ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

CƠ SỞ TRÍ TUỆ NHÂN TẠO – CSC14003_21CLC07

LAB 2

DECISION TREE

21127191 – Nguyễn Nhật Truyền

1. **Preparing the data sets:**

Scikit-learn (or sklearn) is an open-source machine learning library developed for Python, primarily designed to work with digitized data. This means that scikit-learn cannot directly handle string-type data, and you need to convert them into numerical form before using them in models.

In this project, we will use Label Encoding: Converting categorical string values into unique integer values to distinguish different values within the same feature or target value.

```
col_names = ["parents", "has_nurs", "form", "children", "housing", "finance", "social", "health", "class_values"]
mapping = {
    'parents': {'usual': 0, 'pretentious': 1, 'great_pret': 2},
    'has_nurs': {'proper': 0, 'less_proper': 1, 'improper': 2, 'critical': 3, 'very_crit': 4},
    'form': {'complete': 0, 'completed': 1, 'incomplete': 2, 'foster': 3},
    'children': {'1': 1, '2': 2, '3': 3, 'more': 4},
    'housing': {'convenient': 0, 'less_conv': 1, 'critical': 2},
    'finance': {'convenient': 0, 'inconv': 1},
    'social': {'nonprob': 0, 'slightly_prob': 1, 'problematic': 2},
    'health': {'recommended': 0, 'priority': 1, 'not_recom': 2},
    'class_values': {'not_recom': 0, 'recommend': 1, 'very_recom': 2, 'priority': 3, 'spec_prior': 4}
}
datasets = pd.read_csv('nursery.data.csv', names=col_names)

datasets = datasets.replace(mapping)
datasets = datasets.to_numpy()

label = datasets[:, -1]
datasets = datasets[:, :-1]
```

col_names, which is a list of column names, and mapping, which is a nested dictionary that maps categorical values to numerical representations for each column. The mapped values from corresponding columns for each feature are sequentially assigned as 0, 1, 2,... as shown in the code snippet above.

After the *datasets* have been mapped, we separate them into *label* and *datasets* corresponding to target values and features, respectively.

```
train_test_split(datasets, label,test_size = 0.6, train_size = 0.4, shuffle = True, random_state = 0)
train_test_split(datasets, label,test_size = 0.4, train_size = 0.6, shuffle = True, random_state = 0)
train_test_split(datasets, label,test_size = 0.2, train_size = 0.8, shuffle = True, random_state = 0)
train_test_split(datasets, label,test_size = 0.1, train_size = 0.9, shuffle = True, random_state = 0)
```

Then, we use the **train_test_split** function from the **sklearn** library to divide our data into portions for training and testing, with the following ratios: 40/60 (0.4 for train size), 60/40 (0.6 for train size), 80/20 (0.8 for train size), and 90/10 (0.9 for train size). We also add the shuffle parameter to ensure that each time we split, the data is shuffled as required.

Finally, we use the **plt** library to visualize the data correlation after splitting (refer to the code file for more details).

2. **Building the decision tree classifiers:**

Once we have the separated dataset, we use the *DecisionTreeClassifier* function to build a decision tree, and we add the parameter criterion='entropy' to measure the level of impurity. (The result is saved in the code file as a decision tree). (refer to the code file for more details).

3. **Evaluating the decision tree classifiers:**

In this request, we use the *classification_report* and *ConfusionMatrixDisplay.from_predictions* to perform the task, and the obtained results are as follows.

