

DATA2060 Final Report

Team: little cutesy cow and horse

<https://github.com/team-alpha/data2060-project>

Wendi Liao, Yicheng Lu, Xuetong Tang, Ke Zhang

1 Overview of Adaboost

AdaBoost (Adaptive Boosting) is an ensemble learning technique designed to improve the performance of weak classifiers by combining them into a single, strong classifier. It typically addresses binary classification problems, but can be adapted for multi-class tasks. The core principle involves training a series of simple weak learners—often decision stumps—one after another. After each iteration, AdaBoost places more weight on the samples that were misclassified, pushing subsequent learners to pay closer attention to these challenging cases. Through this iterative process, the ensemble steadily refines its decision boundary.

One of the key advantages of AdaBoost is that it can significantly enhance classification accuracy by leveraging multiple weak learners. It is also relatively straightforward to implement, requiring primarily a simple weak base learner and fewer parameters to tune compared to more complex algorithms. Furthermore, because AdaBoost continually adapts the weighting scheme based on the data's difficulty, it often performs well out-of-the-box.

However, there are notable disadvantages. AdaBoost can become overly sensitive to noise in the dataset, as it repeatedly emphasizes misclassified instances that may be outliers rather than genuine hard examples. This can lead to overfitting. Additionally, if the chosen weak learners consistently perform worse than random guessing, the performance of the entire ensemble may degrade.

1.1 Representation

The representation of AdaBoost is:

$$\mathcal{E}(\mathcal{H}, T) = \left\{ x \mapsto h_s(x) = \text{sign} \left(\sum_{t=1}^T w_t h_t(x) \right) : w \in R^T, \forall t, h_t \in \mathcal{H} \right\}$$

$\mathcal{E}(\mathcal{H}, T)$ represents the ensemble of classifiers in AdaBoost. Here, \mathcal{H} represents the set of weak learners, which is the decision trees, over \mathcal{T} rounds of boosting. Essentially, in each round of boosting, the input x is mapped to a prediction made by taking the sign of a weighted sum. The sign function will return +1 if the argument is positive, and thus giving a positive prediction. If the argument is negative, the final prediction will be negative. Inside the sign function, each weak classifier h_t in the hypothesis set \mathcal{H} provides a prediction $h_t(x)$ and is given a weight w_t . Since there is \mathcal{T} rounds of iterations, we have \mathcal{T} numbers of weights.

1.2 Loss

In the context of AdaBoost, there are two important loss metrics. The overall 0-1 loss and the redefined loss on training samples.

1.2.1 Overall 0-1 Loss

The overall 0-1 loss is the metric used to measure the difference between the model's final prediction and the target variable after training is completed.

$$L_S(h_S) = \frac{1}{m} \sum_{i=1}^m 1_{h_S(x_i) \neq y_i}$$

For the total number of m examples, it assigns 1 to misclassified points and 0 otherwise. By summing up the values and dividing by the total number of examples, the 0-1 loss tells us the proportion of incorrect predictions, providing a clear and direct metric for classification accuracy.

1.2.2 Redefined Loss on Training Samples

During training, AdaBoost adjusts its focus based on the difficulty to classify each example.

$$\epsilon_t \stackrel{\text{def}}{=} L_{D^{(t)}}(h_t) \stackrel{\text{def}}{=} \sum_{i=1}^m D_i^{(t)} 1_{[h_t(x_i) \neq y_i]} \quad \text{where } D^{(t)} \in R^m$$

The loss, represented by ϵ_t , calculates the weighted error rate of the weak learner h_t at the t_{th} round of the boosting process. For each training sample, the weak learner's prediction is compared against the true label. If misclassified, the indicator function will output 1. The total error, ϵ_t , is then calculated as the sum of the products of these indicators and the corresponding weights $D_i^{(t)}$ for all training samples. This means each example's contribution to the error rate is weighted by its current weight, with examples that are harder for the model to classify correctly given a higher importance.

1.3 Optimizer

AdaBoost uses a greedy optimization algorithm to determine the optimal weights w_t for each weak learner. The algorithm[4] iterates as follows:

Input:

- Training set $S = (x_1, y_1), \dots, (x_m, y_m)$
- Weak learner WL
- Number of rounds T

Initialize $D^{(1)} = (\frac{1}{m}, \dots, \frac{1}{m})$.

For $t = 1, \dots, T$:

1. Invoke weak learner $h_t = WL(D^{(t)}, S)$

2. Compute $\epsilon_t = \sum_{i=1}^m D_i^{(t)} 1_{[y_i \neq h_t(x_i)]}$

3. Let $w_t = \frac{1}{2} \log \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$

4. Update

$$D_i^{(t+1)} = \frac{D_i^{(t)} \exp(-w_t y_i h_t(x_i))}{\sum_{j=1}^m D_j^{(t)} \exp(-w_t y_j h_t(x_j))} \quad \text{for all } i = 1, \dots, m.$$

Output the hypothesis

$$h_S(x) = \text{sign} \left(\sum_{t=1}^T w_t h_t(x) \right).$$

Here, D represents the weight on each example, not the weak learners. A lower error ϵ_t results in a higher weight w_t , making h_t more influential in the final ensemble. Finally, Each weak learner's prediction $h_t(x)$ is scaled by its weight w_t , and the sign function determines the final predicted class.

