Assignment-4: Decision Tree and XGBoost

Zarin Akter (2011704042), Rashik Iram Chowdhury (2111336642) and Md. Mutasim Farhan (2013123642)
Emails: {zarin.akter@northsouth.edu}, {rashik.chowdhury@northsouth.edu}, {mutasim.farhan@northsouth.edu}

April 14, 2024

I. Introduction

In this assignment, we explore Decision Trees and XGBoost to tackle various classification and regression tasks. Decision Trees and XGBoost are powerful machine learning algorithms known for their versatility and effectiveness across different datasets. Decision Tree is a supervised learning technique used to classify and regression problems. It is a tree-structured classifier where internal nodes represent the features of a dataset, branches represent the decision rules, and each leaf node represents the outcome. XGBoost, or eXtreme Gradient Boosting, is a machine learning algorithm under ensemble learning. It is trendy for supervised learning tasks, such as regression and classification. XGBoost builds a predictive model by combining the predictions of multiple individual models, often decision trees, in an iterative manner. We aim to evaluate and compare the performance of these algorithms using different evaluation metrics on binary classification, multi-class classification, and regression problems using four diverse datasets. The datasets selected for this analysis cover a range of domains, including car evaluation, brain cancer detection, wage prediction, and credit risk assessment. Hyper-parameter tuning uses the validation dataset to select optimal settings for both Decision Trees and XGBoost models. Following hyper-parameter optimization, we report the evaluation metrics for the test dataset, providing a comprehensive assessment of the performance of Decision Trees and XGBoost across different datasets and tasks. The paper is structured into three main sections. Section [2] outlines the methodology for training the Decision Trees and XGBoost models, including data preprocessing, hyper-parameter tuning, and model evaluation. In Section [3], we present the experimental results and analyze the performance of the models across various datasets and tasks. Finally, Section [4] offers conclusions drawn from the findings and discusses the implications for practical applications.

II. Method

In this section, we sequentially implement the experiments with Decision Trees and XGBoost classifiers on four datasets: Multi-class classification (Car Evaluation), Binary classification (Brain Cancer), and two Regression tasks (Wage and Credit). The datasets are split into training, validation, and test sets using a 70:15:15 ratio dataset. Hyperparameters for both models are optimized using the validation set. Evaluation metrics, including accuracy, confusion matrix, precision, recall, F-score, the precision-recall curve for binary classification, and average precision, recall, F-score, and mean-squared error for multi-class classification and regression, are computed, which can be seen in section 4. Fig. 1 shows the workflow of this study.

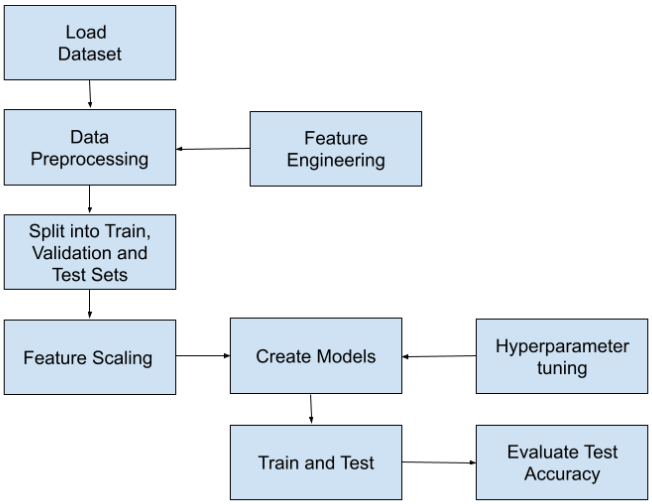

Fig. 1. Methodology Block Diagram

- **Load Dataset**

A. Multi-Class Classification

The dataset was obtained from the UCI Machine Learning Repository using the `fetch_ucirepo` function with the ID set to 19, corresponding to the Car Evaluation Dataset. The following code was used to fetch the dataset and load the features into `X` and the target into `y`:

```
car_evaluation = fetch_ucirepo(id=19)
X = car_evaluation.data.features
y = car_evaluation.data.targets
```

B. Binary Classification

The dataset used for Binary classification is DT-BrainCancer, which is in CSV format. The dataset is loaded using pandas.

```
import pandas as pd
import numpy as np
df = pd.read_csv("G:\\Assignment\\DT-BrainCancer.csv")
```

C. Linear Regression

i) The DT-Credit dataset is used for Linear Regression. It is in CSV format. The dataset is loaded using pandas.

```
import pandas as pd
import numpy as np
df = pd.read_csv("G:\\Assignment\\DT-Credit.csv")
```

ii) The DT-Wage dataset is also used for Linear Regression, which is loaded using pandas library.

```
import pandas as pd
import numpy as np
df = pd.read_csv("G:\\Assignment\\DT-Wage.csv")
```

- **Data Preprocessing:**

A. Multi-Class Classification

*1) **Decision Tree**:* After loading the dataset and splitting it into X (features) and y (target), X and y are further preprocessed by converting the categorical values into numerical values using Label Encoding. This is because the Decision Tree expects both features and target to be in numerical format.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
X.loc[:, 'buying'] = le.fit_transform(X['buying'])
X.loc[:, 'maint'] = le.fit_transform(X['maint'])
X.loc[:, 'doors'] = le.fit_transform(X['doors'])
X.loc[:, 'persons'] = le.fit_transform(X['persons'])
X.loc[:, 'lug_boot'] = le.fit_transform(X['lug_boot'])
X.loc[:, 'safety'] = le.fit_transform(X['safety'])
y.loc[:, 'class'] = le.fit_transform(y['class'])
```

*2) **XGBoost**:* A label encoder was used to transform the 'class' column of the targets into integer values for use in the XGBoost model. Finally, the targets were formatted into a data frame and converted to integers:

```
y.loc[:, 'class'] = le.fit_transform(y['class'])
y = pd.DataFrame(y, columns=["class"])
y = y.astype(int)
```

Now, we handle the categorical features. First, we extract the names of all columns in the dataset 'X' that are not numerical (i.e., categorical features) and store them in a list called 'cats.' Then, we loop over each of these categorical columns and convert their datatype to 'category' using the 'astype('category')' function. This is because XGBoost can work on categorical data.

