Universidad Internacional de La Rioja

Escuela Superior de Ingeniería y Tecnología

Grado en Matemática Computacional

Creación de una aplicación para predecir resultados de combates Pokémon utilizando Machine Learning

Trabajo fin de estudio presentado por:	Pablo Trull Báguena
Tipo de trabajo:	3. Aplicación práctica
Director/a:	Maria Antonella Gieco
Fecha:	13/07/2023

Resumen

En este trabajo, crearemos una aplicación capaz de predecir el resultado de combates Pokémon utilizando machine learning. Para ello estudiaremos las propiedades de diversos métodos y plantearemos como abordar los diversos problemas que se presentan. Durante el trabajo realizaremos diversos ejemplos y prototipos para entender mejor algunos de los conceptos importantes de cada modelo y discutiremos los diversos enfoques que se le pueden dar a este problema. Para realizar nuestra aplicación realizaremos un estudio sobre el rendimiento de los diversos métodos aplicados a nuestro problema para decidir el más adecuado.

Para poder comprender el problema, este trabajo también explicará diversas mecánicas de los combates pokémon y discutirá su importancia en la realización de esta aplicación. Finalmente, implementaremos la aplicación y discutiremos las diversas opciones que existen durante la creación del proyecto y el trabajo futuro.

Abstract

In this project, we will develop a program using machine learning that predicts the outcome of a pokémon combat. To do so, we will initially study the properties of several models and the tackling of the problems found in the process. In order to better grasp some of the concepts described, we will use various prototypes which are important for each model that will later aid in discussing the different approaches for this problem. Throughout the creation of the program, the efficiency of the different models will be taken into account when choosing the most appropriate one.

On the other hand, we will also focus on the mechanics of the Pokémon combats and discuss its importance in the creation process. At last, the program will be implemented and its future applications will also be discussed.

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción y conceptos básicos

1.1 Justificación

El aprendizaje automático o machine learning es una rama de la inteligencia artificial que se centra en el desarrollo de diversas técnicas cuya finalidad es que un ordenador sea capaz de aprender de un conjunto de datos que se le aporta. Esta clase de programas son útiles cuando el programador no es capaz de predecir todas las situaciones que el programa va a tener que analizar en el futuro, por ejemplo, un programa que tenga que trabajar con elementos financieros tiene que ser capaz de seguir funcionando incluso con las constantes fluctuaciones del mercado. Otro caso en el que es interesante poder hacer que el programa aprenda de ejemplos es cuando el programador no sabe muy bien como programar una solución al problema propiamente dicho, esto se puede ver, por ejemplo, en programas capaces de separar imágenes de gatos y de perros. Un proceso que, aunque las personas realizan de forma innata, no es fácil de implementar para que sea realizada por un ordenador.

En este trabajo estudiaremos la creación de una aplicación de *machine learning* para la predicción de resultados entre combates competitivos *Pokémon* [1] individuales.

Además de ser una de las mayores franquicias de videojuegos que existe hoy en día, *Pokémon* también es uno de los videojuegos más jugados a nivel competitivo. *Smogon* [2] es una de las mayores comunidades de *Pokémon* competitivo que proporciona un entorno para combates pokémon de modalidad individual siguiendo una separación en diferentes *tiers,* que corresponden a divisiones o categorías en las que se engloban a los pokémon según su poder y uso en el ámbito competitivo (ver glosario). Esta comunidad se desarrolla en paralelo al simulador de combate *Pokémon Showdown* [3] donde se realizan los combates y torneos. Existen dos modalidades diferentes de combates competitivos pokémons: combates dobles y combates individuales.

- Los combates dobles son gestionados por *Pokémon Company* y se juegan mayoritariamente usando hardware y software oficial de Nintendo. En los combates dobles cada entrenador dispone de dos pokémons en combate simultáneamente.
- En los combates individuales cada entrenador solo cuenta con un pokémon a la vez en combate. Pese a que es posible jugar combates individuales en el software original, *Smogon* es la mayor comunidad competitiva de este tipo de modalidad, con normas diferentes a las que existen en las competiciones oficiales de Pokémon Company,

realizando todos sus torneos y partidas mediante el simulador *Pokémon Showdown*. La comunidad de combates individuales es muy grande, por lo que los torneos oficiales suelen tener cientos de participantes.

En este trabajo me centraré en los combates individuales siguiendo las reglas de *Smogon*. Estos torneos se suelen jugar a lo largo de varias semanas, incluyendo uno o dos sets por semana donde cada set se juega al mejor de tres partidas. Puesto que cada partida utiliza un equipo diferente, hay que preparar entre tres y seis equipos por semana. La preparación de cada equipo es un proceso largo que requiere muchas pruebas y horas de uso de una calculadora especializada [4] para decidir como criar cada pokémon (éste es un proceso largo que explicaremos de forma rápida más adelante cuando planteemos el problema en sí).

Cada equipo está formado por 6 pokémons diferentes y en el combate individual se enfrentan un pokémon de cada equipo cada vez. En cada turno podemos elegir si cambiar nuestro pokémon o atacar. Cuando un pokémon pierde abandona la partida y debe ser substituido por otro pokémon del equipo.

Una aplicación capaz de predecir el vencedor en un combate de dos pokémons (uno contra uno) sería de gran ayuda en la construcción de estos equipos ya que ahorraría una gran cantidad de tiempo de hacer cálculos y pruebas.

Este problema puede abordarse desde la perspectiva del *machine learning* porque los valores con los que vamos a trabajar van a ir variando constantemente. Dado que *Pokémon* es una franquicia enorme que saca uno o dos juegos al año, el problema con el que queremos trabajar también variará significativamente con el lanzamiento de cada nuevo juego ya que cada uno de éstos introduce nuevos pokémons y también introduce cambios en pokémons y objetos antiguos (ver glosario).

En combates individuales competitivos existen diversas *tiers* o categorías en cada una de las cuales solo se pueden usar ciertos pokémons específicos, los cuales varían cada mes dependiendo del porcentaje de uso global de cada pokémon en cada *tier*. Teniendo en cuenta que existen actualmente más de 1000 pokémons y muchos de ellos poseen mecánicas propias, programar una aplicación para predecir los resultados de combates uno contra uno calculando el daño proporcionado por cada ataque y teniendo en cuenta todos los ataques, habilidades, objetos y *stats* (ver glosario) de cada pokémon supone un trabajo muy largo y costoso. Además, las condiciones pueden cambiar mucho con el lanzamiento de un nuevo juego o la prohibición de uso de algún Pokémon o ataque en cada *tier*. Eso sin contar que no todos los ataques de

pokémon hacen daño directo, puesto que existe una clase de ataques, por ejemplo, llamados ataques de estado, que pueden paralizar o envenenar al objetivo.

A partir de lo expuesto queda claro que implementar un programa para predecir el resultado basándose solo en el cálculo de daños debería imitar la toma de decisiones de un jugador de alto nivel en cada enfrentamiento entre dos pokémon específicos, lo cual no resulta muy factible.

En este proyecto partiré de la idea de que un programa que tome como ejemplos los resultados de un cierto número de combates, de los cuales se le especifica quien es el ganador, e intente predecir el resultado de otro combate mediante comparación con los ejemplos que ya tiene, podría aportar una solución mucho más factible, elegante y fácil de mantener.

1.2 Conceptos básicos

Introducimos en primer lugar las bases del problema a tratar explicando las características de los pokémon antes de tratar los detalles sobre la implementación del problema planteado.

Cada pokémon posee una serie de características, resumidas en la siguiente tabla:

Tabla 1. *Esquema de las características de un pokémon*

Tipo	Cada pokémon posee uno o dos (18 tipos diferentes)		
Stats	HP	Attack	Defense
	Speed	Special attack	Special defense
Ataque	Físico	Especial	De estado
Habilidad	Cada pokémon tiene una habilidad que depende de la raza		
Objeto	Se puede llevar equipado un objeto al combate		

Fuente: Elaboración propia

1- TIPO

Los tipos constituyen una de las características principales de cada pokémon. Existen 18 tipos diferentes y la efectividad de un ataque depende del tipo del ataque propio y del tipo del oponente. El daño producido o recibido se mide a través de un parámetro que cuantifica la eficacia de cada tipo, como muestra la figura 1.

Por ejemplo; si tu oponente es tipo *fuego* y le atacas con tipo *agua* le haces el doble de daño mientras que si le atacas con acero solo le haces la mitad de daño.

(ver más detalles en el glosario)

Figura 1. *Tabla de eficacia de tipos*

Efectividad	Tipo del Pokémon del oponente																	
	ACERO	AGUA	BICHO	DRAGÓN	ELÉCTRICO	FANTASMA	FUEGO	HADA	HIELO	LUCHA	NORMAL	PLANTA	PSÍQUICO	ROCA	SINIESTRO	TIERRA	VENENO	VOLADOR
TIPO DEL ATAQUE PROPIO: ACERO	½	½			½		½	x2	x2					x2				
AGUA		½		½			x2					½		x2		x2		
BICHO	½					½	½	½		½		x2	x2		x2		½	½
DRAGÓN	½			x2				0										
ELÉCTRICO		x2		½	½							½				0		x2
FANTASMA						x2					0		x2		½			
FUEGO	x2	½	x2	½			½		x2			x2		½				
HADA	½			x2			½			x2					x2		½	
HIELO	½	½		x2			½		½			x2				x2		x2
LUCHA	x2		½			0		½	x2		x2		½	x2	x2		½	½
NORMAL	½					0								½				
PLANTA	½	x2	½	½			½					½		x2		x2	½	½
PSÍQUICO	½									x2			½		0		x2	
ROCA	½		x2				x2		x2	½						½		x2
SINIESTRO						x2		½		½			x2		½			
TIERRA	x2		½		x2		x2					½		x2			x2	0
VENENO	0					½		x2				x2		½		½	½	
VOLADOR	½		x2		½					x2		x2		½				

Fuente: Pagina de tipo en Wikidex [5]

2- STATS

Los stats son los seis factores principales que determinan cómo se desempeñará un pokémon en un combate: HP, ataque (attack), defensa (defense), ataque especial (special attack), defensa especial (special defense) y velocidad (speed). Constituyen la segunda característica importante de un pokémon.

A cada pokémon se le asigna un cierto valor numérico de cada stat. Este valor depende de su valor base, de su nivel, de su naturaleza, de los *Effort Values* (EVs) y de los *Individual Values* (IVs) y se calcula a través de la fórmula (ecuación (9)). Todos los pokémons están a nivel 100 en nivel competitivo individual.

- En Pokémon competitivo prácticamente todos los IVs se utilizan al máximo, que sería 31, excepto en casos muy específicos que no trataremos en este trabajo.

- En combates competitivos individuales se dispone de un total de 508 EVs que se asignan a los pokémons con un máximo de 252 en un *stat* específico. Cada 4 EVs aumentan una unidad el respectivo *stat* del pokémon.
- En cuanto a las naturalezas, cada una mejora un stat especifico, pero disminuye otro.

En muchos casos los pokémons suelen construirse de forma que cumplan un rol especifico. Por ejemplo, en la segunda imagen de la figura 2 podemos ver un *Pikachu* ofensivo en el que hemos invertido el máximo de EVs en ataque y velocidad mientras que en la tercera tenemos un *Pikachu* más defensivo en el que hemos invertido el máximo en vida y en defensa. Por lo tanto, como se ve en las imágenes, cada stat del pokémon solo puede variar dentro de un rango especifico lo cual hace que las comparaciones con otros pokémons que comparten ciertas características sean bastante asequibles.

Figura 2. *Tres ejemplos de stats de un Pikachu: uno sin EV, uno criado para ser ofensivo y otro para ser defensivo*

Fuente: Team Buider de Pokémon Showdown

3- ATAQUES

La tercera característica que determina a un pokémon son sus ataques. Existen tres tipos de ataques: ataques físicos, ataques especiales y ataques de estado.

- Los ataques físicos utilizan el *stat* de ataque para calcular el daño que se realiza y la defensa para calcular la mitigación.
- Los ataques especiales utilizan los *stats* de ataque y defensa especiales de forma análoga.
- Los ataques de estado, en cambio, no realizan daño directo, sino que causan diversos efectos como curar al usuario o envenenar al rival.

Todos los ataques pertenecen a un tipo especifico lo que determina el daño que hacen dependiendo de las efectividades contra el tipo del pokémon rival según la tabla que hemos visto en la Figura 1. Si el tipo del pokémon coincide con el tipo del ataque que utiliza, éste recibe un STAB (Same Type Atack Bonus) lo que hará que el ataque produzca un 50% de daño adicional.

4- HABILIDADES

Finalmente, cada pokémon puede elegir una habilidad de las varias que puede poseer y un objeto que puede llevar. Estas habilidades están determinadas por la raza del pokémon y pueden tener una multitud de efectos, desde efectos perjudiciales para el pokémon hasta hacerlo inmune a cualquier ataque que no sea súper eficaz. **Muchas de estas habilidades son únicas para una raza de pokémon lo cual hace de esta característica la más difícil de abordar para nuestro programa.** En cambio, la mayoría de objetos no están limitados por ninguna característica del pokémon por lo que normalmente se utilizan un número reducido de ellos que aportan unos beneficios mejores.

Tras analizar los aspectos relevantes de la crianza de pokémon es importante remarcar que en Pokémon competitivo los pokémons que se consideran mejores o más consistentes tienen un uso mucho mayor que los demás y lo mismo aplica a las diversas distribuciones de *stats* y a los ataques. Por ejemplo, un pokémon con muy buena defensa de base, pero con poca velocidad probablemente no tenga ningún EV en stats ofensivos mientras que un pokémon con mucho ataque especial y una buena velocidad tendrá muchos EVs en estos *stats* y ninguno en los demás.

Una vez definidas las bases de nuestro problema debemos plantearnos en qué modo podemos implementar un programa de *machine learning* que aprenda de los ejemplos que le aportamos para decidir cuales resultaran relevantes para este problema y predecir resultados de un combate.

Existen tres maneras principales de aprendizaje para un programa:

- **Aprendizaje supervisado**: en este modelo de aprendizaje el algoritmo genera una función que relaciona el input y el output de los ejemplos. Este tipo de método es muy usado en programas de clasificación.

- **Aprendizaje no supervisado**: en este método de aprendizaje el programa recibe como ejemplos solo un input. Su trabajo será encontrar los patrones que relacionen los diversos ejemplos y agruparlos en clústeres.

- **Aprendizaje por refuerzo**: en este modelo de aprendizaje el programa recibe un feedback por cada acción que realiza. Es decir, recibe un premio o un castigo cuando realiza una acción, y el programa tiene que intentar maximizar los premios que recibe. Este tipo de modelo es muy utilizado para enseñar a un programa a jugar a diversos juegos de mesa, por ejemplo.

La propuesta para este trabajo es conseguir desarrollar una aplicación capaz de predecir el resultado de combates entre dos pokémons basándose en ejemplos de otros combates con resultados que nosotros hemos aportado previamente al programa. Por lo tanto, estaríamos trabajando con un modelo de aprendizaje supervisado. La idea es que el programa sea capaz de comparar las características de los pokémons que están peleando y sea capaz de realizar una predicción acertada utilizando métodos de *forecasting* que se utilizan para la predicción de resultados en algunos deportes y juegos de mesa.

Es interesante remarcar que podríamos obtener el mismo resultado realizando un programa con aprendizaje por refuerzo que fuera capaz de reproducir el propio combate, siendo capaz de jugar como un jugador de alto nivel, pero, como veremos en el apartado de contexto del arte, nadie ha conseguido una implementación real de este programa.

Para el desarrollo de esta aplicación existen diversos lenguajes que disponen de paquetes que incluyen las herramientas necesarias para la realización de estos algoritmos. En este trabajo estudiaré diversos paquetes de Python y de R para analizar los diferentes enfoques con los que quiero trabajar hasta dar con el que aporta mejores resultados.

2. Contexto y Estado del arte

No existen muchos estudios sobre predicción de resultados de combates Pokémon, pero sí que existen diversos notebooks en *Kaggle* [6] de personas que han intentado aplicar diversos

métodos de *machine learning* para poder predecir el resultado de combates más simplificados de lo que queremos estudiar en este trabajo.

Aunque no existan trabajos similares al que planteamos sí que se han realizado diversos estudios que han aplicado diversas ramas de la inteligencia artificial al estudio de combates Pokémon. Muchos de estos estudios utilizan un modelo de aprendizaje por refuerzo para intentar conseguir un programa capaz de jugar de manera similar a la que jugaría un jugador de alto nivel. Si existiera alguno de estos programas que funcionase al nivel esperado sería posible llegar, simulando un combate completo turno por turno, a unos resultados similares a los que busca nuestro programa. Pero, de momento, ninguno de los algoritmos creados con esta finalidad ha conseguido acercarse al nivel deseado. Un equipo de estudiantes de la clase CS221 de Stanford [7] consiguieron implementar un programa que utilizaba *reinforced learning* que consiguió llegar a tener 1340 de ELO, resultado lejano al nivel que consideraríamos de un jugador de alto nivel que suele estar sobre los 1800 de ELO. Otra aplicación de la inteligencia artificial aplicado a Pokémon seria la extensión para navegador *Pokémon Battle Predictor* [8] que analiza el estado de la partida y tras compararlo con diversas repeticiones de otras partidas predice la probabilidad de cada movimiento por parte del rival.

Fuera del ámbito de Pokémon existen diversos modelos desarrollados por el equipo *DeepMind* Team de Google que han conseguido resultados sobre humanos en programas que utilizan aprendizaje por refuerzo para jugar al Go [9], al Shogi y al ajedrez. También existen diversos trabajos que aplican técnicas de *machine learning* para la predicción de resultados a diversos deportes profesionales como el futbol o el baloncesto.

3. Objetivos

3.1 Objetivos principales

El objetivo principal que quiero conseguir en este trabajo es la creación de una aplicación capaz de predecir los resultados de un combate Pokémon mediante un programa de machine learning de aprendizaje supervisado. Para ello me basaré en un **algoritmo de Árbol de Decisión**, otro de **Random Forest**, y uno de **Gradient Tree Boosting,** comparando los resultados y estudiando las diferencias y ventajas que aportan estos tres métodos. También generaré diversos datasets con diferentes formatos para estudiar cómo afectan cada uno de ellos al rendimiento de mi aplicación.

3.2 Objetivos secundarios

En este tipo de aplicaciones es muy importante encontrar una forma en la que nuestro modelo sea flexible y sencillo de modificar para adecuarnos a las diferentes variaciones que vamos a encontrarnos en el futuro. Para ello, me apoyaré en los resultados obtenidos en la aplicación de los diversos métodos y formatos para realizar un estudio teórico sobre cómo una gran cantidad de datos pueden afectar al rendimiento de nuestra aplicación en el futuro.

3.3 Principales dificultades

Para que un modelo de *machine learning* pueda aportar resultados suficientemente buenos de cara a su aplicación en un problema real necesita una gran cantidad de datos en los que poder basarse para la identificación de patrones de una forma adecuada. En este caso he creado mis propios *datasets* por lo que debemos intentar incluir un amplio número de ejemplos para que el programa pueda analizar en un futuro la mayor cantidad de casos posibles. Esto también incluye la creación de *datasets* más pequeños que podamos utilizar para hacer pruebas e ir ajustando el modelo. Dependiendo de si los datos que encontremos son suficientemente variados o no, los resultados del estudio estadístico que usamos para ajustar los parámetros de nuestra identificación de patrones pueden ser muy diferentes y pueden terminar no siendo lo suficientemente aptos para trabajar en un entorno con una cantidad de datos reales mucho más grande.

4. Descripción del modelo

4.1 Diseño del modelo

Existen diversos tipos de algoritmos de *machine learning* que se pueden utilizar para la creación de una aplicación de este estilo. He decidido realizar una implementación basada en árboles de decisión ya que se trata de un modelo relativamente fácil de adaptar a los diferentes casos que queremos implementar y es, en general, un modelo que consigue resultados aceptables de forma consistente. Antes de continuar con la implementación de la aplicación introduciré los modelos de Árbol de Decisión, del de Random Forest y del de Gradient Tree Boosting de manera que sea más fácil entender las explicaciones.

4.1.1 Árbol de Decisión, Random Forest y Gradient Boost Tree

Un **Árbol de Decisión** es una representación de una función en la que se conecta un vector de diferentes atributos a un output especifico que sería nuestra "decisión". Todo árbol de decisiones comienza en su raíz y se van realizando diversos tests sobre las propiedades del elemento con el que estamos trabajando mientras se avanza por las ramas del árbol hasta llegar a una hoja. Denominamos raíz el nodo inicial del árbol y llamamos hoja a un nodo del que no sale ninguna rama, es decir un nodo desde el que ya no existe ninguna posibilidad de tomar una decisión y que por lo tanto es el final del camino. **Una norma importante de un árbol de decisión es que cada hoja tiene que poder ser accesible solamente por un camino**. Esto no implica que no puedan existir hojas con la misma "decisión" sino que una vez te has desviado de un camino es imposible volver a él.

Claramente para un mismo problema existen muchos árboles de decisión que pueden funcionar por lo que existen diversas maneras de ajustar cada árbol dependiendo de los requisitos que queremos que cumpla. Podemos considerar algunos parámetros como el *overfitting*, *underfitting* y otras formas de optimizar un árbol. El overfitting, por ejemplo, sucede cuando un modelo tiene hojas a las que solo llegarán un número de casos muy escasos y normalmente se debe a casos atípicos en nuestros datos de entrenamiento del modelo. Podemos controlar este efecto ajustando el número de niveles del árbol mediante poda de forma que no tengamos un exceso de sobreajuste. En el prototipo presentado posteriormente realizaré un *árbol de decisión* basado en los resultados estadísticos obtenidos y estudiaré como decidir cuál de todas las posibilidades es mejor dependiendo del enfoque que queramos dar al problema.

El algoritmo de Random Forest, resuelve los problemas de optimización de un *árbol de decisión*. Este algoritmo consiste en utilizar árboles de decisión diferentes para un mismo problema y dar como solución el resultado que den la mayoría. Los árboles que se crean en este algoritmo son todos no correlacionados lo cual hace que este algoritmo tenga menos variación en los resultados. Una desventaja, que este algoritmo presenta en comparación a un árbol de decisión, es que resulta mucho más complicado interpretar los resultados y esto hace que pueda ser difícil de ajustar correctamente al problema. Otra desventaja sería que, al trabajar con variables categóricas, los resultados pueden variar dependiendo de la cantidad de niveles que tenga cada variable priorizando las variables con más niveles. Esta segunda desventaja es de particular interés para nosotros ya que vamos a trabajar con varias variables categóricas con una gran diferencia entre los niveles que tienen cada una.

Un modelo de **Gradient Tree Boosting** crea diversos arboles de forma secuencial con la intención de que cada uno sea capaz de solucionar los errores cometidos por el árbol previo. Este modelo suele ser más preciso que los modelos de Árbol de Decisión y Random Forest, pero puede resultar mucho más complicado analizar el modelo creado. Otra desventaja de este modelo es que dependiendo de los datos que se utilicen puede tender al overfitting, por lo que trabajar con un número mayor de árboles suele ser una mejor opción lo que genera un coste computacional mayor.

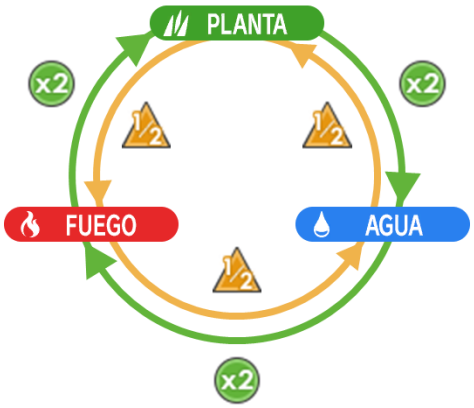
4.2 Primer Prototipo

Antes de empezar a trabajar con el problema completo vamos a realizar algunas pruebas usando una muestra con unos patrones muy definidos de forma que podamos ver si la forma de gestionar los datos del programa es la adecuada.

4.2.1 Criterios para la creación de un árbol de decisión apropiado para nuestro problema

Para este primer prototipo he seleccionado todos los pokémons iniciales de cada generación con sus líneas evolutivas. Un pokémon inicial es el que se le da a elegir al jugador como primer compañero al principio de cada juego, eligiendo entre tres pokémons. Estos pokémons siempre son: uno de tipo agua, uno de tipo fuego y uno de tipo planta. Estos tres tipos tienen una relación entre ellos que se conoce como triángulo de tipos perfecto lo cual hace de ellos un buen ejemplo para ver si el programa es capaz de deducir la eficacia de tipos. Esta relación de triángulo perfecto se muestra en la figura 3 y quiere decir que cada uno de esos tipos es fuerte ofensiva y defensivamente contra uno de los otros dos tipos, pero también es débil ofensiva y defensivamente contra el tercero. Esto hace que, utilizando solamente ataques de su tipo principal es muy difícil que un pokémon de tipo fuego pueda ganarle a uno de tipo agua, por ejemplo.

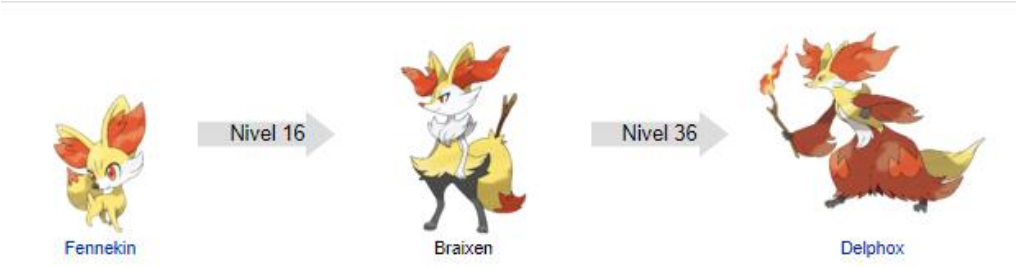
Figura 3. *Ventajas y desventajas entre los tres tipos iniciales*



Fuente: Pagina de Inicial en Wikidex [5]

Otra característica importante de estos pokémons es que todos forman parte de una cadena de evolución con tres estados. Estas cadenas evolutivas están formadas con una forma base, una evolución intermedia y una forma final, donde cada evolución mejora los stats del pokémon siguiendo normalmente una distribución similar en todas ellas, un ejemplo de estas evoluciones se puede ver en la figura 4.

Figura 4. *Línea evolutiva del Pokémon inicial Fennekin*



Fuente: Pagina de Braixen en Wikidex [5]

Otra razón importante que hace de estas líneas evolutivas un buen ejemplo a estudiar es que todos los pokémons pertenecientes a uno de los eslabones de la línea evolutiva tienen un BST (Base Stat Total) muy similar al que tienen los pokémons que se encuentran en el mismo eslabón de otras líneas evolutivas. El BST corresponde al número total que se obtiene al sumar los valores de todos los stats base del pokémon. Esto quiere decir que todos ellos tendrán una fuerza similar a pesar de que sus stats estén distribuidos de forma diferente, lo que nos dará una idea mejor sobre el efecto que tiene cada stat en diversos escenarios, ya que no nos basaremos simplemente en que uno de ellos es muchísimo más fuerte que los otros en diversos campos.

