# **Capítulo 7. Aprendizaje en Conjunto y Random Forest**

* **Introducción**

Supongamos que publicas una pregunta dirigida a miles de personas, y luego combinas las respuestas para tener una en definitiva. En muchos casos una respuesta masiva es mejor que la respuesta de un solo experto. A este método se le conoce como la sabiduría de la multitud o de las masas.

De igual manera, si juntas las predicciones de un montón de predictores, como clasificadores o regresiones, lo esperado sería obtener una mejor respuesta que un predictor individual. Esta técnica se utiliza y se conoce como aprendizaje en conjunto.

Esta técnica se puede realizar mediante un algoritmo llamado método de conjunción. Dentro de los ejemplos que se pueden poner en práctica con este método, se encuentra el entrenamiento de un grupo de árboles de decisión en donde obtienes un modelo por cada árbol que predice la clase. Una vez que todos los árboles emiten su respuesta, se selecciona la respuesta que sea más frecuente dentro de las respuestas.

A todo este método se le conoce como Random Forest o Bosque aleatorio. Y si lo piensan tiene mucho sentido porque ¿cómo le llamas a un lugar que tiene muchos árboles? Bosque.

Cabe mencionar que debido a sus características los bosques aleatorios son uno de los algoritmos más poderosos y utilizados en ML.

Como vimos en el capítulo 2, normalmente se utilizan los métodos de conjunción al final del proyecto, esto una vez que se tengan algunos predictores para combinarlos en un predictor que sea todavía mejor.

De hecho, este método de conjunción es tan efectivo que ha ganado diferentes concursos de Machine Learning siendo uno de los más importantes la llamada Netflix prize competition.

En este capítulo hablaremos sobre los 3 métodos de conjunción más populares:

* Empacar, mejor conocido en ML como BAGGING.
* Impulsar, conocido como BOOSTING.
* Apilar, mejor conocido como STACKING.

Así mismo nos adentraremos en el funcionamiento de los Random Forest.

## **Elección de Clasificadores**

Supongamos que estás trabajando en un proyecto de clasificación y ya has entrenado algunos modelos, todos ellos con un promedio de asertividad del 80%.

Por ejemplo, digamos que tienes un modelo hecho con regresión logística, otro con SVM, otro con un árbol de decisión.

Diagram

Description automatically generated

Pues una forma de crear un clasificador que funcione todavía mejor sería comparar las clasificaciones de cada uno de estos y elegir la que más se repita.

A esto se le conoce como clasificación por votación estricta o **hard voting**, ya que podríamos ver cada predicción como un voto para saber cuál será la elegida.

Diagram

Description automatically generated

Como podemos ver en la imagen, dos de los tres predictores votaron 1, mientras que uno de ellos votó por 0, por lo que el elegido sería la clasificación 1.

Sorprendentemente, en general este tipo de votación logra obtener un mejor porcentaje de asertividad que cualquiera de los clasificadores por separado.

Inclusive, si todos estos clasificadores fueran considerados débiles, el conjunto de ellos podría dar lugar a un clasificador de aprendizaje fuerte (clasificador con un gran porcentaje de asertividad).

Y te preguntarás ¿cómo es esto posible? Crear un buen clasificador a partir de un conjunto de clasificadores que no sirven para nada.

Pues esta analogía puede ayudarte un poco a entender cómo funciona.

Imaginemos que estamos jugando en el juego de casino llamado ruleta. Quizás durante ese juego escuches la famosa frase de “el casino siempre gana” y en el caso de la ruleta, el número 0 es el que inclina ese lado de la balanza.

Una de las apuestas más frecuentes son las binarias, es decir, apostar si será negro o rojo, apostar si es par o impar y son famosas porque las personas piensan que es una apuesta 50 50, pero ¿realmente tienes la mitad de las probabilidades de ganar?

Pues la respuesta es no, hay un número en la ruleta que es completamente imparcial a estas apuestas y se le conoce como 0. Aunque este número tiene una probabilidad de ganar de 2.70%, cuando se juega miles de veces como se hace normalmente en un casino, estamos hablando de que por cada 10000 tiradas el casino gana aproximadamente 550 veces arbitrariamente.

Vamos a ver gráficamente como este 2.70% afecta conforme se juega, para esto vamos a simular las partidas de 50 jugadores que comenzaron con 10,000 dólares. Cabe resaltar que también está presente el casino. Aunque hay algunos jugadores que ya perdieron dinero, hay otros que también ganaron y en general todos están distribuidos dentro de un mismo rango.

A picture containing chart

Description automatically generated

Ahora vamos a probar con 1000 simulaciones:

Chart

Description automatically generated

A simple vista se siguen viendo distribuidos, pero en realidad se comienza a ver una tendencia en decaimiento. Es decir, la mayoría de los jugadores empiezan a perder.

Por último, vamos a ver qué pasa con 10,000 simulaciones:

Chart

Description automatically generated with medium confidence

Aquí ya se puede ver claramente que la tendencia a perder es mucho mayor a la tendencia a ganar cuando se juega miles de veces. Solo hay un jugador que va a la alza y adivinen ¿quién es? El casino. El casino, gracias a la probabilidad del número 0, siempre gana.

