0.986239	0.983537	0.013417	21
10	0.990826	0.984433	0.011042	17
11	0.990826	0.987181	0.001823	12
12	0.990826	0.984433	0.008478	17
13	0.977064	0.983511	0.012164	24
14	0.990826	0.988094	0.007422	9
15	0.990826	0.989008	0.006864	8
16	0.990826	0.989925	0.006063	5
17	0.990826	0.982581	0.013430	26
18	0.995413	0.990842	0.007656	3
19	0.995413	0.983528	0.007964	22
20	0.981651	0.988094	0.008481	9
21	0.995413	0.989016	0.005492	7
22	0.986239	0.990847	0.008171	2
23	1.000000	0.989925	0.007889	6
24	0.995413	0.991764	0.007853	1
25	0.986239	0.985350	0.007322	16
26	0.990826	0.989929	0.007858	4
27	0.995413	0.979863	0.015175	28

After training the model, we find the best parameters and the score of that parameters. In this case,

[29]:
```
print("\n The best score across ALL searched params:\n",gs_dtc.best_score_)
print("\n The best parameters across ALL searched params:\n",gs_dtc.
 ↪best_params_)
```

```
 The best score across ALL searched params:
 0.9917640651836959

 The best parameters across ALL searched params:
 {'criterion': 'entropy', 'max_depth': 10000, 'splitter': 'best'}
```

Now that we have the best parameters we can move on to train the model and check the efficiency of it,

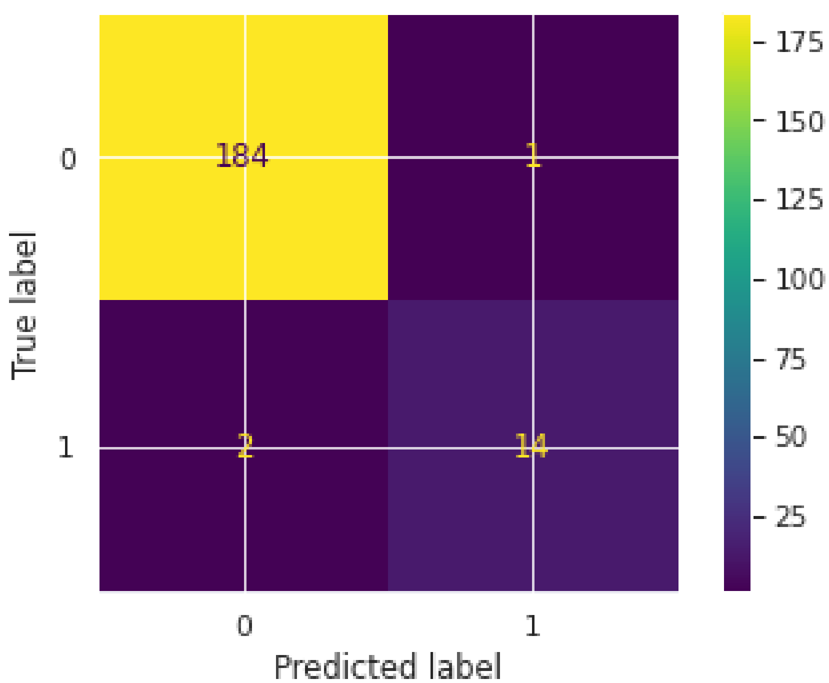
Testing the model and checking metric score:

[30]:
```
y_pred_dtc = gs_dtc.best_estimator_.predict(x_test)
cfx_dtc = confusion_matrix(y_test, y_pred_dtc)
f1_dtc =  f1_score(y_test, y_pred_dtc)
```

```
accu_dtc = accuracy_score(y_test, y_pred_dtc)
prec_dtc = precision_score(y_test, y_pred_dtc)
rec_dtc = recall_score(y_test, y_pred_dtc)
s_dtc = pd.Series({'Model': 'Decision Tree Classifer',
               'F1 Score': f1_dtc,
               'Accuracy': accu_dtc,
               'Precision': prec_dtc,
               'Recall' : rec_dtc})
s_dtc = pd.DataFrame(s_dtc)
print(s_dtc)
ConfusionMatrixDisplay(cfx_dtc).plot()
```

```
                                 0
Model       Decision Tree Classifer
F1 Score                   0.903226
Accuracy                   0.985075
Precision                  0.933333
Recall                        0.875
```

