NYCU Introduction to Machine Learning, Homework 3

[111550196], [狄豪飛 Jorge Tyrakowski]

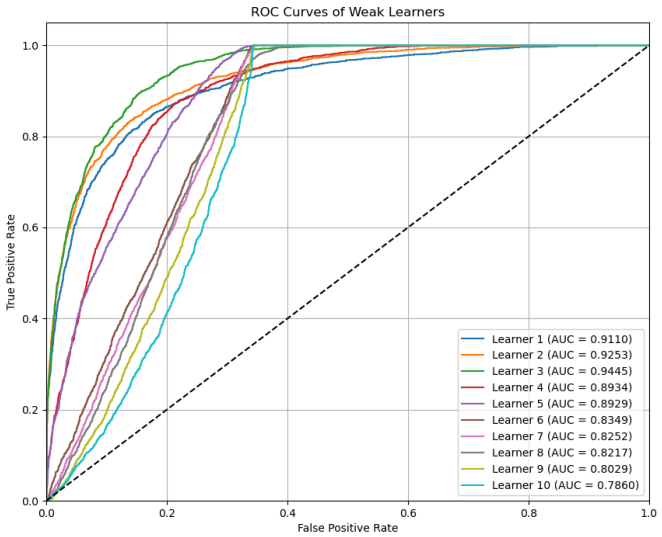
Part. 1, Coding (60%):

(20%) Adaboost

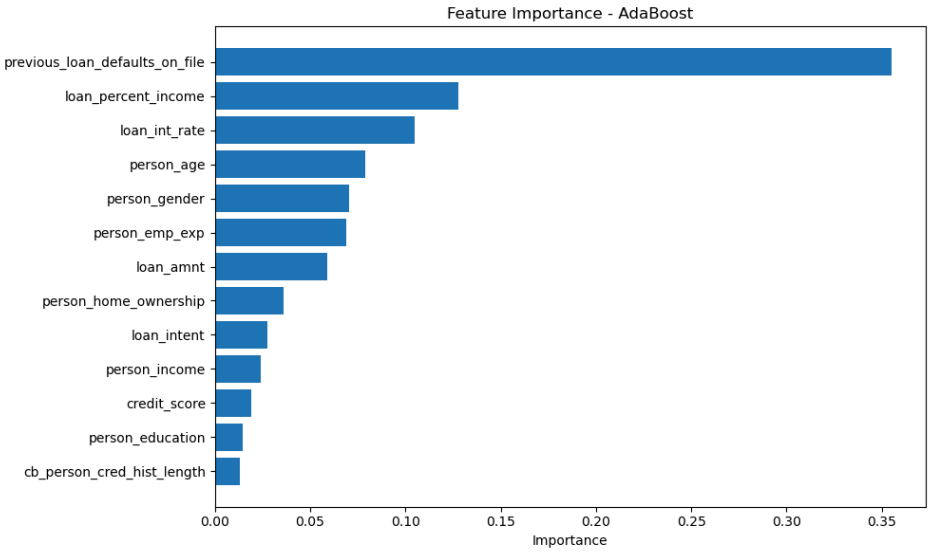
1. (10%) Show your accuracy of the testing data (n_estimators = 10)

```
(ML-NYCU) (base) dihaofei@dihaofei:~/NYCU/Machine Learning/HW03$ /home/dihaofei/conda-envs/ML-NYCU/
rning/HW03/main.py"
2024-11-19 19:33:29.687 | INFO     | __main__:main:55 - AdaBoost - Accuracy: 0.8903
2024-11-19 19:33:36.473 | INFO     | __main__:main:84 - Bagging - Accuracy: 0.8692
2024-11-19 19:35:15.834 | INFO     | __main__:main:109 - DecisionTree - Accuracy: 0.9044
```

2. (5%) Plot the AUC curves of each weak classifier.



3. (5%) Plot the feature importance of the AdaBoost method. Also, you should snapshot the implementation to calculate the feature importance.



