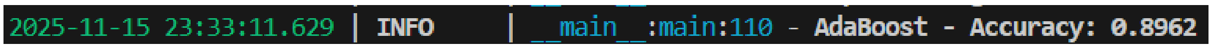
NYCU Introduction to Machine Learning, Homework 3

112550020, 蔡懷恩

Part. 1, Coding (60%):

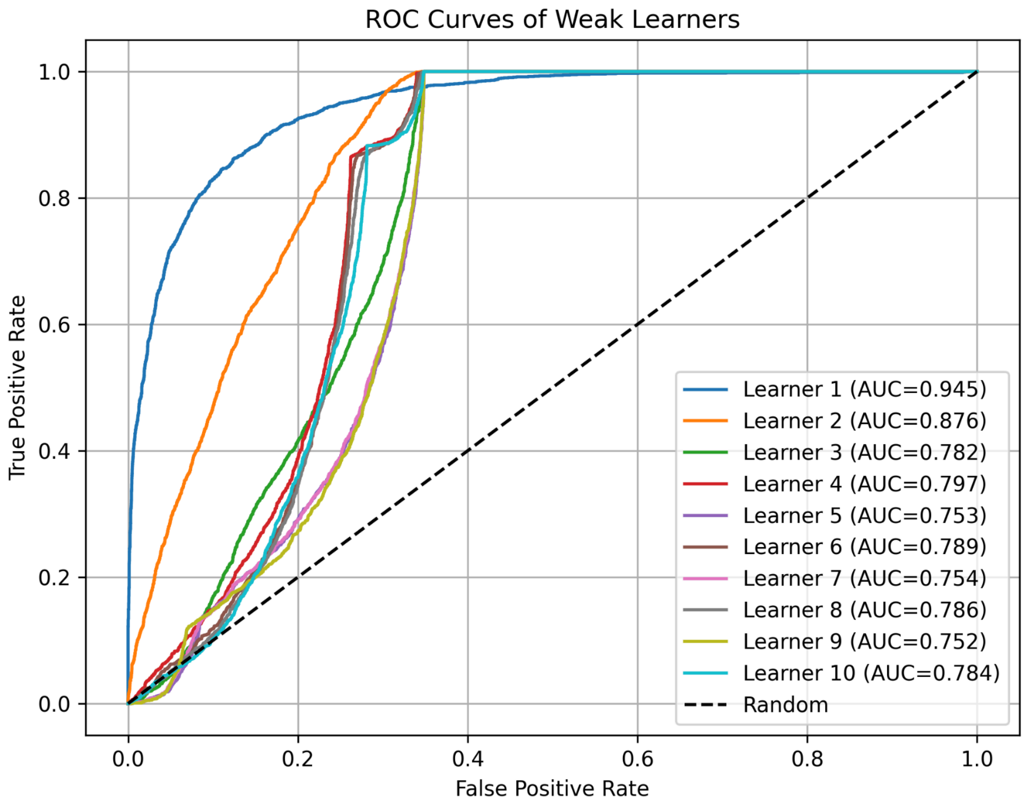
(20%) Adaboost

1. (5%) Show your accuracy of the testing data (n_estimators = 10)