Te sorprendería saber que el casino tiene varios trucos bajo la manga, y es que el casino SIEMPRE GANA.

A este fenómeno se le conoce como la ley de los grandes números y, básicamente consiste en que estas pequeñas probabilidades hacen que la tendencia se cargue de un solo lado mientras aumentan las instancias.

De igual manera supongamos que una conjunción contiene 1000 clasificadores diferentes y cada clasificador tiene un aproximado de 53% de asertividad, lo que es un poco mejor que elegir una clase al azar.

Al hacer una votación entre 1000 o más clasificadores según la ley de los grandes números, nuestro asertividad debería incrementarse.

Aunque para que esto suceda tendríamos que suponer que todos los clasificadores son perfectamente independientes uno del otro. Esto es difícil de lograr porque son entrenados sobre el mismo set de datos.

* **Aplicándolo en Python**

Vamos a ver un ejemplo de cómo ejecutaríamos un método de conjunción en Python.

Primero, tenemos que definir cuáles son los clasificadores que vamos a utilizar, en este caso vamos a utilizar el clasificador SVM y el árbol de decisión.

Como ejemplo vamos a utilizar el data set del capítulo 4, que consistía en la clasificación de empleados admitidos y rechazados de una empresa.

candidates = {'gmat': [780,750,690,710,680,730,690,720,740,690,610,690,710,680,770,610,580,650,540,590,620,600,550,550,570,670,660,580,650,660,640,620,660,660,680,650,670,580,590,690],

'gpa': [4,3.9,3.3,3.7,3.9,3.7,2.3,3.3,3.3,1.7,2.7,3.7,3.7,3.3,3.3,3,2.7,3.7,2.7,2.3,3.3,2,2.3,2.7,3,3.3,3.7,2.3,3.7,3.3,3,2.7,4,3.3,3.3,2.3,2.7,3.3,1.7,3.7],

'work\_experience': [3,4,3,5,4,6,1,4,5,1,3,5,6,4,3,1,4,6,2,3,2,1,4,1,2,6,4,2,6,5,1,2,4,6,5,1,2,1,4,5],

'admitted': [1,1,0,1,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,0,1,0,0,0,0,1,1,0,1,1,0,0,1,1,1,0,0,0,0,1]

}

df = pd.DataFrame(candidates,columns= ['gmat', 'gpa','work\_experience','admitted'])

df.head()

Table

Description automatically generated

Para crear nuestro sistema de conjunción utilizaremos el módulo *VotingClassifier*. Para esto necesitamos importar las librerías necesarias.

from sklearn.ensemble import VotingClassifier

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier

Una vez que tenemos nuestras librerías, separamos las variables predictoras *x* de nuestra variable a predecir *y*, y creamos nuestros sets de entrenamiento y nuestros sets de prueba.

x = df[['work\_experience','gpa','gmat']]

y = df['admitted']

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size = 0.3, random\_state = 42, shuffle = True)

Después, creamos los modelos clasificadores:

svm = SVC()

arbol = DecisionTreeClassifier(max\_depth=2)

Posteriormente, se crea el sistema de conjunción donde primero tenemos que pasar nuestros clasificadores por el parámetro *estimators* y, luego, tenemos que definir que el tipo de votación que queremos ejecutar es hard voting.

votos = VotingClassifier(estimators=[ ("svm", svm), ("arbol", arbol)],voting="hard")

votos.fit(x\_train,y\_train)

Una vez que creamos el modelo de votación y entrenamos el set de entrenamiento, vamos a medir su porcentaje de asertividad, esto lo haremos con la función que vimos antes: *accuracy\_score*, donde tenemos que comparar nuestras predicciones con nuestro set de prueba.

from sklearn.metrics import accuracy\_score

for i in (svm, arbol, votos):

i.fit(x\_train,y\_train)

y\_pred = i.predict(x\_test)

print(i.\_\_class\_\_.\_\_name\_\_,

accuracy\_score(y\_test,y\_pred))

Aquí, lo que estamos haciendo es simplemente tomar nuestros 3 modelos, entrenar cada uno con nuestro set de entrenamiento, crear una predicción para cada uno de ello y, al final imprimimos el nombre del modelo con el método *\_\_class\_\_.\_\_name\_\_,* en conjunto con su porcentaje de asertividad.

Graphical user interface, text

Description automatically generated

Como vemos el crear un modelo de votación hace que nuestro nivel de asertividad aumente.

Si solo utilizamos modelos de clasificación que sean capaces de estimar la clase a la que pertenece la instancia por el método que vimos antes (*predict\_proba*), entonces podemos crear un método de conjunción llamado de votación suave o soft voting.

El soft voting consiste en tomar todas esas probabilidades de que una instancia pertenezca a cierta clase, y de todos los modelos tomar el promedio de estas probabilidades para así dictaminar una respuesta.

Por ejemplo, digamos que tenemos 2 clases, la probabilidad de que sea azul está en el lado izquierdo de la coordenada y la probabilidad de que sea rojo está a la derecha.