2 Model

This section contains the code implementation of our ML algorithm Adaboost and the weak learner model Decision Tree.

2.1 Model Decision Tree (to use as weak learners in AdaBoostClassifier)

```
1 import numpy as np
2 import pandas as pd
3
4 def node_score_error(prob):
5     """
6     Calculate the node score using the train error of the subdataset and return it
7     .
8     For a dataset with two classes,  $C(p) = \min\{p, 1-p\}$ 
9     """
10    return min(prob, 1.0 - prob)
11
12 def node_score_entropy(prob):
13     """
14     Calculate the node score using the entropy of the subdataset and return it.
15     For a dataset with 2 classes,  $C(p) = -p * \log(p) - (1-p) * \log(1-p)$ 
16     For the purposes of this calculation, assume  $0 * \log 0 = 0$ .
17     HINT: remember to consider the range of values that p can take!
18     """
19    # HINT: If  $p < 0$  or  $p > 1$  then entropy = 0
20
21    if prob <= 0.0 or prob >= 1.0:
22        return 0.0
23
24    return -prob * np.log(prob) - (1.0 - prob) * np.log(1.0 - prob)
25
26 def node_score_gini(prob):
27     """
28     Calculate the node score using the gini index of the subdataset and return it.
29     For dataset with 2 classes,  $C(p) = 2 * p * (1-p)$ 
30     """
31
32    return 2.0 * prob * (1.0 - prob)
33
34 class Node:
35     """
36     Helper to construct the tree structure.
37     """
38     def __init__(self, left=None, right=None, depth=0, index_split_on=0, isleaf=False,
39                  label=1):
40         self.left = left
41         self.right = right
42         self.depth = depth
43         self.index_split_on = index_split_on
44         self.isleaf = isleaf
45         self.label = label
46         self.info = {} # used for visualization
47         self.threshold = None
48
49     def _set_info(self, gain, num_samples):
50         """
51         Helper function to add to info attribute.
52         """
53         self.info['gain'] = gain
54         self.info['num_samples'] = num_samples
55
56 class DecisionTree:
57
58     def __init__(self, data, gain_function=node_score_gini, max_depth=40, weight=None,
59                  converted=None):
60         # Initialize the decision tree with data and parameters.
61         if converted is not None:
62             for row in data:
63                 if row[0] == -1:
64                     row[0] = 0 # Convert -1 to 0
65
66         self.majority_class = 1 if sum(row[0] for row in data) > len(data) / 2 else 0
67         self.max_depth = max_depth
```

```

67     self.root = Node(label=self.majority_class)
68     self.gain_function = gain_function
69     if weight is None:
70         self.sample_weight = np.ones(len(data)) / len(data)
71     else:
72         self.sample_weight = weight / np.sum(weight)
73
74     indices = list(range(1, len(data[0])))
75     self._split_rekurs(self.root, data, indices, self.sample_weight)
76
77     def predict(self, features, converted=None):
78         '''
79         Predict the label for given features.
80         '''
81         if features.ndim == 1: # 1d array
82             prediction = self._predict_rekurs(self.root, features)
83             return -1 if converted and prediction == 0 else prediction
84         else: # 2d array
85             predictions = []
86             for feature in features:
87                 prediction = self._predict_rekurs(self.root, feature)
88                 if converted and prediction == 0:
89                     prediction = -1
90                 predictions.append(prediction)
91             return np.array(predictions)
92
93     def accuracy(self, data):
94         '''
95         Calculate accuracy on the given data.
96         '''
97         return 1 - self.loss(data)
98
99     def loss(self, data):
100         '''
101         Calculate loss on the given data.
102         '''
103         test_Y = np.array([row[0] for row in data]) # Get the true labels
104         predictions = self.predict(np.array(data)) # Get the predicted results
105         return np.mean(predictions != test_Y)
106
107     def _predict_rekurs(self, node, row):
108         '''
109         Predict label by traversing the tree.
110         '''
111         if node.isleaf or node.index_split_on == 0:
112             return node.label
113         split_index = node.index_split_on
114         if not row[split_index]:
115             return self._predict_rekurs(node.left, row)
116         else:
117             return self._predict_rekurs(node.right, row)
118
119
120     def _is_terminal(self, node, data, indices):
121         '''
122         Check if the node should stop splitting.
123         '''
124         y = [row[0] for row in data]
125
126         sumy = sum(row[0] for row in data)
127
128         if len(data) - sumy == sumy:
129             majority_label = self.majority_class
130         else:
131             majority_label = 1 if sumy > len(data) / 2 else 0
132
133         if len(set(y)) == 1:
134             return True, y[0]
135         if len(data) == 0:
136             return True, self.majority_class
137         if len(indices) == 0:

