Assignment-4: Decision Tree and XGBoost

Rezwan-Ul-Alam (ID: 2011659042)
Md. Nur Alam Jowel (ID: 2012355042)
Raian Ruku (ID: 2013409642)

Email addresses:
rezwan.alam1@northsouth.edu
alam.jowel@northsouth.edu
raian.ruku@northsouth.edu

Binary classification problem

Introduction

Decision trees are standard machine learning models for binary classification tasks. The natural structure of a binary classification is suitable for predicting a “yes” or “NO” target. In our case, this assignment is about implementing a decision tree model for binary classification to determine whether a patient has been diagnosed with brain cancer. This work is of great clinical importance, as early detection of brain cancer significantly improves the chances of successful treatment and patient outcomes.

Method:

A. **Data Exploration and preprocessing:** Data preprocessing involves handling missing values, encoding categorical variables, and scaling numerical features to ensure compatibility with the decision tree algorithm and XGBoost Algorithm.

1. **Handling missing values:** First of all, we check which column has how many missing values.

```
for column in df.columns:
    total = df[column].isnull().sum()
    print(f"Column '{column}' has {total} null values")
    print("\n")
```

Here, this code iterates over each column in the DataFrame df, calculates the number of null values in each column, and prints this information along with the column name.

After executed this code we find ‘diagnosis’ column has 1 null or missing value. So, to handling this missing value we remove the whole row that contains null values.

```
1. df.dropna(subset=['diagnosis'], inplace=True)
2. print("Number of null values in 'diagnosis' column after
   dropping:",df['diagnosis'].isnull().sum())
```

here, this code removes rows with missing values in the 'diagnosis' column from the DataFrame df and then prints the count of null values in that column after the removal operation.

2. **Encoding categorical variables**: we use both label encoder and one hot encoder for encoding categorical variables into numerical representations.

2.1) Label encoder: label encoder is useful for encoding categorical variables with only two categories. In our case, the column ‘sex’ and ‘loc’ contains only two categories of variables. That’s why we use a label encoder for these two particular columns.

```
1. from sklearn.preprocessing import LabelEncoder,OneHotEncoder
2. label_enc=LabelEncoder()
3. df['sex'] = label_enc.fit_transform(df['sex'])
4. df['loc'] = label_enc.fit_transform(df['loc'])
```

2.2) one hot encoder: One hot encoder is helpful for encoding categorical variables with two or more categories. In our case, the column ‘Diagnosis’ columns contain more than two categories of variables, and that’s why we use one hot encoder to encode this particular column.

```
1. ’’’one_hot_enc = OneHotEncoder(sparse_output=False, drop='first')
2. diagnosis_encoded = one_hot_enc.fit_transform(df[['diagnosis']])
3. diagnosis_categories = one_hot_enc.categories_[0]
4. diagnosis_columns = [f'diagnosis_{category}' for category in
   diagnosis_categories[1:]]
5. diagnosis_df = pd.DataFrame(diagnosis_encoded,
   columns=diagnosis_columns)
6. df = pd.concat([df, diagnosis_df], axis=1)'"
```

This code extends the encoding process by converting the 'diagnosis' column into one-hot encoded representation and adding the new columns to the DataFrame df.

2. **Scaling features:** We use Minmax Scalar to scale features to the specified range, typically between 0 and 1. The code below introduces a min-max scaling function to perform min-max scaling on numeric columns in our data frame. It calculates the scaled values, transforming X features to a standard scale between 0 and 1.

```
1. from sklearn.preprocessing import MinMaxScaler
2. scaler = MinMaxScaler()
3. X_train_scaled = scaler.fit_transform(X_train)
4. X_val_scaled = scaler.transform(X_val)
5. X_test_scaled = scaler.transform(X_test)
```

Here, fit transform() computes the minimum and maximum values of each feature in the training set and then scales the features accordingly.

B. Train, Test and Validation split:

As per the instruction, the first 70% of the data was used for training, and 15% of the data was used for validation and last 15% of the data is used for testing. The code is given below:

```
1. X = df.iloc[:,:5]
2. y = df.iloc[:, 5]
3. X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,
   random_state=42)
4. X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,
   random_state=42)
```

C. Model Training:

1. Decision Tree:

1.1 Utilize the scikit-learn library to Train a decision tree classifier.

1.2 Tune hyperparameters such as max-depth.

Hyperparameter tuning using the validation set for Decision Tree

```
best_accuracy = 0
best_depth = 1
for max_depth in range(1, 11):
    dt_classifier = DecisionTreeClassifier(max_depth=max_depth, random_state=42)
    dt_classifier.fit(X_train_scaled, y_train)
    y_val_pred_tree = dt_classifier.predict(X_val_scaled)
    accuracy = calculate_accuracy(y_val, y_val_pred_tree)
    print(f"depth {max_depth} , accuracy{accuracy}")
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_depth = max_depth

print("Best max_depth for Decison Tree:", best_depth)
print("Best validation accuracy for Decision Tree:", best_accuracy)
```

This process helps determine the optimal value of the max_depth hyperparameter for the Decision Tree classifier that yields the highest accuracy on the validation set.

