Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Intelligent Systems Lab

ENCS5141

Assignment #2:

Comprehensive Comparative Study Between Random Forest and XGBoost

---

Prepared by: Maysam Khatib 1190207.

Section: 3

Instructor: Dr. Mohammad Jubran.

TA: Eng. Hanan Awawdeh.

Date: 25/12/2023.

Abstract

This assignment presents a comprehensive comparative study between Random Forest and XGBoost, two popular ensemble learning techniques, across three scenarios reflecting real-world challenges: imbalanced classes, noisy data/features, and large datasets.

The study evaluates both the advantages and disadvantages of each algorithm through extensive evaluations of datasets in the three scenarios. An in-depth understanding is offered by hyperparameter tuning and metrics evaluations, which include accuracy, precision, recall, and F1 score. The assessment of computational efficiency, considering training time and memory utilization, is a critical part of the study, providing practitioners with vital insights for algorithm selection based on individual problem features and resource restrictions.

Table of Contents

Table of Figures

Table of Tables

Literature Review

For the two algorithms, we will explore the core principles including the training methodologies and key hyperparameters. Also, we will discuss the advantages and disadvantages of each one and highlight the differences between them.

1- Random Forest

Random forest is a popular machine learning technique developed by Leo Breiman and Adele Cutler that combines the output of numerous decision trees to produce a single result. Its ease of use and flexibility, as well as its ability to tackle classification and regression challenges, have boosted its popularity.[1]

- **Core Principles**

- **Ensemble Learning**: Using the Random Forest ensemble learning technique, several decision trees are constructed and then merged to get predictions that are more reliable and accurate.

- **Bagging**: Each tree is trained on a random subset of the training data with replacement using a method known as bagging (Bootstrap Aggregating).

- **Feature Randomization**: To reduce overfitting and introduce unpredictability, each tree in the forest is trained using a distinct subset of features.

- **Training Methodology**

Random Forest works in two stages, first creating an ensemble of N decision trees and then using these trees to make predictions. The following are the operational steps:

Step 1: Select K data points randomly from the training set.

Step 2: Build decision trees using the selected data points (subsets).

Step 3: Enter the required number, N, of decision trees to be constructed.

Step 4: Repeat Steps 1 and 2 iteratively.

Step 5: Determine predictions from each decision tree for new data points and assign the new data points to the category with the most votes.

[1] What is Random Forest? | IBM

This procedure, as depicted in the following diagram, emphasizes the randomness and variety introduced during the training phase, resulting in a strong and accurate predictive model in the succeeding prediction phase.[2]

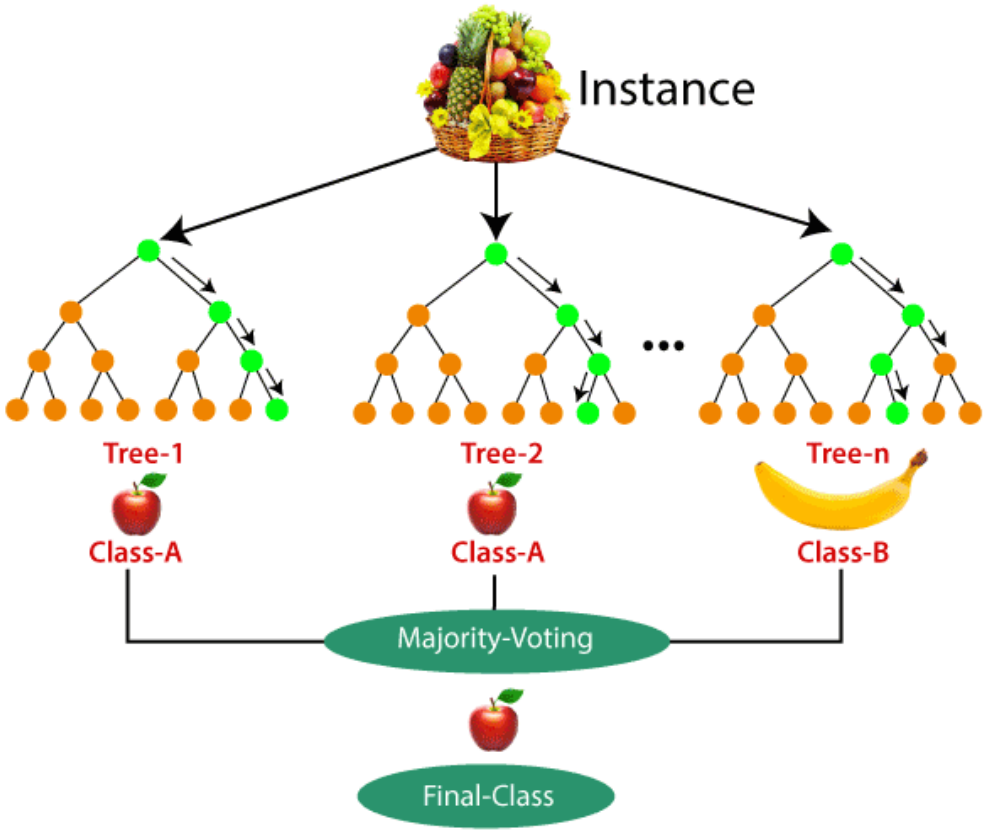

Figure 1: Random Forest Methodology

[2] Machine Learning Random Forest Algorithm - Javatpoint

- **Key Hyperparameters**

Table 1: Random Forest Key Hyperparameters

Hyperparameter	Definition	Recommendation
n_estimators	Number of trees in the forest	Higher values for better performance, but watch for computational complexity
max_depth	Maximum depth of the decision trees	Tune to prevent overfitting
min_samples_split	The minimum number of samples required to split an internal node	Higher values to prevent overfitting
min_samples_leaf	The minimum number of samples required to be at a leaf node	Higher values to control overfitting
max_features	Maximum number of features to consider for splitting a node	"sqrt" (square root of total features) is common
bootstrap	Whether to use bootstrap samples when building trees	Set to True for introducing randomness
criterion	The function used to measure the quality of a split	"gini" or "entropy" based on the problem
random_state	Seed for random number generation ensures reproducibility	Set for reproducibility

The most effective values for these hyperparameters may differ based on the specifics of the dataset and the type of problem to solve. It's common practice to undertake hyperparameter tuning using approaches like grid search or random search to find the optimal combination for the chosen dataset.