```

```

138         return True, majority_label
139
140     if node.depth >= self.max_depth:
141         return True, majority_label
142
143     return False, majority_label
144
145     def _split_recurs(self, node, data, indices, weights):
146         """
147         Recursively split the node based on data.
148         """
149         node.isleaf, node.label = self._is_terminal(node, data, indices)
150
151         if not node.isleaf:
152             max_gain = -1
153             best_threshold = None
154
155             for split_index in indices:
156                 feature_values = sorted(set(row[split_index] for row in data))
157
158                 for i in range(len(feature_values) - 1):
159                     threshold = (feature_values[i] + feature_values[i + 1]) / 2
160                     gain = self._calc_gain(data, split_index, self.gain_function,
threshold, weights)
161
162                     if gain > max_gain:
163                         max_gain = gain
164                         node.index_split_on = split_index
165                         best_threshold = threshold
166
167                     if len(feature_values) == 1:
168                         gain = self._calc_gain(data, split_index, self.gain_function,
feature_values[0], weights)
169                         if gain > max_gain:
170                             max_gain = gain
171                             node.index_split_on = split_index
172                             best_threshold = feature_values[0]
173
174                 node._set_info(max_gain, len(data))
175                 node.threshold = best_threshold
176
177                 node.left = Node(depth=node.depth + 1)
178                 node.right = Node(depth=node.depth + 1)
179                 indices.remove(node.index_split_on)
180
181                 leftData = [row for row in data if row[node.index_split_on] <= node.
threshold]
182                 rightData = [row for row in data if row[node.index_split_on] > node.
threshold]
183
184                 left_weights = weights[[row[node.index_split_on] <= node.threshold for row
in data]]
185                 right_weights = weights[[row[node.index_split_on] > node.threshold for row
in data]]
186
187                 self._split_recurs(node.left, leftData, indices, left_weights)
188                 self._split_recurs(node.right, rightData, indices, right_weights)
189             else:
190                 node._set_info(0, len(data))
191
192     def _calc_gain(self, data, split_index, gain_function, threshold=None, weights=
None):
193         """
194         Calculate gain for the proposed split.
195         """
196         if threshold is None:
197             threshold = 0.5
198         if weights is None:
199             weights = np.ones(len(data)) # Default weights
200         y = [row[0] for row in data]
201         xi = [1 if row[split_index] > threshold else 0 for row in data]

```

```

202         if len(y) != 0 and len(xi) != 0:
203             total_weight = np.sum(weights)
204             probY = np.sum(weights * y) / total_weight
205             probX = np.sum(weights * xi) / total_weight
206
207
208         weights = weights.to_numpy() if isinstance(weights, pd.Series) else
weights
209         y = np.array(y) if not isinstance(y, np.ndarray) else y
210         xi = np.array(xi) if not isinstance(xi, np.ndarray) else xi
211
212         y1x1 = sum(weights[index] for index in range(len(y)) if y[index] == 1 and
xi[index] == 1)
213         y0x0 = sum(weights[index] for index in range(len(y)) if y[index] == 0 and
xi[index] == 0)
214
215         prob1 = y1x1 / total_weight
216         prob2 = y0x0 / total_weight
217
218         probxi_true = (probX * gain_function(prob1 / probX)) if probX > 0 else 0
219         probxi_false = ((1.0 - probX) * gain_function(prob2 / (1.0 - probX))) if
probX < 1.0 else 0
220
221         gain = gain_function(probY) - probxi_true - probxi_false
222     else:
223         gain = 0
224
225     return gain
226
227     def print_tree(self):
228         '''
229         Helper function for tree_visualization.
230         Only effective with very shallow trees.
231         You do not need to modify this.
232         '''
233         print('---START PRINT TREE---')
234         def print_subtree(node, indent=''):
235             if node is None:
236                 return str("None")
237             if node.isleaf:
238                 return str(node.label)
239             else:
240                 decision = 'split attribute = {:d}; gain = {:f}; number of samples =
{:d}'.format(node.index_split_on, node.info['gain'], node.info['num_samples'])
241                 left = indent + '0 -> ' + print_subtree(node.left, indent + '\t\t')
242                 right = indent + '1 -> ' + print_subtree(node.right, indent + '\t\t')
243                 return (decision + '\n' + left + '\n' + right)
244
245         print(print_subtree(self.root))
246         print('----END PRINT TREE---')

```

Listing 1: decision tree

2.2 Model AdaBoostClassifier

```

1 import numpy as np
2
3 class AdaBoostClassifier:
4     """
5     AdaBoost (Adaptive Boosting) Classifier
6     An ensemble learning algorithm that combines multiple weak classifiers to build a
strong classifier.
7     """
8
9     def __init__(self, n_estimators=10, max_depth=1):
10         """
11         Initialize the AdaBoost classifier.
12
13         Parameters:
14         - n_estimators: Number of weak classifiers to use.

