KIBERNUM
IT Academy

Módulo 5 – Aprendizaje de Máquina Supervisado

Algoritmos Boosting

Especialización en Ciencia de Datos

Contenido

- Modelos de Ensamble.
- Modelos de Boosting.
- Adaboost.
- Gradient Boosting.
- XGBoost.

KIBERNUM

Modelos de Ensamble

KIBERNUM

Modelos de Ensamble

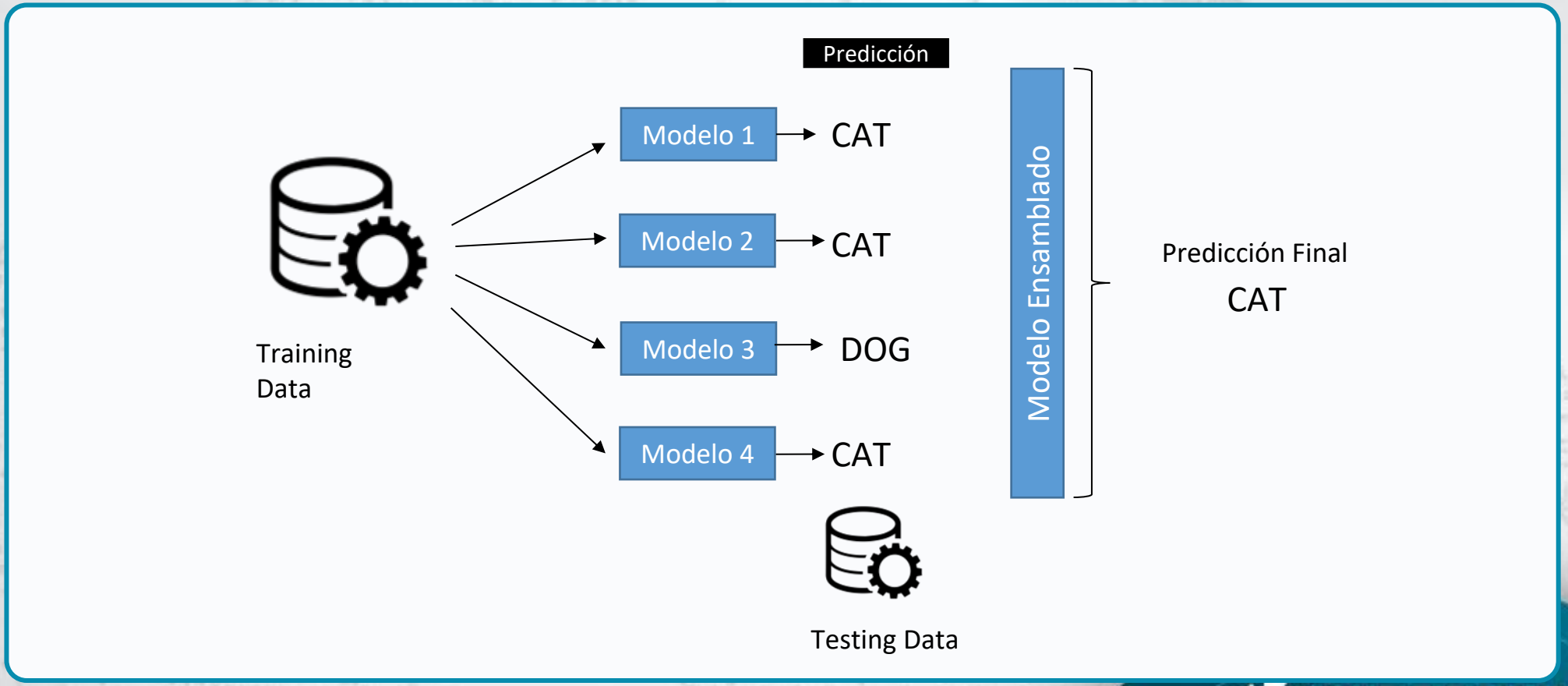
Los modelos de ensamble son técnicas avanzadas de aprendizaje automático que combinan varios modelos más simples en un modelo más fuerte y preciso. En lugar de depender de un solo modelo para hacer una predicción, los modelos de ensamble utilizan la predicción de varios modelos más simples para producir una salida final. Hay varios tipos de modelos de ensamble, pero los más comunes son los siguientes:

- **Bagging:** Es un modelo de ensamble que combina varios modelos más simples, como árboles de decisión, y luego promedia sus resultados para obtener una salida final más precisa. Cada modelo se entrena en un subconjunto aleatorio de datos y luego se combina para formar un modelo final.

- **Boosting:** Es un modelo de ensamble que combina varios modelos más simples, como árboles de decisión, mediante la construcción iterativa de modelos más fuertes y precisos. En cada iteración, el modelo se ajusta para corregir los errores cometidos por los modelos anteriores.

Bagging

Bagging (Bootstrap Aggregating) es una técnica de aprendizaje automático utilizada para mejorar la precisión y la estabilidad de los modelos predictivos. El bagging se utiliza principalmente en conjuntos de datos grandes y complejos para reducir la varianza y el sobreajuste.



Bagging

El proceso de bagging implica la construcción de múltiples modelos más simples, como árboles de decisión por ejemplo, y luego combinar sus resultados para obtener una salida final más precisa. En lugar de entrenar un solo modelo en todo el conjunto de datos, se entrena cada modelo en un subconjunto aleatorio de los datos (con reemplazo). Esto crea múltiples modelos que tienen una alta varianza, pero baja sesgo.