Existen un total de 9 generaciones donde cada una tiene tres líneas evolutivas de iniciales con tres pokémons cada una, lo que nos da un total de 81 pokémons, una muestra suficientemente grande para nuestro primer prototipo.

4.2.2 Preparación de los datos

Para elaborar este prototipo trabajaré solamente con los tipos y *stats* base, en lugar de trabajar con todos los aspectos de un pokémon definidos en la sección de introducción.

Defino dos datasets diferentes:

- Un primer fichero .csv, llamado *Datospok.csv,* almacena el nombre del pokémon, sus tipos y sus stats, además de asignarle una id interna para simplificar el funcionamiento del programa.

Figura 5. *Ejemplo de visualización de Datospok.csv*

ID	Nombre	Tipo1	Tipo2	HP	ATK	DEF	SP.ATK	SP.DEF	SPD
1	Bulbasaur	Grass	Poison	45	49	49	65	65	45
2	Ivysaur	Grass	Poison	60	62	63	80	80	60
3	Venusaur	Grass	Poison	80	82	83	100	100	80
4	Charmander	Fire		39	52	43	60	50	65

Fuente: Elaboración propia

- El segundo dataset contiene combates de un mismo pokémon contra una variedad de rivales diferentes. Este dataset debe contener los datos del primer pokémon, los datos del rival y el resultado. La idea es comprobar cómo estos datos afectarán a la aparición de patrones para poder generar nuestros árboles. Se espera que este formato aporte resultados mejores que el dataset de combates de diversos pokémons que también implementaremos en el trabajo final.

Puesto que la idea de este trabajo es acercarse lo más posible a una aplicación completamente funcional, ésta debería contener un archivo para los combates de cada pokémon. Sin embargo, escribir cada combate manualmente es un trabajo increíblemente largo de cara al problema final. Para agilizar la generación de estos ficheros he creado el programa *Prototipo1.py*.

Este programa realiza los siguientes pasos:

- Empieza importando los datos de Datospok.csv utilizando el paquete *Pandas* [10] de Python.

- Se pide al usuario que introduzca los participantes del combate utilizando la ID interna que les hemos asignado en el dataset Datospok.csv.

- Realizamos un encoding de las variables categóricas. Se le llama encoding a la técnica de cambiar los valores categóricos a valores numéricos asignando a cada una de las categorías un código. Esto es necesario porque para el problema final utilizaremos el paquete *Sklearn* para Python (que se utiliza para muchos campos de la inteligencia artificial) y del que hablaremos con más profundidad más adelante en el trabajo. De momento, lo que necesitamos saber es que este paquete no trabaja con variables cualitativas por lo que si queremos trabajar con los tipos de cada pokémon necesitamos adaptar nuestros datos de forma que el programa los pueda entender. De entre las diversas formas posibles de realizar el encoding de un grupo de variables, para nuestras variables que determinan el tipo de un pokémon utilizo el *One Hot encoding*. Para ello añadimos a cada pokémon 18 campos, cada uno perteneciente a un tipo, asignando un valor 1 si el pokémon pertenece a ese tipo y 0 si no. Para realizar este encoding comprobaremos los campos de Tipo1 y Tipo2 del dataset generado y marcaremos con un uno los que coincidan. Este proceso nos soluciona también un problema adicional, porque pese a que llamamos tipo principal de un pokémon al primero de los dos, no existe ninguna diferencia a efectos prácticos en el orden de los tipos. Es decir, un pokémon de tipo agua-planta será exactamente igual que un pokémon de tipo planta-agua. Esto podría suponer un problema si hubiéramos definido Tipo1 y Tipo2 también dentro del dataset de combates ya que para el programa se tratan de dos campos diferentes cuando claramente no lo son. De esta manera eliminamos el orden de nuestro problema y podemos seguir trabajando.

- Creamos dos arrays que contendrán los datos de los pokémons que van a combatir, es decir, el ID, el nombre, los 18 campos de los tipos, y los stats.

-En este trabajo he utilizado un sistema de almacenamiento de datos en el que cada pokémon tiene su propio fichero -csv que contiene los combates en los que participa. Por lo tanto, el resultado de cada combate queda almacenado en dos archivos diferentes; uno del primer pokémon y otro del segundo, de forma que sea más fácil ir rellenando la información para los ficheros de todos los pokémons. En cada fichero el pokémon titular será siempre el primero y el rival siempre el segundo, esto es importante ya que vamos a marcar en el campo de resultados con el valor 1 si gana el titular y con el valor 0 si gana el rival. Para poder guardar estos combates en los ficheros generamos dos nuevos arrays: rescomb1 y rescomb2. Rescomb1 contendrá los

stats de primer Pokémon, los del segundo y el resultado, mientras que rescomb2 funcionará de forma análoga, pero invirtiendo los pokémons y el resultado.

-Una vez tengamos estos dos vectores importamos los datasets contenidos en los ficheros de los dos pokémons a los que añadiremos rescomb1 y rescomb2 respectivamente, para finalmente sobrescribir los ficheros originales.

4.2.3 Estudio sobre la creación de un árbol de decisión

Utilizando el programa anterior he generado 80 combates usando el pokémon *Braixen*. En la figura 6 podemos ver los stats de *Braixen*. *Braixen* es una segunda evolución lo cual quiere decir que tiene peor BST que una forma final y mejor que una forma base. También es interesante el hecho de que sus defensas no sean iguales y que su velocidad sea decente pero no la mejor.

Como hemos dicho, para estos combates solo hemos tenido en cuenta los stats de cada pokémon y la efectividad de tipos para decidir el resultado. Hemos utilizado solamente un ataque teórico del tipo principal del pokémon, que usa el ataque o el ataque especial dependiendo de cuál sea más alto y en caso de ser iguales emplearemos el que haga más daño.

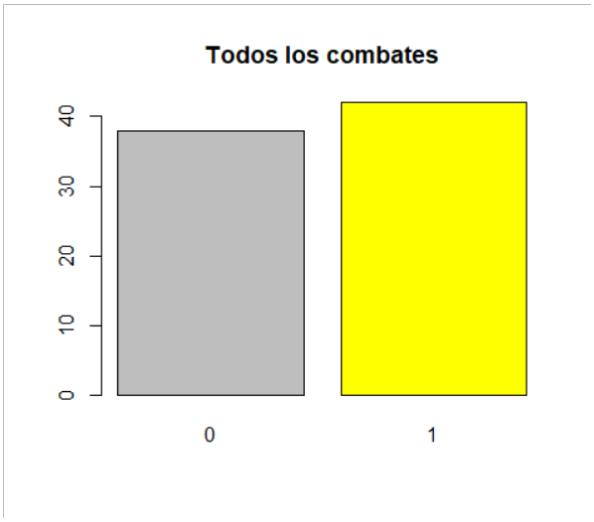
Figura 6. *Stats de base de Braixen*

Fuente: Team Builder de Pokémon Showdown

Para crear el árbol de decisión he estudiado diversos barplots creados en Rstudio que me han permitido analizar gráficamente la cantidad de elementos que tendría cada una de las

bifurcaciones del árbol. En primer lugar, he considerado los 80 combates para ver la tasa de victorias de Braixen.

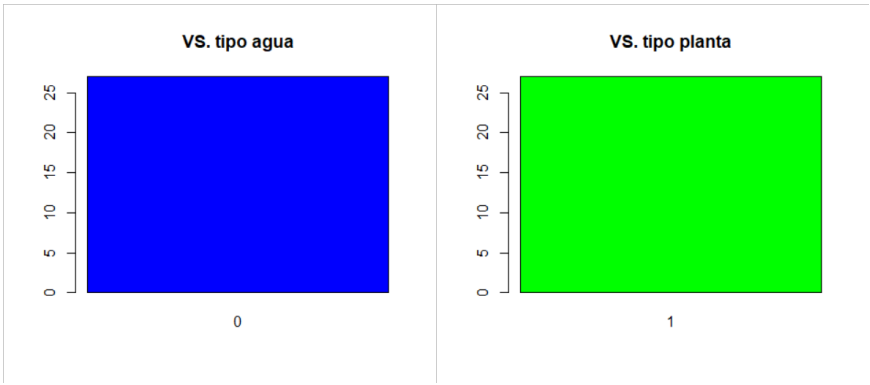
Figura 7. *Gráfico con todos los combates de Braixen*



Fuente: Elaboración propia

A partir de los resultados del gráfico se observa que Braixen gana ligeramente más combates de los que pierde, aunque está bastante igualado. Para decidir la primera separación en el árbol me he planteado cómo cambian los resultados si los separamos según el tipo del segundo Pokémon.

Figura 8. *Gráfico con los combates de tipo agua y planta de Braixen*



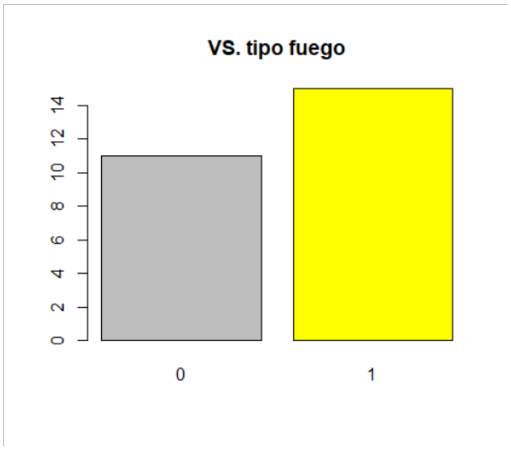
Fuente: Elaboración propia

Este resultado muestra que la eficacia de tipos se vuelve un parámetro muy importante cuando solo podemos utilizar un ataque del tipo principal. Vemos que Braixen pierde todos los combates

contra Pokémon de tipo agua independientemente de los *stats* que tenga cada pokémon pero que gana todos los combates contra los pokémons de tipo planta.

Si analizamos el gráfico contra los pokémons de tipo fuego contra los que no existirá ninguna ventaja de tipo, obtenemos los resultados mostrados en la figura 9.

Figura 9. *Gráfico con los combates de tipo fuego de Braixen*



Fuente: Elaboración propia

Vemos que contra los pokémons de tipo fuego volvemos a tener unos resultados bastante igualados así que tendremos que estudiar maneras de separar los pokémons de tipo fuego de una forma más específica. Un factor muy importante a la hora de decidir si una separación es buena para el árbol es fijarnos en cuantos elementos pertenecientes a uno de los conjuntos nos aportan el mismo resultado. En este primer caso tenemos que todos los pokémons de tipo agua ganan y que todos los pokémons de tipo planta pierden por lo que es una primera separación bastante óptima.

Decidida la primera separación del árbol, podemos estudiar una forma de separar los pokémons de tipo fuego. Teniendo en cuenta únicamente los *stats,* existen dos opciones que parecen bastante buenas a priori:

- La primera es separar entre pokémons que son más rápidos que Braixen y pokémons que son más lentos. La velocidad en un combate Pokémon es siempre uno de los factores más importantes a tener en cuenta ya que al inicio del turno es el pokémon más rápido el que ataca antes. Esto quiere decir que, en una situación con dos pokémons en la cual cada uno de ellos acabaría con el otro en cuatro ataques, el pokémon más rápido ganará siempre ya que será capaz de atacar por cuarta vez después de haber sufrido solamente tres ataques por parte del rival.

- La segunda opción que podemos estudiar es cuál de los dos es capaz de dañar más al rival con un solo ataque. Para calcular esto defino las siguientes variables:

  - 0fense1: diferencia entre el ataque especial de Braixen (1) y la defensa especial del rival (2)

$$Ofense1 = SP.ATK1 - SPDEF2 \quad \text{ec.(1)}$$

  - 0fense 2: Valor máximo de los stats correspondientes a la diferencia entre el ataque del rival (2) y la defensa de Braixen (1) en las dos modalidades posibles

$$Ofense2 = \max(SP.ATK2 - SP.DEF1, ATK2 - DEF1) \quad \text{ec.(2)}$$

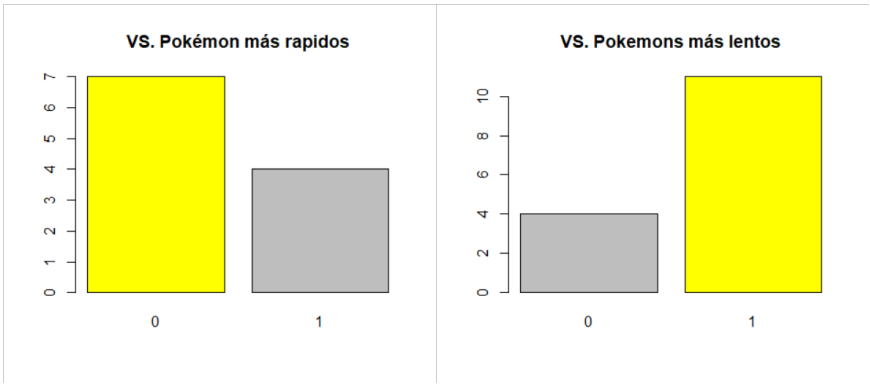
Y defino la variable Difofense: $Difofense = Ofense1 - Ofense2$ ec.(3)

Un valor positivo indica que Braixen es más fuerte mientras que un valor negativo indica que el rival es más fuerte.

Podemos valorar el efecto de estas dos variables en nuestro problema:

Empezaré estudiando los resultados obtenidos por Braixen en función de la velocidad del oponente (figura 10a).

Figura 10a. *Gráfico con los combates separados por velocidad de Braixen*



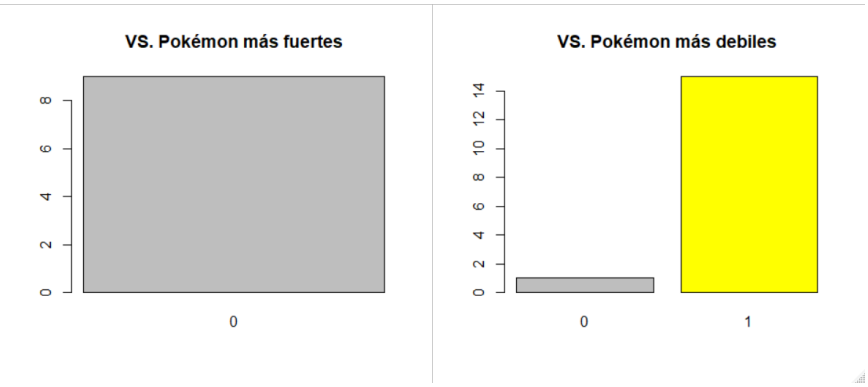
Fuente: Elaboración propia

De esta figura deducimos que, efectivamente, Braixen pierde más combates que gana contra pokémons más rápidos y gana más que pierde contra pokémons más lentos. Estos resultados son buenos, pero ninguno de los dos grupos contiene solo elementos que nos aporten una única

solución por lo que vamos a estudiar también la opción de ver quién es capaz de hacer más daño de un solo ataque antes de tomar una decisión.

El resultado de considerar la segunda de las variables, la fuerza del ataque, usando la variable *Difofense* se muestra en la Figura 10b

Figura 10b. *Gráfico con los combates separados por fuerza de Braixen*

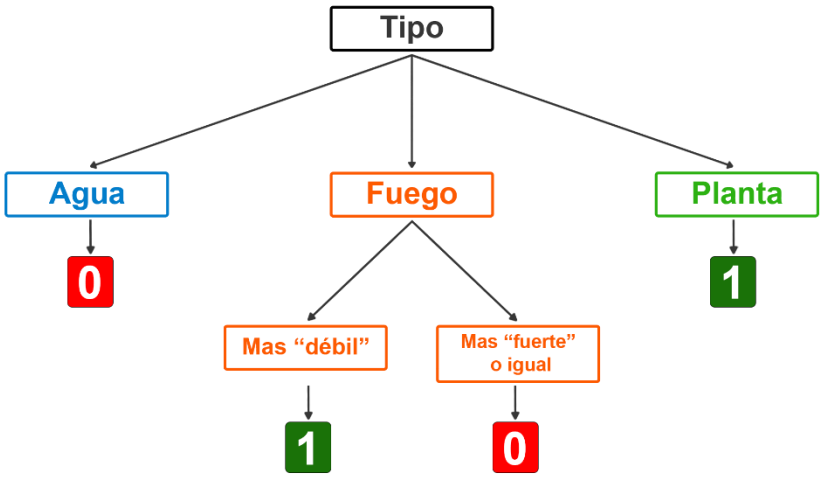


Fuente: Elaboración propia

De estos resultados se deduce que esta separación es notablemente mejor que la separación por velocidades. De hecho, solamente hay dos pokémons que escapan los patrones de este tipo de clasificación. El primero es *Raboot* que pese a ser más débil que Braixen es más rápido y el segundo es *Pignite* que tiene la misma fuerza de Braixen y por lo tanto no se encuentra en estos gráficos. De momento tenemos clasificados correctamente 78 de los combates de Braixen y nos faltan estos dos últimos casos aislados que no siguen los patrones de los demás. Debemos tomar ahora una decisión sobre el ajuste del árbol, podemos extender el árbol generando nuevas hojas en las que solo tenemos un caso, lo que podría provocar *overfitting*, o podemos ignorar estos casos tratándolos como casos excepcionales. Comencemos estudiando realmente estos dos casos: *Raboot* es más "débil" que *Braixen,* pero no por mucho, y en cambio es más rápido y tiene más vida por lo que el combate entre los dos es muy ajustado. *Pignite* en cambio tiene la misma "fuerza" que *Braixen* y pese a ser más lento termina venciendo igualmente debido a que tiene bastantes más puntos de vida. Por lo tanto, vemos que los dos casos son muy diferentes entre sí y que tendríamos que crear una hoja diferente para cada uno de ellos. Teniendo en cuenta que el modelo que hemos creado es capaz de clasificar correctamente 78 de los 80 casos no tiene mucho sentido modificarlo para dos casos aislados. Lo más recomendable sería buscar otros casos similares y estudiarlos para poder encontrar patrones y poderlos clasificar correctamente, pero si eso no es posible lo mejor es tratarlos como casos atípicos e ignorarlos. Si bien el caso de

Raboot podríamos ignorarlo como hemos dicho, el caso de *Pignite* es diferente. En la última separación de nuestro árbol hemos dividido entre los grupos más "débil" y más "fuerte" pero no hemos tenido en cuenta el caso en que sean iguales. Si miramos los stats de *Braixen* en comparación con los demás pokémons con un BST similar podemos ver que sus *stats* están mucho muy optimizados para tener una función ofensiva, siendo mínimamente defensivos solo en la defensa especial. Por lo tanto, tiene bastante sentido asumir que la gran mayoría de pokémons igual de "fuertes" que *Braixen* no estén tan optimizados y por lo tanto es normal que tengan un BST más alto y terminen ganándole el combate. De hecho, *Pignite* tiene un BST de 418 mientas el de *Braixen* es solamente de 409. Por lo tanto, incluiremos la igualdad junto con los pokémons más fuertes que *Braixen*, terminado así nuestro árbol.

Figura 11. *Gráfico con los combates separados por fuerza de Braixen*



Fuente: Elaboración propia

Una vez analizada y decidida la estructura del árbol la implementaré mediante el programa Prototipo2manual.py. Este programa comienza importando el fichero Braixen2.csv, que contiene 50 combates contra 10 pokémons de tipo agua, 30 de tipo fuego y 10 de tipo planta, elegidos al azar. Implementamos el árbol utilizando un conjunto de funciones *if* para ir recorriendo las ramas hasta llegar a las hojas y guardamos los resultados dentro de un array de 50 elementos que al final compararemos con los resultados reales de los combates. El programa finaliza comparando el array con los resultados de los combates y mostrando el porcentaje de precisión que hemos obtenido, que en nuestro caso es un 100%, por lo que podemos asumir que el árbol ha superado el test con los datos de prueba.

Estos programas se pueden encontrar junto con el resto de los programas de este trabajo en el link que se encuentra al inicio de la sección "Resultados".

4.3 Introducción a scikit-learn

Para implementar este prototipo he realizado el estudio necesario para decidir las separaciones que se van a utilizar en el árbol. **De cara a la implementación del programa final utilizaré el paquete *Sklearn* para Python que proporciona las herramientas necesarias para la creación de nuestro modelo** y que explico a continuación.

El paquete de *scikit-learn* o *Sklearn* es una biblioteca para aprendizaje automático de Python que incluye algunos algoritmos de clasificación, regresión y análisis de grupos. En este proyecto he utilizado los algoritmos dedicados a *Árboles de Decisión*, *Random Forest y Gradient Tree Boosting*.

Para hacer uso del algoritmo que genera el árbol de decisión con los datos aportados realizo los siguientes pasos:

- Importar el paquete y crear un nuevo modelo que especificamos como árbol de decisión. En un principio este modelo estará vacío y será necesario entrenarlo con nuestros datos para seguir avanzando.
- En primer lugar, decidir cuál será la variable que queremos predecir. En este caso será la variable *Results* cuyo valor será 0 si el pokémon pierde y 1 si el pokémon gana.
- El paso siguiente es decidir cuáles de las columnas de nuestro dataset vamos a tener en cuenta en la creación de nuestro modelo. En nuestro planteamiento consideraremos todas las variables (menos el nombre y el ID).
- Una vez decididas las variables tenemos que entrenar el modelo y para ello utilizaremos el comando *fit*. Este comando es el corazón del modelo ya que es el que creará nuestro árbol decidiendo qué separaciones utilizar.

4.3.1 Funcionamiento del comando *fit* para árboles de decisión

El comando *fit* se encarga de elegir las separaciones que tendrá el árbol y, por lo tanto, es una parte muy importante del proceso. En este apartado vamos a explicar cómo se toman estas decisiones y observaremos las similitudes con el proceso llevado a cabo en la creación de nuestro árbol de forma manual.

Denotemos por x_i las columnas de nuestro dataset que se utilizarán para la creación del modelo y por y al vector de outputs, correspondiente al campo *Results*, sobre el que vamos a basar nuestras predicciones. Denotemos ahora también los datos que tenemos en un cierto nodo del árbol por Q_m que tendrá n_m ejemplos que queremos clasificar. En cada nodo del árbol el programa debe evaluar todas las potenciales separaciones que se pueden hacer y encontrar una manera de determinar cuál sería la mejor. Estas separaciones las representamos a través de la variable $\theta = (j, t_m)$ donde j será la variable que vamos a utilizar para realizar la separación y t_m será el umbral que utilizaremos para realizar la partición. Cada partición divide nuestro nodo Q_m en los nodos $Q_m^{derecha}$ y $Q_m^{izquierda}$ y la fórmula para decidir la mejor separación se base en estos nodos.

La fórmula que indica cómo de buena es la separación viene dada por:

$$G(Q_m, \theta) = \frac{n_m^{izquierda}}{n_m} H\left(Q_m^{izquierda}(\theta)\right) + \frac{n_m^{derecha}}{n_m} H\left(Q_m^{derecha}(\theta)\right) \qquad \text{ec.(4)}$$

Donde *H()* es una fórmula que determina la impureza del nodo. Denotando por p_{mk} la proporción de ejemplos en un nodo con un output especifico:

$$p_{mk} = \frac{i_k}{n_m}$$

Donde i_k corresponde al número de ejemplos dentro de Q_m cuyo output asociado sea y=k, para el cálculo de los coeficientes H() podemos usar las fórmulas alternativas:

- Función gini:

$$H(Q_m) = 1 - \sum_k (p_{mk})^2 \qquad \text{ec.(5)}$$

- Función entropia:

$$H(Q_m) = -\sum_k p_{mk} \log\left(p_{mk}\right) \qquad \text{ec.(6)}$$

En ambos casos el coeficiente mostrará más homogeneidad cuanto más cercano sea a 0. Esto es fácil de entender ya que un nodo homogéneo solo tendrá ejemplos con un mismo output. Es decir $i_j = n_m$ para un cierto $k = j$ y $i_k = 0$ para $k \neq j$. En este caso tendremos que $p_{mj} = 1$ y que $p_{mk} = 0$ si $k \neq j$. En este caso es sencillo ver que $H(Q_m) = 0$ tanto si usamos el gini o la entropía ya que $\sum_k (p_{mk})^2 = 1$ y por lo tanto $1 - \sum_k (p_{mk})^2 = 0$, y que $\log\left(p_{mj}\right) = 0$ por lo que $-\sum_k p_{mk} \log(p_{mk}) = 0$. Este sería el caso óptimo en el que hemos encontrado una separación que genera un nodo homogéneo que, por lo tanto, terminará siendo una hoja de

nuestro árbol. El caso contrario, es decir, aquel en el que nuestro coeficiente es lo más elevado posible, se daría cuando todos los p_{mk} tienen el mismo valor, es decir, existe el mismo número de ejemplos en Q_m para cada output y. Si pensamos en nuestro problema esto significaría que la mitad de los ejemplos serían victorias (1) y la otra mitad serían derrotas (0). Podemos suponer que nuestro número de ejemplos es par, para que realmente podamos tener dos mitades, $p_{m1} = 0.5$ y $p_{m0} = 0.5$ por lo que:

$$1 - ({p_{m1}}^2 + {p_{m0}}^2) = 0.5 \quad \text{y} \quad -0.5log(0.5) - 0.5\log(0.5) = 1$$

Podemos observar que el *gini* puede variar entre 0 y 0.5 y la entropía entre 0 y 1. En ambos casos cuánto más pequeño sea el coeficiente más homogéneo será el nodo.

Pese a que la función de *Sklearn* tiene una implementación para ambas clases, diversos estudios [11] demuestran que es mejor utilizar el *gini* ya que su coste computacional es mejor y los resultados obtenidos son igual de buenos en la gran mayoría de casos. Por lo tanto, nosotros utilizaremos la formula del *gini* en nuestro trabajo.

Ahora que ya hemos analizado nuestros coeficientes de impureza podemos pensar que la mejor separación será aquella en la que $G(Q_m, \theta)$ sea lo más próximo a 0 posible. Por lo tanto, si no se puede crear una separación tal que $Q_m^{derecha}$ y $Q_m^{izquierda}$ sean nodos homogéneos, se priorizarán las separaciones que creen nodos lo más homogéneos posibles con el mayor número de ejemplos dentro de ese nodo.

En la siguiente sección aplicaremos las funciones de *Sklearn* para generar un árbol con los datos que hemos utilizado en la construcción de nuestro prototipo y realizaremos también un ejemplo del cálculo manual del $G(Q_m, \theta)$.

4.4 Aplicación de Sklearn a los datos de nuestro prototipo

En este apartado estudiaremos, como aplicación de esta biblioteca, la generación de un árbol de decisión sobre los mismos datos que hemos utilizado en nuestro primer prototipo, para poder comparar de este modo los resultados obtenidos con los del apartado 4.2.

Para ello he realizado un nuevo programa de Python *Prototipo2sklearn.py*.

