УНИВЕРСИТЕТ ИТМО

Факультет программной инженерии и компьютерной техники

Системы искусственного интеллекта

Лабораторная работа №6

Деревья решений

Преподаватель: Кугаевских Александр Владимирович
Выполнил: Ле Чонг Дат
Группа: P33302

2023 г.

Лабораторная 6. Деревья решений

Деревья решений. Теоретическая часть (1)

Задание

- Для студентов с четным порядковым номером в группе – датасет с классификацией грибов, а нечетным – датасет с данными про оценки студентов инженерного и педагогического факультетов (для данного датасета нужно ввести метрику: студент успешный/неуспешный на основании грейда)
- Отобрать **случайным** образом sqrt(n) признаков
- Реализовать без использования сторонних библиотек построение дерева решений (дерево не бинарное, numpy и pandas использовать можно, использовать списки для реализации дерева - нельзя)
- Провести оценку реализованного алгоритма с использованием Accuracy, precision и recall
- Построить AUC-ROC и AUC-PR (в пунктах 4 и 5 использовать библиотеки нельзя)

Step 1: Installing libraries

This step is usually done once. If you already have these libraries installed, you can skip this step

In [1]:
```
# !pip install pandas numpy scikit-learn matplotlib
```

Step 2: Importing libraries and defining helper functions

In [2]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.preprocessing import label_binarize
```

In [3]:
```
def print_bold(text):
    BOLD = '\033[1m'
    END = '\033[0m'
    print('-' * 100 + f"\n{BOLD}{text}{END}")
```

Step 3: Performing tasks

Task 1: Отобрать случайным образом sqrt(n) признаков

In [4]:
```
import pandas as pd
```

```python
# Load the dataset
df = pd.read_csv('higher_education_students_performance_evaluation.csv')

# Setting 'STUDENT ID' as the index of the dataframe
df.set_index('STUDENT ID', inplace=True)

# Displaying basic information about the dataset
print(df.info())

# Showing the first few rows of the dataframe to understand its structure
print(df.head())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 145 entries, STUDENT1 to STUDENT145
Data columns (total 32 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   1          145 non-null    int64
 1   2          145 non-null    int64
 2   3          145 non-null    int64
 3   4          145 non-null    int64
 4   5          145 non-null    int64
 5   6          145 non-null    int64
 6   7          145 non-null    int64
 7   8          145 non-null    int64
 8   9          145 non-null    int64
 9   10         145 non-null    int64
 10  11         145 non-null    int64
 11  12         145 non-null    int64
 12  13         145 non-null    int64
 13  14         145 non-null    int64
 14  15         145 non-null    int64
 15  16         145 non-null    int64
 16  17         145 non-null    int64
 17  18         145 non-null    int64
 18  19         145 non-null    int64
 19  20         145 non-null    int64
 20  21         145 non-null    int64
 21  22         145 non-null    int64
 22  23         145 non-null    int64
 23  24         145 non-null    int64
 24  25         145 non-null    int64
 25  26         145 non-null    int64
 26  27         145 non-null    int64
 27  28         145 non-null    int64
 28  29         145 non-null    int64
 29  30         145 non-null    int64
 30  COURSE ID  145 non-null    int64
 31  GRADE      145 non-null    int64
dtypes: int64(32)
memory usage: 37.4+ KB
None
            1  2  3  4  5  6  7  8  9  10  ...  23  24  25  26  27  28  29  \
STUDENT ID                                  ...
STUDENT1    2  2  3  3  1  2  2  1  1   1  ...   1   1   3   2   1   2   1
STUDENT2    2  2  3  3  1  2  2  1  1   1  ...   1   1   3   2   3   2   2
STUDENT3    2  2  2  3  2  2  2  2  4   2  ...   1   1   2   2   1   1   2
STUDENT4    1  1  1  3  1  2  1  2  1   2  ...   1   2   3   2   2   1   3
STUDENT5    2  2  1  3  2  2  1  3  1   4  ...   2   1   2   2   2   1   2

            30  COURSE ID  GRADE
STUDENT ID
STUDENT1     1          1      1
STUDENT2     3          1      1
STUDENT3     2          1      1
STUDENT4     2          1      1
STUDENT5     2          1      1
```

```
[5 rows x 32 columns]
```

In [5]:
```python
# First, separate the features and the target
X = df.drop('GRADE', axis=1)
y = df['GRADE']

# Calculate the number of features to select: sqrt(n)
num_features = int(np.sqrt(X.shape[1]) * 2)

# Randomly select sqrt(n) features
selected_features = np.random.choice(X.columns, num_features, replace=False)
X_selected = X[selected_features]

# Displaying the selected features
X_selected.head()
```

Out[5]:

	12	7	COURSE ID	21	1	14	24	9	3	26	20
STUDENT ID											
STUDENT1	2	2	1	1	2	1	1	1	3	2	1
STUDENT2	3	2	1	1	2	1	1	1	3	2	1
STUDENT3	2	2	1	1	2	1	1	4	2	2	1
STUDENT4	2	1	1	1	1	1	2	1	1	2	1
STUDENT5	3	1	1	1	2	1	1	1	1	2	1

Task 2: Реализовать без использования сторонних библиотек построение дерева решений (дерево не бинарное, numpy и pandas использовать можно, использовать списки для реализации дерева - нельзя)

In [6]:
```python
class DecisionNode:
    def __init__(self, feature_index=None, branches=None, value=None, most_common=N
        self.feature_index = feature_index  # Index of the feature for splitting
        self.branches = branches            # Dictionary of branches
        self.value = value                  # Value at a leaf node (if it is a leaf
        self.most_common = most_common
        self.class_probabilities = class_probabilities
        self.num_classes = num_classes

class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.root = None

    def fit(self, X, y):
        unique_classes = np.unique(y)
        self.root = self._build_tree(X, y, depth=0, num_classes=len(unique_classes))

    def _build_tree(self, X, y, depth, num_classes):
        num_samples, num_features = X.shape
```

```python
        unique_classes = np.unique(y)

        # Stopping conditions
        if depth >= self.max_depth or len(unique_classes) == 1 or num_samples == 0:
            leaf_value = self._calculate_leaf_value(y)
            class_probabilities = self._calculate_class_probabilities(y, num_classe
            return DecisionNode(value=leaf_value, class_probabilities=class_probabi

        # Find the best split
        best_feature = self._find_best_split(X, y, num_features)

        # Creating branches for a non-binary tree
        branches = {}
        most_common = self._calculate_most_common_value(y)
        class_probabilities = self._calculate_class_probabilities(y, num_classes)
        for feature_value in np.unique(X[:, best_feature]):
            indices = X[:, best_feature] == feature_value
            X_subset, y_subset = X[indices], y[indices]
            branches[feature_value] = self._build_tree(X_subset, y_subset, depth +

        return DecisionNode(feature_index=best_feature, branches=branches, most_com

    def _calculate_most_common_value(self, y):
        unique_classes, class_counts = np.unique(y, return_counts=True)
        return unique_classes[class_counts.argmax()]

    def _calculate_class_probabilities(self, y, num_classes):
        class_counts = np.bincount(y, minlength=num_classes)
        class_probabilities = class_counts / np.sum(class_counts)
        return class_probabilities

    def _calculate_leaf_value(self, y):
        unique_classes, class_counts = np.unique(y, return_counts=True)
        return unique_classes[class_counts.argmax()]

    def _find_best_split(self, X, y, num_features):
        best_feature = None
        best_gain = -float('inf')

        for feature_index in range(num_features):
            X_column = X[:, feature_index]
            gain = self._information_gain(X_column, y)

            if gain > best_gain:
                best_gain = gain
                best_feature = feature_index

        return best_feature

    def _entropy(self, y):
        class_probs = np.bincount(y) / len(y)
        return -np.sum([p * np.log2(p) for p in class_probs if p > 0])

    def _information_gain(self, X_column, y):
        # Total entropy before the split
        total_entropy = self._entropy(y)
```

```python
        # Values and counts in this column
        values, counts = np.unique(X_column, return_counts=True)

        # Weighted entropy of branches
        weighted_entropy = sum(
            (counts[i] / len(X_column)) * self._entropy(y[X_column == value])
            for i, value in enumerate(values)
        )

        # Information gain is the total entropy minus the weighted entropy of the b
        return total_entropy - weighted_entropy

    def predict(self, X):
        predictions = []
        for data_point in X:
            node = self.root
            while node.value is None:
                feature_value = data_point[node.feature_index]
                if feature_value in node.branches:
                    node = node.branches[feature_value]
                else:
                    # Handling missing branches
                    node = self._handle_missing_branch(node)
            predictions.append(node.value)
        return np.array(predictions)

    def predict_proba(self, X):
        proba_predictions = []
        for data_point in X:
            node = self.root
            while node.value is None:
                feature_value = data_point[node.feature_index]
                if feature_value in node.branches:
                    node = node.branches[feature_value]
                else:
                    node = self._handle_missing_branch(node)
            proba_predictions.append(node.class_probabilities)
        return np.array(proba_predictions)


    def _handle_missing_branch(self, node):
        # Return the most common value of the current node
        return DecisionNode(value=node.most_common, class_probabilities=node.class_
```

In [7]:

```python
# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.2, r

# Initialize the Decision Tree with a maximum depth of 3
tree = DecisionTree(max_depth=3)

# Fit the Decision Tree model to the training data
tree.fit(X_train.values, y_train.values)
```

Task 3: Провести оценку реализованного алгоритма с использованием Accuracy, precision и recall

In [8]:
```
def accuracy(y_true, y_pred):
    correct_predictions = np.sum(y_true == y_pred)
    total_predictions = len(y_true)
    return correct_predictions / total_predictions

def precision_recall_per_class(y_true, y_pred, class_label):
    true_positives = np.sum((y_true == class_label) & (y_pred == class_label))
    predicted_positives = np.sum(y_pred == class_label)
    actual_positives = np.sum(y_true == class_label)

    precision = true_positives / predicted_positives if predicted_positives > 0 els
    recall = true_positives / actual_positives if actual_positives > 0 else 0

    return precision, recall

def average_precision_recall(y_true, y_pred):
    unique_classes = np.unique(y_true)
    precision_sum = 0
    recall_sum = 0

    for class_label in unique_classes:
        precision, recall = precision_recall_per_class(y_true, y_pred, class_label)
        precision_sum += precision
        recall_sum += recall

    average_precision = precision_sum / len(unique_classes)
    average_recall = recall_sum / len(unique_classes)

    return average_precision, average_recall
```

In [9]:
```
y_pred = tree.predict(X_test.values)
y_proba = tree.predict_proba(X_test.values)

accuracy_value = accuracy(y_test, y_pred)

precision_value, recall_value = average_precision_recall(y_test, y_pred)

print(f"Accuracy: {accuracy_value}")
print(f"Precision: {precision_value}")
print(f"Recall: {recall_value}")
```

```
Accuracy: 0.1724137931034483
Precision: 0.22916666666666666
Recall: 0.19791666666666666
```

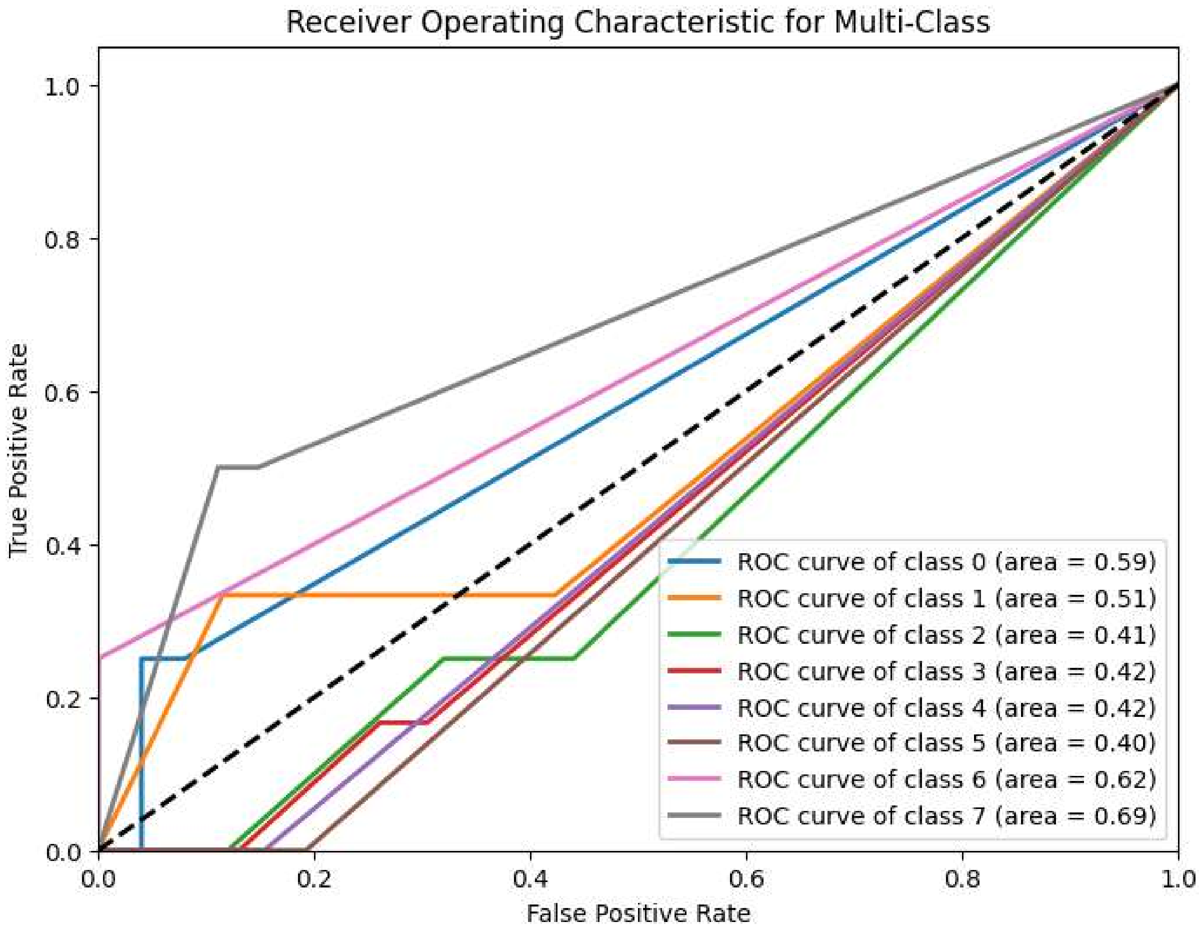
Task 4: Построить AUC-ROC и AUC-PR (в пунктах 4 и 5 использовать библиотеки нельзя)

In [10]:
```
def calculate_tpr_fpr(y_true, y_scores, thresholds):
    tpr_values = []
    fpr_values = []
```

```python
    for threshold in thresholds:
        # Apply threshold to convert probabilities into binary predictions
        y_pred = (y_scores >= threshold).astype(int)

        # Calculate TP, FP, TN, FN
        TP = np.sum((y_true == 1) & (y_pred == 1))
        FP = np.sum((y_true == 0) & (y_pred == 1))
        TN = np.sum((y_true == 0) & (y_pred == 0))
        FN = np.sum((y_true == 1) & (y_pred == 0))

        # Calculate TPR and FPR
        TPR = TP / (TP + FN) if (TP + FN) > 0 else 0
        FPR = FP / (FP + TN) if (FP + TN) > 0 else 0

        tpr_values.append(TPR)
        fpr_values.append(FPR)
    tpr_values.append(0)
    fpr_values.append(0)

    return tpr_values, fpr_values
```

In [11]:

```python
# Binarize the output for multiclass
y_test_binarized = label_binarize(y_test, classes=np.unique(y_train))

n_classes = y_test_binarized.shape[1]

fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(n_classes):
    # Define a range of thresholds from 0 to 1
    thresholds = np.linspace(0, 1, 100)
    tpr[i], fpr[i] = calculate_tpr_fpr(y_test_binarized[:, i], y_proba[:, i], thres
    roc_auc[i] = -np.trapz(tpr[i], fpr[i])

plt.figure(figsize=(8, 6))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=2,
         label='ROC curve of class {0} (area = {1:0.2f})'
         ''.format(i, roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0, 1])
plt.ylim([0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic for Multi-Class')
plt.legend(loc="lower right")
plt.show()
```



In [12]:

```
def calculate_precision_recall(y_true, y_scores, thresholds):
    precision_values = []
    recall_values = []

    for threshold in thresholds:
        # Apply threshold to convert probabilities into binary predictions
        y_pred = (y_scores >= threshold).astype(int)

        TP = np.sum((y_true == 1) & (y_pred == 1))
        predicted_positives = np.sum(y_pred == 1)
        actual_positives = np.sum(y_true == 1)

        precision = TP / predicted_positives if predicted_positives > 0 else 0
        recall = TP / actual_positives if actual_positives > 0 else 0

        precision_values.append(precision)
        recall_values.append(recall)
    precision_values.append(0)
    recall_values.append(0)

    return precision_values, recall_values
```

In [13]:

```
precision = dict()
recall = dict()
auc_pr = dict()
for i in range(n_classes):
    # Define a range of thresholds from 0 to 1
```

```
    thresholds = np.linspace(0, 1, 100)
    precision[i], recall[i] = calculate_precision_recall(y_test_binarized[:, i], y_
    auc_pr[i] = -np.trapz(precision[i], recall[i])

plt.figure(figsize=(8, 6))
for i in range(n_classes):
    plt.plot(recall[i], precision[i], lw=2,
         label='PR curve of class {0} (area = {1:0.2f})'
         ''.format(i, auc_pr[i]))

plt.xlim([0, 1.05])
plt.ylim([0, 1.05])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision Recall curve for Multi-Class')
plt.legend(loc="upper right")
plt.show()
```