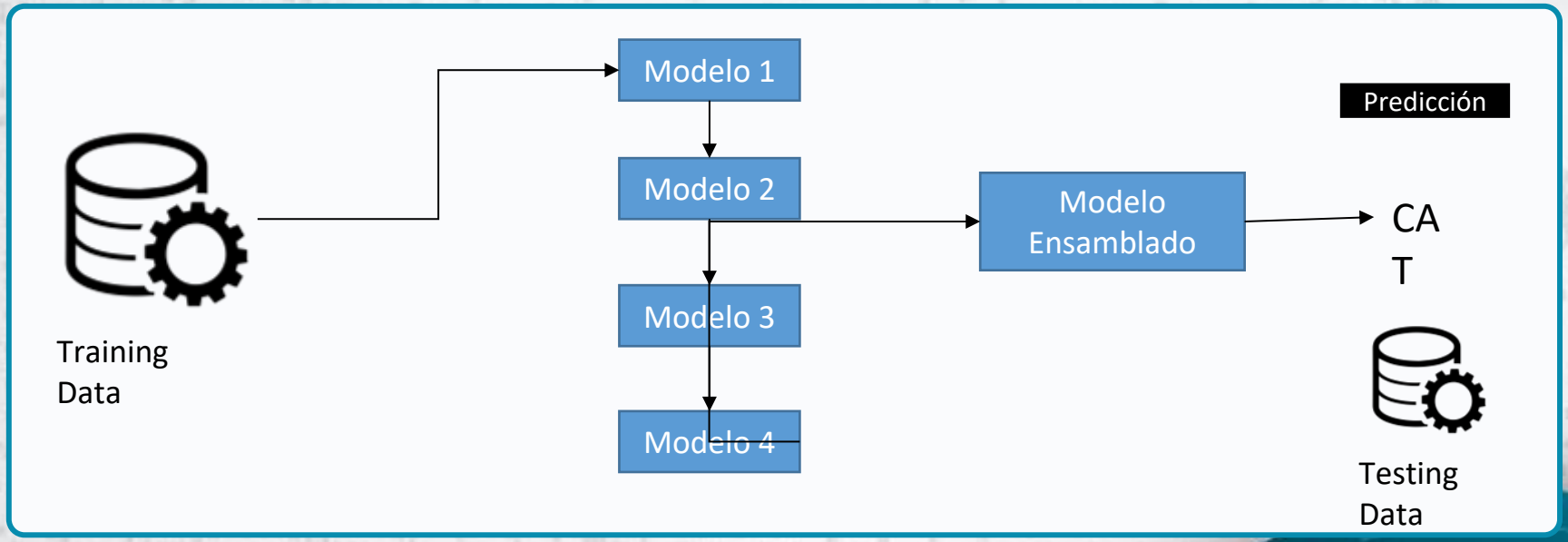
- Luego, los resultados de cada modelo se combinan mediante la promediación o votación para producir una salida final más estable y precisa. En resumen, el proceso de bagging puede mejorar la precisión del modelo y reducir la probabilidad de sobreajuste.

- Una de las implementaciones más populares de bagging es el Random Forest, que utiliza un conjunto de árboles de decisión aleatorios en lugar de un solo árbol de decisión. Random Forest utiliza el proceso de bagging para reducir la varianza de cada árbol de decisión y producir una salida final más precisa.

- En general, el bagging es una técnica útil de aprendizaje automático que se utiliza para mejorar la precisión y la estabilidad de los modelos predictivos en conjuntos de datos grandes y complejos.

¿Qué es un algoritmo de Boosting?

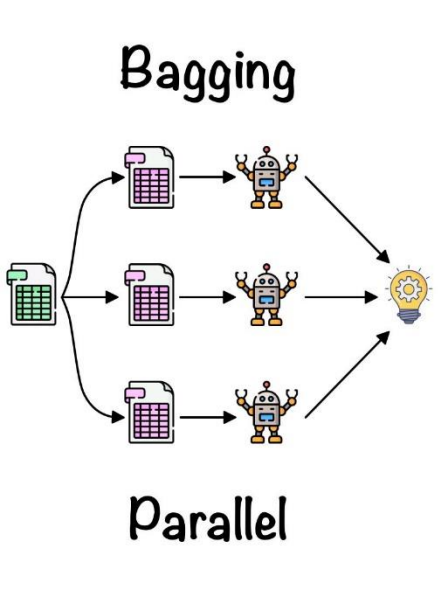
El algoritmo de boosting es una técnica de aprendizaje automático que combina varios modelos más débiles en un modelo más fuerte y preciso. En el proceso de boosting, se ajustan iterativamente una serie de modelos más débiles (llamados "weak learners" o "weak predictors") en el conjunto de datos de entrenamiento. En cada iteración, se presta mayor atención a los casos que se clasificaron incorrectamente en las iteraciones anteriores.

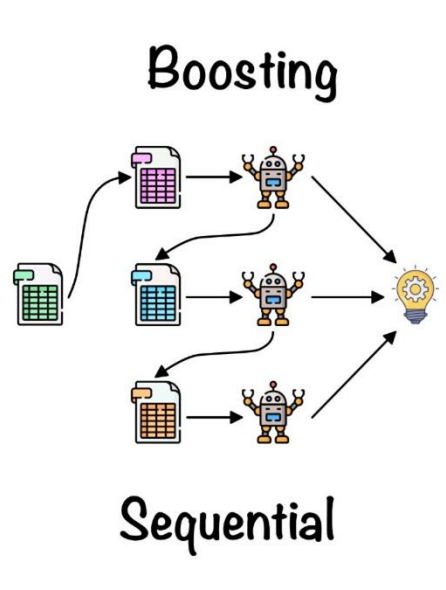


KIBERNUM

Diferencia entre Bagging y Boosting

Bagging	Boosting
Método de ensamble paralelo de modelos.	Métodos de ensamble secuencial de modelos.
Los aprendices débiles son independientes entre sí.	Los aprendices débiles dependen uno de otro.
Disminuyen la varianza del modelo.	Disminuyen el sesgo del modelo.
Usado para disminuir el sobreajuste de un modelo.	Usado para aumentar modelos subajustados.
Data sampling se hace aleatoriamente con reemplazo.	Data sampling es hecho basado en peso asignado a cada registro con reemplazo.





AdaBoost

KIBERNUM

¿Qué es un algoritmo de Boosting?

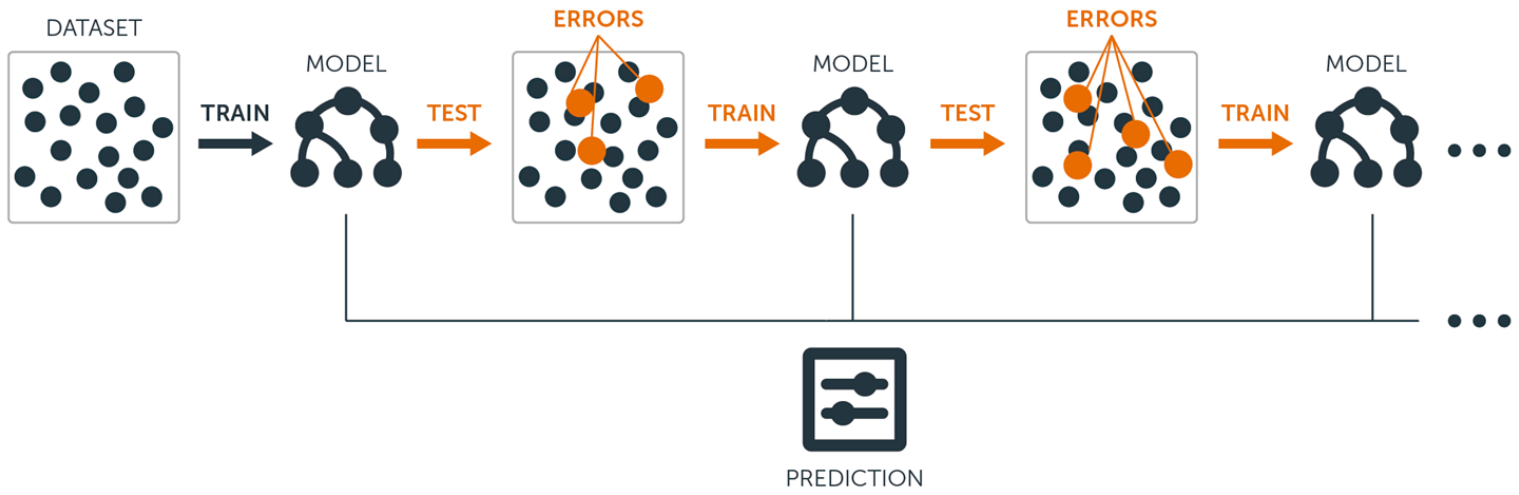
El algoritmo de boosting mejora la precisión del modelo combinando varios modelos más débiles y haciéndolos más robustos, lo que resulta en una mejor generalización y una mayor capacidad de predicción en nuevos datos. Uno de los algoritmos de boosting más populares es el Gradient Boosting, que se utiliza en una amplia gama de aplicaciones de aprendizaje automático, incluyendo clasificación, regresión y detección de anomalías.