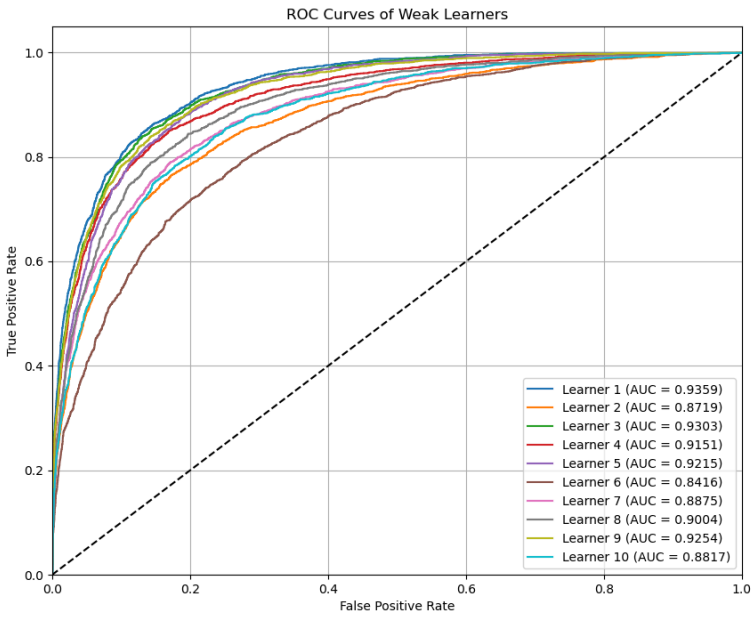
```python
def compute_feature_importance(self) -> t.Sequence[float]:
    """
    Compute feature importance for AdaBoost with two-layer model

    For a two-layer model, we compute importance by considering how each input
    feature contributes to the final output through both layers.

    Returns:
        t.Sequence[float]: Feature importance scores normalized to sum to 1
    """
    importance_scores = np.zeros(self.learners[0].layer1.weight.shape[1])
    total_alpha = sum(abs(alpha) for alpha in self.alphas)

    for model, alpha in zip(self.learners, self.alphas):
        # Get weights from both layers
        W1 = model.layer1.weight.data.numpy()  # (hidden_dim, input_dim)
        W2 = model.layer2.weight.data.numpy()  # (1, hidden_dim)

        # Compute contribution of each feature through both layers
        # W2: (1, hidden_dim), W1: (hidden_dim, input_dim)
        W2 = W2.reshape(-1)  # Convert to (hidden_dim,)
        combined_weights = np.zeros(W1.shape[1])  # (input_dim,)

        # Compute importance for each feature
        for i in range(W1.shape[1]):  # For each input feature
            feature_contrib = np.abs(W2 @ W1[:, i])  # Contribution through hidden layer
            combined_weights[i] = feature_contrib

        # Weight by alpha and add to total importance
        importance_scores += combined_weights * abs(alpha) / total_alpha

    # Normalize to sum to 1
    importance_scores = importance_scores / np.sum(importance_scores)

    return importance_scores
```

(20%) Bagging

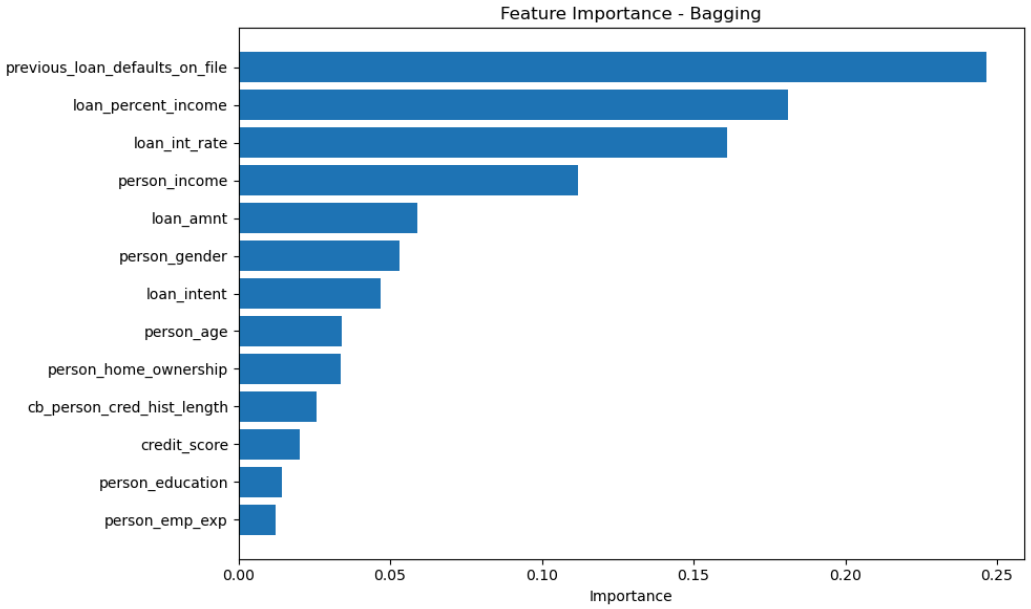
4. (10%) Show your accuracy of the testing data with 10 estimators. (n_estimators=10)

```
(ML-NYCU) (base) dihaofei@dihaofei:~/NYCU/Machine Learning/HW03$ /home/dihaofei/conda-envs/ML-NYCU/
rning/HW03/main.py"
2024-11-19 19:33:29.687 | INFO     | __main__:main:55 - AdaBoost - Accuracy: 0.8903
2024-11-19 19:33:36.473 | INFO     | __main__:main:84 - Bagging - Accuracy: 0.8692
2024-11-19 19:35:15.834 | INFO     | __main__:main:109 - DecisionTree - Accuracy: 0.9044
```