Diagram

Description automatically generated

Como vemos cada modelo calculó sus probabilidades y al final el que tiene un mayor promedio es el color rojo, por lo que este sistema de votación elegirá el rojo.

El método de soft voting normalmente alcanza mejores resultados que el método hard voting porque en su forma de funcionar le da más peso a las clasificaciones con una probabilidad más certera.

Para utilizarlo lo único que tenemos que hacer es cambiar el parámetro de voting en VotingClassifier de voting=”hard” a voting=”soft” asegurándose de que todos tus clasificadores pueden estimar probabilidades.

En el caso del clasificador SVM se ajusta de la siguiente manera:

svm = SVC(probability=True)

arbol = DecisionTreeClassifier(max\_depth=2)

votos = VotingClassifier(estimators=[ ("svm", svm), ("arbol", arbol)],voting="soft")

votos.fit(x\_train,y\_train)

## **Bagging y Pasting**

Una forma de obtener un set de clasificadores diferentes sería utilizar algoritmos que funcionan diferente como lo acabamos de ver. También se puede poner en práctica un algoritmo por cada variable predictora de *x*, y entrenarlos en diferentes sub-sets del set de entrenamiento.

Hay 2 maneras diferentes de utilizar este método. Cuando se hace con reemplazo se le llama *BAGGING* y cuando se hace sin reemplazo se llama PASTING.

Por ejemplo, supongamos que entramos a un concurso donde podemos ganar 5 cosas diferentes, un coche, una casa, una televisión, un viaje y un celular.

El premio que vas a ganar es completamente al azar dependiendo de la ficha que saques, así que si quieres por ejemplo la casa tienes un 1/5 o 20% de probabilidades de ganarla.

Entonces supongamos que en la primera ronda alguien se llevó un coche, si el concurso fuera de tipo bagging entonces se reemplazaría el coche, haciendo que las probabilidades para ganar una casa sigan siendo de 20%.

Mientras que si el concurso fuera de tipo pasting entonces el coche ya no se reemplazaría y por lo tanto, las probabilidades de ganarte una casa sin contar el coche ahora serían de ¼ o 25%.

El truco en esto es la correlación, mientras que bagging hace que las probabilidades de cada predictor sean independientes en pasting, este tipo de correlación no es independiente, ya que la probabilidad de que se escoja algo depende de la decisión que se tomó antes.

En ML es prácticamente lo mismo, mientras que en bagging toma datos y los reemplaza para entrenar el modelo, pasting toma datos, pero no los reemplaza.

Una vez que se ha entrenado a todos los predictores, el sistema de conjunción puede hacer una predicción basándose en todos los resultados obtenidos de los predictores de *x.*

Por ejemplo, en el ejercicio anterior, se hubiera hecho una predicción solo basándose en el score GMAT, otra basándose en el GPA y otra basándose en los años de experiencia.

En el método hard voting el modelo selecciona la predicción más votada o la más frecuente, mientras que para la regresión se utiliza el promedio de las predicciones.

### **Bagging y Pasting con Scikit-Learn-learn**

Para utilizar estos métodos con Python utilizaremos la librería *BaggingClassifier* para clasificadores y *BaggingRegressor* para regresiones.

Como sabemos tenemos que seleccionar algún modelo de los que ya conocemos para utilizar estos métodos. Escojamos el modelo de árboles de decisión.

# Bagging

from sklearn.ensemble import BaggingClassifier

bagging = BaggingClassifier(DecisionTreeClassifier(),

n\_estimators=100,

max\_samples=10,

bootstrap=True)

bagging.fit(x\_train,y\_train)

y\_pred = bagging.predict(x\_test)

Como se puede observar hay varios parámetros que tenemos que conocer para implementar este sistema. El primer parámetro es el modelo por utilizar, en este caso, los árboles de decisión.

Nuestro segundo parámetro es *n\_estimators* o número de estimadores, el cual nos sirve para indicar cuántos modelos queremos crear. En este caso estamos crearemos 100 árboles de decisión.

El parámetro *max\_samples* nos permite señalar cuál es la cantidad de instancias que queremos tomar por cada árbol de decisión, en este caso vamos a tomar 10 instancias de nuestro set de entrenamiento completamente aleatorias.

Por último, el parámetro *Bootstrap* funciona para especificar si queremos que nuestro modelo sea de tipo bagging o pasting.Cuando este parámetro es True entonces utilizaremos reemplazos o bagging mientras que si es False entonces no utilizaremos reemplazos para las instancias y será de tipo pasting.

Un ejemplo de pasting con SVM, 50 clasificadores y 15 instancias aleatorias por clasificar quedaría de la siguiente manera:

# Pasting

from sklearn.ensemble import BaggingClassifier

pasting = BaggingClassifier(SVC(),

n\_estimators=50,

max\_samples=15,

bootstrap=False)

pasting.fit(x\_train,y\_train)

y\_pred = pasting.predict(x\_test)

La verdad es que el método de bagging es más recomendable ya que normalmente obtiene mejores resultados que pasting. Esto se debe a que, al reemplazar se elimina cierta correlación en los resultados haciendo que cada modelo sea más independiente.

