**Modelos de Ensemble Boosting**

Los algoritmos de Boosting son un conjunto de clasificadores de baja precisión para crear clasificadores de alta precisión. El clasificador de baja precisión (o clasificador debil) ofrece una precisión superior a la de lanzar una moneda. El clasificador de alta precisión (o clasificador fuerte) ofrece una tasa de error cercana a cero. El algoritmo de Boosting permite rastrear el modelo que no cumple con la predicción precisa. **Los algoritmos de boosting se ven menos afectados por problemas de sobreajuste.** Cuando hablamos de **Boosting** nos referimos a:

* Algoritmos en los que los modelos se agregan secuencialmente…
* Y los modelos posteriores en la secuencia corrigen las predicciones hechas por los modelos anteriores

[**Algoritmo AdaBoost:**](https://www.datacamp.com/tutorial/adaboost-classifier-python)

Este corresponde a uno de los algoritmos clasificadores de refuerzo de conjunto que combina multiples clasificadores para asi poder aumentar su precisión. **AdaBoost** es un método iterativo de conjunto. El clasificados **AdaBoost** crea un clasificador robusto combinando varios clasificadores de bajo rendimiento obteniendo asi un clasificador robusto de alta precisión. El concepto básico de **AdaBoost** consiste en establecer los pesos de los clasificadores y entrenar la muestra de datos en cada iteración, garantizando asi una predicción precisa de observaciones inusuales.

* [**AdaBoost para clasificación:**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html)

Un clasificador de **AdaBoost** es un meta estimador que comienza ajustando un clasificador en el conjunto de datos originales y luego ajusta copias adicionales del clasificador en el mismo conjunto de datos, pero donde los pesos de las instancias clasificadas incorrectamente se ajustan de modo que los clasificadores posteriores se centran mas en los casos difíciles. Utiliza como base **DecisionTreeClassifier**

Par este modelo se debe de tener en cuenta que:

* + **AdaBoost** es relativamente sensible a los outliers porque incrementa progresivamente su peso en las iteraciones.
  + **AdaBoost** puede tener problemas con clases desbalanceadas porque inicialmente da igual peso a todas las muestras.
  + Variables numéricas
    - No requiere escalamiento/estandarización ya que usa arboles
    - Puede beneficiarse de normalización en algunos casos
  + Variables categóricas
    - Debe de ser codificadas (One-Hot, Ordinal)
    - Evitar **Target Encoding** (Puede causar data Leakage- Fuga de informacion)
      * Para datos de alta cardinalidad se recomienda:
        + Leave-One-Out Encoding: Variante de Target Encoding que calcula la media del target para cada categoría excluyendo la fila actual
        + Bayesian Target Encoding: Versión estadísticamente robusta de Target Encoding que combina Media de la categoría, Media global, Parámetros de regularización (smoothing).
        + Encoding por Frecuencia: Codifica cada categoría por su frecuencia relativa en los datos.
        + One-Hot Encoding con Agrupamiento: One-Hot tradicional pero agrupando categorías poco frecuentes.
  + No maneja datos nulos, estos deben de ser imputados
  + Variables fechas
    - No puden usarse directamente
    - Deben de transformarse en Dias, Mes, Año, diferencia entre fechas o variables cíclicas.

**Ej AdaBoost Clasidicacion:**

# Importación de bibliotecas necesarias

from numpy import mean # Para calcular la media de los resultados

from numpy import std # Para calcular la desviación estándar de los resultados

from sklearn.model\_selection import cross\_val\_score # Para validación cruzada

from sklearn.model\_selection import RepeatedStratifiedKFold # Para estrategia de validación

from sklearn.ensemble import AdaBoostClassifier # El clasificador AdaBoost

# Se crea el modelo AdaBoost

model = AdaBoostClassifier(algorithm="SAMME")

# Configuración de validación cruzada repetida y estratificada

cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=4)

# Evaluación del modelo con validación cruzada

n\_scores = cross\_val\_score(model, X, y, scoring="accuracy", cv=cv, n\_jobs=-1)

# Mostrar resultados

print('Accuracy: %.3f (%.3f)' % (mean(n\_scores), std(n\_scores)))

En este caso de clasificacion, empezamos importando las liberias requeridas para aplicar el modelo y tambien hacer la evaluación de este. La evaluación que se va hacer es por medio de una validación cruzada, por lo tanto no se requiere separar el conjunto de datos en trein y test.