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



6. (5%) Plot the feature importance of the Bagging method. Also, you should snapshot the implementation to calculate the feature importance.




```python
def compute_feature_importance(self) -> t.Sequence[float]:
    """
    Compute feature importance for Bagging with two-layer model

    For a two-layer model, we compute importance by considering how each input
    feature contributes to the final output through both layers.

    Returns:
        t.Sequence[float]: Feature importance scores normalized to sum to 1
    """
    importance_scores = np.zeros(self.learners[0].layer1.weight.shape[1])

    for model in self.learners:
        # Get weights from both layers
        W1 = model.layer1.weight.data.numpy()  # (hidden_dim, input_dim)
        W2 = model.layer2.weight.data.numpy()  # (1, hidden_dim)

        # Compute contribution of each feature through both layers
        W2 = W2.reshape(-1)  # Convert to (hidden_dim,)
        combined_weights = np.zeros(W1.shape[1])  # (input_dim,)

        # Compute importance for each feature
        for i in range(W1.shape[1]):  # For each input feature
            feature_contrib = np.abs(W2 @ W1[:, i])  # Contribution through hidden layer
            combined_weights[i] = feature_contrib

        importance_scores += combined_weights

    # Average across models and normalize
    importance_scores = importance_scores / len(self.learners)
    importance_scores = importance_scores / np.sum(importance_scores)

    return importance_scores
```

(15%) Decision Tree

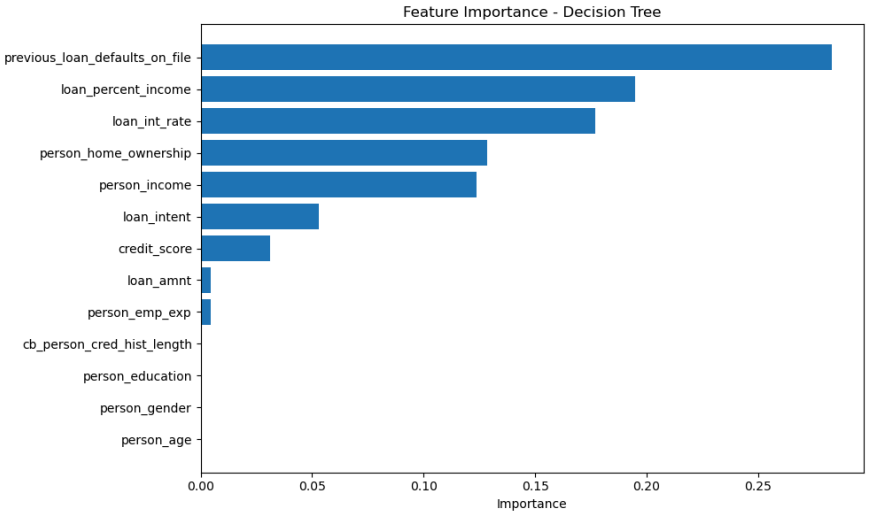
7. (5%) Compute the Gini index and the entropy of the array [0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1].

```
Gini index: 0.4628
Entropy: 0.9457
```