```

```

15     """
16     self.n_estimators = n_estimators
17     self.max_depth = max_depth # Store max_depth for DecisionTree
18     self.w = [] # Store the weights of the classifiers
19     self.models = [] # Store the weak classifiers
20
21     def train(self, X, y):
22         """
23         Fit the AdaBoost model to the training data.
24         For T WLS,
25         1. train WL (DecisionTree with max_depth=1)
26         2. compute error of this WL
27         3. compute the weight of this WL w_t and store it in self.w
28         4. compute and update the distribution D of the samples for WLS next
29
30         Parameters:
31         - X: Training data, shape (n_samples, n_features)
32         - y: Target labels, shape (n_samples,)
33         """
34         n_samples, n_features = X.shape
35         # Initialize weights uniformly
36         D = np.ones(n_samples) / n_samples
37
38         for t in range(self.n_estimators):
39             '''
40             # sklearn
41             # Create a weak classifier (decision stump)
42             std_model = DecisionTreeClassifier(max_depth=2)
43             # Fit the model to the training data
44             std_model.fit(X, y, sample_weight=D) # Add this line to train the model
45             y_pred_sklearn = std_model.predict(X)
46             '''
47             # avoid extremely small sample weight
48             D = np.clip(D, a_min=np.finfo(D.dtype).eps, a_max=None)
49
50             weak_model = DecisionTree(data=np.column_stack((y, X)), max_depth=self.
max_depth, weight=D, converted=True)
51             # weak_model = DecisionTreeClassifier(max_depth=self.max_depth)
52             # weak_model.fit(X, y, sample_weight=D)
53
54             y_pred = weak_model.predict(features=np.column_stack((y, X)), converted =
True)
55             # y_pred = weak_model.predict(X)
56
57             # Calculate the weighted error
58             error = np.mean(np.average(y_pred != y, weights=D, axis=0))
59
60             # Calculate the weight for the weak classifier
61             w_t = 0.5 * np.log((1.0 - error) / (error + 1e-10)) # Avoid division by
zero
62
63             # Update weights for the next iteration
64             D *= np.exp(-w_t * y * y_pred) # Update weights based on prediction
65             D /= np.sum(D * np.exp(-w_t * y * y_pred)) # Normalize weights
66
67             self.models.append(weak_model) # Store the model
68             self.w.append(w_t) # Store the w_t
69
70     def predict(self, X, converted = True):
71         """
72         Predict the class labels for the input data.
73
74         Parameters:
75         - X: Input data, shape (n_samples, n_features)
76
77         Returns:
78         - Predicted class labels, shape (n_samples,)
79         """
80         pred = np.zeros(X.shape[0]) # Initialize predictions
81         X_with_zero = np.insert(X, 0, 0, axis=1) # Insert 0 at the beginning of each
row

```

```

82         for w_i, model in zip(self.w, self.models):
83             pred += w_i * model.predict(X_with_zero, converted) # Weighted sum of
predictions
84             # pred += w_i * model.predict(X)
85         return np.sign(pred) # Return the sign of the predictions
86
87     def accuracy(self, X, y):
88         """
89         Calculate the accuracy of the model.
90
91         Parameters:
92         - X: Input data, shape (n_samples, n_features)
93         - y: True labels, shape (n_samples,)
94
95         Returns:
96         - Accuracy as a float.
97         """
98         predictions = self.predict(X) # Get predictions
99         accuracy = np.mean(predictions == y) # Calculate accuracy
100        return accuracy

```

Listing 2: adaboost classifier

3 Check model

This section is a collection of code and markdown cells that contain the unit tests and a demonstration that our implementation can reproduce the same results as the scikit-learn model.

3.1 tests for Adaboost

```

1  if __name__ == "__main__":
2      # Create a simple dataset
3      X = np.array([
4          [0, 0, 1, 0],
5          [1, 1, 0, 1],
6          [1, 0, 1, 0],
7          [0, 1, 0, 1],
8          [0, 0, 0, 0],
9          [1, 1, 1, 1],
10         [0, 1, 1, 0],
11         [1, 0, 0, 1],
12         [1, 1, 0, 0],
13         [0, 0, 1, 1]
14     ]) # 10 samples with 4 features
15
16     y = np.array([-1, 1, 1, -1, -1, 1, 1, -1, 1, -1]) # Binary labels (-1 and 1)
17     # Initialize the AdaBoost classifier
18     model = AdaBoostClassifier(n_estimators=10, max_depth=1)
19
20     # Train the model
21     model.train(X, y)
22
23     # Calculate accuracy
24     accuracy = model.accuracy(X, y)
25
26     # Print results
27     print("Accuracy:", accuracy)