A continuación, podemos ver la diferencia de cómo se vería el resultado de un clasificador de árbol de decisión sin restricciones contra un clasificador compuesto por un conjunto de árboles de decisión sin restricciones utilizando el método de bagging.

Chart, scatter chart

Description automatically generated

En la izquierda, la tendencia de la clasificación indica un sobreajuste del modelo con los datos, en cambio en la imagen de la derecha, se observa una mejor clasificación gracias a que se utilizó bagging en su entrenamiento.

### **Evaluación Fuera de la Bolsa**

Cuando utilizamos bagging, algunas instancias de nuestro set de datos se pueden utilizar varias veces por cada predictor, mientras que al ser completamente aleatorio puede que algunas ni siquiera se utilicen.

Cuando utilizamos la librería de Scikit-Learn *BaggingClassifier* el predictor sólo utiliza alrededor del 63% de las instancias del set de entrenamiento. El 37% restante es conocido como instancias fuera de la bolsa o out of the bag, aunque mayormente las encontraremos como instancias OOB.

Un modelo de bagging puede ser evaluado utilizando estas instancias OOB, ya que podríamos verlo como si fuera un tipo de set de entrenamiento y set de prueba. De manera que usemos el 63% (aproximadamente) para entrenar el modelo y las instancias OOB utilizarlas para evaluar el mismo.

Ejemplo:

bagging = BaggingClassifier(DecisionTreeClassifier(),

n\_estimators=100,

max\_samples=10,

bootstrap=True, oob\_score=True)

Como vemos el único parámetro que se añadió además de los que ya habíamos mencionado es *el oob\_score* el cual lo activamos estableciéndose como *True.*

Para utilizarlo lo único que tenemos que hacer es entrenar el modelo y mandarlo llamar:

bagging.fit(x\_train,y\_train)

bagging.oob\_score\_



Como vemos, utilizando las instancias OOB para medir la precisión del modelo nos da poco más de un 89%.

Ahora, vamos a medir la precisión utilizando nuestro set de prueba.

y\_pred = bagging.predict(x\_test)

accuracy\_score(y\_test,y\_pred)



Con el set de prueba se alcanzó una precisión de poco más del 91%. Esto está bastante acercado a lo que se logró con la evaluación OOB.

Este método es bastante útil ya que nos permite tener una segunda evaluación además de solo el set de prueba.

Con la evaluación OOB tenemos el método *oob\_decision\_function\_,* el cual nos indicará cual es la probabilidad de que cada instancia que está fuera de la bolsa pertenezca a cada una de nuestras clases.

bagging.oob\_decision\_function\_

Text

Description automatically generated

## **Parches Aleatorios y Subespacios Aleatorios**

Cuando utilizamos el método de bagging, también podemos controlar cuántos predictores queremos utilizar para nuestro modelo. Por ejemplo, en este caso tenemos 3 (GMAT, GPA y experiencia), pero qué tal si tuviéramos un set de datos con 1000 predictores, pues para controlar cuantos predictores queremos utilizar tenemos el parámetro llamado *max\_features.*

Supongamos que en vez de mis 3 predictores solo quiero utilizar 2 de ellos, pues el modelo quedaría de la siguiente manera.

bagging = BaggingClassifier(DecisionTreeClassifier(),

n\_estimators=100,

max\_samples=10,

bootstrap=True,

max\_features=2)

bagging.fit(x\_train,y\_train)

Como vemos lo único diferente al crearlo fue que se añadió el parámetro *max\_features,* especificando que solo queremos utilizar 2 de ellos.

O para tomar y reemplazar predictores como si fuera bagging con las instancias, también podemos utilizar el parámetro *Bootstrap\_features.*

Este método es bastante útil ya que crea sets de datos con instancias y predictores aleatorios haciendo que la correlación entre los diferentes sets de datos disminuya todavía más, y recordemos que, entre menos correlación, mejores modelos podremos crear.

Aquí es donde entran los llamados subespacios y parches aleatorios.Un subespacio aleatorio se utiliza para reducir la correlación del método bagging, y consiste en limitar el uso de predictores para cada set de instancias en el clasificador.

Por lo que se vería como el método que utilizamos antes:

bagging = BaggingClassifier(DecisionTreeClassifier(),

n\_estimators=100,

max\_samples=10,

bootstrap=True,

max\_features=2)

bagging.fit(x\_train,y\_train)

Como vemos un subespacio aleatorio se consigue solo agregando el parámetro *max\_features*.

Mientras que un parche aleatorio requiere que se tome una cantidad especifica de predictores y que se reemplacen cada vez que se utilizan, creando aun mayor independencia entre cada set de instancias, haciendo cada modelo más único y reduciendo su correlación.

Se vería de la siguiente manera:

bagging = BaggingClassifier(DecisionTreeClassifier(),

n\_estimators=100,

max\_samples=10,

bootstrap=True,

max\_features=2,

bootstrap\_features=True)

bagging.fit(x\_train,y\_train)

En este caso señalamos que deseamos que los predictores se reemplacen también.