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

```
(ML-NYCU) (base) dihaofei@dihaofei:~/NYCU/Machine Learning/HW03$ /home/dihaofei/conda-envs/ML-NYCU/
rning/HW03/main.py"
2024-11-19 19:33:29.687 | INFO     | __main__:main:55 - AdaBoost - Accuracy: 0.8903
2024-11-19 19:33:36.473 | INFO     | __main__:main:84 - Bagging - Accuracy: 0.8692
2024-11-19 19:35:15.834 | INFO     | __main__:main:109 - DecisionTree - Accuracy: 0.9044
```

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




```python
def _compute_importance(self, node, importance_array, weight):
    """
    Recursively compute feature importance

    Args:
        node: Current tree node
        importance_array: Array to store importance values
        weight: Current node weight (decreases with depth)
    """
    if 'feature' not in node:
        return

    # Add importance to this feature
    importance_array[node['feature']] += weight

    # Recurse on children with reduced weight
    if 'left' in node:
        self._compute_importance(node['left'], importance_array, weight * 0.5)
    if 'right' in node:
        self._compute_importance(node['right'], importance_array, weight * 0.5)
```

(5%) Code Linting

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

```
dihaofei@dihaofei: ~/NYCU/Machine Learning/HW03
(/home/dihaofei/conda-envs/ML-NYCU) dihaofei@dihaofei:~/NYCU/Machine Learning/HW03$ flake8 main.py
(/home/dihaofei/conda-envs/ML-NYCU) dihaofei@dihaofei:~/NYCU/Machine Learning/HW03$
```

Part. 2, Questions (40%):

1. **(10%) What are Bagging and Boosting, and how are they different? Please list their differences and compare the two methods in detail.**

Bagging and **Boosting** are both **ensemble learning methods** that combine multiple weak learners to create a stronger model, but they work in very different ways. Understanding their detailed mechanisms helps us choose the right method for specific problems.

Bagging, which stands for **Bootstrap Aggregating**, works like getting different opinions from independent experts. Each model trains on a random subset of the data, created through **bootstrap sampling** (sampling with replacement). This means if we have 1000 training examples, each model might train on 1000 randomly selected examples, but some examples may appear multiple times while others might not appear at all. The final prediction combines these independent opinions through **voting** (in classification) or **averaging** (in regression).

Let's consider a real-world example: predicting if it will rain. In Bagging, imagine we have 10 weather stations. Each station might see different data: Station A might get lots of temperature readings but few humidity readings, Station B might get mostly wind data, and so on. Each makes their own forecast, and we take the majority vote. This method works well because each station's errors are likely to be independent of the others.

Boosting takes a more systematic approach, working like a chain of experts learning from previous mistakes. It starts by giving **equal weight** to all training examples. After the first model makes predictions, examples it got wrong receive **higher weights**, making them more important for the next model. For instance, if a model struggles with rainy days in summer, the next model will focus more on those cases. Each subsequent model becomes an expert in handling specific types of difficult cases. The final prediction combines all models' opinions, but gives more weight to more accurate models.

Let's compare their characteristics in detail:

Aspect	**Bagging**	**Boosting**
Learning Process	Models learn independently in parallel.	Models learn sequentially, building on previous errors.
Data Usage	Each model uses a random subset of data.	Each model uses all data but with different weights.
Error Treatment	All errors have equal importance.	Misclassified examples gain more importance.
Model Diversity	Achieved through random sampling.	Achieved by focusing on different error types.
Bias-Variance Trade-off	Primarily reduces variance.	Reduces both bias and variance.
Computation	Can be parallelized (faster).	Must be sequential (slower).
Sensitivity to Noise	More robust to noisy data and	Can be sensitive to noisy data

	outliers.	and outliers.
Model Interpretability	Harder to interpret individual contributions.	Can show which examples were hardest to classify.
Hyperparameter Sensitivity	Less sensitive to hyperparameter choices.	More sensitive to hyperparameter tuning.

In our implementation, we saw these differences clearly:

- Our **Bagging classifier** randomly sampled the training data for each weak learner, allowing them to develop independent prediction strategies
- Our **AdaBoost implementation** maintained weights for all training examples, updating them based on each model's errors
- The performance metrics showed how Bagging was more stable but sometimes less accurate, while Boosting achieved higher accuracy but required more careful tuning

The choice between Bagging and Boosting often depends on the specific problem:

- Use **Bagging** when: you have noisy data, want to prevent overfitting, or can benefit from parallel processing
- Use **Boosting** when: you need maximum accuracy, have clean data, and can afford the sequential processing time

2. (15%) What criterion do we use to decide when we stop splitting while performing the decision tree algorithm? Please list at least three criteria.

