Assignment-4: Decision Trees and XGBoost

Sajan Kumer Sarker
Id: 2111131642
Email: sajan.sarker@northsouth.edu

Rafsan Jani Chowdhury
Id: 2011424642
Email: rafsan.chowdhury@northsouth.edu

Rosely Mohammad
Id: 2014219642
Email: rosely.mohammad@northsouth.edu

November 9, 2024

1. Introduction

In this Assignment report on Decision Tree and XGBoost we will discuss about training and evaluation of the Decision Tree model and XGBoost model on four different datasets: 'Car Evaluation (for Multi-class Classification)', 'DT-BrainCancer (for Binary Classification)', 'DT-Wage (for Regression)', 'DT-Credits (for Regression)'. For these models we'll report Accuracy, Precision, Average Precision, Recall, Average Recall, Confusion Matrix, F-Score, Precision-Recall Curve and Mean-Squared Error. Here, we split this assignment into 4 parts: 1. Multi-class Classification on 'Car Evaluation' dataset, 2. Binary Classification on 'DT-BrainCancer' dataset, 3. Regression on 'DT-Wage' dataset, 4. Regression on 'DT-Credits' dataset. Each part is handled in a separate notebook file, with each file focused on a single dataset. To implement these models we've follow multiples resources. [1] [2] [3] [4]

2. Methods

Necessary Imports:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from xgboost import XGBClassifier, XGBRegressor
```

We have imported all necessary libraries and modules to be used in each notebook.

2.1 Multi-class Classification on 'Car Evaluation' dataset

Loading Dataset:

```
car_evaluation = pd.read_csv('/content/drive/MyDrive/CSE445-Assignment/Assignment-4/dataset/ca

car_evaluation.head()
```

Load the Dataset using pandas library and check first few data from the dataset.

Assign Column name:

```
column_names = ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety', 'class']
car_evaluation.columns = column_names
car_evaluation.head()
```

Assign each columns with a name, since the original dataset didn't have any headings.

Checking Missing Values:

```
print('Number␣of␣missing␣values:␣')
print(car_evaluation.isnull().sum())
```

Checking if any missing values are founds in any features.

Mapping Categorical Data:

```
car_evaluation['buying'] = car_evaluation['buying'].map({'vhigh':3, 'high':2, 'med':1, 'low':0
car_evaluation['maint'] = car_evaluation['maint'].map({'vhigh':3, 'high':2, 'med':1, 'low':0})
car_evaluation['doors'] = car_evaluation['doors'].map({'2':0, '3':1, '4':2, '5more':3})
car_evaluation['persons'] = car_evaluation['persons'].map({'2':0, '4':1, 'more':2})
car_evaluation['lug_boot'] = car_evaluation['lug_boot'].map({'small':0, 'med':1, 'big': 2})
car_evaluation['safety'] = car_evaluation['safety'].map({'low':0, 'med':1, 'high': 2})

car_evaluation['class'] = car_evaluation['class'].map({'unacc':0, 'acc':1, 'good': 2, 'vgood':
car_evaluation.head()
```

Since there is no complex relationship between the categories, so we just mapped them to values ranging from 0 to 4. Then each categorical columns are encoded in 0 to 4.

Split Data into Training, Validation and Test Sets:

```
X = car_evaluation.drop(columns='class')
y = car_evaluation['class']
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42

print(f'Train␣Data␣Shape␣(X,␣y):␣{X_train.shape,␣y_train.shape}')
print(f'Validation␣Data␣Shape␣(X,␣y):␣{X_val.shape,␣y_val.shape}')
print(f'Test␣Data␣Shape␣(X,␣y):␣{X_test.shape,␣y_test.shape}')
```

Splitting 70 percent data for training, 15 percent for validation and rest 15 percent for testing.

Initialize the Classifier Models and Hyper-parameters for GridSearchCV:

```
# initialize the classifier models
dt_classifier = DecisionTreeClassifier(random_state=42)
xgb_classifier = XGBClassifier(random_state=42)

```

```
# initialize hyperparameters for gridsearchCV
dt_param_grid = {
    'max_depth': [2,4,8,10],
    'min_samples_split': [2,4,5,10],
    'min_samples_leaf': [1,3,5,8,10]
}

xgb_param_grid = {
    'max_depth': [3,5,7],
    'learning_rate': [0.1, 0.01, 0.001],
    'n_estimators': [100, 200, 300],
    'subsample': [0.8, 1.0]
}
```

Here, we initialize the Decision Tree and XGBoost classifier models. Then, we define the hyperparameter grids for each model to fine-tune the key parameters. For the Decision Tree classifier, we selected a specific range of values for `max_depth`, `min_samples_split`, and `min_samples_leaf`, allowing the model to explore different tree structures. We do the same for the XGBoost model, where we choose a specific range of values for `max_depth`, `learning_rate`, `n_estimators`, and `subsample`. [1]

Training Decision Tree Model:

```
# Decision Tree:
dt_grid_search = GridSearchCV(estimator=dt_classifier, param_grid=dt_param_grid, cv=5, scoring
dt_grid_search.fit(X_train, y_train)

dt_best_params = dt_grid_search.best_params_
dt_best_score = dt_grid_search.best_score_
print(f"Decision Tree Best Parameters: {dt_best_params}")
print(f"Decision Tree Best Score: {dt_best_score}")
```

Here, we perform hyperparameter tuning for the Decision Tree model using GridSearchCV by testing different combinations of parameters with 5-fold cross-validation to evaluate the models performance. Then we fit it to the training data and find the best parameters and best scores. [1]

Training XGBoost Model:

```
# XGBoost:
xgb_grid_search = GridSearchCV(estimator=xgb_classifier, param_grid=xgb_param_grid, cv=5, scor
xgb_grid_search.fit(X_train, y_train)

xgb_best_params = xgb_grid_search.best_params_
xgb_best_score = xgb_grid_search.best_score_
print(f"XGBoost Best Parameters: {xgb_best_params}")
print(f"XGBoost Best Score: {xgb_best_score}")
```

There we do the same thing for XGBoost model. We perform hyperparameter tuning with 5-fold cross-validation. Then fit it with training data and find the best parameters. [1]

Retraining the Model with Validation and Training Set:

```
# now retrain the model on the combined training and validation set with the best hyperparamet
X_train_val = np.concatenate((X_train, X_val))
y_train_val = np.concatenate((y_train, y_val))

dt_final_model = DecisionTreeClassifier(**dt_best_params, random_state=42)
xgb_final_model = XGBClassifier(**xgb_best_params, random_state=42)

```

```
# retrain decision tree final model
dt_final_model.fit(X_train_val, y_train_val)
print(f"Decision␣Tree␣Best␣Score:␣{dt_best_score}")

# retrain xgboost final model
xgb_final_model.fit(X_train_val, y_train_val)
print(f"XGBoost␣Best␣Score:␣{xgb_best_score}")
```

Here we retrain both models with the combined training and vaalidation data to make the best use of available data with the optimal best hyper parameters.

Evaluate the Final Model on Test Set:

```
# evaluate the final model on test set
y_test_pred_dt = dt_final_model.predict(X_test)
y_test_pred_xgb = xgb_final_model.predict(X_test)
```

This step, we evaluate the final model's performance on test set.

User Define Evaluation Metrics Function:

```
# functions to calculate: accuracy, average precision, average recall, average f-score.

# Accuracy
def accuracy_score(y_actual, y_predict):
  correct = 0
  total_samples = len(y_actual)

  for predict, actual in zip(y_predict, y_actual):
    if predict == actual:
      correct += 1

  return (correct/ total_samples)

# Precision
def precision_score(y_actual, y_predict):
  precision_scores = []

  for class_label in set(y_actual):
    true_positive = 0
    false_positive = 0

    for actual, predict in zip(y_actual, y_predict):
      if predict == class_label:
        if actual == class_label:
          true_positive += 1
        else:
          false_positive += 1

    if true_positive+false_positive ==0:
      class_precision = 0.0
    else:
      class_precision = true_positive / (true_positive + false_positive)
    precision_scores.append(class_precision)
  return precision_scores

# average precision
def average_precision_score_macro(precision_score, y_actual):
  return sum(precision_score)/len(set(y_actual))

# recall
def recall_score(y_actual, y_predict):
  recall_scores = []

  for class_label in set(y_actual):
    true_positive = 0
    false_negative = 0

    for actual, predict in zip(y_actual, y_predict):
      if actual == class_label:
        if predict == class_label:
          true_positive += 1
        else:
          false_negative += 1

    if true_positive+false_negative ==0:
      class_recall = 0.0
    else:
      class_recall = true_positive / (true_positive + false_negative)

    recall_scores.append(class_recall)
  return recall_scores

# average recall
def average_recall_score_macro(recall_score, y_actual):
  return sum(recall_score)/len(set(y_actual))

# f-score
def fscore(y_actual,y_predict):
  f_score_total = []

  for class_label in set(y_actual):
    true_positive = 0
    false_positive = 0
    false_negative = 0

    for actual, predict in zip(y_actual, y_predict):
      if predict == class_label:
        if actual == class_label:
          true_positive += 1
        else:
          false_positive += 1
      elif actual == class_label:
        false_negative += 1

    if true_positive + false_positive == 0:
      precision = 0.0
    else:
      precision = true_positive/(true_positive+false_positive)

    if true_positive + false_negative == 0:
      precision = 0.0
    else:
      recall = true_positive/(true_positive+false_negative)

    if precision + recall ==0:
      f_score = 0.0
    else:
      f_score = 2*true_positive/((2*true_positive)+false_positive+false_negative)

    f_score_total.append(f_score)
  return f_score_total

# fscore average
def fscore_average_macro(fscore, y_actual):
  return sum(fscore)/len(set(y_actual))
```

Here we define some functions manually to report the evaluation matrices (f-score, precision, recall, average precision, average recall) without using evaluation matrices scikit learn's library function.

Evaluation Metrics:

```
print("Decision␣Tree:␣")
print(f'Accuracy:␣{accuracy_score(y_test,␣y_test_pred_dt)}')
precision = precision_score(y_test, y_test_pred_dt)
print(f'Average␣Precision:␣{average_precision_score_macro(precision,␣y_test)}')
recall = recall_score(y_test, y_test_pred_dt)
print(f'Average␣Recall:␣{average_recall_score_macro(recall,␣y_test)}')
f_score = fscore(y_test, y_test_pred_dt)
print(f'Average␣F1-Score:␣{fscore_average_macro(f_score,␣y_test)}')
print()

print("XGBoost:␣")
print(f'Accuracy:␣{accuracy_score(y_test,␣y_test_pred_xgb)}')
precision = precision_score(y_test, y_test_pred_xgb)
print(f'Average␣Precision:␣{average_precision_score_macro(precision,␣y_test)}')
recall = recall_score(y_test, y_test_pred_xgb)
print(f'Average␣Recall:␣{average_recall_score_macro(recall,␣y_test)}')
f_score = fscore(y_test, y_test_pred_xgb)
print(f'Aveerage␣F1-Score:␣{fscore_average_macro(f_score,␣y_test)}')
```

We print the evaluation metrics using the defined functions.

Confusion Matrix:

```
num_class = len(set(y_test))
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 8))