## **Random Forests**

Como ya sabemos, Random Forest o bosque aleatorio es un conjunto de árboles de decisión, generalmente estos son entrenados utilizando el método bagging, aunque también hay casos donde se utiliza pasting.

También en un Random Forest es común encontrar que el máximo de instancias por modelo o *max\_samples* sea igual al número de instancias del set de entrenamiento, por lo que no hay ninguna restricción en esta parte.

El Random Forest es tan utilizado que no necesitamos crear un *BaggingClassifier* y modificarlo para que actúe como un Random Forest

Para esto Scikit-Learn ya nos da un módulo configurado especialmente para Random Forest y se llaman RandomForestClassifier (para entrenar un clasificador) y RandomForestRegressor (para entrenar una regresión).

Es importante mencionar que estos módulos de Random Forest incluyen de igual manera los parámetros que vimos antes tanto para regularizar los árboles de decisión como para configurar el modelo bagging.

Ejemplo, vamos a utilizar el parámetro de árboles de decisión *max\_leaf\_nodes* y el parámetro de bagging, *max\_features* para crear un Random Forest de 100 árboles.

from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n\_estimators=100,

max\_leaf\_nodes=4,

max\_features=2)

rf.fit(x\_train, y\_train)

Listo, así de sencillo es crear un modelo de RandomForest con Scikit-Learn.

Una ventaja del diseño de Random Forest es que introduce mayor aleatoriedad al momento de que los nodos crecen. Recordemos que con el árbol de decisión al dividir los nodos se buscaba el predictor que ofrecía mayor pureza (*Gini*) a los siguientes nodos.

En el caso del Random Forest al tener instancias y predictores aleatorios se crean modelos más independientes y únicos. Como ya vimos en este capítulo, queremos buscar la mayor correlación posible entre modelos por lo que esta característica ayuda a la creación de buenos modelos.

Como ya sabemos, Random Forest proviene del método de conjunción bagging, vamos a crear un modelo igual al que creamos anteriormente de Random Forest pero utilizando bagging.

bagging = BaggingClassifier(DecisionTreeClassifier(max\_leaf\_nodes=4),

n\_estimators=100,

max\_features=2,

bootstrap=True)

bagging.fit(x\_train,y\_train)

Con ese ejemplo tenemos un modelo que funciona de manera similar al que creamos con el módulo de Random Forest, aunque siempre será más sencillo y óptimo utilizar Random Forest.

### **Árboles Adicionales**

Como mencionamos antes, en un Random Forest al dividir un nodo de un árbol de decisión, se toman en cuenta sólo una cantidad aleatoria de predictores para hacer la división pero, para aumentar la aleatoriedad de esa división de nodos, podemos utilizar umbrales aleatorios para cada predictor en vez de buscar el umbral óptimo de división.

Por ejemplo, supongamos que la edad óptima de división para un árbol es mayor o igual a 39. Con este tipo de modelo eso no importa, solo selecciona una edad al azar como mayor o igual a 16.

Esto por supuesto crea árboles que necesitan una cantidad de nodos mucho más grande para optimizarse, a esta técnica se le conoce como árboles extremadamente aleatorizados o árboles adicionales.

Para crear un modelo con este sistema Scikit-Learn nos proporciona la librería *ExtraTreesClassifier* para clasificadores y ExtraTreesRegressor para regresiones.

Ejemplo:

from sklearn.ensemble import ExtraTreesClassifier

extra = ExtraTreesClassifier(n\_estimators=100,

max\_features=2,

max\_leaf\_nodes=4)

extra.fit(x\_train,y\_train)

La verdad es que es difícil saber cuál modelo dará mejor resultado, si Random Forest o árboles adicionales, por lo que se recomienda utilizar ambos y comparar cual da un mejor desempeño. Aunque Random Forest es comúnmente más utilizado.

### **Importancia de Predictores**

Otra gran cualidad de los Random Forest es que sirven para evaluar la importancia en el modelo de cada predictor.

En otras palabras, digamos que para crear nuestro modelo usamos las 3 variables GMAT, GPA y experiencia. Pues Random Forest puede decir cuál es el predictor que más influencia tiene el modelo, puede ser que, por ejemplo, los años de experiencia tengan mucho más peso que las calificaciones.

Esto lo hace midiendo cuánto reducen la impureza en promedio los nodos del árbol que usan esa función.

Para utilizar esta característica solo tenemos que comprimir los nombres y los resultados en uno solo con la función zip y luego imprimirlo.

random = RandomForestClassifier(n\_estimators=100)

random.fit(x\_train,y\_train)

for nombre, score in zip(x.columns, random.feature\_importances\_):

print(nombre,score)

Text

Description automatically generated

Como vemos en este caso la variable que tiene más peso es la de GPA, ya que de este método depende el 42% del porqué una instancia se calificaría de cierta manera, aunque en verdad en este modelo las 3 variables tienen bastante relevancia, en ocasiones habrá variables donde se tenga una importancia de menos de 0%, con las cuales ya veremos técnicas para lidiar con ellas.