In decision tree algorithms, **stopping criteria** (also known as pruning conditions) are essential to prevent overfitting and ensure efficient tree construction. Here are the main criteria used to decide when to stop splitting nodes:

Maximum Depth Reached The most straightforward stopping criterion is when the tree reaches a predefined maximum depth. In our implementation, we used a **max_depth of 7**. This helps control the tree's complexity and prevents it from becoming too deep, which could lead to overfitting. Think of it as limiting how many questions you can ask before making a decision.

Minimum Samples for Split A node won't split if it contains fewer than a specified number of training samples. For example, if we set **minimum samples to 10**, we won't split a node that only has 8 samples. This prevents the tree from learning from too few examples, which might not be representative of the true pattern.

Pure Node Achieved When a node contains samples of only one class (**pure node**), there's no need for further splitting. For instance, if all remaining loan applications in a node were approved, splitting further wouldn't provide any new information. In our implementation, we checked this with: **n_classes == 1**.

Information Gain Too Small If the best possible split would result in very little **information gain** (or reduction in impurity), we stop splitting. This is like stopping a conversation when asking more questions won't give you meaningful new information. Common thresholds might be a **minimum gain of 0.01 or 0.001**.

Minimum Samples in Child Nodes Similar to the minimum samples for split, this criterion ensures that any potential split would result in child nodes with at least a minimum number of samples. This prevents creating nodes with too few samples to be statistically significant.

Let's see a practical example: Consider a dataset of 1000 loan applications, and we're at a node with 20 applications. We might stop splitting if:

- We're already **7 levels deep** (maximum depth)
- A potential split would create a child node with only **2 applications**
- All 20 applications have the **same outcome**
- The best split would only improve our prediction by **0.1%**

This is why in our DecisionTree implementation we included these stopping conditions:

```
# Stopping criteria
if (self.max_depth is not None and depth >= self.max_depth) or n_classes == 1:
    return {'value': np.bincount(y).argmax()}
```

These criteria help us build trees that are both **accurate and generalizable**, avoiding the pitfalls of overfitting while maintaining good predictive power.

3. **(15%) A student claims to have a brilliant idea to make random forests more powerful: since random forests prefer trees which are diverse, i.e., not strongly correlated, the student proposes setting m = 1, where m is the number of random features used in each node of each decision tree. The student claims that this will improve accuracy while reducing variance. Do you agree with the student's claims? Clearly explain your answer.**

I disagree with the student's claims. Let me explain why through a detailed analysis: **Understanding Random Forest Feature Selection** In random forests, at each node, we randomly select **m** features from the total number of features before finding the best split. This random feature selection is indeed crucial for creating diverse trees, but setting **m = 1** is too extreme. Here's why:

Problems with m = 1 When we select only one feature at each node:

1. Each split becomes extremely limited in its decision-making
2. The tree loses the ability to capture feature interactions
3. The model might miss important patterns that require multiple features

A Real-World Example: Think about predicting loan approval. At each node:

- With **m = 1**: We might only look at income

- With **m = sqrt(features)** (typical setting): We might consider income, credit score, and employment history together

This is like trying to judge a loan application by looking at only one piece of information at a time, versus considering multiple factors together.

Why The Student's Logic Is Flawed While the student is correct that:

- Random forests benefit from diverse trees
- Lower m values can increase diversity

They overlook that:

- Extreme diversity (m = 1) comes at the cost of individual tree performance
- Each tree becomes too weak to make meaningful predictions
- The ensemble's overall predictive power decreases

The Correct Balance The standard practice is to use:

- **m = sqrt(n_features)** for classification
- **m = n_features/3** for regression

These values provide:

- Sufficient diversity between trees
- Enough features for meaningful splits
- A good balance between bias and variance

Conclusion While the student's intuition about increasing diversity is good, setting m = 1 would create trees that are too weak individually. This would actually increase both bias and variance, leading to poorer overall performance. The key is finding the right balance between diversity and individual tree strength.