[30]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7fb5cf9d82e0>



From the above values we can tell that our model has been trained with a ***F1 score of 90.32%, Accuracy of 98.50%, Precision of 93.33% and Recall of 87.5%***. Now we will move onto our last model:

4.0.3 K Nearest Neighbors

Training a K Nearest Neighbors Model and fiding the best hyper-parameters using GridSearchCV from Scikit-Learn. Here, we will be defining a dictionary of parameters for the KNN Model, the hyper-parameters for KNN are ***N Neighbors, Weights and Metrics***

[31]:
```
KNN = KNeighborsClassifier()
params_knn= {
    'n_neighbors': [1,3,5,7,9,11,13,15,17,19,21],
    'weights': ['uniform', 'distance'],
    'metric': ['manhattan', 'euclidean', 'minkowski']
}
gs_knn = GridSearchCV(KNN, param_grid = params_knn, cv=5)
gs_knn.fit(x_train, y_train)
gsdf_knn = pd.DataFrame(gs_knn.cv_results_)
gsdf_knn
```

[31]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_metric \
0	0.000764	0.000128	0.023483	0.000244	manhattan
1	0.000677	0.000002	0.017655	0.000056	manhattan
2	0.000674	0.000002	0.024378	0.000022	manhattan
3	0.000665	0.000002	0.018741	0.000362	manhattan
4	0.000675	0.000009	0.024721	0.000598	manhattan
5	0.000670	0.000003	0.019041	0.000530	manhattan
6	0.000679	0.000008	0.024895	0.000674	manhattan
7	0.000677	0.000004	0.019120	0.000501	manhattan
8	0.000672	0.000005	0.024862	0.000583	manhattan
9	0.000671	0.000004	0.019171	0.000758	manhattan
10	0.000678	0.000009	0.024987	0.000532	manhattan
11	0.000679	0.000008	0.019067	0.000558	manhattan
12	0.000676	0.000009	0.024954	0.000578	manhattan
13	0.000674	0.000003	0.019108	0.000558	manhattan
14	0.000673	0.000009	0.024986	0.000553	manhattan
15	0.000668	0.000002	0.019150	0.000545	manhattan
16	0.000670	0.000009	0.024977	0.000570	manhattan
17	0.000671	0.000009	0.019177	0.000575	manhattan
18	0.000674	0.000005	0.025165	0.000778	manhattan
19	0.000668	0.000006	0.019210	0.000562	manhattan
20	0.000671	0.000006	0.025105	0.000572	manhattan
21	0.000669	0.000006	0.019250	0.000571	manhattan
22	0.000964	0.000153	0.040531	0.014663	euclidean
23	0.001030	0.000012	0.030346	0.002601	euclidean
24	0.001033	0.000010	0.039940	0.009029	euclidean
25	0.001030	0.000006	0.030308	0.002629	euclidean
26	0.001032	0.000006	0.038406	0.007632	euclidean
27	0.002631	0.001966	0.033523	0.001680	euclidean
28	0.001030	0.000009	0.042161	0.004582	euclidean