# confusion matrix for Decision Tree
conf_matrix_dt = np.zeros((num_class, num_class), dtype=int)
for actual, predict in zip(y_test, y_test_pred_dt):
    conf_matrix_dt[actual, predict] += 1

# plotting
ax1.imshow(conf_matrix_dt, cmap='Blues', interpolation='nearest')
ax1.set_title('Decision␣Tree', fontsize=16)
ax1.set_xlabel('Predicted␣Label', fontsize=14)
ax1.set_ylabel('Actual␣Label', fontsize=14)
ax1.set_xticklabels(['unacc', 'acc', 'good', 'vgood'], fontsize=12)
ax1.set_yticklabels(['unacc', 'acc', 'good', 'vgood'], fontsize=12)

for i in range(num_class):
    for j in range(num_class):
        ax1.text(j, i, str(conf_matrix_dt[i, j]), ha='center', va='center', color='red', fonts

# confusion matrix for XGBoost
conf_matrix_xgb = np.zeros((num_class, num_class), dtype=int)
for actual, predict in zip(y_test, y_test_pred_xgb):
    conf_matrix_xgb[actual, predict] += 1

# Plotting
ax2.imshow(conf_matrix_xgb, cmap='Blues', interpolation='nearest')
ax2.set_title('XGBoost', fontsize=16)
ax2.set_xlabel('Predicted␣Label', fontsize=14)
ax2.set_ylabel('Actual␣Label', fontsize=14)
ax2.set_xticklabels(['unacc', 'acc', 'good', 'vgood'], fontsize=12)
ax2.set_yticklabels(['unacc', 'acc', 'good', 'vgood'], fontsize=12)

for i in range(num_class):
    for j in range(num_class):
        ax2.text(j, i, str(conf_matrix_xgb[i, j]), ha='center', va='center', color='red', font

# common properties
for ax in (ax1, ax2):
    ax.set_xticks(np.arange(num_class))
    ax.set_yticks(np.arange(num_class))
    ax.tick_params(axis='both', which='major', labelsize=12)

plt.xticks([0, 1, 2, 3], ['unacc', 'acc', 'goood', 'vgood'])
plt.yticks([0, 1, 2, 3], ['unacc', 'acc', 'goood', 'vgood'])
plt.suptitle('Confusion␣Matrices␣for␣Decision␣Tree␣and␣XGBoost', fontsize=18)
plt.show()
```

There we plot the Confusion Matrix for both model based on the predictions and actual values.

2.2 Binary Classification on 'DT-BrainCancer' dataset

Loading Dataset:

```
df = pd.read_csv('/content/drive/MyDrive/CSE445-Assignment/Assignment-4/dataset/DT-BrainCancer
print(df.shape)
print(df.head())
```

Load the Dataset using pandas library and check first few data from the dataset and its Shape.

Checking Missing Values:

```
print('Number␣of␣missing␣values:␣')
print(df.isnull().sum())
```

Checking if any missing values are founds in any features.

Drop Column:

```
df = df.drop(columns=['Unnamed:␣0', 'sex'])
df = df.dropna()
print(df.shape)
print(df.head())
```

Drop the unwanted column and remove the missing values row.

Encoded Categorical Data:

```
le = LabelEncoder()
df['diagnosis'] = le.fit_transform(df['diagnosis'])
df['loc'] = le.fit_transform(df['loc'])
print(df.head(), df.tail())
```

Since there has few complex relationship between the categories, so we encoded them using LabelEnocer. Then we print few samples from the first and few from the last of this encoded dataset.

Split Data into Training, Validation and Test Sets:

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

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42

print(f'Train␣Data␣Shape␣(X,␣y):␣{X_train.shape,␣y_train.shape}')
print(f'Validation␣Data␣Shape␣(X,␣y):␣{X_val.shape,␣y_val.shape}')
print(f'Test␣Data␣Shape␣(X,␣y):␣{X_test.shape,␣y_test.shape}')
```

Splitting 70 percent data for training, 15 percent for validation and rest 15 percent for testing.

Initialize the Classifier Models and Hyper-parameters for GridSearchCV:

```
# initialize the classifier models
dt_classifier = DecisionTreeClassifier(random_state=42)
xgb_classifier = XGBClassifier(random_state=42)

# initialize hyperparameters for gridsearchCV
dt_param_grid = {
    'max_depth': [2,3,4,5,8,10,12,15],
    'min_samples_split': [1,2,4,5,10],
    'min_samples_leaf': [3,5,10,12,15,16]
}

xgb_param_grid = {
    'max_depth': [3,5,7,10,12],
    'learning_rate': [0.1,0.01,0.001],
    'n_estimators': [50,100,150,200,300],
    'subsample': [0.8,1.0]
}
```

Here, we did the same thing that we do for Multi-Class Classification dataset. We initialize the Decision Tree and XGBoost classifier models. Then, we define the hyperparameter grids for each model to fine-tune the key parameters. For the Decision Tree classifier, we selected a specific range of values for `max_depth`, `min_samples_split`, and `min_samples_leaf`, allowing the model to explore different tree structures. We do the same for the XGBoost model, where we choose a specific range of values for `max_depth`, `learning_rate`, `n_estimators`, and `subsample`. [1]

Training Decision Tree Model:

```
# Decision Tree:
dt_grid_search = GridSearchCV(estimator=dt_classifier, param_grid=dt_param_grid, cv=5, scoring
dt_grid_search.fit(X_train, y_train)
```

```

dt_best_params = dt_grid_search.best_params_
dt_best_score = dt_grid_search.best_score_
print(f"Decision␣Tree␣Best␣Parameters:␣{dt_best_params}")
print(f"Decision␣Tree␣Best␣Score:␣{dt_best_score}")
```

Here, we perform hyperparameter tuning for the Decision Tree model using GridSearchCV by testing different combinations of parameters with 5-fold cross-validation to evaluate the models performance. Then we fit it to the training data and find the best parameters and best scores. [1]