- **Advantages and Limitations**

Table 2: Advantages & Limitations of Random Forest

Advantages	Limitations
Sturdy against overfitting	May incur high computational costs
Good results with default hyperparameters	Might not perform well on imbalanced datasets
Suitable for both classification and regression	Limited interpretability with increasing complexity

2- XGBoost

Extreme Gradient Boosting, or XGBoost, is a machine learning algorithm that has gained popularity and widespread usage because it can handle large datasets and achieve state-of-the-art performance in many machine learning tasks, including regression and classification. It's an ensemble learning strategy that combines the predictions of several weak models to get a more accurate forecast.[3]

- **Core Principles**

- **Boosting**: The XGBoost boosting method sequentially constructs trees, fixing the mistakes of the preceding tree as it goes.

- **Gradient Boosting**: By improving the model's parameters using gradient descent, it minimizes a loss function.

- **Regularization**: To avoid overfitting, XGBoost incorporates regularization terms into the goal function.

- **Training Methodology**

Step 1: Initialize the model by setting hyperparameters such as learning rate, maximum tree depth, and the number of boosting rounds.

Step 2: Create an initial prediction using a simple model (e.g., mean for regression).

Step 3: Compute residuals by finding the difference between actual and predicted values.

Step 4: Build a weak model (decision tree) on the residuals with constraints to avoid overfitting.

Step 5: Update predictions by adding the scaled predictions of the weak model to the initial predictions, controlled by the learning rate.

Step 6: Compute new residuals based on the updated predictions.

Step 7: Repeat Steps 4-6 iteratively to add more weak models until reaching the specified number of boosting rounds or meeting a stopping criterion.

Step 8: Finalize the model by combining predictions from all weak models.

[3] XGBoost - GeeksforGeeks

- **Key Hyperparameters**

Table 3: XGBoost Key Hyperparameters

Hyperparameter	Definition	Recommendation
learning_rate	Step size shrinkage to prevent overfitting.	Typically [0.01, 0.3]. Lower values make the algorithm more robust.
n_estimators	Number of trees (boosting rounds) to be run.	Higher values improve performance, tune with learning_rate
max_depth	Maximum depth of a tree. Controls the complexity of the weak learners.	Tune to prevent overfitting, common values: 3-10
min_child_weight	The minimum sum of instance weight (hessian) needed in a child. Controls over-fitting.	Higher values provide more conservative models, often set between 1-10
subsample	Fraction of training data to be used for each boosting round. Controls overfitting.	Typically set between 0.5 and 1
colsample_bytree (or others)	Fraction of features to be randomly sampled for building each tree.	Values between 0.5 and 1, introduce randomness
gamma (min_split_loss)	Minimum loss reduction is required to make a further partition on a leaf node.	Higher values result in fewer splits, common values include 0, 1, 5
lambda (reg_lambda)	L2 regularization term on weights. Controls regularization.	Values between 0 and 3 are common
alpha (reg_alpha)	L1 regularization term on weights. Controls regularization.	Values between 0 and 1 are common
scale_pos_weight	Controls the balance of positive and negative weights, useful for imbalanced classes.	Adjust based on the class imbalance
objective	Specifies the learning task and corresponding objective function.	Choose based on the problem (e.g., "reg:squarederror", "binary:logistic")

These hyperparameters offer a variety of control over how the XGBoost model behaves, and the optimal values for them can change based on the dataset and the problem.

- **Advantages and limitations**

Table 4: Advantages & Limitations of XGBoost

Advantages	Limitations
Very accurate prediction	Highly sensitive to hyperparameter tuning
Effectively manages missing data	Training takes longer time than with Random Forest
Works well with a wide range of datasets	May not perform well with small datasets

3- Key differences

- **Ensemble vs. Boosting:** XGBoost is a boosting technique that creates trees sequentially, whereas Random Forest is an ensemble method that builds trees independently.
- **Regularization:** Whereas Random Forest depends on feature randomization, XGBoost uses regularization to balance model complexity.
- **Learning Rate Optimization:** XGBoost performs finer-grained optimization than Random Forest since it employs a learning rate to regulate each tree's contribution.

Table 5 shows some other key differences between the two algorithms.

Table 5: Key Differences

Feature	Random Forest	XGBoost
Ensemble Method	Bagging (Parallel Training)	Boosting (Sequential Training)
Base Learners	Independent Decision Trees	Sequentially Built Decision Trees
Regularization	Feature Randomization	L1 and L2 Regularization Terms
Handling Imbalance	May struggle with highly imbalanced data	Can handle imbalanced data through weights
Sensitivity to Hyperparameters	Generally, less sensitive	More sensitive, and requires careful tuning
Performance	Robust and good default performance	Higher predictive accuracy in many cases
Training Time	Generally faster due to parallelization	May require more time due to sequential training
Interpretability	Easier to interpret individual trees	Slightly more complex due to boosting
Handling Missing Data	Can handle missing data well	Can handle missing data well
Use Cases	Generally good for diverse datasets	Often used when high predictive accuracy is crucial

Priorities of determining which of Random Forest and XGBoost to use:

- Interpretability and speed: When interpretability is critical and you want quick insights into your data, go for Random Forest.
- High accuracy and complex relationships: When handling nonlinear relationships and optimizing accuracy are critical, choose XGBoost.