Algunos algoritmos de boosting son los siguientes:

- AdaBoost.
- Gradient Boosting.
- XGBoost.
- LightGBM.

¿Cómo funciona Boosting?

Dada la definición anterior, el algoritmo funciona de una manera que ajusta modelos de aprendizaje débiles (aprendices) en versiones ponderadas de los datos, donde se le da más peso a la observación mal clasificada en rondas anteriores del algoritmo (codicioso). Los aprendices pueden ser cualquier algoritmo de clasificación o regresión. Por definición, son muy resistentes al sobreajuste, porque el objetivo de esta técnica es obtener el mejor rendimiento en una "imagen" extendida de los datos (gran poder de generalización) como es un conjunto de "imágenes" más pequeñas (aprendices débiles).



Dataset: set de datos
Train: entrenamiento
Model: modelo
Prediction: predicción

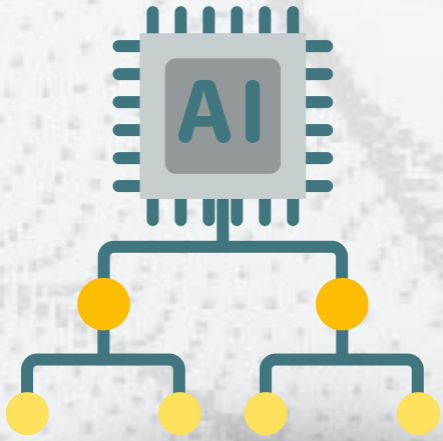
AdaBoost

La base de este algoritmo es el núcleo principal de Boosting: dar más peso a las observaciones mal clasificadas. En particular, AdaBoost significa Adaptive Boosting, lo que significa que el meta-aprendiz se adapta en función de los resultados de los clasificadores débiles, dando más peso a las observaciones mal clasificadas del último aprendiz débil.

Set de Entrenamiento

Modelo 1

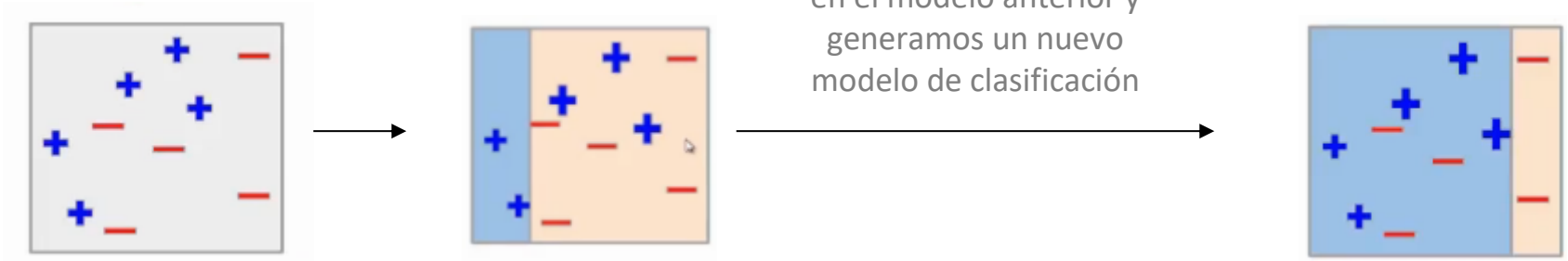
En este ejemplo, el primer clasificador débil es capaz de clasificar dos elementos en la zona azul.



KIBERNUM

AdaBoost

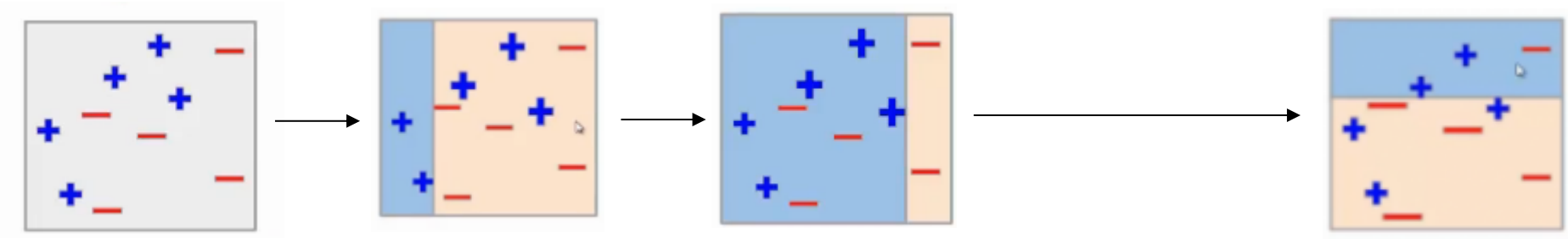
Ahora, se ajustan los pesos de los puntos mal clasificados, dándole un peso mayor para la siguiente clasificación. Dar un mayor peso a estos puntos hará que el modelo se enfoque más en estos valores.



En este segundo modelo, el límite de decisión se desplazó a la izquierda, para así clasificar correctamente los puntos de mayor peso. Aún no es un modelo perfecto, pues aún quedan instancias mal clasificadas.

AdaBoost

Ahora le damos más peso a las dos mediciones rojas que en el modelo 2 fueron mal clasificadas. Con esto, el modelo 3 genera un nuevo límite de decisión como el que se muestra a continuación.