* + Se crea el modelo y se le pasa el parámetro **algorithm=”SAMME”**  que corresponde al algoritmo para problemas de clasificación.
    - Se debe de poner este parámetro porque otro que utilizaba este modelo era **“SAMME.R”** pero este ya se encuentra obsoleto, y como este es el parámetro predeterminado del modelo, debemos de ponerle “SAMME” con el fin de que no salte una alerta.
  + Se configura la validación cruzada para la evaluación del modelo.
    - **n\_splits=10:** 10 folds (particiones) en cada repetición
    - **n\_repeats=3**: Repite el proceso de validación 3 veces
    - **random\_state=4**: Semilla para reproducibilidad

**Nota:** Tambien se puede utilizar le modelo **Kfold** para la validación cruzada: cv = KFold(n\_splits=10, random\_state=7, shuffle=True)

* + Se evalua el modelo con validación cruzada.
    - model : Modelo AdaBoost a evaluar
    - X - Features/variables independientes
    - y - Target/variable dependiente
    - scoring="accuracy" - Métrica de evaluación (precisión)
    - cv=cv - Estrategia de validación configurada
    - n\_jobs=-1 - Paralelización usando todos los núcleos del CPU (**Se puede corres la validación sin este parametro**)
  + Se muestra el resultado de la evaluación, se muestra el promedio de las diferentes iteraciones y la desviación estándar.
* **AdaBoost para regresion:**

Es un estimador que comienza ajustando un regresor al conjunto de datos original y luego ajusta copias adicionales del regresor al mismo conjunto de datos, pero donde los pesos de las instancias se ajustan según el error de la prediccion actual. Por lo tanto, los regresores posteriores se centran mas en casos difíciles. Utiliza como base el modelo **DecisionTreeRegressor**

Para este modelo se debe de tener en cuenta:

* + Se utiliza cuando se tiene problemas de regresión (prediccion de valores continuos)
  + Se quiere mejorar el rendimiento del modelo base simple
  + Los datos tienen patrones complejos que un solo modelo no puede capturar bien
  + Puede verse afectado por los outliers, ya que pueden recibir pesos altos continuamente.
  + Si se hacen demasiadas iteraciones puede sufrir de sobe ajuste
  + En el procesamiento de variables aplica los mismo que el modelo de **AdaBoost para clasificación**
  + Este tipo de modelo puede manejar alta dimensionalidad pero se debe de tratar de reducir con el fin de mejor el modelo y evitar sobreajustes

**Ej AdaBoost regresión**

#Se importan las librerias para el caso de regresion

from numpy import mean

from numpy import std

from sklearn.datasets import make\_regression

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import RepeatedKFold

from sklearn.ensemble import AdaBoostRegressor

#Se crea el modelo

model = AdaBoostRegressor()

#Configuración de validación cruzada repetida

cv = RepeatedKFold(n\_splits=10,n\_repeats=3, random\_state=3)

# Evaluación del modelo con validación cruzada

n\_scores = cross\_val\_score(model, X, y, scoring='neg\_mean\_absolute\_error', cv=cv, n\_jobs=-1)

#Se muestra el resultado de la validacion

print('MAE: %.3f (%.3f)' % (mean(n\_scores), std(n\_scores)))

Como en todos los modelos empezamos importando las librerías requeridas para crear y evaluar el modelo. En este caso, como en de clasificación no se divide el conjunto de datos en train y test, ya que el proceso se hace por medio de validación cruzada **Kfold** y este hace el proceso de división de manera automática.

* + Se crea el modelo, donde se utiliza **AdaBoostRegressor**() y no se le pasa ningún parámetro inicialmente. Tener presente que este funciona con arboles de decisión por defecto y si queremos cambiarle el modelo base debemos de crearlo y pasárselo.
  + Se realiza la configuración de la validación cruzada, donde va a realizar la iteración en 10 divisiones y va a repetir el proceso 3 veces.
  + Se evalua el modelo en las diferentes iteraciones de Kfold, y se utiliza el **scoring** “neg\_mean\_absolute\_error'”
  + Por ultimo se muestra el promedio de las diferentes evaluaciones del modelo y la desviación estándar.
    - Entre mas alto sea el promedio y menor la desviación, mejor será el modelo.
* **Hiperparametros de AdaBoost**
  + Numero de arboles:
    - Cada árbol de decisión esta diseñados para ser un clasificador débil
    - Como tal se utiliza un árbol de decisión de un nivel, siginifica la profundidad del árbol o las hojas que va a crear.
    - El numero de arboles se puede establecer mediante el argumento **n\_estimators** (el valor predeterminado es 50) y se recomienda que sea un numero alto para que el modelo funcione bien

Para revisar que hiperparametro **n\_stimator (**numero de arboles**)** se adapta mejor podemos hacer una iteración con el modelo **AdaboostClassifier()** donde lo evaluemos con diferentes valores en **n\_stimator.**

**Ej validacio hiperparametro n\_stimator:**

Como buena practica se crean funciones para evaluar el modelo y ejecutar el modelo.