Training XGBoost Model:

```
# XGBoost:
xgb_grid_search = GridSearchCV(estimator=xgb_classifier, param_grid=xgb_param_grid, cv=5, scor
xgb_grid_search.fit(X_train, y_train)

xgb_best_params = xgb_grid_search.best_params_
xgb_best_score = xgb_grid_search.best_score_
print(f"XGBoost␣Best␣Parameters:␣{xgb_best_params}")
print(f"XGBoost␣Best␣Score:␣{xgb_best_score}")
```

There we do the same thing for XGBoost model. We perform hyperparameter tuning with 5-fold cross-validation. Then fit it with training data and find the best parameters. [1]

Retraining the Model with Validation and Training Set:

```
# now retrain the model on the combined training and validation set with the best hyperparamet
X_train_val = np.concatenate((X_train, X_val))
y_train_val = np.concatenate((y_train, y_val))

dt_final_model = DecisionTreeClassifier(**dt_best_params, random_state=42)
xgb_final_model = XGBClassifier(**xgb_best_params, random_state=42)

# retrain decision tree final model
dt_final_model.fit(X_train_val, y_train_val)
print(f"Decision␣Tree␣Best␣Score:␣{dt_best_score}")

# retrain xgboost final model
xgb_final_model.fit(X_train_val, y_train_val)
print(f"XGBoost␣Best␣Score:␣{xgb_best_score}")
```

Here we retrain both models with the combined training and vaalidation data to make the best use of available data with the optimal best hyper parameters.

Evaluate the Final Model on Test Set:

```
# evaluate the final model on test set
y_test_pred_dt = dt_final_model.predict(X_test)
y_test_pred_xgb = xgb_final_model.predict(X_test)
```

This step, we evaluate the final model's performance on test set.

Evaluation Metrics:

```
print("Decision␣Tree:␣")
print(f'Accuracy:␣{accuracy_score(y_test,␣y_test_pred_dt)}')
print(f'Precision:␣{precision_score(y_test,␣y_test_pred_dt)}')
print(f'Recall:␣{recall_score(y_test,␣y_test_pred_dt)}')
print(f'F1-Score:␣{fscore(y_test,␣y_test_pred_dt)}')
print()

print("XGBoost:␣")
print(f'Accuracy:␣{accuracy_score(y_test,␣y_test_pred_xgb)}')
print(f'Precision:␣{precision_score(y_test,␣y_test_pred_xgb)}')
print(f'Recall:␣{recall_score(y_test,␣y_test_pred_xgb)}')
print(f'F1-Score:␣{fscore(y_test,␣y_test_pred_xgb)}')
```

We print the evaluation metrics Accuracy, Precision, Recall and F-score using the defined functions.

Confusion Matrix:

```
num_class = len(set(y_test))
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 8))

# confusion matrix for Decision Tree
conf_matrix_dt = np.zeros((num_class, num_class), dtype=int)
for actual, predict in zip(y_test, y_test_pred_dt):
    conf_matrix_dt[actual, predict] += 1

# plotting
ax1.imshow(conf_matrix_dt, cmap='Blues', interpolation='nearest')
ax1.set_title('Decision␣Tree', fontsize=16)
ax1.set_xlabel('Predicted␣Label', fontsize=14)
ax1.set_ylabel('Actual␣Label', fontsize=14)

for i in range(num_class):
    for j in range(num_class):
        ax1.text(j, i, str(conf_matrix_dt[i, j]), ha='center', va='center', color='red', fonts

# confusion matrix for XGBoost
conf_matrix_xgb = np.zeros((num_class, num_class), dtype=int)
for actual, predict in zip(y_test, y_test_pred_xgb):
    conf_matrix_xgb[actual, predict] += 1

# Plotting
ax2.imshow(conf_matrix_xgb, cmap='Blues', interpolation='nearest')
ax2.set_title('XGBoost', fontsize=16)
ax2.set_xlabel('Predicted␣Label', fontsize=14)
ax2.set_ylabel('Actual␣Label', fontsize=14)

for i in range(num_class):
    for j in range(num_class):
        ax2.text(j, i, str(conf_matrix_xgb[i, j]), ha='center', va='center', color='red', font

# common properties
for ax in (ax1, ax2):
    ax.set_xticks(np.arange(num_class))
    ax.set_yticks(np.arange(num_class))
    ax.tick_params(axis='both', which='major', labelsize=12)

plt.suptitle('Confusion␣Matrices␣for␣Decision␣Tree␣and␣XGBoost', fontsize=18)
plt.show()
```

There we plot the Confusion Matrix for both model based on the predictions and actual values.

Precision-Recall Curve :

```
from sklearn.metrics import precision_recall_curve, average_precision_score
y_scores = dt_final_model.predict_proba(X_test)[:, 1]

precision, recall, thresholds = precision_recall_curve(y_test, y_scores)
average_precision = average_precision_score(y_test, y_scores)

plt.figure(figsize=(8, 6))
plt.plot(recall, precision, label=f'Precision-Recall␣curve␣(AP␣=␣{average_precision:.2f})')
plt.xlabel('Recall', fontsize=14)
plt.ylabel('Precision', fontsize=14)
plt.title('Precision-Recall␣Curve', fontsize=16)
plt.legend(loc='best')
plt.grid()
plt.show()

y_scores = xgb_final_model.predict_proba(X_test)[:, 1]

precision, recall, thresholds = precision_recall_curve(y_test, y_scores)
average_precision = average_precision_score(y_test, y_scores)

plt.figure(figsize=(8, 6))
plt.plot(recall, precision, label=f'Precision-Recall␣curve␣(AP␣=␣{average_precision:.2f})')
plt.xlabel('Recall', fontsize=14)
plt.ylabel('Precision', fontsize=14)
plt.title('Precision-Recall␣Curve', fontsize=16)
plt.legend(loc='best')
plt.grid()
plt.show()
```

Here, we plot the Precision-Recall Curve for both models using the built-in function from the scikit-learn module, as defining it manually is complex. This Precision-Recall Curve actually show the trade-off between precision and recall for a model at different thresholds.