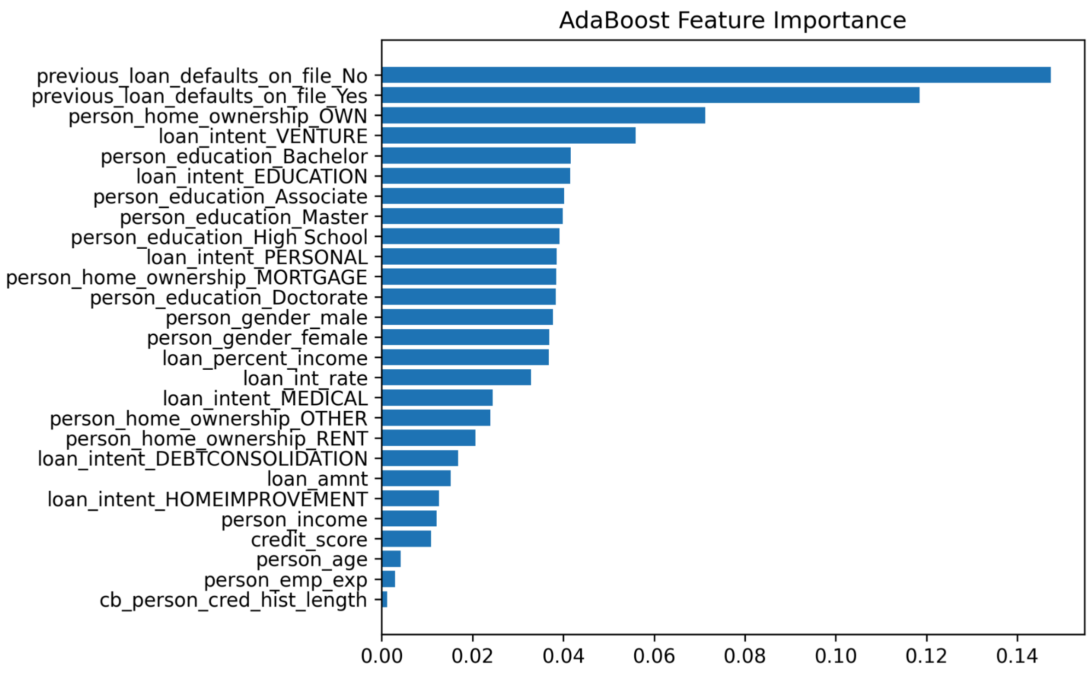


```
2025-11-15 23:33:11.629 | INFO     | __main__:main:110 - AdaBoost - Accuracy: 0.8962
```

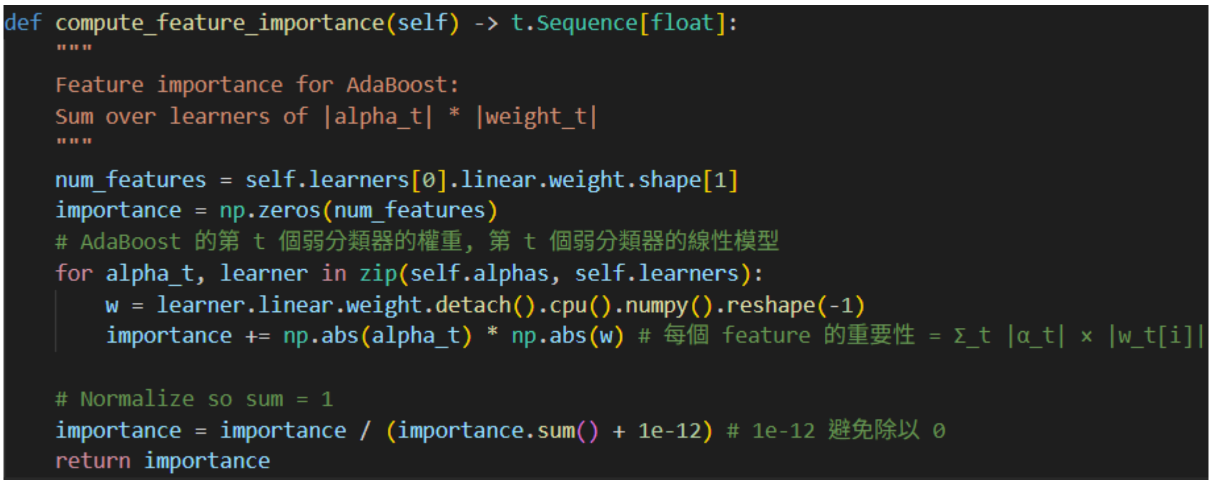
2. (5%) Plot the AUC curves of <u>each</u> weak classifier.



3. (10%)
   a. Plot the feature importance of the AdaBoost method.



b. Paste the snapshot of your implementation of the feature importance estimation.