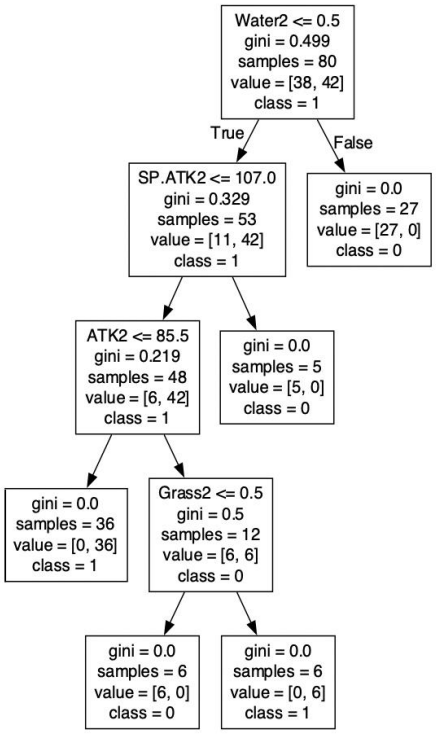
Este programa importa, en primer lugar, el dataset con la ayuda de la biblioteca *pandas*. Para poder crear el árbol se siguen los siguientes pasos:

- Lo primero que hay que hacer es generar una lista, *features,* donde guardaremos los nombres de las columnas que queremos que se tengan en cuenta en la creación del árbol.
- Después asignaremos los elementos de *X* e *Y*. *X* será un dataset que contendrá los datos de las columnas que hemos incluido en *features* e *Y* será el campo que queremos predecir, es decir, la columna *Result* de nuestro dataset. Una vez que ya tenemos definidas *X* e *Y* solamente nos falta construir el modelo y entrenarlo.
- Para crear el modelo importamos de la biblioteca *Sklearn* el paquete *DecisionTreeClasifier* para generar el árbol de decisión. Formaremos un nuevo árbol que llamaremos modelo pero que de momento estará vacío y, para que se convierta en el que buscamos, utilizaremos el comando *fit*. Éste entrenara nuestro árbol con los elementos *X* e *Y* que hemos definido previamente.
- Una vez ejecutado el comando *fit* nuestro árbol ya estará generado y podremos aplicar sobre él los diversos comandos de *Sklearn* para comprobar su funcionamiento y realizar predicciones. Pero en este apartado lo que nos interesa es ver el árbol de manera gráfica. Existen diversas maneras de imprimir un árbol de forma gráfica, pero nosotros utilizaremos la biblioteca *graphviz* [12] de Python. En la figura 12 podemos ver dicho gráfico.

La primera línea de cada nodo nos dirá cuál de todos los campos se ha utilizado para decidir la separación, seguido del umbral por el que hemos separado. La segunda línea de cada nodo es el coeficiente de *gini* que se utiliza para decidir la separación. La tercera fila muestra el número de ejemplos para cada output, en el primer nodo tenemos 38 casos con resultado 0 y 42 con resultado 1. Y la última fila nos muestra cuál es el output con más ejemplos dentro del nodo. En el caso del primer nodo sería 1 puesto que 42>38.

Lo primero que podemos ver en el árbol de la figura 12 es que cada separación divide siempre cada nodo en dos nodos nuevos.

Figura 12. *Primer árbol del prototipo generado con sklearn*



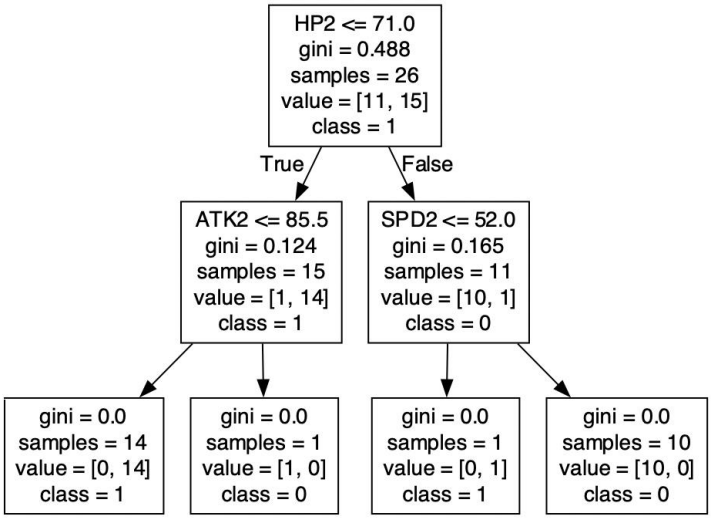
Fuente: Elaboración propia

Si recordamos la fórmula que utiliza el comando *fit* de *Sklearn*, que hemos presentado en la sección de introducción a *sickit-learn*, vemos que solo tiene en cuenta $Q_m^{derecha}$ y $Q_m^{izquierda}$, por lo que cada separación siempre creará dos nodos nuevos. Esto ya presenta una diferencia importante con el árbol que hemos creado manualmente ya que nuestra primera separación generaba tres nodos nuevos, para separar los tipos agua, planta y fuego.

Podemos modificar el árbol añadiendo un nuevo campo a nuestro dataset, que corresponde al valor *Difofens* definido en la ecuación (3)). Sin embargo, al volver a imprimir el árbol vemos que esto no causa ninguna diferencia. Esto se debe a que el árbol generado por *Sklearn* separa primero los pokémons de tipo agua, por lo que en el segundo nodo no tenemos solamente los pokémons de tipo fuego, como nosotros habíamos estudiado, sino que ahora tenemos los pokémons de fuego y planta.

Como esta primera separación en tres ramas no es posible utilizando *Sklearn,* vamos a analizar por separado los combates contra tipo fuego para comparar el resultado con nuestros resultados previos. Generamos otro árbol, de manera análoga a como hemos creado el primer árbol, introduciendo el nuevo dataset *Braixenfuego.csv*. En la figura 13 podemos ver cómo queda el árbol.

Figura 13. *Árbol de los combates contra pokémons de tipo fuego*

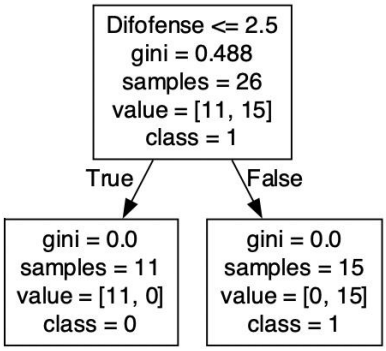


Fuente: Elaboración propia

Podemos ver que este árbol sigue sin parecerse a lo que nosotros hemos decidido en nuestro estudio. Esto es debido a que en los dos árboles que hemos generado hasta ahora (Figuras 12 y 13) el comando *fit* utiliza solamente los *stats* de un campo mientras que nosotros hemos utilizado una combinación de ellos a través de la variable *Difofense* (ecuación (3)), partiendo de nuestro conocimiento previo de las relaciones que algunos campos tenían entre ellos. Por consiguiente, si queremos mostrar las relaciones entre algunas clases, como por ejemplo las capacidades ofensivas que hemos utilizado o las velocidades, propongo incluir un campo nuevo correspondiente a la variable *Difofense*. Aplicando esta opción sobre el mismo grupo de datos sobre el que estábamos trabajando cuando realizamos el estudio previo obtenemos el árbol mostrado en la figura 14.

El resultado muestra que el nuevo árbol generado es muy similar al que nosotros hemos creado a mano. La única diferencia es que en vez de separar entre más “fuerte” o más “débil”, utilizando un umbral igual a 0, se ha utilizado un umbral de 2.5 que corresponde a la diferencia ofensiva con el pokémon *Raboot*, que como hemos dicho es el único pokémon más “débil” que Braixen pero que consigue la victoria en su combate.

Figura 14. *Árbol de los combates contra pokémons de tipo fuego después de añadir el campo Difofense*



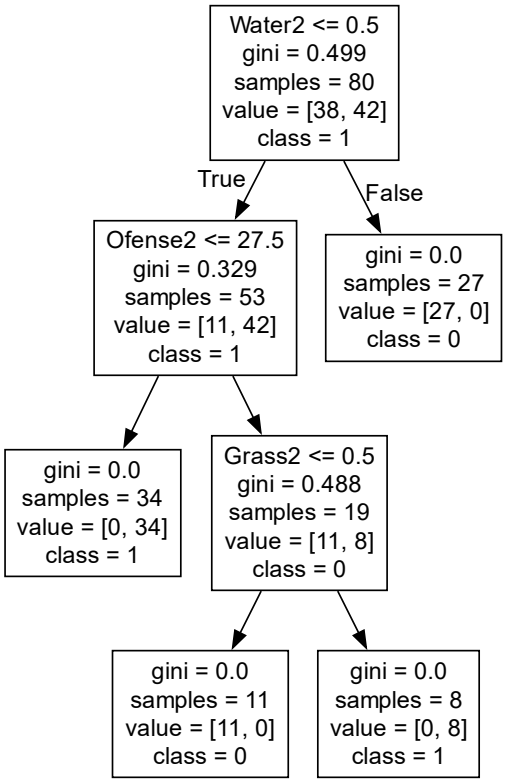
Fuente: Elaboración propia

Volvamos ahora a nuestro primer árbol, el que está representado en la figura 12. Un análisis más detallado muestra que este árbol incluye una separación utilizando el campo SP.ATK2 seguida de una separación usando el ATK2. Nuestra definición de la variable *Ofense2* (ecuación (2)) proporciona un campo que tiene en cuenta tanto el ataque especial como el ataque físico.

Puesto que, normalmente, un pokémon sólo utiliza ataques de su mejor especialidad ofensiva (suele ser o bien un atacante especial o un atacante físico), *Ofense2* no solo tiene en cuenta ambos a la vez y los transforma en un único campo, sino que los relaciona directamente con la DEF y la SP.DEF del pokémon que recibe el ataque lo cual resulta mucho mejor ya que de esta manera las relaciones entre ATK y DEF, y entre SP.ATK y SP.DEF quedan visibles en un nuevo campo. Al incluir este nuevo campo en nuestra X obtenemos el nuevo árbol mostrado en la figura 15.

Estos resultados muestran que nuestro nuevo campo se ha comportado mejor, substituyendo a las dos separaciones que dependían del SP.ATK2 y de ATK2.

Figura 15. *Segundo árbol del prototipo generado con Sklearn añadiendo el campo Ofense2*



Fuente: Elaboración propia

Podemos analizar el efecto en estos nodos mediante un cálculo manual del $G(Q_m, \theta)$ del nodo de la figura 12 que separa usando la variable SP.ATK2 y el nodo de la figura 15 que separa usando la variable *Ofense2*.

Empezamos calculando el factor $G(Q_m, \theta)$ en el nodo correspondiente a la separación que depende del campo SP.ATK2, donde en este caso $\theta_1 = (SP.ATK2, 107)$. El nodo generado a la izquierda separa por ATK2 y contiene 48 ejemplos. El valor *value* nos indica que este nuevo nodo tiene 6 ejemplos con output 0 y 42 con output 1 por lo que el *gini* vale, aplicando la ecuación (5):

$$Gini = 1 - \left[\left(\frac{6}{48}\right)^2 + \left(\frac{42}{48}\right)^2\right] = 0.21875 \approx 0.219$$

En cuanto al nodo que se genera a la derecha al aplicar θ_1 vemos que se trata de un nodo homogéneo de clase 0 (todos los ejemplos tienen output 0 y contiene 5 ejemplos).

Por lo tanto, si denotamos al nodo que se genera a la izquierda después de realizar la separación utilizando el parámetro Water2 como Q_2 tendremos que

$$G(Q_2, \theta_1) = \frac{48}{53} \times 0.219 + \frac{5}{53} \times 0 = 0.1983$$

Pasemos ahora a calcular este valor también para los nodos que se crean al aplicar la separación por *Ofense2*, usando los datos que muestra la Figura 15 y donde $\theta_2 = (Ofense2, 27.5)$. Vemos que el nodo que se genera a la izquierda tras aplicar θ_2 es un nodo homogéneo de clase 1 con 34 ejemplos y que el nodo que se genera a la derecha contiene 19 ejemplos y su valor *gini* es de 0.488, lo que implica que es bastante heterogéneo. Por consiguiente:

$$G(Q_2, \theta_2) = \frac{34}{53} \times 0 + \frac{19}{53} \times 0.488 = 0.1749$$

En consecuencia, puesto que $G(Q_2, \theta_2) < G(Q_2, \theta_1)$, el resultado obtenido al aplicar *Ofense2* será una separación mejor.

Observamos que, pese a que θ_2 genera un nodo casi perfectamente heterogéneo con un *gini* de 0.488, al generar también un nodo homogéneo con 34 ejemplos sigue siendo mejor que θ_1, que genera un grupo homogéneo de solo 5 ejemplos y un nodo menos heterogéneo que el de θ_2 pero mucho más grande. Esta es la razón por la que el programa no ha separado los pokémons de tipo fuego y de tipo planta en este segundo nodo, porque esto nos generaría un grupo homogéneo de 27 elementos que serían los pokémons de tipo planta y un grupo con 26 ejemplos que tendría prácticamente un *gini*=0.5 correspondiente a los pokémons de tipo fuego. Esta separación daría un valor más grande de $G(Q_2, \theta)$ lo que hace que el árbol generado con un máximo de dos nodos nuevos por separación sea diferente al que nosotros hemos creado manualmente con una primera separación que genera tres nodos.

Si realizamos la clasificación de los datos de *testing* que hemos utilizado en nuestro prototipo obtenemos que el árbol de la figura 12 comete 4 errores mientras que el de la figura 15 solo comete 3. Como ya hemos visto, el árbol que hemos hecho a mano no comete ningún error, por lo que observamos que los árboles que hemos creado con variables nuevas generadas a partir de la combinación de otras aportan mejores resultados. Pese al resultado obtenido, todavía no tenemos suficiente información para poder asegurar que este procedimiento mejora la precisión del problema puesto que hay que tener en cuenta que el problema que estamos considerando trabaja con un dataset muy reducido y analizamos los resultados obtenidos sobre un dataset de testeo también muy pequeño.

Para la realización de nuestro programa final he creado un dataset con 389 pokémons y 580 combates sobre los que aplicaré la técnica de Split entre training y testing para poder recoger

más resultados. Al trabajar sobre este problema he creado también este tipo de nuevas variables y podremos analizar si mejoran o no la precisión.

4.5 Planteamiento del problema final

Una vez considerado el modo en el cual *Sklearn* genera un árbol a partir de los datos de nuestro dataset haré el planteamiento del problema final. Para poder establecer el marco del problema es necesario tener en cuenta algunos puntos:

La creación del dataset

Lo óptimo sería poder crear un dataset incluyendo todos los pokémons existentes y en el que cada pokémon tenga, al menos, un combate contra el resto. Esta opción es irrealizable en el tiempo de que disponemos. Empezaré seleccionando una línea evolutiva de cada tipo (18 tipos posibles) para cada generación (9 generaciones) lo que se traduce en un total de 162 líneas evolutivas. Puesto que muchas de estas líneas evolutivas contienen pokémons con dos tipos distintos, acabaremos teniendo más ejemplos de cada tipo. Por ejemplo, en la cuarta generación de tipo *Dragón* hemos seleccionado a *Gibble*, con evoluciones de tipo *Dragón-Tierra* por lo que, además de las 9 líneas evolutivas de tipo *Tierra* que hemos seleccionado, también tendremos la línea de Gibble como ejemplo de pokémon de tipo *Tierra*. Tras seleccionar las líneas evolutivas de cada tipo, nuestro dataset definitivo contiene 389 pokémon.

Con estos pokémons he realizado dos datasets que contendrán combates:

- El primero contiene combates de un solo pokémon contra los 388 restantes lo que nos permite poder valorar, usando un número de datos más reducido, todas las relaciones entre tipos, objetos y ataques que terminan afectando al resultado del combate.
- El segundo dataset contiene 200 combates entre pokémons de este dataset diferentes a nuestro pokémon seleccionado para el primer dataset. Más adelante añadiremos los datos de este segundo dataset al primero y veremos como variará la precisión de las predicciones.

Relaciones entre elementos del dataset

Una parte importante que quiero remarcar antes de empezar a explicar el resto de las decisiones que he tomado es que, en este problema, siempre vamos a tener en cuenta los sucesos que

tengan lugar de forma más probable. Es decir, si un pokémon tiene una probabilidad de 4,2% de dar un golpe crítico al atacar no lo tendremos en cuenta ya que esto no sucederá en el 95,8% de los casos. Esto es de especial interés al trabajar con los daños que producen los ataques de los pokémons.

El daño que produce un ataque se calcula usando la siguiente formula:

$$Daño = 0.01 \times STAB \times Efectividad \times R \times \left(\frac{(0.2 \times Nivel + 1) \times ATK \times Poder}{25 \times DEF} + 2\right) \qquad \text{ec.(7)}$$

donde

Efectividad toma los valores de la tabla de la figura 1,

STAB toma el valor 1.5 si el ataque es del mismo tipo que el pokémon y 1 en caso contrario,

ATK y *DEF* utilizarán el ataque y defensa físico o especial dependiendo de la especialidad del ataque,

El parámetro R corresponde a un valor aleatorio que puede tomar valores entre 85 y 100, proporcionando 16 valores diferentes a la cantidad de daño que un mismo ataque puede causar.

Una consideración importante: Aunque algunos combates reales se deciden por estas pequeñas variaciones, nosotros solo tendremos en cuenta los resultados que suceden con más frecuencia. Por ejemplo, en un combate en el que un pokémon resultase vencedor solamente si en el cálculo del daño de su ataque el valor R estuviera entre 98 y 100, contaríamos el combate como una derrota ya que al recrear el combate 100 veces terminaría perdiendo más veces de las que ganaría. De forma similar, los efectos secundarios que pueda causar un ataque no se tendrán en cuenta si su probabilidad de suceder es inferior al 50%. Esta forma de tener en cuenta los sucesos que tienen un cierto umbral de probabilidad no se adopta solamente para simplificar el problema, sino que tiene en cuenta la situación habitual en la cual al preparar un equipo se suelen considerar los escenarios que suceden con mayor probabilidad evitando los cálculos para casos muy raros.

4.5.1 Adaptación de las diversas mecánicas.

Antes de seleccionar nuestro pokémon principal para el dataset tenemos que decidir el modo de trabajar con las mecánicas que determinan a cada pokémon, es decir los stats, las habilidades, los ataques y los objetos.

- **Stats:** En el primer prototipo hemos utilizado solamente los stats de base, sin tener en cuenta ni los EV ni las naturalezas. Sin embargo, dado que las diferentes variaciones de los stats de un pokémon son uno de los principales factores que se estudian en la creación de un equipo competitivo, esta simplificación no sería útil para poder predecir resultados de combates reales.

  Para considerar estos factores en el problema podemos almacenar los stats de base, los EV y la naturaleza por separado o podemos utilizar la fórmula que determina los stats del pokémon a nivel 100 que hemos descrito en el glosario:

  Los stats de un Pokémon se calculan siguiendo la siguiente fórmula para los HP:

$$HP = \left[\frac{\left(HP_{Base} \times 2 + IV + \left[\frac{EV}{4}\right]\right) \times Nivel}{100}\right] + Nivel + 10 \qquad \text{ec.(8)}$$

  Y la siguiente fórmula para el resto de los stats:

$$Stat = \left(\left[\frac{\left(Stat_{Base} \times 2 + IV + \left[\frac{EV}{4}\right]\right) \times Nivel}{100}\right] + 5\right) \times Naturaleza \qquad \text{ec.(9)}$$

  Al analizar un combate lo realmente importante son los stats finales del pokémon. Tener en cuenta el número de EV que nosotros podemos invertir en un *stat* tendrá menos relevancia que el valor final que obtendremos ya que serán las comparaciones entre los *stats* de los pokémons lo que determinan, en parte, quien será el ganador. Por lo tanto, la mejor decisión será almacenar directamente los stats finales del pokémon, implementando dentro del programa el cálculo de estos stats utilizando las fórmulas presentadas.

- **Habilidades:** Existen 298 habilidades diferentes, muchas de ellas perteneciendo solo a una línea evolutiva especifica. Estas habilidades aportan un efecto pasivo al pokémon que puede activarse o volverse útil solamente en escenarios muy específicos. Debido a lo reducida que es nuestra base de datos no nos sería posible estudiar todas las interacciones en las que cada habilidad tiene utilidad por lo que para nuestro problema no tendremos en cuenta las habilidades de cada pokémon. En la sección de “Trabajo futuro” hablaremos sobre cómo se podrían abordar algunos de los problemas que

presentan el gran número de interacciones diferentes que pueden suceder entre habilidades y efectos de ataques en combates Pokémon.

- **Ataques:** En nuestro prototipo hemos tenido en cuenta solamente un ataque del tipo de nuestro pokémon y siempre del mismo poder, pero en un combate de pokémon real cada pokémon puede tener aprendidos hasta un máximo de 4 ataques, lo que le permite estar preparado para diferentes tipos de escenario. Por ejemplo, si Braixen además de su ataque de fuego tuviera aprendido un ataque de planta podría vencer a muchos pokémons de tipo agua. Existen un total de 913 movimientos y cada movimiento se caracteriza por tener un tipo, una especialización, un poder, una cantidad de PP, una precisión y un efecto secundario. Por ejemplo, en la figura 16 podemos ver las características del ataque lanzallamas. Los dos primeros rectángulos después del nombre nos muestran el tipo y la especialidad, en este caso tipo fuego y ataque especial. Después nos muestra el poder del ataque y su precisión. La precisión de un ataque determina la posibilidad de que este ataque acierte o falle, en caso de que falle el ataque no tendrá efecto. Normalmente en competitivo solo se utilizan ataques con una precisión considerablemente alta y por lo tanto aplicando el mismo criterio que utilizamos para todas las demás mecánicas que suceden con un acierta probabilidad asumiremos que todos los ataques aciertan siempre. La cantidad de PP es el número de veces que se puede utilizar un ataque en un combate, esto no afectará a nuestro problema ya que en un combate 1vs1 no se correrá el riesgo de quedarse sin PP por lo que no lo tendremos en cuenta. En cuanto a los efectos secundarios solamente tendremos en cuenta los que sucedan con más de un 50% de probabilidad.

Figura 16. *Características del ataque lanzallamas*

Fuente: Team Builder de Pokémon Showdown

Existen diferentes maneras en las que podemos guardar los ataques dentro de nuestro dataset. Necesitamos hacer un *encoding* de esta variable, ya que es del tipo categórica, y necesitamos un método que nos permita no tener en cuenta el orden ya que, a efectos prácticos, es independiente cual es el ataque 1 y cual el ataque 2, por ejemplo.

Teniendo en cuenta estos requisitos la mejor opción volverá a ser el uso del *One Hot Encoding*, pudiendo simplemente almacenar el nombre del ataque o también algunas de sus cualidades. Almacenar solamente el nombre del ataque sería una opción muy buena si tuviésemos muchos ejemplos de pokemons con cada ataque, pero eso es muy complicado que suceda debido a que normalmente se utilizan solamente un número reducido de ataques considerados mejores. Una mejor opción es generar 18 columnas, una para cada tipo, ya que normalmente un pokémon lleva aprendido un ataque por tipo y además el poder de estos ataques puede ser utilizado como código para un determinado ataque de un tipo. En la mayoría de tipos existe un gran número de ataques con poder bajo, pero solamente unos pocos con poder alto (por ejemplo, un ataque de *fuego* con 90 de poder solo puede ser *lanzallamas*) y los efectos secundarios de estos ataques también están relacionados con el poder (por ejemplo, normalmente ataques con un poder muy alto de 120 causan al usuario una proporción del daño que hacen al rival después de su uso, mientras que ataques con menos poder tienen efectos secundarios que benefician al usuario, como aumentar su velocidad). En competitivo normalmente se utilizan más los ataques con más poder mientras que el uso de ataques con un poder más reducido es menor y su uso se debe a que tienen un efecto secundario muy potente.

Utilizando esta forma de encoding el programa podrá tener un valor numérico que indica de forma cuantitativa el poder del ataque dentro de las columnas de tipo que hemos utilizado. Además, como solo tenemos en cuenta los efectos secundarios con una probabilidad mayor del 50%, al decidir los resultados del combate tenemos en cuenta que estos efectos se verán reflejados en nuestro modelo porque esta combinación funciona como un código para los ataques más utilizados. En la figura 17 podemos ver un fragmento de nuestro dataset que muestra un pokémon que tiene aprendidos los ataques: *Retribución*, de tipo *Normal* y 102 de poder, *Tijera X*, de tipo *Bicho* y 80 de poder, *Garra umbría* de tipo *Fantasma* y 70 de poder, y *Hoja aguda*, de tipo *Planta* y 90 de poder.

Figura 17. *Fragmento de nuestro dataset mostrando el encoding de los ataques*

ATK.Normal1	ATK.Fighting	ATK.Fying1	ATK.Poison1	ATK.Ground1	ATK.Rock1	ATK.Bug1	ATK.Ghost1	ATK.Steel1	ATK.Fire1	ATK.Water1	ATK.Grass1
102	0	0	0	0	0	80	70	0	0	0	90

Fuente: Elaboración propia

Existen también muchos ataques de estado diferentes, aunque muchos de ellos tienen básicamente el mismo efecto con algún cambio en el tipo o en la precisión. Tendremos en cuenta solo los más importantes ya que muchos de ellos solo tienen uso en escenarios muy específicos. Agruparemos estos ataques en los grupos:

- **Para:** Todos los ataques de estado cuyo efecto secundario es paralizar al rival.
- **Burn:** Todos los ataques de estado cuyo efecto secundario es quemar al rival.
- **Toxic:** Todos los ataques de estado cuyo efecto secundario es intoxicar al rival.
- **Sleep:** Todos los ataques de estado cuyo efecto secundario es dormir al rival.
- **Recover:** Todos los ataques de estado cuyo efecto sea curar al usuario 50% de su vida máxima.

- **Objetos:** Existen un total de 737 objetos en pokémon. Muchos de ellos no se utilizan o incluso ya no existen en los juegos más modernos, pero siguen siendo demasiados para poder tenerlos todos en cuenta en nuestro problema. Pese a que existen muchísimos objetos que funcionan solamente en unos escenarios muy reducidos existen unos pocos que aportan beneficios de forma continua por tenerlos equipados y cuya influencia en el combate será visible en la mayoría de los combates. Este tipo de objetos son los que se utilizan más en competitivo y se clasifican en objetos ofensivos y defensivos. Nosotros trabajaremos con tres objetos ofensivos y tres objetos defensivos.

  A primera vista podría parecer que, habiendo tantos objetos, seleccionar solamente una muestra de seis podría ser demasiado poco, pero no existe ningún equipo de torneo de los últimos meses que no haya utilizado por lo menos tres de estos objetos. De hecho, el uso conjunto de todos estos objetos supera el 70% de los objetos usados en torneos oficiales individuales de los últimos tres años. Todos ellos además tienen efectos siempre visibles ya que 5 de ellos mejoran los stats del pokémon, pero aplican diversas limitaciones y el ultimo regenera una pequeña cantidad de vida al pokémon al final de cada turno.