In the end, it's critical to compare and test the two algorithms to determine which one works best for your set of data and prediction objectives.

Scenarios Designs and Analysis

- Used Functions

Preparing Data Function

The prepare data function is written to preprocess the data and prepare it for the training models, the steps to prepare the data are:

1- Handling missing values by replacing them with mean values for numerical columns, and with the most frequent value (mode) for categorical columns.
2- Dividing the data to features and target, the target column name is passed with the function.
3- Splitting the data into training and testing sets, with a test size of 30% from the whole set.
4- Encoding the categorical columns to be numerical (for model specifications).
5- Standardizing the numerical features to prevent dominance by features with larger numerical values.

The function returns the preprocessed features X_train and X_test, and the target y_train, and y_test. The code is shown below:

```
import pandas as pd

import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder

def prepare_data(data, target_column):

  # If column is categorical, replace with the most frequent value, else
with the mean
  for col in data:
    if data[col].dtype == 'object':
        data[col] = data[col].fillna(data[col].mode().iloc[0])
    else:
        data[col] = data[col].fillna(data[col].mean())

  # Divide the data to features and target
  X = data.drop(target_column, axis=1)
  y = data[target_column]

  # Split the data into training and testing set
```

```
  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

  # Encode categorical columns in features using label encoder
  label_encoder = LabelEncoder()
  categorical_columns = X.select_dtypes(include=['object']).columns

  for column in categorical_columns:
      X_train[column] = label_encoder.fit_transform(X_train[column])
      X_test[column] = label_encoder.transform(X_test[column])

  # Encode the target variable using label encoder
  y_encoder = LabelEncoder()
  y_train = y_encoder.fit_transform(y_train)
  y_test = y_encoder.transform(y_test)

  # Standardize features
  numerical_columns = X_train.select_dtypes(include=['number']).columns
  scaler = StandardScaler()
  X_train[numerical_columns] = scaler.fit_transform(X_train)
  X_test[numerical_columns] = scaler.transform(X_test)

  return X_train, X_test, y_train, y_test
```

Training and Evaluating Random Forest Classifier Function

This function is used for training and evaluating the random forest classifier, the steps are:

1- Creating a random forest model.
2- Defining the hyperparameters for grid search, these parameters will be investigated during the grid search, the parameters are max_depth, min_samples_leaf, min_samples_split, and n_estimators.
3- Performing the grid search on the hyperparameters.
4- Training the best model obtained from the grid search on X_train and y_train.
5- Using the model to predict y for the X_test
6- Evaluating model performance.

This function returns the evaluation metrics that will be used later to plot the results.

The code is as follows:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score
import time
```

```
import memory_profiler

def train_and_evaluate_rfc(X_train, X_test, y_train, y_test):

  # Define the random forest model
  random_forest_model = RandomForestClassifier(random_state=42)

  # Define hyperparameter grid
  param_grid = {
      'max_depth': [5, 10, 15],
      'min_samples_leaf': [1, 2, 4],
      'min_samples_split': [2, 5, 10],
      'n_estimators': [50, 100, 200, 250]
  }

  # Perform Grid Search on the hyperparameters
  t_start = time.time()
  grid_search = GridSearchCV(random_forest_model, param_grid, cv=5)
  grid_search.fit(X_train, y_train)
  t_end = time.time()

  # Get the best model based on the grid search
  best_model = grid_search.best_estimator_
  best_model.fit(X_train, y_train)

  # Predict y for X_test
  y_pred = best_model.predict(X_test)

  best_params = grid_search.best_params_
  print(f'The bast hyperparameters for random forest
classifier:\n{best_params}')

  best_accuracy = grid_search.best_score_
  print(f'The best accuracy for random forest
classifier:\n{best_accuracy}')

  print('****Evaluation metrics:****')

  test_accuracy = accuracy_score(y_test, y_pred)
  print(f'The test accuracy for random forest
classifier:\n{test_accuracy}')

  test_precision = precision_score(y_test, y_pred, average='weighted')
  print(f'The test precision for random forest
classifier:\n{test_precision}')
```

```
  test_recall = recall_score(y_test, y_pred, average='weighted')
  print(f'The test recall for random forest classifier:\n{test_recall}')

  test_f1 = f1_score(y_test, y_pred, average='weighted')
  print(f'The test f1 score for random forest classifier:\n{test_f1}')

  print(f'Training Time for random forest classifier:\n{t_end - t_start}
seconds')
  print(f'Memory Usage for random forest
classifier:\n{memory_profiler.memory_usage()[0]} MB\n')

  results = {'accuracy': test_accuracy, 'precision': test_precision,
'recall': test_recall, 'f1_score': test_f1}

  return results
```

Training and Evaluating XGBoost Function

This function is used for training and evaluating the XGBoost model. The steps are similar to the previous function with some modifications on the hyperparameters as seen in the code below.