29	0.001839	0.001592	0.032736	0.005820	euclidean
30	0.001022	0.000008	0.046159	0.003800	euclidean
31	0.002629	0.001961	0.035172	0.003089	euclidean
32	0.001028	0.000013	0.042969	0.002082	euclidean
33	0.001834	0.001595	0.027939	0.003374	euclidean
34	0.001026	0.000013	0.038179	0.006177	euclidean
35	0.001797	0.001519	0.027939	0.003556	euclidean
36	0.001030	0.000008	0.038225	0.003771	euclidean
37	0.001040	0.000010	0.031081	0.003230	euclidean
38	0.001043	0.000020	0.041363	0.004383	euclidean
39	0.001041	0.000009	0.029472	0.004055	euclidean
40	0.001024	0.000006	0.035022	0.003661	euclidean
41	0.002602	0.001935	0.032708	0.003959	euclidean
42	0.001021	0.000002	0.046250	0.007519	euclidean
43	0.004198	0.002985	0.035914	0.001771	euclidean
44	0.001030	0.000003	0.037105	0.004494	minkowski
45	0.001024	0.000003	0.027130	0.003350	minkowski
46	0.001027	0.000010	0.038349	0.005299	minkowski
47	0.001028	0.000006	0.034311	0.004826	minkowski
48	0.001026	0.000004	0.036022	0.004221	minkowski
49	0.001039	0.000013	0.027914	0.002514	minkowski
50	0.001028	0.000007	0.038979	0.003198	minkowski
51	0.001034	0.000010	0.024715	0.003969	minkowski
52	0.001024	0.000010	0.040591	0.006409	minkowski
53	0.002613	0.001956	0.028732	0.002469	minkowski
54	0.001019	0.000004	0.039025	0.005100	minkowski
55	0.002622	0.001958	0.027121	0.002471	minkowski
56	0.001019	0.000010	0.038191	0.005540	minkowski
57	0.001841	0.001591	0.032704	0.003506	minkowski
58	0.001018	0.000005	0.040621	0.005032	minkowski
59	0.001033	0.000002	0.034304	0.004906	minkowski
60	0.001020	0.000009	0.039745	0.004891	minkowski
61	0.001033	0.000011	0.034367	0.008794	minkowski
62	0.001029	0.000010	0.040555	0.004832	minkowski
63	0.001029	0.000011	0.033544	0.003519	minkowski
64	0.001024	0.000009	0.041400	0.002054	minkowski
65	0.001024	0.000002	0.031913	0.004371	minkowski

	param_n_neighbors	param_weights \
0	1	uniform
1	1	distance
2	3	uniform
3	3	distance
4	5	uniform
5	5	distance
6	7	uniform
7	7	distance

