😊 XG Boosting (Extreme Gradient Boosting)

XGBoost, or Extreme Gradient Boosting, is a popular machine learning algorithm that's part of the boosting family. In simple terms, it's like having a team of decision trees that work together to make better predictions.

Imagine you're organizing a guessing game:

You ask your friends to guess the number of candies in a jar. At first, one friend guesses, but the guess is off by quite a bit. Instead of giving up, you ask another friend to make a guess, but this time their goal is to focus on correcting the mistakes from the first guess. You keep doing this, each time focusing on improving the earlier guesses.

Now, think of this process as XGBoost:

1. **Start with a simple guess**: XGBoost begins with one simple decision tree (like your first friend guessing). This tree isn't very accurate but gives a starting point.
2. **Improve the guess step by step**: After the first tree, XGBoost adds more trees, but each new tree tries to fix the mistakes made by the previous ones, like your friends improving their guesses.
3. **Combine all the guesses**: Once enough trees are built, XGBoost combines all their predictions to make a final, much better prediction.

Example:

Let's say you want to predict the price of houses. The first decision tree might predict all houses cost $200,000, but that's not accurate. So, the next tree sees the mistakes (e.g., the bigger houses are more expensive) and adjusts its guesses. This process repeats, with each tree focusing on fixing mistakes, until the algorithm gives a pretty accurate estimate for each house.

Why is XGBoost special?

- **Fast and efficient**: It's designed to handle large datasets quickly.
- **Highly accurate**: By refining the guesses at each step, it often performs better than other algorithms.

In summary, XGBoost is like a smart guessing game where each new guess focuses on improving the previous ones to make a very good final prediction.

XGBoost, short for **Extreme Gradient Boosting**, is a highly efficient and powerful machine learning algorithm based on the concept of **boosting**. It's designed to handle large datasets and complex models efficiently, often outperforming other machine learning algorithms.

Key Concepts Behind XGBoost:

1. **Boosting**:
    - Boosting is a technique where multiple weak models (usually decision trees) are trained sequentially, and each new model focuses on correcting the errors of the previous one.

    - Instead of building a single strong model, boosting combines several weak models (weak learners) to improve overall performance.

2. **Gradient Boosting**:
    - In gradient boosting, each new tree is built to minimize the errors (residuals) of the previous model by using gradient descent. It's like iteratively adjusting the model to reduce prediction errors.

3. **Extreme Gradient Boosting (XGBoost)**:
    - XGBoost takes gradient boosting to the next level by adding optimizations, such as regularization (to prevent overfitting), parallel processing, and handling missing data efficiently. It also includes techniques to handle large-scale datasets effectively.

How XGBoost Works:

- **Step 1: Initialization**: XGBoost starts by creating a simple initial model, typically by making an average prediction for the target variable.
- **Step 2: Calculate Residuals (Errors)**: The algorithm calculates the residuals, which represent how far off the model's predictions are from the actual values.
- **Step 3: Build New Trees**: A new decision tree is created that tries to reduce the errors from the previous step. This tree learns from the residuals and adjusts the predictions.
- **Step 4: Add Trees Iteratively**: More trees are added, each one focusing on minimizing the errors made by the previous trees. The final prediction is the sum of predictions from all the trees.
- **Step 5: Regularization**: To avoid overfitting (fitting the model too perfectly to the training data and performing poorly on new data), XGBoost applies regularization techniques to penalize overly complex models.

Why XGBoost is Popular:

- **Speed**: It uses advanced techniques like parallelization, making it faster than traditional gradient boosting methods.
- **Accuracy**: XGBoost often produces highly accurate models due to its focus on minimizing errors at each step.
- **Flexibility**: It supports various objective functions (e.g., regression, classification), and it can handle missing data.
- **Prevents Overfitting**: With regularization, XGBoost is less likely to overfit compared to other algorithms.