```
cats = X.select_dtypes(exclude=np.number).columns.tolist()
for col in cats:
   X[col] = X[col].astype('category')
```

B. Binary Classification

1) **Decision Tree:** After loading the data frame, we found a missing value in the diagnosis column, for which that corresponding row or instance was dropped entirely. Then, 3 features - sex, diagnosis, and loc were converted to numerical values using Label Encoder. Then, the corresponding X and y are assigned to the features and target (status,) respectively.

```
df.dropna(subset=['diagnosis'], inplace=True)
df.isnull().sum()

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()for col in cats:
df['sex'] = le.fit_transform(df['sex'])
df['diagnosis'] = le.fit_transform(df['diagnosis'])
df['loc'] = le.fit_transform(df['loc'])

X = df.drop(columns=['status'])
y = df['status']
```

2) **XGBoost:** Similarly, at first, the missing value instance was dropped. Then, the target column 'status' is dropped from the dataset 'df' to create the feature matrix 'X.' The 'status' column itself forms the target vector 'y'.

Next, the names of the categorical features in 'X' are extracted and stored in the 'cats' list. All these categorical features are then converted into the 'category' datatype using the 'astype' function, as XGBoost can handle categorical data. This completes the preprocessing stage for the binary classification task.

```
df.dropna(subset=['diagnosis'], inplace=True)
df.isnull().sum()

X, y = df.drop("status", axis=1), df[['status']]
cats = X.select_dtypes(exclude=np.number).columns.tolist()
for col in cats:
   X[col] = X[col].astype('category')
```

C. Linear Regression

1) **Decision Tree:**

- For the Credit dataset, 4 features - Own, Student, Married, and Region are converted to numerical values using Label Encoder. Using a correlation heatmap, we found that feature Own is weakly correlated. As a result, we dropped this column feature and processed X and y values by keeping and dropping target (Balance), respectively.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df['Own'] = le.fit_transform(df['Own'])
df['Student'] = le.fit_transform(df['Student'])
df['Married'] = le.fit_transform(df['Married'])
df['Region'] = le.fit_transform(df['Region'])

X = df.drop(columns=['Balance', 'Own'])
y = df['Balance']
```

- For the Wage dataset, 7 features - maritl, race, education, region, jobclass, health, and health_ins are converted to numerical values using Label Encoder. Based on the correlation heatmap, we found that the region has no correlation with other features. As a result, it was dropped, and the remaining features were used to create X and y sets by selecting and dropping the target wage.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df['maritl'] = le.fit_transform(df['maritl'])
df['race'] = le.fit_transform(df['race'])
```

```
df['education'] = le.fit_transform(df['education'])
df['region'] = le.fit_transform(df['region'])
df['jobclass'] = le.fit_transform(df['jobclass'])
df['health'] = le.fit_transform(df['health'])
df['health_ins'] = le.fit_transform(df['health_ins'])

X = df.drop(columns=['wage', 'region'])
y = df['wage']
```

2) **XGBoost:** The same preprocessing steps used for the binary classification task are applied for the linear regression tasks.

i) Credit Dataset

```
X = df.drop(columns=['Balance', 'Own'])
y = df['Balance']

cats = X.select_dtypes(exclude=np.number).columns.tolist()

for col in cats:
    X[col] = X[col].astype('category')
```

ii) Wage Dataset

```
X = df.drop('wage', axis=1)
y = df['wage']

cats = X.select_dtypes(exclude=np.number).columns.tolist()

for col in cats:
    X[col] = X[col].astype('category')
```

- **Split into Train and Validation, and Test Sets:** We split the entire data into 70% train, 15% validation, and 15% test sets.This is applicable to all the datasets used.

```
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,
random_state=1)
X_val, X_test, y_valid, y_test = train_test_split(X_temp, y_temp, test_size=0.5,
random_state=1)
```

- **Feature Scaling** Standard Scalar is used to normalize the data. It was only used for Decision Tree problems. The code is applicable to all decision tree classifications for the four datasets.

```
from sklearn.preprocessing import StandardScaler
scaler= StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.fit_transform(X_valid)
X_test = scaler.fit_transform(X_test)
```

- **Hyperparameter Tuning, Create Model and Training and Testing:**

A. Multi-class classification:

1) **Decision Tree:**

After splitting into train, valid, and test sets, a decision tree model is created using a Decision Tree Classifier, where the best hyperparameters - maximum depth and minimum samples split are tuned by iteratively changing these parameters from 2 to 200 to find the best accuracy for the validation dataset. The best-optimized hyperparameters are then used to select the best model on which the unseen test dataset is used for evaluation.