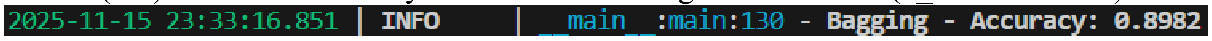
```python
def compute_feature_importance(self) -> t.Sequence[float]:
    """
    Feature importance for AdaBoost:
    Sum over learners of |alpha_t| * |weight_t|
    """
    num_features = self.learners[0].linear.weight.shape[1]
    importance = np.zeros(num_features)
    # AdaBoost 的第 t 個弱分類器的權重, 第 t 個弱分類器的線性模型
    for alpha_t, learner in zip(self.alphas, self.learners):
        w = learner.linear.weight.detach().cpu().numpy().reshape(-1)
        importance += np.abs(alpha_t) * np.abs(w) # 每個 feature 的重要性 = Σ_t |α_t| × |w_t[i]|

    # Normalize so sum = 1
    importance = importance / (importance.sum() + 1e-12) # 1e-12 避免除以 0
    return importance
```

c. Explain how you compute the feature importance for the Adaboost method shortly (within 100 words as possible)

I compute feature importance by summing the contribution of each weak learner.
For each learner, I extract its linear weight vector and multiply it by the learner’s coefficient α_t.
The absolute weighted weights are accumulated across all learners, and the final importance is normalized so that all feature importances sum to 1.

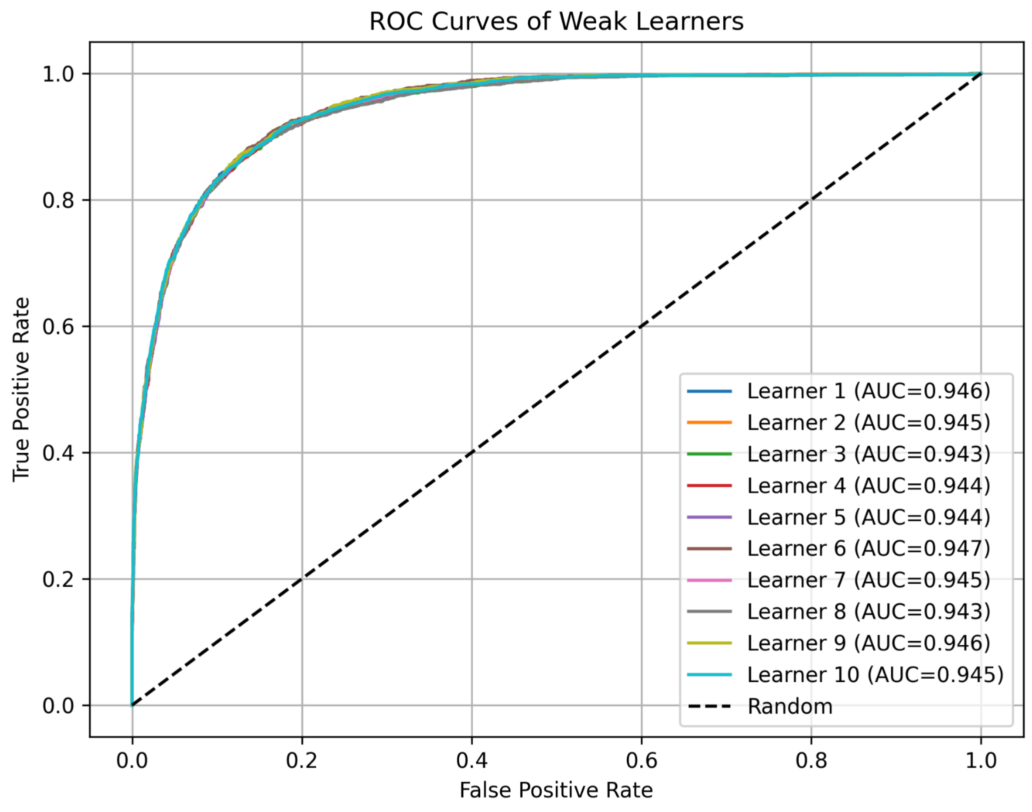
(20%) Bagging

4. (5%) Show the accuracy of the test data using 10 estimators. (n_estimators=10)



