HW7

Lo Chun, Chou
R13922136

December 17, 2024

5.

uniform blending $2M+1$ binary classifiers $\{g_t\}_{t=1}^{2M+1}$
$\Rightarrow$ aggregation binary classifier:

$$G(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^{2M+1} g_t(\mathbf{x})\right)$$

each g_t: test 0/1 error $E_{out}(g_t) = e_t = \mathbb{E}_{\mathbf{x}\sim P}[\![g_t(\mathbf{x}) \neq y]\!]$

Observe the formulation of $G(\mathbf{x})$, we can see that:

$$G(\mathbf{x}) = \begin{cases} 1, & \text{if } \#\{g_t(\mathbf{x}) = 1\} \geq M+1 \\ -1, & \text{if } \#\{g_t(\mathbf{x}) = 1\} \leq M \end{cases}$$

which means that $G(\mathbf{x})$ makes a wrong prediction when $\geq M+1$ of $g_t(\mathbf{x})$ are wrong.

Let X_t be a 0/1 indicator variable that outputs 1 when $g_t(\mathbf{x})$ is wrong, and 0 otherwise. We can write:

$$\mathbb{P}(X_t = i) = e_t^i(1-e_t)^{1-i} \quad , i \in \{0,1\}$$

Thus, we have:

$$\begin{cases} \mathbb{P}(X_t = 1) = e_t \\ \mathbb{P}(X_t = 0) = 1 - e_t \end{cases}$$

If we further let X denote the amount of $g_t(\mathbf{x})$ that are wrong, which is equivalent to the amount of $X_t = 1$. This makes X a random variable that represents "the amount of $X_t = 1$ in the $2M+1$ trials".

Since we are asked to not assume the independency among the binary classifier g_ts, the Hoeffding's inequality cannot be used. Instead, we can use another inequality that does not restrict independency, the Markov's inequality is as follows:

Let $Z \geq 0$ be a nonnegative random variable, then for all $t \geq 0$,

$$\boxed{\mathbb{P}(Z \geq t) \leq \frac{\mathbb{E}[Z]}{t}}$$

In order to calculate this bound, we need to first calculate $\mathbb{E}[X]$, which is the expected amount of $X_t = 1$ in the $2M + 1$ trials:

$$\mathbb{E}[X] = \sum_{t=1}^{2M+1} \mathbb{E}[X_t] = \sum_{t=1}^{2M+1} e_t$$

substisute back in and we'll get:

$$\mathbb{P}(X \geq M + 1) \leq \frac{1}{M+1} \sum_{t=1}^{2M+1} e_t$$

6.

Given:

$$U_t = \sum_{n=1}^{N} u_n^{(t)} \quad \text{which means } U_1 = 1$$
$$\text{assume } 0 < \epsilon_t < \frac{1}{2} \quad \text{for each hypothesis } g_t$$

Also, the definition of ϵ_t is:

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

Need to show:

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

Proof. The updating rule of $\mathbf{u}^{(t)}$ to $\mathbf{u}^{(t+1)}$ is:

$$u_n^{(t+1)} = \begin{cases} u_n^{(t)} \sqrt{\frac{1-\epsilon_t}{\epsilon_t}} & \text{if } g_t(x_n) \neq y_n \\ u_n^{(t)} \frac{1}{\sqrt{\frac{1-\epsilon_t}{\epsilon_t}}} & \text{if } g_t(x_n) = y_n \end{cases}$$

and we have:

$$\alpha_t = \ln\left(\sqrt{\frac{1-\epsilon_t}{\epsilon_t}}\right)$$

Therefore, it is equivalent to let:

$$u_n^{(t+1)} = \begin{cases} u_n^{(t)} e^{\alpha_t} & \text{if } g_t(x_n) \neq y_n \\ u_n^{(t)} e^{-\alpha_t} & \text{if } g_t(x_n) = y_n \end{cases}$$

Using the fact that $g_t(x_n)y_n = -1$ for incorrect examples and $g_t(x_n)y_n = 1$ for correct examples, we can write:

$$u_n^{(t+1)} = u_n^{(t)} e^{-\alpha_t g_t(x_n) y_n}$$

Thus, to calculate U_{t+1}, we have:

$$
\begin{aligned}
U_{t+1} &= \sum_{n=1}^{N} u_n^{(t+1)} \\
&= \sum_{n=1}^{N} u_n^{(t)} e^{-\alpha_t g_t(x_n) y_n} \\
&= \sum_{n=1}^{N} u_n^{(t)} \cdot \left[[\![y_n \neq g_t(x_n)]\!] \exp(\alpha_t) + [\![y_n = g_t(x_n)]\!] \exp(-\alpha_t) \right]
\end{aligned}
$$

Also, by the definition of ϵ_t, we have:

$$
\begin{aligned}
\epsilon_t &= \frac{\sum_{n=1}^{N} u_n^{(t)} [\![y_n \neq g_t(x_n)]\!]}{\sum_{n=1}^{N} u_n^{(t)}} = \frac{\sum_{n=1}^{N} u_n^{(t)} [\![y_n \neq g_t(x_n)]\!]}{U_t} \\
1 - \epsilon_t &= \frac{\sum_{n=1}^{N} u_n^{(t)} [\![y_n = g_t(x_n)]\!]}{U_t}
\end{aligned}
$$

which is equivalent to:

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

$$
\sum_{n=1}^{N} u_n^{(t)} [\![y_n = g_t(x_n)]\!] = (1 - \epsilon_t) U_t
$$

Therefore:

$$
\begin{aligned}
U_{t+1} &= \sum_{n=1}^{N} u_n^{(t)} [\![y_n \neq g_t(x_n)]\!] \exp(\alpha_t) + \sum_{n=1}^{N} u_n^{(t)} [\![y_n = g_t(x_n)]\!] \exp(-\alpha_t) \\
&= \epsilon_t U_t \exp(\alpha_t) + (1 - \epsilon_t) U_t \exp(-\alpha_t) \\
&= U_t \cdot \left[\epsilon_t \exp(\alpha_t) + (1 - \epsilon_t) \exp(-\alpha_t) \right]
\end{aligned}
$$

and

$$
\begin{aligned}
\frac{U_{t+1}}{U_t} &= \epsilon_t \exp(\alpha_t) + (1 - \epsilon_t) \exp(-\alpha_t) \\
&= \epsilon_t \sqrt{\frac{1 - \epsilon_t}{\epsilon_t}} + (1 - \epsilon_t) \sqrt{\frac{\epsilon_t}{1 - \epsilon_t}} \\
&= \sqrt{\epsilon_t (1 - \epsilon_t)} + \sqrt{\epsilon_t (1 - \epsilon_t)} \\
&= 2 \sqrt{\epsilon_t (1 - \epsilon_t)}
\end{aligned}
$$

$\square$

7.

In the original gradient boosting, the update rule is:

$$\theta^{(k)} = \theta^{(k-1)} + \alpha_k T_k$$

In this problem, we are asked to use linear regression (without regularization) to replace decision trees, which means:

$$\theta^{(k)} = \theta^{(k-1)} + \alpha_k \text{LR}_k$$

where LR_k is defined as follows:

$$\text{LR}_k(X) = X\mathbf{w}_k$$

where $X \in \mathbb{R}^{N\times d}$ is the feature matrix, and $\mathbf{w}_k \in \mathbb{R}^d$ is the weight vector at iteration k.
To find $\mathbf{w}_k$, we fit the residuals $\mathbf{r}^{(k)}$, which is calculated by:

$$\mathbf{r}^{(k)} = \mathbf{y} - \hat{\mathbf{y}}^{(k-1)}$$

where $\hat{\mathbf{y}}^{(k-1)} = X\hat{\mathbf{w}}^{(k-1)}$ is the prediction of the previous iteration.
Thus, we find $\mathbf{w}_k$ by using least squares:

$$\mathbf{w}_k = (X^TX)^{-1}X^T\mathbf{r}^{(k)}$$

and we can rewrite $\text{LR}_k(X)$ as:

$$\text{LR}_k(X) = X(X^TX)^{-1}X^T\mathbf{r}^{(k)}$$

To check if $\alpha_1 = 1$ is optimal, we need to check if it results in a minimized error (using the model $\theta^{(1)}$)

If we assume the prediction of the initial model is $\theta^{(0)}(X) = X\mathbf{w}_0$, then prediction of $\theta^{(1)}$ is:

$$\begin{aligned}\theta^{(1)}(X) &= \theta^{(0)}(X) + \alpha_1\text{LR}_1(X)\\ &= X\mathbf{w}_0 + \alpha_1 X(X^TX)^{-1}X^T\mathbf{r}^{(1)}\end{aligned}$$

and the loss incured in iteration 1 is:

$$\begin{aligned}\left\|\mathbf{y} - \theta^{(1)}(X)\right\|^2 &= \left\|\mathbf{y} - X\mathbf{w}_0 - \alpha_1 X(X^TX)^{-1}X^T\mathbf{r}^{(1)}\right\|^2\\ &= \left\|\mathbf{r}^{(1)} - \alpha_1 X(X^TX)^{-1}X^T\mathbf{r}^{(1)}\right\|^2\\ &= \left(\mathbf{r}^{(1)} - \alpha_1 X(X^TX)^{-1}X^T\mathbf{r}^{(1)}\right)^T\left(\mathbf{r}^{(1)} - \alpha_1 X(X^TX)^{-1}X^T\mathbf{r}^{(1)}\right)\\ &= \mathbf{r}^{(1)T}\mathbf{r}^{(1)} - 2\alpha_1\mathbf{r}^{(1)T}X(X^TX)^{-1}X^T\mathbf{r}^{(1)} + \alpha_1^2\mathbf{r}^{(1)T}X(X^TX)^{-1}X^TX(X^TX)^{-1}X^T\mathbf{r}^{(1)}\\ &= \mathbf{r}^{(1)T}\mathbf{r}^{(1)} - 2\alpha_1\mathbf{r}^{(1)T}X(X^TX)^{-1}X^T\mathbf{r}^{(1)} + \alpha_1^2\mathbf{r}^{(1)T}X(X^TX)^{-1}X^T\mathbf{r}^{(1)}\end{aligned}$$

