Homework # 7

姓名	學號
操之晴	R13922A04

5.

The aggregate classifier G makes an error if the majority of 2M+1 classifiers fail

$$\sum_{t=1}^{2M+1} W_t \geq M+1$$

To bound the probability of this error

$$P\left(\sum_{t=1}^{2M+1} W_t \geq M+1\right) \leq \frac{\mathbb{E}[\sum_{t=1}^{2M+1} W_t]}{M+1}$$

Since W_t is binary with $\mathbb{E}[W_t] = e_t$

$$\mathbb{E}\left[\sum_{t=1}^{2M+1} W_t\right] = \sum_{t=1}^{2M+1} e_t$$

Substituting into Markov's inequality

$$E_{out}(G) \leq \frac{\sum_{t=1}^{2M+1} e_t}{M+1}$$

6.

$U_t = \sum_{n=1}^{N} u_n^{(t)}$

$\epsilon_t = \frac{\sum_{n=1}^{N} u_n^{(t)} [|y_n \neq g_t(x_n)|]}{\sum_{n=1}^{N} u_n^t}$

So, we get $\begin{cases} U_t \cdot \epsilon_t = \sum_{n=1}^{N} u_n^{(t)} [|y_n \neq g_t(x_n)|] \\ U_t \cdot (1-\epsilon_t) = \sum_{n=1}^{N} u_n^{(t)} [|y_n = g_t(x_n)|] \end{cases}$

$$U_{t+1} = \sum_{n=1}^{N} u_n^{(t+1)} = \sum_{n=1}^{N} u_n^{(t)} [|y_n \neq g_t(x_n)|] * \blacklozenge_t + \sum_{n=1}^{N} \frac{u_n^{(t)} [|y_n = g_t(x_n)|]}{\blacklozenge_t}$$

$$= U_t \cdot \epsilon_t * \blacklozenge_t + U_t \cdot (1-\epsilon_t) / \blacklozenge_t$$

$$= U_t \cdot \sqrt{\frac{(1-\epsilon_t)}{\epsilon_t}} + U_t \cdot (1-\epsilon_t) \sqrt{\frac{\epsilon_t}{(1-\epsilon_t)}}$$

$$= 2U_t \sqrt{\epsilon_t (1-\epsilon_t)}$$

Thus, we have $\frac{U_{t+1}}{U_t} = 2\sqrt{\epsilon_t (1-\epsilon_t)}$

7.

In gradient boosting, we minimize the loss function $L(y, \hat{y})$, where y is the true value and $\hat{y}$ is the prediction. If we use the squared error loss

$$L(y, \hat{y}) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

At the first step (t=1), the gradient of the loss function is:

$$g_i = -\frac{\partial L}{\partial \hat{y}_i} = y_i - \hat{y}_i$$

Where $\hat{y}_t = 0$ initially (since no predictions are made yet).
Linear regression is fitted to minimize the residuals g_i, which are y_i in this case.
The prediction after the first iteration is:

$$\hat{y}_i^{(1)} = \alpha_1 \cdot f_1(x_i)$$

where $f_1(x_i)$ is the output of the linear regression model fitted to g_i.
The optimal α_1 minimizes the total loss:

$$L(y, \hat{y}) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \alpha_1 f_1(x_i))^2$$

$$\frac{\partial L}{\partial \alpha_1} = \sum_{i=1}^{N} f_1(x_i) \cdot (y_i - \alpha_1 f_1(x_i)) = 0$$

$$\alpha_1 = \frac{\sum_{i=1}^{N} y_i f_1(x_i)}{\sum_{i=1}^{N} f_1(x_i)^2}$$

When $f_1(x_i)$ is the output of a linear regression model fitted to y_i, the weights of the model are chosen such that $f_1(x_i)$ minimizes the residuals $y_i - f_1(x_i)$.
This ensures that $f_1(x_i)$ perfectly matches y_i, making $\sum_{i=1}^{N} y_i f_1(x_i) = \sum_{i=1}^{N} f_1(x_i)^2$
Substituting into the formula for α_1, we get: $\alpha_1 = 1$

8.

In gradient boosting, the predictions at iteration t are updated as:

$$s_t = \alpha_t g_t + s_{t-1}$$

- $s_t = [s_1, s_2, \dots, s_N]$ is the vector of predictions at iteration t
- $g_t = [g_t(x_1), g_t(x_2), \dots, g_t(x_N)]$ is the gradient vector at iteration t
- α_t is the step size, computed as $\alpha_t = \frac{g_t(y-s_{t-1})^T}{g_t g_t^T}$

We can rewrite the summation in vectorized form

$$\sum_{n=1}^{N}(y_n - s_n)g_t(x_n) = (y - s_t)g_t^T$$

Using the update rule, we substitute it into the above equation:

$$y - s_t = y - (\alpha_t g_t + s_{t-1}) = (y - s_{t-1}) - \alpha_t g_t$$

$$\sum_{n=1}^{N}(y_n - s_n)g_t(x_n) = [(y - s_{t-1}) - \alpha_t g_t]g_t^T$$

$$= (y - s_{t-1})g_t^T - \alpha_t g_t g_t^T$$

From the step size formula, the first term $(y - s_{t-1})g_t^T = \alpha_t g_t g_t^T$

Therefore, the summation becomes

$$\sum_{n=1}^{N}(y_n - s_n)g_t(x_n) = \alpha_t g_t g_t^T - \alpha_t g_t g_t^T = 0$$

9.

