Clasificador de Pokémon

Diego Baratto Valdivia

Leonor Cuesta Molinero

Grado de Desarrollo de Videojuegos (UCM)

Curso 2019 – 2020

dbaratto@ucm.es leonorcu@ucm.es

Contenidos

En este documento se especifica el progreso de desarrollo de un sistema de clasificación de pokémon, así como la comparación entre las diferentes técnicas utilizadas para ello y las conclusiones obtenidas a partir de los resultados obtenidos. Se ha hecho uso de las prácticas realizadas, así como de las técnicas estudiadas en clase, entre ellas regresión logística, Support Vector Machines y redes neuronales. Además, se han utilizado nuevas técnicas con el apoyo de los algoritmos de Keras para la realización de redes neuronales y de Boruta, cuyo objetivo es averiguar las características más significativas del conjunto de datos disponible.

Se proporcionan diferentes conjuntos de imágenes, obtenidas en las diferentes pruebas realizadas, así como los diferentes resultados para facilitar la comparación.

La clasificación de pokémon se ha realizado siguiendo dos criterios: agrupar aquellos pokémon legendarios y no legendarios y averiguar el tipo de cada uno de ellos a partir de diferentes características, por ejemplo, ataque, defensa, felicidad base o velocidad.

1. *SUPPORT VECTOR MACHINE* (SVM)

1.1. Descripción del Proyecto

Haciendo uso de las SVM, se ha creado un clasificador de pokémon legendarios en función de las características elegidas por el usuario.

Se ha tomado como referencia la práctica realizada en clase sobre SVM, modificando aquellos fragmentos de códigos necesarios para lograr el objetivo, por ejemplo, dividir los datos en tres grupos diferentes (entrenamiento, validación y testeo) para realizar *cross-validation* (evitando así el sobreajuste) y elección de los parámetros C y sigma.

Se ha añadido también soporte de introducción de datos del usuario, es decir, tras el entrenamiento de la SVM, es posible testear la solución incluyendo por consola nuevos pokémon que no se encuentren en el grupo de los datos utilizados para entrenar.

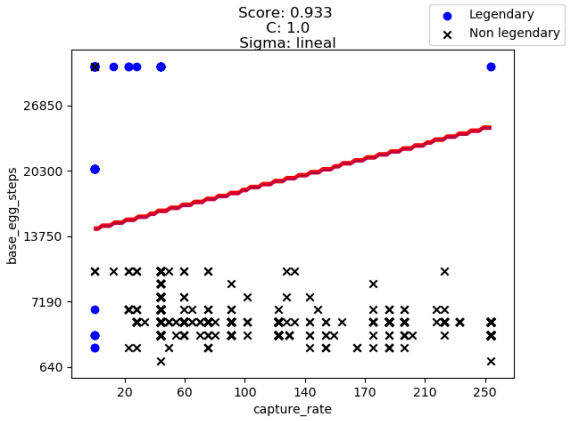
Además de la clasificación anterior, se ha tratado de realizar clasificación multi-clase de los diferentes tipos de pokémon, entre ellos agua, tierra, roca, entre otros, en base a las diferentes características proporcionadas en el conjunto de datos. Se ha hecho uso de la misma metodología que en el caso anterior, añadiendo c*ross-validation*, elección de los mejores parámetros C y sigma y adición de nuevas características polinomiales. Debido a las complicaciones a la hora de elegir las características más representativas para la clasificación, se ha utilizado el paquete de Python denominado como Boruta, el cual ha ayudado a discriminar aquellos datos más importantes a la hora de la clasificación, mediante los *Random Forest* (se establecen varios árboles de decisión y se elige el que mejor resultado ofrezca).

1.2. Resultados Obtenidos

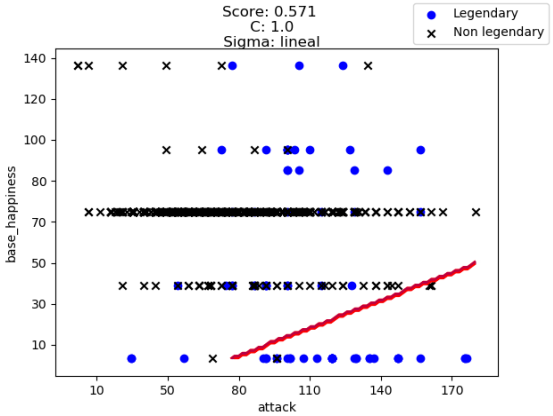
A continuación se incluyen los diferentes resultados obtenidos, así como imágenes de apoyo y explicaciones al pie de las mismas.

1.2.1. Clasificador de Legendarios

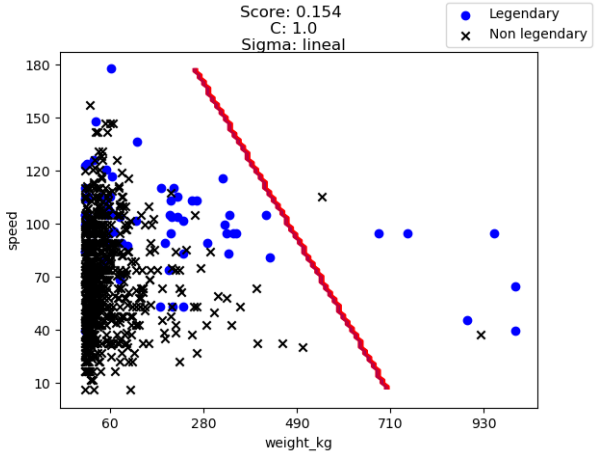
De las primeras pruebas que se realizaron fue con kernel lineal, útil si el número de atributos (n) es grande comparado con el número de ejemplos (m):



Kernel lineal entrenado con los atributos *base egg steps* y *capture rate*. En este caso, al ser atributos diferenciados entre ellos, el kernel lineal funciona de manera aceptable.



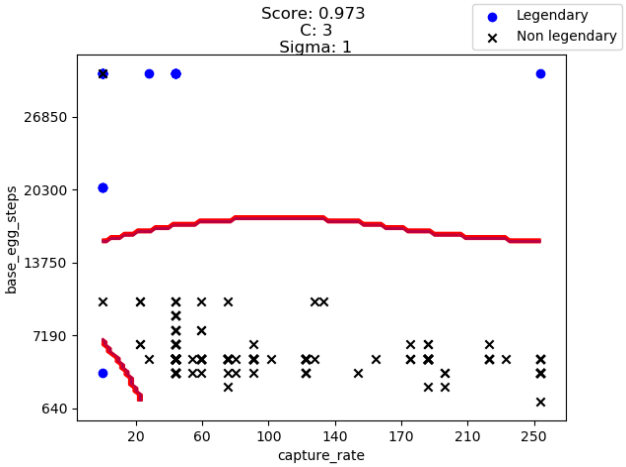
Kernel lineal entrenado con los atributos *base happiness* y *attack*. En este caso, al ser atributos linealmente no diferenciables entre ellos, el kernel lineal no funciona de manera aceptable.



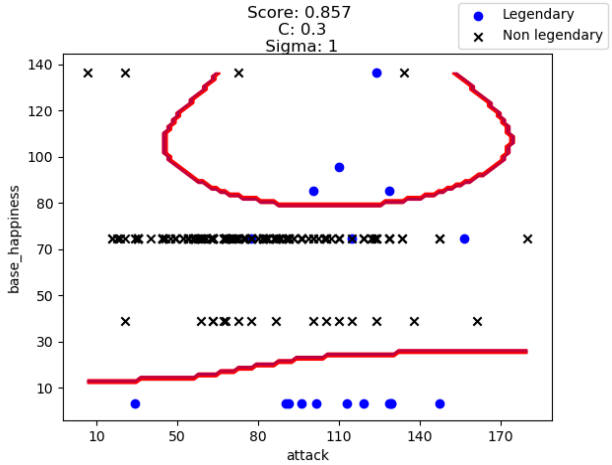
Kernel lineal entrenado con los atributos *speed* y *weight_kg*. En este caso, al ser atributos linealmente no diferenciables, el kernel lineal no funciona de manera aceptable.

En todos los casos con más de un atributo, el kernel lineal no funciona (*score* entre 0.4 y 0.6) debido a que el número de casos de entrenamiento es mayor que el número de atributos.

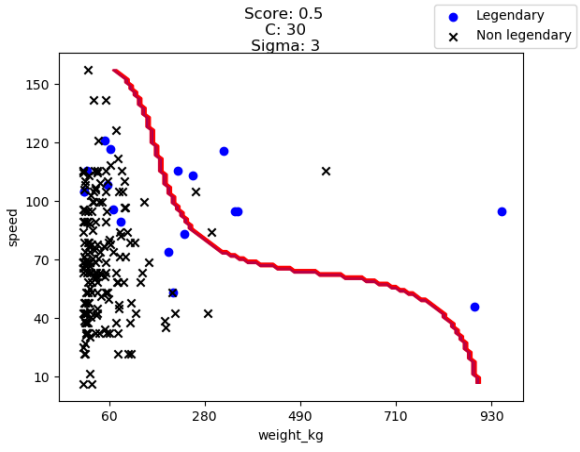
Tras estas pruebas, se decidió usar el kernel Gaussiano, apto para un número de atributos pequeño y un número de casos de ejemplo intermedio.



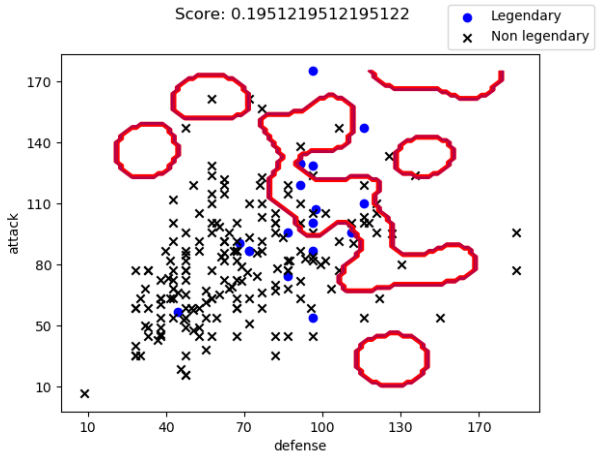
Kernel Gaussiano entrenado con los atributos *base egg steps* y *capture rate*.



Kernel Gaussiano entrenado con los atributos *base happiness* y *attack*.

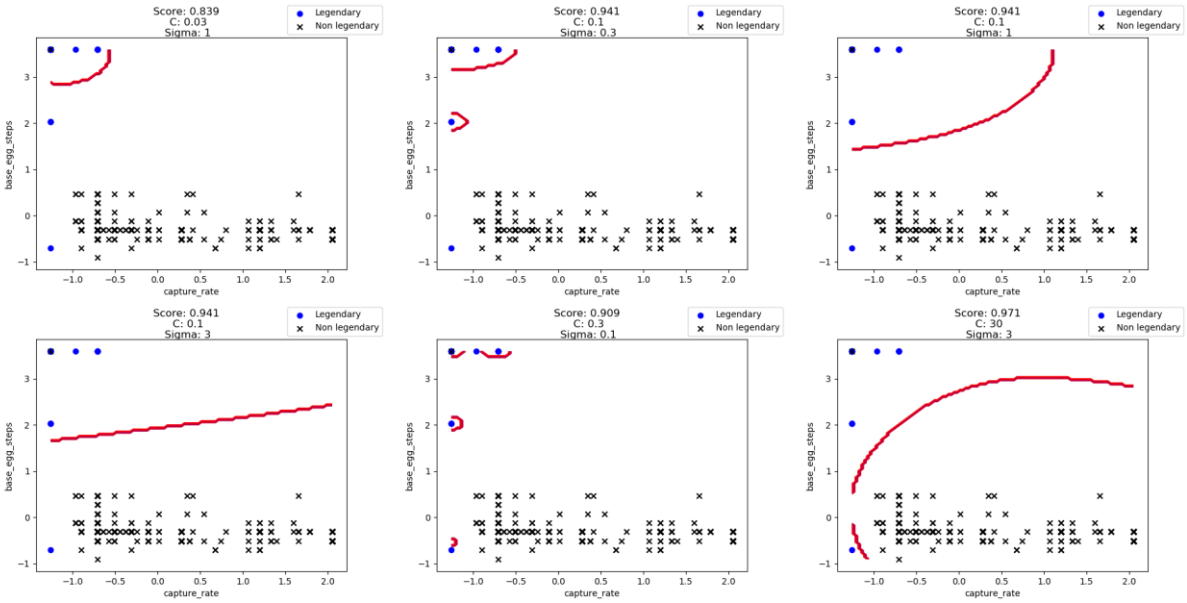


Kernel Gaussiano entrenado con los atributos *speed* y *weight_kg*.



Kernel Gaussiano entrenado con los atributos *attack* y *defense*.

Gracias a la visualización de estas gráficas, se puede observar cuales son los atributos que mejor definen a los pokémon legendarios y cuales ensucian el aprendizaje automático de la SVM. A continuación, se muestra la elección de los parámetros C y sigma con los atributos que mejor definen a los legendarios, *capture rate* y *base egg steps*:

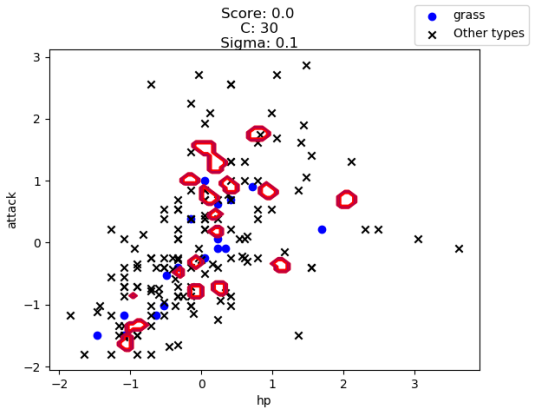


Se puede observar como varia la frontera de decisión de la SVM dependiendo de los valores que reciba C y sigma. Se aprecia de manera detallada el sobreajuste que se da en, por ejemplo, con C = 0.3 y sigma = 0.1.

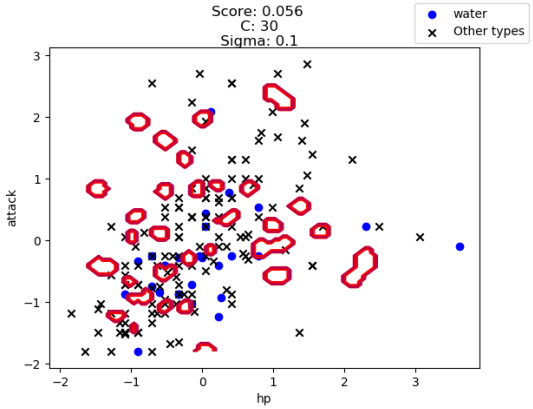
1.2.1. Clasificador de tipos

A pesar de haber hecho uso de Boruta para elegir las características más importantes para la clasificación, no se consigue una media armónica mayor a 0.4 en cada tipo. Dentro de los mismos, se observa mucha diferencia de precisión en la predicción entre ellos.

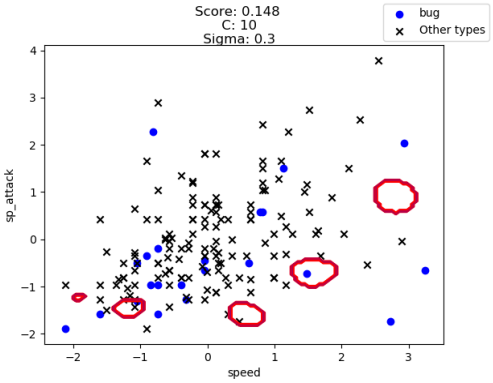
A continuación se muestran las gráficas obtenidas más significativas con diferentes atributos y tipos:



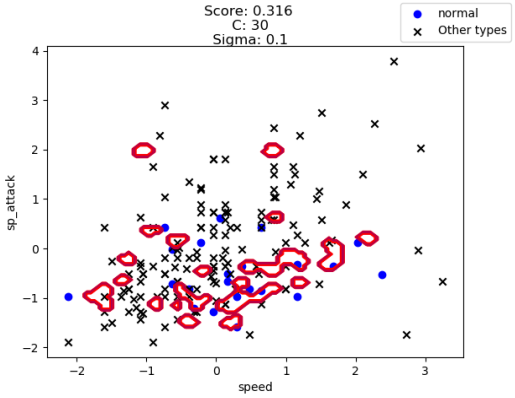
Se puede observar el intento de clasificación del tipo planta a partir de las características ataque y vida. Viendo la gráfica y los resultados obtenidos, se puede deducir que no son atributos que se puedan usar para separar por tipo.