Example:

Let's say you're predicting whether a customer will buy a product (a classification problem). XGBoost will:

1. Start with a basic model that makes initial predictions.

2. Calculate how far off these predictions are (i.e., which customers were misclassified).
3. Build new models (trees) to correct these errors, one step at a time.
4. Combine the predictions from all the models to make a final, more accurate prediction.

Technical Features:

- **Tree Pruning**: XGBoost uses a process called "pruning" to stop growing the trees before they become too complex.
- **Handling Missing Data**: XGBoost can handle missing values directly without requiring preprocessing steps.
- **Regularization**: XGBoost includes L1 (lasso) and L2 (ridge) regularization to control overfitting, which is one of its key strengths over standard gradient boosting methods.

Applications:

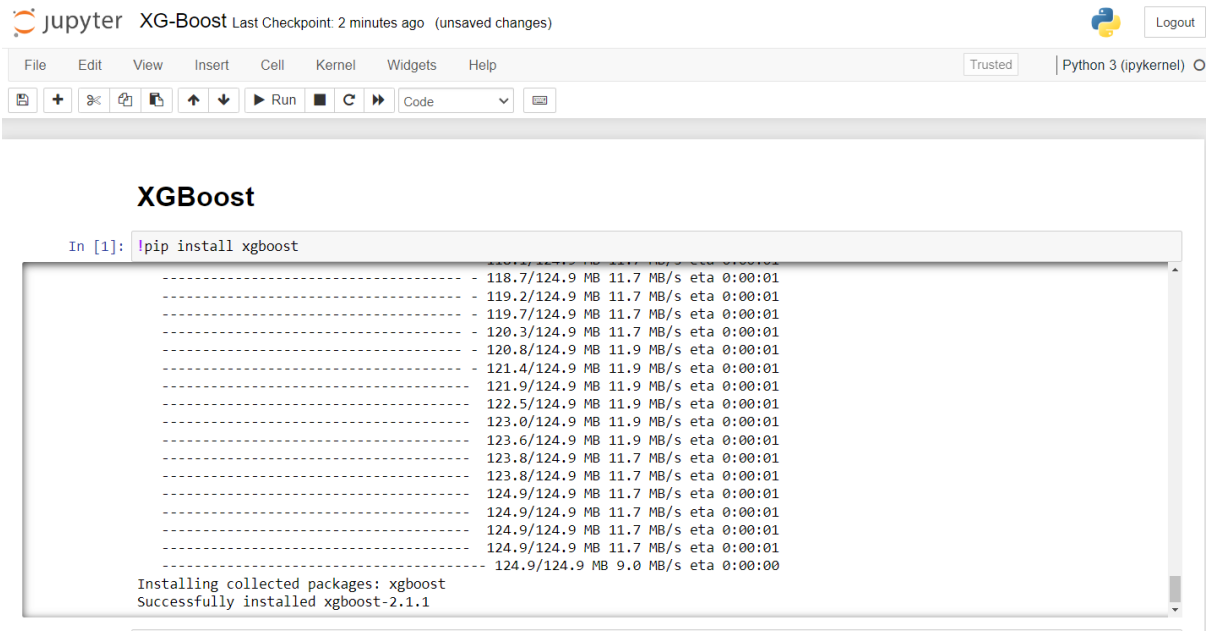
XGBoost is widely used in various fields like finance (predicting stock prices), marketing (customer churn prediction), healthcare (disease diagnosis), and in machine learning competitions (e.g., Kaggle).

Conclusion:

XGBoost is a powerful, fast, and highly flexible machine learning algorithm that refines the predictions made by a series of weak models. It's known for its excellent performance and efficiency, making it a go-to choice for many predictive modeling tasks.

😄 To begin with the Lab:

1. Open your Jupyter Notebook, then create a new Python Kernel. After that we installed XG boost in our Notebook.



2. Then we imported some important libraries and load our dataset as a data frame. We also displayed few entries from our dataset. Also, for this lab we are using our previous dataset which is income evaluation CSV file.