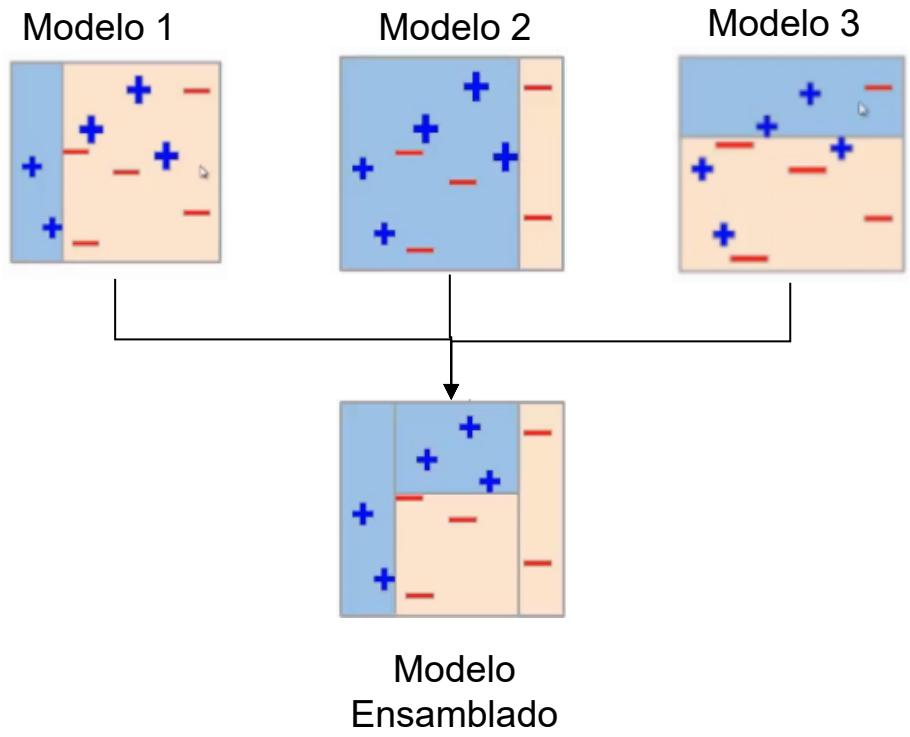
Este nuevo modelo nuevamente hace predicciones incorrectas. Se podría decir que todos los modelos de forma individual no son lo suficientemente fuertes para clasificar los puntos correctamente, por eso son llamados predictores débiles.

AdaBoost

Ahora debemos generar un modelo agregado de los 3 modelos anteriores. Una forma podría ser tomando un promedio ponderado de los modelos débiles para generar el modelo ensamblado. Después de múltiples iteraciones de los pesos, se llega al modelo final, que sería capaz de clasificar los puntos de forma correcta. A este clasificador final se le denomina clasificador fuerte.

$$F(x) = sign(\sum_{m=1}^{M} \theta_m f_m(x))$$

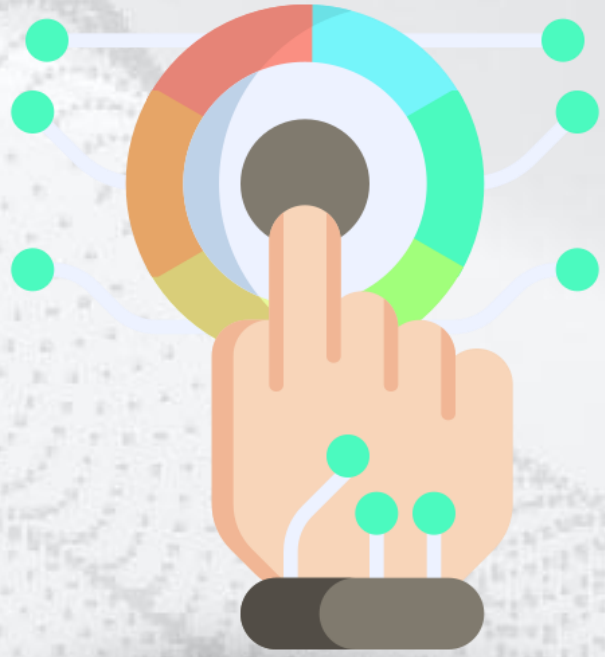
En donde f corresponde al clasificador débil y theta corresponde al peso asociado a dicho clasificador.



Implementación en Python

Revisemos los pasos :

Paso 1	Construir un modelo y hacer predicciones
Paso 2	Asignar mayores pesos a los puntos mal clasificados
Paso 3	Construir el siguiente modelo
Paso 4	Repetir pasos 2 y 3
Paso 5	Hacer un modelo usando los pesos de cada modelo individual



KIBERNUM

Implementación AdaBoost en Python

Revisemos el caso Titanic con AdaBoost. Vamos directamente al modelo.

Importamos clasificador desde el módulo ensamble de la librería sklearn

Instanciamos clasificador AdaBoost con hiperparámetros por defecto

Entrenamos clasificador

Score en set de entrenamiento

Score en set de test

Modelamiento con AdaBoost

```
from sklearn.ensemble import AdaBoostClassifier
```

```
clf = AdaBoostClassifier(random_state=0, algorithm='SAMME')
```

```
clf.fit(X_train,y_train)
```

AdaBoostClassifier

```
AdaBoostClassifier(algorithm='SAMME', random_state=0)
```

```
clf.score(X_train,y_train)
```

```
0.7921348314606742
```

```
# score en set de test
clf.score(X_test,y_test)
```

```
0.7932960893854749
```

Hiperparámetros AdaBoost

Los principales hiperparámetros de AdaBoost son los siguiente:

1. base_estimator: el modelo a ensamblar, por defecto es un árbol de decisión.

1. n_estimators: número de modelos que se entrenarán.

1. learning_rate: reduce la contribución de cada clasificador por este valor.

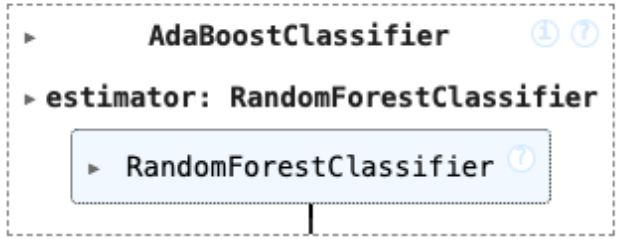
1. random_state: La semilla de números aleatorios, de modo que se generen los mismos números aleatorios cada vez.

Hiperparámetros

```
from sklearn.ensemble import RandomForestClassifier
```

```
clf = AdaBoostClassifier(random_state = 0,
                         estimator = RandomForestClassifier(),
                         n_estimators = 100,
                         learning_rate = 0.01,
                         algorithm='SAMME')
```

```
clf.fit(X_train,y_train)
```




```
# score en set de entrenamiento
clf.score(X_train,y_train)
```

```
0.9775280898876404
```

```
# score en set de test
clf.score(X_test,y_test)
```

```
0.8100558659217877
```

KIBERNUM

Ventajas

1. AdaBoost es fácil de implementar y no requiere mucho ajuste de hiperparámetros.

2. Es un algoritmo rápido y eficiente en términos de tiempo de entrenamiento y predicción.

3. Es altamente preciso en la mayoría de los casos, especialmente cuando se utiliza con modelos débiles (por ejemplo, árboles de decisión simples – “Stumps”).