  Los objetos que utilizaremos son:

  - **Choice Band:** No permite cambiar de ataque al pokémon mientras siga en combate, pero aumenta el stat ATK en un 50%
  - **Choice Specs:** No permite cambiar de ataque al pokémon mientras siga en combate, pero aumenta el stat SP.ATK en un 50%.
  - **Choice Scarf:** No permite cambiar de ataque al pokémon mientras siga en combate, pero aumenta el stat SPD en un 50%.

- **Assault Vest:** No permite utilizar ataques de estado, pero aumenta el stat SP.DEF en un 50%.
- **Eviolite:** Aumenta los stats de DEF y SP.DEF en 50%, pero solamente funciona si el pokémon no es la forma final de su línea evolutiva.
- **Leftovers:** Al final del turno el portador del objeto regenera el 6.25% de su vida máxima.

Realizaremos un encoding de los objetos utilizando otra vez el *One Hot Encoding* ya que para un grupo así de reducido de variable categóricas es la mejor opción.

Si fuésemos a trabajar con todos los objetos existentes (737) usando el one hot encoding, el resultado sería un dataset incluyendo una tabla con 737 columnas en las que cada fila contendría 736 ceros y un solo 1. En este caso sería interesante estudiar el uso del *Binary Encoding*. Este método se basa en la idea de que la representación en binario de un número entre 1 y 737 necesita un máximo de 10 dígitos. En este caso asignamos a cada fila un valor que corresponde al número de columna que contiene el valor 1, expresado como número binario. Por lo tanto, podríamos codificar la misma información usando una tabla de tamaño Nx10 (filas x columnas) en lugar de una de Nx737. Como contrapartida, este método puede generar dificultades en la comprensión del árbol generado a partir de estos datos.

Por ejemplo, para nuestro caso con solo 6 objetos necesitaríamos tres columnas para codificar los números del 1 al 6. Si asignásemos el valor 1 a el ítem Choice Band, el 2 al ítem Choice Specs y el 3 al ítem Choice Scarf y aplicásemos Binary encoding obtendríamos el resultado que se ve en la figura 18.

Figura 18. Ejemplo de Binary Encoding

Item_1	Item_2	Item_3
0	0	1
0	1	0
0	1	1

Fuente: Elaboración propia

La razón por la que no vamos a implementar el *Binary Encoding* en este trabajo es porque en nuestro caso al reducir el número de objetos a seis, el *One hot encoding* es una

solución adecuada. El *binary encoding* podría ser una solución mucho más óptima para la ejecución del programa si estuviésemos trabajando con una gran cantidad de valores.

4.5.1 Selección del pokémon sobre el que realizar el estudio

Como he mencionado anteriormente, el dataset principal contendrá los combates de uno de nuestros 389 pokémon contra los demás. La elección de éste pokémon es muy importante y debemos analizar las diversas características que queremos en el pokémon que vayamos a utilizar para que el estudio tenga la mayor variedad de casos posibles. Lo primero es que debe ser un pokémon con dos tipos. Lo segundo es que esta combinación de tipos debe tener aproximadamente el mismo número de debilidades que de resistencias, a ser posible incluyendo alguna debilidad x4 y alguna resistencia x1/4. Por último, el pokémon debe de tener unos stats balanceados con un BST cerca de 500 que sería la media de lo que tendría una forma final de una cadena evolutiva. Después de comparar diversos candidatos potenciales finalmente el pokémon que vamos a utilizar sera *Leavanny,* cuyos stats base se pueden ver en la figura 19.

Figura 19. *Stats base de Leavanny*

Fuente: Team Builder de Pokémon Showdown

En la figura 19 podemos ver que *Leavanny* tiene unos stats base similares en todos los campos lo que permite que pueda criarse para cumplir diversas funciones ya sean ofensivas o defensivas y por lo que podrá utilizar todos los objetos que hemos mencionado en la sección de objetos menos la *Eviolite* ya que se trata de una forma final de una cadena evolutiva. El BST de *Leavanny*

es exactamente de 500 y con su velocidad en 92 lo hace un candidato ideal ya que, pese a tener un BST promedio, su velocidad le permitirá ser más rápido que muchos pokémons con funciones principalmente defensivas y más lento que pokémons que sean principalmente ofensivos. Claramente esto dependerá de cómo se repartan los EVs, de la naturaleza y del objeto con el que se equipe, pero en muchos combates podremos comparar diversas formas de criar a *Leavanny* lo que le dará a nuestra base de datos un conjunto más rico de ejemplos.

Leavanny es de tipo *Bicho-Planta*, y en la figura 20 podemos ver las debilidades y resistencias defensivas de este tipo.

Figura 20. *Debilidades y resistencias defensivas de Leavanny*

Daño recibido		Tipos
x4	Superdébil a:	FUEGO, VOLADOR
x2	Débil a:	BICHO, HIELO, ROCA, VENENO
-	Daño normal	ACERO, DRAGÓN, FANTASMA, HADA, NORMAL, PSÍQUICO, SINIESTRO
½	Resistente a:	AGUA, ELÉCTRICO, LUCHA
¼	Superresistente a:	PLANTA, TIERRA
0	Inmune a:	*Ninguno*

Estas debilidades y resistencias corresponden a la última generación bajo condiciones normales de combate.

Fuente: Pagina de Leavanny en Wikidex

Podemos ver que *Leavanny* es débil defensivamente a 6 tipos, de los cuales es superdébil a *fuego* y *volador* (si recibe un ataque de estos dos tipos por parte del rival recibirá 4 veces el daño que recibiría sin tener en cuenta los tipos). También vemos que tiene 5 resistencias, dos de las cuales son superresistencias. Por lo tanto, el tipo *Bicho-Planta* está bastante balanceado en cuanto a resistencias y debilidades por lo que será perfecto para nuestro estudio. Muchos pokémons siempre llevan un ataque de cada uno de sus tipos. En el caso de Leavanny los tipos de ataques que obtendrían la mejora STAB serán los ataques de tipo *planta* y los de tipo *bicho*. En la figura 21 podemos ver la cobertura de efectividad que la combinación de estos dos tipos genera.

Figura 21. *Efectividad de la combinación ofensiva Bicho-Planta*

Fuente: Pagina de Pokedex Coverage[13]

4.5 Creación del dataset de combates

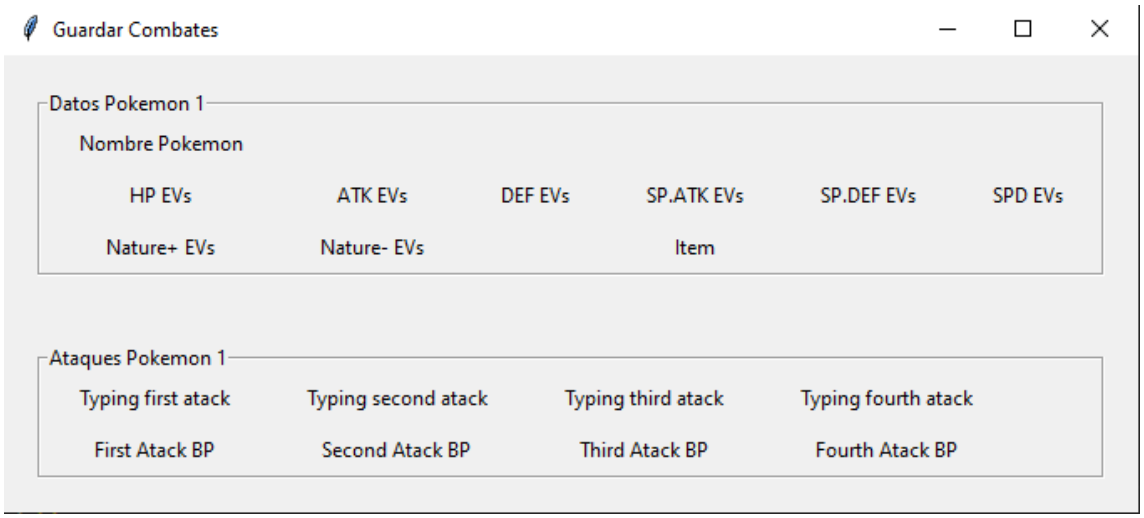
Para crear el dataset de combates implementaremos un nuevo programa llamado *Combates.py*, que es una ampliación del programa *Prototipo1.py* y que utilizará los datos de los nuevos pokémons seleccionados para nuestro problema. Este dataset será bastante más grande que el que realizamos para el prototipo por lo que insertar todos los valores por consola puede resultar incómodo y poco óptimo. Lo mejor sería disponer de una interfaz gráfica para nuestro programa en la que pudiésemos poner todos los datos del combate a la vez y que después almacenase esos datos solamente clicando un botón. De esta manera sería más fácil guardar diversas variaciones de los combates ya que no tendríamos que correr el programa para cada uno de ellos y solamente haría falta modificar los parámetros del combate anterior y volver a guardar. Esta implementación, añadiendo algunos cambios, constituirá la interfaz final en la que el usuario podrá insertar los datos del combate que quiera predecir.

Para la creación de nuestra interfaz he utilizado el paquete de Python *tkinter* [14], que contiene los comandos necesarios para la creación de scrolllbars, Combo boxes y otros widgets útiles para insertar nuestros datos. Lo primero que vamos a hacer es crear nuestra ventana a la que llamaremos *window* y a la que pondremos el título “Guardar combates”. Para distribuir los diferentes widgets crearemos diversos frames y labels para poder simular la distribución que tenemos. Tanto los frames como los labels son elementos que sirven para que el programa sea comprensible de manera visual.

Empezaremos por los datos del primer pokémon: necesitaremos insertar el nombre del pokémon, como queremos distribuir sus EVs, cuál es su naturaleza y que objeto tiene equipado.

También tendremos que especificar el tipo y el poder de los cuatro ataques que ha aprendido y podríamos añadir también si este pokémon es el vencedor del combate o no. Para ello dividiremos estos datos en dos frames diferentes uno para el nombre del pokémon, los EV, naturaleza y objeto y otro frame para los datos de sus ataques. En la figura 22 podemos ver visualmente el resultado.

Figura 22. *Primera aproximación interfaz grafica*



Fuente: Elaboración propia

Ahora necesitamos decidir los widgets que queremos usar para cada uno de los campos que hemos marcado con un label.

- **Nombre pokémon:** Para el nombre del pokémon podemos decidir entre dos opciones: la primera sería un *Entry* de texto normal en el que escribir una palabra que se guarda como *string*, la segunda opción sería una Combo box, que consiste en una lista de elementos entre los cuales se puede elegir. Mientras estemos trabajando con campos que solo tengan unos pocos valores una Combo box será lo más optimo ya que elimina la posibilidad de equivocarse al escribir los datos que se quieren entrar, pero con campos que tengan muchas opciones las listas pueden resultar excesivamente largas y puede ser difícil encontrar lo que se está buscando. Dado que en nuestro caso en nuestra base de datos tenemos 389 pokémons, un *Entry* de texto normal será la mejor opción.

- **EV en cada Stat:** Para los EV lo mejor será utilizar una Spinbox, un widget especialmente pensado para valores numéricos dentro de un cierto rango. Consiste en un Entry numérico al que se le da un valor inicial y que tiene dos flechas en el lado derecho para hacer aumentar o disminuir este valor. Al tratarse de un Entry también permite ingresar el valor de forma escrita pero las flechas permiten ajustar el valor en un par de clicks por lo que en general son preferibles para pequeños incrementos. Como el número de EV que se puede invertir en un Stat es de 252, y cada 4 de estos 4 generarán un aumento de 1 punto en el Stat final, nuestro valor mínimo e inicial del Spinbox será 0 y nuestro máximo 252 y añadiremos un incremento de 4 unidades cada vez que se utilizan las flechas.

- **Naturaleza:** En este campo tendremos que determinar qué campo de los Stats aumentará por la naturaleza y cual disminuirá. Cada uno de estos dos campos se podrá elegir entre ATK, DEF, SP.ATK, SP.DEF y SPD sin poder elegir el mismo en ambos. Por lo tanto, lo mejor será utilizar dos Combo box.

- **Objetos:** Ya hemos explicado anteriormente que para nuestro problema solo tendremos en cuenta 6 objetos por lo que utilizaremos una Combo box.

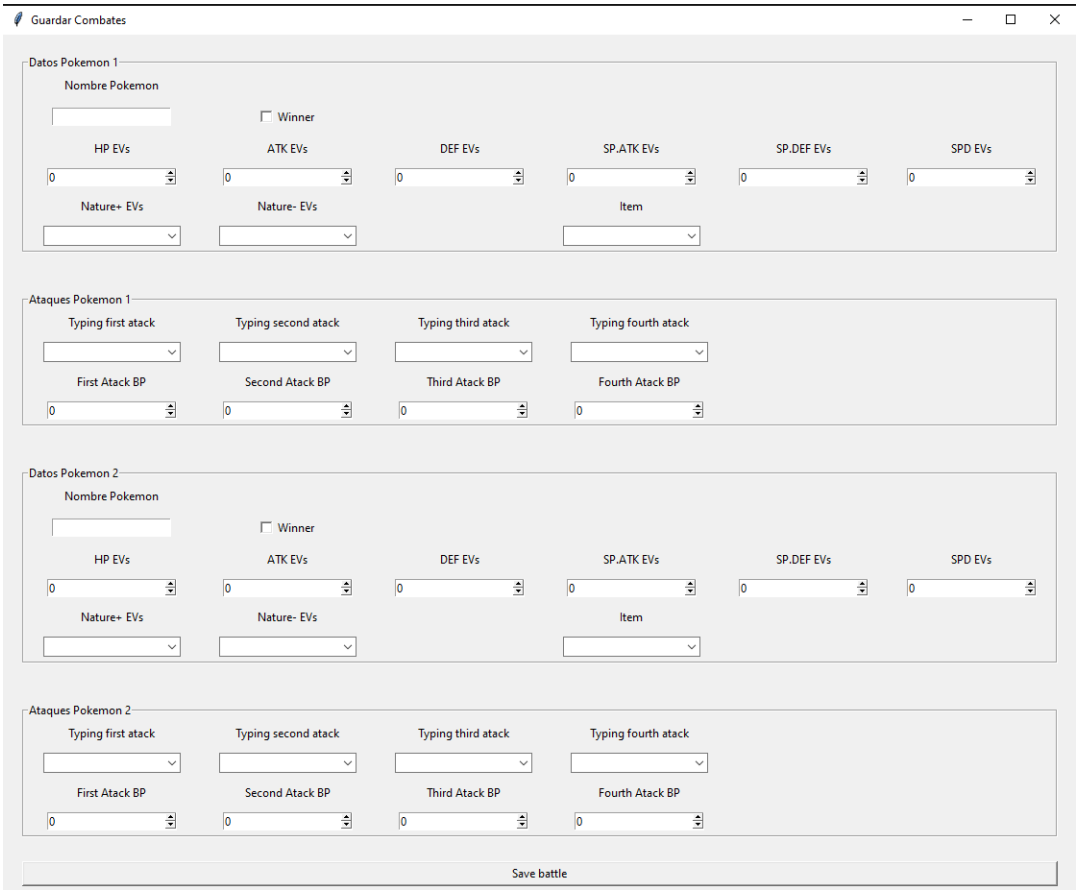
- **Tipo ataque:** Para cada ataque podremos elegir entre uno de los 18 tipos y los 5 efectos de estado, es decir un total de 23 opciones. Pese a ser una lista más larga que la de los objetos y las naturalezas utilizaremos una Combo box ya que sigue siendo suficientemente sencillo encontrar lo que se busca.

- **Poder del ataque:** Necesitaremos que se introduzca el Poder de cada ataque. El poder de la mayoría de los ataques que se existen suelen variar entre 40 y 150, con solamente unos pocos superando los 200 bajo algunas condiciones específicas. Por lo tanto, utilizarmos un Spinbox que tome valores entre 0 y 300.

- **Vencedor del combate:** Para mostrar quién es el vencedor del combate utilizaremos un CheckButton, widget que consiste en un botón seleccionable que devolverá un valor si esta seleccionado y otro si esta sin seleccionar, al lado del nombre del pokémon. En nuestro caso devolverá 1 si seleccionado y 0 si no.

La interfaz gráfica se ha implementado con los widgets que queremos utilizar añadiendo un botón que almacene los datos que se han insertado al pulsarlo. En la figura 23 podemos ver la interfaz gráfica finalizada.

Figura 23. *Captura de nuestra interfaz gráfica*



Fuente: Elaboración propia

Una vez implementada la interfaz gráfica debemos modificar el programa utilizado en el prototipo para guardar todos los datos nuevos. El funcionamiento general de este programa es similar, pero le añadimos el almacenamiento de los ataques, los objetos y en vez de los stats base guardamos los stats finales utilizando la fórmula que hemos presentado anteriormente. Para realizar el encoding de los ataques se utilizará el mismo método que el que hemos utilizado para los tipos, con la única diferencia de que en vez de poner un 1 si el pokémon es de un tipo pondremos el poder del ataque si se trata de un ataque físico o especial, y un 1 si se trata de un ataque de estado. En cuanto a los objetos se realizará un *One Hot encoding* para las seis clases

para cada uno de los pokémons y, si el pokémon lleva equipado uno de los objetos que aumentan sus estadísticas, se aumentará el valor en la cantidad especificada directamente al almacenarlos. Es importante remarcar que la ejecución de esta última parte del programa estará definida dentro de una función llamada *save battle* que se ejecutará cada vez que se pulsa el botón "Save Battle" que se encuentra en la parte más baja de nuestra interfaz gráfica.

4.6 Introducción a Random Forest y Gradient Tree Boosting en Sklearn

Anteriormente he explicado el funcionamiento del paquete de *Sklearn* para la creación de árboles de decisión. Sklearn también dispone de la biblioteca *ensamble methods* que ofrece métodos basados en otros más básicos mejorando su eficiencia. Existen principalmente dos clases de *ensamble methods*:

- *Averaging methods*: métodos que trabajan con el promedio de varias iteraciones diferentes.
- *Boosting methods*: métodos que crean diferentes modelos de manera secuencial donde cada uno intenta disminuir el error generado.

Tanto el algoritmo de *Random Forest* como el algoritmo de *Gradient Tree Boosting* se basan en el algoritmo de Árbol de Decisión, siendo *Random Forest* un *averaging method* y *Gradient Tree Boosting* un *boosting method*.

4.6.1 Random Forest

El algoritmo de *Random Forest* crea diversos árboles de decisión que utiliza para clasificar el ejemplo y finalmente devuelve como respuesta el resultado que haya salido más veces. Este modelo está pensado para combatir el overfitting que se genera en la creación de árboles de decisión, utilizando dos fuentes de aleatoriedad que disminuirán las variaciones que pueden sufrir los árboles pertenecientes al bosque. La primera fuente de aleatoriedad será que no trabajaremos con todos los datos de nuestra muestra, sino que cada uno de los árboles del bosque tomará una sub-muestra aleatoria de la muestra final. La segunda fuente afectará a la creación de nuestras separaciones. Como hemos visto, en la creación de un árbol de decisión se estudia el valor $G(Q_m, \theta)$ de un nodo para cada valor de cada uno de los campos y se elige la separación con el valor más bajo. En el algoritmo de *Random Forest* cada árbol se creará de

manera similar, pero en vez de tener en cuenta todos los campos solo utilizaremos un subgrupo aleatorio de ellos.

Los comandos para generar el modelo de *Random Forest* serán similares a los que hemos utilizado para la creación del árbol de decisión, pero esta vez tenemos más opciones a elegir que determinarán la forma de nuestro bosque:

- n_estimators: Este valor determina el número de árboles que se van a generar en el bosque. El valor por defecto es de 100 árboles, lo que parece un tamaño apropiado para nuestro problema, pero también estudiaremos los resultados para un número mayor y menor de árboles.
- Max_features: Este valor indica el número de campos que se van a seleccionar de manera aleatoria para la selección de cada separación. El valor por defecto es utilizar la raíz cuadrada de el número total de campos, en nuestro caso tendríamos 106 por lo que serían 10. La segunda opción que estudiaremos será utilizar el logaritmo en base 2 del total, en nuestro caso 6.
- Bootstrap: Este parámetro determina si queremos utilizar una sub-muestra aleatoria de los datos de entrenamiento para cada uno de los árboles o si queremos utilizar todos los ejemplos de los datos. Como en nuestro caso el número de ejemplos que hemos introducido es bastante reducido en comparación con la cantidad de diferentes casos que se observan, estudiaremos los resultados utilizando tanto el total como un subgrupo.

4.6.2 Gradient Tree Bosting

El algoritmo de *Gradient Tree Boosting* [15] es un ejemlo de boosting method. Si en el algoritmo de *Random Forest* se crean árboles independientes para, finalmente, escoger como predicción final el resultado más popular entre los árboles del bosque, en este método se crean diversos árboles para minimizar los errores del árbol generado anteriormente. Estos árboles serán poco profundos por lo que sus predicciones pueden ser más débiles pero su combinación terminará aportando una predicción muy fuerte. A diferencia del modelo de *Random Forest,* que está pensado para prevenir el *overfitting*, el modelo de *Gradient Tree Boosting* está pensado para identificar relaciones más complejas y suele tender al *overfitting*.

- El modelo empieza realizando una predicción inicial que utiliza para calcular los residuos, que son la diferencia entre el valor real y el valor de la predicción.

- A continuación, se crea el próximo árbol como un modelo para predecir los residuos generados en el paso anterior y calcula el valor de cada hoja dependiendo de los residuos que se han clasificado en ella.
- Se repite este proceso las n veces que hayamos especificado, creando n estimators.
- Finalmente, la predicción final será la suma de nuestra predicción inicial más el valor obtenido en cada estimator multiplicado por el *learning rate*, donde el *learning rate* es un valor que determina la aportación de cada árbol.

Algunos comandos importantes al generar un modelo de Gradient Tree Boost, utilizando la función *GradientBoostingClassifier* del paquete *sklearn.ensemble,* son:

- learning_rate: Este valor que determina la aportación de un árbol. Puede variar entre 0 e infinito, pero nosotros utilizaremos el valor 0.1.
- n_estimators: Este valor determina el número de estimators que se crearán para el modelo. El valor por defecto es de 100 árboles y es el que utilizaremos. Un valor elevado de estimadores es importante para combatir el overfitting que se puede crear con este método.
- Max_depth: Este valor determina la profundidad de los estimators. El valor por defecto es max_depth=3 y es el que utilizaremos inicialmente ya que unos árboles demasiado profundos podrían producir más overfitting.
- Criterion: Este campo determina el criterio que se utilizará para determinar la calidad de cada separación durante la creación del árbol. Podremos seleccionar entre Friedman_mse y squared_error. Estos criterios cumplirán con la misma función que el *Gini* que hemos explicado anteriormente pero también están pensados para tener en cuenta la proximidad al resultado que buscamos. Nosotros utilizaremos el criterio Friedman_mse que en general se considera mejor ya que aporta una mejor aproximación.

5. Resultados obtenidos

Todos los programas y datasets utilizados para este trabajo se pueden enconrar en:

https://drive.google.com/drive/folders/14FBfbn9s0TgNRfrAtN1-4AXDzVCQxcRE?usp=sharing

5.1 Análisis de los resultados observados en cada método

Estudiaré en esta sección los resultados obtenidos con los diferentes modelos para decidir cual funciona mejor para este proyecto. Para poder realizar este estudio he partido los datos en dos grupos: datos de entrenamiento con los que entrenar el modelo y datos de testeo sobre los que probar su funcionamiento.

He utilizado dos métodos diferentes para generar estas particiones:

- **Train-test Split:** La función *train_test_split* de la biblioteca *sklearn.model_selection* se usa para separar el dataset en los grupos de entrenamiento y test. Esta función recibirá como entrada los X e y que hemos utilizado para la creación del modelo (nuestor dataset contiene 583 ejemplos) y creará *train_X*, *train_y*, *val_X*, *val_y*. Donde *train_X* y *train_y* se utilizarán para generar el modelo y *val_y* se utilizará para estudiar los resultados. El grupo destinado al entrenamiento del modelo contendrá el 75% de los datos de nuestro *dataset* original, en nuestro caso 437 ejemplos, mientras que el grupo destinado al testeo tendrá 25% de los datos, en nuestro caso 146 (Figura 24). Este método permite poder estudiar de forma sencilla los resultados obtenidos comparándolos con los resultados esperados para *val_y* de una manera simple y también estudiar los grupos utilizados en la separación. El único inconveniente es que los resultados obtenidos pueden variar mucho dependiendo de las particiones generadas. Para mitigar este problema con este método he generado tres particiones diferentes, usando el comando *random_state,* para poder comparar los resultados.

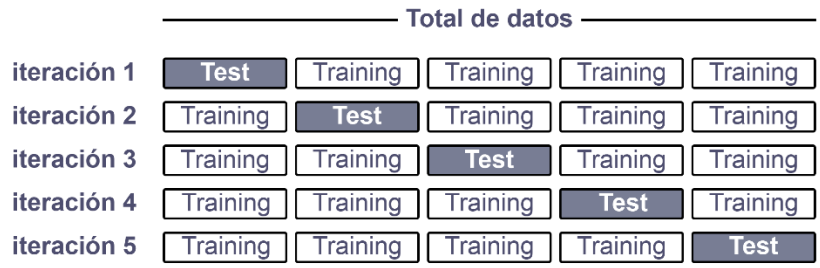
Figura 24. *Esquema del método train-test split*

Fuente: Elaboración propia

- **K-fold cross-validation:** Este método parte nuestro *dataset* en K subconjuntos. Se realizan K iteraciones en las que, en cada iteración, uno de los subconjuntos sirve como grupo de testeo y la unión de los otros K-1 sirve como grupo de entrenamiento. Cada iteración tendrá como grupo de testeo un subconjunto diferente por lo que, al final de las K iteraciones, cada subconjunto habrá sido utilizado una vez como grupo de testeo y

K-1 veces como una parte del grupo de entrenamiento (Figura 25). Una vez se hayan realizado las K iteraciones de la función se devolverá la media de los resultados obtenidos. Para la realización de esta separación utilizamos la función *cross_val_score* de la biblioteca *sklearn.model_selection*. Esta función recibirá como datos de entrada nuestros datos X e y, el modelo sobre el que vamos a estudiar los resultados, el valor de K y la prueba que deseamos realizar, en nuestro caso estudiaremos la precisión de la predicción, y tendrá como output los resultados de las iteraciones y la media final. Este método es mucho más preciso en comparación al *train-test Split,* pero es mucho más lento desde el punto de vista computacional ya que se realizan K iteraciones. También es importante mencionar que la función *cross_val_score* genera todas las particiones de forma interna por lo que es muy difícil trabajar con ellas en comparación con el funcionamiento de *train-test split* que genera directamente *train_X*, *train_y*, *val_X* y *val_y*.