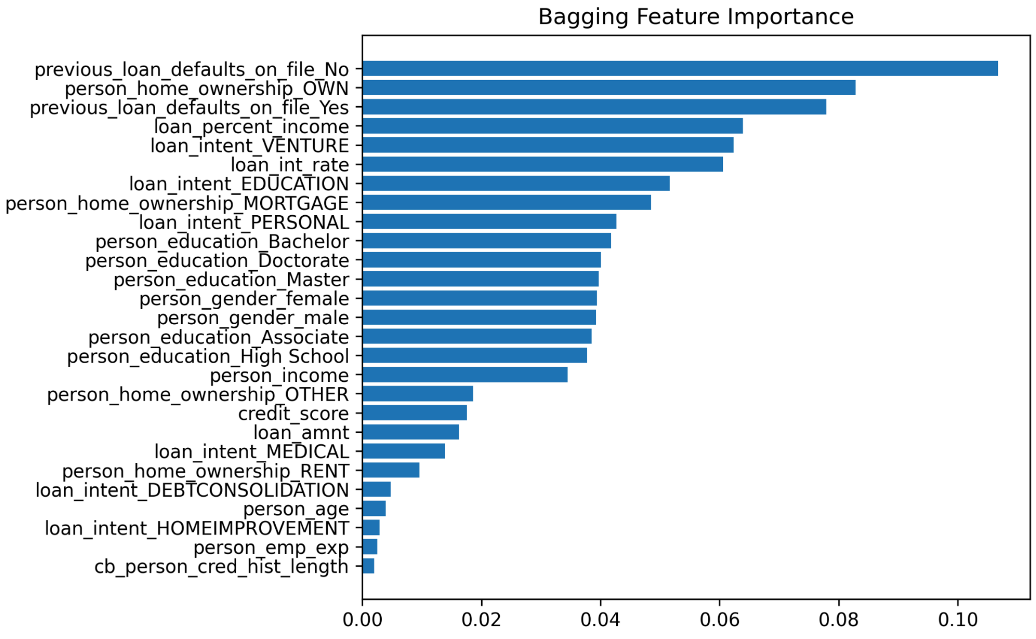
```
2025-11-15 23:33:16.851 | INFO     | __main__:main:130 - Bagging - Accuracy: 0.8982
```

5. (5%) Plot the AUC curves of each weak classifier.

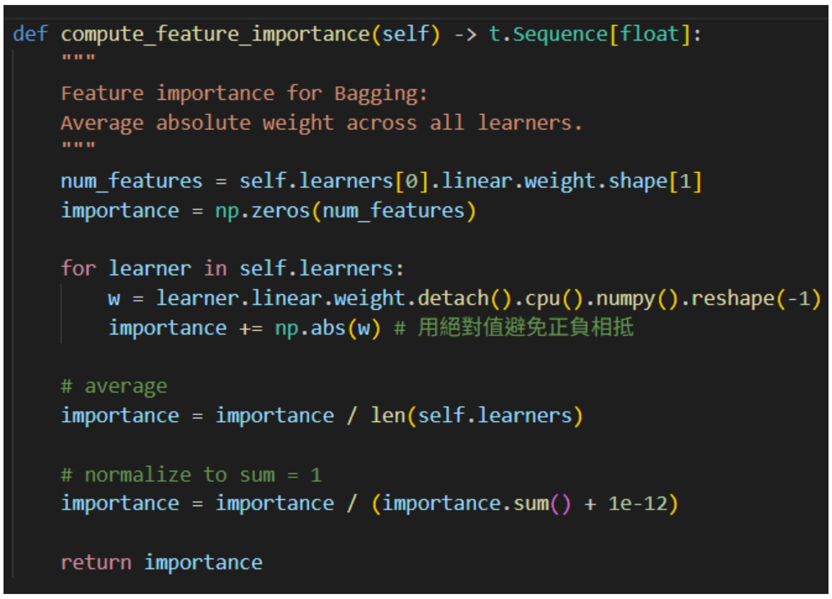


6. (10%)

   a. Plot the feature importance of the Bagging method.