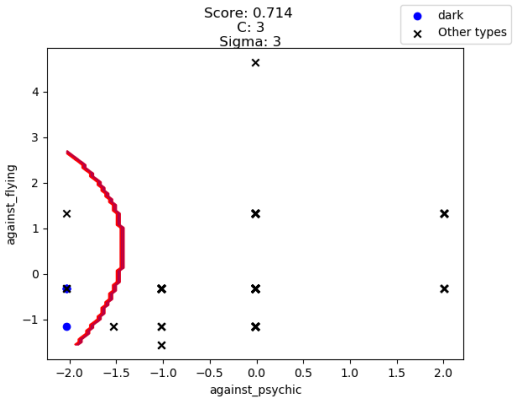
Mismo resultado que la gráfica anterior para el tipo agua.



Otro intento de clasificación del tipo bicho, esta vez con las características de ataque especial y velocidad.



Misma clasificación que la gráfica anterior para el tipo normal. A partir de los resultados obtenidos dependiendo de las características referidas, se deduce que algunos atributos clasifican mejor a ciertos tipos que a otros.



Clasificación de tipo siniestro a partir de dos de sus debilidades. Es una forma muy precisa de clasificación, pero no es la más acertada ya que prácticamente se le ofrece una descripción detallada del tipo al que pertenece.

1.3. Conclusiones:

1.3.1. Clasificador de Legendarios

La técnica empleada con las SVM funciona de manera precisa y evita el *overfitting* / underfitting correctamente. Tras proporcionarle a los resultados de entrenamiento diferentes pokémon nunca vistos por ella, se logran unos resultados acertados y precisos, además de usar un conjunto de testeo como se ha dicho anteriormente para obtener el *score* del mismo (media armónica), logrando una media de 0.85 en dicho *score*. A continuación, se proporciona una captura de pantalla donde se consulta los datos (*attack*, *defense*, *hp*, *sp_attack*, *sp_defense*, *speed*, *capture_rate*) de cuatro pokémon diferentes de la octava generación (Zacian, Snom, Zamazenta, Flapple), la cual no se encuentra en el *dataset*, siendo el primero y el tercero legendarios, mientras que el segundo y el cuarto no.

```
Score con los ejemplos de testing: 0.8823529411764706
Gimme stats: 130 115 92 80 115 148 10
Is your pokemon legendary?: True

Gimme stats: 25 35 30 45 30 20 190
Is your pokemon legendary?: False

Gimme stats: 130 145 92 80 145 128 10
Is your pokemon legendary?: True

Gimme stats: 110 80 70 95 60 70 45
Is your pokemon legendary?: False
```

1.3.2. Clasificador de tipos

Como ya ha sido mencionado anteriormente, se ha conseguido discriminar aquellas características que mejor clasifican por tipo de pokémon. A pesar de ello, no se consiguen resultados equilibrados entre ejecuciones y con una gran diferencia de precisión obtenida en el clasificador de Legendarios.

Con todo ello, se puede concluir que ciertos tipos de pokémon se pueden clasificar de manera más o menos óptima a partir de, por ejemplo, el ataque especial, como el tipo psíquico, pero no es una generalización en todos los tipos. Incluso ciertos tipos son muy difíciles de clasificar siguiendo el algoritmo desarrollado ya que en los ejemplos de entrenamiento o bien hay muy pocos pokémon de ese tipo, por ejemplo, de tipo dragón, o bien no existe ninguno con ese tipo primario, refiriéndose al caso de tipo volador. Por lo tanto, teniendo en cuenta solamente el *dataset* elegido, se obtienen resultados con falta de precisión y credibilidad. Para mejorar dichos resultados, habría que contar con el apoyo de otro *dataset*, como cualquiera que contenga imágenes de los pokémon, facilitando la tarea ya que se cuenta con datos de la paleta de colores y de la forma, atributos que son más significativos a la hora de clasificarlos por tipo.

2. Regresión logística

2.1. Descripción del Proyecto

Con la teoría explicada en clase sobre regresión lineal y logística, se ha aplicado este sistema para el mismo clasificador de legendarios y de tipos descrito.

En este caso, empleando la práctica de regresión logística como base, se ha creado una similar que, recibiendo dos atributos de entre los recogidos en el data set, clasifique los pokémon entre legendario y no legendario y los separe por tipos.

2.2. Resultados Obtenidos

2.2.1. Clasificador de Legendarios

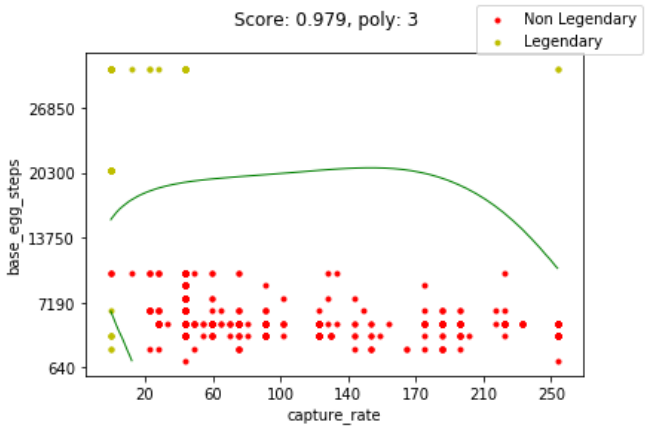
En principio, la clasificación tenía un porcentaje de acierto muy alto (de entorno el 95%), dado que solo tenía en cuenta los clasificados correctamente, no la proporción de legendarios y no legendarios en el grupo evaluado. Esto se hizo mucho más evidente al probar con grupos de menor tamaño con proporciones más altas de legendarios.

Se soluciono parcialmente utilizando score en lugar de precisión como medida de comparación entre datos, pero devolvía resultados distintos de lo esperado al observarse que, en la gran mayoría de casos, la regresión lineal (grado 1) era la mejor opción para definir la separación entre legendarios y no legendarios cuando a simple vista se podía deducir una curva que los separaría con menor error.

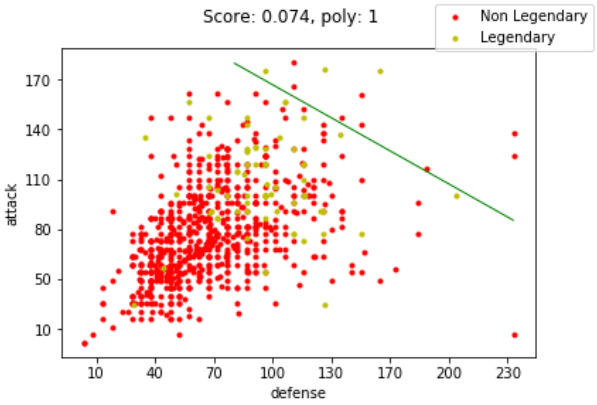
Esto sucedía principalmente al no estar normalizados los datos y dar overflow (y posterior división entre 0) al ejecutar la minimización con TNC, definiendo todos los valores de theta como 0.

Normalizar los datos hacía que comenzasen a aparecer diversas gráficas con el mismo grado del polinomio, por lo que, en función de los datos aportados, hay posibilidad de caer en mínimos locales. Por ello se repite el proceso un número definido de veces.

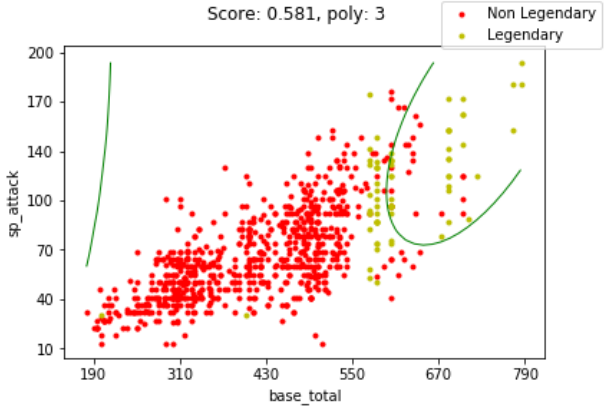
Por último, se hace una búsqueda del mejor polinomio, comparando sus scores hasta que comienzan a descender, punto en el cual se cogen los valores anteriores para su posterior pintado.



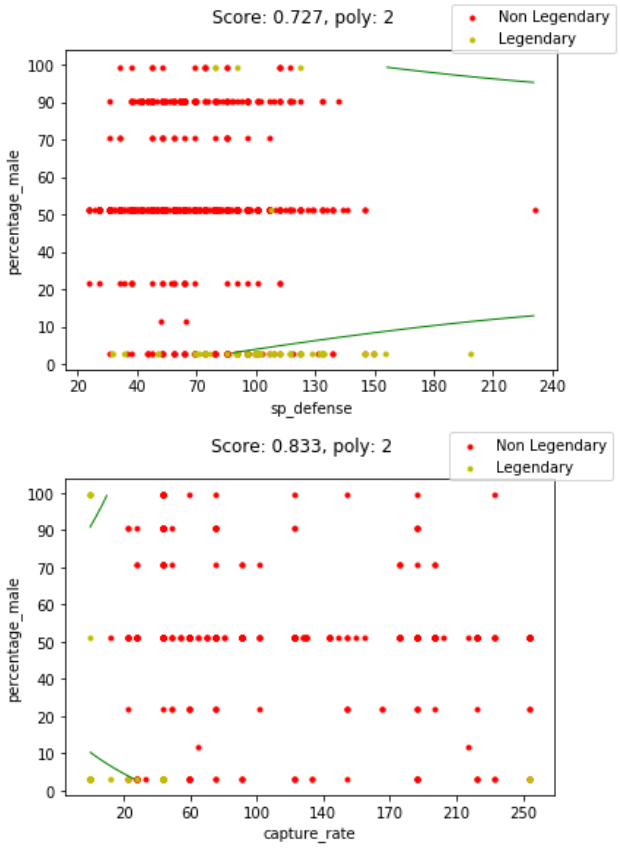
Como se ha podido observar durante todo el proyecto, *base_egg_steps* y *capture_rate* aportan los mejores resultados.



Otros atributos que de forma intuitiva parecen ser determinantes en la clasificación propuesta, no aportan separación clara entre pokémon.



Otras combinaciones aportan una gráfica que aparentemente es mejor que la resultante, pero debido a la densidad de no legendarios, hace descender el score total.



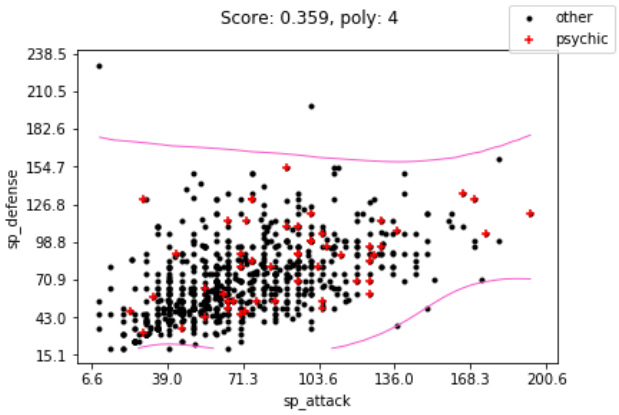
Algunas combinaciones poco esperadas dan resultados por encima de la media, aunque las mejores siguen incluyendo o *capture_rate* o *base_egg_steps*.

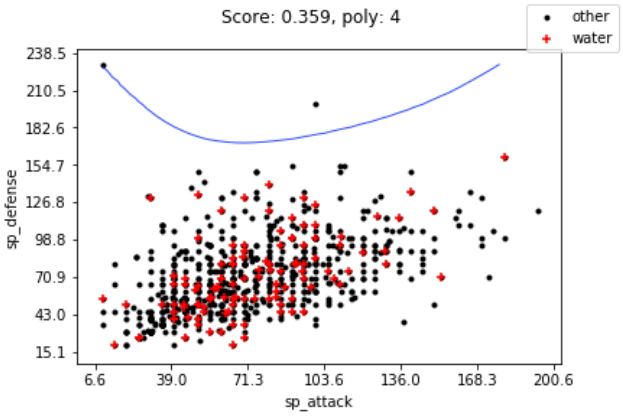
2.2.2. Clasificador de tipos

A partir de la versión creada para la clasificación de legendarios, se ha implementado este clasificador por tipos, manteniendo la normalización de los datos, repitiendo la ejecución un número finito de veces para evitar posibles mínimos locales y buscar el polinomio con el mejor score para el caso tratado.

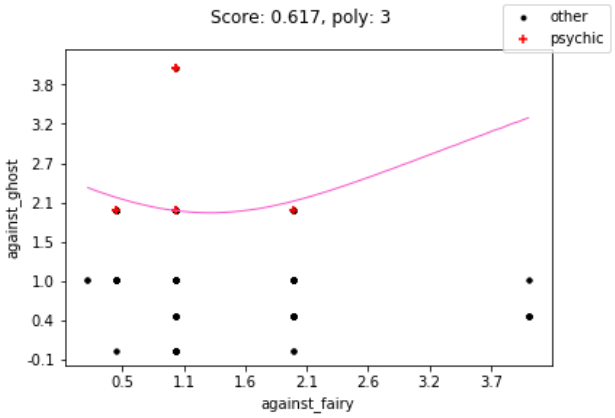
Esto devuelve una tupla con los valores de theta definidos para cada tipo en función de los atributos evaluados, por lo que se puede comprobar que ciertos parámetros pueden clasificar mejor determinados tipos.

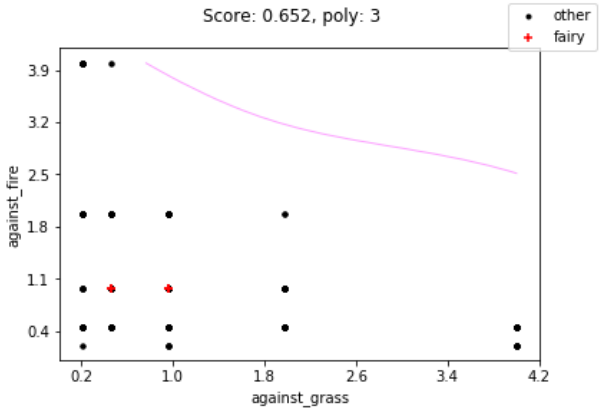
Se ha testeado repetidas veces con diversas combinaciones de atributos y tipos, no hallándose una relación clara determine el tipo de pokémon, pero se ha comprobado que ciertas características son más afines a unos tipos determinados.



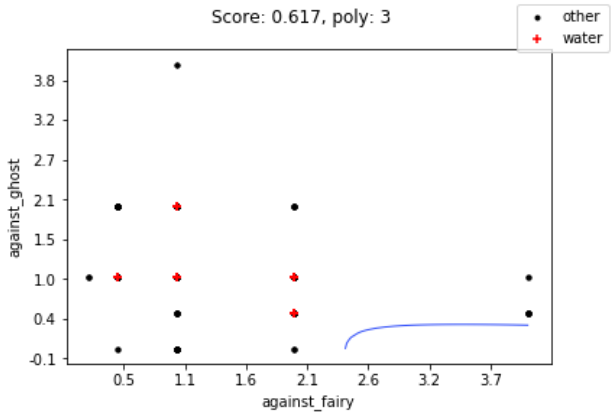


Como se puede comprobar, algunos se ajustan de mejor manera, pero siempre de una forma incorrecta en su mayor parte.