Output:

```
depth 1 , accuracy0.6923076923076923
depth 2 , accuracy0.8461538461538461
depth 3 , accuracy0.8461538461538461
depth 4 , accuracy0.6153846153846154
depth 5 , accuracy0.6153846153846154
depth 6 , accuracy0.6923076923076923
depth 7 , accuracy0.6923076923076923
depth 8 , accuracy0.6923076923076923
depth 9 , accuracy0.6923076923076923
depth 10 , accuracy0.6923076923076923
Best max_depth for Decision Tree: 2
Best validation accuracy for Decision Tree: 0.8461538461538461
```

Here, the highest accuracy in our validation set is **0.8461538461538461, and the** best depth is **2.**

Train model using Decision Tree:

```
final_dt_classifier = DecisionTreeClassifier(max_depth=best_depth, random_state=42)
final_dt_classifier.fit(X_train_scaled, y_train)
y_test_pred = final_dt_classifier.predict(X_test_scaled)
test_accuracy_tree = calculate_accuracy(y_test, y_test_pred)
print("Decision Tree Test accuracy with best hyperparameters: ",test_accuracy_tree)
```

here, this code ensures that the Decision Tree classifier is trained with the best hyperparameters and evaluates its performance on an independent test set to assess its effectiveness in making predictions on new, unseen data, and the test accuracy with the best hyperparameter is 0.8571428571428571.

2. **XGBoost:**
   For training the model, we implement the XGBoost algorithm using the XGBoost library and Fine-tune hyperparameters such as max-depth, learning rate, and n_estimators through cross-validation.

Fine-tune hyperparameter using the validation set for XGBoost:

```
1. xgb_classifier = XGBClassifier()
2. best_accuracy = 0
3. best_parameters = {}
4. max_dept = [1,2,3,4,5,6,7,8,9,10]
5. learning_rat = [0.1, 0.01, 0.001]
6. n_estimator = [50, 100, 150, 200]
7. for max_depth in max_dept:
8. for learning_rate in learning_rat:
9. for n_estimators in n_estimator:
        xgb_classifier.set_params(max_depth=max_depth,
10. learning_rate=learning_rate, n_estimators=n_estimators)
        xgb_classifier.fit(X_train_scaled, y_train)
        y_val_pred_XGB = xgb_classifier.predict(X_val_scaled)
        accuracy = calculate_accuracy(y_val, y_val_pred_XGB)
        print(accuracy)
        if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_parameters['max_depth'] = max_depth
        best_parameters['learning_rate'] = learning_rate
        best_parameters['n_estimators'] = n_estimators
11. print("Best hyperparameters for XGB:", best_parameters)
```

This process exhaustively searches through the hyperparameter space to find the combination that yields the highest validation accuracy, ensuring optimal performance of the XGBoost classifier on unseen data.

Output:

```
0.7692307692307693
0.7692307692307693
0.7692307692307693
0.7692307692307693
0.7692307692307693
0.7692307692307693
0.6923076923076923
      .
      .
0.6923076923076923
0.6923076923076923
Best hyperparameters for XGB: {'max_depth': 2, 'learning_rate': 0.1, 'n_estimators': 150}
Best validation accuracy for XGB: 0.8461538461538461
```

Here the best hyperparameters for xgb is {'max_depth': 2, 'learning_rate': 0.1, 'n_estimators': 150} and best validation accuracy is **0.8461538461538461.**

Train model using XGBoost:

```
1. xgb_classifier.set_params(**best_parameters)
2. xgb_classifier.fit(X_train_scaled, y_train)
3. # Evaluate the model on the test set
4. y_test_pred = xgb_classifier.predict(X_test_scaled)
5. test_accuracy_XGB = calculate_accuracy(y_test, y_test_pred)
6. print("XGBoost Test accuracy with best hyperparameters: ",test_accuracy_XGB)
```

Here, this code ensures that the XGBoost classifier is fine-tuned with the best hyperparameters and evaluates its performance on an independent test set to assess its effectiveness in making predictions on new, unseen data and the test accuracy with the best hyperparameter is 0.8571428571428571

D. Evaluation :

For evaluation, we calculate accuracy, precision, recall, F1-score, and confusion matrix.

a. **Accuracy:** accuracy is the ratio of the number of correctly classified instances (true positives and true negatives) to the total number of instances in the dataset.

b. **Precision:** Precision measures the proportion of true positive predictions among all positive predictions made by the classifier. It answers the question: "Out of all the instances predicted as positive, how many are actually positive?"

c. **Recall:** Recall measures the proportion of true positive predictions among all actual positive instances in the dataset. It answers the question: "Out of all the actual positive instances, how many were correctly predicted as positive?"

d. **F1-score:** The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall. F1-score reaches its best value at 1 and worst at 0.

e. **Confusion matrix:** A confusion matrix is a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known. It presents the count of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions made by the classifier.

Calculation process: *here for calculating accuracy, precision, recall, F1-score and confusion matrix we use our own created function. In below we provide all the created function with necessary equation.*

NOTE here,

TP (True positive): Number of correctly predicted positive instances

TN (True negative): Number of correctly predicted negative instances.

FP (False positive): Number of incorrectly predicted positive instances.

FN (False negative): Number of incorrectly predicted negative instances.

1. Precision [TP / (TP+FP)]:

```
def calculate_precision(y_true, y_pred, label):
    true_positives = sum((y_true == label) & (y_pred == label))
    predicted_positives = sum(y_pred == label)
    if predicted_positives > 0:
      precision = true_positives / predicted_positives
    else :0
    return precision
```

Here, this function `calculate_precision(y_true, y_pred, label):` takes the actual labels, predicted labels, and a specific label for which precision is calculated. It then computes and returns the precision value for that label. If there are no predicted positives, it returns 0 to avoid division by zero.