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

In [3]:
```
df = pd.read_csv("income_evaluation.csv")
df.head()
```

Out[3]:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

3. In this project implementation, we will import the library as import xgboost as xgb. To demonstrate this implementation, I will use the same data preprocessing steps we applied for the **support vector machine**. I will skip the details of the preprocessing we performed on the dataset, including the use of LabelEncoder and MinMaxScaler, as well as the data resampling. Once we have prepared the data, I have started by importing the XGBoost library.

In [4]:
```
df.columns
```

Out[4]:
```
Index(['age', ' workclass', ' fnlwgt', ' education', ' education-num',
       ' marital-status', ' occupation', ' relationship', ' race', ' sex',
       ' capital-gain', ' capital-loss', ' hours-per-week', ' native-country',
       ' income'],
      dtype='object')
```

In [5]:
```
df[' income'].value_counts()
```

Out[5]:
```
 income
 <=50K    24720
 >50K      7841
Name: count, dtype: int64
```

In [6]:
```
df[' marital-status'].unique()
```

Out[6]:
```
array([' Never-married', ' Married-civ-spouse', ' Divorced',
       ' Married-spouse-absent', ' Separated', ' Married-AF-spouse',
       ' Widowed'], dtype=object)
```

In [7]:
```
col_names = df.columns
col_names = [v.strip() for v in col_names]
col_names
```

Out[7]:
```
['age',
 'workclass',
 'fnlwgt',
 'education',
 'education-num',
 'marital-status',
 'occupation',
 'relationship',
 'race',
 'sex',
 'capital-gain',
 'capital-loss',
 'hours-per-week',
 'native-country',
 'income']
```

In [8]:
```
df.columns = col_names
df.drop(columns="fnlwgt",inplace=True)
```

In [9]:
```
df.head()
```

Out[9]:

	age	workclass	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	39	State-gov	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

In [10]:
```
df.isnull().sum()
```

Out[10]:
```
age               0
workclass         0
education         0
education-num     0
marital-status    0
occupation        0
relationship      0
race              0
sex               0
capital-gain      0
capital-loss      0
hours-per-week    0
native-country    0
income            0
dtype: int64
```

In [11]:
```
bins = [16,24,64,90]
labels=['young','adult','old']
df['age_types'] = pd.cut(df['age'], bins=bins,labels=labels)
df['income_num'] = np.where(df['income'] == " >50K",1,0).astype('int16')
```

In [12]:
```
df.head()
```

Out[12]:

	age	workclass	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income	age_types	income_num
0	39	State-gov	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K	adult	0
1	50	Self-emp-not-inc	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K	adult	0
2	38	Private	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K	adult	0
3	53	Private	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K	adult	0
4	28	Private	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K	adult	0

In [13]:
```
df.loc[df['workclass']=='?', 'workclass']= np.NaN
df.loc[df['occupation']=='?', 'occupation']= np.NaN
df.loc[df['native-country']=='?', 'native_country']= np.NaN
```

In [14]:
```
df = df.dropna(axis=1)
```

In [15]:
```
from sklearn.preprocessing import LabelEncoder
def label_encoder(a):
    le = LabelEncoder()
    df[a] = le.fit_transform(df[a])
label_list = ['workclass', 'education','marital-status',
       'occupation', 'relationship', 'race', 'sex','native-country', 'income']
for i in label_list:
    label_encoder(i)
```

In [16]:
```
from sklearn.preprocessing import MinMaxScaler
```

In [17]:
```
scaler = MinMaxScaler()
```

In [18]:
```
scaler.fit(df.drop(['income','age_types','income_num'],axis=1))
```

Out[18]:
```
MinMaxScaler()
```

In [19]:
```
scaled_features = scaler.transform(df.drop(['income','age_types','income_num'],axis=1))
```

In [20]:
```
columns=['age', 'workclass', 'education', 'education_num', 'marital_status',
       'occupation', 'relationship', 'race', 'sex', 'capital_gain',
       'capital_loss', 'hours_per_week', 'native_country']
```

In [21]:
```
df_scaled = pd.DataFrame(scaled_features,columns=columns)
df_scaled.head()
```