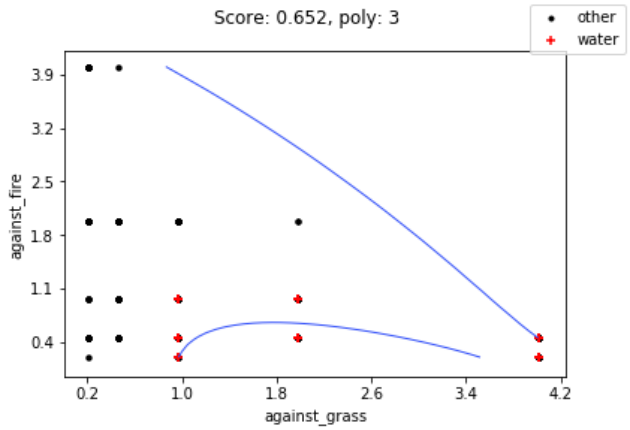




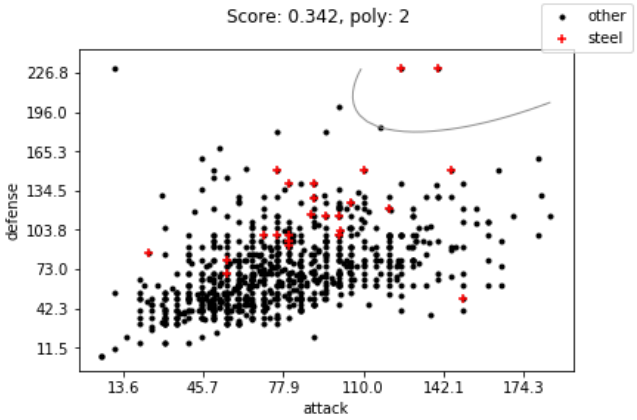
Al emplear los atributos con el nombre "*against_X*" tampoco asegura un buen resultado, ya que solo será relevante en los casos en los que los tipos utilizados y mostrados tengan una relación de tipos clara, ya que en este caso el score puede ser engañoso.

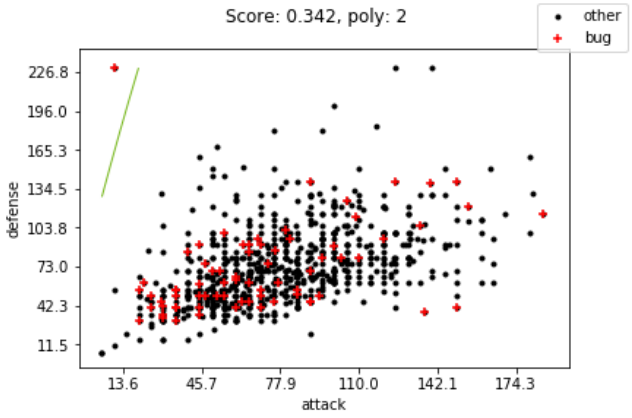


A pesar de tener un score superior al 0.5, esto no implica una clasificación que tiende a la correcta, dado que, en función de la elección del tipo mostrado, estos datos son relevantes o no.

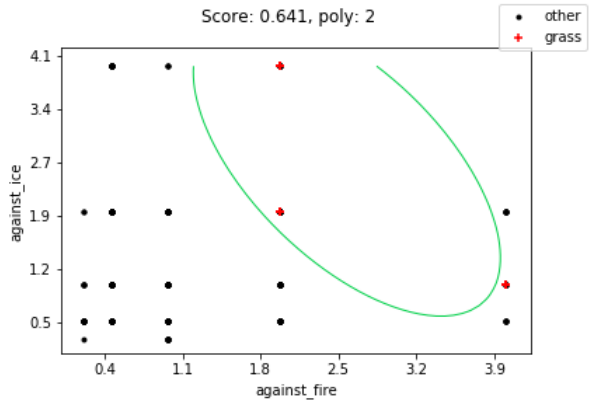


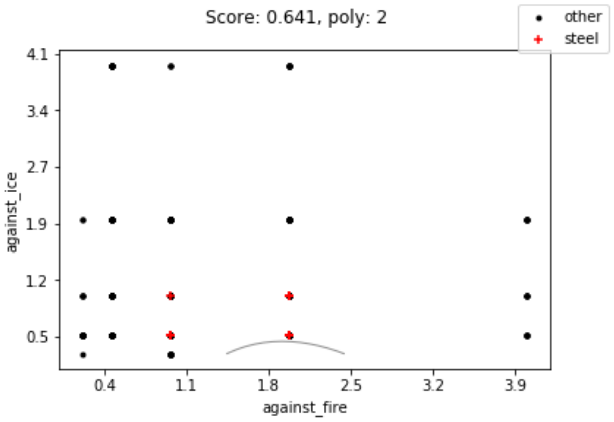
Algunos atributos definen mejor los límites de clasificación, pero sigue sin tener una base sólida para ello.



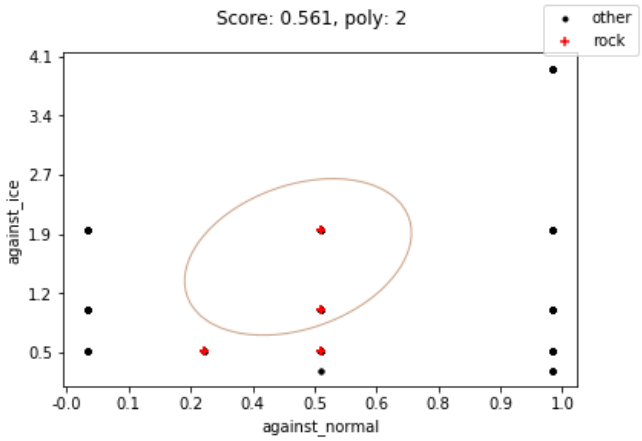


Al existir una densidad tan alta de pokémons de diversos tipos con estadísticas similares, no define correctamente el límite de decisión.





Con otros atributos que los separan de forma más ordenada también supone un problema, ya que la cantidad de pokémons de tipo distinto al elegido es muy superior a la del mostrado, ya que hay 18 tipos distintos con algunos de ellos en menor cantidad que otros.



Los pokémons de tipos compuestos ensucian los datos dado que su segundo tipo puede ser fuerte contra un tipo al que debería ser débil de forma primaria. Esto hace aun más complejo y difuso el límite de decisión.

2.3. Conclusiones

2.3.1. Clasificador de Legendarios

El clasificador de legendarios creado, como se ha mostrado, tiene un score de entre el 0.979 y 0.074 dependiendo de los atributos tomados de referencia, lo que hace que, hallando los atributos correctos, se puede clasificar con una seguridad muy alta de que su resultado sea correcto, sin embargo, depende demasiado de este factor. Se necesitaría ser capaz de utilizar más atributos para que no haya tantos puntos de diferencia.

Como en otras de las opciones testeadas en este proyecto, se ha podido apreciar la gran correlación que conllevan tanto ratio de captura como pasos para eclosionar huevo con ser o no u pokémon legendario. Pero otros elementos que se han podido observar tienen cierta relación, aunque no tan cercana como estos dos parámetros.

2.3.2. Clasificador de tipos

Como en otras técnicas empleadas, se obtienen resultados erróneos e imprecisos al utilizar las estadísticas que no definen las relaciones entre tipos (*attack, hp, etc.*), no hallándose ninguna correlación concluyente entre estos atributos y el tipo de los pokémon estudiados, teniendo una distribución bastante igualada entre ellos.

Por el contrario, al utilizar las relaciones de tipos (*against_X*), debido a que éstas están definidas en función del tipo, clasifican los pokémon de una forma bastante más certera, pero debida a la densidad de las distintas estadísticas de pokémon de todo tipo, sigue siendo muy complicado definir la barrera entre unos y otros.

Se concluye que, con las pruebas realizadas y los datos obtenidos, no es posible predecir el tipo de pokémon en función de sus características con seguridad.

3. Redes Neuronales

3.1. Descripción del Proyecto

Utilizando redes neuronales, hemos creado tanto un clasificador de legendarios en función de sus otras características como un predictor del tipo de Pokémon que es en función de su relación con otros tipos y características.

3.1.1. Clasificador de Legendarios

Tomando como referencia distintas características del conjunto [attack, base_egg_steps, base_happiness, base_total, capture_rate, defense, experience_growth, height_m, hp, percentage_male, sp_attack, sp_defense, weight_kg]

Se ha utilizado como referencia la práctica de laboratorio basada en redes neuronales, así como el módulo de Python "Keras" con "Tensorflow".

En este clasificador, entrena una red neuronal dividida en grupos de entrenamiento, validación y testeo, cogiendo ejemplos del grupo total de forma aleatoria y barajándose para evitar que los legendarios y no legendarios se agrupen al principio o final de los grupos.

Para tratar de asegurar el mejor *score* de entrenamiento, se repite el proceso un número establecido de veces y guarda el mejor.

3.1.2. Clasificador de tipos

Tomando como referencia los "stats de batalla" mencionados antes, no se ha encontrado una forma fiable de separar los pokémon por tipos, sin embargo, al igual que en SVM, al utilizar los atributos de "against_X" el tipo se clasifica con una seguridad alta.

Se ha utilizado la práctica de laboratorio de redes neuronales, al igual que en la clasificación por legendarios, así como Boruta para distinguir las características más relevantes ya que no conseguíamos una combinación que los clasificara correctamente.

3.2. Resultados Obtenidos

3.2.1. Clasificador de Legendarios

En las primeras pruebas se utilizó el grupo de pokémon completo, habiendo una cantidad de entorno 70 legendarios de un total de 802 pokémon, por lo que clasificaba a todos ellos como pokémon normales, que daba una precisión del 91% (que variaba ligeramente con los pesos aleatorios de theta, cayendo en mínimos locales) pero un score de 0.

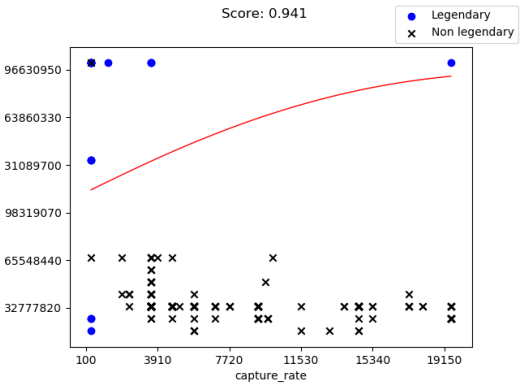
Como primera intención, se probaron algunos atributos por separado, destacando el "*capture_rate"* con una precisión constante del 97.253%, pero aun clasificaba gran parte de los legendarios como pokémon normales.

Decidimos calcular el score del proceso para que se tenga en cuenta el recall y la precisión, ya que solo la precisión aportaba resultados engañosos, así como separar en grupos de entrenamiento (1/4 de normales, ½ de legendarios), validación (1/2 normales, ¼ legendarios) y testeo (1/4 normales, ¼ legendarios). Esto dio mejores resultados en cuanto al score en los ejemplos de

entrenamiento (0.933), aunque bajaba en los ejemplos no entrenados (0.799). Seguimos cayendo en mínimos locales que enturbiaban los pesos de theta.

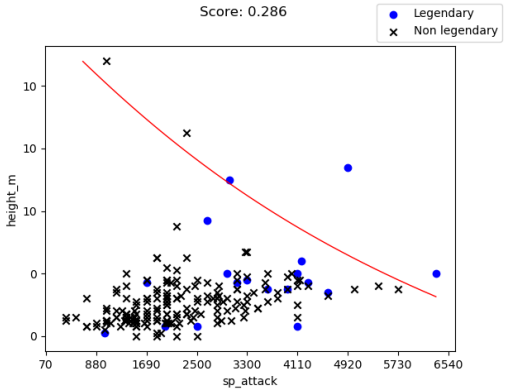
Determinamos que la forma de evitarlos era barajar los grupos para evitar la “condensación” de legendarios, añadido a ejecutar el proceso un número finito de veces y guardar el resultado mejor. Con ello, el score subió a 0.914 con todos los atributos, sorteando los mínimos locales en los que se caía anteriormente.

A partir de este momento, las pruebas se centraron en variar los atributos observados, la cantidad de intentos y el valor de lambda.



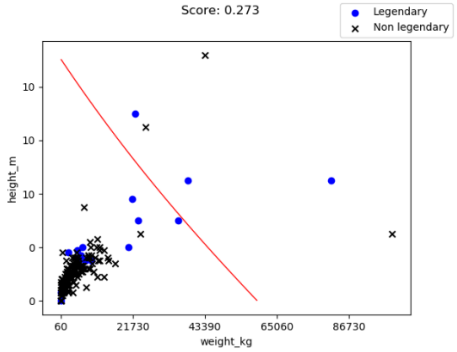
(Capture rate, Base egg steps) lambda = 1

Se puede ver que son unas características que separan claramente los legendarios y no legendarios, teniendo éstos un nivel muy “agrupado” de estas variables.



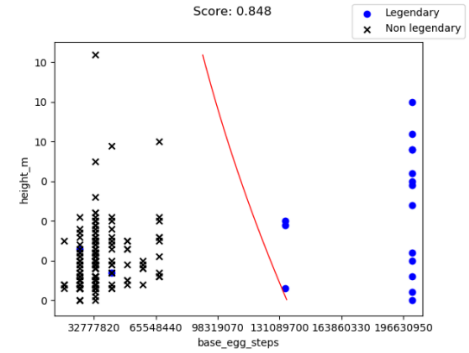
(special attack, height_m) lambda = 1

Aquí se aprecia como los valores de los atributos están mucho más igualados entre legendarios y no legendarios, por lo que se clasifican mucho peor.



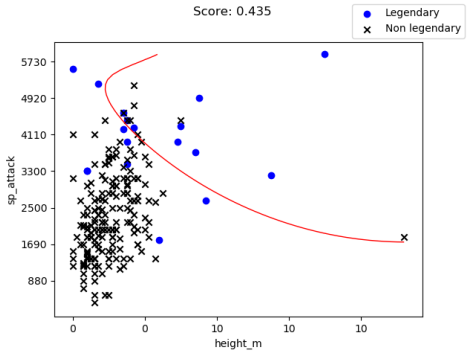
(weight_kg, height_m) lambda = 1

Esta combinación parecía buena a priori debido al gran tamaño gráfico de los distintos pokémon legendarios y la baja media de los normales, pero como se aprecia, algunos normales “inflaban” los valores.



(base egg steps, height) lambda = 1

Hemos comprobado que al añadir los atributos *"base egg steps"* o *"capture rate"* los otros atributos se agrupan con ellos y el score aumenta independientemente de los otros atributos.



(height, special attack) lambda = 1
Polinomio grado 10

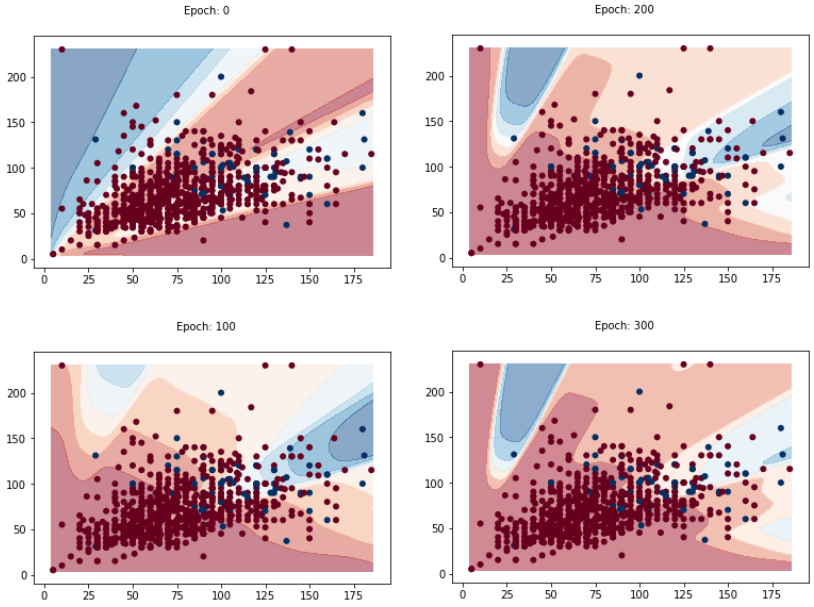
Existe sobreajuste a los ejemplos de entrenamiento.