2.3 Regression on 'DT-Wage' dataset

Loading Dataset:

```
df = pd.read_csv('/content/drive/MyDrive/CSE445-Assignment/Assignment-4/dataset/DT-Wage.csv')
print(df.shape)
print(df.head())
```

Load the Dataset using pandas library and check first few data from the dataset and its Shape.

Checking Missing Values:

```
print('Number␣of␣missing␣values:␣')
print(df.isnull().sum())
```

Checking if any missing values are founds in any features.

Encoded Categorical Data:

```
df = pd.get_dummies(df, columns=['maritl', 'race', 'education', 'region', 'jobclass', 'health'
df.columns = df.columns.str.replace('[', '_').str.replace(']', '_').str.replace('<', '_')
print(df.shape)
print(df.head())
```

Since there has few complex relationship between the categories, so we used `pd.get_dummies` to convert the categorical columns into one-hot encoded columns Then, we replace special characters ([,], ¡) in the column names with underscores. Then we print few samples from this encoded dataset.

Split Data into Training, Validation and Test Sets:

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

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42

print(f'Train␣Data␣Shape␣(X,␣y):␣{X_train.shape,␣y_train.shape}')
print(f'Validation␣Data␣Shape␣(X,␣y):␣{X_val.shape,␣y_val.shape}')
print(f'Test␣Data␣Shape␣(X,␣y):␣{X_test.shape,␣y_test.shape}')
```

Splitting 70 percent data for training, 15 percent for validation and rest 15 percent for testing.

Initialize the Classifier Models and Hyper-parameters for GridSearchCV:

```
# initialize the regressor models
dt_regressor = DecisionTreeRegressor(random_state=42)
xgb_regressor = XGBRegressor(random_state=42)

# initialize hyperparameters for gridsearchCV
dt_param_grid = {
    'max_depth': [2,3,4,5,8,10],
    'min_samples_split': [2,4,5,10],
    'min_samples_leaf': [1,3,5,8,10,12]
}

xgb_param_grid = {
    'max_depth': [1,3,5,7,10],
    'learning_rate': [0.1,0.01,0.001],
    'n_estimators': [10000,15000,16000],
    'subsample': [0.8,1.0]
}
```

Here, we did the same thing that we do for Multi-Class Classification dataset and Binary Classification dataset. We initialize the Decision Tree and XGBoost classifier models. Then, we define the hyperparameter grids for each model to fine-tune the key parameters. For the Decision Tree classifier, we selected a specific range of values for `max_depth`, `min_samples_split`, and `min_samples_leaf`, allowing the model to explore different tree structures. We do the same for the XGBoost model, where we choose a specific range of values for `max_depth`, `learning_rate`, `n_estimators`, and `subsample`. [1]

Training Decision Tree Model:

```
# Decision Tree:
dt_grid_search = GridSearchCV(estimator=dt_regressor, param_grid=dt_param_grid, cv=3, scoring=
dt_grid_search.fit(X_train, y_train)

dt_best_params = dt_grid_search.best_params_
print(f"Decision␣Tree␣Best␣Parameters:␣{dt_best_params}")
```

Here, we perform hyperparameter tuning for the Decision Tree model using GridSearchCV by testing different combinations of parameters with 3-fold cross-validation to evaluate the models performance. Then we fit it to the training data and find the best parameters and best scores. [1]

Training XGBoost Model:

```
# XGBoost:
xgb_grid_search = GridSearchCV(estimator=xgb_regressor, param_grid=xgb_param_grid, cv=3, scori
xgb_grid_search.fit(X_train, y_train)

xgb_best_params = xgb_grid_search.best_params_
print(f"XGBoost␣Best␣Parameters:␣{xgb_best_params}")
```

There we do the same thing for XGBoost model. We perform hyperparameter tuning with 3-fold cross-validation. Then fit it with training data and find the best parameters. [1]

Retraining the Model with Validation and Training Set:

```
# now retrain the model on the combined training and validation set with the best hyperparamet
X_train_val = np.concatenate((X_train, X_val))
y_train_val = np.concatenate((y_train, y_val))

dt_final_model = DecisionTreeClassifier(**dt_best_params, random_state=42)
xgb_final_model = XGBClassifier(**xgb_best_params, random_state=42)

# retrain decision tree final model
dt_final_model.fit(X_train_val, y_train_val)

# retrain xgboost final model
print(f"XGBoost␣Best␣Score:␣{xgb_best_score}")
```

Here we retrain both models with the combined training and vaalidation data to make the best use of available data with the optimal best hyper parameters.

Evaluate the Final Model on Test Set:

```
# evaluate the final model on test set
y_test_pred_dt = dt_final_model.predict(X_test)
y_test_pred_xgb = xgb_final_model.predict(X_test)
```

This step, we evaluate the final model's performance on test set.

Calculating MSE:

```
def mse(y_actual, y_pred):
    error = 0.0

    for actual, predicted in zip(y_actual, y_pred):
        error += (actual - predicted) ** 2
    mse = error / len(y_actual)
    return mse

print(mse(y_test, y_test_pred_dt))
print(mse(y_test, y_test_pred_xgb))
```

Calculates the MSE with user defined function and print it for Decision Tree and XGBoost models.

2.4 Regression on 'DT-Credit' dataset

Loading Dataset:

```
df = pd.read_csv('/content/drive/MyDrive/CSE445-Assignment/Assignment-4/dataset/DT-Credit.csv'
print(df.shape)
print(df.head())
```

Load the Dataset using pandas library and check first few data from the dataset and its Shape.

Checking Missing Values:

```
print('Number␣of␣missing␣values:␣')
print(df.isnull().sum())
```

Checking if any missing values are founds in any features.

Encoded Categorical Data:

```
df = pd.get_dummies(df, columns=['Own', 'Student', 'Married', 'Region'])
print(df.shape)
print(df.head())
```

Since there has few complex relationship between the categories, so we used `pd.get_dummies` to convert the categorical columns into one-hot encoded columns. Then we print few samples from this encoded dataset.

Split Data into Training, Validation and Test Sets:

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

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42

print(f'Train␣Data␣Shape␣(X,␣y):␣{X_train.shape,␣y_train.shape}')
print(f'Validation␣Data␣Shape␣(X,␣y):␣{X_val.shape,␣y_val.shape}')
print(f'Test␣Data␣Shape␣(X,␣y):␣{X_test.shape,␣y_test.shape}')
```

Splitting 70 percent data for training, 15 percent for validation and rest 15 percent for testing.

Initialize the Classifier Models and Hyper-parameters for GridSearchCV:

```
# initialize the Regressor models
dt_regressor = DecisionTreeRegressor(random_state=42)
xgb_regressor = XGBRegressor(random_state=42)

# initialize hyperparameters for gridsearchCV
dt_param_grid = {
    'max_depth': [2,3,4,5,8,10],
    'min_samples_split': [2,4,5,10],
    'min_samples_leaf': [1,3,5,8,10,12]
}

xgb_param_grid = {
    'max_depth': [1,3,5,7,10],
    'learning_rate': [0.1,0.01,0.001],
    'n_estimators': [10000,15000,16000],
    'subsample': [0.8,1.0]
}
```

Here, we did the same thing that we do for Multi-Class Classification dataset and Binary Classification dataset. We initialize the Decision Tree and XGBoost classifier models. Then, we define the hyperparameter grids for each model to fine-tune the key parameters. For the Decision Tree classifier, we selected a specific range of values for `max_depth`, `min_samples_split`, and `min_samples_leaf`, allowing the model to explore different tree structures. We do the same for the XGBoost model, where we choose a specific range of values for `max_depth`, `learning_rate`, `n_estimators`, and `subsample`. [1]

Training Decision Tree Model:

```
# Decision Tree:
dt_grid_search = GridSearchCV(estimator=dt_regressor, param_grid=dt_param_grid, cv=3, scoring=
dt_grid_search.fit(X_train, y_train)

dt_best_params = dt_grid_search.best_params_
print(f"Decision␣Tree␣Best␣Parameters:␣{dt_best_params}")
```

Here, we perform hyperparameter tuning for the Decision Tree model using GridSearchCV by testing different combinations of parameters with 3-fold cross-validation to evaluate the models performance. Then we fit it to the training data and find the best parameters and best scores. [1]

Training XGBoost Model:

```
# XGBoost:
xgb_grid_search = GridSearchCV(estimator=xgb_regressor, param_grid=xgb_param_grid, cv=3, scori
xgb_grid_search.fit(X_train, y_train)

xgb_best_params = xgb_grid_search.best_params_
print(f"XGBoost␣Best␣Parameters:␣{xgb_best_params}")
```

There we do the same thing for XGBoost model. We perform hyperparameter tuning with 3-fold cross-validation. Then fit it with training data and find the best parameters. [1]

Retraining the Model with Validation and Training Set:

```
# now retrain the model on the combined training and validation set with the best hyperparamet
X_train_val = np.concatenate((X_train, X_val))
y_train_val = np.concatenate((y_train, y_val))

dt_final_model = DecisionTreeClassifier(**dt_best_params, random_state=42)
xgb_final_model = XGBClassifier(**xgb_best_params, random_state=42)

# retrain decision tree final model
dt_final_model.fit(X_train_val, y_train_val)

# retrain xgboost final model
print(f"XGBoost␣Best␣Score:␣{xgb_best_score}")
```

Here we retrain both models with the combined training and vaalidation data to make the best use of available data with the optimal best hyper parameters.

Evaluate the Final Model on Test Set:

```
# evaluate the final model on test set
y_test_pred_dt = dt_final_model.predict(X_test)
y_test_pred_xgb = xgb_final_model.predict(X_test)
```

This step, we evaluate the final model's performance on test set.

Calculating MSE:

```
def mse(y_actual, y_pred):
    error = 0.0

    for actual, predicted in zip(y_actual, y_pred):
        error += (actual - predicted) ** 2
    mse = error / len(y_actual)
    return mse

print(mse(y_test, y_test_pred_dt))
print(mse(y_test, y_test_pred_xgb))
```

Calculates the MSE with user defined function and print it for Decision Tree and XGBoost models.

3. Experiment results

3.1 Decision Tree vs XGBoost in Multi-class Classification on 'Car Evaluation' dataset

Accuracy:

Decision Tree	XGBoost
97.31%	98.07%

The XGBoost Classifier Model achieves higher accuracy (98.07%) compared to the Decision Tree (97.31%).

Average Precision:

Decision Tree	XGBoost
0.8985	0.9224

The XGBoost Classifier Model has higher Average Precision (0.9224) compared to Decision Tree (0.8985).