2. ***Recall [TP/ (TP+FN)]:***

```
def calculate_recall(y_true, y_pred, label):
    true_positives = sum((y_true == label) & (y_pred == label))
    actual_positives = sum(y_true == label)
    if actual_positives > 0:
      recall = true_positives / actual_positives
    else :0
    return recall
```

Here, The function `calculate_recall(y_true, y_pred, label)` : essentially computes the recall metric for a given label based on the true and predicted labels provided. It ensures consistence by handling the scenario where there are no actual positive instances for the specified label.

10. F1-Score [2 $*\frac{precision*recall}{precicion+recall}$] :

```
def calculate_f1_score(precision, recall):
if (precision + recall) > 0 :
f1_score = 2 * (precision * recall) / (precision + recall)
else :0
return f1_score
```

Here, the function `calculate_f1_score(precision, recall)` : essentially computes the F1-score metric based on the provided precision and recall values. It ensures consistence by handling the scenario where both precision and recall are zero to avoid division by zero.

11. Accuracy [$\frac{TP+TN}{TP+TN+FP+FN}$] :

```
    def calculate_accuracy(y_true, y_pred):
        correct_predictions = sum(y_true == y_pred)
        total_predictions = len(y_true)
        accuracy = correct_predictions / total_predictions
        return accuracy
```

Here the function calculate_accuracy(y_true, y_pred) essentially computes the accuracy metric based on the provided true labels and predicted labels. It counts the number of correct predictions and divides it by the total number of predictions to obtain the accuracy value.

3. **Confusion matrix** [TN FN
FP TP]:

```
1. def calculate_confusion_matrix(y_true, y_pred):
2. unique_labels = sorted(set(y_true) | set(y_pred))
3. num_labels = len(unique_labels)
4. confusion_matrix = [[0] * num_labels for _ in range(num_labels)]

5. label_to_index = {label: i for i, label in
   enumerate(unique_labels)}

6. for true_label, pred_label in zip(y_true, y_pred):
        true_index = label_to_index[true_label]
        pred_index = label_to_index[pred_label]
        confusion_matrix[true_index][pred_index] += 1

7. return confusion_matrix
```

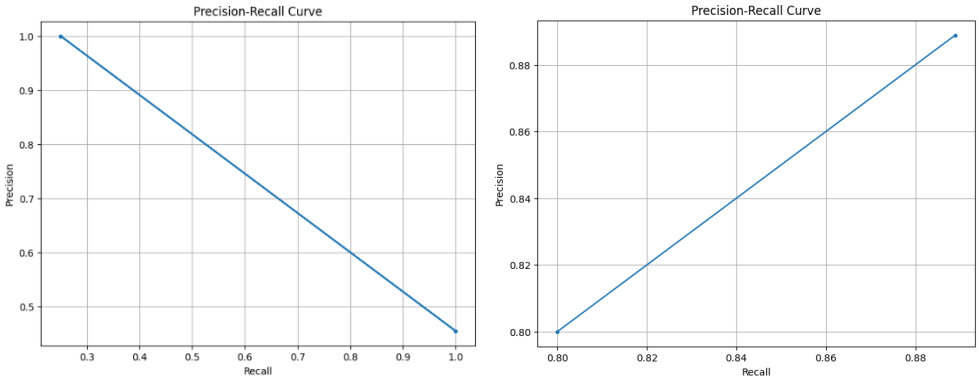
Here the function prints the confusion matrix to the console. It first prints a header indicating that the confusion matrix is for a decision tree model. Then, it iterates over each row in the confusion matrix (conf_matrix) and prints each row. The confusion matrix is typically a 2x2 matrix where rows represent the actual classes and columns represent the predicted classes.

Comparison Decision Tree VS XGBoost:

Class	**Decision Tree(one hot)**	**XGBoost**
Class 0	Precision: 1 Recall: 0.25 F1-score: 0.4	Precision: 0.8888888888888888 Recall: 0.8888888888888888 F1-score: 0.8888888888888888
Class 1	Precision: 0.45 Recall: 1.0 F1-score: 0.625	Precision: 0.8 Recall: 0.8 F1-score: 0.8000000000000002
	Confusion Matrix: [2, 6] [1, 5] Accuracy: 0.8571428571428571	Confusion Matrix: [8, 1] [1, 4] Accuracy: 0.8571428571428571

In this table we see that for class 0 both decision tree and XGBoost algorithm has same precision, recall, F1-Score value. Similarly class 1 has also same precision, recall, F1-score value. And also both algorithm has same accuracy and same confusion matrix.

Precision vs Recall Curves :



Decision tree multiclass classification:

Introduction:

This assignment focuses on employing a decision tree algorithm for multi-class classification tasks. The objective is to classify instances into one of several classes based on a set of input features. Multiclass classification is the process of assigning entities with more than two classes. In our case there is 4 unique classes. Each entity is assigned to one class without any overlap.

Method:

A. **Data Exploration and preprocessing:** Data preprocessing involves handling missing values, encoding categorical variables and scaling numerical features to ensure compatibility with the decision tree algorithm and XGBoost Algorithm.

1) handling missing values: First of all we check how many null values a column has.

After executes this code, we found no null values.

2) **Label encoding:** We manually replaces categorical labels with numerical value using the 'replace ()' method. This is straightforward method to encode categorical labels to numerical values. For example,

```
1. data.classes.replace(('unacc', 'acc', 'good', 'vgood'), (0, 1, 2, 3), inplace = True)
```

in this cdoe the first argument ('unacc', 'acc', 'good', 'vgood') represents the values to be replaced, and the second argument (0, 1, 2, 3) represents the values to replace them with. So, in this case, 'unacc' will be replaced with 0, 'acc' with 1, 'good' with 2, and 'vgood' with 3.