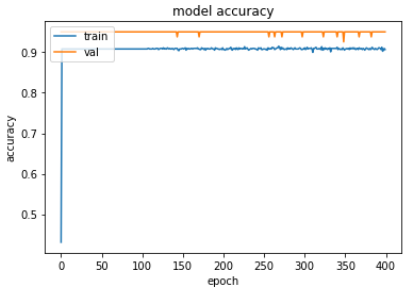
Utilizando los "stats de batalla" y lambda = 1 se obtuvieron scores que iban de 0.940 a 1.0, pero se halló que los atributos más significativos son *"capture_rate"* y *"base_egg_steps"* con lambda = 1, con un score del 0.973.

Al continuar apareciendo mínimos locales en ocasiones, normalizamos los atributos, bajando el score a 0.941 en función de los atributos, pero sorteando mayor cantidad de mínimos locales.

Por último, incluimos Keras para el entrenamiento y pintado del *decision boundary*.

```
Reloaded modules: ML_UtilsModule
error 0.5691162840406249, accuracy 0.9126092195510864
error 0.23436652776900302, accuracy 0.9126092195510864
error 0.22272228969542424, accuracy 0.9126092195510864
error 0.22056160321210355, accuracy 0.9138576984405518
```

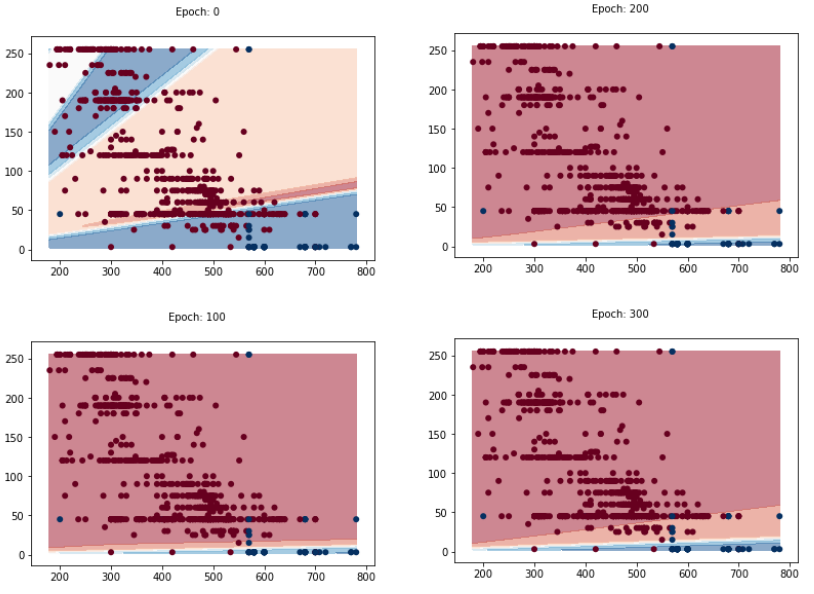


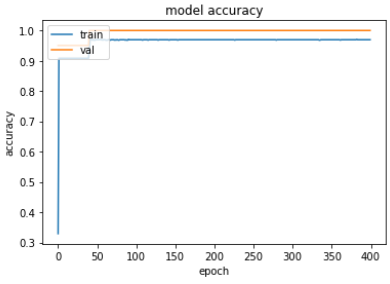


(Attack, defense) 400 vueltas

Se puede apreciar la evolución del *decision boundary* a medida que aumentan las vueltas hasta estabilizarse.

```
error 0.6065992827570245, accuracy 0.9126092195510864
error 0.09723506012836125, accuracy 0.9725343585014343
error 0.09345856864707598, accuracy 0.9725343585014343
error 0.09245994676267275, accuracy 0.9725343585014343
```

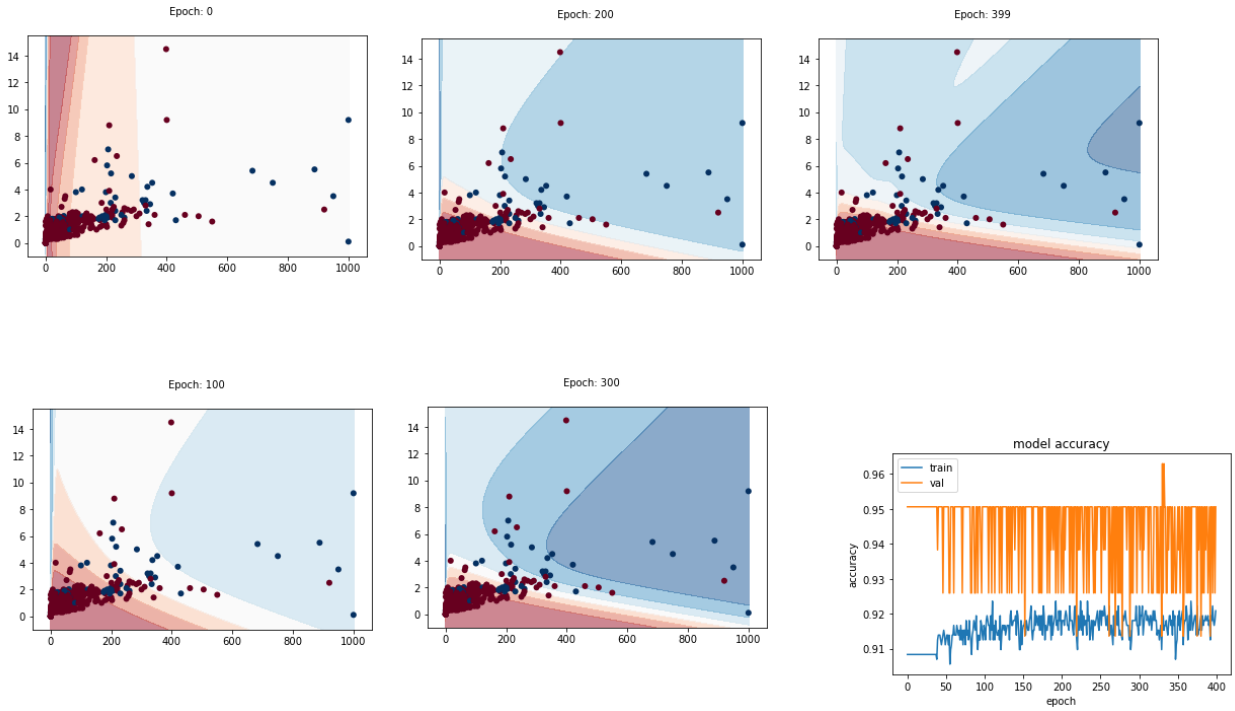




(base total, capture rate) 400 vueltas

Como se puede observar, al añadir el atributo *capture rate*, la gráfica se estabiliza rápidamente y se mantiene muy similar el resto de vueltas.

error 0.29780722368559737, accuracy 0.9126092195510864
error 0.22887185840087287, accuracy 0.9238451719284058
error 0.2257478196225065, accuracy 0.9225967526435852
error 0.22360136020421684, accuracy 0.9213483333587646
error 0.22240472324053, accuracy 0.9213483333587646



(width, height) 400 vueltas

Se puede apreciar que con algunos valores, a pesar de acabar hallando un *decision boundary* bastante preciso, la precisión sufre grandes saltos de una vuelta a otra, llegando a estabilizarse entorno al 0.95 pero dependiendo mucho de la vuelta en la que se finalice.

3.2.2. Clasificador de tipos

La clasificación de los pokémon por tipo con redes neuronales obtiene un resultado muy similar al de las SVM. Haciendo uso de Boruta, se establece que las características más relevantes son *percentage_male* y *sp_attack*. A pesar de ello, se obtienen resultados poco fiables y equilibrados entre tipos. Por ejemplo, se ha observado que ciertos tipos se clasifican mejor respecto a cierto tipo de atributos, mientras que otros no devuelven un resultado igual de óptimo que los demás.

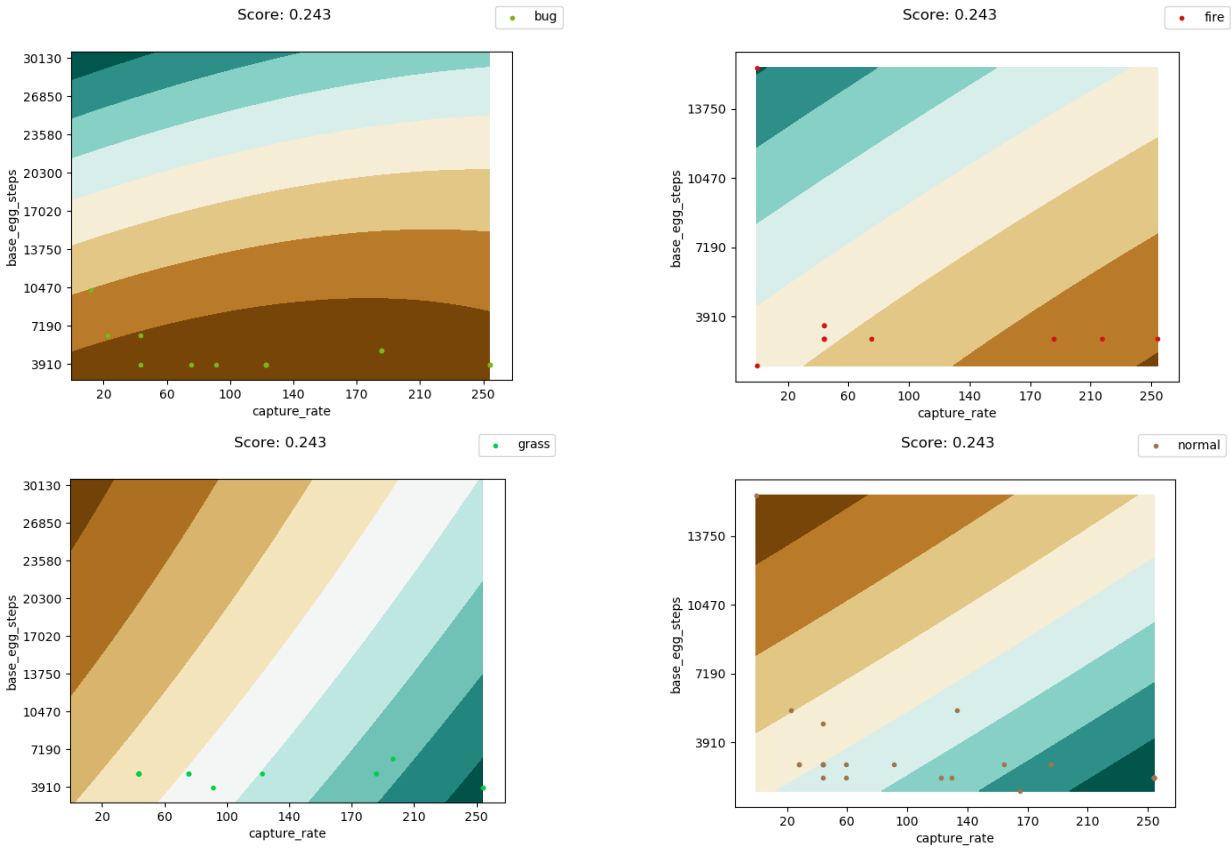
A todo lo anterior, se ha de añadir que ciertos tipos, por ejemplo veneno y dragón, tienen un número muy pequeño de ejemplos de entrenamiento como para poder clasificarlos de manera correcta, por lo tanto, usando solamente los datos de los que disponemos, se obtiene un modelo con un error muy alto y, por lo tanto, poco fiable.

```
Score con los ejemplos de testing: 0.18181818181818182 Type: water
Score con los ejemplos de testing: 0 Type: fire
Score con los ejemplos de testing: 0.23529411764705882 Type: bug
Score con los ejemplos de testing: 0.08333333333333333 Type: normal
```

Atendiendo a la imagen, se puede observar la diferencia de resultados entre los diferentes tipos. Con ello, se puede decir que los atributos seleccionados (*percentage_male* y *sp_attack*) clasifican mejor al tipo bicho y que no sirven para el tipo fuego. Son resultados poco equilibrados y con un error elevado.

```
Score con los ejemplos de testing: 0.1 Type: water
Score con los ejemplos de testing: 0.14285714285714288 Type: fire
Score con los ejemplos de testing: 0 Type: bug
Score con los ejemplos de testing: 0.20689655172413793 Type: normal
```

Con esta comparativa de resultados observando otras características en los mismos tipos (*attack* y *defense*), se confirma la hipótesis anterior, ya que los resultados para el tipo fuego y normal son mejores que en el caso anterior, mientras que el tipo bicho se clasifica erróneamente.



Algunos ejemplos de una misma ejecución con los distintos tipos marcados con sus respectivos resultados.

Como se aprecia, aunque sí tiende a acercarse, en la mayoría de los casos se mantiene muy alejado de una clasificación precisa, ya que por la leyenda utilizada, cuanto más cerca del azul se hallan los puntos, más seguridad hay en su clasificación.

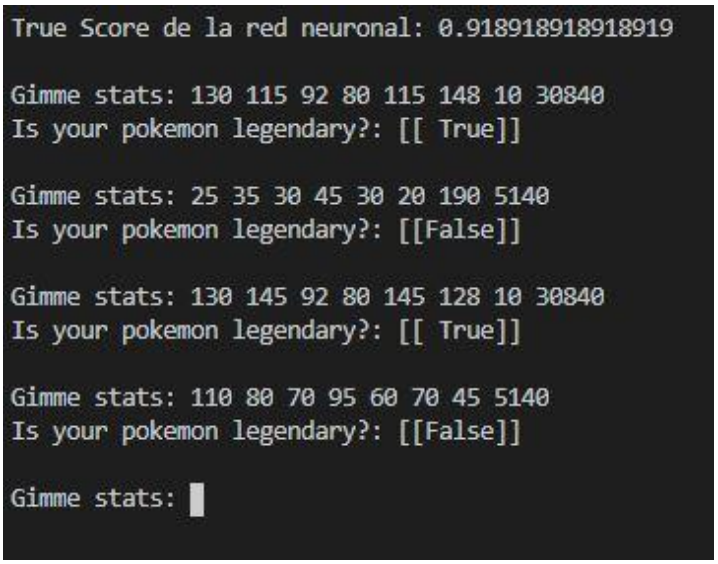
3.3. Conclusiones

3.3.1. Clasificador de legendarios

Haciendo uso de la práctica realizada en clase, se ha conseguido evitar los mínimos locales en los que se estaba cayendo, además de realizar pruebas con diferentes valores de lambda para que la red neuronal no sufriera un *underfitting* / *overfitting* sobre los ejemplos de entrenamiento.

Tras solucionar los problemas anteriores, se concluyó que ciertas características enturbiaban los resultados y se fueron desechando hasta lograr resultados fiables y equilibrados entre diferentes ejecuciones del programa. Atributos como *base_egg_steps*, *capture_rate*, *sp_attack*, *height_m* e incluso *base_happiness* son relevantes a la hora de clasificar legendarios. Se puede concluir que, eligiendo los atributos más importantes y haciendo uso de ejemplos de testeo y ejemplos introducidos por el usuario, la red neuronal consigue catalogar de manera correcta un alto porcentaje de pokémon.

A continuación, se proporciona una captura de pantalla donde se consulta los datos (*attack*, *defense*, *hp*, *sp_attack*, *sp_defense*, *speed*, *capture_rate*) de los mismos pokémon usados anteriormente de la octava generación (Zacian, Snom, Zamazenta, Flapple), además de proporcionar el *score* de la red neuronal, siendo este un 0.92.



3.3.2. Clasificador de tipos

Como ya se ha observado anteriormente en las otras técnicas utilizadas, la clasificación de tipos es complicada atendiendo solamente al conjunto de datos del que se dispone. Se llega a la misma conclusión, son resultados poco fiables y variables entre los diferentes tipos y sus características, además de no conseguir superar la media armónica de 0.4.

Se han consultado diferentes referencias de otros usuarios de Kaggle que han tratado de realizar dicha clasificación a través de redes neuronales, obteniendo resultados parecidos a los explicados anteriormente. Por lo tanto, se concluye que, a pesar de realizar correctamente el algoritmo de clasificación y de elección de parámetros más relevantes, el conjunto de datos disponible no se puede separar de manera precisa y acertada, en cuyo caso, la opción más acertada se basaría en aumentar el número de atributos, por ejemplo, una imagen de cada pokémon y realizar la clasificación a través de las mismas.