#Se crea funcion para evaluar el modelo de clasificacion

def evaluate\_model(model, X, y):

"""Se utiliza una evaluacion con validacion cruzada"""

cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=1)

scores = cross\_val\_score(model, X, y, scoring='accuracy', cv=cv, n\_jobs=-1)

return scores

Esta función se crea con el fin de evaluar las diferentes iteraciones del modelo por medio de validación cruzada, donde se le debe de pasar el modelo, variable X y variables Y y utiliza el **scoring** de **“accuracy”** y retorna el resultado de la evaluación.

#Se crea funcion donde iteramos el modelo con diferentes cantidad de arboles

def get\_models():

models = dict()

# definir el número de árboles

n\_trees = [10,50,100,500,100]

for n in n\_trees:

models[str(n)] = AdaBoostClassifier(algorithm="SAMME", n\_estimators=n)

return models

Esta función se creo con el fin de iterar el modelo de AdaBoost para clasificación con diferentes valores en el parámetro **n\_estimators** y asi al final visualizar que valore tiene mejor resultado.

* + - * En esta función se crea un diccionario **models** donde se almacenara un modelo para cada interacion de **n\_trees (**cantidad de arboles**),** es decir que para cada iteracion en el bucle, agregara un valore al diccionario donde la llave será 10, 50, 100, etc y el valor será le modelo con los parámetros de cada iteracion.

Ej: models[10] = AdaBoostClassifier(algorithm="SAMME", n\_estimators=10)

* + - * Al final devuelve ese diccionario con su respectiva clave, valor

Despues de crear las funciones procedemos a iterar sobre los diferentes hiperparametros establecidos en la función **get\_models()** y evaluar el resultado de estos.

#Se ejecutan las diferentes funciones creada para evaluar los diferentes parametros

models = get\_models()

# evaluar los modelos

results, names = list(),list()

for name, model in models.items():

# evaluar el modelo

scores = evaluate\_model(model, X, y)

# almacenar los resultados

results.append(scores)

names.append(name)

print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))

* + Emepzamos definiendo la variable que va almacenar el diccionario con los modelos que contiene los diferentes hiperparametros de **n\_estimators**
  + Creamos 2 variables que van almacenar el resultado del scoring y el nombre al que le corresponde cada evaluación. (El nombre es que hiperparametro se esta utilizando)
  + Se hace un ciclo, donde vamos a iterar por el nombre y modelo que este almacenos en el diccionario models
  + Se evaluar cada iteración del modelo con la función previamente creada, donde se utiliza la metrica “Accuracy” y se hace por medio de validación cruzada.
  + Se guarda el resultado de la validación y el nombre la que corresponde cada iteracion en las listas definidas antes de bucle
  + Al final se imprime el resultado (promedio y desviacion) de cada evaluación.

Para finalizar podemos visualizar el resultado de las diferentes iteracion y asi poder ver que hiperparametro de **n\_estimators** es el que mejor resultado obtuvo y eso lo hacemos por medio de un diagrama de cajas donde va a mostrar para cada hiperparametro cual fue su rango de la validación cruzada

#Gaficando el resultado de cada iteracion del modelo

pyplot.boxplot(results, labels=names, showmeans=True)

pyplot.show()

**Interpretacion.** Despues de visualizar la grafica se escoge el hiperparametro que muestre menos varianza en el diagrama de caja.

-------------------------------------------------------------------------------------------------

* + **Profundidad del árbol:**

Podemos hacer que los modelos utilizados en el conjunto sean menos débiles (mas habiles) aumentando la profundidad del árbol de decisión. Ya que el valor predeterminado que utiliza **AdaBoost** es 1.

Para poder encontrar el valor mas optimo de profundidad, debemos de crear el modelo débil por separado, es decir que primero debemos de definir el modelo **DecisionTreeClassifier** y con los parámetros de profundidad (**max\_deph=**) que vamos a iterar y posteriormente crear el **AdaBoostClassifier** el cual le especificaremos en el parámetro  **estimator** el modelo de arbol que definimos previamente.

