NYCU Introduction to Machine Learning, Homework 3

[112550198], [簡嫚萱]

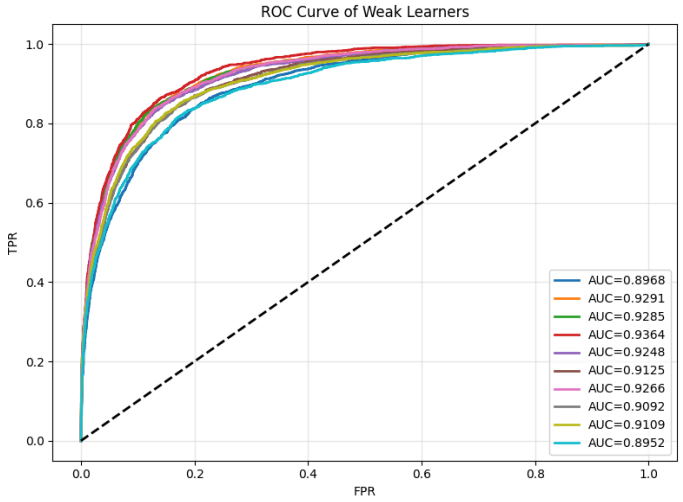
Part. 1, Coding (60%):

(20%) Adaboost

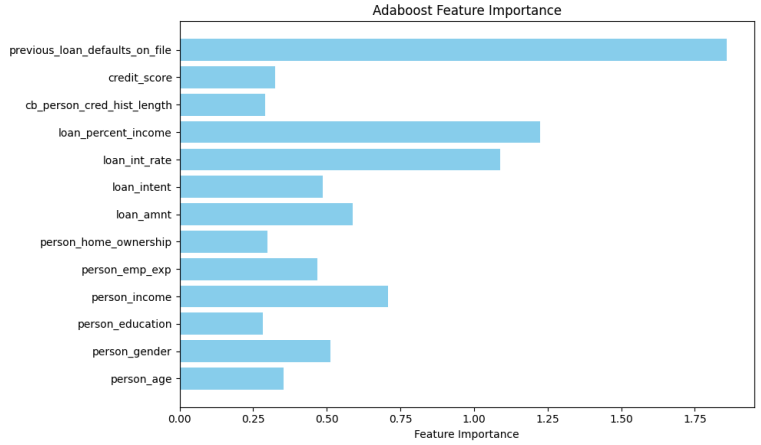
1. (5%) Show your accuracy of the testing data (n_estimators = 10)

```
2025-11-16 22:53:30.226 | INFO     | __main__:main:42 - AdaBoost - Accuracy: 0.8626
```

2. (5%) Plot the AUC curves of each 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 = None

    for alpha, learner in zip(self.alphas, self.learners):
        w1 = learner.fc1.weight.data  # (hidden, input)
        w2 = learner.fc2.weight.data.view(-1)  # (1, hidden) -> (hidden,)

        # calculate sum of each column
        learner_feature = torch.sum(torch.abs(w1) * torch.abs(w2).unsqueeze(1), dim=0)
        learner_feature = abs(alpha) * learner_feature

        if feature_importance is None:
            feature_importance = learner_feature
        else:
            feature_importance += learner_feature