Referencias

- François Chollet (2019). Keras [online]. Disponible en: < https://keras.io/> [Accedido el 12 enero 2020].
- Jody Tubi (2018). Forecasting Pokémon Type [online]. Disponible en: <https://www.kaggle.com/jtubiunich/forecasting-pok-mon-type> [Accedido el 18 enero 2020].
- Rounak Banik (2018). The Complete Pokemon Dataset [online]. Disponible en: <https://www.kaggle.com/rounakbanik/pokemon> [Accedido el 12 diciembre 2019].
- Scikit Learn (2019). Sklearn.svm.SVC [online]. Disponible en: <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html> [Accedido el 6 enero 2020].
- Scikit Learn Contrib (2019). Boruta_py [online]. Disponible en: <https://github.com/scikit-learn-contrib/boruta_py> [Accedido el 19 enero 2020].

Código

SVM Legendarios

```
from matplotlib import pyplot as plt
from sklearn.svm import SVC
import numpy as np
from ML_UtilsModule import Data_Management, Normalization
from sklearn.metrics import confusion_matrix
from mpl_toolkits.mplot3d import Axes3D

NUM_TRIES = 1
feature1 = "capture_rate"
feature2 = "base_egg_steps"
feature3 = "attack"

def draw_decisition_boundary(X, y, svm, true_score, mu, sigma, c, s):
    """
    valid for two feature
    """
    #calculating the graphic
    figure, ax = plt.subplots()
    x1 = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
    x2 = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
    x1, x2 = np.meshgrid(x1, x2)
    yp = svm.predict(np.array([x1.ravel(),x2.ravel()]).T).reshape(x1.shape)
    pos = (y == 1).ravel()
    neg = (y == 0).ravel()
    plt.scatter(X[pos, 0], X[pos, 1], color='blue', marker='o', label = "Legendary")
    plt.scatter(X[neg, 0], X[neg, 1], color='black', marker='x', label = "Non legend
ary")
    plt.contour(x1, x2, yp, colors=['red', 'purple'])

    #formatting the graphic with some labels
    plt.xlabel(feature1)
    plt.ylabel(feature2)
    plt.suptitle(("Score: " + str(float("{0:.3f}".format(true_score))) + "\n C: " +
str(c) + "\n Sigma: " + str(s)))
    figure.legend()

    #set the labels to non-normalized values
    figure.canvas.draw()
    labels = [item for item in plt.xticks()[0]]
    for i in range(len(labels)):
        labels[i] = int(round((labels[i] * sigma[0, 0]) + mu[0, 0], -1))
    ax.xaxis.set_ticklabels(labels)

    labels = [item for item in plt.yticks()[0]]
    for i in range(len(labels)):
        labels[i] = int(round((labels[i] * sigma[0, 1]) + mu[0, 1], -1))
    ax.yaxis.set_ticklabels(labels)
```

```
    #show
    plt.show()

def draw_3D(X, y, svm, true_score, mu, sigma):
    fig = plt.figure()
    ax = Axes3D(fig)

    pos = (y == 1).ravel()
    neg = (y == 0).ravel()
    ax.scatter(X[pos, 0], X[pos, 1], X[pos, 2], color='blue', marker='o', label = "L
egendary")
    ax.scatter(X[neg, 0], X[neg, 1], X[neg, 2], color='red', marker='x', label = "No
n Legendary")
    plt.suptitle(("Score: " + str(true_score)))
    fig.legend()

    plt.xlabel(feature1)
    plt.ylabel(feature2)

    plt.show()

    # fig = plt.figure()
    # ax = Axes3D(fig)
    # x1 = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
    # x2 = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
    # x1, x2 = np.meshgrid(x1, x2)
    # yp = svm.predict(np.array([x1.ravel(),x2.ravel()]).T).reshape(x1.shape)

    # ax.plot_surface(x1, x2, yp)

    # pos = (y == 1).ravel()
    # neg = (y == 0).ravel()
    # ax.scatter(X[pos, 0], X[pos, 1], color='blue', marker='o', label = "Legendary"
)
    # ax.scatter(X[neg, 0], X[neg, 1], color='red', marker='x', label = "Non Legenda
ry")

    # plt.suptitle(("Score: " + str(true_score)))
    # fig.legend()

    # plt.show()

def kernel_lineal(X, y, mu, sigma):
    svm = SVC(kernel='linear', C=1.0)
    svm.fit(X, y.ravel())
    score = true_score(X, y, svm)
```

```
    #ssdraw_decisition_boundary(X, y, svm, score, mu, sigma, 1.0, "lineal")

    return svm

def kernel_gaussiano(X, y, mu, sigma):
    sigma = 0.1

    svm = SVC(kernel = 'rbf' , C= 1, gamma= (1 / ( 2 * sigma ** 2)) )
    svm.fit(X, y.ravel())
    score = true_score(X, y, svm)

    draw_decisition_boundary(X, y, svm, score, mu, sigma, 1, sigma)

    return svm

def true_score(X, y, svm):
    predicted_y = svm.predict(X)
    tn, fp, fn, tp = confusion_matrix(y, predicted_y).ravel()
    score = 0
    if tp != 0:
        precision_score = tp / (tp + fp)
        recall_score = tp / (tp + fn)
        score = 2 * (precision_score * recall_score) / (precision_score + recall_sco
re)
    return score

def eleccion_parametros_C_y_Sigma(X, y, Xval, yval, mu, sigma):
    possible_values = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]
    C_value, sigma = 0, 0
    max_score = 0
    best_svm = None
    selected_C = 0
    selected_Sigma = 0

    for i in range(len(possible_values)):
        C_value = possible_values[i]
        for j in range(len(possible_values)):
            sigma = possible_values[j]
            svm = SVC(kernel = 'rbf' , C = C_value, gamma= (1 / ( 2 * sigma ** 2)))
            svm.fit(X, y.ravel())
            current_score = true_score(Xval, yval, svm) #calcula el score con los ej
emplos de validacion (mayor score, mejor es el svm)
            if current_score > max_score:
                max_score = current_score
                best_svm = svm
                selected_C = C_value
                selected_Sigma = sigma
```

```
    return best_svm, selected_C, selected_Sigma

X, y = Data_Management.load_csv_svm("pokemon.csv", [feature1, feature2])
X, mu, sigma = Normalization.normalize_data_matrix(X)
X, y, trainX, trainY, validationX, validationY, testingX, testingY = Data_Management
.divide_legendary_groups(X, y)

max_score = float("-inf")
best_svm = None

for i in range(NUM_TRIES):
    #THIS IS GIVING THE SAME RESULT, ALWAYS (MAYBE SELECT C AND SIGMA RANDOMLY)
    seed = np.random.seed()
    current_svm, C, s = eleccion_parametros_C_y_Sigma(trainX, trainY, validationX, v
alidationY, mu, sigma)
    current_score = true_score(testingX, testingY, current_svm)
    draw_decisition_boundary(testingX, testingY, current_svm, current_score, mu, sig
ma, C, s)
    print("Score con los ejemplos de testing: " + str(current_score))
    if current_score > max_score:
        max_score = current_score
        best_svm = current_svm

while True:
    user_values = np.array(list(map(float, input("Gimme stats: ").split())), dtype=f
loat) # (features, )
    if user_values.size == 0:
        break
    user_values = np.reshape(user_values, (np.shape(user_values)[0], 1))
    user_values = np.transpose(user_values)
    user_values = Normalization.normalize(user_values, mu, sigma) #normalization of
user values
    sol = best_svm.predict(user_values)
    print("Is your pokemon legendary?: " + str(sol[0] == 1.0) + "\n")
```

SVM Tipos

```
from matplotlib import pyplot as plt
from sklearn.svm import SVC
import numpy as np
from ML_UtilsModule import Data_Management, Normalization
from sklearn.metrics import confusion_matrix
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import PolynomialFeatures as pf

NUM_TRIES = 1
feature1 = "attack"
feature2 = "defense"
feature3 = "attack"
grado = 2

def draw_decisition_boundary(X, y, svm, true_score, mu, sigma, c, s, type):
    """
    valid for two feature
    """
    #calculating the graphic
    figure, ax = plt.subplots()
    x1 = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
    x2 = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
    x1, x2 = np.meshgrid(x1, x2)
    yp = svm.predict(np.array([x1.ravel(),x2.ravel()]).T).reshape(x1.shape)
    pos = (y == 1).ravel()
    neg = (y == 0).ravel()
    plt.scatter(X[pos, 0], X[pos, 1], color='blue', marker='o', label = type)
    plt.scatter(X[neg, 0], X[neg, 1], color='black', marker='x', label = "Other type
s")
    plt.contour(x1, x2, yp, colors=['red', 'purple'])

    #formatting the graphic with some labels
    plt.xlabel(feature1)
    plt.ylabel(feature2)
    plt.suptitle(("Score: " + str(float("{0:.3f}".format(true_score))) + "\n C: " +
str(c) + "\n Sigma: " + str(s)))
    figure.legend()

    #set the labels to non-normalized values
    # figure.canvas.draw()
    # labels = [item for item in plt.xticks()[0]]
    # for i in range(len(labels)):
    #     labels[i] = int(round((labels[i] * sigma[0, 0]) + mu[0, 0], -1))
    # ax.xaxis.set_ticklabels(labels)

    # labels = [item for item in plt.yticks()[0]]
    # for i in range(len(labels)):
```

```
    #     labels[i] = int(round((labels[i] * sigma[0, 1]) + mu[0, 1], -1))
    # ax.yaxis.set_ticklabels(labels)

    #show
    plt.show()

def draw_3D(X, y, svm, true_score, mu, sigma):
    fig = plt.figure()
    ax = Axes3D(fig)

    pos = (y == 1).ravel()
    neg = (y == 0).ravel()
    ax.scatter(X[pos, 0], X[pos, 1], X[pos, 2], color='blue', marker='o', label = "L
egendary")
    ax.scatter(X[neg, 0], X[neg, 1], X[neg, 2], color='red', marker='x', label = "No
n Legendary")
    plt.suptitle(("Score: " + str(true_score)))
    fig.legend()

    plt.xlabel(feature1)
    plt.ylabel(feature2)

    plt.show()

    # fig = plt.figure()
    # ax = Axes3D(fig)
    # x1 = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
    # x2 = np.linspace(X[:, 1].min(), X[:, 1].max(), 100)
    # x1, x2 = np.meshgrid(x1, x2)
    # yp = svm.predict(np.array([x1.ravel(),x2.ravel()]).T).reshape(x1.shape)

    # ax.plot_surface(x1, x2, yp)

    # pos = (y == 1).ravel()
    # neg = (y == 0).ravel()
    # ax.scatter(X[pos, 0], X[pos, 1], color='blue', marker='o', label = "Legendary"
)
    # ax.scatter(X[neg, 0], X[neg, 1], color='red', marker='x', label = "Non Legenda
ry")

    # plt.suptitle(("Score: " + str(true_score)))
    # fig.legend()

    # plt.show()

def true_score(X, y, svm):
    predicted_y = svm.predict(X)
    tp = 0
```

```
    fp = 0
    fn = 0

    for i in range(np.shape(predicted_y)[0]):
        if predicted_y[i] == 1 and y[i] == 1:
            tp += 1
        elif predicted_y[i] == 1 and y[i] == 0:
            fp += 1
        elif predicted_y[i] == 0 and y[i] == 1:
            fn += 1

    score = 0
    if tp != 0:
        precision_score = tp / (tp + fp)
        recall_score = tp / (tp + fn)
        score = 2 * (precision_score * recall_score) / (precision_score + recall_sco
re)

    return score

def eleccion_parametros_C_y_Sigma(X, y, Xval, yval, mu, sigma):
    possible_values = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]
    C_value, sigma = 0, 0
    max_score = float("-inf")
    best_svm = None
    selected_C = 0
    selected_Sigma = 0

    for i in range(len(possible_values)):
        C_value = possible_values[i]
        for j in range(len(possible_values)):
            sigma = possible_values[j]
            svm = SVC(kernel = 'rbf' , C = C_value, gamma= (1 / ( 2 * sigma ** 2)),
probability=True)
            svm.fit(X, y.ravel())
            current_score = true_score(Xval, yval, svm) #calcula el score con los ej
emplos de validacion (mayor score, mejor es el svm)
            if current_score > max_score:
                max_score = current_score
                best_svm = svm
                selected_C = C_value
                selected_Sigma = sigma

    return best_svm, selected_C, selected_Sigma

def transform_y(y, num_etiquetas):
    y = np.reshape(y, (np.shape(y)[0], 1))
    mask = np.empty((num_etiquetas, np.shape(y)[0]), dtype=bool)
```

```
    for i in range( num_etiquetas):
        mask[i, :] = (y[:, 0] == i)

    mask = mask * 1

    return np.transpose(mask)

def divideRandomGroups(X, y):
    X, y = Data_Management.shuffle_in_unison_scary(X, y)
    # ----------------------------------------------------------------------------
----------------------
    percent_train = 0.6
    percent_valid = 0.2
    percent_test = 0.2
    # ----------------------------------------------------------------------------
----------------------
    # TRAINIG GROUP
    t = int(np.shape(X)[0]*percent_train)
    trainX = X[:t]
    trainY= y[:t]
    # ----------------------------------------------------------------------------
----------------------
    # VALIDATION GROUP
    v=int( np.shape(trainX)[0]+np.shape(X)[0]*percent_valid)
    validationX = X[np.shape(trainX)[0] : v]
    validationY= y[np.shape(trainY)[0] : v]
    # ----------------------------------------------------------------------------
----------------------
    # TESTING GROUP
    testingX = X[np.shape(trainX)[0]+np.shape(validationX)[0] :]
    testingY= y[np.shape(trainY)[0]+np.shape(validationY)[0] :]

    return trainX, trainY, validationX, validationY, testingX, testingY

def predict_type(user_values, svms):
    max_security = float("-inf")
    predicted_type = 0

    for i in range(len(svms)):
        sec = svms[i].predict_proba(user_values)
        if sec[0, 1] > max_security:
            max_security = sec[0, 1]
            predicted_type = i

    return max_security, predicted_type


def polynomial_features(X, grado):
    poly = pf(grado)
```

```
    return (poly.fit_transform(X))

# X, y = Data_Management.load_csv_types_features("pokemon.csv",['against_bug', 'agai
nst_dark','against_dragon','against_electric',
#                         'against_fairy','against_fight','against_fire','against_f
lying',
#                         'against_ghost','against_grass','against_ground','against
_ice','against_normal',
#                         'against_poison','against_psychic','against_rock','agains
t_steel','against_water'])

X, y = Data_Management.load_csv_types_features("pokemon.csv", [feature1, feature2])
# TODO: usar el tipo2 para sacar el score tambien (si mi svm predice 1 y una de las
dos y es 1, es truePositive++) y dar el resultado con solo
# 1 tipo, todo lo del entrenamiento se queda igual (se entrena para un solo tipo). L
uego en el score se hace eso y para predecir el tipo se queda igual.
# Tambien puedo sacar dos svm, tipo primario y tipo secundario pero mas lio ?


X = polynomial_features(X, grado)
X, mu, sigma = Normalization.normalize_data_matrix(X[:, 1:])
X = Data_Management.add_column_left_of_matrix(X)

trainX, trainY, validationX, validationY, testingX, testingY = divideRandomGroups(X,
 y)

svms = []

for j in range(18):
    currentTrainY = (trainY == j) * 1
    currentValidationY = (validationY == j) * 1
    currentTestingY = (testingY == j) * 1
    current_svm, C, s = eleccion_parametros_C_y_Sigma(trainX, currentTrainY, validat
ionX, currentValidationY, mu, sigma)
    current_score = true_score(testingX, currentTestingY, current_svm)
    if np.shape(trainX)[1] == 2:
        draw_decisition_boundary(testingX, currentTestingY, current_svm, current_sco
re, mu, sigma, C, s, Data_Management.getTypeByIndex(j))
    svms.append(current_svm)
    print("Score con los ejemplos de testing: " + str(current_score) + " Type: " + D
ata_Management.getTypeByIndex(j))

while True:
    user_values = np.array(list(map(float, input("Gimme stats: ").split())), dtype=f
loat) # (features, )
    if user_values.size == 0:
        break
    user_values = np.reshape(user_values, (np.shape(user_values)[0], 1))
    user_values = np.transpose(user_values)
```

```
    user_values = polynomial_features(user_values, grado)
    user_values = Normalization.normalize(user_values[:, 1:], mu, sigma) #normalizat
ion of user values
    sec, pokemon_type = predict_type(user_values, svms)
    print("Predicted type: " + Data_Management.getTypeByIndex(pokemon_type) + ". Pro
bability of that type: " + str(sec) + "\n")
```