**Ej parámetro profundidad de árbol.**

Como en el caso del hiperparametro de cantidad de arboles, empezamos creando la función que nos va ayudar a evaluar el modelo, podemos utilizar la creada anteriormente, luego creamos la función que nos va ayudar a iterar los diferentes valores para el hiperparametro en cuestión, en este caso profundidad del árbol base, después el proceso completo con el conjunto de datos a evaluar y vemos el resultado en una grafica, en este caso diagrama de cajas.

Empezamos con la funcio que va a iterar los diferentes hiperparametros a evaluar del modelo.

from sklearn.tree import DecisionTreeClassifier

#Se crea funcion donde iteramos el modelo con diferentes profundidad del árbol

def get\_models():

models = dict() #Se almacen los valores en un dic

# explorar profundidad de 1 a 10

for i in range(1,11):

base = DecisionTreeClassifier(max\_depth=i)

models[str(i)] = AdaBoostClassifier(algorithm="SAMME", estimator=base )

return models

Importamos la librería del modelo base **DecisionTreeClassifier** y procedemos a crear la función.

* + - Se define el diccionario donde se va aguardar como titulo el hiperparametro a evaluar y como valor el modelo con dicho hiperparametro ej. **models{“1”: AdaBoostClassifier(algorithm="SAMME", estimator=1 )}** y asi sucesivamente por cada iteracion que se haga.
    - Se crea el bucle que va a iterar en el rango de 1 a 10
    - Se define el modelo base donde le pasamos el parámetro **max\_depth=i** que corresponde a la profundidad del árbol base
    - Se define el modelo **AdaBoost** donde lo vamos a probar con el modelo base definido previamente y sus diferentes profundidades y por ultimo la función devuelve el diccionario con los diferentes hiperparametros y el modelo para cada valor.

Evaluando el modelo con el conjunto de datos

#Se ejecutan las diferentes funciones creada para evaluar los diferentes parametros

models = get\_models()

# evaluar los modelos

results, names = list(), list()

for name, model in models.items():

# evaluar el modelo

scores = evaluate\_model(model, X, y)

#Se almacena el resultado

results.append(scores)

names.append(name)

#Se muestra el resultado de cada iteracion

print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))

Despues de crear las respectivas funciones, se procede a evaluar el modelo con los diferentes hiperparametro establecidos en la función **get\_models()** y el conjunto de datos y la variable objetivo.

* + - Se define la variable que va a contener el diccionario con los modelos y los diferentes hiperparametros a evaluar que en este caso es **max\_depth (**profundidad del árbol base**)**
    - Se crean 2 listas que van almacenar el resultado del accuracy y el nombre del modelo que se utilizo en cada iteracion.
    - Se crea el blucle con los diferentes modelos, se itera por nombre y modelo
    - Se evaluar el modelo en el que se esta iterando y se guardan los resultados en las listas definidas antes de iniciar el bucle. Por ultimo se imprime el resultado de cada iteracion, se muestra el nombre (valor del hiperparametro a evaluar), el promedio de scoring y la desviación estándar.

Se procede a visualizar el resultado del scoring vs cada hiperparametro que se itero

#Se grafica el resultado de las iteraciones

pyplot.boxplot(results, labels=names, showmeans=True)

pyplot.show()

Esta grafica nos muestra un diagrama de caja para cada valor por el cual se itero y cual fue su rango de scoring obtenido.

**Interpretacio:** Como en la grafica del hiperparametro anterior, escogemos el valor que tenga menos varianza.

* + **Tasa de aprendizaje:**

Corresponde al peso aplicado a cada clasificador (modelo) en cada iteracion de refuerzo. La tasa de aprendizaje esta controlada por el parámetro **learning\_rate** (por defecto es 1,0 o contribución completa).

Debemos de tener en cuenta que mas arboles requieren una tasa de aprendizaje menor, menos arboles pueden requerir una mayor tasa de aprendizaje y es común utilizar valores entre 0 y 1 y, a veces valores muy pequeños para evitar el sobre ajuste.

Como en los casos de los hiperparametros anteriores volvemos a modificar la función **get\_models** que es la función que nos ayuda ajustar el modelo con el hiperparametro que estemos evaluando, **de resto se siguen los mismos pasos que los ejemplos anteriores.** Se crea la variable que almacena el diccionario con los modelos de la función **get\_models >** se ejecuta el bucle para que itere los modelos > guarde el scoring y el nombre > imprime el resultado y por ultimo mostramos la grafica y escogemos el valor que menos varianza tenga.