```

Listing 3: test1

Output:

```

1 Accuracy: 1.0

```

Listing 4: test1 output

```

1 import pytest
2 import numpy as np

```



```

3
4 # Sets random seed for testing purposes
5 np.random.seed(0)
6
7 # Creates Test Models
8 test_model1 = AdaBoostClassifier(n_estimators=10)
9 test_model2 = AdaBoostClassifier(n_estimators=50)
10 test_model3 = AdaBoostClassifier(n_estimators=20)
11
12 # Dataset 1
13 x1 = np.array([
14     [0, 0, 1, 0],
15     [1, 1, 0, 1],
16     [1, 0, 1, 0],
17     [0, 1, 0, 1],
18     [0, 0, 0, 0],
19     [1, 1, 1, 1],
20     [0, 1, 1, 0],
21     [1, 0, 0, 1],
22     [1, 1, 0, 0],
23     [0, 0, 1, 1]
24 ]) # 10 samples with 4 features
25
26 y1 = np.array([-1, 1, 1, -1, -1, 1, 1, -1, 1, -1]) # Binary labels (-1 and 1)
27
28 # Dataset 2
29 x2 = np.array([
30     [0, 1, 0, 1, 1, 0],
31     [1, 0, 1, 0, 0, 1],
32     [1, 1, 0, 1, 0, 0],
33     [0, 0, 1, 1, 1, 1],
34     [1, 0, 0, 0, 1, 0],
35     [0, 1, 1, 0, 0, 1],
36     [1, 1, 1, 0, 1, 1],
37     [0, 0, 0, 1, 0, 0],
38     [1, 0, 1, 1, 1, 0],
39     [0, 1, 0, 0, 1, 1]
40 ]) # 10 samples with 6 features
41
42 y2 = np.array([-1, 1, 1, -1, 1, -1, 1, -1, 1, -1]) # Binary labels (-1 and 1)
43
44 # Dataset 3
45 x3 = np.array([
46     [1, 1, 0, 0, 1, 1],
47     [0, 0, 1, 1, 0, 0],
48     [1, 0, 1, 0, 1, 0],
49     [0, 1, 0, 1, 1, 1],
50     [1, 1, 1, 0, 0, 1],
51     [0, 0, 0, 1, 0, 1],
52     [1, 0, 0, 1, 1, 0],
53     [0, 1, 1, 0, 1, 1],
54     [1, 1, 1, 1, 0, 0],
55     [0, 0, 1, 0, 1, 0]
56 ]) # 10 samples with 6 features
57
58 y3 = np.array([1, -1, 1, -1, 1, -1, 1, -1, 1, -1]) # Binary labels (-1 and 1)
59
60 # Test Model Train
61 def check_train_dtype(model, X, y):
62     assert isinstance(model.models, list)
63     assert len(model.models) > 0, "Model should have trained at least one weak learner"
64     assert len(model.w) == len(model.models), "Weights should match the number of models."
65
66 # Train the models
67 test_model1.train(x1, y1)
68 check_train_dtype(test_model1, x1, y1)
69
70 test_model2.train(x2, y2)
71 check_train_dtype(test_model2, x2, y2)

```

```

72 test_model3.train(x3, y3)
73 check_train_dtype(test_model3, x3, y3)
74
75
76 # Test Model Predictions
77 def check_test_dtype(pred, X_test):
78     assert isinstance(pred, np.ndarray)
79     assert pred.ndim == 1 and pred.shape == (X_test.shape[0],)
80
81 # Make predictions
82 pred1 = test_model1.predict(x1)
83 check_test_dtype(pred1, x1)
84 assert (pred1 == y1).all(), "Predictions should match the expected labels for model 1.
85
86 pred2 = test_model2.predict(x2)
87 check_test_dtype(pred2, x2)
88 assert (pred2 == y2).all(), "Predictions should match the expected labels for model 2.
89
90 pred3 = test_model3.predict(x3)
91 check_test_dtype(pred3, x3)
92 assert (pred3 == y3).all(), "Predictions should match the expected labels for model 3.
93
94 # Test Model Accuracy
95 def check_accuracy(model, X, y, expected_accuracy):
96     accuracy = model.accuracy(X, y)
97     assert accuracy == expected_accuracy, f"Expected accuracy: {expected_accuracy},
98     but got: {accuracy}"
99
100 # Check accuracy
101 check_accuracy(test_model1, x1, y1, 1.0) # Expecting 100% accuracy for this simple
102 case
103 check_accuracy(test_model2, x2, y2, 1.0) # Expecting 100% accuracy for this dataset
104 check_accuracy(test_model3, x3, y3, 1.0) # Expecting 100% accuracy for this dataset
105
106 # Additional Tests for Edge Cases
107 def test_empty_train():
108     with pytest.raises(ValueError):
109         test_model1.train(np.array([]), np.array([]))
110
111 def test_empty_predict():
112     with pytest.raises(ValueError):
113         test_model1.predict(np.array([]))
114
115 def test_accuracy_empty():
116     with pytest.raises(ValueError):
117         test_model1.accuracy(np.array([]), np.array([]))
118
119 # Run additional edge case tests
120 test_empty_train()
121 test_empty_predict()
122 test_accuracy_empty()
123
124 # Print a message indicating the tests have completed
125 print("All tests completed successfully.")

```

Listing 5: test2 (unit tests)

Output:

```

1 All tests completed successfully.

```

Listing 6: test2 output

3.2 tests for Decision Tree (weak learner)

```

1 if __name__ == "__main__":
2     # Create a simple dataset

```

```

3     X = np.array([[0, 0],
4                   [1, 1],
5                   [1, 0],
6                   [0, 1],
7                   [0, 0],
8                   [1, 1]])
9
10    # Corresponding labels
11    y = np.array([0, 0, 1, 1, 0, 1]) # Labels should be -1 and 1 for AdaBoost
12
13    # Initialize the AdaBoost classifier
14    weak_model = DecisionTree(data=np.column_stack((y, X)), max_depth=4)
15    y_pred = np.zeros_like(y) # Initialize y_pred
16    for i, (y_i, x_i) in enumerate(zip(y, X)):
17        combined_input = np.append(y_i, x_i)
18        y_pred_i = weak_model.predict(combined_input) # Predictions from the model
19        y_pred[i] = y_pred_i # Update y_pred with the prediction
20    print("y_pred: ", y_pred)

```

Listing 7: test1

Output:

```

1 y_pred:  [0 1 1 1 0 1]