8	9	uniform
9	9	distance
10	11	uniform
11	11	distance
12	13	uniform
13	13	distance
14	15	uniform
15	15	distance
16	17	uniform
17	17	distance
18	19	uniform
19	19	distance
20	21	uniform
21	21	distance
22	1	uniform
23	1	distance
24	3	uniform
25	3	distance
26	5	uniform
27	5	distance
28	7	uniform
29	7	distance
30	9	uniform
31	9	distance
32	11	uniform
33	11	distance
34	13	uniform
35	13	distance
36	15	uniform
37	15	distance
38	17	uniform
39	17	distance
40	19	uniform
41	19	distance
42	21	uniform
43	21	distance
44	1	uniform
45	1	distance
46	3	uniform
47	3	distance
48	5	uniform
49	5	distance
50	7	uniform
51	7	distance
52	9	uniform
53	9	distance
54	11	uniform

```
55                  11      distance
56                  13       uniform
57                  13      distance
58                  15       uniform
59                  15      distance
60                  17       uniform
61                  17      distance
62                  19       uniform
63                  19      distance
64                  21       uniform
65                  21      distance

                                                 params  split0_test_score  \
0   {'metric': 'manhattan', 'n_neighbors': 1, 'wei…           0.986301
1   {'metric': 'manhattan', 'n_neighbors': 1, 'wei…           0.986301
2   {'metric': 'manhattan', 'n_neighbors': 3, 'wei…           0.986301
3   {'metric': 'manhattan', 'n_neighbors': 3, 'wei…           0.986301
4   {'metric': 'manhattan', 'n_neighbors': 5, 'wei…           0.981735
5   {'metric': 'manhattan', 'n_neighbors': 5, 'wei…           0.981735
6   {'metric': 'manhattan', 'n_neighbors': 7, 'wei…           0.972603
7   {'metric': 'manhattan', 'n_neighbors': 7, 'wei…           0.972603
8   {'metric': 'manhattan', 'n_neighbors': 9, 'wei…           0.972603
9   {'metric': 'manhattan', 'n_neighbors': 9, 'wei…           0.972603
10  {'metric': 'manhattan', 'n_neighbors': 11, 'we…           0.977169
11  {'metric': 'manhattan', 'n_neighbors': 11, 'we…           0.977169
12  {'metric': 'manhattan', 'n_neighbors': 13, 'we…           0.968037
13  {'metric': 'manhattan', 'n_neighbors': 13, 'we…           0.968037
14  {'metric': 'manhattan', 'n_neighbors': 15, 'we…           0.972603
15  {'metric': 'manhattan', 'n_neighbors': 15, 'we…           0.972603
16  {'metric': 'manhattan', 'n_neighbors': 17, 'we…           0.972603
17  {'metric': 'manhattan', 'n_neighbors': 17, 'we…           0.972603
18  {'metric': 'manhattan', 'n_neighbors': 19, 'we…           0.968037
19  {'metric': 'manhattan', 'n_neighbors': 19, 'we…           0.968037
20  {'metric': 'manhattan', 'n_neighbors': 21, 'we…           0.963470
21  {'metric': 'manhattan', 'n_neighbors': 21, 'we…           0.968037
22  {'metric': 'euclidean', 'n_neighbors': 1, 'wei…           0.990868
23  {'metric': 'euclidean', 'n_neighbors': 1, 'wei…           0.990868
24  {'metric': 'euclidean', 'n_neighbors': 3, 'wei…           0.990868
25  {'metric': 'euclidean', 'n_neighbors': 3, 'wei…           0.990868
26  {'metric': 'euclidean', 'n_neighbors': 5, 'wei…           0.990868
27  {'metric': 'euclidean', 'n_neighbors': 5, 'wei…           0.990868
28  {'metric': 'euclidean', 'n_neighbors': 7, 'wei…           0.968037
29  {'metric': 'euclidean', 'n_neighbors': 7, 'wei…           0.968037
30  {'metric': 'euclidean', 'n_neighbors': 9, 'wei…           0.968037
31  {'metric': 'euclidean', 'n_neighbors': 9, 'wei…           0.968037
32  {'metric': 'euclidean', 'n_neighbors': 11, 'we…           0.972603
33  {'metric': 'euclidean', 'n_neighbors': 11, 'we…           0.972603
```

```
34  {'metric': 'euclidean', 'n_neighbors': 13, 'we…           0.968037
35  {'metric': 'euclidean', 'n_neighbors': 13, 'we…           0.968037
36  {'metric': 'euclidean', 'n_neighbors': 15, 'we…           0.968037
37  {'metric': 'euclidean', 'n_neighbors': 15, 'we…           0.968037
38  {'metric': 'euclidean', 'n_neighbors': 17, 'we…           0.963470
39  {'metric': 'euclidean', 'n_neighbors': 17, 'we…           0.963470
40  {'metric': 'euclidean', 'n_neighbors': 19, 'we…           0.963470
41  {'metric': 'euclidean', 'n_neighbors': 19, 'we…           0.963470
42  {'metric': 'euclidean', 'n_neighbors': 21, 'we…           0.972603
43  {'metric': 'euclidean', 'n_neighbors': 21, 'we…           0.972603
44  {'metric': 'minkowski', 'n_neighbors': 1, 'wei…           0.990868
45  {'metric': 'minkowski', 'n_neighbors': 1, 'wei…           0.990868
46  {'metric': 'minkowski', 'n_neighbors': 3, 'wei…           0.990868
47  {'metric': 'minkowski', 'n_neighbors': 3, 'wei…           0.990868
48  {'metric': 'minkowski', 'n_neighbors': 5, 'wei…           0.990868
49  {'metric': 'minkowski', 'n_neighbors': 5, 'wei…           0.990868
50  {'metric': 'minkowski', 'n_neighbors': 7, 'wei…           0.968037
51  {'metric': 'minkowski', 'n_neighbors': 7, 'wei…           0.968037
52  {'metric': 'minkowski', 'n_neighbors': 9, 'wei…           0.968037
53  {'metric': 'minkowski', 'n_neighbors': 9, 'wei…           0.968037
54  {'metric': 'minkowski', 'n_neighbors': 11, 'we…           0.972603
55  {'metric': 'minkowski', 'n_neighbors': 11, 'we…           0.972603
56  {'metric': 'minkowski', 'n_neighbors': 13, 'we…           0.968037
57  {'metric': 'minkowski', 'n_neighbors': 13, 'we…           0.968037
58  {'metric': 'minkowski', 'n_neighbors': 15, 'we…           0.968037
59  {'metric': 'minkowski', 'n_neighbors': 15, 'we…           0.968037
60  {'metric': 'minkowski', 'n_neighbors': 17, 'we…           0.963470
61  {'metric': 'minkowski', 'n_neighbors': 17, 'we…           0.963470
62  {'metric': 'minkowski', 'n_neighbors': 19, 'we…           0.963470
63  {'metric': 'minkowski', 'n_neighbors': 19, 'we…           0.963470
64  {'metric': 'minkowski', 'n_neighbors': 21, 'we…           0.972603
65  {'metric': 'minkowski', 'n_neighbors': 21, 'we…           0.972603

    split1_test_score  split2_test_score  split3_test_score  \
0            0.995434           0.990826           0.986239
1            0.995434           0.990826           0.986239
2            0.995434           0.981651           0.967890
3            0.995434           0.981651           0.967890
4            0.986301           0.981651           0.963303
5            0.986301           0.981651           0.963303
6            0.986301           0.981651           0.963303
7            0.986301           0.981651           0.963303
8            0.981735           0.981651           0.963303
9            0.981735           0.981651           0.963303
10           0.977169           0.986239           0.963303
11           0.977169           0.986239           0.963303
12           0.977169           0.986239           0.958716
```