Average Recall:

Decision Tree	XGBoost
0.9265	0.9327

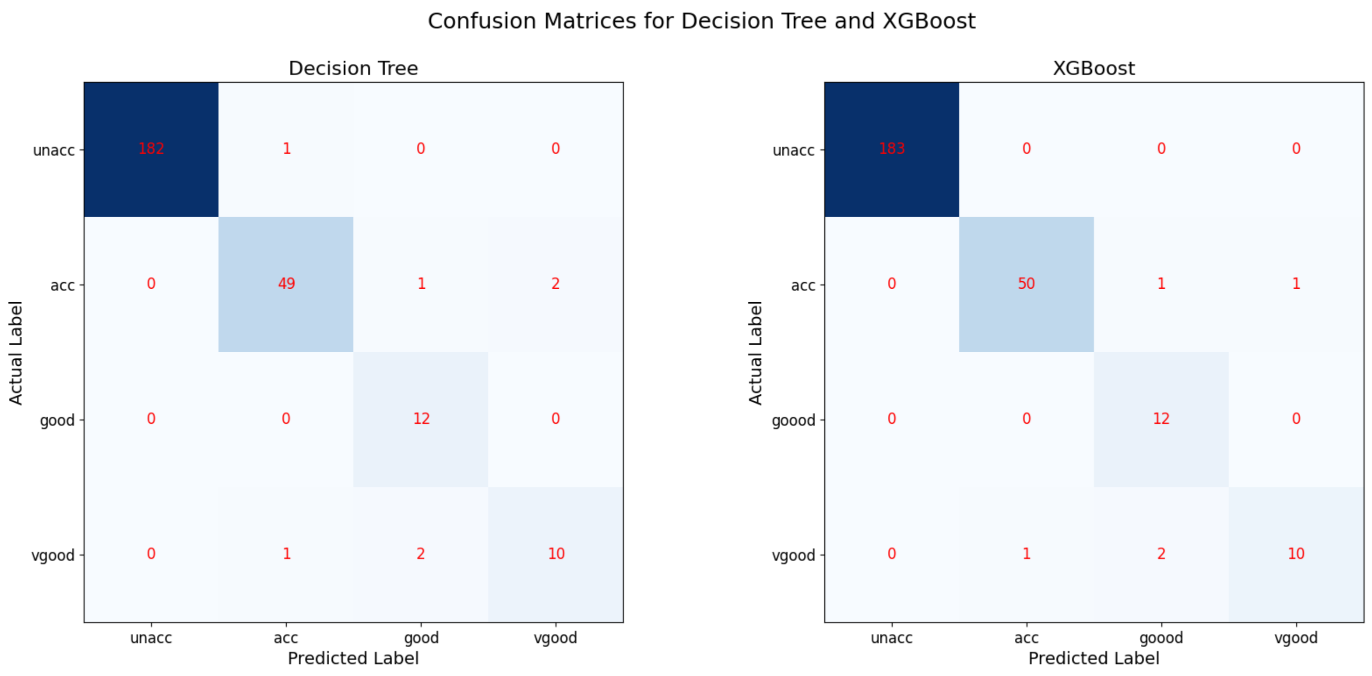
The XGBoost Classifier Model has higher Average Precision (0.9224) compared to Decision Tree (0.8985).

F-Score:

Decision Tree	XGBoost
0.9094	0.9233

The XGBoost Classifier Model has higher Average Precision (0.9224) compared to Decision Tree (0.8985).

Confusion Matrix of Decision Tree vs XGBoost:



3.2 Decision Tree vs XGBoost in Binary Classification on 'DT-BrainCancer' dataset

Accuracy:

Decision Tree	XGBoost
78.57%	92.85%

The XGBoost Classifier Model achieves higher accuracy (92.85%) compared to the Decision Tree (78.57%).

Precision:

Decision Tree		XGBoost	
Class 0	**Class 1**	**CLass 0**	**Class 1**
0.75	1.00	1.0	0.83

Overall XGBoost has higher recall compare to Decision Tree.

Recall:

Decision Tree		XGBoost	
Class 0	**Class 1**	**CLass 0**	**Class 1**
1.00	0.40	0.88	1.00

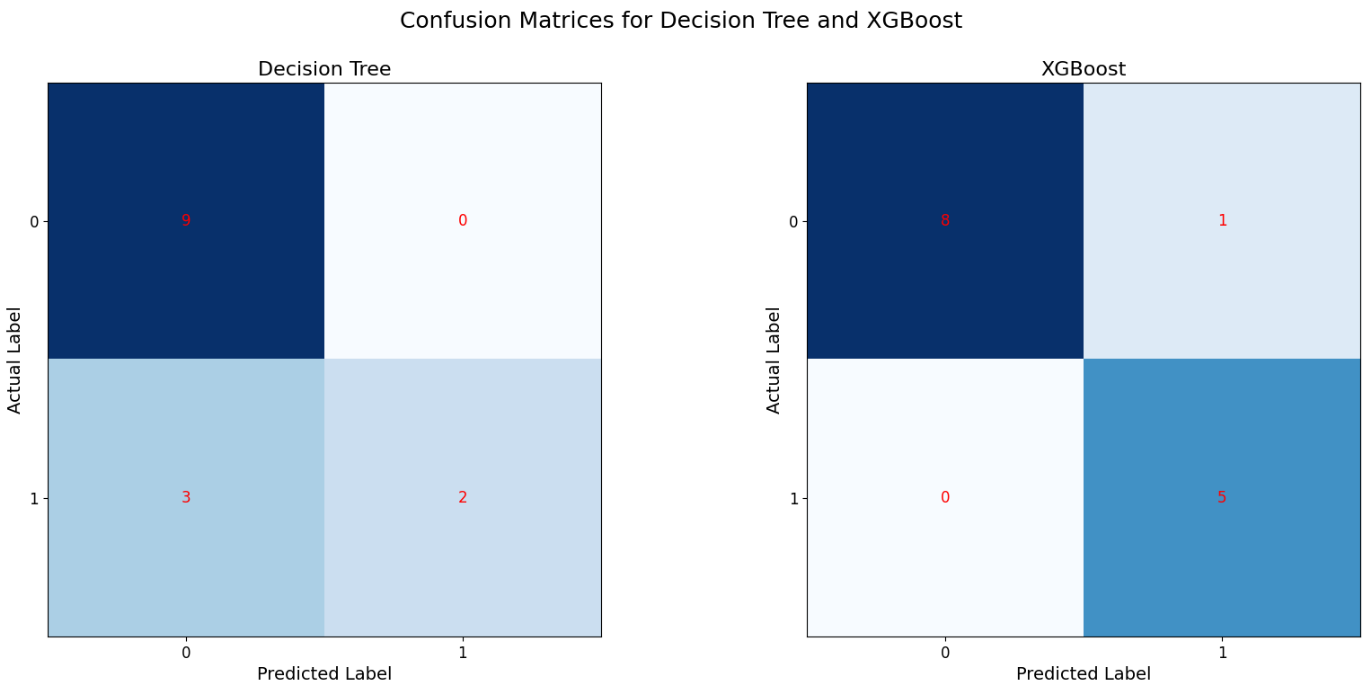
Overall XGBoost has higher precision compare to Decision Tree.