```
from xgboost import XGBClassifier

def train_and_evaluate_xgb(X_train, X_test, y_train, y_test):

  # Define the XGBoost classifier
  xgb_model = XGBClassifier(random_state=42)

  # Define hyperparameter grid
  param_grid = {
      'n_estimators': [50, 100, 200, 250],
      'max_depth': [5, 10, 15],
      'learning_rate': [0.05, 0.1, 0.2],
      'subsample': [0.8, 0.9, 1.0],
  }

  # Perform Grid Search on the hyperparameters
  t_start = time.time()
  grid_search = GridSearchCV(xgb_model, param_grid, cv=5)
  grid_search.fit(X_train, y_train)
  t_end = time.time()

  # Get the best model based on the grid search
  best_model = grid_search.best_estimator_
```

```
  best_model.fit(X_train, y_train)

  # Predict y for X_test
  y_pred = best_model.predict(X_test)

  best_params = grid_search.best_params_
  print(f'The bast hyperparameters for XGBoost:\n{best_params}')

  best_accuracy = grid_search.best_score_
  print(f'The best accuracy for XGBoost:\n{best_accuracy}')

  print('****Evaluation metrics:****')

  test_accuracy = accuracy_score(y_test, y_pred)
  print(f'The test accuracy for XGBoost:\n{test_accuracy}')

  test_precision = precision_score(y_test, y_pred, average='weighted')
  print(f'The test precision for XGBoost:\n{test_precision}')

  test_recall = recall_score(y_test, y_pred, average='weighted')
  print(f'The test recall for XGBoost:\n{test_recall}')

  test_f1 = f1_score(y_test, y_pred, average='weighted')
  print(f'The test f1 score for XGBoost:\n{test_f1}')

  print(f'Training Time XGBoost:\n{t_end - t_start} seconds')
  print(f'Memory Usage XGBoost:\n{memory_profiler.memory_usage()[0]} MB')

  results = {'accuracy': test_accuracy, 'precision': test_precision,
'recall': test_recall, 'f1_score': test_f1}

  return results
```

Plotting Results Function

This function is used to plot the evaluation metrics for both models in the same dataset by using the matplot library.

```
import matplotlib.pyplot as plt


def plot_comparison(rfc_metrics, XGBoost_metrics):

  metrics = ['accuracy', 'precision', 'recall', 'f1_score']
  metrics_n = 4
```

```
  index = np.arange(metrics_n)

  fig, ax = plt.subplots()

  # Plot Random Forest results
  rfc_values = [rfc_metrics[i] for i in metrics]
  rfc_bars = ax.bar(index, rfc_values, 0.35, label='Random Forest
Classifier')

  # Plot XGBoost results
  xgb_values = [XGBoost_metrics[i] for i in metrics]
  xgb_bars = ax.bar(index + 0.35, xgb_values, 0.35, label='XGBoost')

  # Configure the plot
  ax.set_xlabel('Metrics')
  ax.set_ylabel('Scores')
  ax.set_title('Comparison of Random Forest and XGBoost Results')
  ax.set_xticks(index + 0.35 / 2)
  ax.set_xticklabels(metrics)
  ax.legend()

  # Show the plot
  plt.show()
```

- ## Scenario 1: Imbalanced Classes

Imbalanced classes relate to a condition in a classification issue in which the distribution of classes is not equal, implying that one class greatly outnumbers the other. In simple terms, the dataset has an unequal number of cases for each type. Typically, one class, known as the majority class, has far more instances than the other, identified as the minority class. The objective of this scenario is to see which algorithm is better in this case.

➢ Dataset

The chosen dataset is the **Parkinson's Disease dataset** that obtained from the UCI Machine Learning Repository. Features include various speech signal processing measures, and the target variable is 'status' indicating the presence or absence of Parkinson's disease (1 for presence, 0 for absence). This dataset has a majority class which is when status equals 1.

At first, the dataset has been loaded and prepared for the training models.

```
# Load the data from the website
url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/parkinsons/parkinsons.data"
parkinson_data = pd.read_csv(url, header=0)
```

```
# Remove the name column because its not needed
parkinson_data = parkinson_data.drop('name', axis=1)

# Print the number of status when its 1 and when its 0
status_counts = parkinson_data['status'].value_counts()

print("Count when status is 0:", status_counts[0])
print("Count when status is 1:", status_counts[1])

# Prepare and split the data
X_train, X_test, y_train, y_test = prepare_data(parkinson_data, 'status')

parkinson_data.info()
```

As shown in the output below, the dataset contains 195 entries with 147 instances of the majority class (status = 1) and 48 instances of the minority class (status = 0).

```
Count when status is 0: 48
Count when status is 1: 147
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 195 entries, 0 to 194
Data columns (total 23 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   MDVP:Fo(Hz)       195 non-null    float64
 1   MDVP:Fhi(Hz)      195 non-null    float64
 2   MDVP:Flo(Hz)      195 non-null    float64
 3   MDVP:Jitter(%)    195 non-null    float64
 4   MDVP:Jitter(Abs)  195 non-null    float64
 5   MDVP:RAP          195 non-null    float64
 6   MDVP:PPQ          195 non-null    float64
 7   Jitter:DDP        195 non-null    float64
 8   MDVP:Shimmer      195 non-null    float64
 9   MDVP:Shimmer(dB)  195 non-null    float64
 10  Shimmer:APQ3      195 non-null    float64
 11  Shimmer:APQ5      195 non-null    float64
 12  MDVP:APQ          195 non-null    float64
 13  Shimmer:DDA       195 non-null    float64
 14  NHR               195 non-null    float64
 15  HNR               195 non-null    float64
 16  status            195 non-null    int64
 17  RPDE              195 non-null    float64
 18  DFA               195 non-null    float64
 19  spread1           195 non-null    float64
 20  spread2           195 non-null    float64
 21  D2                195 non-null    float64
 22  PPE               195 non-null    float64
dtypes: float64(22), int64(1)
memory usage: 35.2 KB
```

Figure 2: Parkinson Dataset Info

➢ Training and evaluation in both models

The preprocessed data was used in both functions clarified above.