```
from sklearn.tree import DecisionTreeClassifier
import numpy as np

def evaluate_DecisionTree(maximum_depth, minimum_samples_split):
  model = DecisionTreeClassifier(max_depth=maximum_depth, min_samples_split=minimum_
  samples_split)
```

```
  model.fit(X_train, y_train)

  y_valid_predict = model.predict(X_valid)
  accuracy = np.mean(y_valid_predict == y_valid)
  return accuracy,  model

best_accuracy = 0.0
best_depth = None
best_min_samples = None
best_model = None

for depth in range(2, 200):
    for min_samples in range(2, 200):
        accuracy, model = evaluate_DecisionTree(depth, min_samples)
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_depth = depth
            best_min_samples = min_samples
            best_model = model

print(f"Best␣Hyperparameters:␣max_depth={best_depth},␣min_samples_split=
{best_min_samples}")
print(f"Best␣Validation␣Accuracy:␣{round((best_accuracy*100),2)}")
y_test_predict = best_model.predict(X_test)
test_accuracy = np.mean(y_test_predict == y_test)
print(f"Test␣Accuracy:␣{round((test_accuracy*100),2)}")
```

2) **XGBoost:** A dictionary, hyperparameters_to_tune, is defined with various values for the max_depth, learning_rate, min_child_weight, and gamma hyperparameters of the XGBoost model. The max_depth hyperparameter controls the maximum depth of the trees, learning_rate shrinks the feature weights to make the boosting process more conservative, min_child_weight defines the minimum sum of instance weight needed in a child, and gamma is the minimum loss reduction required to make a split. To perform an exhaustive search, GridSearchCV is used to determine the best hyperparameters using 5-fold cross-validation and accuracy as the scoring method. The GridSearchCV object then fits with the validation data, effectively training the XGBoost model with all combinations of hyperparameters and finding the combination that gives the highest accuracy. Lastly, the XGBoost classifier is trained using the best hyperparameters obtained from the previous grid search, and this model is trained on the training data set. Once trained, the model predicts the test data set to evaluate its performance.

```
hyperparameters_to_tune = {
    "max_depth": [3, 5, 7],
    "learning_rate": [0.001, 0.01, 0.1, 0.3],
    "min_child_weight": [1, 3, 5, 7],
    "gamma": [0, 0.1, 0.2, 0,3],
}
from sklearn.model_selection import GridSearchCV

xgb_model = xgb.XGBClassifier(objective="multi:softprob", tree_method="gpu_hist",
enable_categorical = True)

grid_search = GridSearchCV(estimator=xgb_model, param_grid=hyperparameters_to_tune,
cv=5, scoring="accuracy")
grid_search.fit(X_val, y_valid)
best_params = grid_search.best_params_
print("Best␣Hyperparameters:", best_params)

best_model = xgb.XGBClassifier(objective="multi:softprob", tree_method="gpu_hist",
**best_params, enable_categorical = True)
best_model.fit(X_train, y_train)
y_pred = best_model.predict(X_test)
```

B. Binary Classification

1) **Decision Tree:** The same method as discussed for Multi-class classification using a decision tree has been applied for Binary classification.

2) **XGBoost:** The hyperparameters for the binary classification task were similarly tuned using GridSearchCV on the validation set with a range of values for max_depth, learning_rate, min_child_weight, and gamma. The best parameters were then used to train an XGBoost model with the objective set to "binary: logistic." The trained model was subsequently used to make predictions on the test set.

```
hyperparameters_to_tune = {
    "max_depth": [3, 5, 7],
    "learning_rate": [0.001, 0.01, 0.1, 0.3],
    "min_child_weight": [1, 3, 5, 7],
    "gamma": [0, 0.1, 0.2, 0,3],
}

from sklearn.model_selection import GridSearchCV
xgb_model = xgb.XGBClassifier(objective="binary:logistic", tree_method="gpu_hist",
enable_categorical = True)
grid_search = GridSearchCV(estimator=xgb_model, param_grid=hyperparameters_to_tune,
cv=5, scoring="accuracy")
grid_search.fit(X_val, y_valid)
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)
best_model = xgb.XGBClassifier(objective="binary:logistic", tree_method="gpu_hist",
**best_params, enable_categorical = True)
best_model.fit(X_train, y_train)

y_pred = best_model.predict(X_test)
```

C. Linear Regression

1) **Decision Tree:**

i) For the Credit Dataset, a decision tree model is created using a Decision Tree Regressor, where the best hyperparameters - maximum depth and minimum samples split are tuned by iteratively changing these parameters from 2 to 200 to find the best MSE (mean squared error) for the validation dataset. The best-optimized hyperparameters are then used to select the best model on which the unseen test dataset is used for evaluation.

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
def evaluate_DecisionTree(maximum_depth, minimum_samples_split):
  model = DecisionTreeRegressor(max_depth=maximum_depth,
min_samples_split=minimum_samples_split)
  model.fit(X_train, y_train)

  y_valid_predict = model.predict(X_valid)
  MSE_valid = calculate_mse(y_valid, y_valid_predict)
  y_test_predict = model.predict(X_test)
  MSE_test = calculate_mse(y_test, y_test_predict)
  return MSE_valid, MSE_test

best_mse = float("inf")
best_depth = None
best_min_samples = None
best_test = float("inf")
for depth in range(2, 200):  # Adjust the range as needed
  for min_samples in range(2, 200):  # Adjust the range as needed
    mse, mse_test = evaluate_DecisionTree(depth, min_samples)
    if mse < best_mse:
      best_mse = mse
      best_depth = depth
```

```
      best_test = mse_test
      best_min_samples = min_samples

print(f"Best␣Hyperparameters:␣max_depth={best_depth},␣min_samples_split=
{best_min_samples}")
print(f"Best␣Validation␣MSE:␣{best_mse}")
print(f"Mean␣Squared␣Error␣of␣Test␣Dataset:␣{best_test}")
```

ii) The same method has been applied to Wage Dataset.

