CS422 Data Mining
Homework 2

Name:Liner Zhang

ID:A20563408

CONTENTS

1. Problem 1

1.1 Data

First, I imported the Iris dataset from the sklearn library. After converting it into a DataFrame by pandas library and examining the basic information, I confirmed that there were no missing value so no data preprocessing was necessary. I simply added the target values as a new attribute to the dataset and separated the data into feature and target variables.

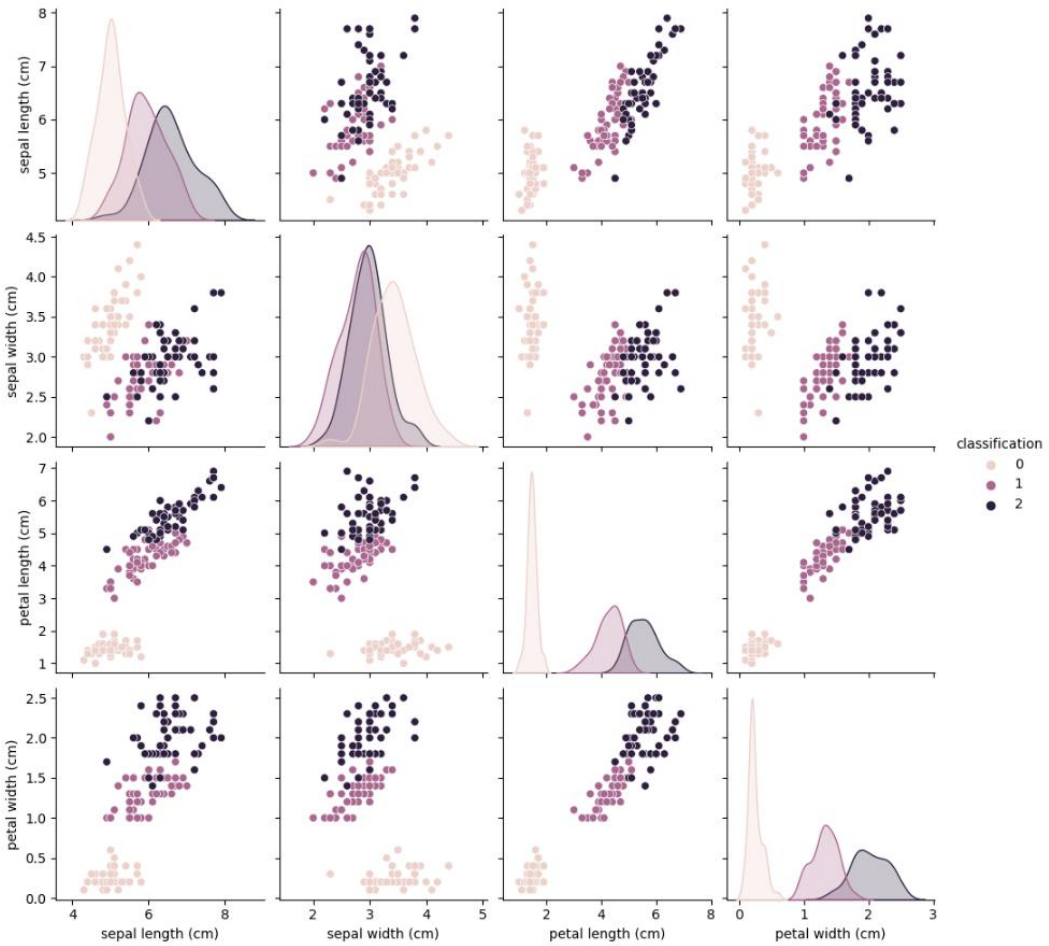
```
#read the data sets
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 4 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
dtypes: float64(4)
memory usage: 4.8 KB
```

```
#prepare the data
data['classification']=iris.target
X=data.select_dtypes(['float'])
y=data.classification
data.tail()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target	class	classification
145	6.7	3.0	5.2	2.3	2	2	2
146	6.3	2.5	5.0	1.9	2	2	2
147	6.5	3.0	5.2	2.0	2	2	2
148	6.2	3.4	5.4	2.3	2	2	2
149	5.9	3.0	5.1	1.8	2	2	2

Then, I conducted a data visualization to better understand the differences and relationships between the three classification categories. Through these plots, I found that the distributions of the three different categories across various features were distinctly different, which should make classification relatively straightforward.



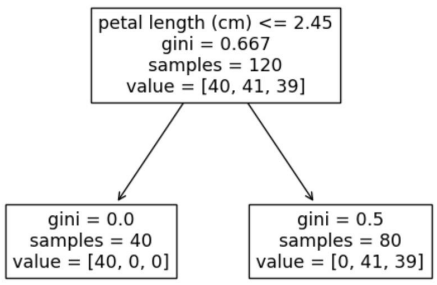
1.2 Binary Decision Trees

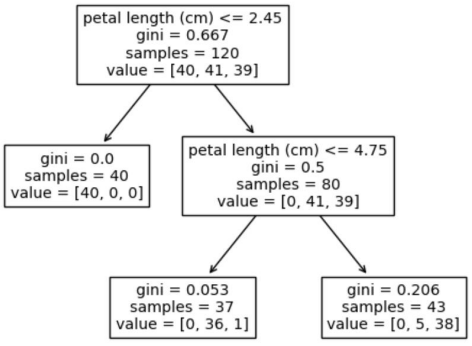
To achieve the goal of training decision trees with maximum depths ranging from 1 to 5, I utilized a for loop to iterate through each desired depth value. Within the loop, I instantiated the DecisionTreeClassifier with the max_depth parameter set to the current depth value and printed the current tree. To facilitate subsequent use, I have stored all the models in the models list.

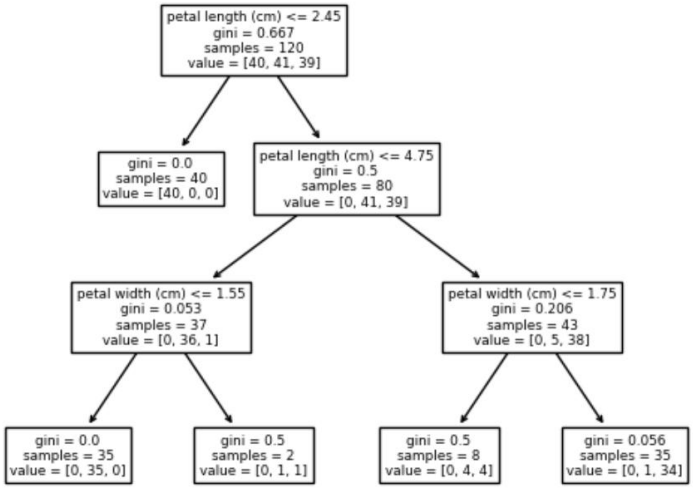
```python
#induce a set of binary trees
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)#train:test=8:2