Figura 25. *Esquema del K-fold cross-validation para K=5*



Fuente: Elaboración propia

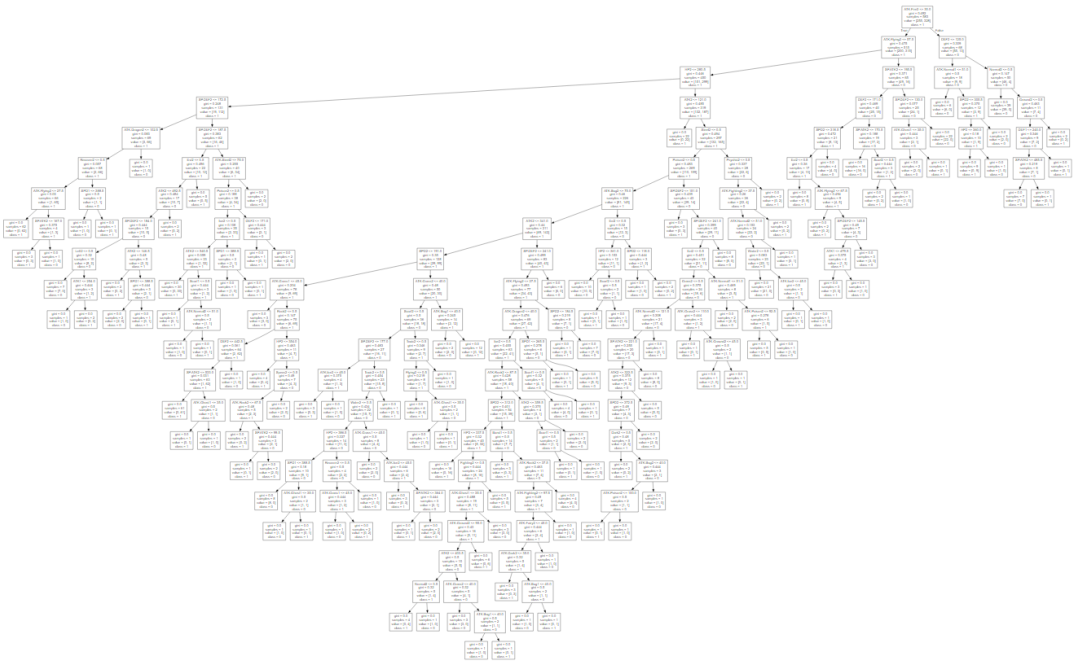
Utilizaré estos dos métodos para estudiar el rendimiento de los tres modelos que he presentado en diferentes escenarios:

- El primero utiliza solamente los datos de los combates de *Leavanny*.
- En el segundo añadiremos 200 combates, donde Leavanny no participa, a los datos de entrenamiento del modelo para simular el funcionamiento de un *dataset* con combates de muchos pokémons diferentes.
- Finalmente añadiremos nuevos campos al *dataset* de manera similar a como hemos añadido en el prototipo los campos *Ofense1, Ofense2* y *Difofense*.

5.1.1 Resultados para el algoritmo de árbol de decisión

En nuestro prototipo ya hemos generado un árbol de decisión utilizando *Sklearn,* pero utilizado una cantidad mucho menor de datos. En el caso que vamos a estudiar ahora tenemos un total de 583 combates almacenados que, después de aplicar el *train_test_split*, se transforman en 437 combates destinados a entrenar el modelo y 146 destinados al testeo. Estudiemos ahora el árbol que se genera. En la figura 26 podemos observar el árbol generado utilizando random_state=0 en el train_test_split.

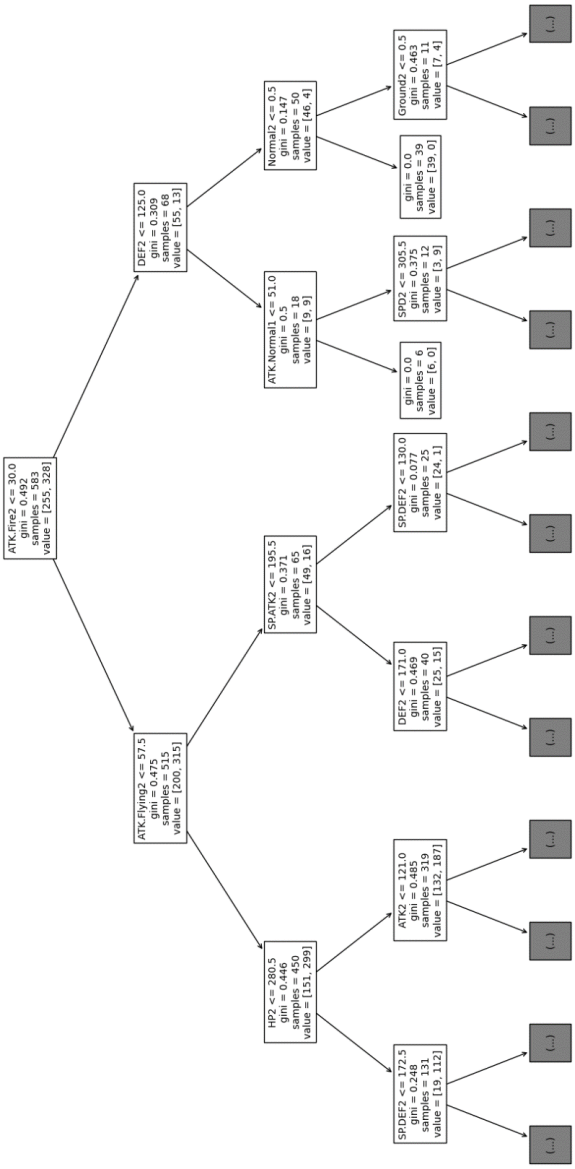
Figura 26. *Árbol de decisión para combates de Leavanny*



Fuente: Elaboración propia

Claramente este árbol es mucho más grande que el que habíamos generado en nuestro prototipo por lo que nos vamos a limitar a analizar los primeros splits del árbol, que serán los más importantes ya que los nodos tendrán mayor número de ejemplos.

Figura 27. *Primeras particiones del árbol de decisión para combates de Leavanny*



Fuente: Elaboración propia

En la figura 27 podemos ver que las primeras particiones de nuestro árbol de decisión generado comprueban si el pokémon rival tiene un ataque de fuego o uno volador. Esto tiene sentido ya que estos dos tipos son súper-efectivos contra *Leavanny*, es decir, hacen x4 de daño. Estas primeras particiones separan inicialmente sólo una pequeña parte de nuestros ejemplos, las

cuales presentan un patrón muy marcado, pero dejan la mayoría de casos todavía por clasificar. Si miramos la figura 26 vemos que, efectivamente, la parte de la derecha del árbol es mucho menos profunda que la de la izquierda. Esto se debe a que el algoritmo ha separado primero los grupos más homogéneos, los cuales presentan un patrón más claro, y ha dejado los más complicados para el final.

Aunque las separaciones que se han utilizado para crear el árbol parecen tener sentido, lo que interesa para nuestra aplicación es ver la tasa de precisión que el árbol tiene sobre los datos de testeo. Para ello he utilizado el método de *K-fold cross-validation* con K=10 para estudiar la precisión del modelo. En la figura 28 podemos ver los resultados obtenidos.

Figura 28. *Resultados de 10-fold cross-validation para medir la precisión del modelo Árbol de Decisión*

```
Los resultados obtenidos en cada iteracion son:
[0.77966102 0.55932203 0.50847458 0.39655172 0.4137931  0.56896552
 0.62068966 0.5862069  0.68965517 0.81034483]
La media de las iteraciones es: 0.5933664523670368
```

Fuente: Elaboración propia

Observamos que los resultados obtenidos en las diversas iteraciones presentan muchísima variación. La iteración con menos precisión seria la cuarta, con 39.6% de precisión, mientras que la iteración con más precisión sería la última, con 81%. Una variación tan grande hace que finalmente la media termine siendo de 59.3% lo que no parece un valor muy bueno.

A continuación, he estudiado el efecto de añadir los 200 combates extra a nuestros resultados, utilizando el método *train-test Split*, puesto que este método nos permitirá añadir estos datos extra a nuestro *train_X* para generar un segundo modelo y poder comprobar la precisión sobre el mismo grupo de testeo, *val_y*. Como hemos visto que los resultados de nuestro modelo tienen mucha variabilidad, estudiaré la precisión para tres casos diferentes usando los random states 0, 1 y 2 al crear el split.

Figura 29. *Resultados de la precisión utilizando los random_state 0, 1 y 2*

```
La precisión usando los datos originales es:  0.726027397260274
La precisión usando los datos ampliados es:  0.6232876712328768
```

```
La precisión usando los datos originales es:  0.6917808219178082
La precisión usando los datos ampliados es:  0.6232876712328768
```

```
La precisión usando los datos originales es:  0.6986301369863014
La precisión usando los datos ampliados es:  0.6643835616438356
```

Fuente: Elaboración propia

La figura 29 muestra los resultados que hemos obtenido al realizar estas pruebas, indicando claramente que los resultados ampliados obtienen siempre una precisión notablemente más baja que los datos que solamente contienen combates de *Leavanny*.

Para poder valorar donde se están cometiendo estos errores, tanto en el modelo generado con los datos originales como en el ampliado, realizamos una *Confusion Matrix* para cada una de las pruebas. Una *Confusion Matrix* es una matriz que muestra el número de ejemplos con resultado 1 predichos correctamente (True positives), el número de ejemplos con resultado 1 predichos erróneamente (False positives), el número de ejemplos con resultado 0 predichos correctamente (True negatives) y el número de ejemplos con resultado 0 predichos erróneamente (False negatives). Las filas de esta matriz corresponden al resultado real o *True Label*, mientras que las columnas corresponden el resultado predicho. Por lo tanto, los True negatives se muestran arriba a la izquierda, los False positives arriba a la derecha, los True positives abajo a la derecha y los False negatives abajo a la izquierda (Figura 30)

Figura 30. *Confusion Matrix*

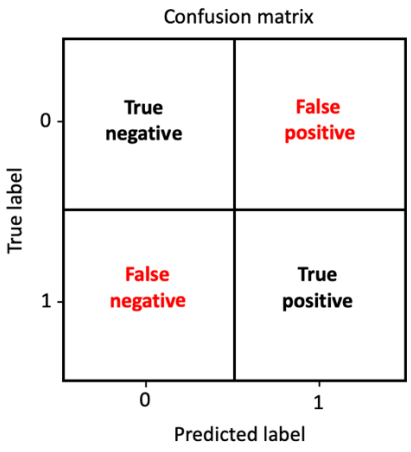
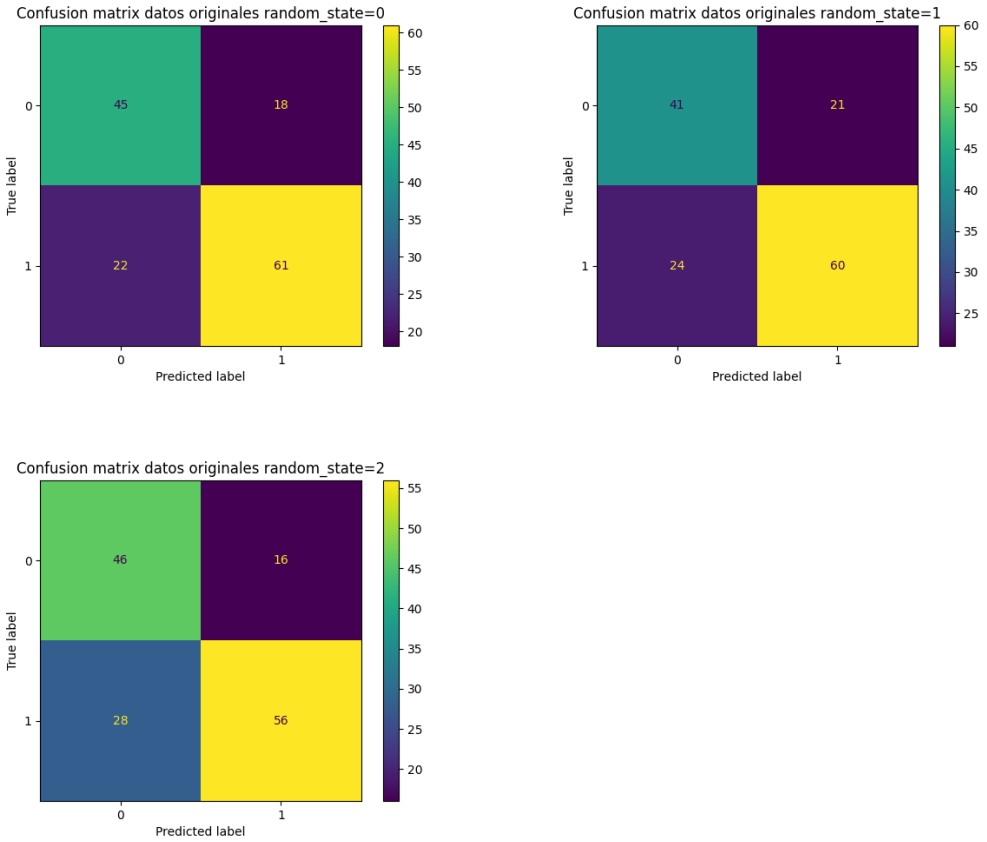
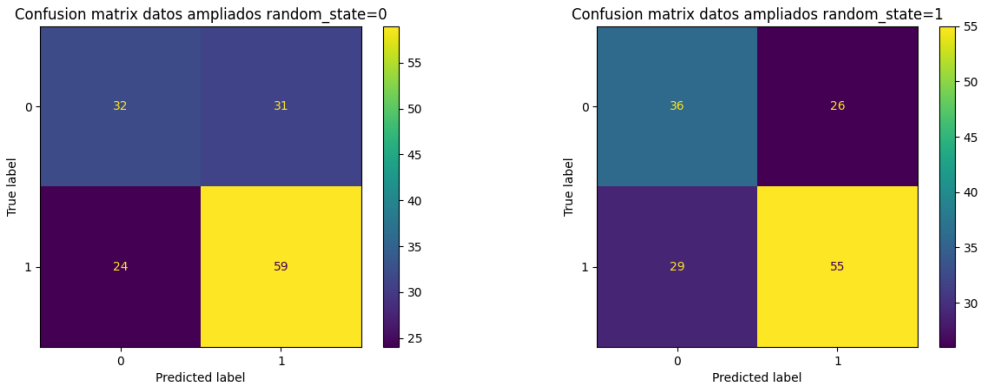


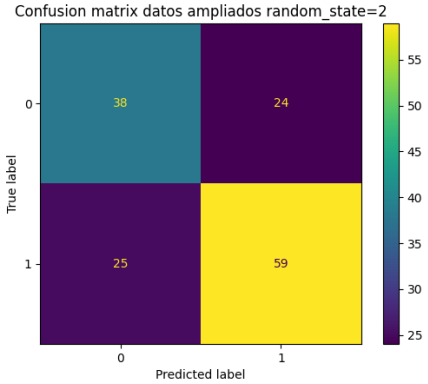
Figura 31a. *Confusion Matrix del modelo generado con los datos originales utilizando los random_state 0, 1 y 2*



Fuente: Elaboración propia

Figura 31b. *Confusion Matrix del modelo generado con los datos ampliados utilizando los random_state 0, 1 y 2*



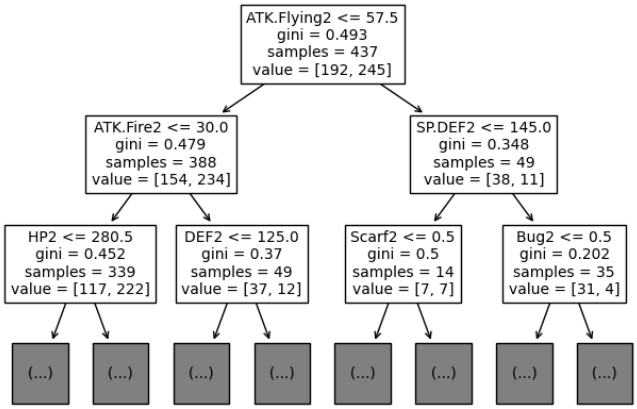


Fuente: Elaboración propia

En las figuras 31a y 31b podemos ver las *Confusion Matrix* de las diferentes pruebas. Un análisis de los resultados obtenidos para el modelo generado con los datos originales (Figura 31a) muestra que el número de errores cometidos por los False Positives suelen estar sobre el 30%-35% del número total de ejemplos, mientras que el número de errores por False Negatives se encuentran entre el 25%-30%. Por lo tanto, podemos decir que los errores están distribuidos equitativamente independientemente del resultado obtenido. Si miramos ahora las matrices de la figura 31b, las que pertenecen al modelo creado a partir de los datos ampliados, podemos ver que el porcentaje de False Negatives se mantiene similar al primer modelo pero que se cometen muchos más errores por False Positives.

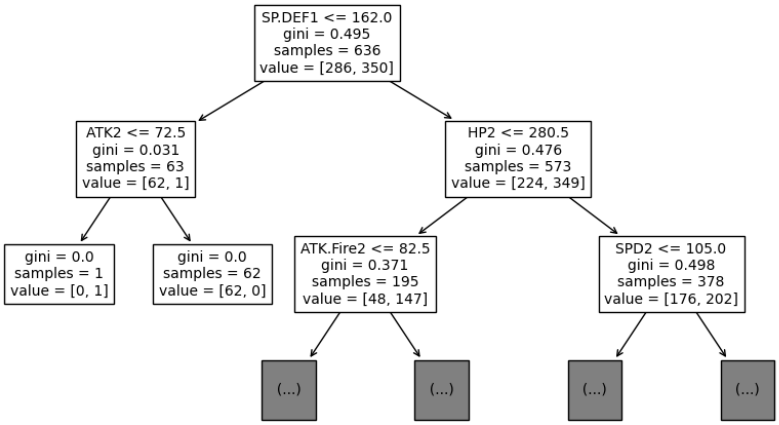
Para entender mejor el porqué de estas diferencias vamos a analizar las primeras particiones de los árboles generados usando el caso con random_state=0.

Figura 32a. *Primeras particiones del árbol generado con los datos originales y random_state=0*



Fuente: Elaboración propia

Figura 32b. *Primeras particiones del árbol generado con los datos ampliados y random_state=0*



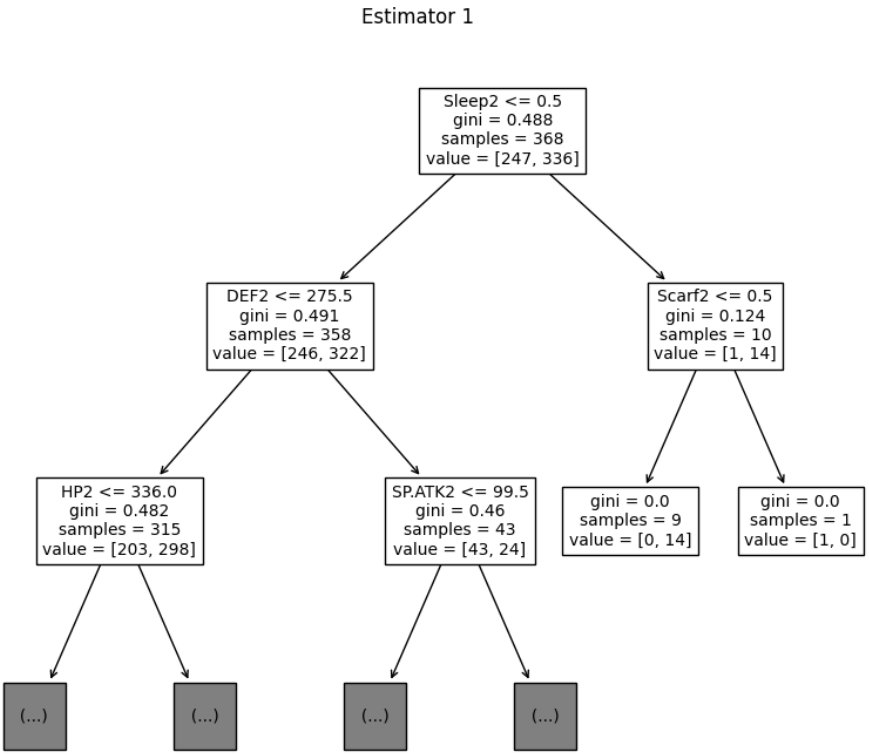
Fuente: Elaboración propia

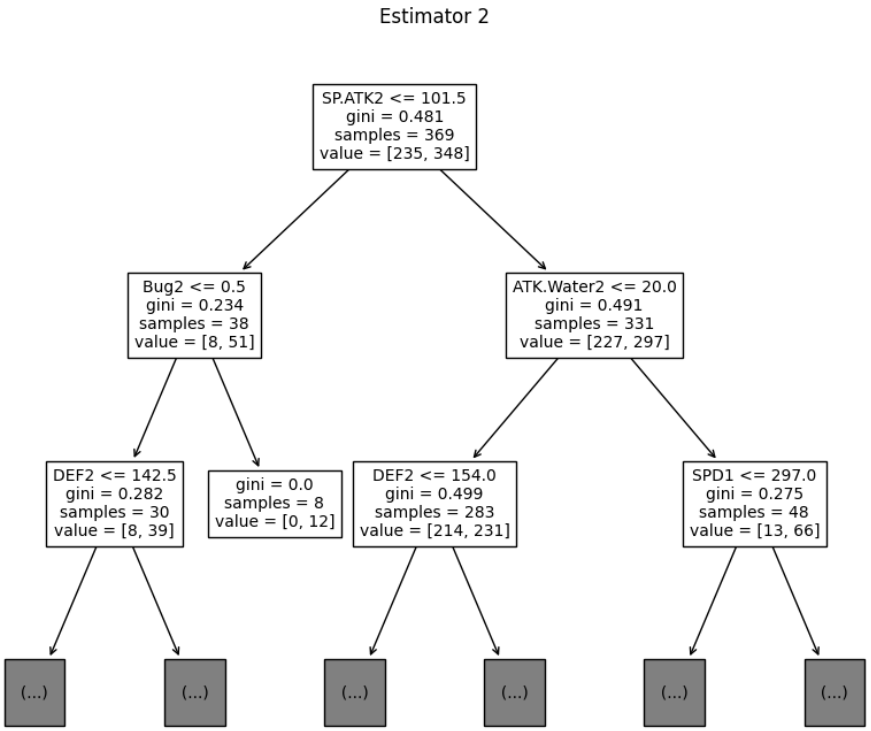
En la figura 32a podemos observar las primeras separaciones del árbol generado con *random_state=0* utilizando los datos originales de nuestro *split*. Vemos que estas separaciones son similares a las que teníamos en el árbol de la figura 26 y que dan relevancia en los combates a algunas de las efectividades y características importantes que son específicas del pokémon *Leavanny*. En cambio, si miramos la figura 32b, que muestra el árbol generado con los datos ampliados vemos rápidamente que algunas de esas efectividades y características se han difuminado y en su lugar tenemos particiones mucho más generales. Lo más interesante es que la primera partición se hace utilizando la defensa especial del pokémon titular y que, después de otra separación por el ataque del pokémon rival, se crean directamente dos hojas. Esto no parece tener mucho sentido a priori ya que la defensa especial no está relacionada directamente con el ataque y todavía existen muchos factores que pueden determinar el resultado de un combate. Por lo tanto, podemos empezar a pensar que un dataset con combates de diversos pokémons en cualquiera de las posiciones tendrá más problemas en encontrar las diversas relaciones y efectividades que pueden afectar a un pokémon especifico, pero esperaremos a ver los resultados para los otros dos modelos antes de formular una conclusión.

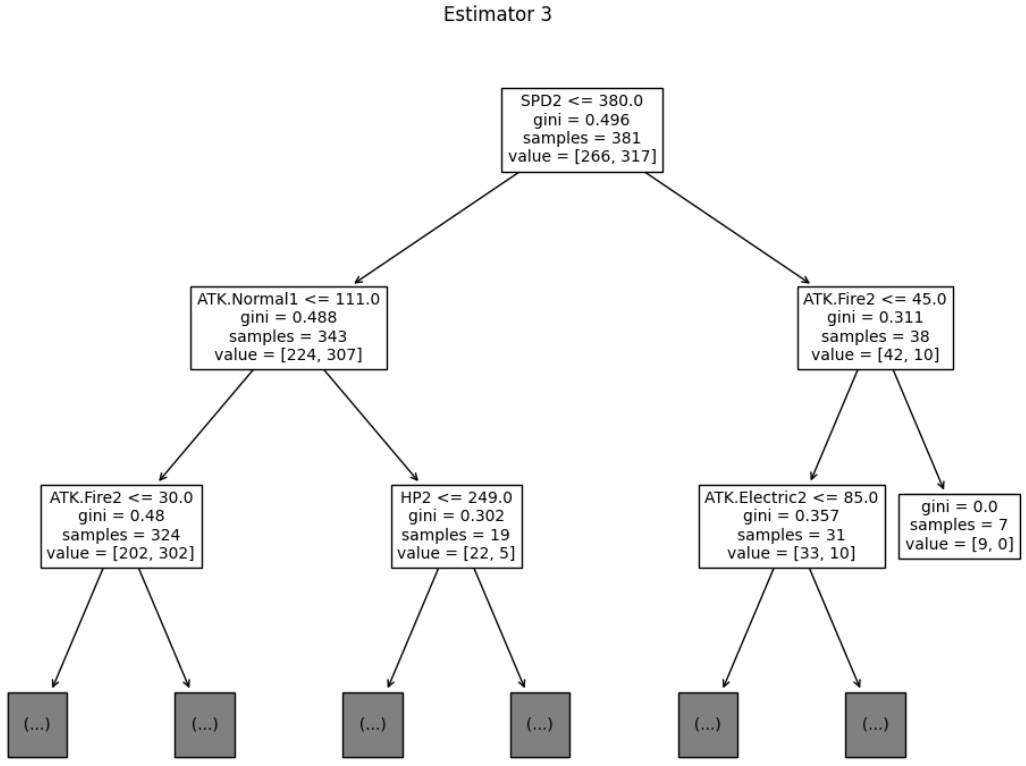
5.1.2 Resultados para el algoritmo de *Random Forest*

Estudiemos ahora los resultados obtenidos utilizando un modelo de *Random Forest.* Empezamos viendo las primeras separaciones de los tres primeros estimadores para poder observar un ejemplo de lo que hemos explicado en la sección 4.6.1

Figura 33. *Primeras particiones de los primeros estimadores de nuestro modelo de Random Forest*







Fuente: Elaboración propia

En la figura 33 podemos ver como cada *estimator* tiene un numero diferente de ejemplos y todas las separaciones también son diferentes. Esto se debe a que estamos utilizando *max_features="sqrt"* por lo que cada separación solamente tiene en cuenta 10 campos lo que permite que algunos campos que no tienen uso en un árbol de decisión, como por ejemplo ATK.Water2 (que no debería tener mucha importancia en un principio) se tengan en cuenta aunque solamente sea en unos pocos de los árboles del bosque. Como he dicho, el modelo utiliza 100 estimators eligiendo como predicción el resultado más popular entre los 100, lo que debería, supuestamente, aportar resultados más estables y con menos variación. En la figura 34 podemos ver los resultados del 10-fold cross-validation.