2) **XGBoost:** For the linear regression task, hyperparameters were tuned manually by iterating over a predefined range of values for max_depth, learning_rate, subsample, and colsample_bytree. An XGBoost model was trained for each hyperparameter combination, and its performance was evaluated based on Mean Squared Error (MSE) on the validation set. The combination of hyperparameters that resulted in the model with the lowest MSE on the validation set was selected as the best parameter. These parameters were then used to train the final model and assess its performance on the test set.

i) For Credit Dataset:

```
params = {"objective": "reg:squarederror", "tree_method": "gpu_hist"}

hyperparams = {
    "max_depth": [3, 5, 7],
    "learning_rate": [0.1, 0.01, 0.001],
    "subsample": [0.7, 0.8, 0.9],
    "colsample_bytree": [0.7, 0.8, 0.9]
}

best_mse = float("inf")
best_params = None
best_test_mse = float("inf")
for max_depth in hyperparams["max_depth"]:
    for learning_rate in hyperparams["learning_rate"]:
      for subsample in hyperparams["subsample"]:
        for colsample_bytree in hyperparams["colsample_bytree"]:
          params["max_depth"] = max_depth
          params["learning_rate"] = learning_rate
          params["subsample"] = subsample

          params["colsample_bytree"] = colsample_bytree

          model = xgb.train(params=params, dtrain=dtrain_reg, num_boost_round=100)

          preds_valid = model.predict(dval_reg)
             val_mse = calculate_mse(y_valid, preds_valid)
             preds_test = model.predict(dtest_reg)
             test_mse = calculate_mse(y_test, preds_test)

               if val_mse < best_mse:
                  best_mse = val_mse
                  best_params = params.copy()
                  best_test_mse = test_mse
print("Best␣Parameters:␣Maximum␣Depth␣is␣", best_params["max_depth"], ",
Learning␣Rate␣is␣", best_params["learning_rate"], ",␣Subsample␣is␣",
best_params["subsample"], ",␣Colsample␣by␣tree␣is␣", best_params["colsample_bytree"])
print(f"Best␣Validation␣MSE:␣{best_mse}")
print(f"Mean␣Squared␣Error␣on␣Test␣Set:␣{best_test_mse}")
```

ii) For Wage Dataset:

```
params = {"objective": "reg:squarederror", "tree_method": "gpu_hist"}

hyperparams = {
    "max_depth": [3, 5, 7],
    "learning_rate": [0.1, 0.01, 0.001],
    "subsample": [0.7, 0.8, 0.9],
    "colsample_bytree": [0.7, 0.8, 0.9]
}

best_mse = float("inf")
best_params = None
best_test_mse = float("inf")
for max_depth in hyperparams["max_depth"]:
    for learning_rate in hyperparams["learning_rate"]:
       for subsample in hyperparams["subsample"]:
          for colsample_bytree in hyperparams["colsample_bytree"]:
            params["max_depth"] = max_depth
            params["learning_rate"] = learning_rate
            params["subsample"] = subsample

            params["colsample_bytree"] = colsample_bytree

            model = xgb.train(params=params, dtrain=dtrain_reg, num_boost_round=100)

            preds_valid = model.predict(dval_reg)
            val_mse = calculate_mse(y_valid, preds_valid)
            preds_test = model.predict(dtest_reg)
            test_mse = calculate_mse(y_test, preds_test)

            if val_mse < best_mse:
              best_mse = val_mse
              best_params = params.copy()
              best_test_mse = test_mse
print("Best␣Parameters:␣Maximum␣Depth␣is␣", best_params["max_depth"], ",
Learning␣Rate␣is␣", best_params["learning_rate"], ",␣Subsample␣is␣",
best_params["subsample"], ",␣Colsample␣by␣tree␣is␣", best_params["colsample_bytree"])
print(f"Best␣Validation␣MSE:␣{best_mse}")
print(f"Mean␣Squared␣Error␣on␣Test␣Set:␣{best_test_mse}")
```

D. Evaluate Test Accuracy:

In this section, we present test accuracy evaluation for the six trained models using the accuracy metrics. The performance of each model is assessed on an unseen test dataset, and a comparative analysis is conducted to determine the effectiveness of the models in recognizing handwritten digits. By following this comprehensive methodology, we aimed to systematically develop, evaluate, and refine neural network models for handwritten digit recognition, ultimately achieving accurate and robust performance across diverse datasets and application scenarios.

The evaluation metrics employed for binary classification include accuracy, confusion matrix, precision, recall, F-score, and precision-recall curve. We assess accuracy, confusion matrix, average precision, average recall, and average F-score for multi-class classification. In regression tasks, we measure performance using mean squared error.