for depth in range(1,6):
    model=DecisionTreeClassifier(max_depth=depth, min_samples_leaf=2, min_samples_split=5).fit(X_train,y_train)
    tree=export_text(model,feature_names=list(X_train.columns))
    print(tree)
```

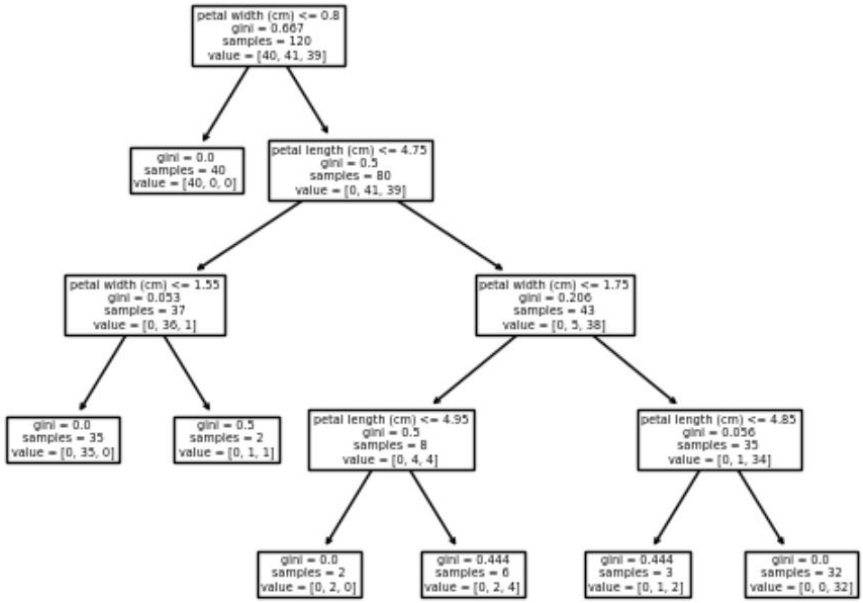
```
|--- petal length (cm) <= 2.45
|   |--- class: 0
|--- petal length (cm) >  2.45
|   |--- class: 1

|--- petal length (cm) <= 2.45
|   |--- class: 0
|--- petal length (cm) >  2.45
|   |--- petal length (cm) <= 4.75
|   |   |--- class: 1
|   |--- petal length (cm) >  4.75
|   |   |--- class: 2

|--- petal width (cm) <= 0.80
|   |--- class: 0
|--- petal width (cm) >  0.80
|   |--- petal length (cm) <= 4.75
|   |   |--- petal width (cm) <= 1.55
|   |   |   |--- class: 1
|   |   |--- petal width (cm) >  1.55
|   |   |   |--- class: 1
|   |--- petal length (cm) >  4.75
|   |   |--- petal width (cm) <= 1.75
|   |   |   |--- class: 1
|   |   |--- petal width (cm) >  1.75
|   |   |   |--- class: 2

|--- petal length (cm) <= 2.45
|   |--- class: 0
|--- petal length (cm) >  2.45
|   |--- petal length (cm) <= 4.75
|   |   |--- sepal length (cm) <= 4.95
|   |   |   |--- class: 1
|   |   |--- sepal length (cm) >  4.95
|   |   |   |--- class: 1
|   |--- petal length (cm) >  4.75
|   |   |--- petal width (cm) <= 1.75
|   |   |   |--- petal length (cm) <= 4.95
|   |   |   |   |--- class: 1
|   |   |   |--- petal length (cm) >  4.95
|   |   |   |   |--- class: 2
|   |   |--- petal width (cm) >  1.75
|   |   |   |--- petal length (cm) <= 4.85
|   |   |   |   |--- class: 2
|   |   |   |--- petal length (cm) >  4.85
|   |   |   |   |--- class: 2