Regresión Logística Legendarios

```
from scipy.optimize import fmin_tnc as tnc
from ML_UtilsModule import Data_Management
from ML_UtilsModule import Normalization
from matplotlib import pyplot as plt
from sklearn.preprocessing import PolynomialFeatures as pf
import numpy as np
import sys

learning_rate = 1.0
NUM_TRIES = 10

def polinomial_features(X, grado):
    poly = pf(grado)
    return poly, poly.fit_transform(X)

def J(theta, X, Y):
    m = np.shape(X)[0]
    n = np.shape(X)[1]
    theta = np.reshape(theta, (1, n))
    var1 = np.dot(np.transpose((np.log(g(np.dot(X, np.transpose(theta)))))), Y)

    aux1 = np.dot(X, np.transpose(theta))
    aux2 = 1 - g(aux1)
    aux3 = np.log(aux2)

    var2 = np.dot(np.transpose(aux3), (1 - Y))
    var3 = (learning_rate/(2*m)) * np.sum(theta[1:]**2)
    return (((-1/m)*(var1 + var2)) + var3)

def gradient(theta, X, Y):
    m = np.shape(X)[0]
    n = np.shape(X)[1]
    theta = np.reshape(theta, (1, n))
    var1 = np.transpose(X)
    var2  = g(np.dot(X, np.transpose(theta)))-Y

    theta = np.c_[[0], theta[:, 1:]]
    var3 = (learning_rate/m) * theta
    return ((1/m) * np.dot(var1, var2)) + np.transpose(var3)

def g(z):
    """
    1/ 1 + e ^ (-0^T * x)
    """
    return 1/(1 + np.exp(-z))
```

```
def draw_data(X, Y):
    pos = np.where(Y == 0)[0] #vector with index of the Y = 1
    plt.scatter(X[pos, 0], X[pos, 1], marker='.', c='r', label = "Non Legendary")
    pos = np.where(Y == 1)[0].ravel() #vector with index of the Y = 1
    plt.scatter(X[pos, 0], X[pos, 1], marker='.', c='y', label = "Legendary")


def draw_decision_boundary(theta, X, Y, poly):
    x0_min, x0_max = X[:,0].min(), X[:,0].max()
    x1_min, x1_max = X[:,1].min(), X[:,1].max()
    xx1, xx2 = np.meshgrid(np.linspace(x0_min, x0_max), np.linspace(x1_min, x1_max))

    sigm = g(poly.fit_transform(np.c_[ xx1.ravel(), xx2.ravel()]).dot(theta))
    sigm = sigm.reshape(xx1.shape)

    plt.contour(xx1, xx2, sigm, [0.5], linewidths = 1, colors = 'g')

def format_graphic (figure, ax, graphic_attr_names, score, polyDegree, sigma, mu):

    #formatting the graphic with some labels
    plt.xlabel(graphic_attr_names[0])
    plt.ylabel(graphic_attr_names[1])
    plt.suptitle(("Score: " + str(float("{0:.3f}".format(score))) + ", poly: " + str
(polyDegree)))
    figure.legend()

    #set the labels to non-normalized values
    figure.canvas.draw()
    labels = [item for item in plt.xticks()[0]]
    for i in range(len(labels)):
        labels[i] = int(round((labels[i] * sigma[0, 0]) + mu[0, 0], -1))
    ax.xaxis.set_ticklabels(labels)

    labels = [item for item in plt.yticks()[0]]
    for i in range(len(labels)):
        labels[i] = int(round((labels[i] * sigma[0, 1]) + mu[0, 1], -1))
    ax.yaxis.set_ticklabels(labels)

def draw(theta, X, Y, poly, graphic_attr_names, score, polyDegree, sigma, mu):
    figure, ax = plt.subplots()

    draw_data(X, Y)
    draw_decision_boundary(theta, X, Y, poly)

    format_graphic(figure, ax, graphic_attr_names, score, polyDegree, sigma, mu)

    plt.show()

def checkLearned(X, Y, theta):
```

```
    checker = g(np.dot(X, np.transpose(theta)))
    checker = np.around(checker)
    checker = np.reshape(checker, (np.shape(checker)[0], 1))
    truePositives = 0
    falsePositives = 0
    falseNegatives = 0

    for i in range(np.size(checker)):
        if checker[i] == 1 and y[i] == 1:
            truePositives += 1
        elif checker[i] == 1 and y[i] == 0:
            falsePositives += 1
        elif checker[i] == 0 and y[i] == 1:
            falseNegatives += 1

    if truePositives == 0:
        return 0

    recall = (truePositives/(truePositives + falseNegatives))
    precision = (truePositives/(truePositives + falsePositives))
    score = 2 *(precision*recall/(precision + recall))

    return score

graphic_attr_names = ["capture_rate", "base_egg_steps"]
X, y = Data_Management.load_csv_svm("pokemon.csv", graphic_attr_names)
X, mu, sigma = Normalization.normalize_data_matrix(X)

X, y, trainX, trainY, validationX, validationY, testingX, testingY = Data_Management
.divide_legendary_groups(X, y)

#-------------------------------------------------------------------------------------
--------------

allMaxPercent = []
allMaxElev = []
allMaxPoly = []
allMaxThetas = []

Xused = validationX
Yused = validationY

for t in range(NUM_TRIES):
    i = 1
    polyMaxPercent = 0
    maxPercent = 0
    currentPercent = 0
    maxTh = None
```

```
    maxPoly = None

    poly, X_poly = polinomial_features(Xused, i)
    theta = np.zeros([1, np.shape(X_poly)[1]], dtype=float)

    theta = tnc(func=J, x0=theta, fprime=gradient, args=(X_poly, Yused))[0]

    currentPercent = checkLearned(X_poly, Yused, theta)

    maxTh = theta
    maxPoly = poly

    while currentPercent > maxPercent:
        maxPercent =  currentPercent
        polyMaxPercent = i
        maxTh = theta
        maxPoly = poly

        i = i + 1

        poly, X_poly = polinomial_features(Xused, i)
        theta = np.zeros([1, np.shape(X_poly)[1]], dtype=float)

        theta = tnc(func=J, x0=theta, fprime=gradient, args=(X_poly, Yused))[0]

        currentPercent = checkLearned(X_poly, Yused, theta)

    allMaxPercent.append(maxPercent)
    allMaxElev.append(polyMaxPercent)
    allMaxPoly.append(maxPoly)
    allMaxThetas.append(maxTh)

indx = allMaxPercent.index(max(allMaxPercent))

draw(allMaxThetas[indx], Xused, Yused, allMaxPoly[indx], graphic_attr_names, max(all
MaxPercent), allMaxElev[indx],sigma, mu)

#-------------------------------------------------------------------------------
--------------
```