   b. Paste the snapshot of your implementation of the feature importance estimation.



```python
def compute_feature_importance(self) -> t.Sequence[float]:
    """
    Feature importance for Bagging:
    Average absolute weight across all learners.
    """
    num_features = self.learners[0].linear.weight.shape[1]
    importance = np.zeros(num_features)

    for learner in self.learners:
        w = learner.linear.weight.detach().cpu().numpy().reshape(-1)
        importance += np.abs(w) # 用絕對值避免正負相抵

    # average
    importance = importance / len(self.learners)

    # normalize to sum = 1
    importance = importance / (importance.sum() + 1e-12)

    return importance
```

   c. Explain how you compute the feature importance for the Bagging method shortly (within 100 words as possible)

   I compute the Bagging feature importance by averaging the absolute weights of all base learners.
   For each learner, I extract its linear weight vector, take the absolute value, and accumulate it.
   The sum is then divided by the number of learners to obtain the average contribution of each feature.
   Finally, the importance vector is normalized so that all feature importances sum to 1.

(15%) Decision Tree

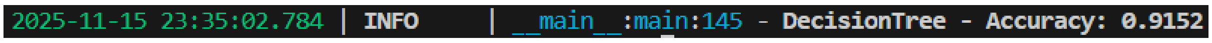
7. (5%) Compute the Gini index and the entropy of the array [0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1].

   $p_0 = \frac{6}{11}, p_1 = \frac{5}{11}$

   Entropy $= -p_0 log_2(p_0) - p_1 log_2(p_1) = -\frac{6}{11} log_2(\frac{6}{11}) - \frac{5}{11} log_2(\frac{5}{11}) \approx 0.994$

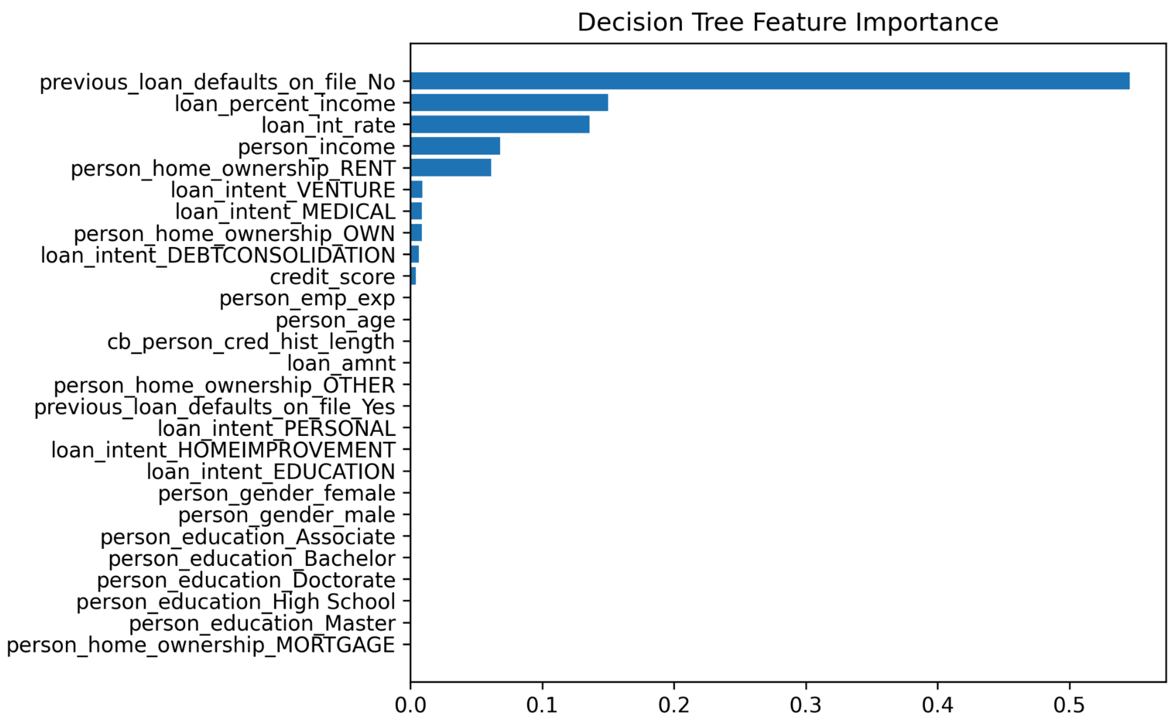
   Gini index $= 1 - {p_0}^2 - {p_1}^2 = 1 - \frac{36}{121} - \frac{25}{121} \approx 0.496$