|--- petal length (cm) <= 2.45
|   |--- class: 0
|--- petal length (cm) >  2.45
|   |--- petal length (cm) <= 4.75
|   |   |--- sepal length (cm) <= 4.95
|   |   |   |--- class: 1
|   |   |--- sepal length (cm) >  4.95
|   |   |   |--- class: 1
|   |--- petal length (cm) >  4.75
|   |   |--- petal width (cm) <= 1.75
|   |   |   |--- petal length (cm) <= 4.95
|   |   |   |   |--- class: 1
|   |   |   |--- petal length (cm) >  4.95
|   |   |   |   |--- petal width (cm) <= 1.55
|   |   |   |   |   |--- class: 2
|   |   |   |   |--- petal width (cm) >  1.55
|   |   |   |   |   |--- class: 1
|   |   |--- petal width (cm) >  1.75
|   |   |   |--- petal length (cm) <= 4.85
|   |   |   |   |--- class: 2
|   |   |   |--- petal length (cm) >  4.85
|   |   |   |   |--- class: 2
```

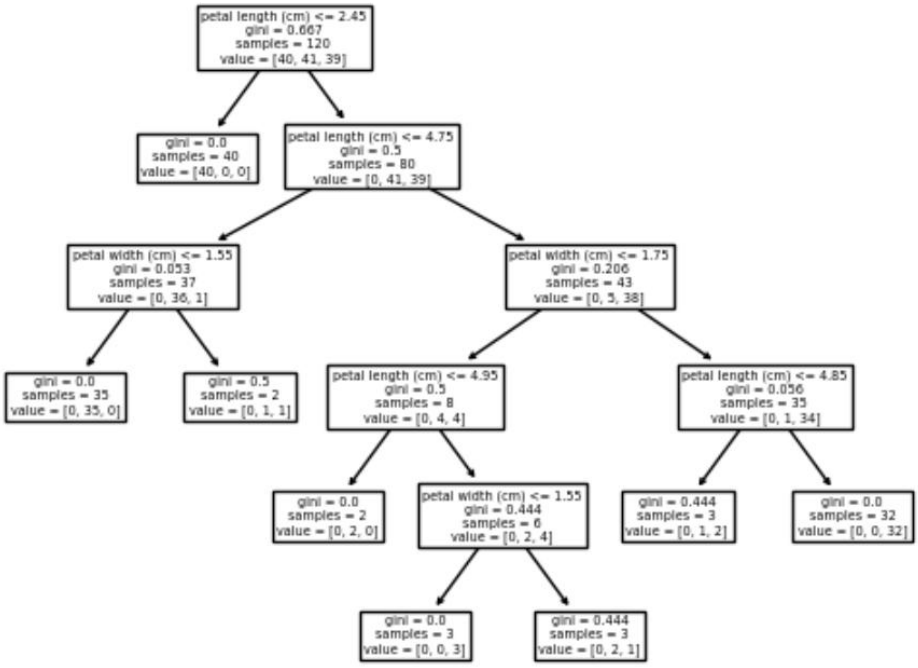
The more intuitive decision tree plots are drawn as follows:






petal length (cm) <= 2.45
gini = 0.667
samples = 120
value = [40, 41, 39]
gini = 0.0
samples = 40
value = [40, 0, 0]
petal length (cm) <= 4.75
gini = 0.5
samples = 80
value = [0, 41, 39]
petal width (cm) <= 1.55
gini = 0.053
samples = 37
value = [0, 36, 1]
petal width (cm) <= 1.75
gini = 0.206
samples = 43
value = [0, 5, 38]
gini = 0.0
samples = 35
value = [0, 35, 0]
gini = 0.5
samples = 2
value = [0, 1, 1]
gini = 0.5
samples = 8
value = [0, 4, 4]
gini = 0.056
samples = 35
value = [0, 1, 34]


petal width (cm) <= 0.8
gini = 0.667
samples = 120
value = [40, 41, 39]
gini = 0.0
samples = 40
value = [40, 0, 0]
petal length (cm) <= 4.75
gini = 0.5
samples = 80
value = [0, 41, 39]
petal width (cm) <= 1.55
gini = 0.053
samples = 37
value = [0, 36, 1]
petal width (cm) <= 1.75
gini = 0.206
samples = 43
value = [0, 5, 38]
gini = 0.0
samples = 35
value = [0, 35, 0]
gini = 0.5
samples = 2
value = [0, 1, 1]
petal length (cm) <= 4.95
gini = 0.5
samples = 8
value = [0, 4, 4]
petal length (cm) <= 4.85
gini = 0.056
samples = 35
value = [0, 1, 34]
gini = 0.0
samples = 2
value = [0, 2, 0]
gini = 0.444
samples = 6
value = [0, 2, 4]
gini = 0.444
samples = 3
value = [0, 1, 2]
gini = 0.0
samples = 32
value = [0, 0, 32]


petal length (cm) <= 2.45
gini = 0.667
samples = 120
value = [40, 41, 39]
gini = 0.0
samples = 40
value = [40, 0, 0]
petal length (cm) <= 4.75
gini = 0.5
samples = 80
value = [0, 41, 39]
petal width (cm) <= 1.55
gini = 0.053
samples = 37
value = [0, 36, 1]
petal width (cm) <= 1.75
gini = 0.206
samples = 43
value = [0, 5, 38]
gini = 0.0
samples = 35
value = [0, 35, 0]
gini = 0.5
samples = 2
value = [0, 1, 1]
petal length (cm) <= 4.95
gini = 0.5
samples = 8
value = [0, 4, 4]
petal length (cm) <= 4.85
gini = 0.056
samples = 35
value = [0, 1, 34]
gini = 0.0
samples = 2
value = [0, 2, 0]
petal width (cm) <= 1.55
gini = 0.444
samples = 6
value = [0, 2, 4]
gini = 0.444
samples = 3
value = [0, 1, 2]
gini = 0.0
samples = 32
value = [0, 0, 32]
gini = 0.0
samples = 3
value = [0, 0, 3]
gini = 0.444
samples = 3
value = [0, 2, 1]

1.3 Related Analysis

1.3.1 Recall

I directly calculated the recall values for each model by invoking the functions from the library.

```
#recall
from sklearn.metrics import recall_score, precision_score, f1_score
for depth in range(1,6):
    y_pred=models[depth-1].predict(X_test)
    recall = recall_score(y_test, y_pred, average='macro')
    print(f"depth={depth}, recall={recall}\n")
```

```
depth=1, recall=0.6666666666666666

depth=2, recall=0.9629629629629629

depth=3, recall=1.0

depth=4, recall=1.0

depth=5, recall=1.0
```

The calculations clearly shows that the recall values are relatively low when the depth is 1 and 2, as the decision trees at these levels are too simple to capture the complex patterns in the data.

The recall reaches its maximum value when the depth is 3. Given the small size of this dataset, which consists of only 150 rows, simple classification can already achieve a fairly high recall rate. However, as the depth increases to 4 and 5, the decision trees become overly complex, which may lead to over fitting.

1.3.2 Precision

Initially, when I used a for loop to calculate the precision values, all five scenarios yielded a precision of 1.0. However, a warning was raised indicating the absence of positive predictions. Upon re-evaluating each scenario individually, I discovered that the warning occurred specifically when the depth was set to 1. At this depth, the model failed to predict any positive samples for class 2, resulting in a precision value of 1.0. Given this, the precision might actually be the lowest in this case.