Similar way we replaces all the categorical labels into numerical values.

3. **Scaling features:** we use StandardScalar for scaling features. The StandardScaler in scikit-learn is used for standardizing features by removing the mean and scaling them to unit variance.

```
1. from sklearn.preprocessing import StandardScaler
2. scaler = StandardScaler()
3. X_train_scaled = scaler.fit_transform(X_train)
4. X_val_scaled = scaler.transform(X_val)
5. X_test_scaled = scaler.transform(X_test)
```

This code snippet standardizes the features in the training, validation, and test datasets using the StandardScaler, ensuring that the features have zero mean and unit variance across all datasets.

Train, Test and Validation split:

As per the instruction, the first 70% of the data was used for training, and 15% of the data was used for validation and last 15% of the data is used for testing.

Model Training:

1. Decision Tree:

1.1 Utilize the scikit-learn library to Train a decision tree classifier.

1.2 Tune hyper parameter such as max-depth.

Hyperparameter tuning using the validation set for Decision Tree

```
1. best_accuracy = 0
2. best_depth = 1
3. for max_depth in range(1, 11):
4.     dt_classifier = DecisionTreeClassifier(max_depth=max_depth, random_state=42)
5.     dt_classifier.fit(X_train_scaled, y_train)
6.     y_val_pred_tree = dt_classifier.predict(X_val_scaled)
7.     accuracy = calculate_accuracy(y_val, y_val_pred_tree)
8.     print(f"depth {max_depth} , accuracy{accuracy}")
9.     if accuracy > best_accuracy:
10.        best_accuracy = accuracy
11.        best_depth = max_depth
12.
13. print("Best max_depth for Decison Tree:", best_depth)
14. print("Best validation accuracy for Decision Tree:", best_accuracy)
```

This process helps determine the optimal value of the max_depth hyperparameter for the Decision Tree classifier that yields the highest accuracy on the validation set.

Output:

```
depth 1, accuracy0.6872586872586872
depth 2, accuracy0.7953667953667953
depth 3, accuracy0.7799227799227799
depth 4, accuracy0.8455598455598455
depth 5, accuracy0.8378378378378378
depth 6, accuracy0.9305019305019305
depth 7, accuracy0.9073359073359073
depth 8 , accuracy0.9498069498069498
depth 9 , accuracy0.9613899613899614
depth 10 , accuracy0.9652509652509652
Best max_depth for Decision Tree: 10
Best validation accuracy for Decision Tree: 0.9652509652509652
```

Here, the highest accuracy in our validation set is **0.9652509652509652**, and the best depth is **10.**

Train model using Decision Tree:

Here, this ensures that the Decision Tree classifier is trained with the best hyperparameters and evaluates its performance on an independent test set to assess its effectiveness in making predictions on new, unseen data and the test accuracy with best hyperparameter is 0.9653846153846154.

2. **XGBoost:**

For training the model, we implement the XGBoost algorithm using the XGBoost library and also Fine-tune hyperparameters such as max-depth, learning rate, and n_estimators through cross-validation.

Fine-tune hyperparameter using the validation set for XGBoost:

```
1. xgb_classifier = XGBClassifier()
2. best_accuracy = 0
3. best_parameters = {}
4. max_dept = [1,2,3,4,5,6,7,8,9,10]
5. learning_rat = [0.1, 0.01, 0.001]
6. n_estimator = [50, 100, 150, 200]
7. for max_depth in max_dept:
8. for learning_rate in learning_rat:
9. for n_estimators in n_estimator:
        xgb_classifier.set_params(max_depth=max_depth,
10. learning_rate=learning_rate, n_estimators=n_estimators)
        xgb_classifier.fit(X_train_scaled, y_train)
        y_val_pred_XGB = xgb_classifier.predict(X_val_scaled)
        accuracy = calculate_accuracy(y_val, y_val_pred_XGB)
        print(accuracy)
        if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_parameters['max_depth'] = max_depth
        best_parameters['learning_rate'] = learning_rate
        best_parameters['n_estimators'] = n_estimators
11. print("Best hyperparameters for XGB:", best_parameters)
```

This process exhaustively searches through the hyperparameter space to find the combination that yields the highest validation accuracy, ensuring optimal performance of the XGBoost classifier on unseen data.

Output:

```
Best hyperparameters for XGB: {'max_depth': 3, 'learning rate': 0.1, 'n_estimators': 200}
Best validation accuracy for XGB: 0.9845559845559846
```

Train model using XGBoost:

```
1. xgb_classifier.set_params(**best_parameters)
2. xgb_classifier.fit(X_train_scaled, y_train)
3. # Evaluate the model on the test set
4. y_test_pred = xgb_classifier.predict(X_test_scaled)
5. test_accuracy_XGB = calculate_accuracy(y_test, y_test_pred)
6. print("XGBoost Test accuracy with best hyperparameters: ",test_accuracy_XGB)
```

Here, this code ensures that the XGBoost classifier is fine-tuned with the best hyperparameters and evaluates its performance on an independent test set to assess its effectiveness in making predictions on new, unseen data and the test accuracy with the best hyperparameter is 0.9653846153846154.

Evaluation :

For evaluation we calculate accuracy, precision, recall, F1-score and confusion matrix.

1. **Accuracy:** accuracy is the ratio of the number of correctly classified instances (true positives and true negatives) to the total number of instances in the dataset.

2. **Precision:** Precision measures the proportion of true positive predictions among all positive predictions made by the classifier. It answers the question: "Out of all the instances predicted as positive, how many are actually positive?"