    return feature_importance.cpu().numpy()
```

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

In AdaBoost, each weak learner has a weight α representing its influence on the final prediction. To compute feature importance:

1. Taking the first-layer weights w_1 (feature → hidden)
2. Taking the second-layer weights w_2 (hidden → output)
3. Computing $|w_1| \times |w_2|$ to measure each feature's contribution
4. Multiply this by the learner's weight α to account for its importance
5. Sum contributions from all learners to get a weighted total.

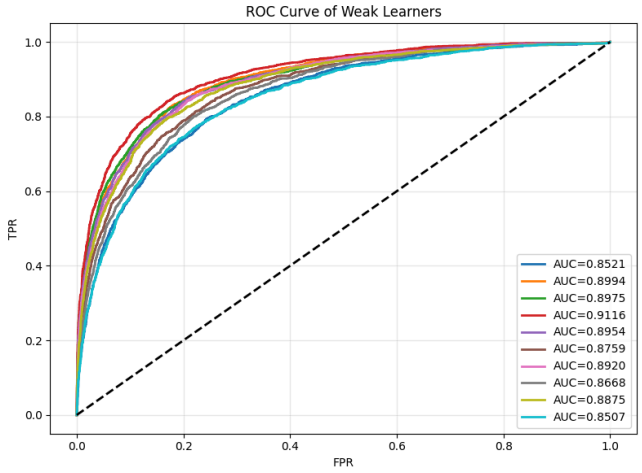
The result is a vector reflecting **how much each input feature contributes to the final AdaBoost ensemble prediction**.

(20%) Bagging

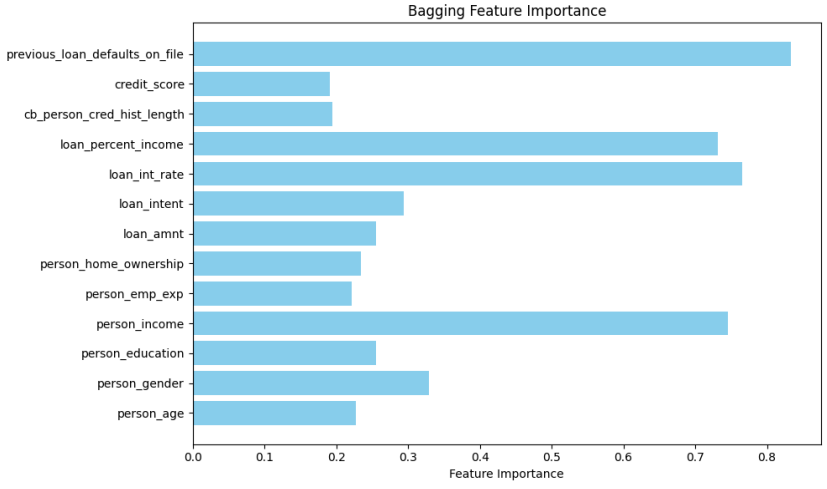
4. (5%) Show the accuracy of the test data using 10 estimators. (n_estimators=10)

```
2025-11-16 19:34:14.338 | INFO     | __main__:main:65 - Bagging - Accuracy: 0.8626
```

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_importances = None
    for learner in self.learners:
        w1 = learner.fc1.weight.data
        w2 = learner.fc2.weight.data.view(-1)
        learner_importance = torch.sum(torch.abs(w1) * torch.abs(w2).unsqueeze(1), dim=0)

        if feature_importances is None:
            feature_importances = learner_importance
        else:
            feature_importances += learner_importance

    return (feature_importances / len(self.learners)).cpu().numpy()
```

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

In Bagging, the overall feature importance is computed by **averaging the feature importance scores from all base learners**. For a neural-network learner, a feature's importance can be estimated from how strongly it influences the output. The code does this by:

1. Taking the first-layer weights w_1 (feature → hidden)
2. Taking the second-layer weights w_2 (hidden → output)
3. Computing $|w_1| \times |w_2|$ to measure each feature's contribution.
4. Summing across hidden units and averaging across all learners.

This yields the final Bagging feature importance vector.

(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].