```
depth=1
y_pred=models[depth-1].predict(X_test)
precision=precision_score(y_test, y_pred, average='macro')
print(f"depth={depth}, precision={recall}\n")
```

```
depth=1, precision=1.0
```

```
D:\A\aboutAna\lib\site-packages\sklearn\metrics\_classification.py:1344: UndefinedMetricWarning: Precision is ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

```
depth=2
y_pred=models[depth-1].predict(X_test)
precision=precision_score(y_test, y_pred, average='macro')
print(f"depth={depth}, precision={recall}\n")
```

```
depth=2, precision=1.0
```

```
depth=3
y_pred=models[depth-1].predict(X_test)
precision=precision_score(y_test, y_pred, average='macro')
print(f"depth={depth}, precision={recall}\n")
```

```
depth=3, precision=1.0
```

```
depth=4
y_pred=models[depth-1].predict(X_test)
precision=precision_score(y_test, y_pred, average='macro')
print(f"depth={depth}, precision={recall}\n")
```

```
depth=4, precision=1.0
```

```
depth=5
y_pred=models[depth-1].predict(X_test)
precision=precision_score(y_test, y_pred, average='macro')
print(f"depth={depth}, precision={recall}\n")
```

```
depth=5, precision=1.0
```

1.3.3 F1 score

While the F1 score is 1.0 across all depths, suggesting that the model has achieved a perfect balance between precision and recall at each depth, the presence of a warning for precision at depth=1, which results from the absence of predicted samples and leads to a default value, indicates that the model may not be making predictions for some classes at this depth. Therefore, depths 2, 3, 4, and 5, which do not trigger such warnings, should be considered as having the best F1 score.

```
#f1 score
for depth in range(1,6):
    y_pred=models[depth-1].predict(X_test)
    f1=f1_score(y_test, y_pred, average='macro')
    print(f"depth={depth}, f1={recall}\n")
```

```
depth=1, f1=1.0

depth=2, f1=1.0

depth=3, f1=1.0

depth=4, f1=1.0

depth=5, f1=1.0
```

1.3.4 Micro，Macro and Weighted

The Micro method aggregates the results of all classes into a global total before calculating the overall metric, making it suitable for imbalanced datasets as it considers the overall performance across all classes, though it may mask the performance of some minority classes.

The Macro method calculates metrics for each class separately and then averages them, which is appropriate for balanced datasets since it assigns equal weight to all classes, but it can be influenced by extreme values in minority classes.

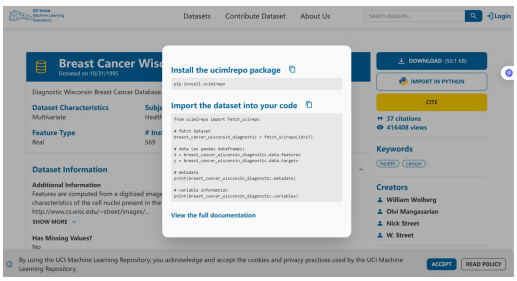
The Weighted method calculates metrics for each class separately and then averages them with weights proportional to the number of samples in each class, making it suitable for imbalanced datasets as it combines the strengths of both Micro and Macro methods, considering overall performance while also giving weight to minority classes.

In this problem, given that the Iris dataset is balanced, the **Macro method** is the most appropriate choice. It assigns equal weight to all classes and provides detailed information about the performance of each class.

2. Problem 2

2.1 Data

I first located the corresponding dataset as per the requirements of the task, copied and executed the import statement in Python to obtain the basic information of the dataset. The data is already in the form of a DataFrame, with features and target separated.



The dataset comprises a total of 32 attributes, among which 'Diagnosis' serves as the target variable, while the remaining attributes function as features, all of which are of the continuous type. There are no null values, so no data preprocessing is needed. Additionally, the target variable is binary.

	name	role	type	demographic	description	units	missing_values
0	ID	ID	Categorical	None	None	None	no
1	Diagnosis	Target	Categorical	None	None	None	no
2	radius1	Feature	Continuous	None	None	None	no
3	texture1	Feature	Continuous	None	None	None	no
4	perimeter1	Feature	Continuous	None	None	None	no
5	area1	Feature	Continuous	None	None	None	no
6	smoothness1	Feature	Continuous	None	None	None	no
7	compactness1	Feature	Continuous	None	None	None	no
8	concavity1	Feature	Continuous	None	None	None	no
9	concave_points1	Feature	Continuous	None	None	None	no
10	symmetry1	Feature	Continuous	None	None	None	no
11	fractal_dimension1	Feature	Continuous	None	None	None	no
12	radius2	Feature	Continuous	None	None	None	no
13	texture2	Feature	Continuous	None	None	None	no
14	perimeter2	Feature	Continuous	None	None	None	no
15	area2	Feature	Continuous	None	None	None	no
16	smoothness2	Feature	Continuous	None	None	None	no
17	compactness2	Feature	Continuous	None	None	None	no
18	concavity2	Feature	Continuous	None	None	None	no
19	concave_points2	Feature	Continuous	None	None	None	no
20	symmetry2	Feature	Continuous	None	None	None	no
21	fractal_dimension2	Feature	Continuous	None	None	None	no
22	radius3	Feature	Continuous	None	None	None	no
23	texture3	Feature	Continuous	None	None	None	no
24	perimeter3	Feature	Continuous	None	None	None	no
25	area3	Feature	Continuous	None	None	None	no
26	smoothness3	Feature	Continuous	None	None	None	no
27	compactness3	Feature	Continuous	None	None	None	no
28	concavity3	Feature	Continuous	None	None	None	no
29	concave_points3	Feature	Continuous	None	None	None	no
30	symmetry3	Feature	Continuous	None	None	None	no
31	fractal_dimension3	Feature	Continuous	None	None	None	no

```
data_1=pd.concat([X, y], axis=1)
data_1
```