Figura 34. *Resultados de 10-fold cross-validation para medir la precisión del modelo Árbol de Decisión*

```
Los resultados obtenidos en cada iteracion son:
[0.76271186 0.55932203 0.44067797 0.22413793 0.37931034 0.63793103
 0.55172414 0.63793103 0.65517241 0.74137931]
La media de las iteraciones es: 0.5590298071303331
```

Fuente: Elaboración propia

Podemos observar que los resultados obtenidos son peores que los que se obtienen utilizando el modelo de Árbol de Decisión, teniendo una media de precisión más baja (55.9%) y una variabilidad más alta (entre el 22.4% para el peor resultado y un 76.2% para el mejor).

Comprobemos ahora los resultados obtenidos utilizando el *train-test split* comparado con los datos ampliados y las Confusion Matrix .

Figura 35. *Resultados de la precisión utilizando los random_state 0, 1 y 2*

```
La precisión usando los datos originales es:  0.7054794520547946
La precisión usando los datos ampliados es:  0.6643835616438356
```

```
La precisión usando los datos originales es:  0.6917808219178082
La precisión usando los datos ampliados es:  0.6917808219178082
```

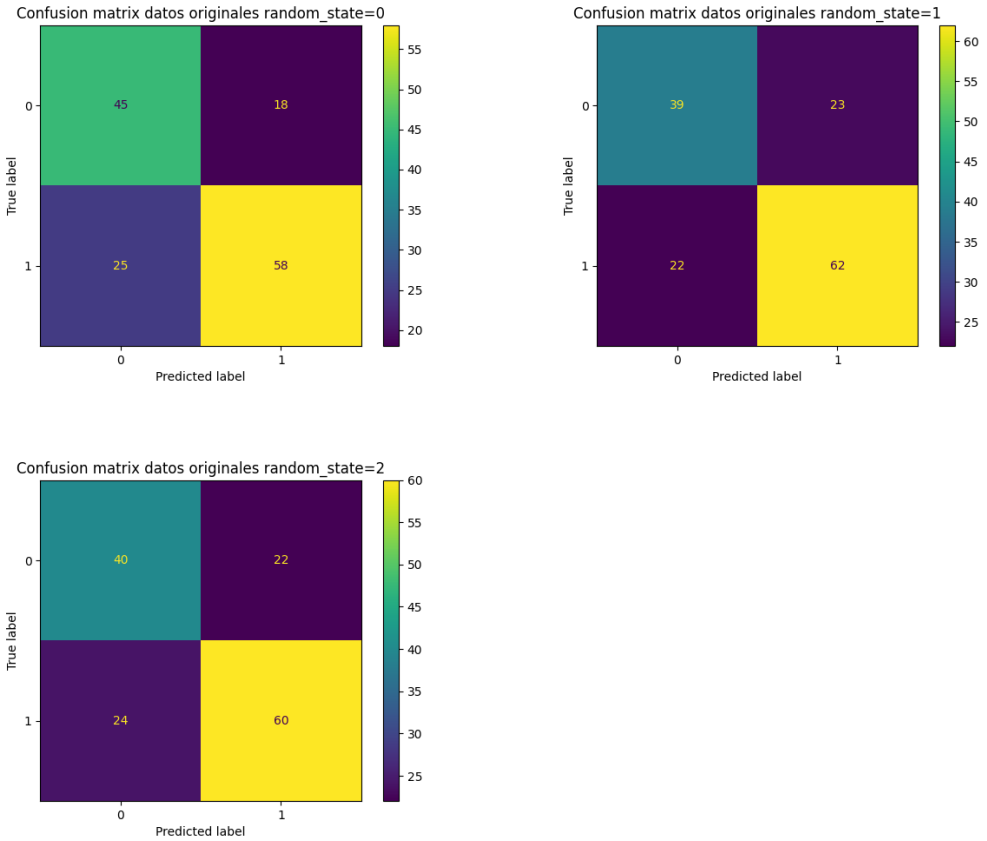
```
La precisión usando los datos originales es:  0.684931506849315
La precisión usando los datos ampliados es:  0.678082191780822
```

Fuente: Elaboración propia

En la figura 35 podemos ver las diferencias entre los resultados obtenidos utilizando los datos originales y los datos ampliados. Observamos que, de forma similar a lo que hemos visto en el modelo del Árbol de Decisión, la precisión obtenida utilizando los datos ampliados es peor de la que se obtiene utilizando los datos originales, aunque en este caso la diferencia entre los dos es mucho menor llegando incluso a ser iguales para random_state=1.

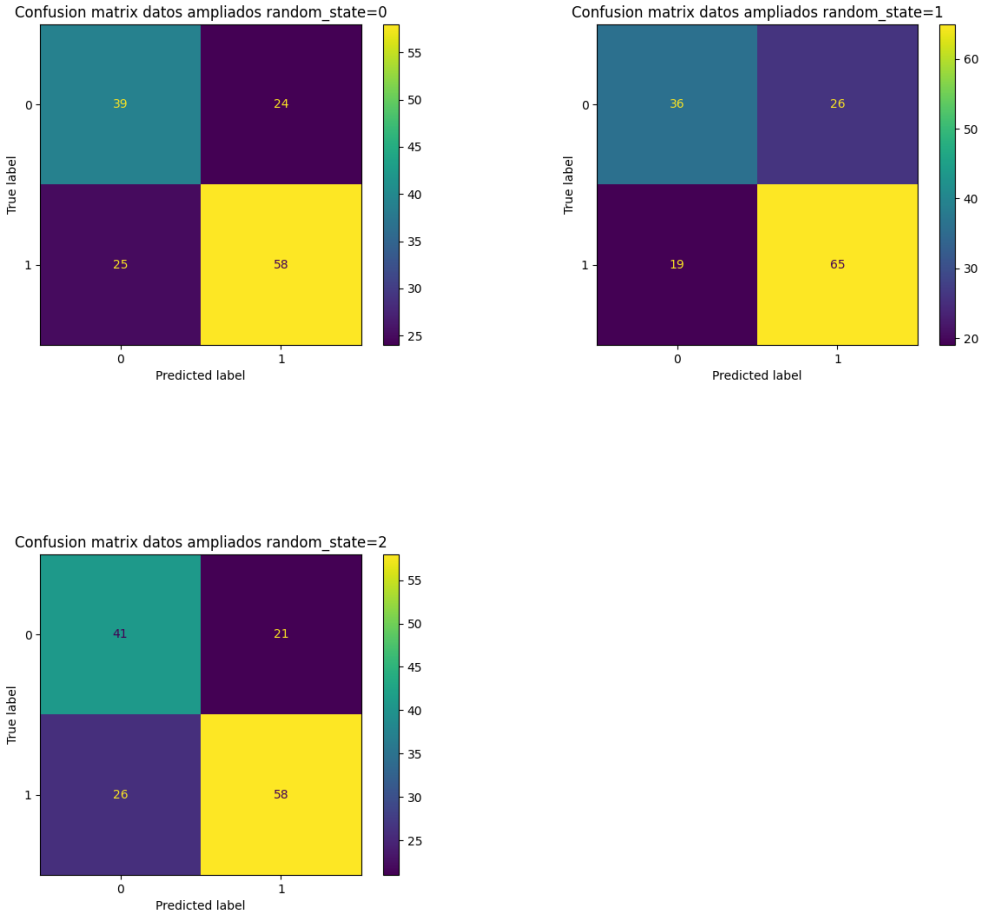
Observemos ahora las Confusion Matrix:

Figura 36a. *Confusion Matrix del modelo generado con los datos originales utilizando los random_state 0, 1 y 2*



Fuente: Elaboración propia

Figura 36b. *Confusion Matrix del modelo generado con los datos ampliados utilizando los random_state 0, 1 y 2*



Fuente: Elaboración propia

Los datos que obtenemos en las figuras 36a y 36b son bastante similares a los que hemos obtenido en las figuras 31a y 31b por lo que no los comentaremos mucho. El único random_state sobre el que podemos observar algo interesante es el random_state=1. Los resultados de la figura 35 para este random_state muestran que la precisión es exactamente la misma para los dos grupos de datos, por lo tanto, podríamos esperar que los dos modelos fueran iguales. Pero, de las Confusion Matrix para random_state=1, se deduce que los errores cometidos no son los mismos, ya que la matriz de los datos ampliados tiene un mayor número de False Negatives y un número menor de False Positives, confirmando que los modelos generados son diferentes.

En un principio esperaba que el modelo de *Random Forest* aportase mejores resultados que el modelo de *Árbol de Decisión* ya que está pensado para disminuir la variación en los resultados. En cambio, hemos obtenido que los resultados observados tienen una mayor variación y un peor rendimiento. También hemos visto que hay menos diferencia entre los resultados obtenidos con

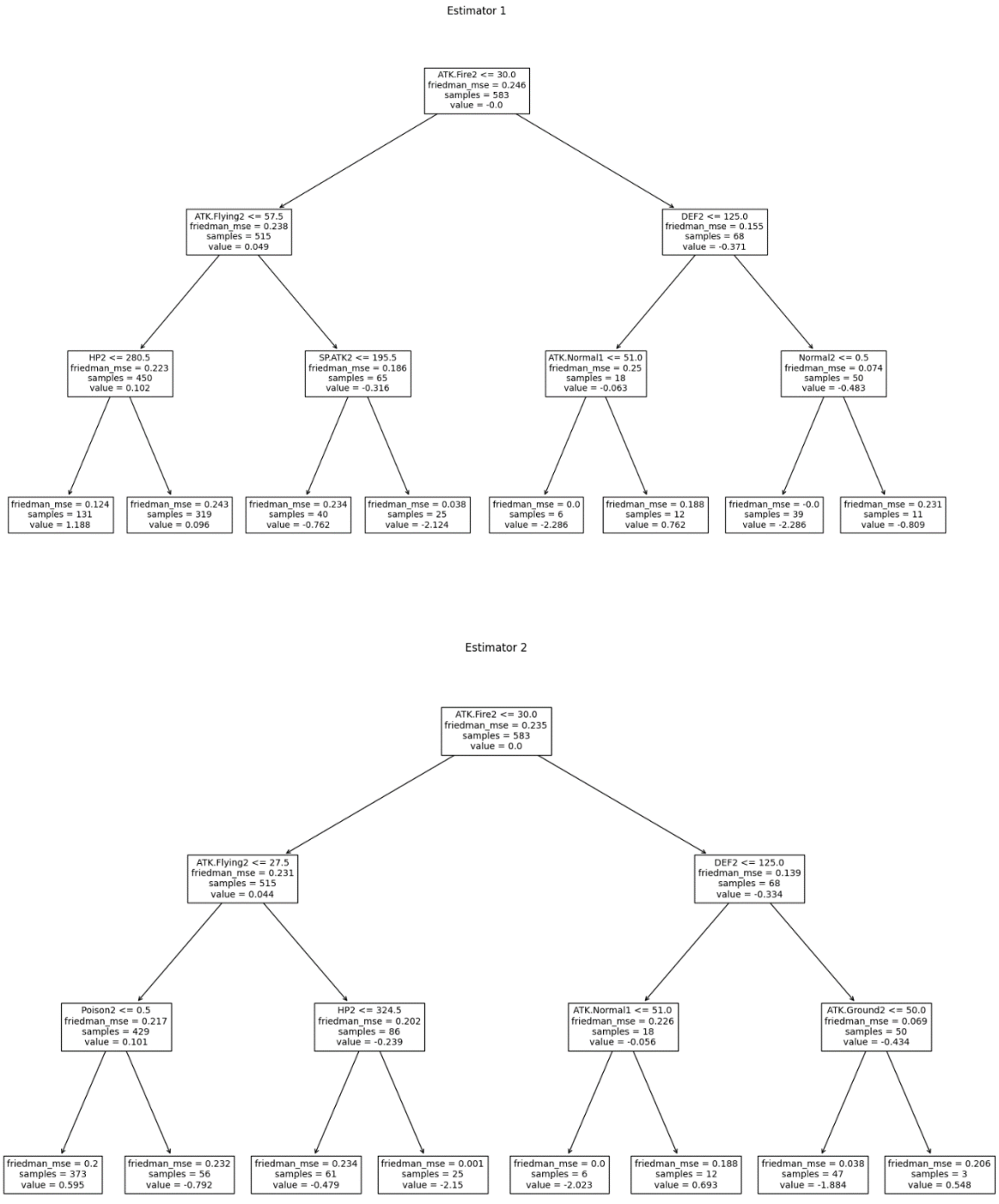
los datos originales y los datos ampliados. Como hemos dicho varias veces a lo largo de este trabajo el resultado de un combate pokémon puede verse determinado por algunas relaciones que solo toman importancia en algunos escenarios específicos. En la sección 5.1.1 hemos visto como algunas de estas relaciones parecían ser menos evidentes al añadir datos de diversos pokémons al dataset y los resultados obtenidos hacen pensar que la aleatoriedad del algoritmo de *Random Forest* en vez de disminuir la variación de los resultados complica la identificación de algunas de estas relaciones. Por lo tanto, según los resultados observados, el algoritmo de *Random Forest* sería un peor modelo para este problema específico.

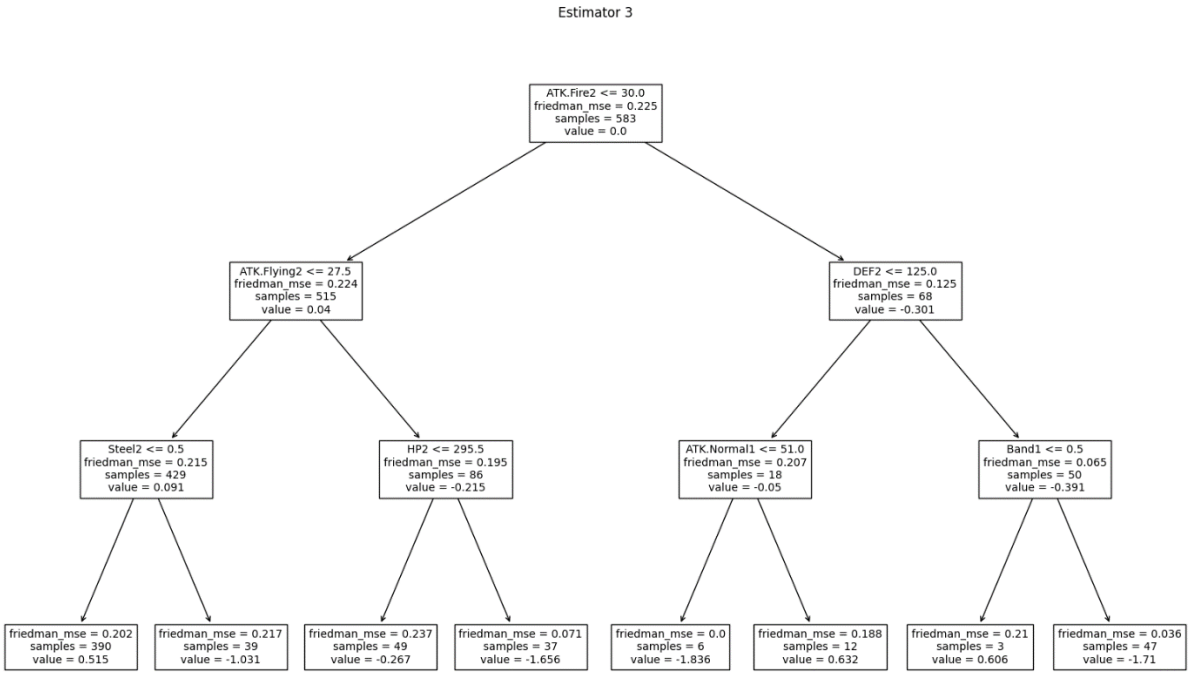
5.1.3 Resultados para el algoritmo de Gradient Tree Boosting

Como hemos dicho en la sección 4.6.2, el algoritmo de *Gradient Tree Boosting* está pensado para encontrar relaciones complejas, pero suele tender al *overfitting*. También habíamos dicho que el algoritmo de *Random Forest* está pensado para disminuir la variabilidad de los resultados y prevenir el *overfitting*. En la sección anterior hemos visto como los resultados del Random Forest no son buenos al aplicarlo a nuestro problema por lo que será natural preguntarnos si el algoritmo de *Gradient Tree Boosting* supondrá una mejor alternativa.

Empezaré presentando los tres primeros estimators creados en el modelo para tener un ejemplo de lo que hemos hablado en la sección 4.6.2.

Figura 37. *Primeros estimadores de nuestro modelo de Gradient Tree Boosting*





Fuente: Elaboración propia

En la figura 37 podemos ver los estimadores que se han creado. En cada nodo podemos ver el valor del nodo y el valor del parámetro *Friedman_mse*, junto con el criterio de la partición y el número de ejemplos. El valor de las hojas que se devolverá para el cálculo del resultado final tras multiplicarlo por el learning rate será el valor proporcionado por el parámetro *value*. Podemos ver que el valor de las hojas varía en cada uno de los árboles, incluso si el camino que se sigue es el mismo. Por ejemplo, en los tres árboles existe un camino que sigue las separaciones que dependen del ATK.Fire2, DEF2 y ATK.Normal1. Podemos ver que las hojas generadas por esta última partición tienen un valor diferente en cada árbol pese a tener los mismos ejemplos y valores de *Friedman_mse*. Esto se debe a que cada árbol se genera a partir de los errores del anterior y los valores se calculan de forma que nuestro resultado final se vaya acercando cada vez más al resultado esperado.

Analicemos el estudio de las precisiones, empezando por el resultado de la prueba del 10-fold cross-validation.

Figura 38. *Resultados de 10-fold cross-validation para medir la precisión del modelo de Gradient Tree Boosting*

```
Los resultados obtenidos en cada iteracion son:
[0.72881356 0.55932203 0.54237288 0.39655172 0.46551724 0.67241379
 0.63793103 0.65517241 0.68965517 0.79310345]
La media de las iteraciones es: 0.6140853302162479
```

Fuente: Elaboración propia

La figura 38 muestra que esta prueba aporta los mejores resultados que hemos obtenido hasta el momento, con una precisión promedio de 61.4%, una peor partición con el 39.7% de precisión y una mejor partición con un 79.3%.

Comprobemos ahora los resultados obtenidos utilizando el train-test split comparado con los datos ampliados y las Confusion Matrix para observar si los resultados obtenidos también son mejores que los que se obtienen utilizando los otros métodos.

Figura 39. *Resultados de la precisión utilizando los random_state 0, 1 y 2*

```
La precisión usando los datos originales es:  0.7397260273972602
La precisión usando los datos ampliados es:  0.7397260273972602
```

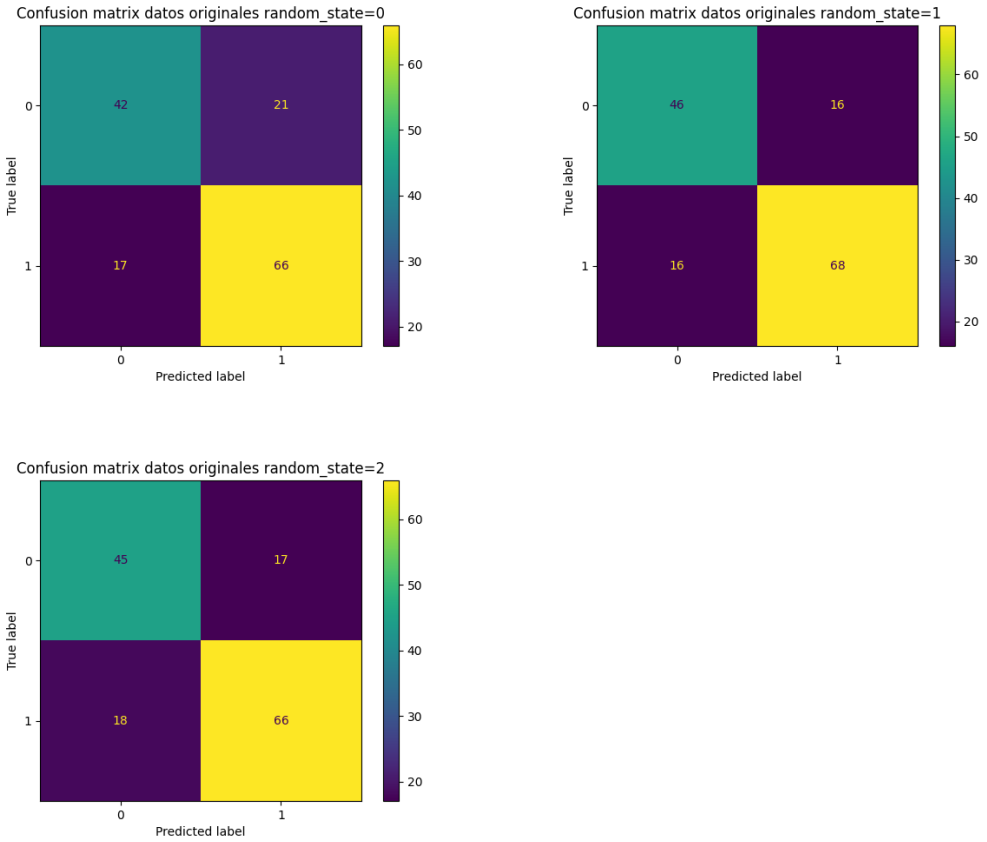
```
La precisión usando los datos originales es:  0.7808219178082192
La precisión usando los datos ampliados es:  0.7465753424657534
```

```
La precisión usando los datos originales es:  0.7602739726027398
La precisión usando los datos ampliados es:  0.7191780821917808
```

Fuente: Elaboración propia

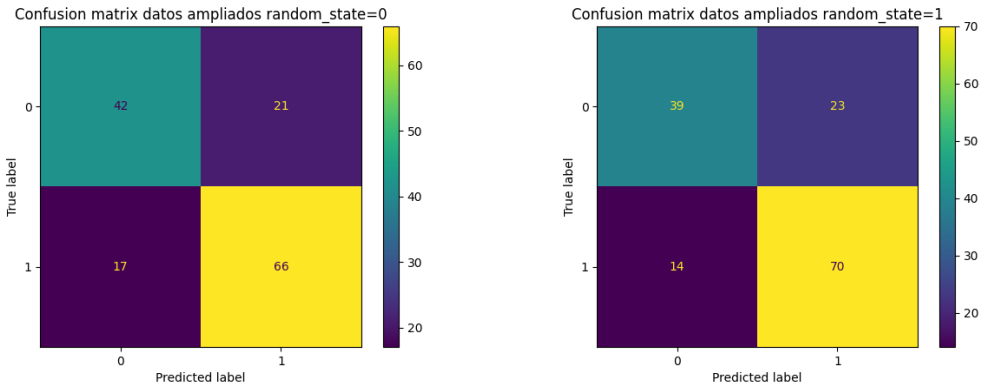
En la figura 39 podemos ver los resultados de la precisión alcanzada mediante el *train-test split*. Los resultados de ambos grupos son mejores que los que hemos obtenido usando los otros métodos. También vemos que entre los dos grupos de datos hay una diferencia similar a la que hemos obtenido en el estudio del método del *Random Forest*, volviendo a tener incluso que para la primera partición la precisión en los dos grupos de datos es la misma. Durante el estudio de los resultados del *Random Forest* hemos visto como la poca diferencia entre los dos grupos de datos puede deberse a que el propio algoritmo falla en identificar correctamente algunas de las relaciones, para saber más sobre lo que pasa en el modelo de Gradient Tree Boosting vamos a estudiar las Confusion Matrix.

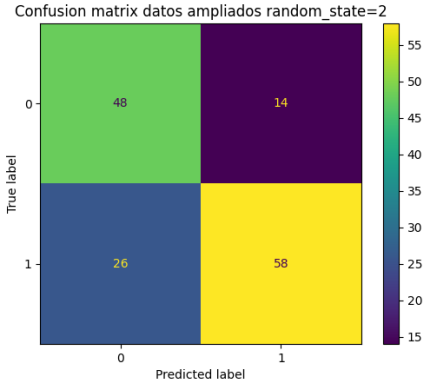
Figura 40a. *Confusion Matrix del modelo generado con los datos originales utilizando los random_state 0, 1 y 2*



Fuente: Elaboración propia

Figura 40b. *Confusion Matrix del modelo generado con los datos ampliados utilizando los random_state 0, 1 y 2*

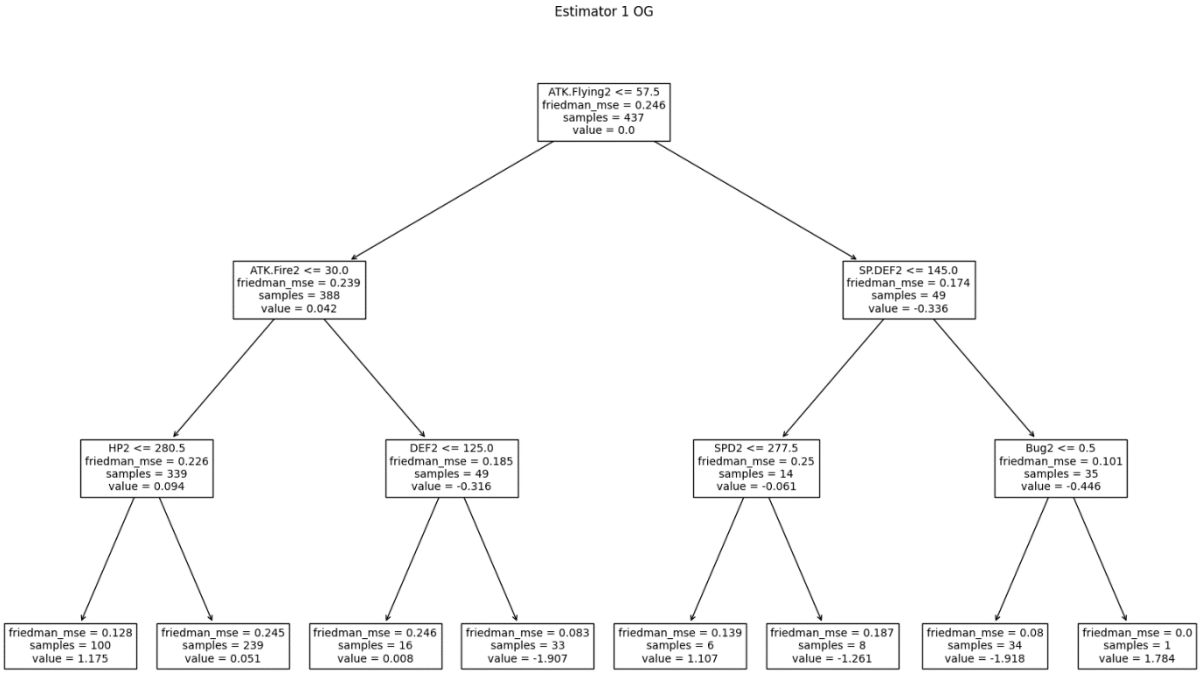




Fuente: Elaboración propia

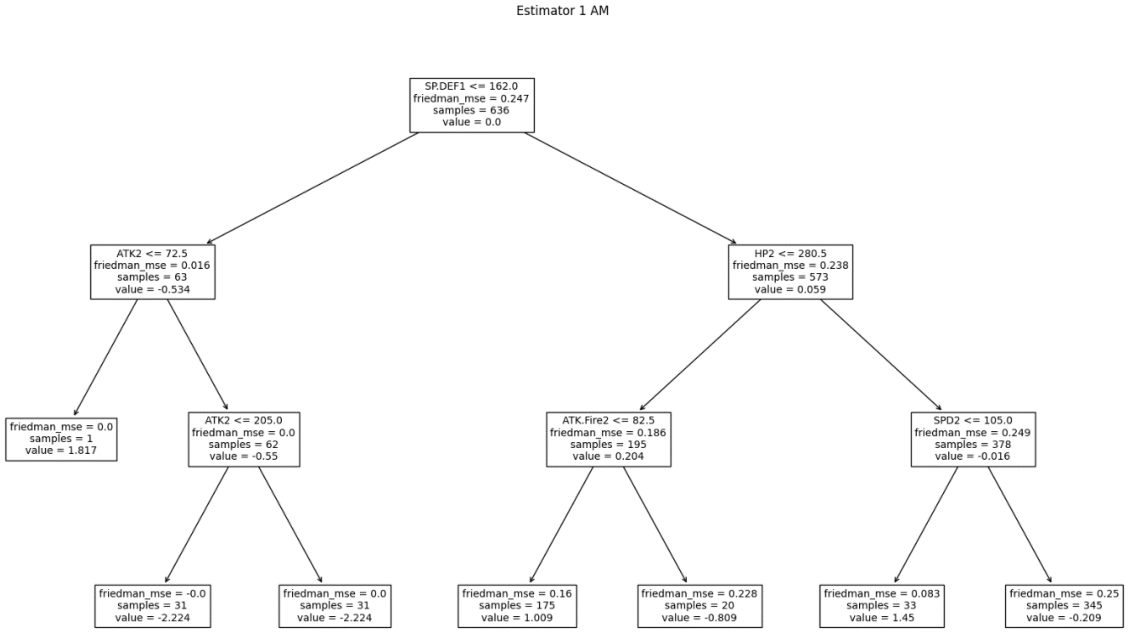
Si miramos los resultados de las Confusion Matrix para random_state=0 podemos ver que tenemos los mismos errores por False Positives y por False Negatives en las dos matrices, por lo que es posible que los dos modelos generados sean el mismo. Para terminar de ver si efectivamente se trata o no del mismo modelo vamos a estudiar el primer estimador generado por los dos modelos (Figuras 41a y 41b) y una comparación entre las propias predicciones (Figura 42). El primer estimador nos da una idea de cómo han afectado los nuevos datos a la importancia que tiene cada campo. Como cada árbol del *Gradient Tree Boosting* se crea para reducir los errores del árbol generado previamente, es posible que al final el modelo sea capaz de compensar por estas diferencias iniciales. Por lo tanto, imprimiremos los resultados de restar la lista con las segundas predicciones a la lista con las primeras. De esta forma tendremos que todo valor diferente de 0 será una predicción diferente en las dos listas.