3. **Recall:** Recall measures the proportion of true positive predictions among all actual positive instances in the dataset. It answers the question: "Out of all the actual positive instances, how many were correctly predicted as positive?"

4. **F1-score:** The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall. F1-score reaches its best value at 1 and worst at 0.

5. **Confusion matrix:** A confusion matrix is a table that is often used to describe the performance of a classification model on a set of test data for which the true values

are known. It presents the count of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions made by the classifier.

Calculation process: *here for calculating accuracy, precision, recall, F1-score and confusion matrix we use our own created function. In below we provide all the created function with necessary equation.*

NOTE here,

TP (True positive): Number of correctly predicted positive instances

TN (True negative): Number of correctly predicted negative instances.

FP (False positive): Number of incorrectly predicted positive instances.

FN (False negative): Number of incorrectly predicted negative instances.

a. Precision [TP / (TP+FP)]:

Here, this function `calculate_precision(y_true, y_pred, label)`: takes the true labels, predicted labels, and a specific label for which precision is calculated. It then computes and returns the precision value for that label. If there are no predicted positives, it returns 0 to avoid division by zero.

b. Recall [TP/ (TP+FN)]:

Here, The function `calculate_recall(y_true, y_pred, label)`: essentially computes the recall metric for a given label based on the true and predicted labels provided. It ensures consistence by handling the scenario where there are no actual positive instances for the specified label.

c. F1-Score [$2 * \frac{precision * recall}{precicion + recall}$ *] :*

Here, the function `calculate_f1_score(precision, recall)`: essentially computes the F1-score metric based on the provided precision and recall values. It ensures consistence by handling the scenario where both precision and recall are zero to avoid division by zero.

d. Accuracy [$\frac{TP+TN}{TP+TN+FP+FN}$ *] :*

Here the function calculate_accuracy(y_true, y_pred) essentially computes the accuracy metric based on the provided true labels and predicted labels. It counts the number of correct predictions and divides it by the total number of predictions to obtain the accuracy value.

e. **Confusion matrix** [TN FN
FP TP]:

Here the function prints the confusion matrix to the console. It first prints a header indicating that the confusion matrix is for a decision tree model. Then, it iterates over each row in the confusion matrix (conf_matrix) and prints each row. The confusion matrix is typically a 2x2 matrix where rows represent the actual classes and columns represent the predicted classes.

Finding average performance matrix :

```
unique_labels = set(y_test) | set(y_test_pred)
average_precision = 0
average_recall = 0
average_f1 = 0
for label in unique_labels:
precision = calculate_precision(y_test, y_test_pred, label)
recall = calculate_recall(y_test, y_test_pred, label)
f1_score = calculate_f1_score(precision, recall)
average_precision += precision
    average_recall += recall
    average_f1 += f1_score
    num_classes = len(unique_labels)
    average_precision /= num_classes
    average_recall /= num_classes
    average_f1 /= num_classes

    print("Decision tree Average Precision:{:.3f}".format(average_precision))
    print("Decision tree Average Recall:{:.3f}".format(average_recall))
    print("Decision tree Average F-score:{:.3f}".format(average_f1))
```

Comparison Decision Tree VS XGBoost:

Decision Tree	XGBoost
Precision: 0.918 **Recall:** 0.927 **F1-score:** 0.915 **Confusion Matrix:** [178, 2, 0, 0] [2, 56, 3, 0] [0, 0, 9, 0] [0, 2, 0, 8] **Accuracy:** 0.9653846153846154.	**Precision**: 0.896 **Recall:** 0.948 **F1-score:** 0.911 **Confusion Matrix:** [178, 2, 0, 0] [1, 55, 5, 0] [0, 0, 9, 0] [0, 1, 0, 9] **Accuracy:** 0.9653846153846154.

In this table, we see that the value of precision, recall, F1-score is quite different for both Decision tree and XGBoost algorithm. But the confusion matrix and accuracy is similar for both algorithm.

Regression problem (credit.csv)

Introduction:

Our aim of this assignment is to explore the application of decision tree algorithm and XGBoost algorithm in solving a regression problem related to credit assessment. Regression analysis is a statistical method used for predicting the relationship between dependent and independent variables. In our case of credit assessment, regression can be utilized to predict the creditworthiness of individuals based on various factors such as income, age, education, marriage, region, cards, student etc.

Method:

A. Data Exploration and preprocessing:

Data preprocessing involves handling missing values, encoding categorical variables and scaling numerical features to ensure compatibility with the decision tree algorithm and XGBoost Algorithm.

1) Handling missing values: First of all we check how many null values a column has.

After executes this code, we found no null values.

2) **Label encoding:** We manually replaces categorical labels with numerical value using the 'replace ()' method. This is straightforward method to encode categorical labels to numerical values. For example,

```
1. df.Own.replace(('Yes', 'No'), (1, 0), inplace = True)
2. df.Student.replace(('Yes', 'No'), (1, 0), inplace = True)
3. df.Married.replace(('Yes', 'No'), (1, 0), inplace = True)
4. df.Region.replace(('South', 'West', 'East'), (0, 1, 2), inplace = True)
```

in this code ,the first 3 lines has the first argument ('yes', 'No') represents the values to be replaced, and the second argument (0, 1) represents the values to replace them with. So, in this case, 'yes' will be replaced with 1, 'No' with 1. And last line of code has first argument (('South', 'West', 'East') represents the values to be replaced, and the second argument (0, 1, 2) represents the values to replace them with. So, in this case, 'south' will be replaced with 1, 'West' with 1 and 'East' with replace2.