8. (5%) Show your accuracy of the testing data with a max-depth = 7



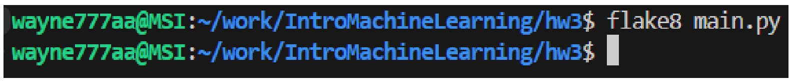
```
2025-11-15 23:35:02.784 | INFO     | __main__:main:145 - DecisionTree - Accuracy: 0.9152
```

9. (5%) Plot the feature importance of the decision tree.



(5%) Code Linting

10. Show the snapshot of the flake8 linting result (paste the execution command even when there is no error).



```
wayne777aa@MSI:~/work/IntroMachineLearning/hw3$ flake8 main.py
wayne777aa@MSI:~/work/IntroMachineLearning/hw3$
```

Part. 2, Questions (40%):

1. (15%)

In the AdaBoost algorithm, each selected weak classifier, h_t is assigned a weight:

$$\alpha_t = \frac{1}{2} ln \frac{1-\epsilon_t}{\epsilon_t}$$

(a) What is the definition of ϵ_t in this formula?

(b) If a weak classifier h_t performs only as well as "random guessing" (i.e., $\epsilon_t = 0.5$), what will α_t be?

(c) Based on your answer in (b), briefly explain how this α_t weight formula ensures the robustness of AdaBoost.

(a) $$\mathcal{E}_t = \sum_{i=1}^{N} W_i^{(t)} \left[y_i \neq h_t(x_i) \right]$$

$W_i^{(t)}$: The weight of sample i at iteration t

$h_t(x_i)$: The prediction of the weak classifier on sample x_i

y_i : The true label of sample i

ε_t is computed as the weighted classification error of the weak learner h_t at iteration t.
It is equal to the sum of the sample weights of all misclassified training samples.

(b) $$\alpha_t = \frac{1}{2} \ln \frac{1-0.5}{0.5} = 0$$

(c)
If a classifier performs no better than random guessing (ϵ□ = 0.5), its weight α□ becomes zero, meaning it contributes nothing to the final ensemble.
Better classifiers (ϵ□ < 0.5) get positive α□, so their influence is amplified.
Thus, AdaBoost suppresses useless weak learners and emphasizes informative ones, making the final model robust.

2. (15%) Random Forest is often considered an improvement over Bagging (Bootstrap Aggregating) when used with Decision Trees as the base learners.

(a) Briefly explain the potential problem that can exist among the T classifiers (C_1, ..., C_T) produced by Bagging when using decision trees.

(b) To mitigate the problem from (a), Random Forest introduces a second source of randomness in addition to Bagging. Specifically describe what this second source of randomness is and how it works.

(a)

When Bagging uses decision trees as base learners, the T classifiers can become highly correlated.

Because all trees see many similar samples and can freely choose any feature at every split, they tend to pick the same dominant features and form similar tree structures. This reduces the benefit of bagging and limits variance reduction.

(b)

Random Forest introduces a second source of randomness: **Random vector** at each split.

Instead of considering all features, each tree only chooses the best split from a random subset of features. This makes each classifier more diverse. The increased diversity reduces variance and improves overall performance.

3. (10%) Is it always possible to find the smallest decision tree that codes a dataset with no error in polynomial time? If not, what algorithm do we usually use to search a decision tree in a reasonable time? Also, list the criterion that we stop splitting the decision tree and leaf nodes are created (List two).

No, finding the smallest decision tree with zero training error is NP-complete, so it cannot be solved in polynomial time.
we use **greedy algorithms** to build a decision tree efficiently.

Stopping criteria:

1. The data subset is pure.
2. Maximum tree depth is reached.