Regresión Logística Tipos:

```
from scipy.optimize import fmin_tnc as tnc
from ML_UtilsModule import Data_Management
from ML_UtilsModule import Normalization
from matplotlib import pyplot as plt
from sklearn.preprocessing import PolynomialFeatures as pf
import numpy as np
import sys

learning_rate = 0.1
NUM_TRIES = 10

def polinomial_features(X, grado):
    poly = pf(grado)
    return poly, poly.fit_transform(X)

def J(theta, X, Y):
    m = np.shape(X)[0]
    n = np.shape(X)[1]
    theta = np.reshape(theta, (1, n))
    var1 = np.dot(np.transpose((np.log(g(np.dot(X, np.transpose(theta)))))), Y)

    aux1 = np.dot(X, np.transpose(theta))
    aux2 = 1 - g(aux1)
    aux3 = np.log(aux2)

    var2 = np.dot(np.transpose(aux3), (1 - Y))
    var3 = (learning_rate/(2*m)) * np.sum(theta[1:]**2)
    return (((-1/m)*(var1 + var2)) + var3)

def gradient(theta, X, Y):
    m = np.shape(X)[0]
    n = np.shape(X)[1]
    theta = np.reshape(theta, (1, n))
    var1 = np.transpose(X)
    var2  = g(np.dot(X, np.transpose(theta)))-Y

    theta = np.c_[[0], theta[:, 1:]]
    var3 = (learning_rate/m) * theta
    return ((1/m) * np.dot(var1, var2)) + np.transpose(var3)

def g(z):
    """
    1/ 1 + e ^ (-0^T * x)
    """
    return 1/(1 + np.exp(-z))

def transform_y(y, num_etiquetas):
```

```
    y = np.reshape(y, (np.shape(y)[0], 1))
    mask = np.empty((num_etiquetas, np.shape(y)[0]), dtype=bool)
    for i in range( num_etiquetas):
        mask[i, :] = (y[:, 0] == i)

    mask = mask * 1

    return np.transpose(mask)

def des_transform_y(y, num_etiquetas):
    deTransY = np.where(y == 1)
    return deTransY[1]

def paint_pkmTypes(X, y, types = None, paintAll = False):
    '''
    Creates plt figure and draws the pkms used as input by the first two values of X
    and changes the color of them depending on the type

    X = attributes
    y = list of types, deTransformed (from 0 to 17)
    types = types that are going to be printed
    '''
    typesIndx = []
    typesNotSelectedIndx = []

    if(types == None):
        types = Data_Management.types_
        # esto es para que meta los indices del 0 al 17, de los tipos completos, com
o no lo conseguia, he pasado
        typesIndx = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]
    else:
        for t in range(len(types)):
            typesIndx.append(Data_Management.types_.index(types[t]))


    colors = Data_Management.colors_

    if paintAll:
        plt.scatter(X[:, 0], X[:, 1], color=colors[18], marker='.', label = 'other')

    for i in range(len(typesIndx)):
        pos = (y == typesIndx[i]).ravel()
        plt.scatter(X[pos, 0], X[pos, 1], color='#ff0303', marker='+', label = types
[i])


def draw_decision_boundary(theta, X, Y, poly, indx = 0):
    x0_min, x0_max = X[:,0].min(), X[:,0].max()
    x1_min, x1_max = X[:,1].min(), X[:,1].max()
```

```
    xx1, xx2 = np.meshgrid(np.linspace(x0_min, x0_max), np.linspace(x1_min, x1_max))

    sigm = g(poly.fit_transform(np.c_[ xx1.ravel(), xx2.ravel()]).dot(theta))
    sigm = sigm.reshape(xx1.shape)

    plt.contour(xx1, xx2, sigm, [0.5], linewidths = 1, colors = Data_Management.colo
rs_[indx])

def format_graphic (figure, ax, graphic_attr_names, score, polyDegree, sigma, mu):

    #formatting the graphic with some labels
    plt.xlabel(graphic_attr_names[0])
    plt.ylabel(graphic_attr_names[1])

    if polyDegree == 0:
        polyDegree = 1

    plt.suptitle(("Score: " + str(float("{0:.3f}".format(score))) + ", poly: " + str
(polyDegree)))
    figure.legend()

    #set the labels to non-normalized values
    figure.canvas.draw()
    labels = [item for item in plt.xticks()[0]]
    for i in range(len(labels)):
        labels[i] = (round((labels[i] * sigma[0, 0]) + mu[0, 0], 1)) #int
    ax.xaxis.set_ticklabels(labels)

    labels = [item for item in plt.yticks()[0]]
    for i in range(len(labels)):
        labels[i] = (round((labels[i] * sigma[0, 1]) + mu[0, 1], 1)) #int
    ax.yaxis.set_ticklabels(labels)

def draw(theta, X, Y, poly, graphic_attr_names, score, polyDegree, sigma, mu, types
= None, paintAll = False):
    figure, ax = plt.subplots()

    paint_pkmTypes(X, Y, types, paintAll)

    if(types == None):
        for i in range(18):
            draw_decision_boundary(theta[i], X, Y, poly, i)
    else:
        for t in range(len(types)):
            draw_decision_boundary(theta[Data_Management.types_.index(types[t])], X,
 Y, poly, Data_Management.types_.index(types[t]))

    format_graphic(figure, ax, graphic_attr_names, score, polyDegree, sigma, mu)
```

```
    plt.show()

def checkLearned(X, Y, theta):
    checker = g(np.dot(X, np.transpose(theta)))
    maxIndexV = np.argmax(checker, axis = 1)
    checker = (Y[:] == maxIndexV)

    truePositives = 0
    falsePositives = 0
    falseNegatives = 0

    for i in range(np.shape(Y)[0]):
        for j in range(18):
            if maxIndexV[i] == j and y[i] == j:
                truePositives += 1
                break
            elif maxIndexV[i] == j and y[i] != j:
                falsePositives += 1
                break
            elif maxIndexV[i] != j and y[i] == j:
                falseNegatives += 1
                break

    if truePositives == 0:
        return 0

    recall = (truePositives/(truePositives + falseNegatives))
    precision = (truePositives/(truePositives + falsePositives))
    score = 2 *(precision*recall/(precision + recall))

    return score

def checkLearnedByType(X, Y, theta, indxTypeChecked):
    checker = g(np.dot(X, np.transpose(theta)))
    maxIndexV = np.argmax(checker, axis = 1)
    checker = (Y[:] == maxIndexV)

    truePositives = 0
    falsePositives = 0
    falseNegatives = 0


    for i in range(np.shape(Y)[0]):
            if maxIndexV[i] == indxTypeChecked and y[i] == indxTypeChecked:
                truePositives += 1
                break
            elif maxIndexV[i] == indxTypeChecked and y[i] != indxTypeChecked:
```

```
                falsePositives += 1
                break
            elif maxIndexV[i] != indxTypeChecked and y[i] == indxTypeChecked:
                falseNegatives += 1
                break

    if truePositives == 0:
        return 0

    recall = (truePositives/(truePositives + falseNegatives))
    precision = (truePositives/(truePositives + falsePositives))
    score = 2 *(precision*recall/(precision + recall))

    return score


graphic_attr_names = ["against_normal", "against_ice"]
types_rendered = ["steel"]
num_tipos = 18
X, y = Data_Management.load_csv_types_features("pokemon.csv", graphic_attr_names)
X, mu, sigma = Normalization.normalize_data_matrix(X)

X, y, trainX, trainY, validationX, validationY, testingX, testingY = Data_Management
.divide_legendary_groups(X, y)

#-------------------------------------------------------------------------------------
--------------
allMaxPercent = []
allMaxElev = []
allMaxPoly = []
allMaxThetas = []

Xused = X
Yused = y
Yused = transform_y(Yused, num_tipos)

for t in range(NUM_TRIES):
    i = 1
    polyMaxPercent = 0
    maxPercent = 0
    currentPercent = 0
    maxTh = None
    maxPoly = None

    poly, X_poly = polinomial_features(Xused, i)
    theta = np.zeros([num_tipos, np.shape(X_poly)[1]], dtype=float)

    for j in range(num_tipos):
```

```
        yTipo = np.reshape(Yused[:,j], (np.shape(Yused)[0], 1))
        theta[j] = tnc(func=J, x0=theta[j], fprime=gradient, args=(X_poly, yTipo))[0
]

    currentPercent = checkLearned(X_poly, Yused, theta)

    maxTh = theta
    maxPoly = poly

    while currentPercent > maxPercent:
        maxPercent =  currentPercent
        polyMaxPercent = i
        maxTh = theta
        maxPoly = poly

        i = i + 1

        poly, X_poly = polinomial_features(Xused, i)
        theta = np.zeros([num_tipos, np.shape(X_poly)[1]], dtype=float)

        for j in range(num_tipos):
            yTipo = np.reshape(Yused[:,j], (np.shape(Yused)[0], 1))
            theta[j] = tnc(func=J, x0=theta[j], fprime=gradient, args=(X_poly, yTipo
))[0]

        currentPercent = checkLearned(X_poly, Yused, theta)

    allMaxPercent.append(maxPercent)
    allMaxElev.append(polyMaxPercent)
    allMaxPoly.append(maxPoly)
    allMaxThetas.append(maxTh)

indx = allMaxPercent.index(max(allMaxPercent))
Yused = des_transform_y(Yused, num_tipos)

for pkm in range(18):
    draw(allMaxThetas[indx], Xused, Yused, allMaxPoly[indx], graphic_attr_names, max
(allMaxPercent), allMaxElev[indx],sigma, mu, [Data_Management.types_[pkm]], True)

#-------------------------------------------------------------------------------
--------------
```

```
from ML_UtilsModule import Data_Management
from scipy.optimize import minimize as sciMin
from sklearn.preprocessing import PolynomialFeatures as pf
import numpy as np
from matplotlib import pyplot as plt
from ML_UtilsModule import Normalization


lambda_ = 1
NUM_TRIES = 1

def g(z):
    """
    1/ 1 + e ^ (-0^T * x)
    """
    return 1/(1 + np.exp(-z))

def derivada_de_G(z):
    result = g(z) * (1 - g(z))
    return result

def polinomial_features(X, grado):
    poly = pf(grado)
    return (poly, poly.fit_transform(X))

def pesos_aleat(L_in, L_out):
    pesos = np.random.uniform(-0.12, 0.12, (L_out, 1+L_in))

    return pesos

def transform_y(y, num_etiquetas):
    #y = np.reshape(y, (np.shape(y)[0], 1))
    mask = np.empty((num_etiquetas, np.shape(y)[0]), dtype=bool)
    for i in range( num_etiquetas):
        mask[i, :] = ((y[:, 0] + num_etiquetas - 1) % num_etiquetas == i)
        #codificado con el numero 1 en la posicion 0 y el numero 0 en la posicion 9

    mask = mask * 1

    return np.transpose(mask)

def J(X, y, a3, num_etiquetas, theta1, theta2):
    m = np.shape(X)[0]
    aux1 = -y * (np.log(a3))
    aux2 = (1 - y) * (np.log(1 - a3))
    aux3 = aux1 - aux2
    aux4 = np.sum(theta1**2) + np.sum(theta2**2)
```

```
    print (aux4)
    return (1/m) * np.sum(aux3) + (lambda_/(2*m)) * aux4

def forward_propagate(X, theta1, theta2):
    m = X.shape[0]
    a1 = np.hstack([np.ones([m, 1]), X])
    z2 = np.dot(a1, theta1.T)
    a2 = np.hstack([np.ones([m, 1]), g(z2)])
    z3 = np.dot(a2, theta2.T)
    h = g(z3)
    return a1, z2, a2, z3, h

def propagation(a1, theta1, theta2):
    a1 = Data_Management.add_column_left_of_matrix(a1)
    a2 = g(np.dot(a1, np.transpose(theta1)))
    a2 = Data_Management.add_column_left_of_matrix(a2)

    a3 = g(np.dot(a2, np.transpose(theta2)))

    return a1, a2, a3

def backdrop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg):
    """
    return coste y gradiente de una red neuronal de dos capas
    """
    theta1 = np.reshape(params_rn[:num_ocultas*(num_entradas + 1)],
        (num_ocultas, (num_entradas + 1)))
    theta2 = np.reshape(params_rn[num_ocultas*(num_entradas + 1):],
        (num_etiquetas, (num_ocultas + 1)))

    #--------------------PASO1----------------------------------------

    a1, a2, a3 = propagation(X, theta1, theta2)
    m = np.shape(X)[0]
    delta_3 = a3 - y # (5000, 10)
    #--------------------PASO2----------------------------------------
    #delta_3 = a3 - y # (5000, 10)
    delta_matrix_1 = np.zeros(np.shape(theta1))
    delta_matrix_2 = np.zeros(np.shape(theta2))

    aux1 = np.dot(delta_3, theta2) #(5000, 26)
    aux2 = Data_Management.add_column_left_of_matrix(derivada_de_G(np.dot(a1, np.tra
nspose(theta1))))
    delta_2 = aux1 * aux2 #(5000, 26)
    delta_2 = np.delete(delta_2, [0], axis=1) #(5000, 25)

    # #--------------------PASO4----------------------------------------
```

```
    delta_matrix_1 = delta_matrix_1 + np.transpose(np.dot(np.transpose(a1), delta_2)
) #(25, 401)
    delta_matrix_2 = delta_matrix_2 + np.transpose(np.dot(np.transpose(a2), delta_3)
) #(10, 26)
    #--------------------PASO6-----------------------------------
    delta_matrix_1 = (1/m) * delta_matrix_1
    delta_matrix_1[:, 1:] = delta_matrix_1[:, 1:] + (reg/m) * theta1[:, 1:]

    delta_matrix_2 = (1/m) * delta_matrix_2
    delta_matrix_2[:, 1:] = delta_matrix_2[:, 1:] + (reg/m) * theta2[:, 1:]


    cost = J(X, y, a3, num_etiquetas, theta1, theta2)
    gradient = np.concatenate((np.ravel(delta_matrix_1), np.ravel(delta_matrix_2)))

    return cost, gradient

def checkLearned(y, outputLayer):
    checker = (outputLayer > 0.7)
    truePositives = 0
    falsePositives = 0
    falseNegatives = 0

    for i in range(np.size(checker)):
        if checker[i, 0] == True and y[i, 0] == 1:
            truePositives += 1
        elif checker[i, 0] == True and y[i, 0] == 0:
            falsePositives += 1
        elif checker[i, 0] == False and y[i, 0] == 1:
            falseNegatives += 1

    if truePositives == 0:
        return 0

    recall = (truePositives/(truePositives + falseNegatives))
    precision = (truePositives/(truePositives + falsePositives))
    score = 2 *(precision*recall/(precision + recall))

    # PORCENTAJE DE ACIERTOS TOTALES
    #count = np.size(np.where(checker[:, 0] == çy[:, 0]))
    #fin = count/np.shape(y)[0] * 100

    return score

def pintaTodo(X, y, error, errorTr, true_score):
    plt.figure()
    #plt.ylim(0,1)
```

```
    plt.plot(np.linspace(0,len(error)-
1, len(error), dtype = int), error[:], color="grey")
    plt.plot(np.linspace(0,len(errorTr)-
1, len(errorTr), dtype = int), errorTr[:], color="green")
    plt.suptitle(("Score: " + str(true_score)))

    plt.show()

def paint_graphic(X, y, true_score, theta1, theta2, mu, sigma):
    figure, ax = plt.subplots()

    pos = (y == 1).ravel()
    neg = (y == 0).ravel()
    plt.scatter(X[pos, 0], X[pos, 1], color='blue', marker='o', label = "Legendary")
    plt.scatter(X[neg, 0], X[neg, 1], color='black', marker='x', label = "Non legend
ary")

    x0_min, x0_max = X[:,0].min(), X[:,0].max()
    x1_min, x1_max = X[:,1].min(), X[:,1].max()
    xx1, xx2 = np.meshgrid(np.linspace(x0_min, x0_max), np.linspace(x1_min, x1_max))

    aux = np.c_[ xx1.ravel(), xx2.ravel()]
    p, aux = polinomial_features(aux, 5)
    aux = Normalization.normalize(aux[:, 1:], mu, sigma)
    sigm = forward_propagate(aux, theta1, theta2)[4]
    sigm = np.reshape(sigm, np.shape(xx1))
    plt.contour(xx1, xx2, sigm, [0.5], linewidths = 1, colors=['red', 'purple'])

    #formatting the graphic with some labels
    plt.xlabel("weight_kg")
    plt.ylabel("speed")
    plt.suptitle(("Score: " + str(float("{0:.3f}".format(true_score)))))
    figure.legend()

    #set the labels to non-normalized values
    figure.canvas.draw()
    labels = [item for item in plt.xticks()[0]]
    for i in range(len(labels)):
        labels[i] = int(round((labels[i] * sigma[0, 0]) + mu[0, 0], -1))
    ax.xaxis.set_ticklabels(labels)

    labels = [item for item in plt.yticks()[0]]
    for i in range(len(labels)):
        labels[i] = int(round((labels[i] * sigma[0, 1]) + mu[0, 1], -1))
    ax.yaxis.set_ticklabels(labels)

    plt.show()
```

```
X, y = Data_Management.load_csv_svm("pokemon.csv", ["base_egg_steps", "base_happines
s"])
nX, ny, ntrainX, ntrainY, nvalidationX, nvalidationY, ntestingX, ntestingY = Data_Ma
nagement.divide_legendary_groups(X, y)

#normalize
p, X = polinomial_features(X, 5)
X, mu, sigma = Normalization.normalize_data_matrix(X[:, 1:])
#ssX = Data_Management.add_column_left_of_matrix(X)

X, y, trainX, trainY, validationX, validationY, testingX, testingY = Data_Management
.divide_legendary_groups(X, y)


num_entradas = np.shape(X)[1]
num_ocultas = 25
num_etiquetas = 1
true_score_max = float("-inf")

thetaTrueMin1 = None
thetaTrueMin2 = None

for j in range(NUM_TRIES):
    theta1 = pesos_aleat(num_entradas, num_ocultas)
    theta2 = pesos_aleat(num_ocultas, num_etiquetas)

    theta_vector = np.concatenate((np.ravel(theta1), np.ravel(theta2)))

    auxErr = []
    auxErrTr = []
    thetaMin1 = None
    thetaMin2 = None
    errorMin = float("inf")

    for i in range(1, np.shape(trainX)[0]):
        thetas = sciMin(fun=backdrop, x0=theta_vector,
        args=(num_entradas, num_ocultas, num_etiquetas, trainX[:i], trainY[:i], lamb
da_),
        method='TNC', jac=True,
        options={'maxiter': 70}).x

        theta1 = np.reshape(thetas[:num_ocultas*(num_entradas + 1)],
                (num_ocultas, (num_entradas + 1)))
        theta2 = np.reshape(thetas[num_ocultas*(num_entradas + 1):],
                (num_etiquetas, (num_ocultas + 1)))

        auxErr.append(J(validationX, validationY, forward_propagate(validationX, the
ta1, theta2)[4], num_etiquetas, theta1, theta2))
```

```
        auxErrTr.append(J(trainX[:i], trainY[:i], forward_propagate(trainX[:i], thet
a1, theta2)[4], num_etiquetas, theta1, theta2))

        if errorMin > auxErr[-1]:
            errorMin = auxErr[-1]
            thetaMin1 = theta1
            thetaMin2 = theta2


    true_score = checkLearned(testingY, forward_propagate(testingX, thetaMin1, theta
Min2)[4])
    if true_score > true_score_max:
        true_score_max = true_score
        thetaTrueMin1 = thetaMin1
        thetaTrueMin2 = thetaMin2
        pintaTodo(testingX, testingY, auxErr, auxErrTr, true_score)

paint_graphic(ntestingX, ntestingY, true_score_max, thetaTrueMin1, thetaTrueMin2, mu
, sigma)

print("True Score de la red neuronal: " + str(true_score_max) + "\n")
while True:
    user_values = np.array(list(map(float, input("Gimme stats: ").split())), dtype=f
loat) # (features, )
    if user_values.size == 0:
        break
    user_values = np.reshape(user_values, (np.shape(user_values)[0], 1))
    user_values = np.transpose(user_values)
    user_values = Normalization.normalize(user_values, mu, sigma) #normalization of
user values
    sol = forward_propagate(user_values, thetaTrueMin1, thetaTrueMin2)[4]
    print("Is your pokemon legendary?: " + str(sol[0, 0] > 0.7) + "\n")

#print(check.checkNNGradients(backdrop, 0))
```

```
from ML_UtilsModule import Data_Management
from scipy.optimize import minimize as sciMin
from sklearn.preprocessing import PolynomialFeatures as pf
import numpy as np
from matplotlib import pyplot as plt
from ML_UtilsModule import Normalization


lambda_ = 1
NUM_TRIES = 5

def g(z):
    """
    1/ 1 + e ^ (-0^T * x)
    """
    return 1/(1 + np.exp(-z))

def derivada_de_G(z):
    result = g(z) * (1 - g(z))
    return result

def polinomial_features(X, grado):
    poly = pf(grado)
    return (poly, poly.fit_transform(X))

def pesos_aleat(L_in, L_out):
    pesos = np.random.uniform(-0.12, 0.12, (L_out, 1+L_in))

    return pesos

def transform_y(y, num_etiquetas):
    y = np.reshape(y, (np.shape(y)[0], 1))
    mask = np.empty((num_etiquetas, np.shape(y)[0]), dtype=bool)
    for i in range( num_etiquetas):
        mask[i, :] = (y[:, 0] == i)

    mask = mask * 1

    return np.transpose(mask)

def des_transform_y(y, num_etiquetas):
    deTransY = np.where(y == 1)
    return deTransY[1]


def divideRandomGroups(X, y):
```

```
    X, y = Data_Management.shuffle_in_unison_scary(X, y)
    # ------------------------------------------------------------------------------
----------------------
    percent_train = 0.6
    percent_valid = 0.2
    percent_test = 0.2
    # ------------------------------------------------------------------------------
----------------------
    # TRAINIG GROUP
    t = int(np.shape(X)[0]*percent_train)
    trainX = X[:t]
    trainY= y[:t]
    # ------------------------------------------------------------------------------
----------------------
    # VALIDATION GROUP
    v=int( np.shape(trainX)[0]+np.shape(X)[0]*percent_valid)
    validationX = X[np.shape(trainX)[0] : v]
    validationY= y[np.shape(trainY)[0] : v]
    # ------------------------------------------------------------------------------
----------------------
    # TESTING GROUP
    testingX = X[np.shape(trainX)[0]+np.shape(validationX)[0] :]
    testingY= y[np.shape(trainY)[0]+np.shape(validationY)[0] :]

    return trainX, trainY, validationX, validationY, testingX, testingY


def J(X, y, a3, num_etiquetas, theta1, theta2):
    m = np.shape(X)[0]
    aux1 = -y * (np.log(a3))
    aux2 = (1 - y) * (np.log(1 - a3))
    aux3 = aux1 - aux2
    aux4 = np.sum(theta1**2) + np.sum(theta2**2)
    print (aux4)
    return (1/m) * np.sum(aux3) + (lambda_/(2*m)) * aux4

def forward_propagate(X, theta1, theta2):
    m = X.shape[0]
    a1 = np.hstack([np.ones([m, 1]), X])
    z2 = np.dot(a1, theta1.T)
    a2 = np.hstack([np.ones([m, 1]), g(z2)])
    z3 = np.dot(a2, theta2.T)
    h = g(z3)
    return a1, z2, a2, z3, h

def propagation(a1, theta1, theta2):
    a1 = Data_Management.add_column_left_of_matrix(a1)
    a2 = g(np.dot(a1, np.transpose(theta1)))
    a2 = Data_Management.add_column_left_of_matrix(a2)
```

```
    a3 = g(np.dot(a2, np.transpose(theta2)))

    return a1, a2, a3

def backdrop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg):
    """
    return coste y gradiente de una red neuronal de dos capas
    """
    theta1 = np.reshape(params_rn[:num_ocultas*(num_entradas + 1)],
        (num_ocultas, (num_entradas + 1)))
    theta2 = np.reshape(params_rn[num_ocultas*(num_entradas + 1):],
        (num_etiquetas, (num_ocultas + 1)))

    #--------------------PASO1----------------------------------------

    a1, a2, a3 = propagation(X, theta1, theta2)
    m = np.shape(X)[0]
    delta_3 = a3 - y # (5000, 10)
    #--------------------PASO2----------------------------------------
    #delta_3 = a3 - y # (5000, 10)
    delta_matrix_1 = np.zeros(np.shape(theta1))
    delta_matrix_2 = np.zeros(np.shape(theta2))

    aux1 = np.dot(delta_3, theta2) #(5000, 26)
    aux2 = Data_Management.add_column_left_of_matrix(derivada_de_G(np.dot(a1, np.tra
nspose(theta1))))
    delta_2 = aux1 * aux2 #(5000, 26)
    delta_2 = np.delete(delta_2, [0], axis=1) #(5000, 25)

    # #--------------------PASO4----------------------------------------

    delta_matrix_1 = delta_matrix_1 + np.transpose(np.dot(np.transpose(a1), delta_2)
) #(25, 401)
    delta_matrix_2 = delta_matrix_2 + np.transpose(np.dot(np.transpose(a2), delta_3)
) #(10, 26)
    #--------------------PASO6----------------------------------------
    delta_matrix_1 = (1/m) * delta_matrix_1
    delta_matrix_1[:, 1:] = delta_matrix_1[:, 1:] + (reg/m) * theta1[:, 1:]

    delta_matrix_2 = (1/m) * delta_matrix_2
    delta_matrix_2[:, 1:] = delta_matrix_2[:, 1:] + (reg/m) * theta2[:, 1:]


    cost = J(X, y, a3, num_etiquetas, theta1, theta2)
    gradient = np.concatenate((np.ravel(delta_matrix_1), np.ravel(delta_matrix_2)))

    return cost, gradient
```

```
def checkLearned(y, outputLayer):
    maxIndexV = np.argmax(outputLayer, axis = 1)
    checker = (y[:] == maxIndexV)

    truePositives = 0
    falsePositives = 0
    falseNegatives = 0

    for i in range(np.shape(y)[0]):
        for j in range(18):
            if maxIndexV[i] == j and y[i] == j:
                truePositives += 1
                break
            elif maxIndexV[i] == j and y[i] != j:
                falsePositives += 1
                break
            elif maxIndexV[i] != j and y[i] == j:
                falseNegatives += 1
                break

    if truePositives == 0:
        return 0

    recall = (truePositives/(truePositives + falseNegatives))
    precision = (truePositives/(truePositives + falsePositives))
    score = 2 *(precision*recall/(precision + recall))

    # PORCENTAJE DE ACIERTOS TOTALES
    #count = np.size(np.where(checker[:, 0] == cy[:, 0]))
    #fin = count/np.shape(y)[0] * 100

    return score

def pintaTodo(X, y, error, errorTr, true_score):
    plt.figure()
    #plt.ylim(0,1)
    plt.plot(np.linspace(0,len(error)-
1, len(error), dtype = int), error[:], color="grey")
    plt.plot(np.linspace(0,len(errorTr)-
1, len(errorTr), dtype = int), errorTr[:], color="green")
    plt.suptitle(("Score: " + str(true_score)))

    plt.show()

def paint_pkmTypes(X, y, types = None):
    '''
    Creates plt figure and draws the pkms used as input by the first two values of X
    and changes the color of them depending on the type
```

```
    X = attributes
    y = list of types, deTransformed (from 0 to 17)
    types = types that are going to be printed
    '''
    typesIndx = []

    if(types == None):
        types = Data_Management.types_
        # esto es para que meta los indices del 0 al 17, de los tipos completos, com
o no lo conseguia, he pasado
        typesIndx = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]
    else:
        for t in range(len(types)):
            typesIndx.append(Data_Management.types_.index(types[t]))


    colors = Data_Management.colors_

    for i in range(len(typesIndx)):
        pos = (y == typesIndx[i]).ravel()
        plt.scatter(X[pos, 0], X[pos, 1], color=colors[typesIndx[i]], marker='.', la
bel = types[i])


def paint_graphic(X, y, true_score, theta1, theta2, types = None):
    plt.figure()

    paint_pkmTypes(X, y, types)

    x0_min, x0_max = X[:,0].min(), X[:,0].max()
    x1_min, x1_max = X[:,1].min(), X[:,1].max()
    xx1, xx2 = np.meshgrid(np.linspace(x0_min, x0_max), np.linspace(x1_min, x1_max))

    aux = np.c_[ xx1.ravel(), xx2.ravel()]

    sigm = forward_propagate( aux, theta1, theta2)[4]
    sigm = np.argmax(sigm, axis = 1) #coge el valor maximo sacado por el frw_Propaga
te
    sigm = np.reshape(sigm, np.shape(xx1))
    plt.contour(xx1, xx2, sigm, linewidths = 0.25, colors = 'b')


    plt.xlabel("weight_kg")
    plt.ylabel("speed")
    plt.suptitle(("Score: " + str(true_score)))

    plt.show()
```

```
def paint_graphic_norm_full(sigm, xx1, xx2):
    """
    Pinta las funciones de todos los tipos del color de cada tipo
    """
    for i in range(0,18):
        contorn = sigm[:, i]
        contorn = np.reshape(contorn, np.shape(xx1))
        #plt.contour(xx1, xx2, contorn, linewidths = 0.5, colors=Data_Management.col
ors_[i])
        CS = plt.contourf(xx1, xx2, contorn)
        #plt.clabel(CS, fontsize=8, colors='black')
        #sscbar = plt.colorbar(CS)

def paint_graphic_norm_partial(sigm, xx1, xx2, types = None):
    """
    Pinta las funciones de los tipos elegidos del color de cada tipo
    """
    typesIndx = []
    for t in range(len(types)):
        typesIndx.append(Data_Management.types_.index(types[t]))

    for i in range(len(typesIndx)):
        contorn = sigm[:, typesIndx[i]]
        contorn = np.reshape(contorn, np.shape(xx1))
        #plt.contour(xx1, xx2, contorn, linewidths = 0.5, colors=Data_Management.col
ors_[typesIndx[i]])
        CS = plt.contourf(xx1, xx2, contorn, cmap='BrBG')
        #plt.clabel(CS, fontsize=8, colors='black')
        #cbar = plt.colorbar(CS)


def paint_graphic_norm(X, y, true_score, theta1, theta2, mu, sigma, graphic_attr_nam
es, types = None, show_allTypes = False):
    figure, ax = plt.subplots()

    x0_min, x0_max = X[:,0].min(), X[:,0].max()
    x1_min, x1_max = X[:,1].min(), X[:,1].max()
    xx1, xx2 = np.meshgrid(np.linspace(x0_min, x0_max), np.linspace(x1_min, x1_max))

    aux = np.c_[ xx1.ravel(), xx2.ravel()]
    #p, aux = polinomial_features(aux, 2)
    #aux = Normalization.normalize(aux[:, 1:], mu, sigma)
    sigm = forward_propagate(aux, theta1, theta2)[4]

    if types == None:
        paint_graphic_norm_full(sigm, xx1, xx2)
    else:
        paint_graphic_norm_partial(sigm, xx1, xx2, types)
```

```
    #formatting the graphic with some labels
    plt.xlabel(graphic_attr_names[0])
    plt.ylabel(graphic_attr_names[1])
    plt.suptitle(("Score: " + str(float("{0:.3f}".format(true_score)))))

    #set the labels to non-normalized values
    figure.canvas.draw()
    labels = [item for item in plt.xticks()[0]]
    for i in range(len(labels)):
        labels[i] = int(round((labels[i] * sigma[0, 0]) + mu[0, 0], -1))
    ax.xaxis.set_ticklabels(labels)

    labels = [item for item in plt.yticks()[0]]
    for i in range(len(labels)):
        labels[i] = int(round((labels[i] * sigma[0, 1]) + mu[0, 1], -1))
    ax.yaxis.set_ticklabels(labels)

    if show_allTypes:
        paint_pkmTypes(X, y)
    else:
        paint_pkmTypes(X, y, types)

    figure.legend()

    plt.show()


attr_names = ["percentage_male", "sp_attack"]
types_to_paint =["fire"]
X, y = Data_Management.load_csv_types_features("pokemon.csv", attr_names)



#normalize
X, mu, sigma = Normalization.normalize_data_matrix(X)



num_entradas = np.shape(X)[1]
num_ocultas = 25
num_etiquetas = 18
true_score_max = float("-inf")

thetaTrueMin1 = None
thetaTrueMin2 = None

y_transformed =  transform_y(y, num_etiquetas)
```

```
trainX, trainY, validationX, validationY, testingX, testingY = divideRandomGroups(X,
 y_transformed)

for j in range(NUM_TRIES):
    theta1 = pesos_aleat(num_entradas, num_ocultas)
    theta2 = pesos_aleat(num_ocultas, num_etiquetas)

    theta_vector = np.concatenate((np.ravel(theta1), np.ravel(theta2)))

    auxErr = []
    auxErrTr = []
    thetaMin1 = None
    thetaMin2 = None
    errorMin = float("inf")

    for i in range(1, np.shape(trainX)[0]):
        thetas = sciMin(fun=backdrop, x0=theta_vector,
        args=(num_entradas, num_ocultas, num_etiquetas, trainX[:i], trainY[:i], lamb
da_),
        method='TNC', jac=True,
        options={'maxiter': 70}).x

        theta1 = np.reshape(thetas[:num_ocultas*(num_entradas + 1)],
                (num_ocultas, (num_entradas + 1)))
        theta2 = np.reshape(thetas[num_ocultas*(num_entradas + 1):],
                (num_etiquetas, (num_ocultas + 1)))

        auxErr.append(J(validationX, validationY, forward_propagate(validationX, the
ta1, theta2)[4], num_etiquetas, theta1, theta2))
        auxErrTr.append(J(trainX[:i], trainY[:i], forward_propagate(trainX[:i], thet
a1, theta2)[4], num_etiquetas, theta1, theta2))

        if errorMin > auxErr[-1]:
            errorMin = auxErr[-1]
            thetaMin1 = theta1
            thetaMin2 = theta2


    true_score = checkLearned(y, forward_propagate(X, thetaMin1, thetaMin2)[4])
    if true_score > true_score_max:
        true_score_max = true_score
        thetaTrueMin1 = thetaMin1
        thetaTrueMin2 = thetaMin2
        #pintaTodo(testingX, testingY, auxErr, auxErrTr, true_score)

deTransTestY = des_transform_y(testingY, num_etiquetas)
# Pintado ------------------------------------------------------------------------
-----------------------------------------------
```

```
# paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTru
eMin2, mu, sigma, attr_names, ["bug", "normal", 'flying', 'ghost'], True)
# paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTru
eMin2, mu, sigma, attr_names, ["bug", "normal", 'flying', 'ghost'])

paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTrueM
in2, mu, sigma, attr_names, ["bug"])
paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTrueM
in2, mu, sigma, attr_names, ["normal"])
paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTrueM
in2, mu, sigma, attr_names, ["water"])
paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTrueM
in2, mu, sigma, attr_names, ["fire"])
paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTrueM
in2, mu, sigma, attr_names, ["grass"])
# paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTru
eMin2, mu, sigma, attr_names, ["flying"])
# paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTru
eMin2, mu, sigma, attr_names, ["ghost"])

# paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTru
eMin2, mu, sigma, attr_names, ["bug"], True)
# paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTru
eMin2, mu, sigma, attr_names, ["normal"], True)
# paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTru
eMin2, mu, sigma, attr_names, ["flying"], True)
# paint_graphic_norm(testingX, deTransTestY, true_score_max, thetaTrueMin1, thetaTru
eMin2, mu, sigma, attr_names, ["ghost"], True)
# ------------------------------------------------------------------------------
-----------------------------------------------

print("True Score de la red neuronal: " + str(true_score_max) + "\n")
while True:
    user_values = np.array(list(map(float, input("Gimme stats: ").split())), dtype=f
loat) # (features, )
    if user_values.size == 0:
        break
    user_values = np.reshape(user_values, (np.shape(user_values)[0], 1))
    user_values = np.transpose(user_values)
    user_values = Normalization.normalize(user_values, mu, sigma) #normalization of
user values
    sol = forward_propagate(user_values, thetaTrueMin1, thetaTrueMin2)[4]
    print("Is your pokemon legendary?: " + str(sol[0, 0] > 0.7) + "\n")
```