```
print('Impalanced classes: Random Forest Classifier')
rfc_results = train_and_evaluate_rfc(X_train, X_test, y_train, y_test)

print('##################################################################
#########\n')

print('Impalanced classes: XGBoost')
XGBoost_results = train_and_evaluate_xgb(X_train, X_test, y_train, y_test)
```

- Random forest results:

```
Impalanced classes: Random Forest Classifier
The bast hyperparameters for random forest classifier:
{'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 50}
The best accuracy for random forest classifier:
0.9187830687830688
****Evaluation metrics:****
The test accuracy for random forest classifier:
0.9322033898305084
The test precision for random forest classifier:
0.9318065869281786
The test recall for random forest classifier:
0.9322033898305084
The test f1 score for random forest classifier:
0.9305353779930052
Training Time for random forest classifier:
97.111257314682 seconds
Memory Usage for random forest classifier:
975.578125 MB
```

Figure 3: Random Forest: Imbalanced Classes Results

- Random forest results:

```
Impalanced classes: XGBoost
The bast hyperparameters for XGBoost:
{'learning_rate': 0.05, 'max_depth': 5, 'n_estimators': 250, 'subsample': 0.9}
The best accuracy for XGBoost:
0.9261904761904762
****Evaluation metrics:****
The test accuracy for XGBoost:
0.9322033898305084
The test precision for XGBoost:
0.9378531073446327
The test recall for XGBoost:
0.9322033898305084
The test f1 score for XGBoost:
0.9284621053228275
Training Time XGBoost:
26.262768983840942 seconds
Memory Usage XGBoost:
975.578125 MB
```

Figure 4: XGBoost: Imbalanced Classes Results

```
# Plot the results
plot_comparison(rfc_results, XGBoost_results)
```

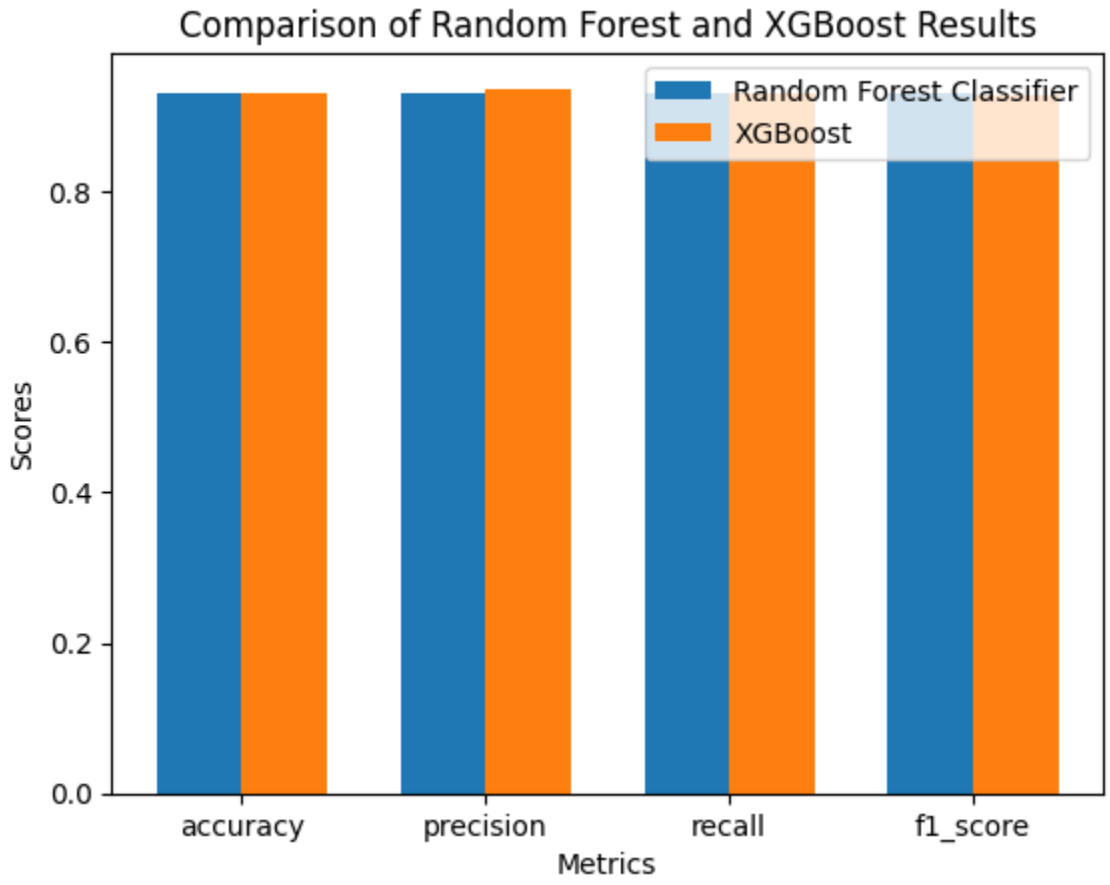

Figure 5: Comparison of Random Forest and XGBoost Results: Imbalanced Classes

We can conclude from the results above that:

- On the imbalanced dataset, both models perform well, reaching high accuracy and displaying a balanced trade-off between precision and recall.
- XGBoost shows slightly better precision, and recall. The random forest has the highest F1 score.
- XGBoost took less training time than Random Forest, which is supposed to be the opposite. Mostly this happened because of the hyperparameter settings.
- The memory usage is comparable between the two algorithms.

So, in scenarios with imbalanced classes, the choice between Random Forest and XGBoost depends on the specific requirements.

- ## Scenario 2: Noisy Data/Features

Noisy data is defined as data including errors, outliers, or irrelevant information that might introduce mistakes or mislead analysis and modeling operations. Data noise can come from a variety of sources, and it may reduce the effectiveness of machine learning models and statistical studies. The objective of this scenario is to see which algorithm is better in the case of outliers and noisy data.

➢ Dataset

The chosen dataset is the **heart disease dataset** that was obtained from the UCI Machine Learning Repository. Features include various clinical measurements, and the target variable is 'target' indicating the degree of presence or absence of heart disease.

At first, the data was loaded.

