XGBoost Algorithm

Introduction:

XGBoost (Extreme Gradient Boosting) is a highly efficient, scalable, and accurate implementation of gradient boosting. It has gained widespread popularity in data science and machine learning competitions (like Kaggle) due to its excellent performance, flexibility, and optimization techniques.

Motivation:

XGBoost is an extension of the Gradient Boosting framework designed to optimize speed and performance by adding regularization, improving parallelization, and reducing overfitting. It builds multiple decision trees sequentially, where each new tree corrects the errors of the previous trees.

Mathematical Intuition:

XGBoost minimizes a regularized objective function, which consists of a loss function and a regularization term.

Objective Function:

The objective function for XGBoost is:

```
Obj(θ) = L(θ) + Ω(θ)
```

Where:

- (L(Θ)) is the loss function (measuring how well the model fits the data),
- (Ω(Θ)) is the regularization term (to penalize complexity and prevent overfitting).

For classification tasks, the typical loss function is **log loss** or **cross-entropy loss**, and for regression tasks, it can be **mean squared error (MSE)**.

The regularization term is:

```
Ω(θ) = γT + 0.5λ∑(w_j^2)
```

Where:

- (γ) controls the complexity of the tree,
- (λ) penalizes large weights ((w_j)) to prevent overfitting,
- (T) is the number of terminal nodes (leaves) in the tree.

Gradient Descent in XGBoost:

XGBoost uses **additive training**, where new trees are added to minimize the residual errors. The new model is combined with the existing one by learning a function (h(x)) that minimizes the residuals between the true value (y) and the predicted value (\hat{y}).

The algorithm minimizes the following objective:

```
L = ∑ l(y_i, ŷ_i^(t-1) + f_t(x_i)) + Ω(f_t)
```

Where:

- (l) is the loss function (such as squared error for regression),
- (ŷ_i^{(t-1)}) is the prediction from the model at iteration (t-1),
- (f_t(x_i)) is the new model added at iteration (t).

For the new learner (f_t), the second-order Taylor expansion of the loss function is used:

```
Obj^(t) ≈ ∑(g_i f_t(x_i) + 0.5 h_i f_t^2(x_i)) + Ω(f_t)
```

Where:

- (g_i) is the first-order gradient of the loss function w.r.t (ŷ_i),
- (h_i) is the second-order gradient (Hessian) of the loss function w.r.t (ŷ_i).

Example of XGBoost for Classification:

```python
import xgboost as xgb
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Create synthetic dataset
X, y = make_classification(n_samples=1000, n_features=20,
n_classes=2, random_state=42)

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Initialize XGBoost classifier
xgb_clf = xgb.XGBClassifier(n_estimators=100, learning_rate=0.1,
max_depth=3, random_state=42)

# Train the model
xgb_clf.fit(X_train, y_train)

# Predict on the test set
y_pred = xgb_clf.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

How it works:

- **Dataset**: We create a synthetic binary classification dataset.
- **Model**: `XGBClassifier` is used with parameters such as `n_estimators` (number of trees) and `learning_rate`.

- **Prediction and Evaluation**: The model is trained, and accuracy is calculated on the test set.

Key Features of XGBoost:

1. **Regularization**: L1 and L2 regularization to prevent overfitting.
2. **Sparsity Awareness**: Handles sparse data efficiently.
3. **Tree Pruning**: Uses a max-depth parameter and automatically prunes unnecessary branches to prevent overfitting.
4. **Weighted Quantile Sketch**: Helps find the best split for weighted datasets.
5. **Cross-validation**: Supports in-built cross-validation for hyperparameter tuning.

Pros of XGBoost:

1. **Highly Efficient and Fast**: Optimized for performance and speed using techniques like tree pruning, parallelism, and handling missing values.
2. **Great Predictive Power**: Delivers high accuracy for classification and regression tasks.
3. **Regularization**: Built-in regularization helps prevent overfitting, improving generalization.
4. **Scalable**: Can handle large datasets with millions of instances efficiently.
5. **Feature Importance**: Provides insights into the most important features.

Cons of XGBoost:

1. **Requires Careful Tuning**: To achieve optimal performance, hyperparameters like learning rate, depth, and number of trees need to be fine-tuned.
2. **Memory Intensive**: Large datasets can lead to high memory consumption.
3. **Long Training Time**: Though efficient, training time can still be longer compared to simpler models, especially if the dataset is very large.
4. **Complexity**: The algorithm can be harder to interpret than simple decision trees or linear models.

Conclusion:

XGBoost is a powerful and flexible boosting algorithm that has been widely adopted for machine learning tasks requiring high accuracy and performance. Its mathematical foundation is rooted in gradient boosting, but with enhancements like regularization, parallel processing, and improved handling of missing data. Proper hyperparameter tuning is critical to fully leverage its potential in both classification and regression tasks.

Import all the required frameworks

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

In [3]:

```
import xgboost as xgb
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegre
import warnings

# Ignore warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

Load the data and splitting it into train and test set

In [4]:

```
# Load the California Housing dataset
california_housing = fetch_california_housing()
X, y = california_housing.data, california_housing.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
```

In [5]:

```
print(california_housing)
```

```
{'data': array([[   8.3252    ,   41.        ,    6.98412698, ...,    2.55
555556,
          37.88      , -122.23      ],
       [   8.3014    ,   21.        ,    6.23813708, ...,    2.10984183,
          37.86      , -122.22      ],
       [   7.2574    ,   52.        ,    8.28813559, ...,    2.80225989,
          37.85      , -122.24      ],
       ...,
       [   1.7       ,   17.        ,    5.20554273, ...,    2.3256351 ,
          39.43      , -121.22      ],
       [   1.8672    ,   18.        ,    5.32951289, ...,    2.12320917,
          39.43      , -121.32      ],
       [   2.3886    ,   16.        ,    5.25471698, ...,    2.61698113,
          39.37      , -121.24      ]]), 'target': array([4.526, 3.585, 3.
521, ..., 0.923, 0.847, 0.894]), 'frame': None, 'target_names': ['MedHouse
Val'], 'feature_names': ['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'P
opulation', 'AveOccup', 'Latitude', 'Longitude'], 'DESCR': '.. _california
_housing_dataset:\n\nCalifornia Housing dataset\n--------------------------
-\n\n**Data Set Characteristics:**\n\n:Number of Instances: 20640\n\n:Numb
er of Attributes: 8 numeric, predictive attributes and the target\n\n:Attr
ibute Information:\n    - MedInc        median income in block group\n    
- HouseAge      median house age in block group\n    - AveRooms      avera
ge number of rooms per household\n    - AveBedrms     average number of be
drooms per household\n    - Population    block group population\n    - Av
eOccup      average number of household members\n    - Latitude      block
group latitude\n    - Longitude     block group longitude\n\n:Missing Attr
ibute Values: None\n\nThis dataset was obtained from the StatLib repositor
y.\nhttps://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html\n\nThe ta
rget variable is the median house value for California districts,\nexpress
ed in hundreds of thousands of dollars ($100,000).\n\nThis dataset was der
ived from the 1990 U.S. census, using one row per census\nblock group. A b
lock group is the smallest geographical unit for which the U.S.\nCensus Bu
reau publishes sample data (a block group typically has a population\nof 6
00 to 3,000 people).\n\nA household is a group of people residing within a
home. Since the average\nnumber of rooms and bedrooms in this dataset are 
provided per household, these\ncolumns may take surprisingly large values 
for block groups with few households\nand many empty houses, such as vacat
ion resorts.\n\nIt can be downloaded/loaded using the\n:func:`sklearn.data
sets.fetch_california_housing` function.\n\n.. rubric:: References\n\n- Pa
ce, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,\n  Statist
ics and Probability Letters, 33 (1997) 291-297\n'}
```

In [6]:

```
# Initialize the models
xgb_regressor = xgb.XGBRegressor(objective='reg:squarederror', random_sta
rf_regressor = RandomForestRegressor(random_state=42)
gb_regressor = GradientBoostingRegressor(random_state=42)

# Data Modeling
model_xgb = xgb_regressor.fit(X_train, y_train)
model_rf = rf_regressor.fit(X_train, y_train)
model_gb = gb_regressor.fit(X_train, y_train)

# Define parameter grids for each model
"""
param_grid_xgb = {
    'lambda': [0.01, 0.1, 1, 10],
    'gamma': [0, 0.1, 1, 10],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
    'n_estimators': [100, 200, 300]
```

```
}

param_grid_rf = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

param_grid_gb = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Set up GridSearchCV for each model
grid_search_xgb = GridSearchCV(estimator=xgb_regressor, param_grid=param_
grid_search_rf = GridSearchCV(estimator=rf_regressor, param_grid=param_gr
grid_search_gb = GridSearchCV(estimator=gb_regressor, param_grid=param_gr

# Fit the models
print("Tuning XGBoost...")
grid_search_xgb.fit(X_train, y_train)
print("Tuning RandomForest...")
grid_search_rf.fit(X_train, y_train)
print("Tuning GradientBoosting...")
grid_search_gb.fit(X_train, y_train)

# Best models
best_xgb = grid_search_xgb.best_estimator_
best_rf = grid_search_rf.best_estimator_
best_gb = grid_search_gb.best_estimator_


# Predictions
y_pred_xgb = best_xgb.predict(X_test)
y_pred_rf = best_rf.predict(X_test)
y_pred_gb = best_gb.predict(X_test)
"""
```