F-Score:

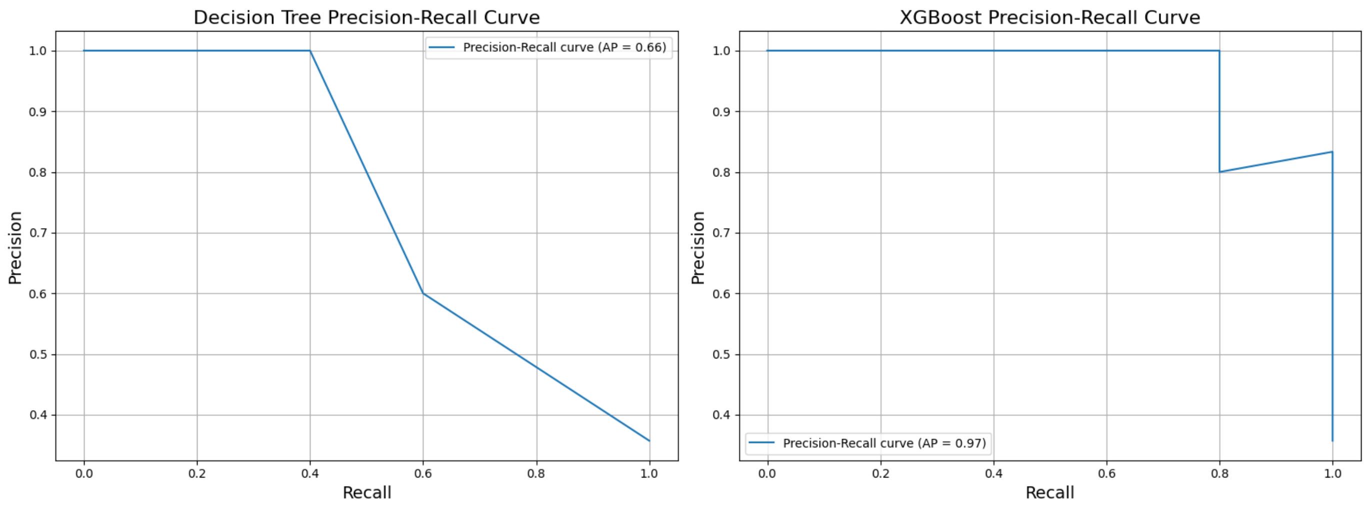
Decision Tree		XGBoost	
Class 0	**Class 1**	**CLass 0**	**Class 1**
0.8571	0.5714	0.9411	0.9090

The XGBoost Classifier demonstrates higher F-score for both classes compared to decision tree.

Confusion Matrix of Decision Tree vs XGBoost:



Precision-Recall Curve of Decision Tree vs XGBoost:



3.3 Decision Tree vs XGBoost in Regression on ’DT-Wage’ dataset

Mean Squared Error:

Decision Tree	XGBoost
0.6654	2.6027

The XGBoost has a higher mean squared error 2.6027 compared to Decision Tree which have 0.6654. So the decision tree is a better fit for this data.

3.4 Decision Tree vs XGBoost in Regression on ’DT-Credit’ dataset

Mean Squared Error:

Decision Tree	XGBoost
17559.2752	3757.1172

The XGBoost has a lower mean squared error 3757.11752 compared to Decision Tree which have 17559.2752. So the XGBoost is a better fit for this data though both do not fit the dataset well ude to extremely high MSE.

4. Discussion

During this assignment, we learned more about Decision Tree, XGBoost, Cross Validation, Evaluation Metrics. We train the two different model on 4 different datasets. On classification task for both binary classification and multiclass classification XGBoost model perform better. F-score, Precision was almost larger compare to decision tree. For Regression task XGBoost work good on DT-Wage dataset but on DT-Credit dataset both models has high MSE. During this whole assignment we follow multiple resources, whole links are given below on the References section.

References

[1] Fermatsavant. Decision tree high accuracy using gridsearchcv. https://www.kaggle.com/code/fermatsavant/decision-tree-high-acc-using-gridsearchcv, 2020.

[2] T. D. Science. An exhaustive guide to classification using decision trees. https://towardsdatascience.com/an-exhaustive-guide-to-classification-using-decision-trees-8d472e77223f, 2024.

[3] S. SDR. Decision tree classification in python. https://medium.com/@shuv.sdr/decision-tree-classification-in-python-b1e59205949c, 2024.

[4] A. Vidhya. An end-to-end guide to understand the math behind xgboost. https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/, 2018.