## **Boosting**

Boosting o impulso se refiere a cualquier método de conjunción que tiene la posibilidad de combinar un grupo de weak learners o modelo que no tienen buena asertividad y juntarlos para crear un strong learner o modelo con un nivel de asertividad alto.

Este método funciona para entrenar modelos secuencialmente, lo que significa que se entrenan en forma de cadena, mientras cada modelo intenta mejorar a su predecesor.Por eso se llama impulsar, porque cada modelo en la cadena impulsa al siguiente a que sea un mejor modelo.

Recordemos que en bagging cada modelo se entrenaba por separado y mientras más independientes fueran mejor, en este caso boosting si toma el modelo anterior y lo mejora.

Diagram

Description automatically generated

Para lograr que se cumpla este objetivo hay diferentes métodos disponibles, pero hay 2 que sobresalen por mucho del resto, uno es conocido como adaptive boosting y abreviado como **AdaBoost** y al otro se le conoce como **gradient boosting** o impulso por gradiente.

### **AdaBoost**

La técnica en la que se enfoca AdaBoost consiste en presentar atención a los errores cometidos, de manera que cuando llega un nuevo predictor, este pone atención a los errores de su antecesor para poder mejorar.

Por ejemplo, cuando utilizamos un clasificador con AdaBoost, primero se entrena el clasificador base (árbol de decisión, SVM o el que tengas en mente) después, el modelo toma los pesos de las instancias que no se clasificaron correctamente y utiliza estos pesos para entrenar el siguiente clasificador, repitiendo estos pasos sucesivamente.

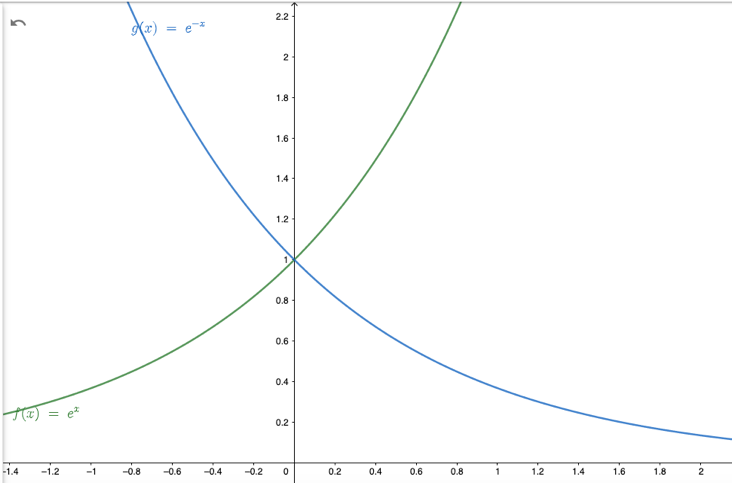
Por ejemplo, digamos que tenemos un set de datos con 10 instancias, empezamos con nuestro clasificador base y cada una de esas instancias tiene un peso de .

Supongamos que, de esas 10 instancias, nuestro clasificador predijo 3 incorrectas, si sumamos esas 3 incorrectas tendríamos . Para cada clasificador en nuestro modelo, se utiliza una fórmula conocida como el peso de cada clasificador.

Esa formula es por lo que en el caso de nuestro clasificador tendríamos un peso de lo cual es igual a 0.42.

Una vez que calculamos el peso de nuestro clasificador lo que hace el AdaBoost es calcular los pesos por cada instancia en el modelo.

Como dijimos cada instancia es igual a , esto solo aplica para nuestro clasificador base, para los siguientes clasificadores lo que sucede es que se calcula de nuevo el valor de cada instancia con la fórmula para las instancias que se clasificaron bien y para las instancias que se clasificaron mal.



Cuando se trata de aquellas instancias bien clasificadas, mientras más aumenta el peso del clasificador menor será el peso de la instancia. Inversamente proporcional, para las instancias mal clasificadas, mientras más aumenta el peso del clasificador mayor será el peso de la instancia.

Por ejemplo, para nuestro clasificador número 2, las instancias que fueron clasificadas bien en el clasificador número 1, si suponemos un ritmo de aprendizaje de , tendrán un peso de mientras que las instancias que se clasificaron mal tendrán un peso de .

De esta manera el segundo clasificador tendrá más énfasis en estas instancias al momento de hacer sus divisiones o sus clasificaciones en estas instancias. El procedimiento se repite hasta llegar al último clasificador.

* + - **AdaBoost con Scikit-Learn**

Para utilizar el método AdaBoost con Scikit-Learn, lo único que tenemos que hacer es utilizar la librería *AdaBoostClassifier* para clasificadores y *AdaBoostRegressor* para regresiones.

Un tema importante con el AdaBoost es cuidar que el ritmo de aprendizaje sea adecuado para no apresurar ni aletargar la velocidad. Veamos un ejemplo de cómo influye el ritmo de aprendizaje en nuestro clasificador.