13	0.977169	0.986239	0.958716
14	0.977169	0.981651	0.954128
15	0.977169	0.986239	0.954128
16	0.977169	0.981651	0.954128
17	0.977169	0.986239	0.954128
18	0.972603	0.972477	0.958716
19	0.977169	0.972477	0.958716
20	0.963470	0.977064	0.954128
21	0.972603	0.977064	0.954128
22	0.995434	0.990826	0.986239
23	0.995434	0.990826	0.986239
24	0.990868	0.977064	0.977064
25	0.990868	0.977064	0.977064
26	0.986301	0.977064	0.967890
27	0.986301	0.977064	0.967890
28	0.981735	0.977064	0.967890
29	0.981735	0.977064	0.967890
30	0.981735	0.972477	0.963303
31	0.981735	0.972477	0.963303
32	0.977169	0.977064	0.963303
33	0.977169	0.977064	0.963303
34	0.972603	0.967890	0.972477
35	0.972603	0.967890	0.972477
36	0.963470	0.967890	0.972477
37	0.963470	0.972477	0.972477
38	0.968037	0.967890	0.963303
39	0.968037	0.967890	0.963303
40	0.968037	0.967890	0.963303
41	0.968037	0.967890	0.963303
42	0.968037	0.967890	0.958716
43	0.968037	0.967890	0.958716
44	0.995434	0.990826	0.986239
45	0.995434	0.990826	0.986239
46	0.990868	0.977064	0.977064
47	0.990868	0.977064	0.977064
48	0.986301	0.977064	0.967890
49	0.986301	0.977064	0.967890
50	0.981735	0.977064	0.967890
51	0.981735	0.977064	0.967890
52	0.981735	0.972477	0.963303
53	0.981735	0.972477	0.963303
54	0.977169	0.977064	0.963303
55	0.977169	0.977064	0.963303
56	0.972603	0.967890	0.972477
57	0.972603	0.967890	0.972477
58	0.963470	0.967890	0.972477
59	0.963470	0.972477	0.972477