Since all weights $w_{ij}^{(1)} = 0.5$, the input to each hidden-layer neuron $z_j^{(1)}$ is computed as

$$z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i.$$

Because all weights $w_{ij}^{(1)}$ are the same and the input x_i is the same for all neurons (if symmetrical), then $z_j^{(1)}$ will be equal for all j

The hidden layer's activation is $h_j^{(1)} = \tanh\left(z_j^{(1)}\right)$, since $z_j^{(1)}$ is the same for all j, all $h_j^{(1)}$ are equal.

In the output layer, since all hidden layer activations $h_j^{(1)}$ are equal, the output neuron receive symmetrical input. Consequently, the loss function will compute gradients symmetrically for all weights connecting the input to the hidden layer.

The gradient for a weight $w_{ij}^{(1)}$ is given by

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(1)}} = \delta_j^{(1)} \cdot x_i,$$

$$\delta_j^{(1)} = \frac{\partial \mathcal{L}}{\partial z_j^{(1)}} \cdot \tanh'(z_j^{(1)})$$

Because $z_j^{(1)}$ and $h_j^{(1)}$ are the same for all j, their derivatives $\tanh'(z_j^{(1)})$ are also equal.

Therefore, all $\delta_j^{(l)}$ are equal. Furthermore, since the input x_i is identical for corresponding weights, the gradients for $w_{ij}^{(1)}$ and $w_{i(j+1)}^{(1)}$ are identical

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(1)}} = \frac{\partial \mathcal{L}}{\partial w_{i(j+1)}^{(1)}}$$

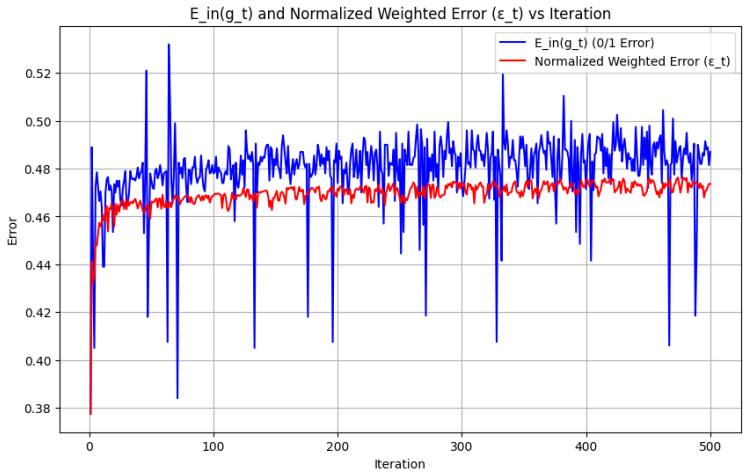
The weights are updated using the rule

$$w_{ij}^{(1)} \leftarrow w_{ij}^{(1)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{ij}^{(1)}}$$

Since the gradients are identical for $w_{ij}^{(l)}$ and $w_{i(j+1)}^{(l)}$, and the initial weights were equal ($w_{ij}^{(l)} = w_{i(j+1)}^{(l)}$=0.5), the weights will remain equal after each update.

Therefore, $w_{ij}^{(l)} = w_{i(j+1)}^{(l)}$ holds for all iterations.

10.




```python
# AdaBoost Implementation
def adaboost_fit(X, y, n_iterations=500):
    n_samples = X.shape[0]
    weights = np.ones(n_samples) / n_samples
    classifiers = []

    alphas = []
    errors = []

    for t in range(n_iterations):
        feature, threshold, polarity, error = decision_stump_fit(X, y, weights)

        alpha = 0.5 * np.log((1 - error) / (error + 1e-10))
        predictions = decision_stump_predict(X, feature, threshold, polarity)

        weights *= np.exp(-alpha * y * predictions)
        weights /= np.sum(weights)

        classifiers.append((feature, threshold, polarity, alpha))
        alphas.append(alpha)
        errors.append(error)

    return classifiers, alphas, errors
```

✓ 0.0s

```python
# Prediction
def adaboost_predict(X, classifiers):
    n_samples = X.shape[0]
    final_predictions = np.zeros(n_samples)

    for feature, threshold, polarity, alpha in classifiers:
        predictions = decision_stump_predict(X, feature, threshold, polarity)
        final_predictions += alpha * predictions

    return np.sign(final_predictions)
```

```
E_in: 0.041, E_out: 0.3933333333333333
```

Blue Line $E_{in}(G_t)$

The blue line represents the 0/1 error of each weak classifier on the training data. The fluctuations indicate that each weak classifier is optimized for the current weighted samples, and some classifiers may perform worse overall but are better at handling hard-to-classify samples.