Scatter chart

Description automatically generated

Para ejemplificar estos ejemplos, aquí está un código que crea un modelo AdaBoost con un ritmo de aprendizaje de 0.1 y 100 árboles de decisión. Para explorar pueden cambiar el ritmo de aprendizaje y observar los resultados.

from sklearn.ensemble import AdaBoostClassifier

ada = AdaBoostClassifier(DecisionTreeClassifier(),

n\_estimators=100,

learning\_rate=0.1)

ada.fit(x\_train,y\_train)

### **Boosting con Gradiente**

Otra forma muy popular de utilizar el boosting es con el método del gradiente. Justo como el AdaBoost, el boosting con gradiente funciona agregando predictores al método de conjunción secuencialmente, cada uno corrigiendo los errores cometidos por su antecesor. La gran diferencia es que este modelo mide los errores residuales del predictor anterior, no como el AdaBoost que modifica los pesos de cada instancia una por una.

Para ejemplificar este método, vamos a crear un modelo de regresión utilizando el boosting con gradiente.

Empecemos creando nuestros datos, para que sea fácil de entender visualmente empezaremos con una parábola e intentaremos que el modelo se ajuste lo mejor posible a ella.

m = 100

x = np.linspace(-0.5,0.5,m)

y = 25\*x\*\*2 + np.random.random(m) -0.5

plt.scatter(x,y)

Chart, scatter chart

Description automatically generated

Es hora de crear el predictor con árboles de decisión, recordemos que siempre es una buena idea regularizarlos, aunque sea un poco por lo que le daremos un máximo de nodos de 2 más el nodo base.

from sklearn.tree import DecisionTreeRegressor

x = x.reshape(-1,1)

arbol1 = DecisionTreeRegressor(max\_depth=2)

arbol1.fit(x,y)

Así es como se vería nuestro modelo entrenado.

Chart, scatter chart

Description automatically generated

Una vez que tenemos nuestro modelo entrenado, recordemos que los errores residuales son la diferencia entre las predicciones del modelo y el valor de la variable a predecir, por lo que simplemente tenemos que calcular esas predicciones y restarlas a la variable y.

y2 = y - arbol1.predict(x)

Ahora que tenemos los residuales calculados podemos crear nuestro segundo predictor con base a esos residuales.

arbol2 = DecisionTreeRegressor(max\_depth=2)

arbol2.fit(x,y2)

Recordemos que estamos entrenando el modelo sobre los residuales, por lo que se vería de esta forma.

Chart

Description automatically generated

El truco en este tipo de boosting es que si sumamos las predicciones estas se van adaptando mejor al modelo, ya que eliminan el ruido de los residuales.

Ejemplo:

Solo con un árbol de decisión las predicciones se ven de esta manera.

Chart, scatter chart

Description automatically generated

Ahora vamos a ver que sucede si sumamos las predicciones del primer predictor con el segundo.

y\_pred = sum(tree.predict(x) for tree in (arbol1, arbol2))

Digamos que al adaptar un modelo a los residuales y luego sumárselo al modelo original, este al enfocarse en los lugares mal clasificados reduce estos residuales y se ve de esta forma.

Chart, scatter chart

Description automatically generated

Como podemos observar ahora el modelo se adapta mejor a nuestros datos.

Vamos a repetir el mismo procedimiento para crear el árbol 3.

y3 = y - arbol2.predict(x)

arbol3 = DecisionTreeRegressor(max\_depth=2)

arbol3.fit(x,y3)

Al estar mejor adaptado el segundo modelo que el primero, los residuales son menores por lo que nuestro tercer modelo se adapta mejor.

Chart, scatter chart

Description automatically generated

Ahora vamos a ver que sucede si sumamos los 3 árboles que tenemos hasta ahora:

y\_pred = sum(arbol.predict(x) for arbol in (arbol1, arbol2, arbol3))

Chart, scatter chart

Description automatically generated

Como vemos ahora el modelo se adapta mucho mejor que el segundo modelo.

También es importante mencionar que entre más árboles agreguemos mejor se adaptará nuestro modelo a los datos, aunque siempre hay que cuidar no llegar al sobreajuste.

* + - **AdaBoost con Scikit-Learn**

Afortunadamente Scikit-Learn cuenta con un módulo específico para este tipo de boosting, por lo que no necesitamos hacerlo paso por paso como lo hicimos anteriormente, esto era solo para cuestiones de entendimiento.

Esta librería se llama *GradientBoostingRegressor* y además de tener los parámetros para regularizar los árboles de decisión que ya conocemos, tiene 2 parámetros que son muy importantes. *n\_estimators* indica cuántos predictores queremos crear y *learning\_rate* indica el ritmo de aprendizaje con el que queremos que nuestro modelo intente adaptarse a los datos.

Vamos a crear exactamente el mismo modelo que creamos antes con la librería de Scikit-Learn.

from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max\_depth=2,

n\_estimators=3,

learning\_rate=1.0)

gbrt.fit(x,y)

En este caso cuando el ritmo de aprendizaje es 1, el modelo se adapta a la velocidad que llevábamos cuando sumamos modelo por modelo, pero le puedes restar o aumentar la velocidad de adaptación.