**Ej función modelo con tasa de aprendizaje:**

#Se crea funcion para iterar el hiperparametro a evaluar

def get\_models():

models = dict()

# tasa de aprendizaje de 0.1 a 2 con incrementos de 0.1

for i in arange(0.1,2.1,0.1):

key = '%.1f' % i #Para que i quede con 1 decimal

#Se crea el modelo

models[key] = AdaBoostClassifier(algorithm='SAMME', learning\_rate=i)

return models

Se crea función que va ayudar a crear un modelo para cada iteracion en el parámetro de tasa de aprendizaje **learning\_rate** y los hacemos en un rango de 0.1 a 2 con la función **range()** de **numpy**

Despues de ejecutar el bucle for tal cual como en los casos anteriores, se procede a graficar el resultado

#Se grafica el resultado de las diferentes iteraciones

pyplot.boxplot(results, labels=names, showmeans=True)

#Pone las etiquetas del eje X en una rotacion de 45 grados

pyplot.xticks(rotation=45)

pyplot.show()

Y en este caso Tambien seleccionamos el valor que tenga menos varianza.

------------------------------------------------------------------------------------------

* + **Búsqueda de hiperparametro con GridSearchCV**

Otra manera de realizar la búsqueda del valor de hiperparametros que mejos se ajuste al modelo y conjunto de datos es por medio de la función **GridSearchCV** la cual ayuda a iterar entre los diferentes valores que le pasemos y podemos ver con la propiedad **best\_score\_** cual fue el mejor scoring que se obtuvo con **best\_params\_** cuales fueron los parámetros que se obtuvo el mejor scoring

**Ej GridSearchCV:**

from sklearn.model\_selection import GridSearchCV

# definir AdaBoost con hiperparámetros por defecto

model = AdaBoostClassifier(algorithm="SAMME")

#definir valores para n\_stimators y learning\_rate en un diccionario

grid = dict()

grid['n\_estimators'] = [10,50,100,500,600]

grid['learning\_rate'] = [0.0001,0.001,0.01, 1.0]

#Se configura Kfold para realzar validacion cruzada

cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=1)

# definir el procedimiento de búsqueda y ejecutarlo en el dataset

grid\_search = GridSearchCV(estimator=model, param\_grid=grid, n\_jobs=-1, cv=cv, scoring="accuracy")

#Se guarda el resultado de la ejecución con los datos

grid\_result = grid\_search.fit(X,y)

print("Mejor combinación: %f usando %s" % (grid\_result.best\_score\_, grid\_result.best\_params\_))

Definimos el modelo de **AdaBoost** al cual le queremos buscar los hiperparametros

* + - Ceamos un diccionario donde le vamos a pasar como llave el hiperparametro a evaluar y en el valor los valores que queremos iterar
    - Configuramos el **Kfold** el cual nos ayuda a realizar la validación cruzada de los hiperparametros
    - Utilizamos la función GridSearchCV que nos ayudara a buscar el mejor resultado con los parámetros que le pasemos en el diccionario definido y el cual va en el valor de **param\_grid= ,** en **estimator=** va el modelo de **AdaBoost,** en **cv=** va el Kfold configurado, y por ultimo en **scoring=** la metrica que se va a utilizar para evaluar el modelo, en este caso se utiliza el **“Accuracy”**
    - Por ultimo se guarda el resultado de la ejecución con los datos y se imprime la mejor metrica obtenidad con **best\_score\_** y cuales fueron los parámetros con los que se obtuvo esa metrica con **best\_params\_**

--------------------------------------------------------------------------------------------------------------

Cambio de modelo

--------------------------------------------------------------------------------------------------------------

* [**Modelo XGBoost**](https://xgboost.readthedocs.io/en/stable/python/index.html)

**XGBoost** (eXtreme Gradient Boosting) es un algoritmo de aprendizaje basado en el principio de **Boosting,** el cual combina multiples modelos débiles (generalemente arboles de decision) para asi poder crear un modelo fuerte.