Out[21]:

	age	workclass	education	education_num	marital_status	occupation	relationship	race	sex	capital_gain	capital_loss	hours_per_week	native_countr
0	0.301370	0.875	0.600000	0.800000	0.666667	0.071429	0.2	1.0	1.0	0.02174	0.0	0.397959	0.95122
1	0.452055	0.750	0.600000	0.800000	0.333333	0.285714	0.0	1.0	1.0	0.00000	0.0	0.122449	0.95122
2	0.287671	0.500	0.733333	0.533333	0.000000	0.428571	0.2	1.0	1.0	0.00000	0.0	0.397959	0.95122
3	0.493151	0.500	0.066667	0.400000	0.333333	0.428571	0.0	0.5	1.0	0.00000	0.0	0.397959	0.95122
4	0.150685	0.500	0.600000	0.800000	0.333333	0.714286	1.0	0.5	0.0	0.00000	0.0	0.397959	0.12195

In [22]:
```
from imblearn.combine import SMOTETomek
from imblearn.under_sampling import NearMiss

X = df_scaled
y= df.income

# Implementing Oversampling for Handling Imbalanced
smk = SMOTETomek(random_state=42)
X_res,y_res=smk.fit_resample(X,y)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_res,y_res,test_size=0.20,random_state=101,shuffle=True)
```

4. In the code, you first import the XGBoost library and create data matrices for your training and testing datasets. You specify parameters for the model, including the learning rate and maximum tree depth, and set it up for a binary classification task.
5. Next, you train the model using the training data and make predictions on the testing data. You check the model's performance by calculating the area under the ROC curve (AUC), which measures how well the model distinguishes between classes.
6. Then, you set up a watchlist to monitor the performance on both training and testing datasets during the training process. You update the model parameters to include an evaluation metric (AUC) and train the model again, this time running for more iterations and displaying progress every ten rounds. Lastly, you suppress warnings for a cleaner output.

In [23]:
```
import xgboost as xgb
```

In [25]:
```
dtrain = xgb.DMatrix(X_train, label=y_train, feature_names=X_train.columns.tolist())
dtest = xgb.DMatrix(X_test, label=y_test, feature_names=X_test.columns.tolist())
```

In [26]:
```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

In [27]:
```
model = xgb.train(xgb_params, dtrain, num_boost_round=10)
```

```
C:\Users\PULKIT\anaconda3\Lib\site-packages\xgboost\core.py:158: UserWarning: [11:39:18] WARNING: C:\buildkite-agent\builds\buildkite-windows-cpu-autoscaling-group-i-0015a694724fa8361-1\xgboost\xgboost-ci-windows\src\learner.cc:740:
Parameters: { "silent" } are not used.

  warnings.warn(smsg, UserWarning)
```

In [28]:
```
y_pred = model.predict(dtest)
```

In [29]:
```
y_pred[:5]
```

Out[29]:
```
array([0.6628817 , 0.21896796, 0.6606529 , 0.03792919, 0.02435006],
      dtype=float32)
```

In [30]:
```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test, y_pred)
```

Out[30]:
```
0.9404533177881182
```

In [31]:
```
watchlist = [(dtrain, 'train'), (dtest, 'test')]
```

In [32]:
```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

In [33]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [34]:
```
model = xgb.train(xgb_params, dtrain,
                  num_boost_round=100,
                  evals=watchlist, verbose_eval=10)
```

```
[0]     train-auc:0.91008       test-auc:0.91124
[10]    train-auc:0.94493       test-auc:0.94202
[20]    train-auc:0.95961       test-auc:0.95431
[30]    train-auc:0.96472       test-auc:0.95841
[40]    train-auc:0.96866       test-auc:0.96149
[50]    train-auc:0.97315       test-auc:0.96521
[60]    train-auc:0.97571       test-auc:0.96719
[70]    train-auc:0.97711       test-auc:0.96815
[80]    train-auc:0.97935       test-auc:0.96981
[90]    train-auc:0.98053       test-auc:0.97042
[99]    train-auc:0.98142       test-auc:0.97094
```