- **Confusion Matrix**

```
confusion_matrix = np.zeros((len(np.unique(y_test.values)), len(np.unique
(y_test.values))))
for i in range(len(y_test)):
    confusion_matrix[y_test.values[i], y_pred[i]] += 1
print("Confusion␣Matrix:")
```

```
print(confusion_matrix)

import matplotlib.pyplot as plt
import seaborn as sns

# Plot Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix, annot=True, cmap='Blues', fmt=".0f")
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

- **Accuracy**

In classification tasks, accuracy measures the percentage of correctly predicted instances out of all instances in the dataset. It is calculated as the number of correct predictions divided by the total number of predictions.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \times 100\% \tag{1}$$

- **Precision**

Precision is a metric that measures the accuracy of positive predictions made by a model. Precision is calculated as the ratio of true positive predictions to the total number of positive predictions made by the model.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{2}$$

- **Recall**

Recall, also known as sensitivity or true positive rate, is a metric to measure the model's ability to identify all relevant instances from a dataset correctly. It quantifies the proportion of actual positive instances that were correctly predicted by the model. Recall is calculated as the ratio of true positive predictions to the total number of actual positive instances in the dataset.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3}$$

- **F1-Score**

The F1 score is a metric used to evaluate the model's accuracy, balancing precision, and recall. It considers false positives and negatives, making it a useful metric when the classes are imbalanced. The F1 score is calculated as the harmonic mean of precision and recall.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

- **Confusion Matrix**

A confusion matrix, also known as an error matrix, is a table that is used to evaluate the performance of a classification model. It presents a summary of the model's predictions against the actual outcomes across different classes in the dataset.

```
accuracy = np.mean(y_pred == y_test.values)
print("Accuracy:", accuracy)
```

```
precision_per_class = []
recall_per_class = []
fscore_per_class = []

for class_label in np.unique(y_test_values_reshaped):
    true_positives = np.sum((y_test_values_reshaped == class_label) &
    (y_pred == class_label))
    false_positives = np.sum((y_test_values_reshaped != class_label) &
    (y_pred == class_label))
    false_negatives = np.sum((y_test_values_reshaped == class_label) &
```

```
    (y_pred != class_label))

    precision = true_positives / (true_positives + false_positives)
    if (true_positives + false_positives) > 0 else 0
    recall = true_positives / (true_positives + false_negatives)
    if (true_positives + false_negatives) > 0 else 0
    fscore = 2 * precision * recall / (precision + recall)
    if (precision + recall) > 0 else 0

    precision_per_class.append(precision)
    recall_per_class.append(recall)
    fscore_per_class.append(fscore)

# Compute average precision, recall, and F-score across all classes
average_precision = np.mean(precision_per_class)
average_recall = np.mean(recall_per_class)
average_fscore = np.mean(fscore_per_class)