Este modelo puede ser ideal cuando:

* + Se requiere alto rendimiento predictivo
  + Hay relaciones no lineales complejas entre las variables
  + Se tienen variables tanto numéricas como categóricas
  + Se tiene muchos datos y necesitamos velocidad de entrenamiento
  + Funciona perfectamente para clasificación binaria y multiclase pero se le debe de especificar en el parámetro **objective= 'multi:softprob'** para clasificación multiclase o ‘**binary:logistic'** para clasificación binaria.
  + **XGBoost** es robusto a **outliers** porque los arboles de decisión no se ven muy afectados. Sin embargo, muchos **outliers** pueden aun generar sobreajuste.
  + A partir de la versión 1.3 **XGBoost** permite variables categóricas directamente si se le especifica como tipo **category** en un DF y se le activa el parámetro **enable\_categorical=True.**
  + Los valores nuelos los manjea internamente, pero tambien se pueden imputar para mejorar el rendimiento.
  + **XGBoost** no maneja fechas directamente, se deben de extaer como el año, mes, dia y diferencia de días.
  + Utiliza técnicas como regularización L1 (Lasso) y L2 (Ridge) las cuales hacen que el modelo no se complique innecesariamente y tambien maneja una técnica de poda, lo cual quiere decir que elimina ramas de los arboles que el modelo considere innecesarias.

**Ej básico de XGBoost para clasificacion**

#Se importan libreririas para aplicar XGBoost

from numpy import mean

from numpy import std

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import RepeatedStratifiedKFold

from xgboost import XGBClassifier

#Se define la varible con el modelo

model = XGBClassifier()

#Se realiza la configuracion de Kfold para validacion cruzada

cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=1)

#Se realiza la validacion cruzada y se utiliza la metrica Accuracy

n\_scores = cross\_val\_score(model, X, y, scoring='accuracy', cv=cv, n\_jobs=-1)

#Se muestra el resultado de la validacion cruzada

print('Accuracy: %.3f (%.3f)' % (mean(n\_scores), std(n\_scores)))

Para este ejemplo basico, empezamos importante las librerías requeridas para aplicar y evaluar el modelo **XGBoost para clasificacion.**

* + De define la variable con el modelo **XGBClassifier()**
  + Se realiza la configuración la validación cruzada y repetida, dode realizara 10 iteraciones y dividiendo asi el DF en diferentes partes si repetir combinación y repetirá el proceso 3 veces.
    - Se utiliza **RepeatedStratifiedKFold** cuando se tienen clases desbalanceadas
  + Se realiza la evaluación del modelo y el conjunto de datos con las iteración definidas en el modelo Kfold por medio de la metrica “Accuracy”
  + Por ultimo se muestra el promedio de score de las diferentes iteraciones y el desviación estándar.
* [**XGBoost para regresión (XGBRegressor):**](https://xgboost.readthedocs.io/en/stable/python/python_api.html#xgboost.XGBRegressor)

Este modelo permite predecir valores predecir valores continuos y particularmente útil en:

* + Problemas de regresión donde la relación entre variables no es lineal.
  + Cuando se requere un modelo de alto rendimiento y preciso.
  + Tambien permite valores nulos como la versión de clasificación y no es propenso a verse afectado por outliers. El la variables categóricas y de tipo fecha aplica igual que la versión de clasificación.

**Ej básico XGBoost para regresión.**

#Se importan librerias para aplicar XGBoost para regresion

from sklearn.model\_selection import RepeatedKFold

from xgboost import XGBRegressor

#Se define el modelo

model = XGBRegressor()

#Se reliza la configuracion para realizar validacion cruzada

cv = RepeatedKFold(n\_splits=10, n\_repeats=3, random\_state=1)

#Se aplica la validacion cruzada y se evalua con la metrica neg\_mean\_absolute\_error

n\_scores = cross\_val\_score(model, X, y, scoring='neg\_mean\_absolute\_error', cv=cv, n\_jobs=-1)

#Se muestra el resultado

print('MAE: %.3f (%.3f)' % (mean(n\_scores), std(n\_scores)))

Como en todos los casos, emepezamos imporntando las librerías requeridas para crear y evaluar el modelo para regresión XGBRegressor.

* + Se crea y almacena el modelo básico
  + Se realiza la configuración la validación cruzada y repetida, dode realizara 10 iteraciones y dividiendo asi el DF en diferentes partes si repetir combinación y repetirá el proceso 3 veces.
    - Se utiliza RepeatedKFold para problemas de regresión clases balanceadas
  + Se realiza la evaluación del modelo y el conjunto de datos con las iteración definidas en el modelo Kfold por medio de la metrica “neg\_mean\_absolute\_error”
  + Por ultimo se muestra el promedio de score de las diferentes iteraciones y el desviación estándar.