Figura 41a. *Estimator 1 con los datos originales utilizando el train-test split con random_state=0*



Fuente: Elaboración propia

Figura 41b. *Estimator 1 con los datos ampliados utilizando el train-test split con random_state=0*



Fuente: Elaboración propia

Figura 42. *Resultado de las diferencias entre las precisones*

```
[ 0  0  0  0  0  0  1  0 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  1  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0
 -1  0  0  0  0  0  0 -1  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  1  0 -1  0  0  0
  0  0  0  1  0  1  0  0  0 -1  0  0  0  0  0  0  0  0  0 -1  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1
  0  0]
```

Fuente: Elaboración propia

Observando solamente la figura 42 ya podemos concluir que el modelo generado no es el mismo puesto que podemos observar diversas diferencias en las dos predicciones. Si observamos las figuras 41a y 41b también podemos ver que los árboles tienen separaciones diferentes al utilizar los dos grupos de datos. Lo más interesante es que, al comparar los estimadores de las figuras 41a y 41b con las primeras separaciones del modelo de Árbol de Decisión de las figuras 32a y 32b, podemos observar que son muy similares. Pese a las similitudes que podemos encontrar en estas figuras la diferencia de precisión entre los dos grupos de datos para el modelo de árbol de decisión es del 10%, mientras que la precisión del modelo de *Gradient Tree Boosting* es la misma. **Esto nos hace pensar que, pese a que estas primeras separaciones no parecen las mejores para nuestro problema, los árboles generados a continuación van ajustando los errores cometidos por estos primeros árboles de forma que el modelo de *Gradient Tree Boosting* termina siendo capaz de encontrar las relaciones importantes de una forma mucho más consistente que el algoritmo de *Árbol de Decisión*.**

5.1.4 Estudio de la incorporación de nuevos campos

Durante la creación de nuestro prototipo hemos hablado sobre como la incorporación de nuevos campos que consisten en una combinación de campos anteriores que estén relacionadas, como por ejemplo *Ofense1* o *Difofense* podría ser beneficiosa para el modelo de predicción. En esta sección he añadido algunos de estos campos a los datos para ver cómo afectan a los resultados de los modelos.

Empezamos con diversos campos nuevos que relacionan los stats de los dos pokémons:

- **Ofense1 y Ofense2:** Funcionarán de una manera similar a como los hemos definido en el prototipo, pero en vez de buscar el máximo entre la diferencia de los stats de ataque con las defensas correspondientes del rival, se mirará cuál de los dos stats de ataque es más alto para un pokémon. Una vez sepamos si el pokémon es un atacante especial o

físico se comparará el valor del ataque seleccionado con la defensa correspondiente del rival y este valor será nuestro valor de *Ofense*. Existen algunos sets de pokémon que pueden utilizar tanto ataques físicos como especiales, pero se trata de casos muy particulares que no trataré en este trabajo.

- **Difofense:** Corresponde exactamente a la definición dada en nuestro prototipo.
- **difHP:** Es la diferencia entre los puntos de vida del pokémon 1 y el pokémon 2. Siendo positiva si HP1>HP2 y negativa en caso contrario.
- **difSPD:** Es la diferencia entre las velocidades del pokémon 1 y el pokémon 2. difSPD es positiva cuando el pokemon 1 sea más rápido y negativa si es más lento.

También he creado otros dos campos que no tienen que ver con los *stats* de los pokémons, sino con las efectividades entre tipos. La figura 1 mostraba que las relaciones entre tipos pueden ser complejas y todavía más cuando un pokémon puede tener dos tipos diferentes. Como cada pokémon puede llevar 4 ataques diferentes lo más común es usar ataques de diferentes tipos para enfrentarse a diversas situaciones. Las efectividades pueden resultar muy difíciles de encontrar para nuestro programa ya que muchas combinaciones de tipos solamente las podemos encontrar en unas pocas líneas evolutivas. Si observamos los resultados obtenidos en la creación de los árboles de los diversos modelos podemos ver que las eficacias contra *Leavanny* son muy importantes ya que los campos ATK.Flying2 y ATK.Fire2 aparecen diversas veces entre las primeras separaciones. Los campos relacionados con las eficacias pueden tener los valores: super-efectivo, efectivo, neutro, resistido, super-resistido e inmune

El problema es que, de forma similar a lo que pasaba al añadir los combates extras a nuestros datos, al tener muchos pokémons con características diferentes como pokémon rival resulta difícil identificar qué tipo es eficaz contra cada uno. Para implementar estos campos se comprueba la efectividad de cada ataque del pokémon contra su rival y se almacena el valor máximo (ya que, normalmente, se utilizará el ataque con mayor efectividad). Estos campos podrán tener los valores; súper-efectivo, efectivo, neutro, resistido, súper-resistido e inmune. Podemos ver que existe un orden dentro de estos valores por lo que he utilizado un encoding ordinal que podemos ver en la tabla 2

Tabla 2. *Esquema del encoding de efectividades*

Valor	Encoding
Súpereficaz	2
Eficaz	1
Neutro	0
Resistido	-1
Súper-resistido	-2
Inmune	≤-3

Fuente: Elaboración propia

Para realizar este encoding he creado el programa *Efectividades.py* que compara cada ataque del pokémon principal con cada tipo del pokémon rival, creando un vector de longitud cuatro (tenemos como máximo cuatro ataques por pokémon) inicializado con ceros.

Para cada ataque analiza la efectividad de su tipo contra los tipos del rival siguiendo las relaciones de la figura 1. Si un ataque es eficaz contra un tipo sumaremos uno al valor de la efectividad del ataque, si es resistido restaremos uno y si es inmune restaremos 3. De esta manera tendremos que si, por ejemplo, un ataque es eficaz contra los dos tipos del rival el resultado será 2 (querrá decir que es súper-eficaz), y si es eficaz contra un tipo, pero el otro lo resiste, entonces el valor será 0 (neutro).

Vamos ahora a comprobar como estos campos afectan a la precisión de nuestros modelos usando el 10-fold cross-validation:

Figura 43. *Resultado de la prueba de 10-fold cross-validation para los modelos de Árbol de Decisión, Random Forest y Gradient Tree Boosting usando los nuevos campos.*

```
Los resultados obtenidos en cada iteracion son:
[0.61016949 0.59322034 0.54237288 0.72413793 0.53448276 0.77586207
 0.75862069 0.72413793 0.84482759 0.72413793]
La media de las iteraciones es: 0.6831969608416132
```

```
Los resultados obtenidos en cada iteracion son:
[0.71186441 0.57627119 0.49152542 0.48275862 0.55172414 0.74137931
 0.68965517 0.75862069 0.77586207 0.79310345]
La media de las iteraciones es: 0.6572764465225014
```

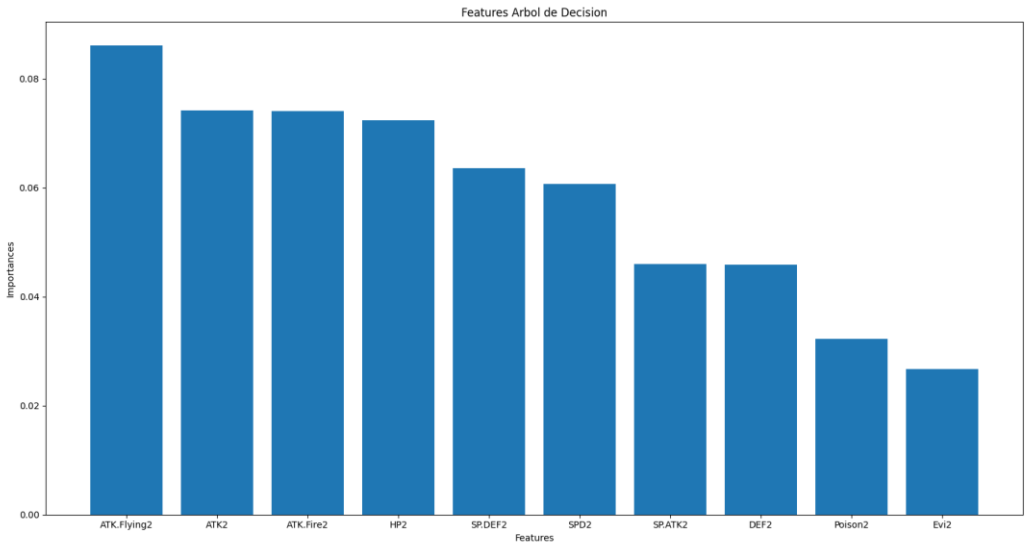
```
Los resultados obtenidos en cada iteracion son:
[0.6779661  0.6440678  0.72881356 0.65517241 0.75862069 0.86206897
 0.79310345 0.87931034 0.89655172 0.84482759]
La media de las iteraciones es: 0.7740502630040911
```

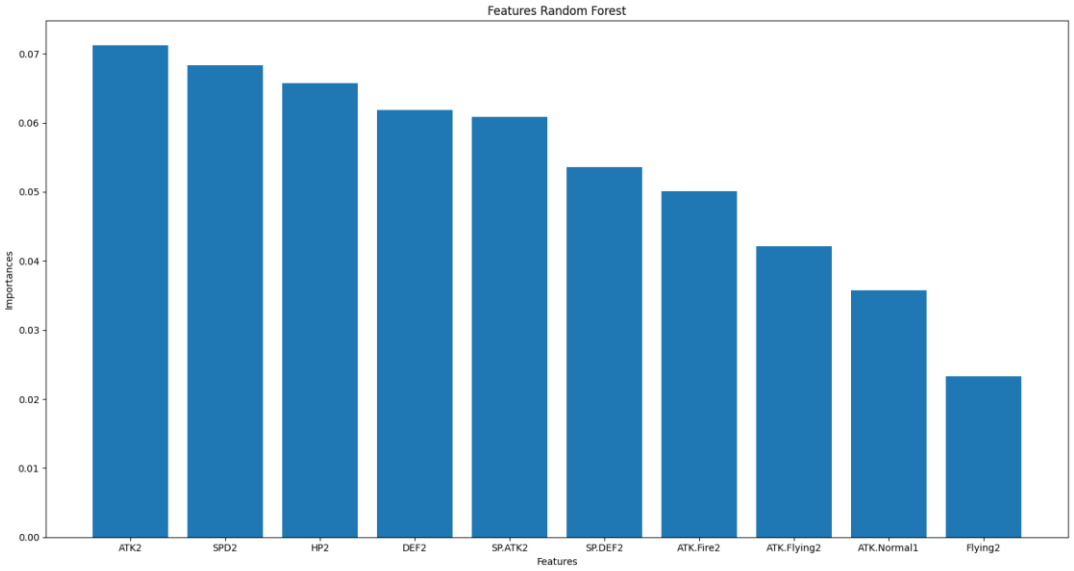
Fuente: Elaboración propia

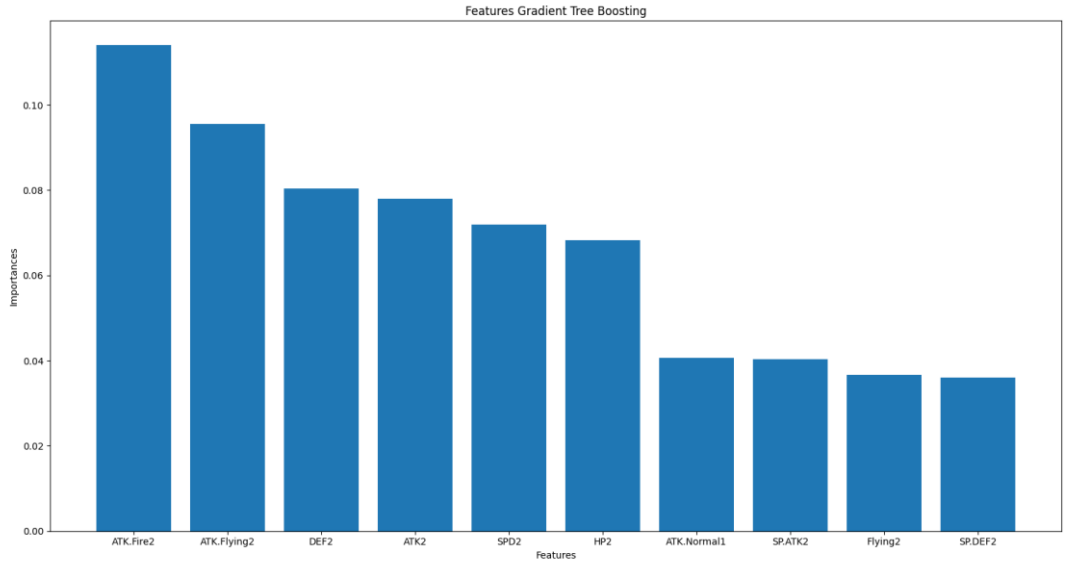
Podemos ver que los resultados mostrados en la figura 43 son bastante mejores que los resultados obtenidos con los datos originales. De hecho, todos los métodos han aumentado en un 10% o más la media de precisión. La mejor partición posee un valor de 89.6% utilizando el método de *Gradient Tree Boosting* y la peor ha mostrado un valor de 48.3% utilizando el método de *Random Forest*. Utilizando los datos normales no solamente habíamos obtenido particiones con valores del 20% de precisión, sino que solamente una partición había llegado al 80%. **Por lo tanto, podemos afirmar que estos nuevos campos mejoran de forma significativa la precisión de las predicciones.**

Para estudiar la importancia de los diferentes campos al añadir las nuevas variables utilizaremos el comando *features_importances*. Este comando calcula la relevancia de cada campo dependiendo de la importancia de los nodos que utilizan dicho campo para realizar la separación. La importancia de cada nodo vendrá determinada por la impureza y numero de ejemplos de los nodos generados por la partición y por la probabilidad de llegar a dicho nodo que se calcula como la razón entre el número de ejemplos que llegan al nodo y los ejemplos totales.

Figura 44. *Gráficos de las feature_importances limitado a los diez campos con mayor importancia para los modelos originales*

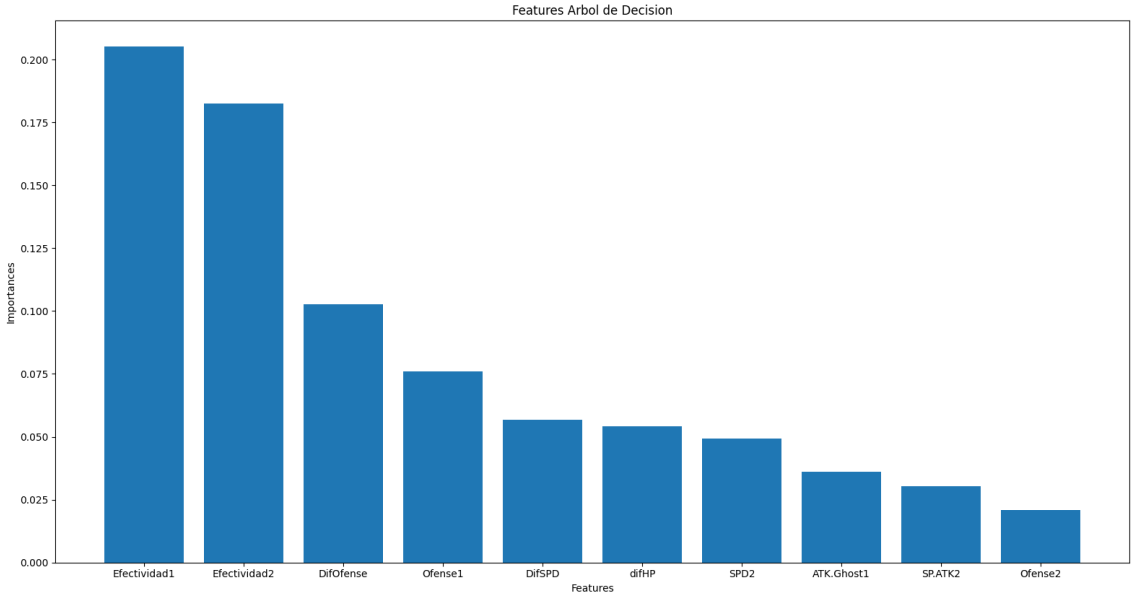


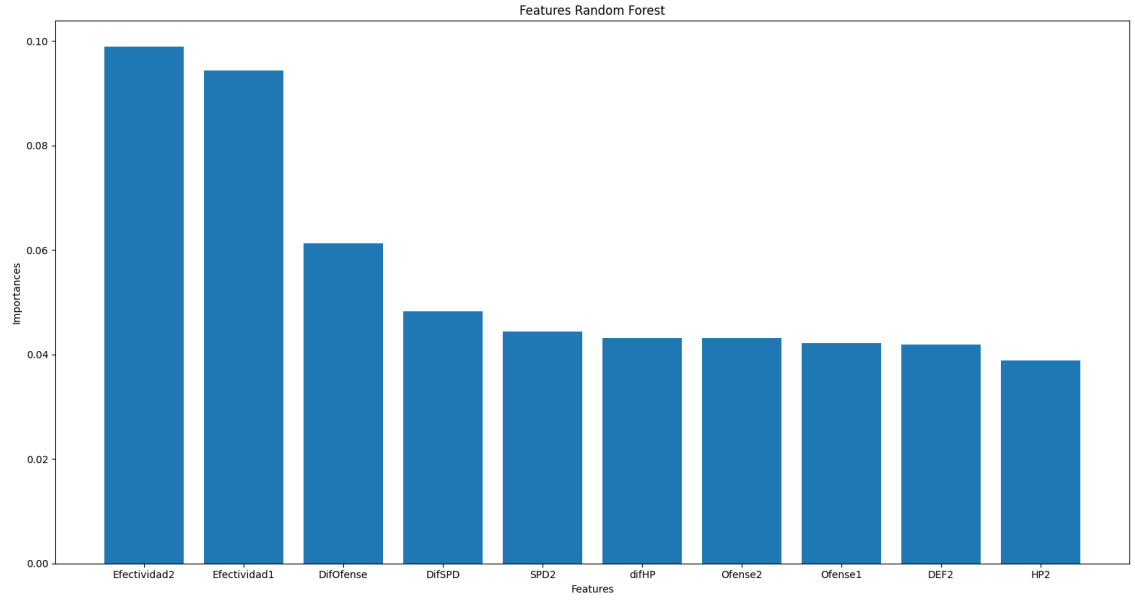


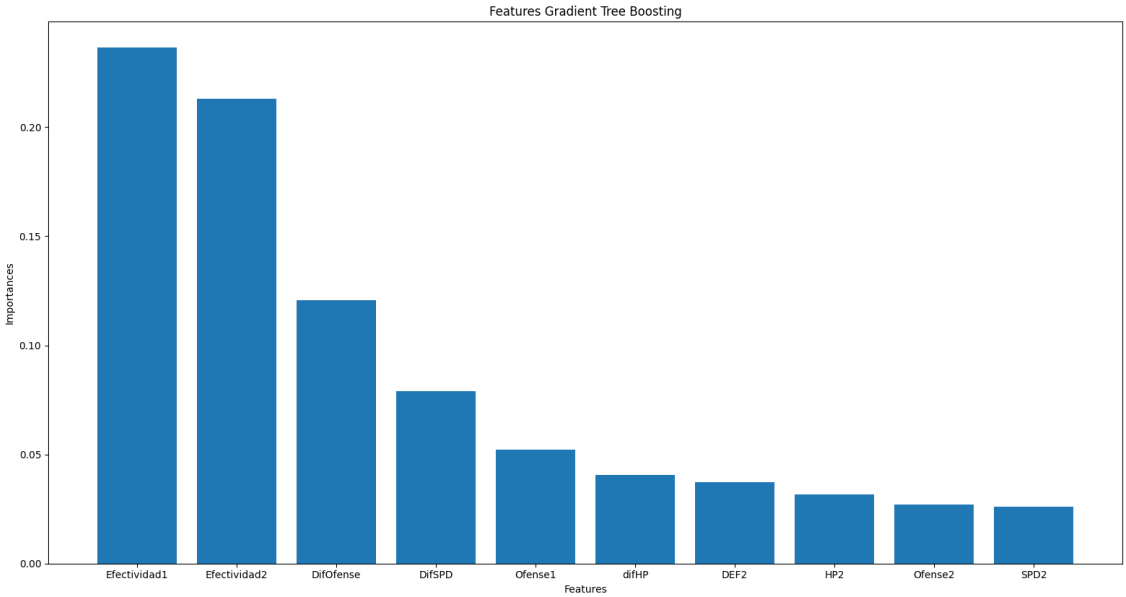


En la figura 44 podemos ver los campos con mayor importancia de los modelos creados inicialmente. Podemos ver que en los gráficos generados con los métodos de *Árbol de Decisión* y *Gradient Tree Boosting* las relaciones de eficacia contra *Leavanny*, como ATK.Flying2 y ATK.Fire2, tienen bastante importancia. En cambio, en el gráfico del método de *Random Forest* estas relaciones tienen mucha menos importancia, lo que refuerza las ideas que habíamos presentado durante la sección 5.1.2. Podemos comparar estos gráficos con los que se obtienen de los modelos creados con la inclusión de los nuevos campos.

Figura 45. *Gráficos de las feature_importances limitado a los diez campos con mayor importancia para los modelos con la introducción de los nuevos campos*







La figura 45, usando las nuevas features_importances, muestra que tanto las Efectividades como las diferencias en los *stats* se han convertido en los campos más importantes en los tres modelos, con las Efectividades siendo los campos con los valores más elevados. Algo a notar es que, pese a que esto se cumple para los tres modelos, en el modelo de *Random Forest* la importancia que estos campos tienen es significativamente menor que en los otros dos, lo que sigue reforzando nuestras ideas sobre los resultados de este algoritmo.

5.1.5 Conclusiones sobre los resultados

Después de las diversas pruebas que hemos realizado en este trabajo hemos visto que el mejor modelo para nuestro problema será el de Gradient Tree Boosting independientemente de cual de todas las versiones de datos estamos utilizando.

En cuanto a los propios datos para crear el modelo hemos visto diversos resultados interesantes.

- El primero es que definiendo nuevas relaciones establecidas entre los campos iniciales el modelo creado mejora la precisión. Por lo tanto, como continuación de este proyecto sería interesante encontrar formas de presentar algunas de las relaciones que se vayan necesitando de forma más clara o incluso buscar otro modelo diferente a los que hemos estudiado que sea capaz de ver mejor la importancia de las relaciones entre nuestras clases.

- El segundo sería que estudiando los combates de un solo pokémon como pokémon titular obtenemos mejores resultados que si estudiamos combates entre muchos pokémons diferentes. Este resultado da una idea de cómo se podrían abordar algunos de los problemas complejos que hemos tenido que dejar fuera del trabajo. Si tuviéramos un dataset suficientemente grande sería posible no solamente utilizar combates de un mismo pokémon si no que podríamos realizar una búsqueda para que los rivales tuvieran siempre alguna similitud con el pokémon rival del combate que queremos predecir, por ejemplo, el mismo tipo, la misma habilidad o incluso algún ataque igual. Si fuera posible realizar estas búsquedas podríamos encontrar muchos combates similares al que queremos predecir ya que pokémons que cumplen funciones similares suelen tener ataques, habilidades o objetos similares. Esto permitiría obtener unos datos para la creación del modelo en los que sería mucho más sencillo encontrar las relaciones importantes que pueden aparecer en escenarios específicos y nos permitiría trabajar con la gran variedad de objetos y habilidades que existen.