```
# Load the data from url
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data"

column_names = [
    'age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg',
    'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'target'
]

heart_disease_data = pd.read_csv(url, names=column_names, na_values="?")


# Calculate the percentage of rows that has outliers
numerical_columns =
heart_disease_data.select_dtypes(include=['float64']).columns
outliers = heart_disease_data[numerical_columns].apply(lambda x: x[(x -
x.mean()).abs() > 2.5 * x.std()])
rows_with_outliers = outliers[~outliers.isnull().all(axis=1)].shape[0]

# Display the result
print(f"Percentage of outliers
{(rows_with_outliers/heart_disease_data.shape[0])*100}%")

X_train, X_test, y_train, y_test = prepare_data(heart_disease_data,
'target')

heart_disease_data.info()
```

The output was as shown:

```
Percentage of outliers 6.9306930693069315%
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       303 non-null    float64
 1   sex       303 non-null    float64
 2   cp        303 non-null    float64
 3   trestbps  303 non-null    float64
 4   chol      303 non-null    float64
 5   fbs       303 non-null    float64
 6   restecg   303 non-null    float64
 7   thalach   303 non-null    float64
 8   exang     303 non-null    float64
 9   oldpeak   303 non-null    float64
 10  slope     303 non-null    float64
 11  ca        303 non-null    float64
 12  thal      303 non-null    float64
 13  target    303 non-null    int64
dtypes: float64(13), int64(1)
memory usage: 33.3 KB
```

Figure 6: Heart Disease Dataset Info

As seen above, the dataset contains 303 entries with 14 features, while the percentage of outliers in the numerical columns is approximately 6.93%.

➢ Training and evaluation in both models

The preprocessed data was used in both functions clarified above.

```
print('Noisy Data: Random Forest Classifier')
rfc_results = train_and_evaluate_rfc(X_train, X_test, y_train, y_test)

print('##########################################################################
#########\n')

print('Noisy Data: XGBoost')
XGBoost_results = train_and_evaluate_xgb(X_train, X_test, y_train, y_test)
```

- Random Forest results:

```
Noisy Data: Random Forest Classifier
The bast hyperparameters for random forest classifier:
{'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
The best accuracy for random forest classifier:
0.6129568106312292
****Evaluation metrics:****
The test accuracy for random forest classifier:
0.5494505494505495
The test precision for random forest classifier:
0.4475908706677937
The test recall for random forest classifier:
0.5494505494505495
The test f1 score for random forest classifier:
0.48562389361901115
Training Time for random forest classifier:
107.22732138633728 seconds
Memory Usage for random forest classifier:
975.578125 MB
```

Figure 7: Random Forest: Noisy Data Results

- XGBoost results:

```
Noisy Data: XGBoost
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Undef
  _warn_prf(average, modifier, msg_start, len(result))
The bast hyperparameters for XGBoost:
{'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 50, 'subsample': 0.8}
The best accuracy for XGBoost:
0.6228128460686599
****Evaluation metrics:****
The test accuracy for XGBoost:
0.5164835164835165
The test precision for XGBoost:
0.44897959183673475
The test recall for XGBoost:
0.5164835164835165
The test f1 score for XGBoost:
0.4801352493660186
Training Time XGBoost:
67.06549549102783 seconds
Memory Usage XGBoost:
975.578125 MB
```

Figure 8: XGBoost: Noisy Data Results