**Hiperparametros de XGBoost**

1. **Numero de arboles:** 
   1. Los arboles de decisión se agregan al modelo secuencialmente en un esfuerzo por corregir y mejorar las predicciones por los arboles anteriores. Por lo tanto, entre mas arboles haya, mejorara la prediccion y este se define en el parámetro **n\_estimators=** de **XGBClassifier**
   2. Si bien entre mas arboles mejor, se debe de tener cuidado porque llega a un punto donde se puede incurrir en sobre ajuste y perder rendimiento del modelo. Y el valor por defecto es 100

Como en el caso de la búsqueda de hiperparametros de **AdaBoost,** se crea una función para evaluar las diferentes iteraciones del modelo, se crea la función que va a crear el modelo en cuestión con el hiperparametro a evaluar y por ultimo mostrar el resultado.

**Ej búsqueda parámetro n\_estimators=**

#Se crea funcion para config el modelo

def get\_models():

"""Esta funcion crea un diccionario con los diferentes modelos

que se definan con loa parametros de trees"""

models = dict()

trees = [10, 50, 100, 250, 500, 1000]

for n in trees:

models[str(n)] = XGBClassifier(n\_estimators=n)

return models

Como se menciono, emepezamos creando la función que creara los diferentes modelos con el hiperparametro que estemos evaluando, en este caso **n\_estimators** o numero de arboles.

* + Funcion en la cual empezamos definiendo un diccionario que almacenara los diferentes modelos para cada valor de la iteración en **tree.** Donde la llave del diccionario será el valor del parámetro que este ejecutando y el valor del diccionario será el modelo creado con el parámetro **n\_estimators** del mismo valor de la llave.
  + Se definen los valores a iterar en el hiperparametro
  + Se ejecuta un bucle for donde recorrerá cada valor previamente definido y creara un modelo para cada valor iterado.

Despues de crear la función se itera y evaluar cada modelo con el valor definido en la función con el conjunto de datos.

#Se define el modelo

models = get\_models()

#Se crean listas para guardar el resultado de la iteracion

results, names = list(), list()

#Se corre bucle para evaluar los diferentes modelos

for name, model in models.items():

#Se evaluar el modelo que se este ejecutando

scores = evaluate\_model(model, X, y)

results.append(scores)

names.append(name)

#Se imprime el resultado

print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))

* + Se empieza definiendo la variable (diccionario) que va a contener los modelos que están definidos en la función **get\_models()**
  + Se crean las listas donde se van almacenar los resultados de cada iteración para cada modelo y el nombre de cada modelo para luego graficar el resultado
  + Se ejecuta un bucle con el diccionario de los modelos
  + Se realiza evaluacion de cada modelo que se itere, se utiliza la función creada (Es la misma que se utilizo para evaluar los modelos de **AdaBoost**), la cual realiza una validación cruzada utilizando la metrica “**Accuracy”**
  + Se guaran los resultados del modelo en cada iteración y se imprime el modelo del “Accuracy” y la desviación estándar de cada iteración.

Despues de tener los resultados se procede a graficar utilizando un diagrama de cajas

#Se grafican los diferentes resultados obtenidos

pyplot.boxplot(results, labels=names, showmeans=True)

pyplot.show()

**Se escoge el valor que menos varianza tenga en el diagrama**

1. **Profundidad del árbol:**

La profundidad del árbol controla que tan especializado esta cada árbol en train. Debemos de tener presente que entre mas arboles se puede incurrir en un sobre ajuste

* + Se prefieren arboles que no sean demasiado superficiales y generales ni demasiado profundos y especializados
  + La profundidad del árbol árbol se controla mediante el parámetro **max\_depth=** que por defecto es 6.

Se modifica la función que crea el modelo pero esta vez para evaluar el hiperparametro **max\_depth**

#Se crea funcion del modelo

def get\_models():

models = dict()

# Explorar entre 1 y 10 en profundidad de cada arbol

for i in range(1,11):

models[str(i)] = XGBClassifier(max\_depth=i)

return models

Se modifica la función que crea el diccionario con los modelos **XGBClassifier** para cada iteración con el valor del hiperparametri, en este caso con **max\_depth** que se va a explorar entre 1 y 10 en profundidad de cada árbol

Se itera en cada modelo con el conjunto de datos y se evaluar el resultado con la función que realiza la validación cruzada.