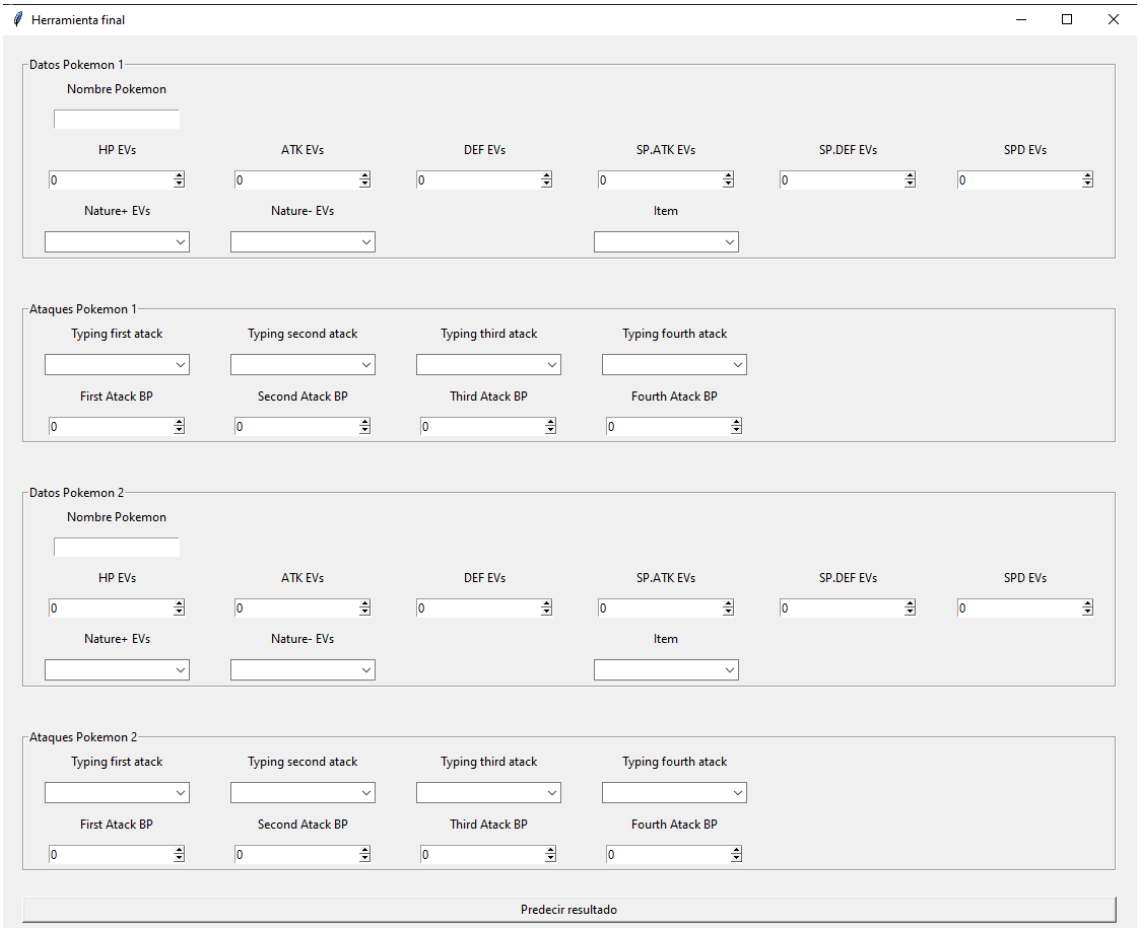
Durante la creación del prototipo he hablado de que los datos del combate se almacenaban en dos ficheros diferentes, cada uno perteneciente a un pokémon. Esta es una forma de mantener los datos que vamos a usar de forma ordenada que haría que la búsqueda de combates, donde el rival comparte alguna característica con el del combate que queremos predecir, fuera más rápida. El problema con este método es que puede tener un coste de almacenamiento muy alto ya que cada combate se almacenara dos veces. De momento existen 1010 pokémons por lo que, en un principio, no parece un aspecto que suponga una limitación importante, pero podría llegar a serlo si va aumentando el número de datos considerablemente. Una opción para la solución de este problema sería eliminar los combates más antiguos a medida que se van añadiendo combates nuevos para cada pokémon. También existe la opción de almacenar todos los datos en un mismo dataset y después buscar los combates en los que participe el pokémon titular y, entre esos, realizar otra búsqueda para los rivales. Un solo dataset disminuiría los costes de almacenamiento a la mitad, pero el proceso de búsqueda de los datos puede afectar al tiempo de ejecución del programa e introducir nuevas complicaciones.

5.2. Resultado final

Una vez que ya hemos decidido el método que vamos a utilizar para la creación del modelo de nuestra aplicación ya podemos empezar la construcción de la herramienta final.

Lo primero que vamos a necesitar es una interfaz gráfica que permita al usuario entrar los datos del combate que quiere predecir. En nuestro caso nos basaremos en el programa que habíamos creado para almacenar los combates dentro de nuestro dataset. En la figura 46 podemos ver la interfaz gráfica de nuestra herramienta final.

Figura 46. *Interfaz gráfica de nuestra herramienta.*



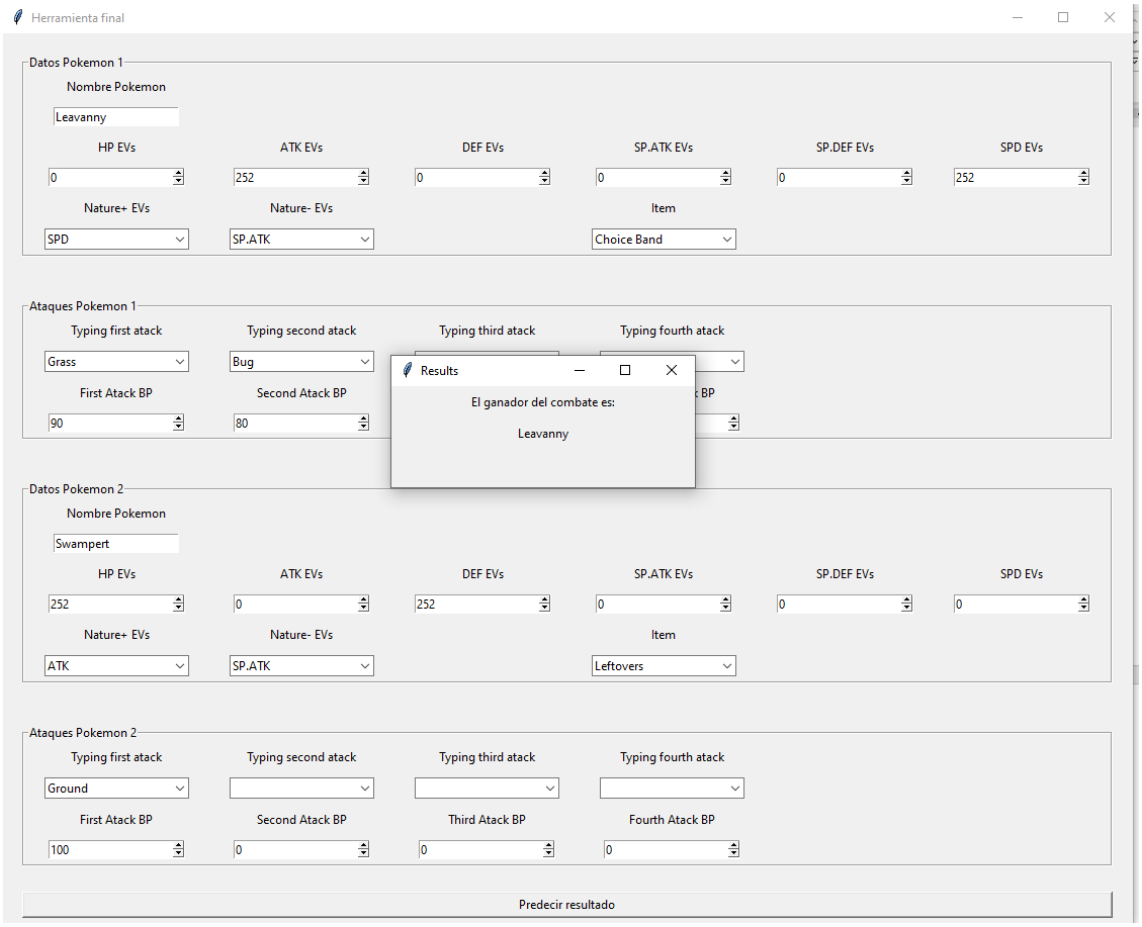
Fuente: Elaboración propia

El programa que correrá al pulsar el botón *Predecir resultado* será una combinación de los diversos programas que hemos realizado hasta ahora:

- El primer paso comprueba que todos los datos introducidos por el usuario son correctos. Si existe alguna incongruencia, como que hemos superado el límite de EVs o que el nombre del pokémon no está en la base de datos, se lanzará un mensaje de error.
- Lo siguiente es almacenar los datos de los pokémons en un vector de la misma manera en la que lo hemos hecho para almacenar los datos en el dataset.

- Ahora, en vez de añadir este nuevo vector en el dataset, le añadimos los nombres de las columnas para convertirlo en un dataframe que nuestro modelo pueda predecir.
- Se crea el modelo utilizando el algoritmo elegido, en nuestro caso *Gradient Tree Boosting* ya que es el que nos ha aportado mejores resultados, y se predice el resultado.
- Finalmente, un mensaje por pantalla muestra quién es el vencedor del combate.

Figura 47. *Herramienta mostrando por pantalla el resultado final*



Fuente: Elaboración propia

6. Conclusiones y trabajo futuro

6.1 Conclusiones

Conclusiones generales:

El objetivo de este trabajo era la creación de una aplicación capaz de predecir el resultado de un combate entre dos pokémons mediante un algoritmo de machine learning. La intención era que

dicha aplicación pudiese resultar útil a los jugadores competitivos en la preparación de equipos de torneo, aportando resultados de manera rápida y simple sobre que combates podría ganar o perder un Pokémon criado de una manera específica. Pese a que en este trabajo hemos tenido que trabajar con un problema más simplificado al que nos encontraríamos en un caso real, los resultados obtenidos son lo suficientemente buenos como para poder afirmar que una aplicación de este tipo es totalmente factible. De hecho, pese a no ser capaz de poder tener en cuenta todos los parámetros que serían necesarios para una herramienta ya lista para el uso público, nuestra aplicación sería capaz de predecir los resultados de la mayoría de los combates siempre y cuando dispongamos de los datos suficientes. Debido a que hemos realizado el estudio disponiendo solamente de unos datasets que hemos tenido que crear nosotros mismos, muchos de los resultados que hemos obtenido podrían variar si se tuviese un mayor número de ejemplos.

Conclusiones especificas:

- He abordado en este proyecto el estudio de una aplicación de ML para predecir el resultado de combates pokémon
- Para el trabajo me he centrado en el uso de bibliotecas de Python para la creación de diversos modelos basados en Árboles de Decisión
- Incluso limitándonos a combates individuales, hemos descrito la gran cantidad de variables que condicionan el resultado final
- Para tratar de entender e identificar los rasgos más importantes del problema he empezado tratando un problema más simplificado, en el cual he elaborado un árbol manualmente. De este ejercicio se deduce que utilizar directamente las propiedades básicas puede resultar en un problema débilmente definido. Teniendo en cuenta relaciones entre dichas propiedades podemos obtener resultados mucho más prometedores.
- He implementado posteriormente un ejemplo completo de elaboración de datasets, generación de modelos, entrenamiento y test de un caso basándome en los algoritmos de Árbol de Decisión, Random Forest y Gradient Tree Boosting. En este ejemplo hemos podido estudiar el funcionamiento y rendimiento de los modelos sobre nuestro problema.
- Como aspecto muy importante he observado que crear un modelo es un proceso sencillo. Pero encontrar un modelo apropiado para el tratamiento de un problema específico puede resultar notablemente más difícil, por lo que es importante tener una buena comprensión del problema con el que estamos trabajando.

- He elaborado personalmente **todo el material** de este proyecto por lo que el tamaño de los datasets es un poco limitado y es posible que haya cometido errores.
- Resultados muestran que diversos modelos aportan diversos resultados que pueden variar dependiendo del problema que estemos tratando.
- Herramientas útiles: Confusion Matrix, Gini, train-test split, K-fold cross-validation, feature importances

6.1 Trabajo futuro

Como hemos ido mostrando a lo largo de este trabajo, los combates pokémons son un problema muy complejo con un gran número de interacciones y escenarios diferentes. Para este trabajo hemos tenido que trabajar con un problema simplificado debido, en parte, a la limitación que suponía la cantidad de datos que tenía nuestra muestra. Para poder analizar correctamente el resto de los casos que nosotros no hemos podido abordar lo primero que haría falta sería la ampliación de nuestro dataset con número mucho mayor de combates, lo óptimo sería disponer de por lo menos un combate de cada pokémon contra todos los demás de forma que dispongamos de información sobre el mayor número de escenarios posibles para aplicar las ideas mencionadas en el apartado de “Resultados”.

También sería interesante estudiar la compatibilidad de esta herramienta con las herramientas ya existentes que se utilizan para la preparación de equipos competitivos. Un ejemplo sería la implementación de la posibilidad de importar lo que en la comunidad de pokémon llamamos un *pokepaste*, que es un fichero -txt en el que tenemos especificados todos los datos de un pokémon: raza, EVs, objeto, ataques, etc. De esta manera, una vez creado el *pokepaste,* solo hace falta importarlo en cada herramienta en vez de tener que introducir los datos cada vez, esto ahorraría tiempo y minimizaría las posibilidades de que el usuario se equivoque al entrar los datos.

Otra función que resultaría muy útil sería poder guardar un ejemplo a partir de la grabación de una partida del simulador. De esta manera a medida que los jugadores van realizando las pruebas para las preparaciones de sus equipos la base de datos irá aumentando y además se mantendría siempre actualizada con los combates más recientes.

En este trabajo hemos hablado de las diferentes *tiers* de Smogon y de las limitaciones que presentan en diferentes aspectos (por ejemplo, sobre los pokémons que son utilizables). Esto claramente genera que los pokémons criados para los combates dentro de una *tier* en particular

puedan ser diferentes a como son criados otros pokémons de la misma raza para competir en otras *tiers* diferentes. Por lo tanto, implementar una aplicación que tenga en cuenta los datos de los combates legales dentro de una *tier* dada sería de gran ayuda para muchos jugadores.

Finalmente, esta aplicación está pensada para ser utilizada por jugadores de todo el mundo por lo que si se pusiera a disposición del público lo mejor sería crearla como herramienta web para lo que tendríamos que crear un dominio y crear servidores para nuestros datos.

Referencias bibliográficas

[1] Nintendo 1996-2018 Pokemon Series, Game Freak. Pokemon Series, 1996-2023.

[2] Smogon, https://www.smogon.com/ , 2023

[3] Pokémon Showdown, https://pokemonshowdown.com/, 2023

[4] Pokémon Damage Calculator, https://calc.pokemonshowdown.com/, 2023

[5] Wikidex, https://www.wikidex.net/wiki/WikiDex, 2023

[6] Kaggle, https://www.kaggle.com/, 2023

[7] Kush Khosla, Lucas Lin, and Calvin Qi. Artificial Intelligence for Pokémon Showdown. PhD thesis, Stanford University, 2017.

[8] Pokémon Battle Predictor, https://www.pokemonbattlepredictor.com/,2023

[9] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. Nature, 550(7676):354–359, 2017.

[10] McKinney, W., & others. (2010). Data structures for statistical computing in python. In Proceedings of the 9th Python in Science Conference (Vol. 445, pp. 51–56).

[11] Raileanu, L.E., Stoffel, K. Theoretical Comparison between the Gini Index and Information Gain Criteria. Annals of Mathematics and Artificial Intelligence 41, 77–93 (2004)

[12] E.R. Gansner and S.C. North. An open graph visualization system and its applications to software engineering. Software— Practice and Experience, 30:1203–1233, 2000.

[13] Pokedex Coverage, https://www.pkmn.help/offense/, 2023

[14] Lundh, F. An introduction to tkinter. https://www.Pythonware.com/Library/Tkinter/ , (1999)

[15] J. H. Friedman, Stochastic Gradient Boosting, Technical Report, Stanford University, Stanford, 1999

Glosario

- **Raza de un Pokémon**: Todos los pokémons de una misma raza comparten stats de base, tipo, habilidad y que ataques pueden aprender. Ninguna de estas características se puede cambiar de manera directa dentro de un combate.

- **Tipo**:Cada raza de pokémon tiene un mínimo de un tipo y un máximo de dos que serán los mismos para todos los individuos de una misma raza. No solo los pokémons tienen un tipo asociado, sino que cada ataque también tiene un tipo al que pertenece. En la figura 1 podemos ver como interactúan los diferentes tipos entre sí. Si en la casilla pone un x2 quiere decir que si atacas con un ataque del tipo de la fila a un pokémon del tipo de la columna el daño del ataque será el doble del que sería sin tener en cuenta los tipos. Si esto sucede diremos que el ataque es supereficaz. Si en la casilla pone x1/2 quiere decir que el ataque hará la mitad del daño que haría sin tener en cuenta los tipos. Si esto sucede diremos que el ataque es resistido. Si en la casilla pone un 0 quiere decir que este ataque no afectara de ninguna manera al pokémon rival. Si esto sucede diremos que el pokémon es inmune. En los demás casos se dirá que el ataque es nuestro y se calculara el daño sin tener en cuenta los tipos. Si un pokémon tiene dos tipos las debilidades y las resistencias se multiplicarán al calcular los daños. Por ejemplo, si atacamos a un pokémon de hielo planta con un ataque de fuego como el fuego es super eficaz contra el hielo y la planta el ataque causará cuatro veces el daño que se causaría sin tener en cuenta los tipos. Esto también sirve para neutralizar debilidades, si atacamos ahora a un pokémon de Hielo-Agua
  con un ataque de fuego vemos que al hielo seria x2 y al agua seria x1/2 como 2x1/2=1 el ataque termina siendo neutro. Finalmente mencionamos que si un pokémon utiliza un ataque del mismo tipo que el suyo el daño que realiza este ataque será multiplicado por 1.5. A esto se le llama STAB(Same Type Atack Bonus) y es la principal razón por la que mucho pokémons suelen utilizar más a menudo ataques de su mismo tipo que de otros.

- **Habilidad**: Característica pasiva que tiene un pokémon. Cada raza de pokémon tiene un numero de diferentes habilidades posibles y cada individuo de la raza solo podrá tener una de ellas a la vez. Las habilidades otorgan un efecto pasivo al pokémon en combate que puede ir desde hacerlo inmune a ataques a un cierto tipo hasta impedirle actuar en un turno especifico.

- **Ataque**: Movimientos que puede utilizar un pokémon en combate. Se pueden llevar un máximo de 4 en cada pokémon. Cada ataque tiene cuatro características diferentes, el tipo del ataque, su potencia, su especialidad y su efecto secundario. El tipo del ataque ya lo hemos explicado en la sección de Tipo. La potencia del ataque determina la cantidad de daño que puede causar al rival. La especialidad determinara si un ataque es físico, especial o de estado. Los ataques de estado producen diferentes efectos en el combate como sanar al usuario o envenenar al rival, pero nunca realizan daño directo por lo que su potencia es siempre 0. Los ataques físicos utilizan el stat de ATK del usuario y del DEF del rival para el cálculo del daño y los ataques especiales funcionaran de manera análoga a los físicos, pero utilizando el SP.ATK y SP.DEF. El daño que causa un ataque sigue la formula:

$$Daño = 0.01 \times STAB \times Efectividad \times R \times \left(\frac{(0.2 \times Nivel + 1) \times ATK \times Poder}{25 \times DEF} + 2\right)$$

  Donde Efectividad toma los valores de la tabla de la figura 1, STAB toma el valor 1.5 si el ataque es del mismo tipo que el Pokémon y ATK y DEF utilizaran el ataque y defensa físico o especial dependiendo de la especialidad del ataque. R será un valor aleatorio entre 85 y 100.

  En cuanto a los efectos secundarios muchos de ellos solo tienen una pequeña probabilidad de suceder, pero otros suceden de manera garantizada. En este trabajo solo tendremos en cuenta los que suceden de manera garantizada y se hablara de esto más en la sección de desarrollo del problema principal.

- **Stats**: Valores que determinan como de fuerte es un pokémon en diferentes campos. Todos los pokémons tienen los mismos campos con variaciones en los valores de cada uno

    - HP: Cantidad de puntos de vida de un Pokémon. Si un pokémon recibiese 500 puntos de daño por un ataque esto será 50% de su vida si tuviera 1000 HP pero solo un 25% si tuviera 2000HP

    - ATK: Valor del ataque físico del pokémon. Este valor determina el daño que se realiza al utilizar un ataque físico sobre otro pokémon.

    - DEF: Valor de la defensa física del pokémon. Este valor determina la cantidad de daño mitigado al recibir un ataque físico de otro pokémon.

- SP.ATK: Valor del ataque especial del pokémon. Este valor determina el daño que se realiza al utilizar un ataque especial sobre otro pokémon.

- SP.DEF: Valor de la defensa física del pokémon. Este valor determina la cantidad de daño mitigado al recibir un ataque físico de otro pokémon.

- SPD: Valor de la velocidad de un Pokémon. Pokémon es un juego que se basa en turnos simultáneos, es decir los dos jugadores eligen su jugada para este turno y a continuación el turno se juega automáticamente. La velocidad es importante porque el Pokémon más rápido será el que realizara la acción que se le ha asignado antes.

Los stats de un Pokémon se calculan siguiendo la siguiente fórmula para los HP:

$$HP = \left[\frac{\left(HP_{Base} \times 2 + IV + \left[\frac{EV}{4}\right]\right) \times Nivel}{100}\right] + Nivel + 10$$

Y la siguiente fórmula para el resto de los stats:

$$Stat = \left(\left[\frac{\left(Stat_{Base} \times 2 + IV + \left[\frac{EV}{4}\right]\right) \times Nivel}{100}\right] + 5\right) \times Naturaleza$$

A continuación, explicaremos que son los stats base, EV, IV, nivel y Naturaleza que aparecen en nuestras formulas.

- **Stats Base**: Valores determinados por la raza del Pokémon que determina la capacidad de crecimiento de sus stats en los diversos campos.

- **Nivel**: Dentro del modo historia de Pokémon esta es la principal manera de mejorar tus Pokémon y se va incrementando desde 1 hasta 100. Cuanto mayor es el nivel de un pokémon mejores son sus stats, esto hace que la experiencia casual de jugar el modo historia de Pokémon sea accesible para toda clase de jugadores ya que si se dispone de pokémons de mayor nivel que el rival se tendrá una ventaja notable en el combate. En combate competitivo contra otro jugador real y no contra un personaje controlado por el juego como sucede en el modo historia, el nivel de los pokémons se iguala para que no exista esta ventaja y se juegue en igualdad de condiciones. En nuestro caso, en

combates competitivos individuales todos los pokémons estarán a nivel máximo, es decir, 100.

- **EV**: Cada pokémon dispone de un total de 508 EV que se pueden distribuir entre todos los campos de los Stats. El máximo que se puede invertir en un campo es 252 y en el caso de estar a nivel 100 es fácil deducir de las fórmulas que por cada 4 EV invertidos en un stat este aumentara en una unidad. Los Stats se calculan en números enteros por lo que si se invierten 14 EV se obtendrá el mismo resultado que invirtiendo 12 EV, es decir, una mejora de tres unidades. Esta es la principal forma de variar los stats de un pokémon y funcionan de la misma manera para todos los pokémons.

- **IV**: Valores que determinan el potencial de cada campo de Stats de un Pokémon. Puede variar entre 0 y 31 y cuanto más grande sea mayor será el stat en cuestión. Los IV sirven principalmente en el modo historia para diferenciar los diferentes individuos de una raza ya que en muchos juegos no existe ninguna forma de cambiarlos. En competitivo, en cambio se utilizarán los mejores Pokémon que se pueda por lo que todos los pokémons tienen 31 IV en todos los stats exceptuando estrategias muy específicas que no trataremos en este trabajo.

- **Naturaleza**: Existen 25 clases de naturaleza diferente que cada pokémon puede tener independientemente de su raza. Si miramos las fórmulas para el cálculo de los stats que hemos presentado antes vemos que en el cálculo de HP no tenemos esta variable. Esto se debe a que este stat es el único que no puede verse afectado por ninguna naturaleza. Cada naturaleza establece uno de los stats como favorable, otro como desfavorable y los demás como neutro. En nuestra fórmula para el cálculo de los stats al calcular el valor del stat con naturaleza favorable Naturaleza=1.1, si es neutro será Naturaleza=1 y si es desfavorable será Naturaleza=0.9. Algunas de las naturalezas no tienen ningún están favorable ni ninguno desfavorable, estas naturalezas se denominan neutra y como no tienen uso en competitivo no las tendremos en cuenta en nuestro problema.

- **Objetos**: Cada pokémon puede llevar un objeto al combate. Todos los pokémons pueden llevar equipado cualquier objeto, aunque una parte muy pequeña de ellos solo tienen efecto si están equipados a una raza de pokémon especifica. Podemos separar los

objetos entre objetos defensivos y defensivos. Los ataques ofensivos aumentan las capacidades ofensivas del pokémon a cambio de alguna restricción o solo se activan en ciertas situaciones y los objetos defensivos previenen daño en algunas situaciones o regeneran una pequeña cantidad de vida al final de cada turno, por ejemplo.

- **Circuito VGC**: circuito oficial organizado por Pokémon Company en diversos lugares del mundo. Modalidad de combates dobles.

- **Smogon**: Comunidad de jugadores no oficial en la que se juegan combates de modalidad individual. Dentro de esta comunidad se han separado los diferentes pokémons en tiers y se juegan ligas, torneos y Ladder para cada una de estas tiers. Esta comunidad se desarrolla exclusivamente utilizando el simulador online Pokémon Showdown.

- **Tiers:** Clasificación de los pokémons dependiendo de su uso. Todo pokémon con un uso en Ladder mayor de un cierto umbral pertenecerá a la tier. Todos los pokémons que no tengan el uso requerido para pertenecer a una tier bajaran a la tier siguiente donde se ira repitiendo el proceso. En cada tier se pueden utilizar lo pokémons que pertenecen a esta tier y a las que están por debajo pero nunca a los que pertenecen a tiers superiores. Este sistema solo sirve para clasificar los pokémons pero no a los jugadores. La tier que juegue un jugador no es relativa a su calidad ya que son ligas y torneo diferentes. Muchos jugadores participan en circuitos competitivos de diferentes tiers a la vez.

- **Ladder**: Modalidad de juego donde buscas un combate con un adversario cualquiera. Se rige por un sistema de ELO que es un valor que determina como de bueno es un jugador. Este ELO es diferente en cada tier para cada jugador comenzando siempre desde 1000. Si se gana una partida este ELO aumenta mientras que si se pierde disminuye. Se le llama Ladder porque existe una clasificación de ELO por lo que los mejores jugadores intentan mantenerse entre los primeros rankings de cada tier.

- **Pokémon Showdown**: Simulador online en el que se pueden simular combates pokémons sin tener que criar tus pokémons en el juego original. Sirve principalmente para jugar a las diversas tiers y circuitos de la comunidad de Smogon pero también es utilizado por los jugadores del circuito VGC para practicar ya que es mucho más sencillo criar un pokémon con un par de clicks del ratón que pasarte horas en el juego original,

esto permite hacer variaciones más fácilmente y simplifica las pruebas necesarias para la creación de un equipo de torneo.