```

Listing 8: test1 output

```

1 if __name__ == "__main__":
2     # Create a simple dataset
3     X = np.array([[0, 0],
4                   [1, 1],
5                   [1, 0],
6                   [0, 1],
7                   [0, 0],
8                   [1, 1]])
9
10    # Corresponding labels
11    y = np.array([0, 0, 1, 1, 0, 1]) # Labels should be -1 and 1 for AdaBoost
12
13    # Initialize the AdaBoost classifier
14    weak_model = DecisionTree(data=np.column_stack((y, X)), max_depth=2)
15    y_pred = np.zeros_like(y) # Initialize y_pred
16    for i, (y_i, x_i) in enumerate(zip(y, X)):
17        combined_input = np.append(y_i, x_i)
18        y_pred_i = weak_model.predict(combined_input) # Predictions from the model
19        y_pred[i] = y_pred_i # Update y_pred with the prediction
20    print("y_pred: ", y_pred)
21    # Make predictions

```

Listing 9: test2

Output:

```

1 y_pred:  [0 1 1 1 0 1]

```

Listing 10: test2 output

```

1 import pytest
2 import random
3
4 np.random.seed(0)
5 random.seed(0)
6
7 # Tests for node_score_error
8 assert node_score_error(.3) == .3
9 assert node_score_error(.6) == .4
10
11 # Tests for node_score_entropy
12 assert node_score_entropy(.5) == pytest.approx(.69, .01)
13 assert node_score_entropy(0) == node_score_entropy(1) == 0
14 assert node_score_entropy(.7) == pytest.approx(.61, .01)
15
16 # Tests for node_score_gini

```

```

17 assert node_score_gini(1) == node_score_gini(0) == 0
18 assert node_score_gini(.4) == .48
19
20 # Creates Test Model and Dummy Data
21 x = np.array([[0,1,0,0],[1,0,1,1],[1,1,0,1],[0,0,1,0],[0,1,1,1],[0,0,0,0]])
22 test_model = DecisionTree(x, gain_function=node_score_entropy)
23
24 # Test for majority_class
25 assert test_model.majority_class == 0
26
27 # Tests for _is_terminal
28 node1 = Node(left=None, right=None, depth=0, index_split_on=3, isleaf=False, label=0)
29 x_filtered_node2 = np.array([row for row in x if row[3] == 1])
30 node2 = Node(left=None, right=None, depth=1, index_split_on=1, isleaf=False, label=1)
31 x_filtered_node3 = np.array([row for row in x_filtered_node2 if row[1] == 1])
32 node3 = Node(left=None, right=None, depth=2, index_split_on=2, isleaf=False, label=0)
33 x_filtered_node4 = np.array([row for row in x_filtered_node3 if row[2] == 1])
34 node4 = Node(left=None, right=None, depth=3, index_split_on=None, isleaf=True, label
    =0)
35
36 assert test_model._is_terminal(node=node1, data=x, indices=[1, 2, 3]) == (False, 0)
37 assert test_model._is_terminal(node=node2, data=x_filtered_node2, indices=[1, 2]) == (
    False, 1)
38 assert test_model._is_terminal(node=node3, data=x_filtered_node3, indices=[2]) == (
    False, 0)
39 assert test_model._is_terminal(node=node4, data=x_filtered_node4, indices=[]) == (True
    , 0)
40
41 # Tests _calc_gain
42 # Testing gain for index 3
43 print("test model", test_model._calc_gain(x, 3, node_score_error))
44 print("----start-----")
45 print()
46 assert test_model._calc_gain(x, 3, node_score_error) == pytest.approx(0.166, .01)
47
48 assert test_model._calc_gain(x, 3, node_score_entropy) == pytest.approx(0.318, .01)
49 assert test_model._calc_gain(x, 3, node_score_gini) == pytest.approx(0.222, .01)
50
51 # Testing gain for index 1
52 assert test_model._calc_gain(x_filtered_node2, 1, node_score_error) == pytest.approx
    (5.551115123125783e-17, abs=1e-18)
53 assert test_model._calc_gain(x_filtered_node2, 1, node_score_entropy) == pytest.approx
    (0.174, .01)
54 assert test_model._calc_gain(x_filtered_node2, 1, node_score_gini) == pytest.approx
    (0.111, .01)
55
56 # Testing gain for index 2
57 assert test_model._calc_gain(x_filtered_node3, 2, node_score_error) == pytest.approx
    (0.5, .01)
58 assert test_model._calc_gain(x_filtered_node3, 2, node_score_entropy) == pytest.approx
    (0.693, .01)
59 assert test_model._calc_gain(x_filtered_node3, 2, node_score_gini) == pytest.approx
    (0.5, .01)
60
61 # Check Tree is created Properly, Compare with text below
62 '''
63 Decision Trees should look similar to below (the second one is the pruned tree)
64
65 ---START PRINT TREE---
66 split attribute = 3; gain = 0.318257; number of samples = 6
67 0 -> False
68 1 -> split attribute = 1; gain = 0.174416; number of samples = 3
69     0 -> True
70     1 -> split attribute = 2; gain = 0.693147; number of samples = 2
71         0 -> True
72         1 -> False
73 ----END PRINT TREE---
74 '''
75 test_model.print_tree()
76
77 # check loss