Aunque si bajamos la velocidad entonces puede que necesites más predictores para que el modelo se adapte bien, y en muchos casos es recomendable hacerlo, ejemplo:

Chart, scatter chart

Description automatically generated

Aunque es difícil saber cuántos árboles o predictores son los óptimos para obtener el mejor resultado, tenemos la opción de utilizar el método de early stoppping que vimos en el capítulo 4.

El método es el mismo, básicamente tenemos que crear un ciclo for donde evaluamos el modelo con una métrica de error. En este caso utilizaremos el error medio cuadrado.

from sklearn.metrics import mean\_squared\_error

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y)

gbrt = GradientBoostingRegressor(max\_depth=2, n\_estimators=120)

gbrt.fit(x\_train, y\_train)

errores = [mean\_squared\_error(y\_test,y\_pred) for y\_pred in gbrt.staged\_predict(x\_test)]

mejor = np.argmin(errores)

mejor



En este caso el algoritmo arroja que el valor óptimo de árboles a utilizar es 56 de los 120 árboles que probamos.

Chart, scatter chart

Description automatically generated

Nota: en la imagen, a la izquierda podemos ver que el error mínimo está en 56 y en la derecha se muestra el modelo con 56 predictores.

Vale la pena mencionar que si no se quiere programar con early stopping existe una librería para Python con la cual podemos evaluar el early stopping automáticamente.

La librería se llama **xgboost** y lo primero que tenemos que hacer es instalarla:

pip install xgboost

Una vez instalado lo importamos (funciona bastante parecido a las librerías de Scikit-Learn) en este caso la función para regresiones se llama *XGBRegressor.*

from xgboost import XGBRegressor

Después, creamos nuestro modelo.

xgb = XGBRegressor()

xgb.fit(x\_train, y\_train)

y\_pred = xgb.predict(x\_test)

Esa sería la forma tradicional de crear predicciones, justo como se haría con Scikit-Learn pero, si queremos utilizar la función de early stopping tendríamos que hacerlo de la siguiente manera:

xgb = XGBRegressor()

xgb.fit(x\_train, y\_train,

eval\_set=[(x\_test,y\_test)], early\_stopping\_rounds=1)

y\_pred = xgb.predict(x\_test)

Como se puede observar, para utilizar el early stopping solo se necesita indicar con que set de datos queremos evaluar el modelo, en este caso utilizamos los sets de prueba.

También podemos indicar el número de veces que queremos que se evalué, ya que los resultados pueden variar.

Esta librería utiliza 12 árboles de decisión predeterminadamente pero con el parámetro *n\_estimators* se puede especificar cuántos árboles utilizar.

Al correr el código automáticamente nos da la lista de errores calculados.

Text

Description automatically generated

Aquí claramente podemos ver que el error mínimo de los 12 árboles es cuando se utilizan solo 11. Lo que da pauta a ajustar el modelo para crear solo 11 árboles y ahorrar la carga computacional.

## **Stacking**

El ultimo método de conjunción del que vamos a hablar en este capítulo se llama **stacking** que en español se traduce como apilar, en algunos casos también se encuentra como stacked generalization o generalización apilada.

Este método se basa en una simple idea, dejar de lado las funciones triviales (como utilizar hard voting) y solo sumar los diferentes modelos que tenemos y crear un modelo que mezcle todas las predicciones. En el caso de stacking también es común combinar diferentes tipos de clasificadores o regresores.

Por ejemplo, supongamos que tenemos un regresor lineal, un regresor polinomial y un Random Forest para crear una predicción.

Diagram

Description automatically generated

Entonces llega una nueva instancia, pasa por nuestros modelos predictores, estos modelos arrojan una predicción, y un blender que literalmente significa batidor o mezclador, crea una nueva predicción promediando las predicciones anteriores.

El método de stacking también puede funcionar dividiendo el set de datos inicial para así poder utilizar un set de datos para entrenamiento y un set de datos para crear predicciones.

Diagram

Description automatically generated

Como podemos observar, esta división de datos se utiliza porque se entrena con un set y con otro se crean las predicciones, lo cual ayuda a evitar el sobreajuste desde el inicio del modelo.

* + - **Stacking en Scikit-Learn**

Podemos utilizar el método stacking con Scikit-Learn con la librería StackingClassifier para clasificadores y StackingRegressor para regresiones.

from sklearn.ensemble import StackingRegressor

from sklearn.linear\_model import LinearRegression

from sklearn.ensemble import RandomForestRegressor

arbol = DecisionTreeRegressor(max\_depth=20)

lineal = LinearRegression()

random = RandomForestRegressor()

stacking = StackingRegressor(estimators=[("arbol", arbol),

("lineal", lineal),

("random", random)])

stacking.fit(x,y)

Como vemos lo único que tenemos que hacer para entrenar nuestro modelo de stacking es, crear los diferentes predictores que queremos utilizar y con el parámetro *estimators* añadiros al modelo, de esta forma el módulo de Scikit-Learn creará automáticamente el *blender* y, por ende, las predicciones se basarán en nuestros tres diferentes modelos de clasificación.