4. Es resistente al sobreajuste, lo que significa que puede generalizar bien a datos no vistos.

Desventajas

1. AdaBoost es sensible al ruido en los datos de entrenamiento. Los valores atípicos y los errores pueden afectar negativamente la precisión del modelo.

2. AdaBoost puede ser propenso al sobreajuste si se utilizan modelos demasiado complejos o si se entrena durante demasiadas iteraciones.

3. AdaBoost puede ser computacionalmente costoso si se utilizan modelos complejos o si se entrena con grandes conjuntos de datos.

Gradient Boosting

KIBERNUM

Gradient Boosting

- Gradient Boosting es un algoritmo de aprendizaje automático utilizado para problemas de regresión y clasificación. Es una técnica de ensamblaje de modelos que utiliza varios modelos débiles (normalmente árboles de decisión) para construir un modelo predictivo más fuerte.

- En Gradient Boosting, los modelos se construyen de forma secuencial, en lugar de en paralelo, como en otros algoritmos de ensamblaje. En cada iteración, el modelo busca ajustar los errores del modelo anterior, de modo que el siguiente modelo se ajusta a los residuos del modelo anterior.

- El algoritmo utiliza una función de pérdida y un algoritmo de optimización, como el descenso del gradiente, para minimizar la función de pérdida y ajustar el modelo a los datos. El resultado final es un modelo predictivo que puede generalizar bien a datos no vistos y mejorar el rendimiento en comparación con un solo modelo.

- Gradient Boosting es ampliamente utilizado en competiciones de aprendizaje automático y es uno de los algoritmos de ensamblaje más populares debido a su capacidad para manejar datos de alta dimensionalidad y su capacidad para evitar el sobreajuste.

Gradient Boosting

Para ilustrar cómo funciona Gradient Boosting, tomemos el siguiente ejemplo regresivo, en donde a partir de datos de edad y ciudad, se quiere predecir el ingreso de un individuo.

Features | Variable objetivo

ID	Age	City	Income
1	32	A	51000
2	30	B	78000
3	21	A	20000
4	27	B	44000
5	36	B	89000
6	25	A	37000
7	47	A	56000
8	54	B	92000

KIBERNUM

Gradient Boosting

Lo primero que haremos será entrenar un modelo regresivo (Model 1) y obtendremos predicciones, para calcular el error residual en cada predicción.

ID	Age	City	Income	Model 1 Income
1	32	A	51000	53500
2	30	B	78000	61000
3	21	A	20000	28500
4	27	B	44000	61000
5	36	B	89000	90500
6	25	A	37000	28500
7	47	A	56000	53500
8	54	B	92000	90500

TARGET	PREDICTION	RESIDUAL
Income	Predictions	Error
51000	53500	-2500
78000	61000	17000
20000	28500	-8500
44000	61000	-17000
89000	90500	-1500
37000	28500	8500
56000	53500	2500
92000	90500	1500

Variable objetivo

Predicción Modelo 1

Error residual

KIBERNUM

Gradient Boosting

Ahora construiremos un modelo sobre los errores obtenidos y haremos predicciones, la idea es determinar si hay patrones ocultos en el error. Nótese que los features siguen siendo edad y ciudad, pero en este nuevo modelo, la variable objetivo será el error del modelo 1.

Features

ID	Age	City	Income	Model 1 Income
1	32	A	51000	53500
2	30	B	78000	61000
3	21	A	20000	28500
4	27	B	44000	61000
5	36	B	89000	90500
6	25	A	37000	28500
7	47	A	56000	53500
8	54	B	92000	90500

Variable objetivo

TARGET	PREDICTION	RESIDUAL
Error	Predictions	
-2500	-5500	
17000	8000	
-8500	-5500	
-17000	-4300	
-1500	8000	
8500	8000	
2500	-4300	
1500	-4300	

Error residual

Predicción del error residual

KIBERNUM

Gradient Boosting

Ahora actualizaremos las predicciones del modelo 1. Agregaremos la predicción del paso anterior y se lo agregaremos al modelo 1 para obtener el modelo 2.

Modelo 2 Ingreso = Modelo 1 Ingreso + Predicción error

De esta forma, el modelo 2 queda de la siguiente manera:

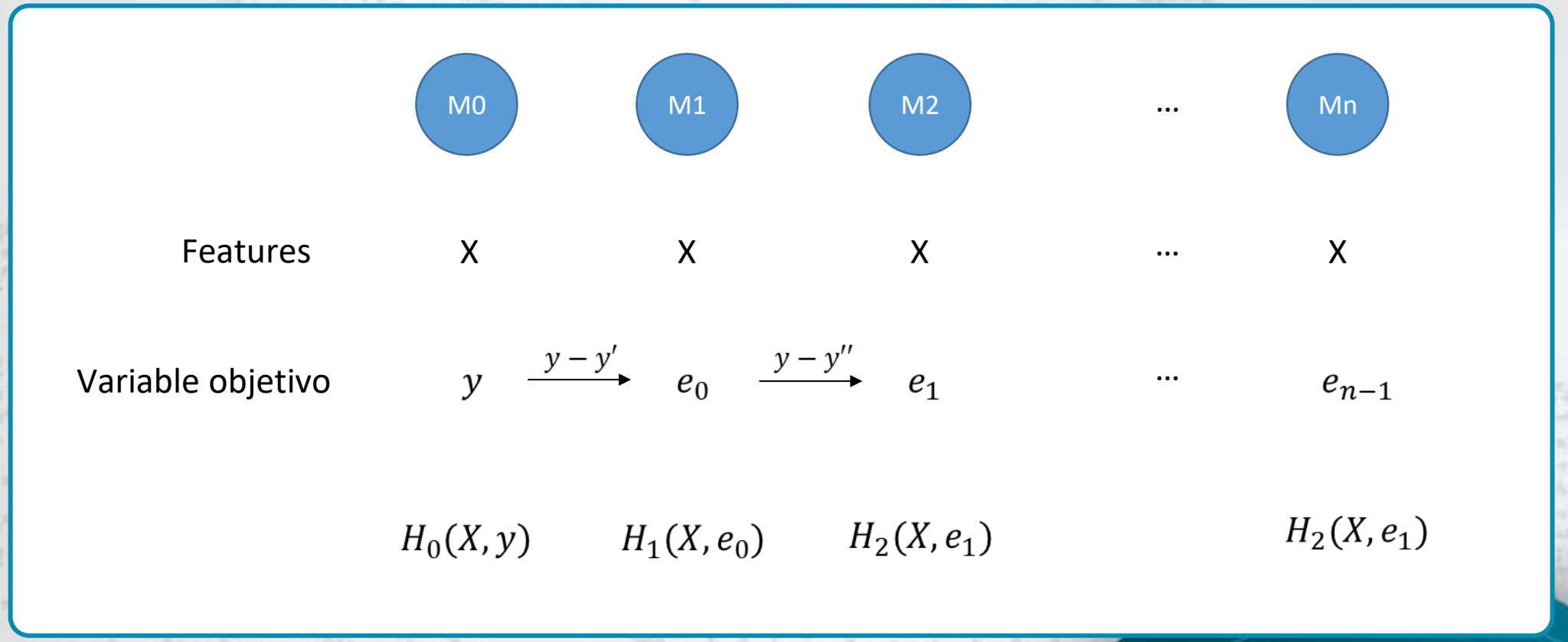
ID	Age	City	Income	Model 1 Income	Model 2 Income
1	32	A	51000	53500	48000
2	30	B	78000	61000	69000
3	21	A	20000	28500	23000
4	27	B	44000	61000	56700
5	36	B	89000	90500	98500
6	25	A	37000	28500	36500
7	47	A	56000	53500	49200
8	54	B	92000	90500	86200