```

```

78 print('training loss:', test_model.loss(x))
79 x_val = np.array([[1,1,1,1],[1,0,0,1]])
80 print('validation loss:', test_model.loss(x_val), '\n')

```

Listing 11: test3 (unit tests)

Output:

```

1 test model 0.16666666666666663
2 ----start-----
3
4 ---START PRINT TREE---
5 split attribute = 3; gain = 0.318257; number of samples = 6
6 0 -> 0
7 1 -> split attribute = 1; gain = 0.174416; number of samples = 3
8     0 -> 1
9     1 -> split attribute = 2; gain = 0.693147; number of samples = 2
10        0 -> 1
11        1 -> 0
12 ----END PRINT TREE---
13 training loss: 0.0
14 validation loss: 0.5

```

Listing 12: test3 output

3.3 reproduce previous work using our model

For previous work, we choose `scikit-learn`'s implementation of AdaBoost (`AdaBoostClassifier`) with a shallow decision tree (`DecisionTreeClassifier`) as the base estimator. We apply `scikit-learn`'s AdaBoost model[2] to the *mushroom dataset*[1], which comprises 23 binary features describing mushroom characteristics, such as shape, color, and odor, with a binary target indicating edibility (-1 for edible, 1 for poisonous). The model achieves high accuracy, demonstrating the power of AdaBoost in correctly classifying tumors.

3.3.1 code & results of previous work

```

1 # Load the Mushroom Dataset from UCI repository
2 import pandas as pd
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import seaborn as sns
6 from sklearn import datasets
7 from sklearn.model_selection import train_test_split
8 from sklearn.ensemble import AdaBoostClassifier as AdaBoostClassifier_sklearn
9 from sklearn.metrics import confusion_matrix, classification_report
10 from sklearn.preprocessing import LabelEncoder
11 from sklearn.tree import DecisionTreeClassifier
12
13 # URL for Mushroom dataset
14 url = "https://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/agaricus-
    lepiota.data"
15
16 # Column names for the dataset as per UCI documentation
17 columns = [
18     "class", "cap-shape", "cap-surface", "cap-color", "bruises", "odor",
19     "gill-attachment", "gill-spacing", "gill-size", "gill-color",
20     "stalk-shape", "stalk-root", "stalk-surface-above-ring",
21     "stalk-surface-below-ring", "stalk-color-above-ring",
22     "stalk-color-below-ring", "veil-type", "veil-color", "ring-number",
23     "ring-type", "spore-print-color", "population", "habitat"
24 ]
25
26 # Load dataset
27 mushroom_data = pd.read_csv(url, header=None, names=columns)
28
29 # Preprocess the data
30 # Convert categorical variables into binary using one-hot encoding
31 # Convert 'class' column (edible=e, poisonous=p) into binary labels

```

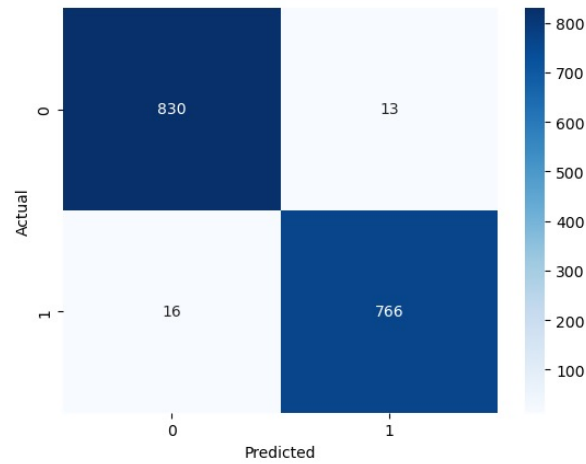


Figure 1: sklearn model result

```

32 mushroom_data["class"] = mushroom_data["class"].apply(lambda x: 1 if x == "p" else -1)
33 mushroom_data_encoded = pd.get_dummies(mushroom_data.drop(columns=["class"]))
34
35 # Combine the binary features with the target
36 mushroom_dataset = pd.concat([mushroom_data["class"], mushroom_data_encoded], axis=1)
37
38 X = mushroom_dataset.drop(columns=["class"])
39 y = mushroom_dataset["class"]
40
41 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
42 =42)
43 base_estimator = DecisionTreeClassifier(max_depth=1)
44 adaboost = AdaBoostClassifier_sklearn(estimator=base_estimator, algorithm='SAMME',
45 n_estimators=10, learning_rate=1, random_state=42)
46
47 # Train AdaBoost classifier
48 adaboost.fit(X_train, y_train)
49
50 # Making predictions
51 y_pred = adaboost.predict(X_test)
52
53 # Create confusion matrix
54 conf_matrix = confusion_matrix(y_test, y_pred)
55
56 # Visualize the confusion matrix
57 sns.heatmap(conf_matrix, annot=True, fmt='d', cmap=plt.cm.Blues)
58 plt.xlabel('Predicted')
59 plt.ylabel('Actual')
60 plt.show()
61
62 # Generate classification report
63 class_report = classification_report(y_test, y_pred)
64 print(class_report)
65
66 # Calculate accuracy
67 accuracy = adaboost.score(X_test, y_test)
68 print("Model accuracy: ", accuracy)