<table>
<tr>
<td>40/60</td>
<td>
<table>
<tr><th></th><th>precision</th><th>recall</th><th>f1-score</th><th>support</th></tr>
<tr><td>not_recom</td><td>1.00</td><td>1.00</td><td>1.00</td><td>2593</td></tr>
<tr><td>recommend</td><td>0.00</td><td>0.00</td><td>0.00</td><td>0</td></tr>
<tr><td>very_recom</td><td>0.99</td><td>0.94</td><td>0.97</td><td>177</td></tr>
<tr><td>priority</td><td>0.98</td><td>0.99</td><td>0.99</td><td>2539</td></tr>
<tr><td>spec_prior</td><td>0.99</td><td>0.99</td><td>0.99</td><td>2467</td></tr>
<tr><td>accuracy</td><td></td><td></td><td>0.99</td><td>7776</td></tr>
<tr><td>macro avg</td><td>0.79</td><td>0.78</td><td>0.79</td><td>7776</td></tr>
<tr><td>weighted avg</td><td>0.99</td><td>0.99</td><td>0.99</td><td>7776</td></tr>
</table>
</td>
<td>Precision: The "recommend" and "very_recom" classes have low precision, even down to 0. Other classes have high precision (1.00).
Recall: The "very_recom" and "recommend" classes also have low recall, indicating poor predictive ability for these classes. Other classes have high recall.
F1-score: The "recommend" and "very_recom" classes have low F1-score. Other classes have high F1-score.
Support: The "recommend" class has no data (support = 0). Other classes have support ranging from 177 to 2539.</td>
</tr>
<tr>
<td>60/40</td>
<td>
<table>
<tr><th></th><th>precision</th><th>recall</th><th>f1-score</th><th>support</th></tr>
<tr><td>not_recom</td><td>1.00</td><td>1.00</td><td>1.00</td><td>1744</td></tr>
<tr><td>recommend</td><td>0.00</td><td>0.00</td><td>0.00</td><td>0</td></tr>
<tr><td>very_recom</td><td>1.00</td><td>0.97</td><td>0.98</td><td>119</td></tr>
<tr><td>priority</td><td>0.99</td><td>1.00</td><td>1.00</td><td>1668</td></tr>
<tr><td>spec_prior</td><td>1.00</td><td>1.00</td><td>1.00</td><td>1653</td></tr>
<tr><td>accuracy</td><td></td><td></td><td>1.00</td><td>5184</td></tr>
<tr><td>macro avg</td><td>0.80</td><td>0.79</td><td>0.80</td><td>5184</td></tr>
<tr><td>weighted avg</td><td>1.00</td><td>1.00</td><td>1.00</td><td>5184</td></tr>
</table>
</td>
<td>Precision: The "recommend" and "very_recom" classes still have low or even 0 precision. Other classes have high precision.
Recall: The "very_recom" and "recommend" classes have increased recall compared to the 40/60 ratio, but it's still low. Other classes have high recall.
F1-score: The "recommend" and "very_recom" classes have higher F1-score compared to the 40/60 ratio, but it's still low. Other classes have high F1-score.
Support: The "recommend" class has no data (support = 0). Other classes have support ranging from 119 to 1668.</td>
</tr>
<tr>
<td>80/20</td>
<td>
<table>
<tr><th></th><th>precision</th><th>recall</th><th>f1-score</th><th>support</th></tr>
<tr><td>not_recom</td><td>1.00</td><td>1.00</td><td>1.00</td><td>882</td></tr>
<tr><td>recommend</td><td>0.00</td><td>0.00</td><td>0.00</td><td>0</td></tr>
<tr><td>very_recom</td><td>1.00</td><td>1.00</td><td>1.00</td><td>57</td></tr>
<tr><td>priority</td><td>1.00</td><td>1.00</td><td>1.00</td><td>833</td></tr>
<tr><td>spec_prior</td><td>1.00</td><td>1.00</td><td>1.00</td><td>820</td></tr>
<tr><td>micro avg</td><td>1.00</td><td>1.00</td><td>1.00</td><td>2592</td></tr>
<tr><td>macro avg</td><td>0.80</td><td>0.80</td><td>0.80</td><td>2592</td></tr>
<tr><td>eighted avg</td><td>1.00</td><td>1.00</td><td>1.00</td><td>2592</td></tr>
</table>
</td>
<td>Precision: Precision of the "recommend" and "very_recom" classes is still low. Other classes have high precision.
Recall: Recall of the "very_recom" and "recommend" classes is still low. Other classes have high recall.
F1-score: The "recommend" and "very_recom" classes have low F1-score. Other classes have high F1-score.
Support: The "recommend" class has no data (support = 0). Other classes have support ranging from 25 to 1653.</td>
</tr>
<tr>
<td>90/10</td>
<td>
<table>
<tr><th></th><th>precision</th><th>recall</th><th>f1-score</th><th>support</th></tr>
<tr><td>not_recom</td><td>1.00</td><td>1.00</td><td>1.00</td><td>443</td></tr>
<tr><td>recommend</td><td>0.00</td><td>0.00</td><td>0.00</td><td>0</td></tr>
<tr><td>very_recom</td><td>1.00</td><td>1.00</td><td>1.00</td><td>25</td></tr>
<tr><td>priority</td><td>1.00</td><td>1.00</td><td>1.00</td><td>410</td></tr>
<tr><td>spec_prior</td><td>1.00</td><td>1.00</td><td>1.00</td><td>418</td></tr>
<tr><td>micro avg</td><td>1.00</td><td>1.00</td><td>1.00</td><td>1296</td></tr>
<tr><td>macro avg</td><td>0.80</td><td>0.80</td><td>0.80</td><td>1296</td></tr>
<tr><td>weighted avg</td><td>1.00</td><td>1.00</td><td>1.00</td><td>1296</td></tr>
</table>
</td>
<td>Precision: Precision of the "recommend" and "very_recom" classes is still low. Other classes have high precision.
Recall: Recall of the "very_recom" and "recommend" classes is still low. Other classes have high recall.
F1-score: The "recommend" and "very_recom" classes have low F1-score. Other classes have high F1-score.
Support: The "recommend" class has no data (support = 0). Other classes have support ranging from 25 to 1653.</td>
</tr>
</table>