print(f"Average Precision: {average_precision}")
print(f"Average Recall: {average_recall}")
print(f"Average F-score: {average_fscore}")
```

III. Experiment Results

A. Data Analysis

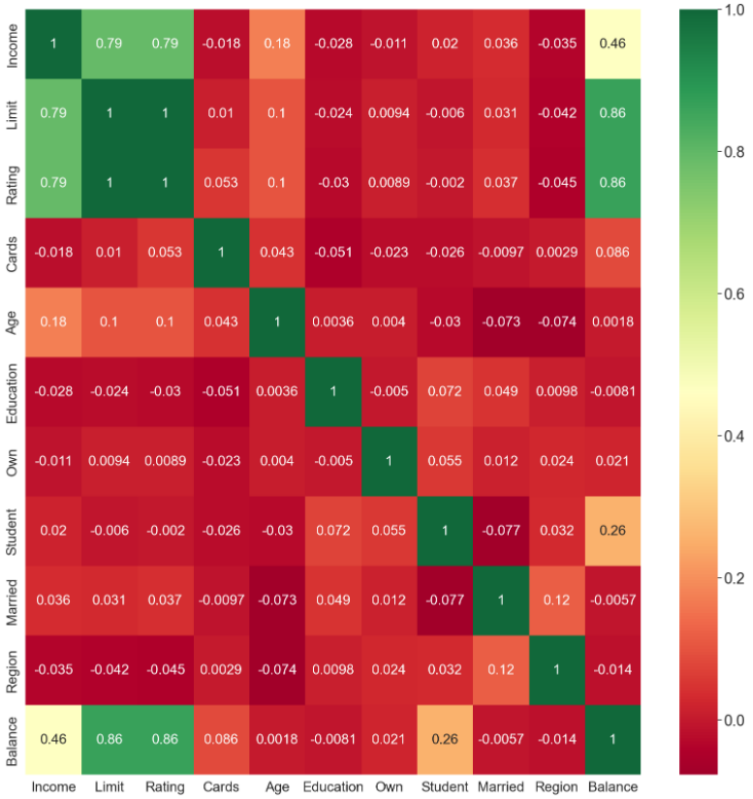

Fig. 2. Heat map of Regression Dataset DT-Credit

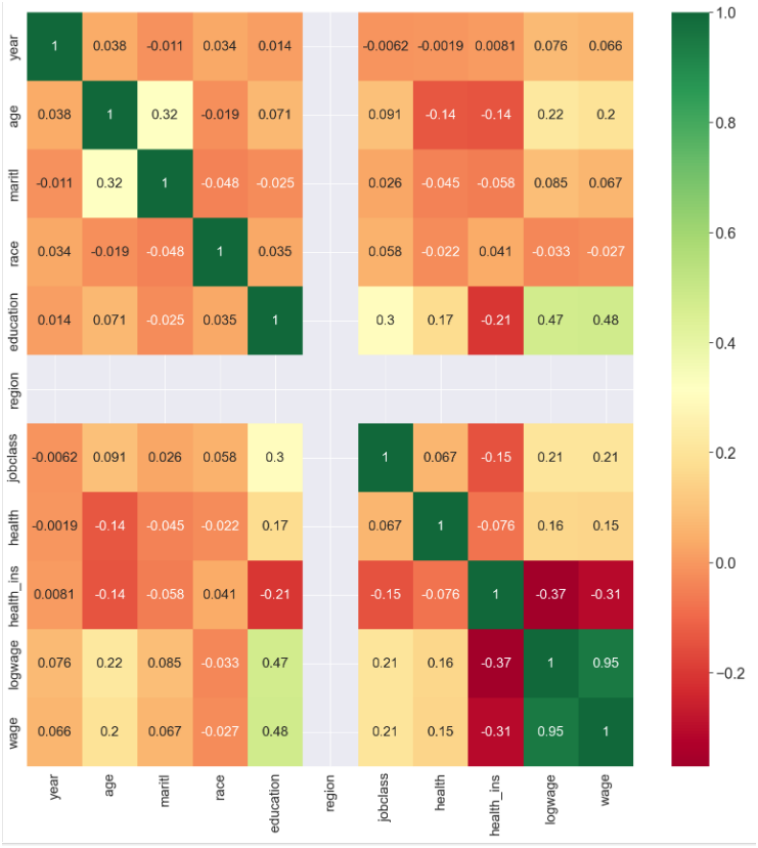

Fig. 3. Heat map of Regression Dataset DT-Wage

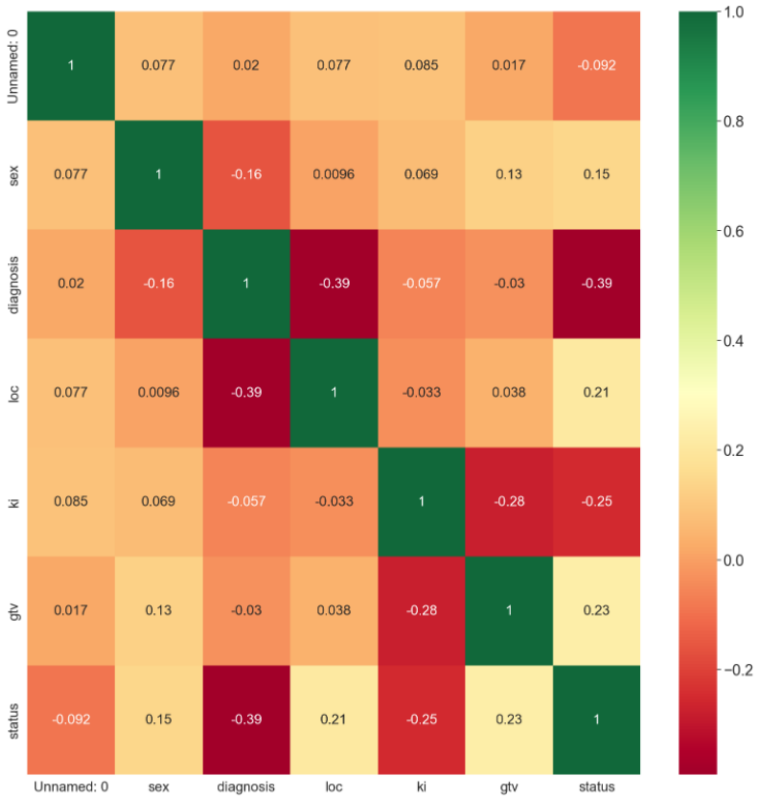

Fig. 4. Heat map of Binary Classification Dataset DT-BrainCancer

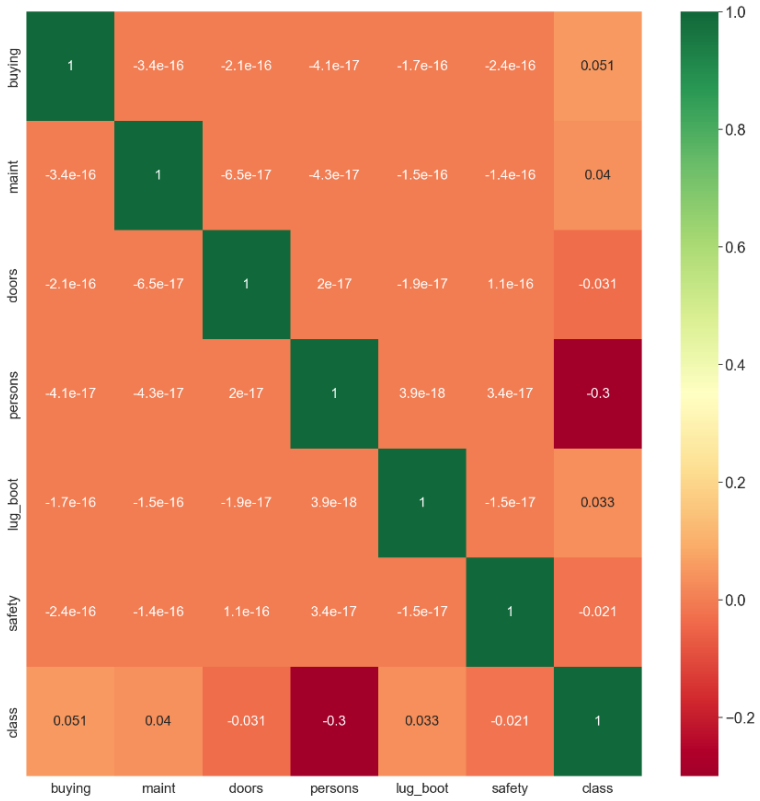

Fig. 5. Heat map of Multi-class classification Dataset

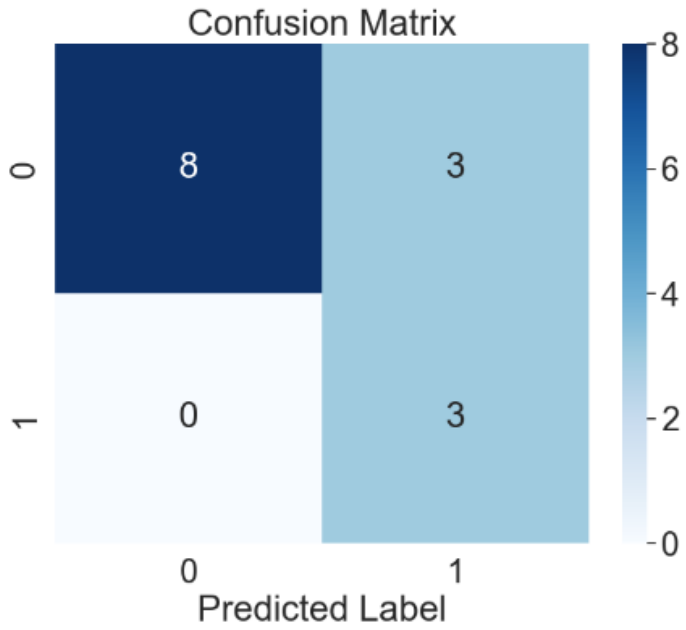

Fig. 6. Confusion matrix of decision tree on Brain Cancer dataset (Binary Classification)

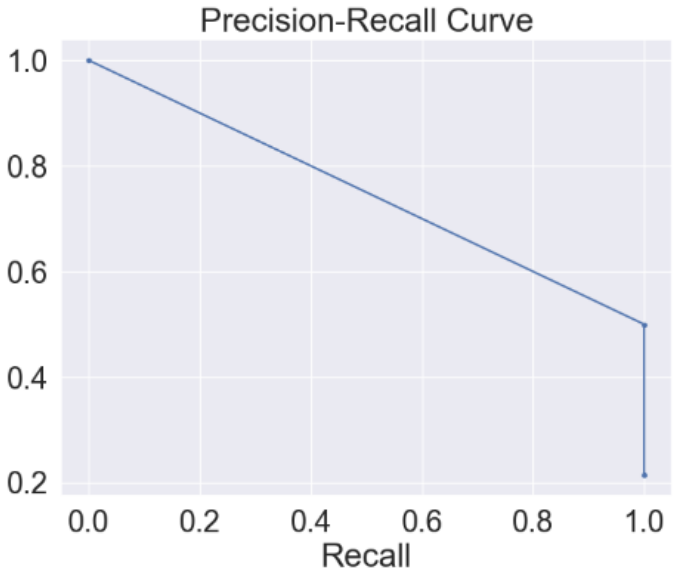

Fig. 7. Precision recall curve of decision tree on Brain Cancer dataset (Binary Classification)

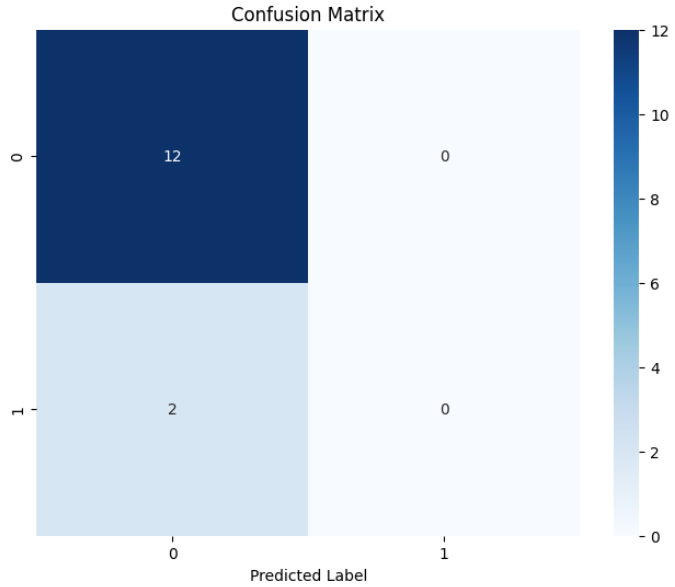

Fig. 8. Confusion matrix of XGBoost on Brain Cancer dataset (Binary Classification)

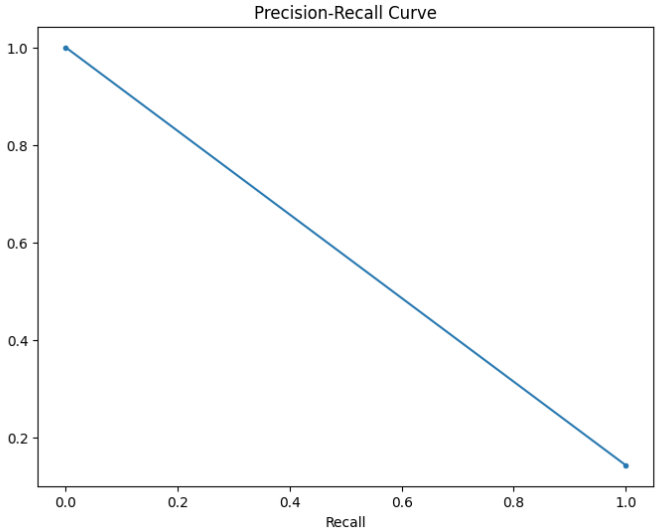

Fig. 9. Precision recall curve of XGBoost on Brain Cancer dataset (Binary Classification)

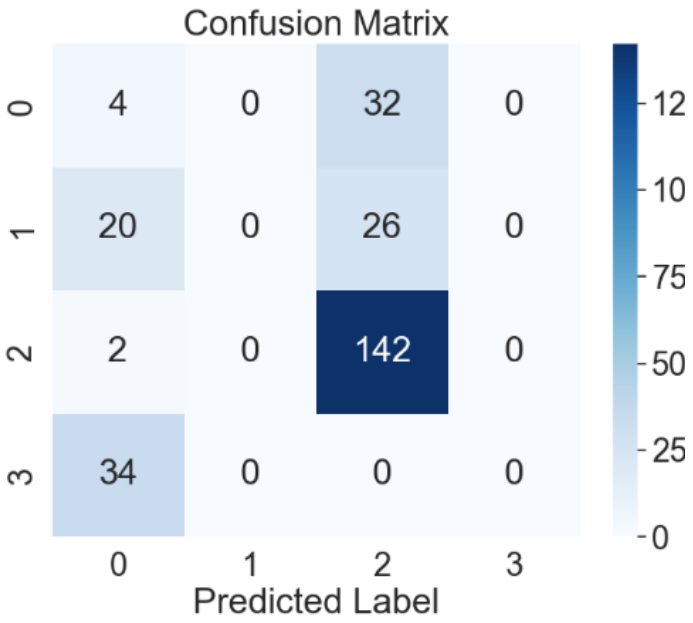

Fig. 10. Confusion matrix of decision tree (Multi-class classification)

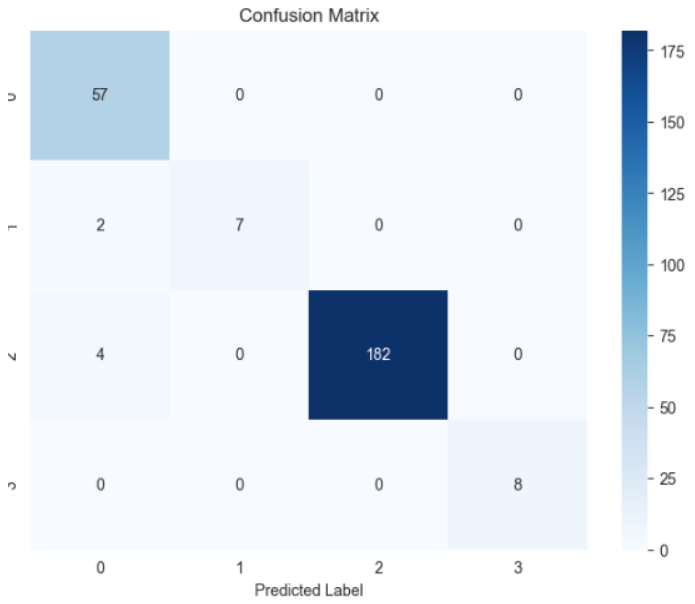

Fig. 11. Confusion matrix of XGBoost (Multi-class classification)