```

Listing 13: sklearn model for classification

Output:

	precision	recall	f1-score	support
-1	0.98	0.98	0.98	843
1	0.98	0.98	0.98	782
accuracy			0.98	1625
macro avg	0.98	0.98	0.98	1625

```

8 weighted avg      0.98      0.98      0.98      1625
9
10 Model accuracy:  0.9821538461538462

```

Listing 14: sklearn model performance

3.3.2 code & results using our own model

```

1 # URL for Mushroom dataset
2 url = "https://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/agaricus-
    lepiota.data"
3
4 # Column names for the dataset as per UCI documentation
5 columns = [
6     "class", "cap-shape", "cap-surface", "cap-color", "bruises", "odor",
7     "gill-attachment", "gill-spacing", "gill-size", "gill-color",
8     "stalk-shape", "stalk-root", "stalk-surface-above-ring",
9     "stalk-surface-below-ring", "stalk-color-above-ring",
10    "stalk-color-below-ring", "veil-type", "veil-color", "ring-number",
11    "ring-type", "spore-print-color", "population", "habitat"
12 ]
13
14 # Load dataset
15 mushroom_data = pd.read_csv(url, header=None, names=columns)
16
17 # Preprocess the data
18 # Convert categorical variables into binary using one-hot encoding
19 # Convert 'class' column (edible=e, poisonous=p) into binary labels
20 mushroom_data["class"] = mushroom_data["class"].apply(lambda x: 1 if x == "p" else -1)
21 mushroom_data_encoded = pd.get_dummies(mushroom_data.drop(columns=["class"]))
22
23 # Combine the binary features with the target
24 mushroom_dataset = pd.concat([mushroom_data["class"], mushroom_data_encoded], axis=1)
25
26 X = mushroom_dataset.drop(columns=["class"])
27 y = mushroom_dataset["class"]
28
29 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
    =42)
30 adaboost = AdaBoostClassifier(n_estimators=10, max_depth=1)
31 adaboost.train(X_train, y_train)
32
33 # Making predictions
34 y_pred = adaboost.predict(X_test)
35
36 # Create confusion matrix
37 conf_matrix = confusion_matrix(y_test, y_pred)
38
39 # Visualize the confusion matrix
40 sns.heatmap(conf_matrix, annot=True, fmt='d', cmap=plt.cm.Blues)
41 plt.xlabel('Predicted')
42 plt.ylabel('Actual')
43 plt.show()
44
45 # Generate classification report
46 class_report = classification_report(y_test, y_pred)
47 print(class_report)
48
49 # Calculate accuracy
50 accuracy = adaboost.accuracy(X_test, y_test)
51 print("Model accuracy: ", accuracy)

```

Listing 15: run our model for the same task

Output:

	precision	recall	f1-score	support
-1	0.98	0.98	0.98	843
1	0.98	0.98	0.98	782

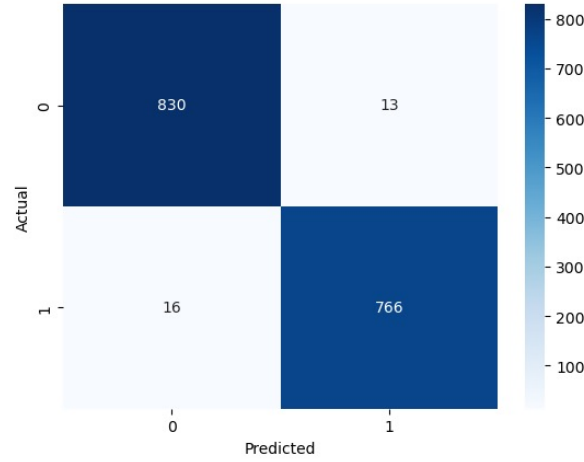


Figure 2: our model result

```

6      accuracy      0.98      0.98      0.98      1625
7      macro avg      0.98      0.98      0.98      1625
8      weighted avg    0.98      0.98      0.98      1625
9
10 Model accuracy: 0.9821538461538462

```

Listing 16: our model performance

3.3.3 summary

We can see that our own implemented model reproduces the same results from the sklearn model.

4 Github repo

<https://github.com/kzhangaz/data2060-final-project-boosting>

References

- [1] University of California, Irvine. (1987). *Mushroom Data Set*. [Online]. Available at: <https://archive.ics.uci.edu/dataset/73/mushroom> [Accessed 10 December 2024].
- [2] scikit-learn. (2024). *Weighted Boosting Module*. GitHub repository. Available at: https://github.com/scikit-learn/scikit-learn/blob/6e9039160/sklearn/ensemble/_weight_boosting.py#L339 [Accessed 10 December 2024].
- [3] scikit-learn. (2024). *AdaBoost Classifier*. [Online]. Available at: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html> [Accessed 10 December 2024].
- [4] Brubaker, M., n.d. *AdaBoost: Adaptive Boosting*. [PDF]. Available at: <https://www.cs.toronto.edu/~mbrubake/teaching/C11/Handouts/AdaBoost.pdf> [Accessed 10 December 2024].