Red Line ϵ_t

The red line represents the weighted error for each iteration, showing the performance of the weak classifier on the current weighted sample distribution. The red line is more stable and consistently below 0.5, which is essential for AdaBoost since $\epsilon_t \geq 0.5$ would mean the weak classifier does not contribute to improving the model.

The blue line illustrates the variability in weak classifier performance, while the red line reflects the stability of the weighted error rate. The results show that AdaBoost progressively enhances the performance of the weak classifiers, and the overall trend is reasonable and aligns with theoretical expectations.

For problems 10 to 12, the full code has been made available on GitHub.
Feel free to check it out if required or interested!

11.

Blue Line $E_{in}(G_t)$

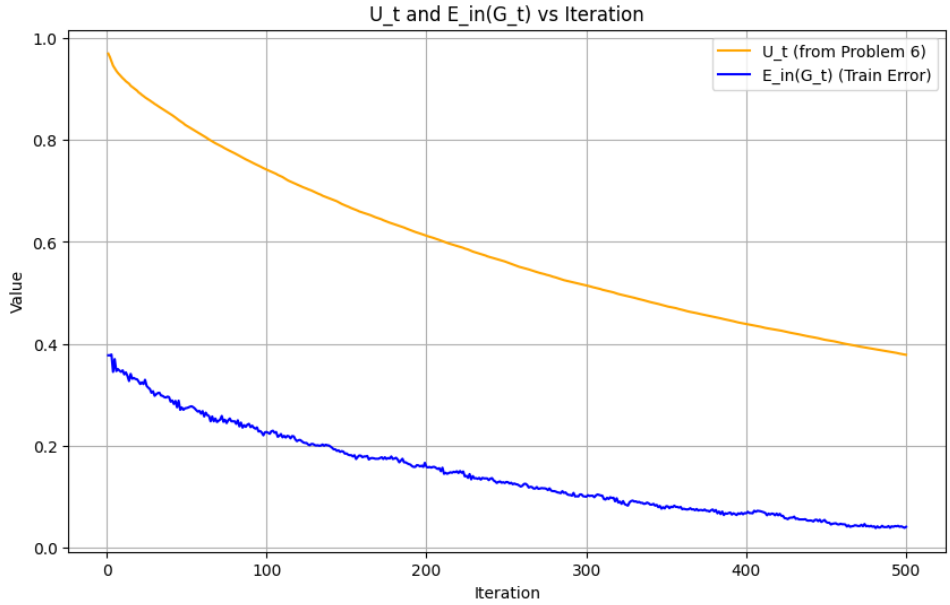
This line represents the 0/1 error on the training data. As iterations increase, the blue line decreases rapidly and approaches 0. This shows that the model is learning to classify the training data increasingly well, which aligns with AdaBoost's goal. Early iterations contribute significantly to reducing the error, while later iterations show slower improvement.

Blue Line $E_{out}(G_t)$

The green line represents the 0/1 error on the testing data. While it decreases in the early iterations, it eventually stabilizes and even slightly increases, indicating that the model may be overfitting the training data as the number of iterations grows.

The blue line shows the model's performance on the training data, while the green line indicates its generalization ability on test data. The rapid decrease in training error demonstrates the model's learning capability, while the stabilization or increase in test error reflects potential overfitting.

12.



Orange Line U_t

U_t represents a theoretical upper bound on the cumulative weight adjustment in AdaBoost. The orange line shows an exponential decay trend, which is reasonable because each iteration adjusts the weights based on the error rate. When the weighted error ϵ_t is close to 0.5, the decay is slower. When ϵ_t is closer to 0 or 1, the decay speeds up.

Blue Line $E_{in}(G_t)$

The blue line represents the 0/1 training error of the model. As iterations increase, the error steadily decreases, which makes sense because the model is improving its classification, especially on harder-to-classify samples. Eventually, $E_{in}(G_t)$ approaches 0, showing the model has nearly perfectly fit the training data.

The orange line represents the theoretical bound, and the blue line represents actual error. The theoretical U_t decreases faster, while the training error $E_{in}(G_t)$ slows down in the later iterations, indicating the model is well-trained but may start overfitting. Overall, the results are reasonable and align with expectations.

13.

ChatGPT: It is possible to implement d-bit XOR with d−1 hidden linear-threshold neurons.
2023 fall bonus homework: It is impossible to implement d-bit XOR with any d−(d−1)−1=0hidden layers, meaning a single-layer feed-forward network.

The difference is that ChatGPT assumes d−1 hidden neurons. With enough hidden neurons, we can split the XOR computation into smaller parts and combine them in the output layer. This approach works.
However, 2023 homework limits the network to 0 hidden layers. Without hidden layers, XOR is impossible to compute because linear boundaries can't solve it.

The reason why XOR can't be implemented with a single layer is that, for a Linear-Threshold Neuron (LTU) outputs $y = sign(\sum w_i x_i + b)$
Its decision boundary is always linear, meaning it can only separate data that is linearly separable. XOR outputs 1 if the number of 1s in the input is odd, otherwise 0, which we can't draw a straight line to separate 1s and 0s.

In conclusion, a single layer of LTUs cannot compute XOR because it lacks the non-linear power to handle XOR's structure. For XOR, we need at least one hidden layer to handle its non-linear separability.