Boruta

```
from sklearn.ensemble import RandomForestClassifier
import numpy as np
from ML_UtilsModule import Data_Management, Normalization
from boruta import BorutaPy

# load X and y
# NOTE BorutaPy accepts numpy arrays only, hence the .values attribute
X, y = Data_Management.load_csv_types_features("pokemon.csv", ["hp", "attack", "defe
nse", "sp_attack", "sp_defense","speed", "height_m", "weight_kg"
    , "percentage_male", "generation"])
y = y.ravel()

# define random forest classifier, with utilising all cores and
# sampling in proportion to y labels
rf = RandomForestClassifier(n_jobs=-1, class_weight='balanced', max_depth=5)

# define Boruta feature selection method
feat_selector = BorutaPy(rf, n_estimators='auto', verbose=2, random_state=1)

# find all relevant features - 5 features should be selected
feat_selector.fit(X, y)

# check selected features - first 5 features are selected
feat_selector.support_

# check ranking of features
feat_selector.ranking_

# call transform() on X to filter it down to selected features
X_filtered = feat_selector.transform(X)
```

Keras

```
import os
import numpy as np
np.random.seed(0)
from sklearn import datasets
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from keras.callbacks import Callback
from keras import metrics
from ML_UtilsModule import Data_Management

epoch_count=0
count=0

def plot_decision_boundary(X, y, model,epoch_count,count,steps=1000, cmap='Paired'):
    """
    Function to plot the decision boundary and data points of a model.
    Data points are colored based on their actual label.
    """
    cmap = plt.get_cmap(cmap)

    # Define region of interest by data limits
    xmin, xmax = X[:,0].min() - 1, X[:,0].max() + 1
    ymin, ymax = X[:,1].min() - 1, X[:,1].max() + 1
    steps = 1000
    x_span = np.linspace(xmin, xmax, steps)
    y_span = np.linspace(ymin, ymax, steps)
    xx, yy = np.meshgrid(x_span, y_span)

    # Make predictions across region of interest
    labels = model.predict(np.c_[xx.ravel(), yy.ravel()])

    # Plot decision boundary in region of interest
    z = labels.reshape(xx.shape)

    fig, ax = plt.subplots()
    ax.contourf(xx, yy, z, cmap=cmap, alpha=0.5)
    fig.suptitle("Epoch: "+str(epoch_count), fontsize=10)
    # Get predicted labels on training data and plot
    train_labels = model.predict(X)
    ax.scatter(X[:,0], X[:,1], c=y, cmap=cmap, lw=0)
    plt.show()
    return epoch_count

def plot_loss_accuracy(history):
    plt.figure()
```

```
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')
    plt.show()

# Keras callback to save decision boundary after each epoch
class prediction_history(Callback):
    def __init__(self):
        self.epoch_count=epoch_count
        self.count=count
    def on_epoch_end(self,epoch,logs={}):
        if self.epoch_count%100==0 or self.epoch_count==399:
            plot_decision_boundary(X, y, model,self.epoch_count,self.count,cmap='RdB
u')
            score, acc = model.evaluate(X, y, verbose=0)
            print("error " + str(score) + ", accuracy " + str(acc))
            self.count=self.count+1
        self.epoch_count=self.epoch_count+1
        return self.epoch_count



if __name__ == "__main__":
    X, y = datasets.make_moons(n_samples=1000, noise=0.1, random_state=0)
    X, y = Data_Management.load_csv_svm("pokemon.csv", ['weight_kg', 'height_m'])

    X, y, trainX, trainY, validationX, validationY, testingX, testingY = Data_Manage
ment.divide_legendary_groups(X, y)

    y  =  y.ravel()
    trainY = trainY.ravel()
    # Create a directory where image will be saved
    os.makedirs("images_new", exist_ok=True)

    # Define our model object
    model = Sequential()
    # Add layers to our model
    model.add(Dense(units=25, input_shape=(2,), activation="sigmoid", kernel_initial
izer="glorot_uniform"))
    model.add(Dense(units=25, activation="sigmoid", kernel_initializer="glorot_unifo
rm"))
    model.add(Dense(units=1, activation="sigmoid", kernel_initializer="glorot_unifor
m"))
    sgd = SGD(lr=0.1)
    # Compile model
```

```
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']
)
    predictions=prediction_history()
    history = model.fit(X, y, validation_split = 0.1, verbose=0,epochs=400, shuffle=
True,callbacks=[predictions])

    plot_loss_accuracy(history)
```