TABLE I
MULTI-CLASS CLASSIFICATION FOR BOTH DECISION TREE AND XGBOOST

Multi-class classification							
Decision tree				XGBoost			
Test Accuracy	Average Precision	Average Recall	Average F1-Score	Test Accuracy	Average Precision	Average Recall	Average F1-Score
56.15%	0.19	0.27	0.23	97.69%	0.98	0.94	0.95

TABLE II
BINARY CLASSIFICATION FOR BOTH DECISION TREE AND XGBOOST

Binary classification							
Decision tree				XGBoost			
Test Accuracy	Precision	Recall	F1-Score	Test Accuracy	Precision	Recall	F1-Score
78.57%	0.75	0.86	0.75	85.71%	0.85	0.5	0.92

TABLE III
MSE CALCULATION FOR LINEAR REGRESSION (DT-CREDIT)

Linear Regression (DT-Credit)	
MSE	
Decision Tree	XGBoost
23985.6	10286.08

TABLE IV
MSE CALCULATION FOR LINEAR REGRESSION (DT-WAGE)

Linear Regression (DT-Wage)	
MSE	
Decision Tree	**XGBoost**
7.058	6.588

IV. DISCUSSION

In this paper, Decision Tree and XGBoost classifiers are fine-tuned and applied to four distinguishing datasets comprising classification and regression problems. The validation datasets play a significant role in finding the best hyperparameters to build the optimized model. This model is then evaluated using the unseen test dataset.

Among both Binary and Multi Class Classification problems, we can see that XGBoost performed better, with around 86% and 98% accuracy, respectively, compared to Decision Tree. It is also important to note that XGBoost demonstrated superior Precision, Recall, and harmonic mean (F1-Score) comparatively. Notably, in the case of multi-class classification on the car evaluation dataset, we can see that classes 1 and 3 are not predicted in the confusion matrix. This highlights that the decision tree model is not adequately trained in all classes. This might be due to the lack of training samples, which results in poor accuracy overall. In the case of Regression on both datasets, XGBoost showed lower MSE (mean squared error), 6.588 and 10286.08 on the Wage and Credit dataset, respectively, compared to Decision Tree, which engendered 7.058 and 23985.6, respectively. However, both demonstrated high MSE in the DT-Credit dataset. This is most likely because the model during training has been overfitted. This error could be reduced significantly by increasing or augmenting existing dataset samples.

In conclusion, XGBoost performed better than a single decision tree because of its gradient-boosting technique, which encompasses ensemble learning based on sequential weak learners.

REFERENCES

Bohanec, Marko. (1997). Car Evaluation. UCI Machine Learning Repository. https://doi.org/10.24432/C5JP48.