CS422 Data Mining Homework2

Problem1:

First Step:

We use binary decision trees, which can also be split by multiple levels to distinguish between the three categories. (In Iris dataset)
Here's the Python code:

```
import pandas as pd
from sklearn.datasets import load_iris
#The first step is to load the iris dataset into a pandas data frame
iris = load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target
X = df.drop('target', axis=1)  # 特征
y = df['target']
from sklearn.metrics import recall_score, precision_score, f1_score
#Define a list of evaluation metrics
recall_scores = []
precision_scores = []
f1_scores = []
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
#In the third step we iterate the depth from 1 to 5
for depth in range(1, 6):
    X_train, X_test, y_train, y_test = train_test_split(X, y,)
    model = DecisionTreeClassifier(min_samples_leaf=2,
min_samples_split=5, max_depth=depth)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    recall = recall_score(y_test, y_pred, average='macro')
    precision = precision_score(y_test, y_pred, average='macro',
zero_division=0)
    f1 = f1_score(y_test, y_pred, average='macro')
    recall_scores.append(recall)
    precision_scores.append(precision)
    f1_scores.append(f1)
    print(f"Depth: {depth}, Recall: {recall:.4f}, Precision:
{precision:.4f}, F1 Score: {f1:.4f}")
results = pd.DataFrame({
    'Depth': range(1, 6),
    'Recall': recall_scores,
    'Precision': precision_scores,
    'F1 Score': f1_scores
})
print("\nResults:\n", results)
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
plt.plot(range(1, 6), recall_scores, marker='o', label='Recall')
plt.plot(range(1, 6), precision_scores, marker='o',
label='Precision')
```

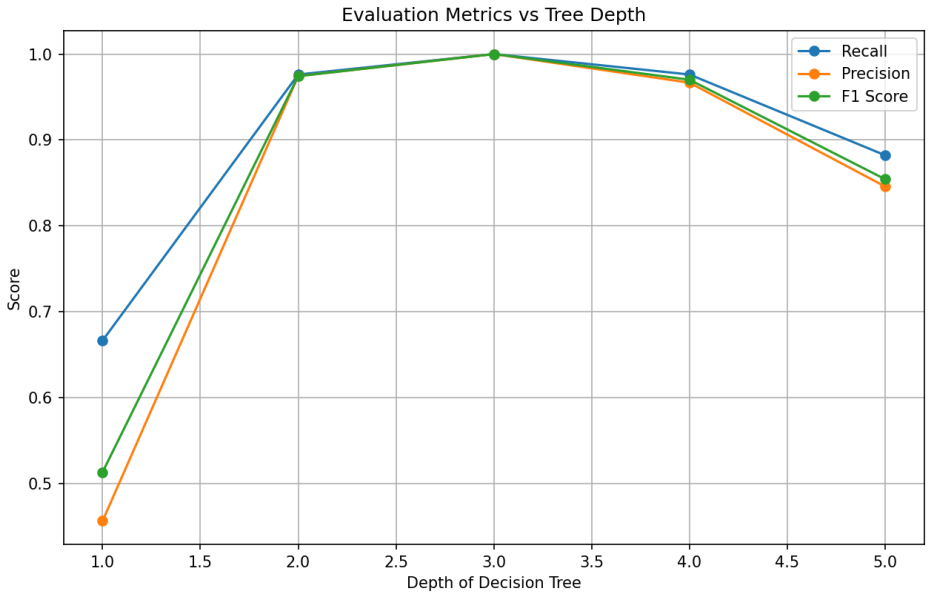
```
plt.plot(range(1, 6), f1_scores, marker='o', label='F1 Score')
plt.xlabel('Depth of Decision Tree')
plt.ylabel('Score')
plt.title('Evaluation Metrics vs Tree Depth')
plt.legend()
plt.grid(True)
plt.show()
```

Here is the result of running the program:

```
D:\Users\Dell\PycharmProjects\build_tree\.venv\Scripts\python.exe D:\Users\Dell\PycharmProjects\build_tree\main.py
Depth: 1, Recall: 0.6667, Precision: 0.4568, F1 Score: 0.5135
Depth: 2, Recall: 0.9762, Precision: 0.9744, F1 Score: 0.9743
Depth: 3, Recall: 1.0000, Precision: 1.0000, F1 Score: 1.0000
Depth: 4, Recall: 0.9762, Precision: 0.9667, F1 Score: 0.9701
Depth: 5, Recall: 0.8822, Precision: 0.8458, F1 Score: 0.8543

Results:
    Depth    Recall  Precision  F1 Score
0       1  0.666667   0.456790  0.513514
1       2  0.976190   0.974359  0.974321
2       3  1.000000   1.000000  1.000000
3       4  0.976190   0.966667  0.970110
4       5  0.882206   0.845833  0.854342
```

Figure 1



Second step:

Let's analyze the results of the data:
We first analyze the recall rate:
After several runs, it can be found that the recall rate of almost every test increases with the depth from 1 to 3, and the recall rate begins to decrease when the depth increases from 3 to 5. (For the sake of rigor, after I ran it enough times it also appeared that our recall and other metrics were increasing monotonically as the depth increased , but this is not representative compared to the previous case).
Increased model complexity: When the depth of the decision tree is shallow (e.g.depth 1 or 2), the model has low complexity and may not adequately capture patterns in the data, leading to underfitting. The model has a lower recall As the depth of the tree increases (e.g., a depth of 3), the model becomes more complex and is able to fit the data better, catching more True Positives, thus increasing the recall.

My explanation for the subsequent drop in recall is overfitting: as the depth of the tree continues to increase (e.g., to a depth of 4 or 5), the model grows in complexity and may start overfitting the training data. Overfitting means that the model performs very well on the training set, but generalizes poorly on the test set. Overfitting models can be overly sensitive to noise and outliers in the training data, leading to inaccurate predictions on the test set, missing some positive examples (increasing false negatives), and thus reducing recall.

The change of the precision rate is analyzed next:

Precision is lowest at depth 1 for the same reason stated above: the model is underfitted, The best accuracy is achieved when the depth is 3.

The best F1 score we can see from the running results is achieved with a depth of 3: Obviously, the reason we can know is consistent with the above analysis: that is, when the depth is too small, the problem of underfitting occurs, and as the depth increases, the problem of overfitting occurs, resulting in the phenomenon of first increasing and then decreasing.

Third Step：

The difference between micro, macro, and weighted methods of score calculation

Micro: The F1 score is calculated directly using the quasicall of the population sample,Focus on overall performance and do not distinguish between categories, An overall performance evaluation can be given quickly, but may mask certain types of performance issues.

Macro: It first computes the referencesand their f1 score for each class, and then takes the average to get the f1 over the entire sample, Focus on the performance of each category rather than the overall performance.

Weighted methods: F1 values are calculated independently for each class, but are weighted according to the number of instances of each class when averaging, The problem of class imbalance is considered and is therefore a more realistic evaluation metric.

Problem 2：

First Step:

The first thing we need to do is import the dataset and make the binary decision tree as required. Here's the code for building this binary decision tree.

```
# The first thing to do is load the Wisconsin Breast Cancer sample dataset into the pandas data frame
import pandas as pd
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/breast-cancer-wisconsin.data"
column_names = [
    'id', 'clump_thickness', 'uniformity_cell_size', 'uniformity_cell_shape',
    'marginal_adhesion', 'single_epithelial_cell_size', 'bare_nuclei',
    'bland_chromatin', 'normal_nucleoli', 'mitoses', 'class'
]
df = pd.read_csv(url, names=column_names)
df.replace( to_replace: '?', pd.NA, inplace=True)
df.dropna(inplace=True)
df['class'] = df['class'].apply(lambda x: 0 if x == 2 else 1)
# The second step is to build the binary decision tree
X = df.drop( labels: ['id', 'class'], axis=1).values
y = df['class'].values
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2, random_state=42)
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(min_samples_leaf=2, min_samples_split=5, max_depth=2, criterion='gini', random_state=42)
model.fit(X_train, y_train)
```

Second Step:

Next we need to design functions to compute each of the subterms of the problem.

First, we compute the entropy of the first split:

After the code of step 1, we design the function "Calculating_entropy". The figure below shows the structure of the function and the result of the first entropy calculation

```python
import numpy as np
# Function to calculate entropy
def Calculating_entropy(y):  3 用法
    hist = np.bincount(y)
    ps = hist / len(y)
    entropy = -np.sum([p * np.log2(p) for p in ps if p > 0])
    return entropy

initial_entropy = Calculating_entropy(y_train)
print(f"Initial Entropy: {initial_entropy}")
tree = model.tree_
first_split_feature = tree.feature[0]
first_split_threshold = tree.threshold[0]
print(f"First split feature: {column_names[first_split_feature + 1]}")  # +1是因为X中不包含id和class列
print(f"First split threshold: {first_split_threshold}")
X_train_split_left = X_train[X_train[:, first_split_feature] <= first_split_threshold]
y_train_split_left = y_train[X_train[:, first_split_feature] <= first_split_threshold]
X_train_split_right = X_train[X_train[:, first_split_feature] > first_split_threshold]
y_train_split_right = y_train[X_train[:, first_split_feature] > first_split_threshold]
entropy_left = Calculating_entropy(y_train_split_left)
entropy_right = Calculating_entropy(y_train_split_right)
weighted_entropy = (len(y_train_split_left) / len(y_train)) * entropy_left + (len(y_train_split_right) / len(y_train)) * entropy_right
print(f"Entropy after first split: {weighted_entropy}")
```

.

Next we calculate the Gini Index for the first split:

```python
# Function to calculate gini index
def Calculating_gini_index(y):  3 用法
    hist = np.bincount(y)
    ps = hist / len(y)
    gini_index = 1 - np.sum(ps ** 2)
    return gini_index

initial_gini_index = Calculating_gini_index(y_train)
print(f"Initial Gini Index: {initial_gini_index}")

gini_index_left = Calculating_gini_index(y_train_split_left)
gini_index_right = Calculating_gini_index(y_train_split_right)
weighted_gini_index = (len(y_train_split_left) / len(y_train)) * gini_index_left + (len(y_train_split_right) / len(y_train)) * gini_index_right
print(f"Gini Index after first split: {weighted_gini_index}")
```

The third one calculates the misclassification error:

```python
# Function to calculate misclassification error
def Calculating_misclassification(y):  3 用法
    hist = np.bincount(y)
    ps = hist / len(y)
    misclassification = 1 - np.max(ps)
    return misclassification
initial_misclassification = Calculating_misclassification(y_train)
print(f"Initial Misclassification Error: {initial_misclassification}")
misclassification_left = Calculating_misclassification(y_train_split_left)
misclassification_right = Calculating_misclassification(y_train_split_right)
weighted_misclassification = (len(y_train_split_left) / len(y_train)) * misclassification_left + (len(y_train_split_right) / len(y_train)) * misclassification_right
print(f"Misclassification Error after first split: {weighted_misclassification}")
```

The information gain after the first segmentation is calculated next:

The code is simple:

```python
information_gain = initial_entropy - weighted_entropy
print(f"Information Gain after first split: {information_gain}")
```

Finally, Run the program to get all the results:

```
D:\Users\Dell\PycharmProjects\Wisconsin\.venv\Scripts\python.exe D:\Users\Dell\PycharmProjects\Wisconsin\main.py
Initial Entropy: 0.9164534336173732
First split feature: uniformity_cell_size
First split threshold: 3.5
Entropy after first split: 0.31445174257352704
Initial Gini Index: 0.44321673442552567
Gini Index after first split: 0.10709682733492265
Initial Misclassification Error: 0.33150183150183155
Misclassification Error after first split: 0.056776556776556825
Information Gain after first split: 0.6020016910438462
```

The initial entropy is 0.9164534336173732.
The first split is characterized by uniformity_cell_size with a splitting threshold of 3.5.
The entropy after the first split is 0.31445174257352704.
The initial Gini index is 0.44321673442552567.
The Gini index after the first split is 0.10709682733492265.
The initial misclassification error rate is 0.33150183150183155.
The misclassification error rate after the first split is 0.056776556776556825.
The information gain after the first split is 0.6020016910438462.

Problem 3：

For the continuous version we can just use the data processed by sklearn without using urls

First Step:

We import the continuous version of the dataset from sklearn into a pandas dataframe. Because of the comparison between the one-factor model involving principal component analysis and the two binary decision trees of the original (continuous) data version, we need to build the binary decision tree based on the original (continuous) data first:

```python
#The first thing to do is load the Wisconsin Breast Cancer sample dataset into the pandas data frame
import pandas as pd
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
#The second step is to build the binary decision tree
X = data.data
y = data.target
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2, random_state=42)
from sklearn.tree import DecisionTreeClassifier,export_text
model = DecisionTreeClassifier(min_samples_leaf=2, min_samples_split=5, max_depth=2, criterion='gini', random_state=42)
model.fit(X_train, y_train)
# Compute the classification report, including precision, recall, and F1 score
from sklearn.metrics import accuracy_score, classification_report
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model accuracy: {accuracy:.4f}")
report = classification_report(y_test, y_pred)
print("Classification report:")
print(report)
```

We used the classification_report function from scikit-learn to compute the F1 score, precision, and recall in one click. Below are the results:

```
D:\Users\Dell\PycharmProjects\Wisconsin2\.venv\Scripts\python.exe D:\Users\Dell\PycharmProjects\Wisconsin2\main.py
Model accuracy: 0.9298
Classification report:
              precision    recall  f1-score   support

           0       0.95      0.86      0.90        43
           1       0.92      0.97      0.95        71

    accuracy                           0.93       114
   macro avg       0.93      0.92      0.92       114
weighted avg       0.93      0.93      0.93       114
```

Second Step:

Next we need to reduce the dimensionality using Principal Component Analysis (PCA) beforehand. Fit a binary decision tree using only the first principal component of the data:

```
#Data after dimensionality reduction using PCA
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
pca = PCA(n_components=1)
X_pca = pca.fit_transform(X_scaled)
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split( *arrays: X_pca, y, test_size=0.2, random_state=42)
model_pca = DecisionTreeClassifier(min_samples_leaf=2,min_samples_split=5,max_depth=2,criterion='gini',random_state=42)
model_pca.fit(X_train_pca, y_train_pca)
y_pred_pca = model_pca.predict(X_test_pca)
accuracy_pca = accuracy_score(y_test_pca, y_pred_pca)
print("\nSecond model (PCA dimensionality reduction): ")
print(f"Model accuracy: {accuracy_pca:.4f}")
report_pca = classification_report(y_test_pca, y_pred_pca)
print("Classification report:")
print(report_pca)
```

We use the same "classification_report" as we did in the first step to measure the performance of the binary decision tree:

```
Second model (PCA dimensionality reduction):
Model accuracy: 0.9649
Classification report:
              precision    recall  f1-score   support

           0       0.98      0.93      0.95        43
           1       0.96      0.99      0.97        71

    accuracy                           0.96       114
   macro avg       0.97      0.96      0.96       114
weighted avg       0.97      0.96      0.96       114
```

To sum up: we obtain the F1 score, precision and recall after binary decision tree obtained under two different methods of processing data.

Third Step：

We next need to perform PCA dimensionality reduction using the first two principal components and fit a binary decision tree.

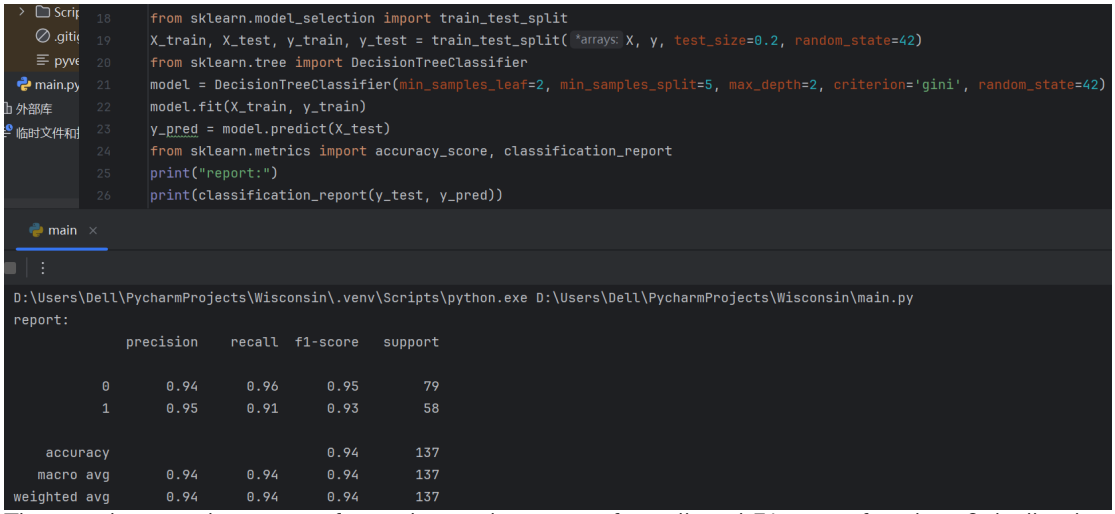
```python
#Principal Component Analysis (PCA) is used for dimensionality reduction, keeping the first two principal components
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split( *arrays: X_pca, y, test_size=0.2, random_state=42)
model_pca = DecisionTreeClassifier(min_samples_leaf=2,min_samples_split=5,max_depth=2,criterion='gini',random_state=42)
model_pca.fit(X_train_pca, y_train_pca)
y_pred_pca = model_pca.predict(X_test_pca)
accuracy_pca = accuracy_score(y_test_pca, y_pred_pca)
print("\nThe third model (PCA to reduce the dimensionality of the two principal components) after the data")
print(f"Model accuracy: {accuracy_pca:.4f}")
report_pca = classification_report(y_test_pca, y_pred_pca)
print("Classification report:")
print(report_pca)
cm = confusion_matrix(y_test_pca, y_pred_pca)
print("confusion matrix:")
print(cm)
TN = cm[0, 0]
FP = cm[0, 1]
FN = cm[1, 0]
TP = cm[1, 1]
FPR = FP / (FP + TN)
TPR = TP / (TP + FN)
print(f"FP: {FP}")
print(f"TP: {TP}")
```

According to the confusion matrix, the values of false positive (FP) and true positive (TP), false positive rate (FPR) and true positiveThe value of the rate (TPR) is as follows:

```
confusion matrix:
[[38  5]
 [ 1 70]]
FP: 5
TP: 70
FPR: 0.1163
TPR: 0.9859
```

Fourth Step:

The first aspect :
The benefit of using a continuous dataset over using reduced data (e.g., PCA dimensionality reduction using only the first or first two principal components). Based on the experimental results of the first three steps, we can get the performance of the three models. From the results, the performance of the continuous data (the original model) is slightly lower than the model after using PCA dimensionality reduction. In particular, the model using PCA for dimensionality reduction to 1 principal component has higher accuracy and F1 score than the original model. This suggests that in some cases, reduced data may be more useful for model performance. At the same time, we should note the potential benefits of dimensionality reduction models in some cases, such as reducing the risk of overfitting or improving training efficiency.
The second aspect:
If we contrast the performance on the continuous dataset with the discrete dataset of Problem 2, We added a function to the source program to calculate its performance and ran the calculation. The result is as follows:



The continuous dataset performs better in terms of recall and F1 score for class 0, indicating that the continuous dataset is more effective in identifying benign tumors.
The overall accuracy is slightly higher than for the discrete dataset.
However, it performs better in precision for class 0 and recall for class 1, indicating that discrete datasets are more effective in avoiding misclassifying benign tumors as malignant and identifying malignant tumors.