```
# Plot the results
plot_comparison(rfc_results, XGBoost_results)
```

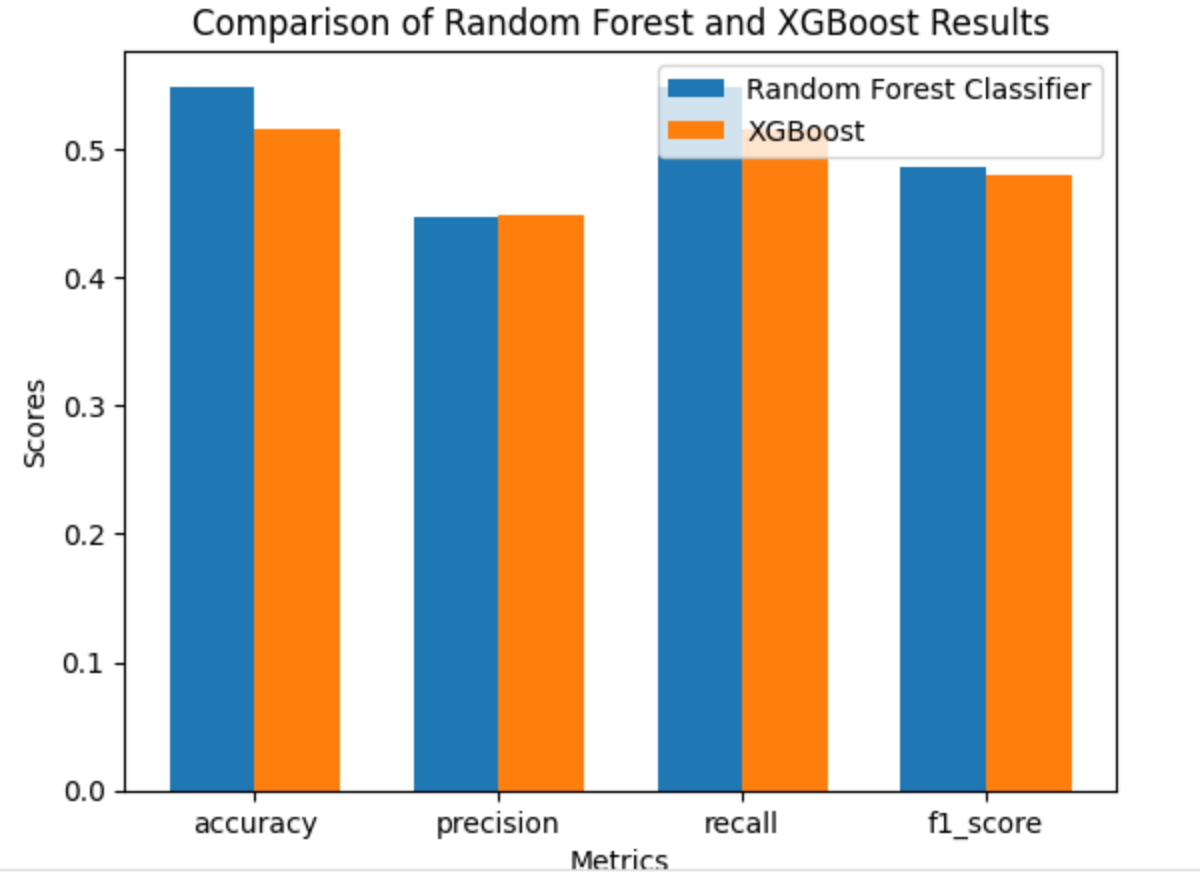

Figure 9: Comparison of Random Forest and XGBoost Results: Noisy Data

We can conclude from the results above that:

- On the noisy data, both models struggle to achieve good accuracy which makes sense given the difficulties that errors and outliers provide.
- Random Forest shows slightly better accuracy, recall, and F1 score. While XGBoost has better precision.
- XGBoost took less training time than Random Forest, which is supposed to be the opposite. Mostly this happened because of the hyperparameter settings.
- The memory usage is comparable between the two algorithms.

So, in scenarios with noisy data and outliers, Random Forests may perform effectively when the data has a moderate number of outliers or errors. Noise can be reduced by averaging the effects of many trees. Furthermore, preprocessing methods such as outlier removal and feature engineering can significantly improve model performance on noisy data.

- ## Scenario 3: Large Dataset

The term "large dataset" is largely relative and might vary depending on the topic or application. A huge dataset is defined as having a large volume of data that may be difficult to handle or process using typical methods. A dataset's size is frequently expressed in terms of the number of records (rows) and features (columns).

➢ Dataset

The chosen dataset is the **Adult Income dataset** that was obtained from the UCI Machine Learning Repository. Features include various demographic and employment-related features, and the target variable is 'income,' which indicates if an individual makes more than $50,000 per year.

At first, the data was loaded.

```
# Load the data from the website
url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/adult/adult.data"
column_names = [
    'age', 'workclass', 'fnlwgt', 'education', 'education-num', 'marital-
status',
    'occupation', 'relationship', 'race', 'sex', 'capital-gain', 'capital-
loss',
    'hours-per-week', 'native-country', 'income'
]

incomes_data = pd.read_csv(url, header=None, names=column_names,
na_values=' ?', skipinitialspace=True)
print(f"The number of rows in the incomes dataset:
{incomes_data.shape[0]}")

X_train, X_test, y_train, y_test = prepare_data(incomes_data, 'income')

incomes_data.info()
```

As shown in the output below, the dataset contains 32,561 entries with 15 features.

```
The number of rows in the incomes dataset: 32561
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32561 entries, 0 to 32560
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             32561 non-null  int64
 1   workclass       32561 non-null  object
 2   fnlwgt          32561 non-null  int64
 3   education       32561 non-null  object
 4   education-num   32561 non-null  int64
 5   marital-status  32561 non-null  object
 6   occupation      32561 non-null  object
 7   relationship    32561 non-null  object
 8   race            32561 non-null  object
 9   sex             32561 non-null  object
 10  capital-gain    32561 non-null  int64
 11  capital-loss    32561 non-null  int64
 12  hours-per-week  32561 non-null  int64
 13  native-country  32561 non-null  object
 14  income          32561 non-null  object
dtypes: int64(6), object(9)
memory usage: 3.7+ MB
```

Figure 10: Adult Income Dataset Info

➢ Training and evaluation in both models

The preprocessed data was used in both functions clarified above.

```
print('Large Dataset: Random Forest Classifier')
rfc_results = train_and_evaluate_rfc(X_train, X_test, y_train, y_test)

print('##################################################################
#########')

print('Large Dataset: XGBoost')
XGBoost_results = train_and_evaluate_xgb(X_train, X_test, y_train, y_test)
```

- Random Forest results:

```
Large Dataset: Random Forest Classifier
The bast hyperparameters for random forest classifier:
{'max_depth': 15, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 250}
The best accuracy for random forest classifier:
0.8639437145144241
****Evaluation metrics:****
The test accuracy for random forest classifier:
0.8645716040536391
The test precision for random forest classifier:
0.8588297698294555
The test recall for random forest classifier:
0.8645716040536391
The test f1 score for random forest classifier:
0.8586092478613121
Training Time for random forest classifier:
870.6161205768585 seconds
Memory Usage for random forest classifier:
975.578125 MB
```

Figure 11: Random Forest: Large Data Results

- XGBoost results:

```
Large Dataset: XGBoost
The bast hyperparameters for XGBoost:
{'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 250, 'subsample': 1.0}
The best accuracy for XGBoost:
0.8729379349932112
****Evaluation metrics:****
The test accuracy for XGBoost:
0.8736820554816256
The test precision for XGBoost:
0.8694769179025768
The test recall for XGBoost:
0.8736820554816256
The test f1 score for XGBoost:
0.8703643218985504
Training Time XGBoost:
262.00363850593567 seconds
Memory Usage XGBoost:
975.578125 MB
```

Figure 12: XGBoost: Large Data Results