TARGET	PREDICTION	RESIDUAL
Error	Predictions	
-2500	-5500	
17000	8000	
-8500	-5500	
-17000	-4300	
-1500	8000	
8500	8000	
2500	-4300	
1500	-4300	

KIBERNUM

Gradient Boosting

Finalmente, vamos repitiendo el procesos recalculando y modelando los nuevos errores obtenidos como variable objetivo. Esto lo repetiremos hasta que el error sea cero o bien hasta que se cumpla alguna condición de detención. Resumiendo el proceso brevemente:



KIBERNUM

Gradient Boosting

En cada iteración vamos modelando el error del modelo anterior. Como se puede observar, cada modelo está tratando de impulsar al performance del modelo anterior.

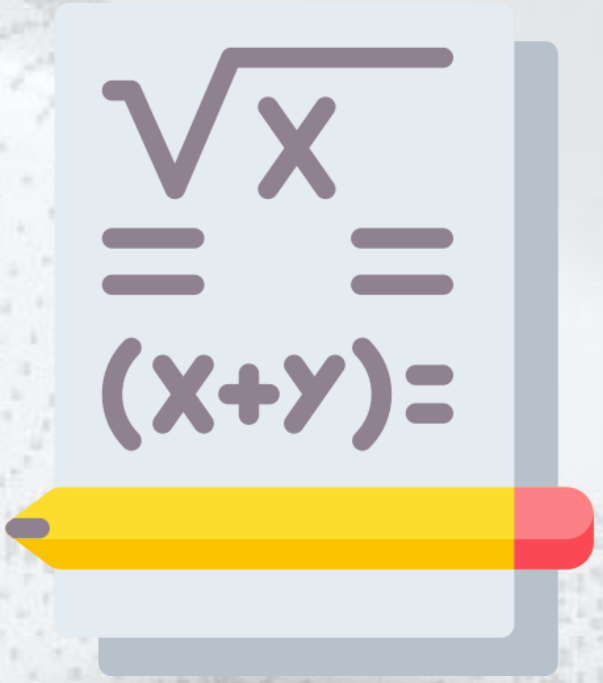
$$F_0(X) = H_0(X, y) + e_0$$

$$F_1(X) = F_0(X) + H_1(X, e_0) + e_1$$

$$F_2(X) = F_1(X) + H_2(X, e_1) + e_2$$

$$\vdots$$

$$F_n(X) = F_{n-1}(X) + H_n(X, e_{n-1}) + e_n$$



Gradient Boosting

Pero, ¿por qué usamos el término gradiente?

En lugar de agregar directamente estos modelos, los agregamos con peso o coeficiente, y el valor correcto de este coeficiente se decide mediante la técnica de aumento de gradiente. Una forma más general de escribir la ecuación es la siguiente:

$$F_{n+1}(X) = F_n(X) + \gamma_n H_n(X, e_n)$$

Por lo tanto, el modelo final contendrá los valores de γ_n que minimizan la respectiva función de pérdida.

KIBERNUM

Implementación en Python

Revisemos el caso Titanic con Gradient Boosting. Vamos directamente al modelo:

Importamos clasificador desde el módulo ensemble de la librería sklearn

Instanciamos clasificador AdaBoost con hiperparámetros por defecto

Entrenamos clasificador

Score en set de entrenamiento

Score en set de test

Modelamiento con GradientBoosting

```
from sklearn.ensemble import GradientBoostingClassifier
```

```
clf = GradientBoostingClassifier(random_state=0)
```

```
clf.fit(X_train,y_train)
```

```
GradientBoostingClassifier
GradientBoostingClassifier(random_state=0)
```

```
# score en set de entrenamiento
clf.score(X_train,y_train)
```

```
0.8778089887640449
```

```
# score en set de test
clf.score(X_test,y_test)
```

```
0.8268156424581006
```

KIBERNUM

Hiperparámetros Gradient Boosting

Los principales hiperparámetros de GB son los siguientes:

- **n_estimators**: Número de árboles que se construyen en el modelo.
- **learning_rate**: Tasa de aprendizaje del modelo.
- **max_depth**: Profundidad máxima de los árboles de decisión en el modelo.
- **subsample**: Proporción de muestras que se utilizan para entrenar cada árbol en el modelo.
- **loss**: Este parámetro controla la función de pérdida utilizada en el modelo.
- **random_state**: Semilla aleatoria utilizada para generar los números aleatorios en el modelo.

Hiperparámetros

```
clf = GradientBoostingClassifier(
                    random_state = 0,
                    n_estimators=150,
                    learning_rate=0.01,
                    max_depth=8,
                    subsample=0.75)
```

```
clf.fit(X_train,y_train)
```

GradientBoostingClassifier

```
GradientBoostingClassifier(learning_rate=0.01, max_depth=8, n_estimators=150,
                           random_state=0, subsample=0.75)
```

```
# score en set de entrenamiento
clf.score(X_train,y_train)
```

```
0.9213483146067416
```

```
# score en set de test
clf.score(X_test,y_test)
```

```
0.8324022346368715
```

KIBERNUM

Ventajas

1. Es altamente preciso y puede producir modelos predictivos de alta calidad para una amplia variedad de problemas de aprendizaje automático.

2. El algoritmo es flexible y puede ser utilizado para problemas de regresión y clasificación, así como para problemas con datos estructurados y no estructurados.

3. Es resistente al sobreajuste y puede generalizar bien a datos no vistos si se ajustan correctamente los hiperparámetros.

4- Puede manejar datos de alta dimensionalidad y puede manejar tanto características continuas como categóricas.

Desventajas

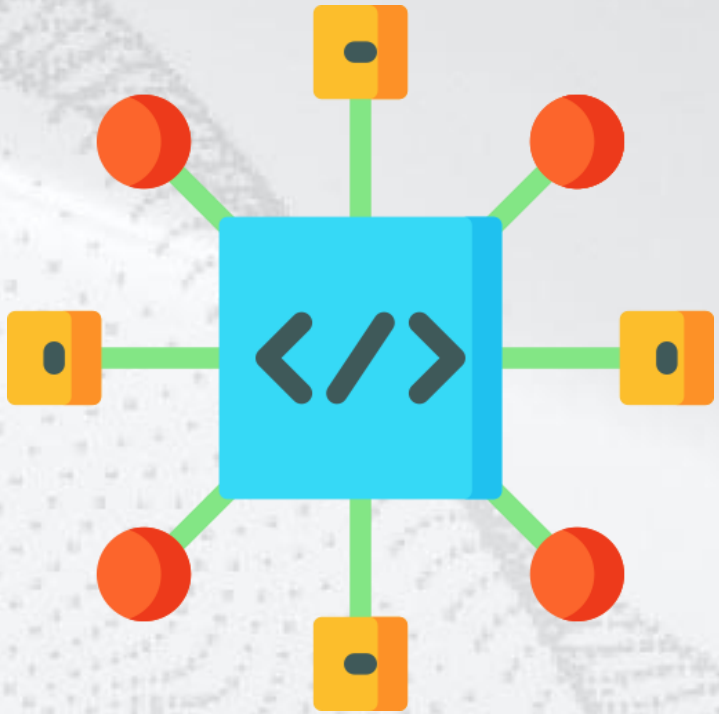
1. Puede ser computacionalmente costoso debido a la construcción iterativa de modelos y la necesidad de ajustar muchos hiperparámetros.

2. El algoritmo puede ser sensible a valores atípicos y errores en los datos de entrenamiento, lo que puede afectar la precisión del modelo.

3. Puede ser difícil de interpretar debido a la complejidad del modelo y la falta de transparencia de los modelos de árboles de decisión subyacentes.

4. Puede ser sensible a la selección de características y puede ser necesario realizar una selección cuidadosa de características para obtener un modelo de alta calidad.

XGBoost



KIBERNUM

¿Qué es XGBoost?

XGBoost (Extreme Gradient Boosting) es una biblioteca de software de código abierto desarrollada para implementar Gradient Boosting. Es una implementación mejorada del algoritmo Gradient Boosting que utiliza una técnica de regularización llamada "Gradient-based Regularization", que reduce el sobreajuste y mejora la generalización del modelo.

XGBoost es conocido por su eficiencia y velocidad de entrenamiento, lo que lo hace popular en competencias de aprendizaje automático y aplicaciones en tiempo real. También, es compatible con múltiples lenguajes de programación, como Python, R, Java, Scala y Julia.

Entre las características de XGBoost se incluyen:

- Árboles de decisión personalizados con diferentes criterios de división.
- Técnicas de regularización, como la penalización L1 y L2, para evitar el sobreajuste.
- Capacidad para manejar datos faltantes.
- Paralelización de procesamiento para acelerar el entrenamiento del modelo.
- Soporte para múltiples objetivos, como regresión y clasificación binaria y multiclase.
- Interfaz fácil de usar y flexible.

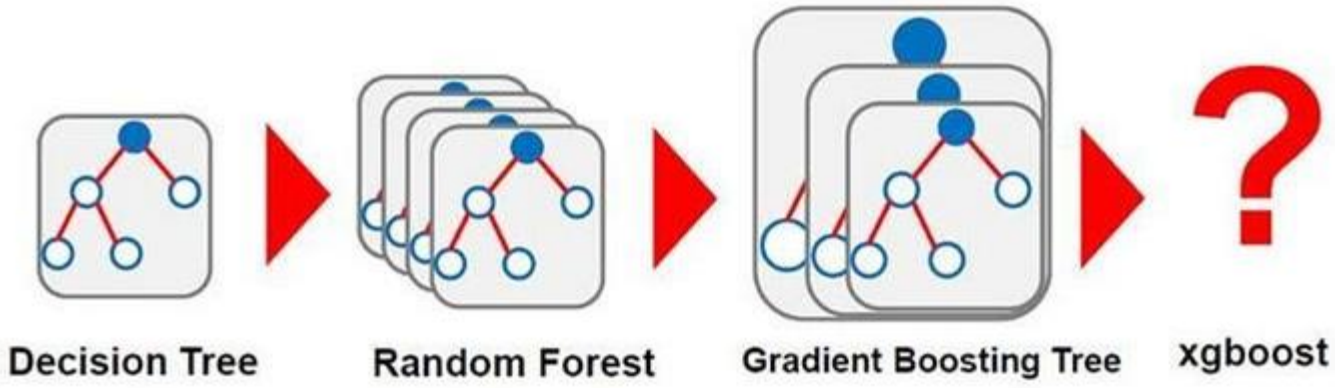
KIBERNUM

¿Qué es XGBoost?

XGBoost ha demostrado su eficacia en muchos conjuntos de datos en los que ha superado a otros algoritmos de aprendizaje automático en términos de rendimiento y precisión. Es una herramienta útil para los científicos de datos y los desarrolladores de aprendizaje automático que buscan construir modelos de alta calidad con rapidez y eficiencia.

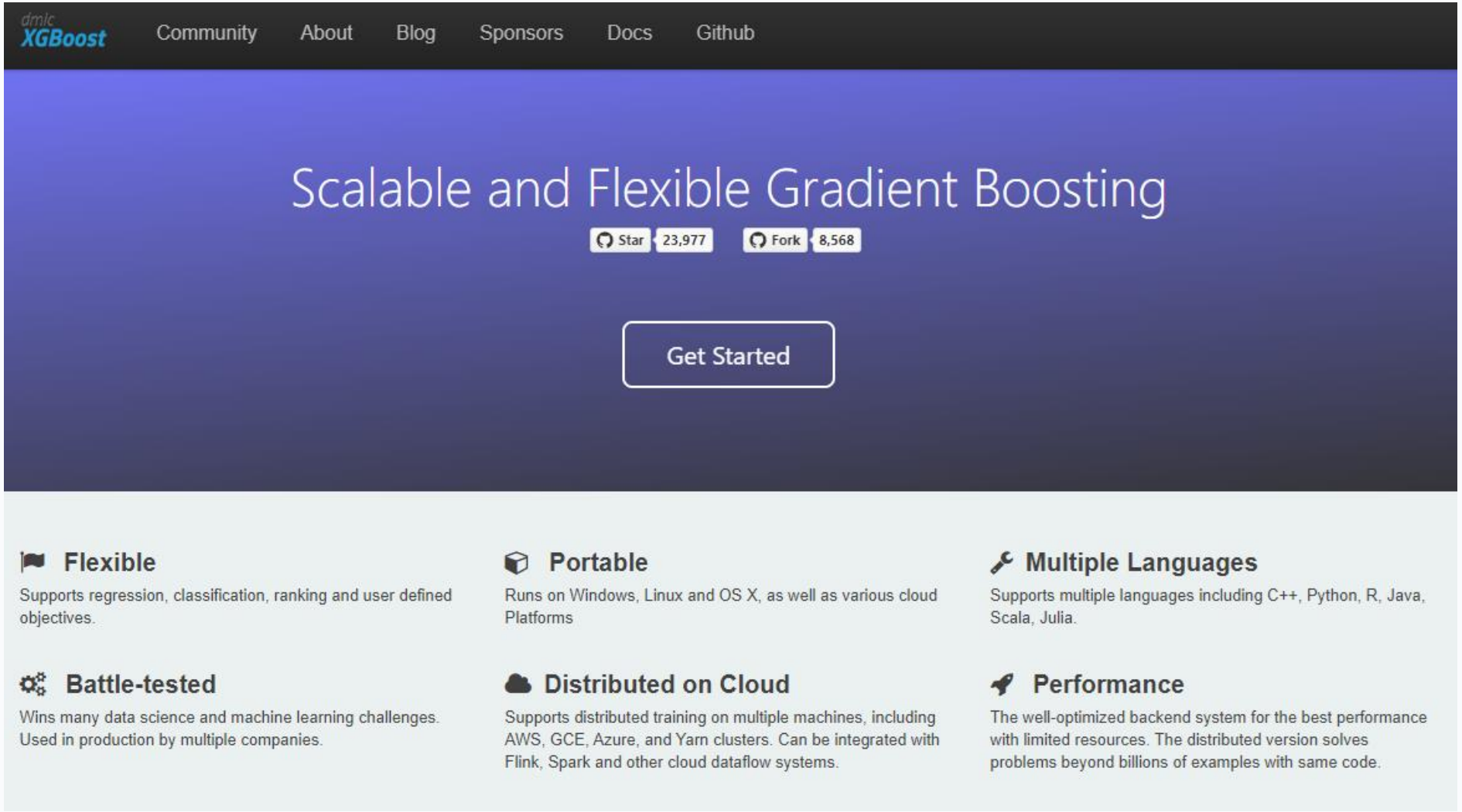
Objetivos de XGBoost:

- **Velocidad de ejecución**: XGBoost casi siempre fue más rápido que las otras implementaciones comparativas de R, Python Spark y H2O y es realmente más rápido en comparación con los otros algoritmos.
- **Rendimiento del modelo**: XGBoost domina los conjuntos de datos estructurados o tabulares en problemas de modelado predictivo de clasificación y regresión.



KIBERNUM

Web XGBoost



https://xgboost.ai/

KIBERNUM

Implementación en Python

Esta librería no viene en la distribución Anaconda, por lo tanto, debe ser instalada.

Conda

You may use the Conda packaging manager to install XGBoost:

```
conda install -c conda-forge py-xgboost
```

Conda should be able to detect the existence of a GPU on your machine and install the correct variant of XGBoost. If you run into issues, try indicating the variant explicitly:

```
# CPU only
conda install -c conda-forge py-xgboost-cpu
# Use NVIDIA GPU
conda install -c conda-forge py-xgboost-gpu
```

Python

Nightly builds are available. You can go to this page, find the wheel with the commit ID you want and install it with pip:

```
pip install <url to the wheel>
```

The capability of Python pre-built wheel is the same as stable release.

KIBERNUM

Implementación XGBoost en Python

Revisemos el caso Titanic con XGBoost Boosting. Vamos directamente al modelo:

Importamos clasificador desde el módulo ensemble de la librería sklearn

Instanciamos clasificador AdaBoost con hiperparámetros por defecto

Entrenamos clasificador

Score en set de entrenamiento

Score en set de test

Modelamiento con XGBoost

```
from xgboost import XGBClassifier
```

```
clf = XGBClassifier(random_state = 0)
```

```
clf.fit(X_train,y_train)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, random_state=0, ...)
```

```
# score en set de entrenamiento
clf.score(X_train,y_train)
```

```
0.9550561797752809
```

```
# score en set de test
clf.score(X_test,y_test)
```

```
0.8268156424581006
```

Hiperparámetros XGBoost

Algunos de los hiperparámetros más importantes de XGBoost son:

- **n_estimators**: este hiperparámetro representa el número de árboles que se deben ajustar en el modelo.
- **max_depth**: profundidad máxima de cada árbol.
- **learning_rate**: controla la tasa de aprendizaje del modelo.
- **subsample**: representa la fracción de observaciones que se muestrean aleatoriamente para construir cada árbol.
- **colsample_bytree**: representa la fracción de columnas que se muestrean aleatoriamente para construir cada árbol.

Hiperparámetros

```
xgb = XGBClassifier(n_estimators=100,
                    max_depth=10,
                    learning_rate=0.01,
                    colsample_bytree=0.8,
                   random_state = 0)
```

```
xgb.fit(X_train,y_train)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=0.8, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.01, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=10, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, random_state=0, ...)
```

```
# score en set de entrenamiento
xgb.score(X_train,y_train)
```

```
0.8637640449438202
```

```
# score en set de test
xgb.score(X_test,y_test)
```

```
0.8268156424581006
```

KIBERNUM

Ventajas

1. Escalabilidad: es altamente escalable y se puede utilizar en conjuntos de datos grandes.

1. Rendimiento: es conocido por su alto rendimiento en comparación con otros algoritmos de aprendizaje automático.

1. Regularización: proporciona técnicas de regularización incorporadas para evitar el sobreajuste del modelo.

1. Flexibilidad: se puede utilizar para una variedad de problemas de aprendizaje automático, como clasificación, regresión y ranking.

1. Interpretación: proporciona una medida de la importancia de cada característica para ayudar en la interpretación del modelo.

Desventajas

1. Hiperparámetros: tiene muchos hiperparámetros que deben ajustarse correctamente para obtener un buen rendimiento.

1. Tiempo de entrenamiento: El entrenamiento de modelos XGBoost puede llevar mucho tiempo en conjuntos de datos muy grandes.

1. Memoria: El uso de memoria de XGBoost puede ser alto en conjuntos de datos grandes.

1. Requerimientos de recursos: requiere recursos de hardware considerables, como memoria y procesamiento de CPU o GPU para su entrenamiento y predicción.

Dudas y consultas

KIBERNUM

Fin de la Presentación

KIBERNUM