#Se define el diccionario con los modelos

models = get\_models()

#Se crean listas para almacenar los resultados

results, names = list(), list()

#Se itera por cada modelo que este en el diccionario

for name, model in models.items():

#Se evalua el modelo de cada iteracion

scores = evaluate\_model(model, X, y)

#Se guardan los resultados de cada iteracion

results.append(scores)

names.append(name)

#Se imprime el promedio y desviacion de cada modelo

print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))

Como en todos los casos visto anteriormente, se realiza el mismo proceso, de creación del diccionario con cada modelo > se crean las listar que van almacenar cada resultado > se itera por cada modelo que este en el diccionario previamente definido > > se evaluar cada modelo iterado por medio de validación cruzada > se guardan los resultados de cada iteración en cada modelo y se muestra el resultado.

Se grafica el resultado de la ejecución anterior donde se muestra las métricas obtenidas para cada modelo iterado

# Se grafica el resultado de las iteraciones

pyplot.boxplot(results, labels=names, showmeans=True)

pyplot.show()

Como en los casos anteriores se escoge el valor para el hiperparametro **max\_depth** que menos varianza tenga

1. **Tasa de aprendizaje (eta=)**

La tasa de aprendizaje controla en el modelo la cantidad de contribución que cada modelo tiene en la prediccion del conjunto. Y se controla por medio del parámetro **eta=** que por defecto es 0.3

* + Tener una tasa mas pequeña puede requerir mas arboles de decisión en el conjunto y viceversa.

**Ej parámetro tasa de aprendizaje (eta=)**

Modificamos la función que crea el diccionario con los modelos que contienen cada valor del hiperparametro **eta= (**Tasa de aprendizaje**)**

#Se crea funcion del modelo

def get\_models():

models = dict()

# explorar estas tasas de aprendizaje [0.0001, 0.001, 0.01, 0.1, 1.0]

rates = [0.0001, 0.001, 0.01, 0.1, 1.0]

#Se itera cada valor del parametro a evaluar

for r in rates:

#Se crea la llave del diccionario con 4 decimales

key = '%.4f' % r

#Se crea el modelo con cada valor recorriedo

models[key] = XGBClassifier(eta=r)

return models

Despues de crear la función se siguen los mismo pasos del hiperparametro anterior > se define el diccionario con los modelo > se recorre cada modelo en un bucle > se evalua y se almacenan los resultados > se imprime el promedio de cada iteración y la desviación

Por ultimo igual que los casos anteriores, se grafica el resultado y se escoge le valor que menor varianza muestre en la grafica de cajas.

* **Light Gradient Boosted Machine (LightGBM)**

LightGBM es un framework de boosting desarrollado por Microsoft que utiliza arboles de decisión como base y es mas rápido y eficiente en memoria que XGBoost y es particularmente útil para:

* + Problemas con grandes volúmenes de datos
  + Cuando se necesita rapidez en entrenamiento
  + Cuando no se tienen variables categóricas codificadas
  + Sirve para para problemas de clasificación binaria, multiclase y regresión.

Funcionamiento para clasificación:

* + Utiliza la estrategia “one vs all” internamente – Problemas de multi clase
  + Requiere especificar el parámetro “num\_class”
  + Es relativamente robusto a outliers (Outliers extremos pueden afectar el modelo)
  + Maneja variables categóricas en las variables independientes (Se debe de especificar con el parámetro “**categorical\_feature**”)
  + Maneja valores nulos
  + No maneja fechas directamente

**Ej básico funcionamiento LightGBM**

# Se crea el modelo

model = LGBMClassifier()

#Se configura la validacion cruzada

cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=1)

#Se realiza la validacion cruzada

n\_scores = cross\_val\_score(model, X, y, scoring='accuracy', cv=cv)

#Se muestra el resultado

print('Accuracy: %.3f (%.3f)' % (mean(n\_scores), std(n\_scores)))

Para este problema de clasificacion, empezamos definiendo el modelo con la función **LGBMClassifier()** para este ejemplo básico no se le especifica ningún hiperparametro.

* + Se realiza la configuración de la validación cruzada, donde va a realizar 10 particiones diferentes y va a repetir el proceso 3 veces
  + Se realiza la validación cruzada del modelo, teniendo presente el conjutno de datos y la metrica de Scoring “Accuracy” con la configuración de Kfold
  + Por ultimo se muestra el resultado, el promedio del Accuracy de todas las iteraciones, y el promedio de la desviación estándar.