60	0.968037	0.967890	0.963303
61	0.968037	0.967890	0.963303
62	0.968037	0.967890	0.963303
63	0.968037	0.967890	0.963303
64	0.968037	0.967890	0.958716
65	0.968037	0.967890	0.958716

	split4_test_score	mean_test_score	std_test_score	rank_test_score
0	0.990826	0.989925	0.003426	5
1	0.990826	0.989925	0.003426	5
2	0.981651	0.982586	0.008906	11
3	0.981651	0.982586	0.008906	11
4	0.981651	0.978928	0.008015	17
5	0.981651	0.978928	0.008015	17
6	0.977064	0.976184	0.007897	19
7	0.977064	0.976184	0.007897	19
8	0.977064	0.975271	0.006871	25
9	0.977064	0.975271	0.006871	25
10	0.967890	0.974354	0.008013	27
11	0.967890	0.974354	0.008013	27
12	0.963303	0.970692	0.009887	45
13	0.963303	0.970692	0.009887	45
14	0.972477	0.971606	0.009371	40
15	0.972477	0.972523	0.010470	38
16	0.977064	0.972523	0.009632	38
17	0.977064	0.973441	0.010625	29
18	0.977064	0.969779	0.006225	48
19	0.977064	0.970692	0.006874	45
20	0.972477	0.966122	0.007975	66
21	0.972477	0.968862	0.007901	51
22	0.990826	0.990838	0.002908	1
23	0.990826	0.990838	0.002908	1
24	0.981651	0.983503	0.006242	7
25	0.981651	0.983503	0.006242	7
26	0.986239	0.981672	0.008222	13
27	0.986239	0.981672	0.008222	13
28	0.981651	0.975275	0.006205	21
29	0.981651	0.975275	0.006205	21
30	0.981651	0.973441	0.007337	29
31	0.981651	0.973441	0.007337	29
32	0.977064	0.973441	0.005360	29
33	0.977064	0.973441	0.005360	29
34	0.972477	0.970697	0.002233	41
35	0.972477	0.970697	0.002233	41
36	0.967890	0.967953	0.002849	52
37	0.967890	0.968870	0.003371	49
38	0.972477	0.967035	0.003405	54

39	0.972477	0.967035	0.003405	54
40	0.972477	0.967035	0.003405	54
41	0.972477	0.967035	0.003405	54
42	0.967890	0.967027	0.004532	62
43	0.967890	0.967027	0.004532	62
44	0.990826	0.990838	0.002908	1
45	0.990826	0.990838	0.002908	1
46	0.981651	0.983503	0.006242	7
47	0.981651	0.983503	0.006242	7
48	0.986239	0.981672	0.008222	13
49	0.986239	0.981672	0.008222	13
50	0.981651	0.975275	0.006205	21
51	0.981651	0.975275	0.006205	21
52	0.981651	0.973441	0.007337	29
53	0.981651	0.973441	0.007337	29
54	0.977064	0.973441	0.005360	29
55	0.977064	0.973441	0.005360	29
56	0.972477	0.970697	0.002233	41
57	0.972477	0.970697	0.002233	41
58	0.967890	0.967953	0.002849	52
59	0.967890	0.968870	0.003371	49
60	0.972477	0.967035	0.003405	54
61	0.972477	0.967035	0.003405	54
62	0.972477	0.967035	0.003405	54
63	0.972477	0.967035	0.003405	54
64	0.967890	0.967027	0.004532	62
65	0.967890	0.967027	0.004532	62

After training the model, we find the best parameters and the score of that parameters. In this case,

[32]:
```
print("\n The best score across ALL searched params:\n",gs_knn.best_score_)
print("\n The best parameters across ALL searched params:\n",gs_knn.
 ↪best_params_)
```

```
 The best score across ALL searched params:
 0.9908382556239788

 The best parameters across ALL searched params:
 {'metric': 'euclidean', 'n_neighbors': 1, 'weights': 'uniform'}
```

Now that we have the best parameters we can move on to train the model and check the efficiency of it,