ıetry1	fractal_dimension1	...	texture3	perimeter3	area3	smoothness3	compactness3	concavity3	concave_points3	symmetry3	fractal_dimension3	Diagnosis
).2419	0.07871	...	17.33	184.60	2019.0	0.16220	0.66560	0.7119	0.2654	0.4601	0.11890	M
).1812	0.05667	...	23.41	158.80	1956.0	0.12380	0.18660	0.2416	0.1860	0.2750	0.08902	M
).2069	0.05999	...	25.53	152.50	1709.0	0.14440	0.42450	0.4504	0.2430	0.3613	0.08758	M
).2597	0.09744	...	26.50	98.87	567.7	0.20980	0.86630	0.6869	0.2575	0.6638	0.17300	M
).1809	0.05883	...	16.67	152.20	1575.0	0.13740	0.20500	0.4000	0.1625	0.2364	0.07678	M
...	...	...	...	...	...	...	...	...	...	...	...	...
).1726	0.05623	...	26.40	166.10	2027.0	0.14100	0.21130	0.4107	0.2216	0.2060	0.07115	M
).1752	0.05533	...	38.25	155.00	1731.0	0.11660	0.19220	0.3215	0.1628	0.2572	0.06637	M
).1590	0.05648	...	34.12	126.70	1124.0	0.11390	0.30940	0.3403	0.1418	0.2218	0.07820	M
).2397	0.07016	...	39.42	184.60	1821.0	0.16500	0.86810	0.9387	0.2650	0.4087	0.12400	M
).1587	0.05884	...	30.37	59.16	268.6	0.08996	0.06444	0.0000	0.0000	0.2871	0.07039	B

```
data_1['Diagnosis'].unique()
```

```
array(['M', 'B'], dtype=object)
```

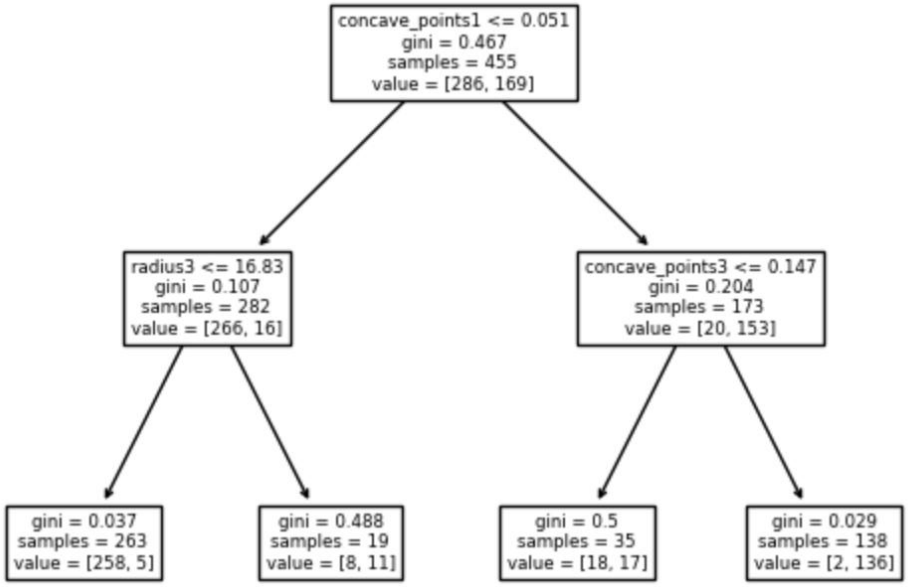
2.2 Binary Decision Tree

In accordance with the requirements of the problem, construct the decision tree with the following constraints: each leaf node must contain at least 2 instances, the minimum number of samples required to split a subset is 5, and the maximum depth of the tree is 2 (using the default Gini criterion).

```
#induce a binary decision tree
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.2, random_state=42) #train:test=8:2

model = DecisionTreeClassifier(max_depth=2, min_samples_leaf=2, min_samples_split=5, criterion='gini').fit(X_train, y_train)
tree=export_text(model, feature_names=list(X_train.columns))
print(tree)
```

```
|--- concave_points1 <= 0.05
|   |--- radius3 <= 16.83
|   |   |--- class: B
|   |--- radius3 >  16.83
|   |   |--- class: M
|--- concave_points1 >  0.05
|   |--- concave_points3 <= 0.15
|   |   |--- class: B
|   |--- concave_points3 >  0.15
|   |   |--- class: M
```

The tree plot is drawn as follows:



2.3 Related Analysis

2.3.1 Entropy

From the plot above, we can see that before the first split, there are 455 train samples in total. 286 belong to one category, while the remaining samples belong to the other category. Given that there are two classes (c = 2), we can calculate the entropy of the root node as follows:

$$\mathrm{E(father)} = -\sum_0^1 \mathrm{p_i(1)log_2\, p_i(1)} = -\frac{286}{455} log_2 \frac{286}{455} - \frac{169}{455} log_2 \frac{169}{455}$$

The entropy of the left node and right node are as follows:

$$E(\mathrm{left}) = -\sum_0^1 \mathrm{p_i(1)log_2\, p_i(1)} = -\frac{266}{282} log_2 \frac{266}{282} - \frac{16}{282} log_2 \frac{16}{282}$$

$$E(\mathrm{right}) = -\sum_0^1 \mathrm{p_i(1)log_2\, p_i(1)} = -\frac{20}{173} log_2 \frac{20}{173} - \frac{153}{173} log_2 \frac{153}{173}$$

The entropy of the first split is as follows:

$$E(\text{son}) = \frac{286}{455}\ \text{E(left)} + \frac{173}{455}\ \text{E(right)}$$