Out[6]:

```
'\nparam_grid_xgb = {\n    \'lambda\': [0.01, 0.1, 1, 10],\n    \'gamma
\': [0, 0.1, 1, 10],\n    \'max_depth\': [3, 5, 7],\n    \'learning_rate
\': [0.01, 0.1, 0.2],\n    \'n_estimators\': [100, 200, 300]\n}\n\nparam
_grid_rf = {\n    \'n_estimators\': [100, 200, 300],\n    \'max_depth\':
[None, 10, 20, 30],\n    \'min_samples_split\': [2, 5, 10],\n    \'min_s
amples_leaf\': [1, 2, 4]\n}\n\nparam_grid_gb = {\n    \'n_estimators\':
[100, 200, 300],\n    \'learning_rate\': [0.01, 0.1, 0.2],\n    \'max_de
pth\': [3, 5, 7],\n    \'min_samples_split\': [2, 5, 10],\n    \'min_sam
ples_leaf\': [1, 2, 4]\n}\n\n# Set up GridSearchCV for each model\ngrid_
search_xgb = GridSearchCV(estimator=xgb_regressor, param_grid=param_grid
_xgb, cv=5, scoring=\'neg_mean_squared_error\', verbose=1, n_jobs=-1)\ng
rid_search_rf = GridSearchCV(estimator=rf_regressor, param_grid=param_gr
id_rf, cv=5, scoring=\'neg_mean_squared_error\', verbose=1, n_jobs=-1)\n
grid_search_gb = GridSearchCV(estimator=gb_regressor, param_grid=param_g
rid_gb, cv=5, scoring=\'neg_mean_squared_error\', verbose=1, n_jobs=-1)
\n\n# Fit the models\nprint("Tuning XGBoost...")\ngrid_search_xgb.fit(X_
train, y_train)\nprint("Tuning RandomForest...")\ngrid_search_rf.fit(X_t
rain, y_train)\nprint("Tuning GradientBoosting...")\ngrid_search_gb.fit
(X_train, y_train)\n\n# Best models\nbest_xgb = grid_search_xgb.best_est
imator_\nbest_rf = grid_search_rf.best_estimator_\nbest_gb = grid_search
_gb.best_estimator_\n\n\n# Predictions\ny_pred_xgb = best_xgb.predict(X_
test)\ny_pred_rf = best_rf.predict(X_test)\ny_pred_gb = best_gb.predict
(X_test)\n'
```

Model Prediction

In [7]:

```
y_pred_xgb = model_xgb.predict(X_test)
y_pred_rf = model_rf.predict(X_test)
y_pred_gb = model_gb.predict(X_test)
```

Model Evaluation

In [8]:

```
# Calculate Mean Squared Error
mse_xgb = mean_squared_error(y_test, y_pred_xgb)
mse_rf = mean_squared_error(y_test, y_pred_rf)
mse_gb = mean_squared_error(y_test, y_pred_gb)

# Print the results
#print("Best parameters for XGBRegressor: ", grid_search_xgb.best_params_
print("Mean Squared Error for XGBRegressor: ", mse_xgb)

#print("Best parameters for RandomForestRegressor: ", grid_search_rf.best
print("Mean Squared Error for RandomForestRegressor: ", mse_rf)

#print("Best parameters for GradientBoostingRegressor: ", grid_search_gb.
print("Mean Squared Error for GradientBoostingRegressor: ", mse_gb)
```

```
Mean Squared Error for XGBRegressor:  0.2225899267544737
Mean Squared Error for RandomForestRegressor:  0.2557259876588585
Mean Squared Error for GradientBoostingRegressor:  0.2939973248643864
```