```
# Plot the results
plot_comparison(rfc_results, XGBoost_results)
```

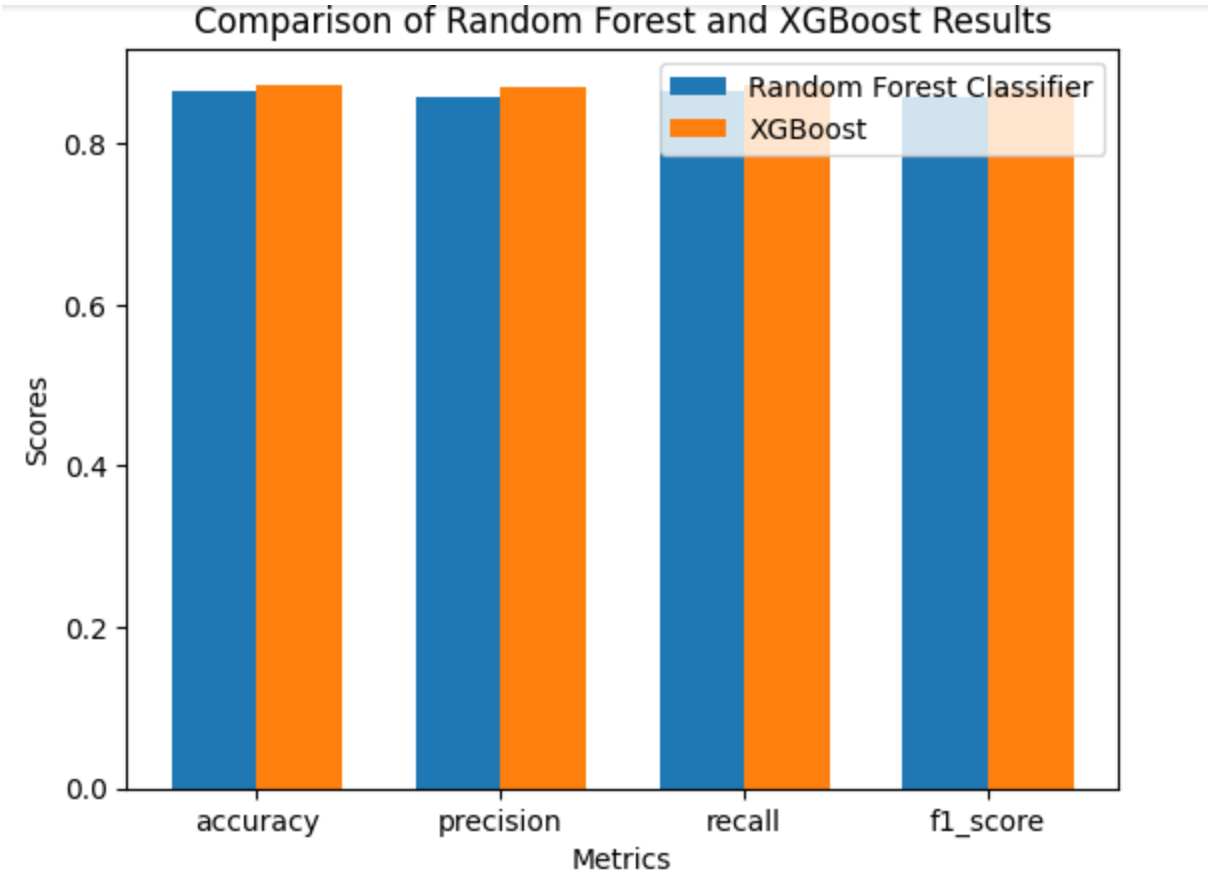

Figure 13: Comparison of Random Forest and XGBoost Results: Large Data

We can conclude from the results above that:

- On the large data, both Random Forest and XGBoost perform well, with XGBoost slightly outperforming Random Forest.
- XGBoost shows slightly better accuracy, recall, precision, and F1 score.
- XGBoost took less training time than Random Forest, which is supposed to be the opposite. Mostly this happened because of the hyperparameter settings.
- The memory usage is comparable between the two algorithms.

So, in scenarios with large data, XGBoost may perform more effectively than Random Forest. XGBoost's capacity to efficiently handle huge datasets makes it a competitor, for instance when computational resources are limited.

Conclusion

Random Forest and XGBoost have specific strengths and considerations in different scenarios. Both strategies perform well in situations where classes are balanced. Random Forest is distinguished by its simplicity and robustness. The algorithm's default hyperparameters frequently produce good results, making it a practical alternative for a wide range of applications. XGBoost, on the other hand, excels when precision and accuracy are required, particularly in circumstances with imbalanced classes. Despite the running time, XGBoost's ability to manage class imbalances makes it a better choice when accuracy is the primary goal and computational resources allow for slightly longer training periods.

When dealing with noisy data, Random Forest's tolerance to noise and outliers makes it acceptable for datasets with moderate noise levels. Its ensemble approach helps to reduce the impact of noise, resulting in more reliable predictions.

The choice between Random Forest and XGBoost for large datasets is based on the trade-off between interpretability and processing performance. While Random Forest's ensemble nature provides a more interpretable model, XGBoost's efficiency in handling huge datasets makes it the preferred alternative when processing time is a requirement. The decision should be guided by project requirements, with Random Forest being preferred for interpretability and XGBoost being preferred for scalability and efficiency.

In conclusion, choosing between Random Forest and XGBoost should be based on careful analysis of unique project requirements, dataset features, and available computational resources. It is recommended that the person tests both algorithms, fine-tune the hyperparameters, and confirms their performance on the target dataset. The final decision will be based on the desired balance of interpretability, accuracy, and computational efficiency for the given situation.