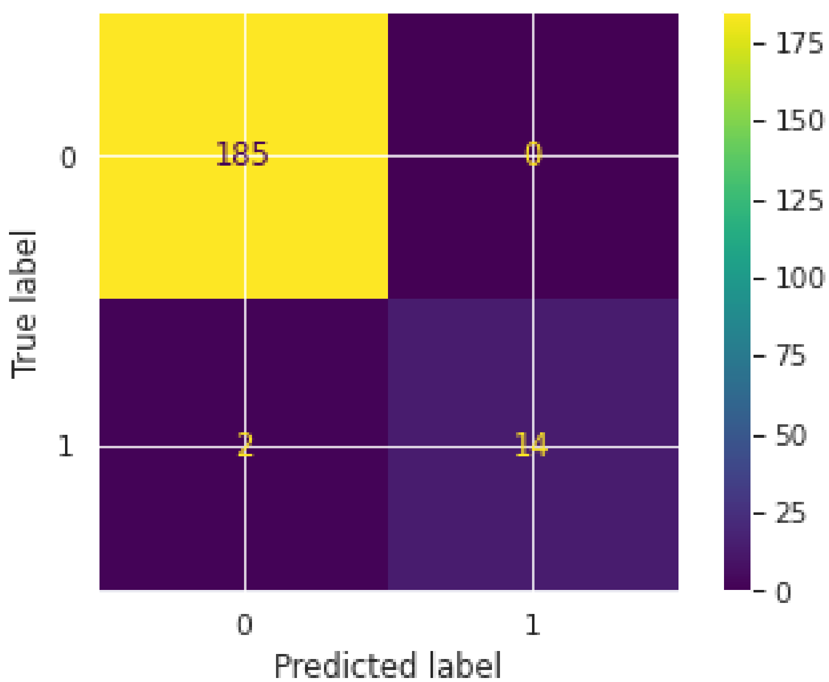
Testing the model and checking metric score:

[33]:
```python
y_pred_knn = gs_knn.best_estimator_.predict(x_test)
cfx_knn = confusion_matrix(y_test, y_pred_knn)
f1_knn =  f1_score(y_test, y_pred_knn)
accu_knn = accuracy_score(y_test, y_pred_knn)
prec_knn = precision_score(y_test, y_pred_knn)
rec_knn = recall_score(y_test, y_pred_knn)
s_knn = pd.Series({'Model': 'K Nearest Neighbors',
               'F1 Score': f1_knn,
               'Accuracy': accu_knn,
               'Precision': prec_knn,
               'Recall' : rec_knn})
s_knn = pd.DataFrame(s_knn)
print(s_knn)

ConfusionMatrixDisplay(cfx_knn).plot()
```

```
                             0
Model      K Nearest Neighbors
F1 Score              0.933333
Accuracy               0.99005
Precision                  1.0
Recall                   0.875
```

[33]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7fb5cdfca310>



From the above values we can tell that our model has been trained with a ***F1 score of 93.33%, Accuracy of 99.01%, Precision of 100% and Recall of 87.5%***.

5 Conclusion

5.0.1 Perfromance Analysis of Models

Now that we have Trained and Tested all of our models and calculated all the necessary metric score for each of them, we can move to analyze the results.

[34]:
```
score = pd.concat([s_svm, s_dtc, s_knn], axis=1, ignore_index=True)
score
```

[34]:
```
                                0                         1  \
Model      Support Vector Machines  Decision Tree Classifer
F1 Score                  0.896552                 0.903226
Accuracy                  0.985075                 0.985075
Precision                      1.0                 0.933333
Recall                      0.8125                    0.875

                          2
Model      K Nearest Neighbors
F1 Score              0.933333
Accuracy               0.99005
Precision                  1.0
Recall                   0.875
```

Combining all the results in a single Data Frame we can see that **K Nearest Neighbors** is, relatively, the more efficient model in predicting the legendary status from the Pokemon Dataset.

6 Reference

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

[ ]: