Boosting

1. Introdução

Em *Machine Learning*, os modelos de *Ensemble* compõem uma classe de algoritmos que se assemelham por **combinar as predições de diferentes estimadores** para ter uma resposta mais robusta. Os métodos de *Ensemble* podem ser divididos em dois principais tipo:

- **Bagging**: utilizando da composição de duas técnicas (*Bootstraping* e *Aggregation*), o método de *Bagging* tem como foco construir diversos estimadores independentes, que irão compor a predição final a partir da média dos resultados ou mesmo pela classe mais votada. O principal objetivo deste tipo de modelo é reduzir **variância**, de maneira que o modelo final seja melhor que todos os modelos individuais. Ex.: **Random Forest**;

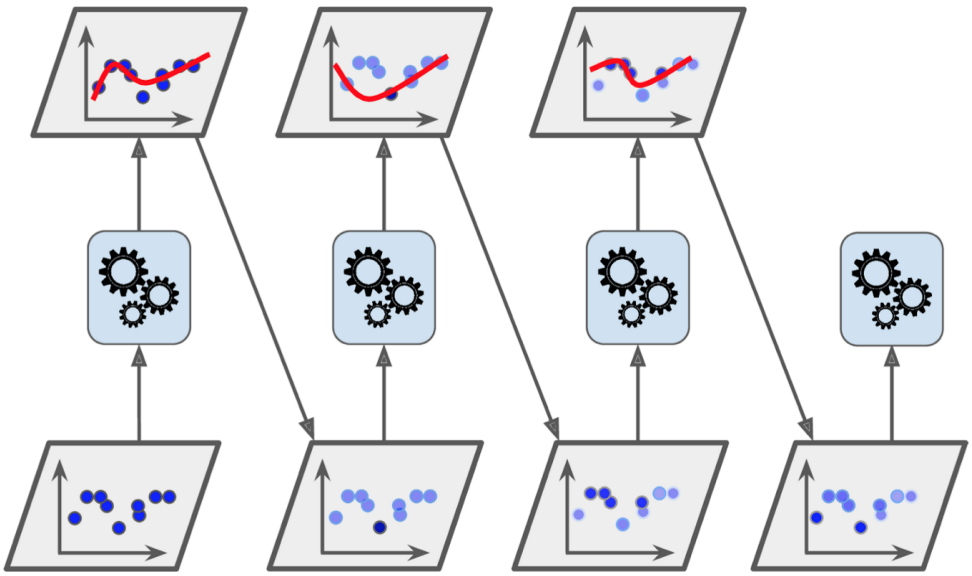
- **Boosting**: o método de *Boosting* funciona a partir da construção de estimadores de forma sequencial, de modo que estimadores anteriores influenciem os próximos estimadores na sequência, com o objetivo de reduzir o viés do modelo como um todo, este processo feito a partir dos valores resíduos em cada etapa do modelo. Ex.: **AdaBoost**, **GradientBoosting**, **XGBoosting** entre outros

Nos tópicos a seguir, serão desenvolvidos alguns dos principais modelos de **Boosting** utilizados em aplicações de classificação (novamente vale ressaltar que os mesmos modelos podem ser utilizados em caso de Regressão) e suas premissas como a composição dos modelos, por exemplo o **AdaBoost** e **GradientBoosting**.

2. Adaboost

O modelo **Adaptive Boosting** (comumente chamado de **AdaBoost**) é formado a partir da sequência de vários modelos de árvores simples com apenas um nó, sendo estes modelos fracos chamados de *stumps*. Este processo ocorre de forma **sequencial** pois a cada estágio de

treinamento do modelo, são atribuídos pesos às observações a partir dos valores residuais do modelo anterior. Então a cada estágio de treinamento, são atualizados os pesos de modo a dar significância diferentes para cada uma das observações.



Fonte: Medium

O procedimento por trás do *AdaBoost* pode ser resumido nas etapas a seguir:

- **Determinar o atributo que melhor separa os dados no primeiro nó:** em processo análogo ao que ocorre nos modelos de árvore de Decisão, utiliza-se de cálculos como **Critério de Gini** ou **Critério de Entropia** para determinar qual dos atributos separa melhor os dados entre as classes;

Exemplo: Dadas as informações a respeito de pacientes com COVID e alguns sintomas que apresentaram mostrados na tabela a seguir:

Tosse	Febre	Está com COVID?
Sim	Sim	Sim
Não	Não	Sim
Sim	Sim	Sim
Sim	Sim	Sim
Sim	Não	Não
Não	Não	Não
Sim	Não	Sim
Sim	Sim	Não
Não	Não	Não
Não	Não	Não
Sim	Sim	Sim
Sim	Não	Não
Sim	Sim	Sim

Tosse	Febre	Está com COVID?
Sim	Não	Sim
Não	Sim	Sim
Não	Sim	Não

A partir da tabela, qual dos atributos separa melhor o conjunto de dados inicialmente, **Tosse** ou **Febre**?

Pegando por exemplo, a partir do critério de **impureza de Gini**, que mede o quão "impuras" são as folhas das árvores construídas após as quebras nos nós. O coeficiente é dado por:

$$Gini(D) = 1 - \sum p_i^2$$

Onde p_i são as proporções de separação do target em cada quebra. Fazendo o devido agrupamento dos dados em relação ao **Tosse**, pode-se resumir na tabela a seguir:

Tosse			
Não		Sim	
Está com COVID	NÃO Está com COVID	Está com COVID	NÃO Está com COVID
2	4	7	3
6		10	

No caso das **Impureza antes da divisão**: Como não havia separação alguma, a impureza era dada simplesmente pelo balanço natural dos dados:

$$G(\text{pré-divisão}) = 1 - ((8/16)^2 + (8/16)^2) = 0.5$$

Calculando o Gini para cada um dos casos de **Tosse**:

$$G(\text{Tosse} \mid \text{Não}) = 1 - (\frac{2}{6}^2 + \frac{4}{6}^2) = 0.444$$

$$G(\text{Tosse} \mid \text{Sim}) = 1 - (\frac{7}{10}^2 + \frac{3}{10}^2) = 0.420$$

Ou seja, após a divisão, a impureza total passa a ser a média ponderada:

$$G(\text{pós-divisão}) = \frac{6}{16} \times G(\text{Tosse} \mid \text{Não}) + \frac{10}{16} \times G(\text{Tosse} \mid \text{Sim}) = 0.375 \times 0.444 + 0.625 \times 0.420 = 0$$

Assim, **a perda de impureza proporcionada pela quebra** dos dados segundo o atributo **Tosse** é de:

$$\Delta G_{\text{sexo}} = G(\text{pré-divisão}) - G(\text{pós-divisão}) = 0.5 - 0.42275 = 0.07725$$

Agora deve-se avaliar em relação ao atributo **Febre**:

Febre			
Não		Sim	
Está com COVID	NÃO Está com COVID	Está com COVID	NÃO Está com COVID
2	6	6	2
8		8	

Calculando de forma análoga para o atributo **Febre**:

$$G(\text{Febre} \mid \text{Não}) = 1 - (\frac{6}{8}^2 + \frac{2}{8}^2) = 0.375$$

$$G(\text{Febre} \mid \text{Sim}) = 1 - (\frac{2}{8}^2 + \frac{6}{8}^2) = 0.375$$

Ou seja, após a divisão, a impureza total passa a ser a média ponderada:

$$G(\text{pós-divisão}) = \frac{8}{16} \times G(\text{Febre} \mid \text{Não}) + \frac{8}{16} \times G(\text{Febre} \mid \text{Sim}) = 0.5 \times 0.375 + 0.5 \times 0.375 = 0.375$$

Assim, **a perda de impureza proporcionada pela quebra** dos dados para o atributo **Febre** é de:

$$\Delta G_{\text{Série}} = G(\text{pré-divisão}) - G(\text{pós-divisão}) = 0.5 - 0.375 = 0.125$$

Comparando ambos os resultados, nota-se que o atributo que teve melhor quebra foi a **Febre**;

- **Atribuir os pesos para cada uma das observações:** Para a primeira iteração, os pesos são definidos igualmente para todas as observações do conjunto de dados. Para o exemplo da tabela de sintomas e casos de COVID, tem-se que:

Tosse	Febre	Está com COVID?	Peso
Sim	Sim	Sim	1/16
Não	Não	Sim	1/16
Sim	Sim	Sim	1/16
Sim	Sim	Sim	1/16
Sim	Não	Não	1/16
Não	Não	Não	1/16
Sim	Não	Sim	1/16
Sim	Sim	Não	1/16
Não	Não	Não	1/16
Não	Não	Não	1/16
Sim	Sim	Sim	1/16
Sim	Não	Não	1/16
Sim	Sim	Sim	1/16
Sim	Não	Sim	1/16
Não	Sim	Sim	1/16
Não	Sim	Não	1/16

- **Construir a árvore com o melhor atributo e levantar os erros:** Neste processo, deve-se avaliar os erros cometidos pelo modelo. Voltando ao exemplo dos sintomas e COVID, tem-se que:

Febre			
Não		Sim	
Está com COVID	NÃO Está com COVID	Está com COVID	NÃO Está com COVID
2	6	6	2
8		8	

Do conjunto de dados, têm-se 3 erros para o caso de pessoas sem Febre, mas com COVID e 2 casos de pessoas com Febre, mas sem COVID. Portanto, o erro total ϵ_{tot} do modelo é:

$$\epsilon_{tot} = \frac{5}{16}$$

- **Cálculo do ajuste do peso:** A partir do erro total do modelo, deve-se definir o ajuste a ser atribuído aos pesos das observações. Seguindo o exemplo anterior:

$$\alpha = \eta * \log \frac{1 - \epsilon_{tot}}{\epsilon_{tot}}$$

Onde o parâmetro η é a taxa de aprendizagem do modelo, usualmente utilizado para valores de $\eta = $1. Calculando o parâmetro α para o exemplo em desenvolvimento:

$$\alpha = 1 * \log \frac{1 - \frac{5}{16}}{\frac{5}{16}} = 1.0414$$

- **Definindo os novos pesos para o modelo a partir do ajuste:** Dado o ajuste dos pesos, calcula-se os novos valores de peso de acordo com os erros e acertos do modelo. Para o caso de acertos do modelo, deve-se calcular os novos pesos com a fórmula abaixo:

$$peso_{novo} = peso * e^{-\alpha}$$

Já para os casos em que o modelo errou as predições, tem-se que:

$$peso_{novo} = peso * e^{\alpha}$$

Calculando ambos os casos para o exemplo:

Para os casos **corretos**:

$$peso_{novo} = \frac{1}{16} * e^{-1.0414} = 0.0221$$

;

Para os casos **errados**:

$$peso_{novo} = \frac{1}{16} * e^{1.0414} = 0.1771$$

;

Atualizando os valores na tabela:

Tosse	Febre	Está com COVID?	Peso	Novo Peso
Sim	Sim	Sim	1/16	0.0221
Não	Não	Sim	1/16	0.1771
Sim	Sim	Sim	1/16	0.0221
Sim	Sim	Sim	1/16	0.0221
Sim	Não	Não	1/16	0.0221
Não	Não	Não	1/16	0.0221
Sim	Não	Sim	1/16	0.1771
Sim	Sim	Não	1/16	0.1771
Não	Não	Não	1/16	0.0221
Não	Não	Não	1/16	0.0221
Sim	Sim	Sim	1/16	0.0221
Sim	Não	Não	1/16	0.0221
Sim	Sim	Sim	1/16	0.0221
Sim	Não	Sim	1/16	0.1771
Não	Sim	Sim	1/16	0.0221
Não	Sim	Não	1/16	0.1771

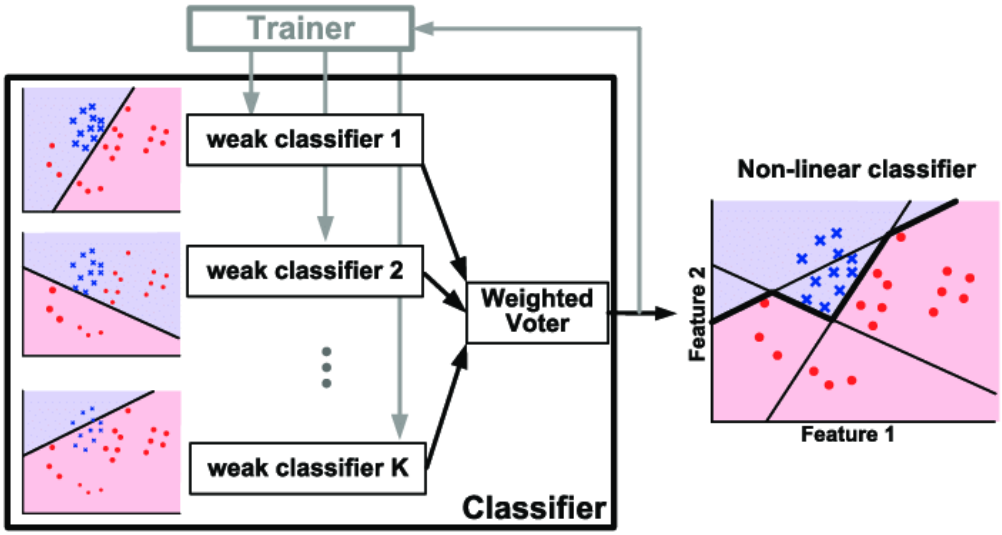
- **Normalização dos novos pesos:** Para finalizar a iteração, deve-se normalizar os pesos novos atribuídos para cada uma das observações. No caso do exemplo desenvolvido, os pesos normalizados ficam da seguinte forma:

Tosse	Febre	Está com COVID?	Peso	Novo Peso	Novos Pesos Normal.
Sim	Sim	Sim	1/16	0.0221	0.0196
Não	Não	Sim	1/16	0.1771	0.1822
Sim	Sim	Sim	1/16	0.0221	0.0196
Sim	Sim	Sim	1/16	0.0221	0.0196
Sim	Não	Não	1/16	0.0221	0.0196
Não	Não	Não	1/16	0.0221	0.0196
Sim	Não	Sim	1/16	0.1771	0.1822
Sim	Sim	Não	1/16	0.1771	0.1822
Não	Não	Não	1/16	0.0221	0.0196
Não	Não	Não	1/16	0.0221	0.0196
Sim	Sim	Sim	1/16	0.0221	0.0196
Sim	Não	Não	1/16	0.0221	0.0196
Sim	Sim	Sim	1/16	0.0221	0.0196
Sim	Não	Sim	1/16	0.1771	0.1822
Não	Sim	Sim	1/16	0.0221	0.0196
Não	Sim	Não	1/16	0.1771	0.1822

- **Repetir os passos anteriores com um novo conjunto de dados:** Para as novas iterações, deve-se gerar um novo conjunto de dados levando em consideração a proporção dos pesos. Ou seja, as observações com pesos maiores vão aparecer em maior quantidade

que as com pesos menores. Após isto, segue-se novamente o fluxo de determinação da melhor quebra e recálculo dos pesos;

- **Cálculo da predição final do modelo:** A determinação final das predições vai levar em consideração os pesos em cada um dos modelos fracos (*stumps*).



Fonte: ResearchGate

A construção de um modelo AdaBoost em Python, a partir da implementação do *Scikit-learn* pode ser feita seguinte o código abaixo:

```
# Carrega o modelo do AdaBoostClassifier
from sklearn.ensemble import AdaBoostClassifier

# Instancia o Modelo
model = AdaBoostClassifier(random_state = 42)

# Fit do modelo utilizando os dados de treino
model.fit(X_train, y_train)

# gera as predições do modelo utilizando os dados de teste
y_pred = model.predict(X_test)
```

Exemplo Prático

```
# Aplicando o modelo de AdaBoost na classificação de sobreviventes no
Titanic
# Carregando as bibliotecas necessárias
```

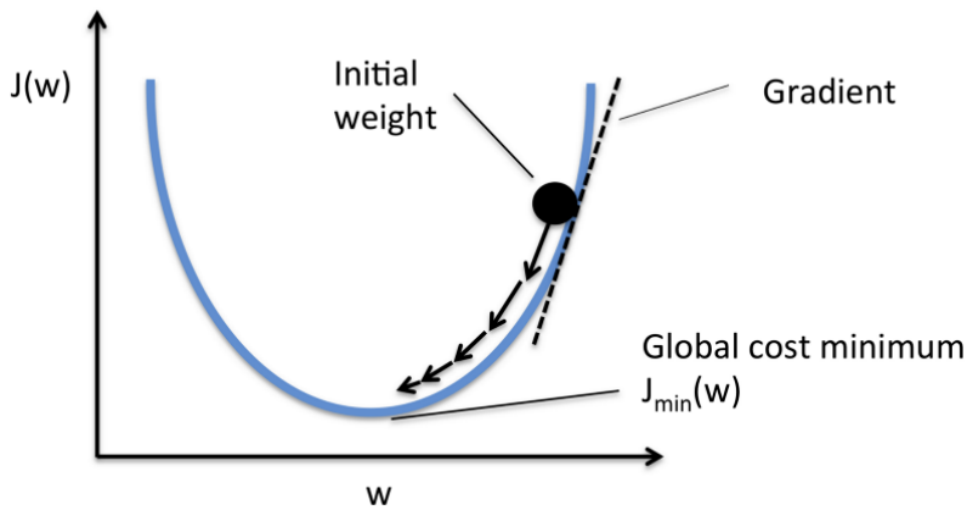
```
import pandas as pd
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import classification_report

# carrega o conjunto de dados
titanic = sns.load_dataset('titanic')

# Removendo variáveis com problemas ou redundantes
# OBS.: sempre dê uma olhada no conjunto de dados para avaliar a
qualidade deles
titanic.drop(['embarked', 'who', 'class', 'adult_male', 'deck',
'alive'], axis = 1, inplace = True)

# Avaliando os dados faltantes
titanic.isna().sum()

# Preenchendo a idade com a média das idades
titanic['age'].fillna(titanic['age'].mean(), inplace = True)

# Removendo as linhas com valores nulos da variável "embark_town"
titanic.dropna(inplace = True)

# Aplicando um get_dummies nos dados categóricos
titanic_new = pd.get_dummies(titanic,                          #
conjunto de dados
                             columns = ['sex', 'embark_town'], #
colunas com variável categórica
                             drop_first = True)                # remove
a primeira categoria

# Separação dos dados em atributos (X) e variável resposta (y)
X = titanic_new.drop('survived', axis = 1)
y = titanic_new['survived']

# Separação da base em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X,
# atributos
                                                    y,
# Variável resposta
                                                    test_size = 0.3,
# proporção base de treino
                                                    random_state = 42,
# semente aleatório
                                                    stratify = y)
```

```
# mantém a proporção das classes

# Instancia o modelo
model = AdaBoostClassifier(random_state = 42)

# Fit do modelo com os dados de treino
model.fit(X_train, y_train)

# Gera as predições com os dados de teste
y_pred = model.predict(X_test)

# Avaliando o desempenho do modelo
print(classification_report(y_test, y_pred))
```

3. Gradiente Descendente, *Gradient Boosting* e *XGBoost*

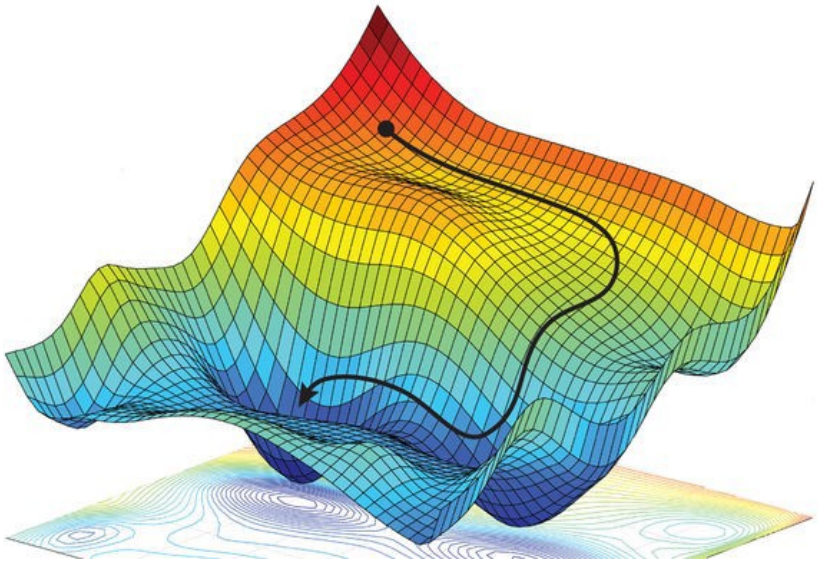
Todos os modelos de *Boosting* partem da mesma estratégia utilizando árvores mais simples e estimando erros. A principal diferença dos diversos modelos de *Boosting* que existe é justamente o algoritmo de otimização do modelo, ou seja, como é feita a otimização da **função de perda** (*loss function*).

No caso de modelos como o **GradientBoosting**, o principal diferencial é justamente utilizar um otimizador através de **gradiente descente**. Conforme o modelo faz uma iteração e calcula a função de custo do modelo, utiliza-se de gradientes para definir o ajuste necessário na função inicial e repetir o processo iterativo:



Fonte: Medium

Ou seja, para a otimização dos pesos o modelo utiliza de gradientes (que são as derivadas da função) para chegar nos menores valores da função de perda e dessa forma chegar nos valores ideais de pesos.



Fonte: Medium

A construção de um modelo GradientBoosting em Python, a partir da implementação do *Scikit-Learn*, pode ser feita com o código abaixo:

```
# Carrega o modelo do AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier

# Instancia o Modelo
model = GradientBoostingClassifier(random_state = 42)

# Fit do modelo utilizando os dados de treino
model.fit(X_train, y_train)

# gera as predições do modelo utilizando os dados de teste
y_pred = model.predict(X_test)
```

Exemplo Prático

```
# Continuando com a aplicação na classificação de sobreviventes no
Titanic
# Carregando a biblioteca para o GradientBoosting
from sklearn.ensemble import GradientBoostingClassifier
```

```
# Instancia o modelo
model = GradientBoostingClassifier(random_state = 42)

# Fit do modelo com os dados de treino
model.fit(X_train, y_train)

# Gera as predições com os dados de teste
y_pred = model.predict(X_test)

# Avaliando o desempenho do modelo
print(classification_report(y_test, y_pred))
```

No caso de outros modelos baseados em gradiente descendente como **XGBoost** (sigla para *Extreme Gradient Boosting*) e o **LightGBM**, modelo largamente utilizado em competições no *Kaggle*, são modelos que partem da mesma estratégia de otimização mas têm arquiteturas robustas e custo computacional menor para gerar as predições.

No caso do *XGBoost* algumas vantagens que existem em comparação ao *Gradient Boosting* são: aplicação de regularização (L1 e L2), processamento paralelo, gerenciamento de valores faltantes, utilização de técnicas de poda de árvore (conhecidas como *pruning*, onde o modelo remove as separações que não têm ganho positivo ao longo da árvore) e possibilidade de aplicar validação cruzada entre as iterações, isto já nativo do próprio *XGBoost*. Agora no comparativo entre o *XgBoost* e o *LightGBM*, o *LightGBM* têm algumas vantagens no quesito tempo de processamento, uso de memória e compatibilidade com conjuntos de dados grandes.

As implementações tanto do *XGBoost* e do *LightGBM* pode ser feito com os códigos a seguir:

XGBoost:

```
# Instalação da biblioteca xgboost
!pip install xgboost

# Carrega a função do modelo
from xgboost import XGBClassifier

# Instancia o Modelo
model = XGBClassifier(random_state = 42)

# Fit do modelo utilizando os dados de treino
model.fit(X_train, y_train)

# gera as predições do modelo utilizando os dados de teste
y_pred = model.predict(X_test)
```

Exemplo Prático

```
# Continuando com a aplicação na classificação de sobreviventes no
Titanic
# Instalação da biblioteca xgboost
!pip install xgboost

# Carrega a função do modelo
from xgboost import XGBClassifier

# Instancia o Modelo
model = XGBClassifier(random_state = 42)

# Fit do modelo com os dados de treino
model.fit(X_train, y_train)

# Gera as predições com os dados de teste
y_pred = model.predict(X_test)

# Avaliando o desempenho do modelo
print(classification_report(y_test, y_pred))
```

LightGBM:

```
# Instalação da biblioteca LightGBM
!pip install lightgbm

# Carrega a função do modelo
import lightgbm as lgb

# Instancia o modelo
model = lgb.LGBMClassifier(random_state = 42)

# Fit do modelo utilizando os dados de treino
model.fit(X_train, y_train)

# gera as predições do modelo utilizando os dados de teste
y_pred = model.predict(X_test)
```

Exemplo Prático

```
# Continuando com a aplicação na classificação de sobreviventes no
Titanic
# Instalação da biblioteca LightGBM
!pip install lightgbm

# Carrega a função do modelo
import lightgbm as lgb

# Instancia o modelo
model = lgb.LGBMClassifier(random_state = 42)

# Fit do modelo com os dados de treino
model.fit(X_train, y_train)

# Gera as predições com os dados de teste
y_pred = model.predict(X_test)

# Avaliando o desempenho do modelo
print(classification_report(y_test, y_pred))
```

Materiais Complementares

Canal *StatsQuest*, vídeo sobre AdaBoost Clearly Explained;

Canal *StatsQuest*, vídeo sobre Gradient Boosting Classification;

Canal *StatsQuest*, vídeo sobre XGBoost Classification;

Documentação no Scikit-Learn sobre o AdaBoost;

Documentação no Scikit-Learn sobre o Gradient Boosting;

Documentação da biblioteca XGBoost;

Artigo publicado por Jonhy Silva no Medium - **Uma breve introdução ao algoritmo de Machine Learning Gradient Boosting utilizando a biblioteca Scikit-Learn**;

Artigo publicado por Pedro Azambuja no Medium - **AdaBoost (Adaptive Boosting)**;

Artigo publicado por Arthur Lamblet Vaz no Medium - **Gradientes Descendentes na prática — melhor jeito de entender**;

Referências

James, Gareth, et al. An Introduction to Statistical Learning: With Applications in R. Alemanha, Springer New York, 2013;

Talwalkar, Ameet, et al. Foundations of Machine Learning. Reino Unido, MIT Press, 2018.

Tópico anterior

Próximo Tópico