Observations: The "recommend" and "very_recom" classes exhibit low precision, recall, and F1-score, with support being 0 (for the "recommend" class). Other classes show better performance, and the support varies depending on the train/test ratio. Overall, the model's performance doesn't change significantly with different train/test ratios. However, the model may still struggle to predict minority classes like "recommend" and "very_recom".

Across different train/test ratios (40/60, 60/40, 80/20, 90/10), the micro avg, macro avg, and weighted avg consistently have relatively high values, indicating that the model's overall predictive capability aligns well with the actual labels.

4. **The depth and accuracy of a decision tree:**

We will proceed similarly to Part 2 on the dataset with an 80/20 train/test ratio. However, this time we will add a depth limit to the experiment. Subsequently, we will calculate the *accuracy_score* and record the results.

Report to the following table the *accuracy_score* (on the test set) of the decision tree classifier when changing *max_depth*:

max_depth	None	2	3	4	5	6	7
accuracy_score	0.99845679012345 68	0.82947530864197 53	0.85069444444444 44	0.86419753086419 75	0.89390432098765 43	0.91550925925925 93	0.94020061728395 07

Comment on the above statistics:

The provided table shows the accuracy scores of a decision tree classifier on a test set as the max_depth hyperparameter is varied. The max_depth hyperparameter controls the maximum depth of the decision tree, which in turn affects its complexity and ability to capture intricate patterns in the data. Let's analyze the statistics presented:

When max_depth is set to None: The accuracy score is very high, at 0.998, indicating that the decision tree model is likely overfitting the training data. A very high accuracy on the test set can be a sign of overfitting, where the model has learned the training data too well, including noise, and doesn't generalize well to new, unseen data.

As max_depth increases from 2 to 7: The accuracy score on the test set generally improves. This is expected, as increasing the depth of the decision tree allows it to capture more complex relationships in the data. However, it's worth noting that the increase in accuracy starts to slow down as max_depth increases.

Notable accuracy jumps occur at certain max_depth values: The accuracy score jumps from 0.829 (at max_depth 2) to 0.850 (at max_depth 3), then to 0.864 (at max_depth 4), and further improves to 0.893 (at max_depth 5). These jumps suggest that increasing the depth up to a certain point is helping the model better represent the underlying patterns in the data.

The accuracy improvement slows down at higher depths: The accuracy improvements become smaller as max_depth increases beyond 5. This indicates that increasing the depth beyond a certain point is providing diminishing returns in terms of predictive performance on the test set.

Accuracy continues to increase: The accuracy scores keep increasing as max_depth goes beyond 5, reaching 0.915 at max_depth 6 and 0.940 at max_depth 7. However, it's important to be cautious with overly deep trees, as they can become too specialized to the training data and may not generalize well to new data.

Overall, the statistics suggest a classic case of the bias-variance trade-off. A very shallow tree (max_depth 2) has high bias and may not capture the underlying relationships well, resulting in lower accuracy. As the tree becomes deeper, it captures more details from the training data, reducing bias and potentially increasing accuracy. However, there's a point at which increasing complexity leads to overfitting, causing the model's performance on unseen data to degrade.

It's important to consider these results in the context of the problem you're trying to solve and potentially use techniques like cross-validation to choose the optimal max_depth that balances between model complexity and generalization.