3. **Scaling features:** we use StandardScalar for scaling features. The StandardScaler in scikit-learn is used for standardizing features by removing the mean and scaling them to unit variance.

B. Train, Test and Validation split:

As per the instruction, the first 70% of the data was used for training, and 15% of the data was used for validation and last 15% of the data is used for testing. The code is given below

C. Model Training:

1. Decision Tree:

1.1 Utilize the scikit-learn library to Train a decision tree classifier.

1.2 Tune hyper parameter such as max-depth.

Hyperparameter tuning using validation set for Decision Tree:

```
1.
2.  max_depth_values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
3.  min_samples_split_values = [2, 5, 10]
4.  min_samples_leaf_values = [1, 2, 3]
5.  best_mse = 100000.0
6.  best_parameters = {}
7.  for max_depth in max_depth_values:
8.  for min_samples_split in min_samples_split_values:
9.  for min_samples_leaf in min_samples_leaf_values:
            dt_regressor = DecisionTreeRegressor(max_depth=max_depth,
                        a. min_samples_split=min_samples_split,
                        b. min_samples_leaf=min_samples_leaf)
            dt_regressor.fit(X_train_scaled, y_train)
            y_val_pred = dt_regressor.predict(X_val_scaled)
            mse = calculate_mse(y_val, y_val_pred)
            print("Mean Squared Error on Validation Set:", mse)
            if mse < best_mse:
            best_mse = mse
            best_parameters['max_depth'] = max_depth
            best_parameters['min_samples_split'] = min_samples_split
            best_parameters['min_samples_leaf'] = min_samples_leaf

10. print("Best Hyperparameters:", best_parameters)
11. print("Best MSE for validation set:", best_mse)
```

This process helps in determining the optimal value of the max_depth hyperparameter for the Decision Tree classifier that yields the highest accuracy on the validation set.

Output:

```
Mean Squared Error on Validation Set: 14476.625462962962
Mean Squared Error on Validation Set: 22025.093115740743
Mean Squared Error on Validation Set: 16061.92214742116
Mean Squared Error on Validation Set: 21514.28621608875
Mean Squared Error on Validation Set: 16457.066666666666
Mean Squared Error on Validation Set: 14363.096759259259
Mean Squared Error on Validation Set: 22267.952375
                    .
                    .
Mean Squared Error on Validation Set: 16694.546221495235
Mean Squared Error on Validation Set: 16575.399925198937
Mean Squared Error on Validation Set: 22027.763993866523
Best Hyperparameters: {'max_depth': 12, 'min_samples_split': 2, 'min_samples_leaf': 2}
Best MSE for validation set: 14363.096759259259
```

Here, the best MSE in our validation set is **14363.096759259259** and best hyperparameter max-depth is **12.**

Train model using Decision Tree:

```
1. best_dt_regressor = DecisionTreeRegressor(**best_parameters)
2. best_dt_regressor.fit(X_train_scaled, y_train)
3. y_pred = best_dt_regressor.predict(X_test_scaled)
4. mse = calculate_mse(y_test, y_pred)
5. print("Mean Squared Error on Test Set:", mse)
```

Here, this code ensures that the Decision Tree classifier is trained with the best hyperparameters and evaluates its performance on an independent test set to assess its effectiveness in making predictions on new, unseen data and MSE on the TEST set with best hyperparameter is **17808.22962962963.**

2. XGBoost:

For training the model we also implement XGBoost algorithm using XGBoost library and also Fine-tune hyperparameter such as max-depth, learning rate and n_estimators through cross validation.

Fine-tune hyperparameter using validation set for XGBoost:

```
max_depth_values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
learning_rate_values = [0.01, 0.05, 0.1, 0.2, 0.3]
n_estimators_values = [50, 100, 150, 200]

best_mse = 100000.0
best_parameters = {}

for max_depth in max_depth_values:
for learning_rate in learning_rate_values:
for n_estimators in n_estimators_values:
xgb_regressor = XGBRegressor(max_depth=max_depth,
                             learning_rate=learning_rate,
                             n_estimators=n_estimators,
                             random_state=42)
xgb_regressor.fit(X_train_scaled, y_train)
y_val_pred = xgb_regressor.predict(X_val_scaled)
mse = calculate_mse(y_val, y_val_pred)
print("Mean Squared Error on Validation Set:", mse)

if mse < best_mse:
        best_mse = mse
        best_parameters['max_depth'] = max_depth
        best_parameters['learning_rate'] = learning_rate
        best_parameters['n_estimators'] = n_estimators

print("Best Hyperparameters:", best_parameters)
print("Best MSE for validation set:", best_mse)
```

This process exhaustively searches through the hyperparameter space to find the combination that yields the best MSE on validation set, ensuring optimal performance of the XGBoost classifier on unseen data.

Output:

```
Mean Squared Error on Validation Set: 100721.3902101979
Mean Squared Error on Validation Set: 47544.09073492393
Mean Squared Error on Validation Set: 25151.572453870882
Mean Squared Error on Validation Set: 15484.318122845023
Mean Squared Error on Validation Set: 10329.179931134766
Mean Squared Error on Validation Set: 5551.242862233653
Mean Squared Error on Validation Set: 5098.948672557632
Mean Squared Error on Validation Set: 4979.638710721745
Mean Squared Error on Validation Set: 6157.53712815169
                    .
                    .
Mean Squared Error on Validation Set: 9263.907698916892
Mean Squared Error on Validation Set: 10474.605604708331
Mean Squared Error on Validation Set: 10474.605604702416
Best Hyperparameters: {'max_depth': 5, 'learning_rate': 0.05, 'n_estimators': 200}
Best MSE for validation set: 4979.638710721745
```

Here, the best MSE in our validation set is **4979.638710721745** and best hyperparameter max-depth is **5.**

Train model using XGBoost:

Here, the XGBoost classifier is fine-tuned with the best hyperparameters and evaluates its performance on an independent test set to assess its effectiveness in making predictions on new, unseen data and MSE on the TEST set with the best hyperparameter is 10474.605604702416.

D. Evaluation:
For valuation regression problem we calculate Mean Squared Error.

1. **Mean Squared Error:** Mean Squared Error (MSE) is a common metric used to evaluate the performance of regression models, including decision tree, XGBoost regression models. It measures the average squared difference between the actual values (observed values) and the predicted values generated by the model. In the context of a decision tree regression problem, MSE quantifies how well the model's predictions match the actual values of the target variable.

The equation for calculating MSE is:

$$\textbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)\ \text{^}2$$

Here, n= number of data values

y_i = observed values

$\hat{y}_i$ = predicted values.

Calculation process: *here for calculating mean squared error we use our own created function. In below we provide our created MSE function with necessary equation.*

MSE function (y_true, y_pred):

```
1) import numpy as np
2) def calculate_mse(y_true, y_pred):
       mse = np.mean((y_true - y_pred) ** 2)
       return mse
```

Here this code snippet provides a reusable function to calculate the Mean Squared Error (MSE) between true values and predicted values, which can be useful for evaluating the performance of regression models in machine learning tasks.

Comparison Decision Tree VS XGBoost:

Decision Tree	XGBoost
Best Hyperparameters: {'max_depth': 12, 'min_samples_split': 2, 'min_samples_leaf': 2} **Best MSE for validation set:** 14363.096759259259 **Mean Squared Error on Test Set:** 17808.22962962963	**Best Hyperparameters**: {'max_depth': 5, 'learning_rate': 0.05, 'n_estimators': 200} **Best MSE for validation set:** 4979.638710721745 **Mean Squared Error on Test Set:** 10474.605604702416

In this table based on the provided results, the XGBoost model outperforms the Decision Tree model in terms of predictive accuracy, as evidenced by the lower MSE values on both the validation and test set

Regression problem (wage.csv)

Introduction: Our objective of this assignment is to explore the application of decision tree and XGBoost algorithm in solving a regression problem. The dataset used for this assignment is "wage.csv", which contains various attributes related to wages of individuals such as education, jobclass, health, race, maritl, age, etc. The task is to build a regression model to predict the wage of an individual based on these attributes using both decision tree and XGBoost algorithm.

Method:

B. Data Exploration and preprocessing:

Data preprocessing involves handling missing values, encoding categorical variables and scaling numerical features to ensure compatibility with the decision tree algorithm and XGBoost Algorithm.

1) Handling missing values: First of all we check how many null values a column has.

After executes this code, we found no null values.

2) **Label encoding:** we use labelEncoder for encoding categorical labels into numerical values. LabelEncoder is a utility class provided by scikit-learn (sklearn) for encoding categorical features as integer values. It essentially converts categorical labels into numerical representations

2. **Scaling features:** we use StandardScalar for scaling features. The StandardScaler in scikit-learn is used for standardizing features by removing the mean and scaling them to unit variance.

C. Train, Test and Validation split:

As per the instruction, the first 70% of the data was used for training, and 15% of the data was used for validation and last 15% of the data is used for testing.

D. Model Training:

1. Decision Tree:

1.1 Utilize the scikit-learn library to Train a decision tree classifier.

1.2 Tune hyper parameter such as max-depth.

Hyperparameter tuning using validation set for Decision Tree:

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

max_depth_values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
min_samples_split_values = [2, 5, 10]
min_samples_leaf_values = [1, 2, 3]
best_mse = 100.0
best_parameters = {}
for max_depth in max_depth_values:
    for min_samples_split in min_samples_split_values:
        for min_samples_leaf in min_samples_leaf_values:
            dt_regressor = DecisionTreeRegressor(max_depth=max_depth,
                                        min_samples_split=min_samples_split,
                                        min_samples_leaf=min_samples_leaf)
            dt_regressor.fit(X_train_scaled, y_train)
            y_val_pred = dt_regressor.predict(X_val_scaled)
            mse = calculate_mse(y_val, y_val_pred)
            print("Mean Squared Error on Validation Set:", mse)

            if mse < best_mse:
                best_mse = mse
                best_parameters['max_depth'] = max_depth
                best_parameters['min_samples_split'] = min_samples_split
                best_parameters['min_samples_leaf'] = min_samples_leaf

print("Best Hyperparameters:", best_parameters)
print("Best MSE for validation set:", best_mse)
```

This process helps in determining the optimal value of the max_depth hyperparameter for the Decision Tree classifier that yields the highest accuracy on the validation set.

Output:

```
Mean Squared Error on Validation Set: 0.060006105870497255
Mean Squared Error on Validation Set: 0.1934014745495293
Mean Squared Error on Validation Set: 0.7917138886169095
Mean Squared Error on Validation Set: 0.08221710583343751
Mean Squared Error on Validation Set: 0.23350965112566674
Mean Squared Error on Validation Set: 0.7899265667431775
Mean Squared Error on Validation Set: 0.3213617028570155
Mean Squared Error on Validation Set: 0.4409966370685461
Mean Squared Error on Validation Set: 0.64662042423184
Mean Squared Error on Validation Set: 0.029177094680618968 .
                    .
                    .
                    .
                    .
Mean Squared Error on Validation Set: 0.2336100848257631
Mean Squared Error on Validation Set: 0.7915694504226165
Mean Squared Error on Validation Set: 0.321159362857007
Mean Squared Error on Validation Set: 0.44099663706854597
Mean Squared Error on Validation Set: 0.6450282173058152
Best Hyperparameters: {'max_depth': 14, 'min_samples_split': 2, 'min_samples_leaf': 1}
Best MSE for validation set: 0.029177094680618968
Mean Squared Error on Test Set: 0.5300233631731901
```

Here, the best MSE in our validation set is **0.029177094680618968** and best hyperparameter max- depth is **14.**

Train model using Decision Tree:

Here, the Decision Tree classifier is trained with the best hyperparameters and evaluates its performance on an independent test set to assess its effectiveness in making predictions on new, unseen data and MSE on the TEST set with best hyperparameter is **0.5300233631731901.**

2. **XGBoost:**

For training the model we also implement XGBoost algorithm using XGBoost library and also Fine-tune hyperparameter such as max-depth, learning rate and n_estimators through cross validation.

Fine-tune hyperparameter using validation set for XGBoost:

```
from xgboost import XGBRegressor
import numpy as np

max_depth_values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
learning_rate_values = [0.01, 0.05, 0.1, 0.2, 0.3]
n_estimators_values = [50, 100, 150, 200]

best_mse = 1000.0
best_parameters = {}

for max_depth in max_depth_values:
    for learning_rate in learning_rate_values:
        for n_estimators in n_estimators_values:
            xgb_regressor = XGBRegressor(max_depth=max_depth,
                                         learning_rate=learning_rate,
                                         n_estimators=n_estimators,
                                         random_state=42)
            xgb_regressor.fit(X_train_scaled, y_train)
            y_val_pred = xgb_regressor.predict(X_val_scaled)
            mse = calculate_mse(y_val, y_val_pred)
            print("Mean Squared Error on Validation Set:", mse)

            if mse < best_mse:
                best_mse = mse
                best_parameters['max_depth'] = max_depth
                best_parameters['learning_rate'] = learning_rate
                best_parameters['n_estimators'] = n_estimators

print("Best Hyperparameters:", best_parameters)
print("Best MSE for validation set:", best_mse)
```

This process exhaustively searches through the hyperparameter space to find the combination that yields the best MSE on validation set, ensuring optimal performance of the XGBoost classifier on unseen data

Output:

```
Mean Squared Error on Validation Set: 0.38694784807310373
Mean Squared Error on Validation Set: 0.3635433718345799
Mean Squared Error on Validation Set: 604.7251932018544
Mean Squared Error on Validation Set: 227.72100865285606
Mean Squared Error on Validation Set: 88.20260820017648
Mean Squared Error on Validation Set: 34.90984169302973
Mean Squared Error on Validation Set: 13.32635180679704
Mean Squared Error on Validation Set: 0.7336015666989046
                    .
                    .
                    .
                    .
                    .
                    .
Mean Squared Error on Validation Set: 1.2946888438490647
Mean Squared Error on Validation Set: 1.2961174210961577
Mean Squared Error on Validation Set: 1.29627185965044
Mean Squared Error on Validation Set: 1.29627185965044
Best Hyperparameters: {'max_depth': 4, 'learning_rate': 0.1, 'n_estimators': 100}
Best MSE for validation set: 0.25953458291985254
```

Here, the best MSE in our validation set is **0.25953458291985254** and best hyperparameter max-depth is **4.**

Train model using XGBoost:

Here, the XGBoost classifier is fine-tuned with the best hyperparameters and evaluates its performance on an independent test set to assess its effectiveness in making predictions on new, unseen data and MSE on the TEST set with the best hyperparameter is 1.29627185965044.

E. Evaluation:
For evaluation on regression problem we calculate Mean Squared Error.

1. **Mean Squared Error:** Mean Squared Error (MSE) is a common metric used to evaluate the performance of regression models, including decision tree, XGBoost regression models. It measures the average squared difference between the actual values (observed values) and the predicted values generated by the model. In the context of a decision tree regression problem, MSE quantifies how well the model's predictions match the actual values of the target variable.

The equation for calculating MSE is:

$$\textbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)\,\hat{}\,2$$

Here, n= number of data values

y_i = observed values

$\hat{y}_i$ = predicted values.

Calculation process: *here for calculating mean squared error we use our own created function. In below we provide our created MSE function with necessary equation.*

MSE function (y_true, y_pred):

```
import numpy as np
def calculate_mse(y_true, y_pred):
    mse = np.mean((y_true - y_pred) ** 2)
    return mse
```

Here this code snippet provides a reusable function to calculate the Mean Squared Error (MSE) between true values and predicted values, which can be useful for evaluating the performance of regression models in machine learning tasks.

Comparison Decision Tree VS XGBoost:

Decision Tree	XGBoost
Best Hyperparameters: {'max_depth': 14, 'min_samples_split': 2, 'min_samples_leaf': 1} **Best MSE for validation set:** 0.029177094680618968 **Mean Squared Error on Test Set:** 0.5300233631731901	**Best Hyperparameters:** {'max_depth': 4, 'learning_rate': 0.1, 'n_estimators': 100} **Best MSE for validation set:** 0.25953458291985254 **Mean Squared Error on Test Set:** 1.29627185965044.

In this table based on the provided results, Decision Tree performed better in terms of minimizing error on the validation set, but it seems to have over fitted the data, leading to poorer performance on the test set. XGBoost, despite having a higher validation set error, demonstrated better generalization to unseen data, as shown by its lower MSE on the test set.