The information gain is as follows:

$$\Delta = \text{E(father)} - \text{E(son)}$$

```
entropy_father=-(286/455)*np.log2(286/455)-(169/455)*np.log2(169/455)
entropy_left=-(266/282)*np.log2(266/282)-(16/282)*np.log2(16/282)
entropy_right=-(20/173)*np.log2(20/173)-(153/173)*np.log2(153/173)
entropy_son=(286/455)*entropy_left+(173/455)*entropy_right
gain=entropy_father-entropy_son
print(f"The entropy of the father:{entropy_father}")
print(f"The entropy of the left son:{entropy_left}")
print(f"The entropy of the right son:{entropy_right}")
print(f"The entropy of the sons:{entropy_son}")
print(f"The information gain:{gain}")
```

```
The entropy of the father:0.9517626756348311
The entropy of the left son:0.31435586359270684
The entropy of the right son:0.5165998933608694
The entropy of the sons:0.39401661217350453
The information gain:0.5577460634613265
```

The results show that in the first split of the decision tree, by using the feature concave_points1 <= 0.051, the entropy of the root node decreased from 0.95 to the weighted entropy of 0.300 after the split, with an information gain of 0.55. This indicates that concave_points1 is a very important feature, which can significantly reduce the uncertainty of the data and effectively distinguish samples of different categories. The entropy of the left node is 0.31 and that of the right node is 0.51, both lower than that of the root node, suggesting that this feature split can effectively divide the data into two relatively pure subsets. In the left node, the majority of samples belong to one category, while in the right node, although the category distribution still has some uncertainty, it has improved significantly compared to the root node.

2.3.2 Gini

Similar to entropy, the Gini index of the root node can be calculated as follows. However, since the decision tree is constructed based on the Gini index, which has already been provided, there is no need to perform the calculation again.

$$G(root) = 1 - \sum_0^1 p_i(1)^2 = 1 - (\frac{286}{455})^2 - (\frac{169}{455})^2 = 0.467$$

The Gini index of the left node and right node are as follows:

$$G(left) = 1 - \sum_0^1 p_i(1)^2 = 1 - (\frac{266}{282})^2 - (\frac{16}{282})^2 = 0.107$$

$$G(right) = 1 - \sum_0^1 p_i(1)^2 = 1 - (\frac{153}{173})^2 - (\frac{20}{173})^2 = 0.204$$

The Gini index of the first split is as follows:

$$G(son) = \frac{286}{455} G(left) + \frac{173}{455} G(right)$$

The information gain is as follows:

$$\Delta = G(father) - G(son)$$

```
gini_father=0.467
gini_left=0.107
gini_right=0.204
gini_son=(286/455)*gini_left+(173/455)*gini_right
gain=gini_father-gini_son
print(f"The gini index of the father:{gini_father}")
print(f"The gini index of the sons:{gini_son}")
print(f"The information gain:{gain}")
```

```
The gini index of the father:0.467
The gini index of the sons:0.144821978021978
The information gain:0.32217802197802203
```

In the first split of the decision tree, the Gini index of the root node is 0.467, indicating a certain degree of impurity in the class distribution at the root node. The Gini index of the left node is 0.107, and that of the right node is 0.204, both of which are lower than that of the root node. This suggests that the current feature split has significantly improved the class purity of the data, resulting in a more concentrated class distribution. The high information gain allows us to confirm that the current feature split is rational, as it effectively enhances the class distribution of the data and provides a solid foundation for the subsequent construction of the decision tree.

2.3.3 Misclassification Error

the Misclassification error of the root node can be calculated as follows:

$$M(root) = 1 - \max[p_i(1)] = 1 - \max[\frac{286}{455}, \frac{169}{455}]$$

The Misclassification error of the left node and right node are as follows:

$$M(\text{left}) = 1 - \max[p_i(1)] = 1 - \max[\frac{266}{282}, \frac{16}{282}]$$

$$M(\text{right}) = 1 - \max[p_i(1)] = 1 - \max[\frac{153}{173}, \frac{20}{173}]$$

The Misclassification error of the first split is as follows:

$$M(\text{son}) = \frac{286}{455} M(\text{left}) + \frac{173}{455} M(\text{right})$$

The information gain is as follows:

$$\Delta = M(\text{father}) - M(\text{son})$$

```
misclass_father=1-max(286/455, 169/455)
misclass_left=1-max(266/282, 16/282)
misclass_right=1-max(20/173, 153/173)
misclass_son=(286/455)*misclass_left+(173/455)*misclass_right
gain=misclass_father-misclass_son
print(f"The misclassification error of the father:{misclass_father}")
print(f"The misclassification error of the sons:{misclass_son}")
print(f"The information gain:{gain}")
```

```
The misclassification error of the father:0.37142857142857144
The misclassification error of the sons:0.07961967110903281
The information gain:0.29180890031953866
```

In the first split of the decision tree, the root node has a misclassification error of 0.371, while the left and right nodes have errors of 0.057 and 0.116, respectively. This shows that the current feature split significantly improves class purity, making the child nodes "purer" than the root node. The information gain of 0.285 indicates a successful reduction in misclassification error and enhanced classification accuracy, confirming that the feature split is effective and provides a strong basis for further decision tree construction.

2.3.4 Feature and Decision Value

In the first split of the decision tree, the feature selected is concave_points1, and the decision value is 0.051. This means that the decision tree uses the condition concave_points1 <= 0.051 to split the data into two subsets. This feature and value are chosen because they provide the best information gain or reduction in impurity (as measured by entropy, Gini index, or misclassification error) among all possible splits.

The selection of this feature and value indicates that concave_points1 is a highly informative attribute for distinguishing between the two classes in the dataset.

3. Problem 3

3.1 Data

The problem utilizes the same dataset as in Problem 2, specifically the Breast Cancer Wisconsin (Diagnostic) dataset. However, in this scenario, we are required to perform PCA dimensionality reduction on the data before constructing the binary decision tree.

```
from sklearn.decomposition import PCA

#first principal
pca=PCA(n_components=1)
X_pca_1=pca.fit_transform(X)

#second principal
pca=PCA(n_components=2)
X_pca_2=pca.fit_transform(X)
```

3.2 Binary Decision Trees

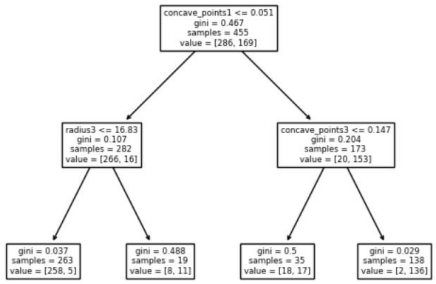
As required by the task, three decision trees need to be constructed: one using the original data, and the other two using the first principal component and the first and second principal components after PCA dimensionality reduction, respectively. To facilitate the subsequent parameter calculations and comparisons, I have placed all the decision tree models into a list.

```
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.2, random_state=42)
X_pca_1_train, X_pca_1_test, y_pca_1_train, y_pca_1_test=train_test_split(X_pca_1, y, test_size=0.2, random_state=42)
X_pca_2_train, X_pca_2_test, y_pca_2_train, y_pca_2_test=train_test_split(X_pca_2, y, test_size=0.2, random_state=42)

models=[]

model_original= DecisionTreeClassifier(min_samples_leaf=2, min_samples_split=5, max_depth=2).fit(X_train, y_train)
models.append(model_original)
tree=export_text(model_original,feature_names=list(X_train.columns))
print(tree)
```

```
|--- concave_points1 <= 0.05
|   |--- radius3 <= 16.83
|   |   |--- class: B
|   |--- radius3 >  16.83
|   |   |--- class: M
|--- concave_points1 >  0.05
|   |--- concave_points3 <= 0.15
|   |   |--- class: B
|   |--- concave_points3 >  0.15
|   |   |--- class: M
```

```
treePlot=plot_tree(model_original,feature_names=list(X_train.columns), filled = False)
print(treePlot)
```

```
[Text(0.5, 0.8333333333333334, 'concave_points1 <= 0.051\ngini = 0.467\nsamples = 455\nvalue = [286, 169]'), Text(0.25, 0.5, 'radius3 <= 16.83\ngini = 0.107\nsamples = 282\nvalue = [266, 16]'), Text(0.125, 0.16666666666666666, 'gini = 0.037\nsamples = 263\nvalue = [258, 5]'), Text(0.375, 0.16666666666666666, 'gini = 0.488\nsamples = 19\nvalue = [8, 11]'), Text(0.75, 0.5, 'concave_points3 <= 0.147\ngini = 0.204\nsamples = 173\nvalue = [20, 153]'), Text(0.625, 0.16666666666666666, 'gini = 0.5\nsamples = 35\nvalue = [18, 17]'), Text(0.875, 0.16666666666666666, 'gini = 0.029\nsamples = 138\nvalue = [2, 136]')]
```

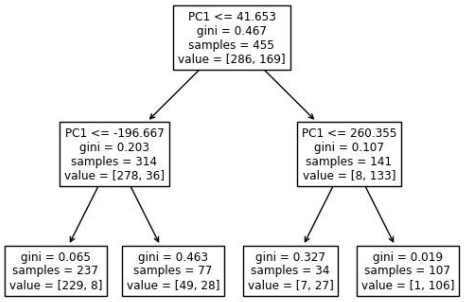



```
model_1= DecisionTreeClassifier(min_samples_leaf=2, min_samples_split=5, max_depth=2).fit(X_pca_1_train, y_pca_1_train)
models.append(model_1)
tree = export_text(model_1, feature_names=['PC1'])
print(tree)
```

```
|── PC1 <= 41.65
|   |── PC1 <= -196.67
|   |   |── class: B
|   |── PC1 >  -196.67
|   |   |── class: B
|── PC1 >  41.65
|   |── PC1 <= 260.35
|   |   |── class: M
|   |── PC1 >  260.35
|   |   |── class: M
```

```
treePlot = plot_tree(model_1, feature_names=['PC1'], filled=False)
print(treePlot)
```

```
[Text(0.5, 0.8333333333333334, 'PC1 <= 41.653\ngini = 0.467\nsamples = 455\nvalue = [286, 169]'), Text(0.25, 0.5, 'PC1 <= -196.667\ngini = 0.203\nsamples = 314\nvalue = [278, 36]'), Text(0.125, 0.16666666666666666, 'gini = 0.065\nsamples = 237\nvalue = [229, 8]'), Text(0.375, 0.16666666666666666, 'gini = 0.463\nsamples = 77\nvalue = [49, 28]'), Text(0.75, 0.5, 'PC1 <= 260.355\ngini = 0.107\nsamples = 141\nvalue = [8, 133]'), Text(0.625, 0.16666666666666666, 'gini = 0.327\nsamples = 34\nvalue = [7, 27]'), Text(0.875, 0.16666666666666666, 'gini = 0.019\nsamples = 107\nvalue = [1, 106]')]
```

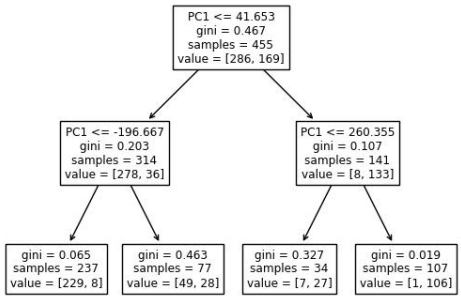


```
model_2= DecisionTreeClassifier(min_samples_leaf=2, min_samples_split=5, max_depth=2).fit(X_pca_2_train, y_pca_2_train)
models.append(model_2)
tree=export_text(model_2,feature_names=['PC1','PC2'])
print(tree)
```

```
|── PC1 <= 41.65
|   |── PC1 <= -196.67
|   |   |── class: B
|   |── PC1 >  -196.67
|   |   |── class: B
|── PC1 >  41.65
|   |── PC1 <= 260.35
|   |   |── class: M
|   |── PC1 >  260.35
|   |   |── class: M
```

```
treePlot = plot_tree(model_2, feature_names=['PC1','PC2'], filled=False)
print(treePlot)
```

```
[Text(0.5, 0.8333333333333334, 'PC1 <= 41.653\ngini = 0.467\nsamples = 455\nvalue = [286, 169]'), Text(0.25, 0.5, 'PC1 <= -196.667\ngini =
0.203\nsamples = 314\nvalue = [278, 36]'), Text(0.125, 0.16666666666666666, 'gini = 0.065\nsamples = 237\nvalue = [229, 8]'), Text(0.375, 0.
16666666666666666, 'gini = 0.463\nsamples = 77\nvalue = [49, 28]'), Text(0.75, 0.5, 'PC1 <= 260.355\ngini = 0.107\nsamples = 141\nvalue =
[8, 133]'), Text(0.625, 0.16666666666666666, 'gini = 0.327\nsamples = 34\nvalue = [7, 27]'), Text(0.875, 0.16666666666666666, 'gini = 0.019
\nsamples = 107\nvalue = [1, 106]')]
```



3.3 Related Analysis

3.3.1 F1 score, Precision and Recall

To evaluate the three models, I calculated the F1 score, precision, and recall for each model using a loop. Since each model had a different test set, I went back to the earlier steps and created lists for the test sets, which facilitated the looping process

```
f1_scores = []
precisions = []
recalls = []
for i in range(3):
    X_test_i = x_tests[i]
    y_test_i = y_tests[i]
    if isinstance(X_test_i, pd.DataFrame):
        X_test_i = X_test_i.values
    if isinstance(y_test_i, pd.Series):
        y_test_i = y_test_i.values

    y_pred = models[i].predict(X_test_i)

    f1 = f1_score(y_test_i, y_pred, pos_label='M')
    precision = precision_score(y_test_i, y_pred, pos_label='M')
    recall = recall_score(y_test_i, y_pred, pos_label='M')

    f1_scores.append(f1)
    precisions.append(precision)
    recalls.append(recall)

for i in range(3):
    print(f"model: {i+1}:")
    print(f"F1 score:{f1_scores[i]:.2f}")
    print(f"Accuraacy:{precisions[i]:.2f}")
    print(f"Recall:{recalls[i]:.2f}\n")
```

```
model: 1:
F1 score:0.90
Accuraacy:0.95
Recall:0.86

model: 2:
F1 score:0.92
Accuraacy:1.00
Recall:0.86

model: 3:
F1 score:0.92
Accuraacy:1.00
Recall:0.86
```

The results suggest that PCA dimensionality reduction, especially using the first principal component, can effectively capture the essential information for accurate classification. Model 2, which uses only the first principal component, achieves the best performance with perfect accuracy and a slightly higher F1 score compared to Model 1. Model 3, which includes the second principal component, does not significantly improve recall but maintains the high accuracy and precision. This implies that the first principal component alone is sufficient for achieving high performance in this classification task.

3.3.2 Confusion Matrix

I also used a for loop to construct the confusion matrix for each of the three models and calculated the FP, TP, FPR, and TPR, which were then outputted collectively.

```
from sklearn.metrics import confusion_matrix

fps = []
tps = []
fprs = []
tpRs = []

for i in range(3):
    y_pred = models[i].predict(x_tests[i])
    cm = confusion_matrix(y_tests[i], y_pred, labels=['B', 'M'])

    TN, FP, FN, TP = cm.ravel()
    FPR = FP / (FP + TN)
    TPR = TP / (TP + FN)

    fps.append(FP)
    tps.append(TP)
    fprs.append(FPR)
    tpRs.append(TPR)

for i in range(3):
    print(f"Model {i+1}:")
    print(f"False Positives (FP): {fps[i]}")
    print(f"True Positives (TP): {tps[i]}")
    print(f"False Positive Rate (FPR): {fprs[i]:.2f}")
    print(f"True Positive Rate (TPR): {tpRs[i]:.2f}\n")
```

```
Model 1:
False Positives (FP): 2
True Positives (TP): 37
False Positive Rate (FPR): 0.03
True Positive Rate (TPR): 0.86

Model 2:
False Positives (FP): 0
True Positives (TP): 37
False Positive Rate (FPR): 0.00
True Positive Rate (TPR): 0.86

Model 3:
False Positives (FP): 0
True Positives (TP): 37
False Positive Rate (FPR): 0.00
True Positive Rate (TPR): 0.86
```

The analysis of the confusion matrices for the three models reveals that Model 2,

which uses only the first principal component, achieves optimal performance with zero false positives and a true positive rate of 86%. This indicates that the first principal component alone is highly effective in capturing the essential information for accurate classification. Model 3, incorporating both the first and second principal components, maintains the same performance metrics as Model 2, suggesting that the second principal component does not add significant value in this context. Model 1, based on the original data, performs well but has a slightly higher false positive rate compared to Models 2 and 3. This suggests that while the original data is useful, PCA dimensionality reduction can achieve similar performance with reduced complexity. Overall, the results demonstrate the effectiveness of using the first principal component for accurate classification in this task.

3.3.3 Beneficial or Not

In this case, using continuous data is beneficial for the model, but the model based on PCA dimensionality reduction also demonstrates strong performance. The original data (continuous data) provides the model with rich details and information, enabling it to more accurately capture the characteristics of the positive class. For instance, Model 1 achieves a True Positive Rate (TPR) of 86%, despite having 2 False Positives (FP) and a False Positive Rate (FPR) of 0.03, which still indicates relatively good overall performance. However, Model 2, which employs only the first principal component after PCA, achieves zero False Positives and the same TPR of 86%. This shows that the first principal component can effectively condense the key information in the data, maintaining high classification accuracy even after dimensionality reduction. This suggests that in some cases, PCA dimensionality reduction can not only reduce computational complexity but also enhance the robustness of the model while preserving its performance.