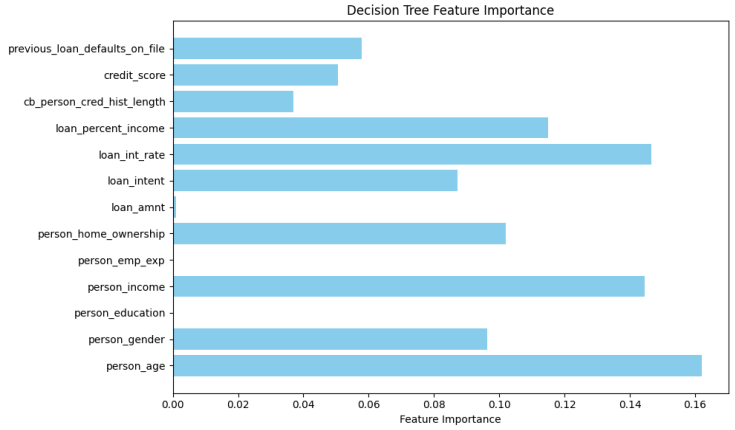
$$Gini = 1 - \sum_i {p_i}^2 = 1 - (\frac{6}{11})^2 - (\frac{5}{11})^2 = \frac{60}{121} \approx 0.496$$

$$Entropy = -\sum_i p_i log_2(p_i) = -[(\frac{6}{11})log_2(\frac{6}{11}) + (\frac{5}{11})log_2(\frac{5}{11})] \approx 0.994$$

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

```
2025-11-16 19:36:54.139 | INFO     | __main__:main:82 - DecisionTree - Accuracy: 0.9167
```

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).

```
(hw3) michelle@LAPTOP-E7SHGCCE:~/Intro.ML/hw3/src$ flake8 adaboost.py
(hw3) michelle@LAPTOP-E7SHGCCE:~/Intro.ML/hw3/src$ flake8 bagging.py
(hw3) michelle@LAPTOP-E7SHGCCE:~/Intro.ML/hw3/src$ flake8 decision_tree.py
(hw3) michelle@LAPTOP-E7SHGCCE:~/Intro.ML/hw3/src$ flake8 utils.py
```

```
(hw3) michelle@LAPTOP-E7SHGCCE:~/Intro.ML/hw3$ flake8 main.py
(hw3) michelle@LAPTOP-E7SHGCCE:~/Intro.ML/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) ϵ_t represents the weighted error of the weak classifier, calculated as the sum of the weights of the training examples that it misclassifies. In other words, it measures how many “important” samples the weak learner gets wrong, taking into account the current distribution of sample weights. It is defined as: $\epsilon_t = \sum_{i=1}^{m} D_t(i)[y_i \neq h_t(x_i)]$ where m is the total number of training samples, $D_t(i)$ is the weight of sample i at iteration t, $h_t(x_i)$ is the prediction of the weak classifier h_t on sample x_i and y_i is the true label of sample i.

(b) If a weak classifier performs as well as random guessing (i.e., its error rate ϵ_t =0.5), then its weight α_t will be zero. This means the classifier does not contribute to the final ensemble, as AdaBoost only gives higher influence to classifiers that perform better than chance.

(c) It ensures robustness of AdaBoost by assigning higher weights to weak classifiers that perform better than random guessing and zero or negative weight to classifiers that perform poorly. This way, AdaBoost focuses more on reliable classifiers, reducing the influence of weak or misleading ones and improving the overall ensemble performance.

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) Even though Bagging uses bootstrap samples (random subsets of the data with replacement), if the dataset has a few very strong or "dominant" features, most of the trees will still tend to split on those same dominant features first. This makes the trees look very similar, i.e., they become correlated, because the strong features dominate the decision paths.

(b) Aside from randomly sampling datasets (bagging), Random Forest introduces a second source of randomness by selecting a random subset of features at each split in the decision tree. This means that instead of considering all features when deciding the best split, each node only evaluates a random subset of features. This forces different trees to make splits on different features, reducing correlation among the trees and improving the ensemble's robustness.

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 is NP-complete, so it cannot be guaranteed in polynomial time. In practice, **greedy top-down algorithms** are used to construct trees efficiently. We typically stop splitting and create leaf nodes when **all examples in a subset belong to the same class (pure)** or when **the tree reaches a predefined maximum depth**.