Taking the derivative w.r.t. α_1 and set it to 0, we have:

$$\begin{aligned}
&\frac{d}{d\alpha_1}\left(\mathbf{r}^{(1)T}\mathbf{r}^{(1)} - 2\alpha_1\mathbf{r}^{(1)T}X(X^TX)^{-1}X^T\mathbf{r}^{(1)} + \alpha_1^2\mathbf{r}^{(1)T}X(X^TX)^{-1}X^T\mathbf{r}^{(1)}\right) = 0\\
&\Rightarrow -2\mathbf{r}^{(1)T}X(X^TX)^{-1}X^T\mathbf{r}^{(1)} + 2\alpha_1\mathbf{r}^{(1)T}X(X^TX)^{-1}X^T\mathbf{r}^{(1)} = 0\\
&\Rightarrow \alpha_1 = \frac{2\mathbf{r}^{(1)T}X(X^TX)^{-1}X^T\mathbf{r}^{(1)}}{2\mathbf{r}^{(1)T}X(X^TX)^{-1}X^T\mathbf{r}^{(1)}}\\
&\Rightarrow \alpha_1 = 1
\end{aligned}$$

Therefore, $\alpha_1 = 1$ is optimal.

8.

After updating all s_n in iteration t, using the steepest η as α_t need to prove that:

$$\sum_{n=1}^{N}(y_n - s_n^{(t)})g_t(\mathbf{x}_n) = 0$$

Proof. At iteration t, since $\alpha_t = \eta_t$, we update the model by:

$$\theta^{(t+1)} = \theta^{(t)} + \eta_t g_t$$

To update individual predictions $s_n^{(t)}$, we have:

$$s_n^{(t)} = s_n^{(t-1)} + \eta_t g_t(\mathbf{x}_n)$$

To obtain g_t, we need to train it by fitting the residuals:

$$r_n^{(t)} = y_n - s_n^{(t-1)}$$

so that g_t minimizes the loss:

$$\sum_{n=1}^{N}\left(r_n^{(t)} - g_t(\mathbf{x}_n)\right)^2 = \sum_{n=1}^{N}\left(y_n - s_n^{(t-1)} - g_t(\mathbf{x}_n)\right)^2$$

Thus, we take the derivative w.r.t. g_t and set it to 0, and we'll have:

$$\begin{aligned}
&\frac{d}{dg_t}\sum_{n=1}^{N}\left(y_n - s_n^{(t-1)} - g_t(\mathbf{x}_n)\right)^2 = 0 \\
&\Rightarrow -2\sum_{n=1}^{N}\left(y_n - s_n^{(t-1)} - g_t(\mathbf{x}_n)\right) = 0 \\
&\Rightarrow \sum_{n=1}^{N}\left(y_n - s_n^{(t-1)} - g_t(\mathbf{x}_n)\right) = 0 \\
&\Rightarrow \sum_{n=1}^{N}\left(y_n - s_n^{(t-1)}\right) = \sum_{n=1}^{N} g_t(\mathbf{x}_n) \\
&\Rightarrow \sum_{n=1}^{N} r_n^{(t)} = \sum_{n=1}^{N} g_t(\mathbf{x}_n)
\end{aligned}$$

The original equation that we want to prove can then be derived as follows:

$$\begin{aligned}\sum_{n=1}^{N}(y_n - s_n^{(t)})g_t(\mathbf{x}_n) &= \sum_{n=1}^{N}\left(y_n - s_n^{(t-1)} - \eta_t g_t(\mathbf{x}_n)\right) g_t(\mathbf{x}_n) \\ &= \sum_{n=1}^{N}\left(r_n^{(t)} - \eta_t g_t(\mathbf{x}_n)\right) g_t(\mathbf{x}_n) \\ &= \sum_{n=1}^{N} r_n^{(t)} g_t(\mathbf{x}_n) - \eta_t \sum_{n=1}^{N} g_t(\mathbf{x}_n)^2 \qquad (*)\end{aligned}$$

By lecture note 11, p.19, we have:

$$\frac{\sum_{n=1}^{N} g_t(\mathbf{x}_n)(y_n - s_n^{(t-1)})}{\sum_{n=1}^{N} g_t(\mathbf{x}_n)^2} = \arg\min_{\eta} \frac{1}{N}\sum_{n=1}^{N}\left((y_n - s_n^{(t-1)}) - \eta g_t(\mathbf{x}_n)\right)^2$$

Thus, by plugging in this result into equation $(*)$, we have:

$$\begin{aligned}&\sum_{n=1}^{N}(y_n - s_n^{(t)})g_t(\mathbf{x}_n) \\ &= \sum_{n=1}^{N} r_n^{(t)} g_t(\mathbf{x}_n) - \frac{\sum_{n=1}^{N} g_t(\mathbf{x}_n)(y_n - s_n)}{\sum_{n=1}^{N} g_t(\mathbf{x}_n)^2} \sum_{n=1}^{N} g_t(\mathbf{x}_n)^2 \\ &= \sum_{n=1}^{N} r_n^{(t)} g_t(\mathbf{x}_n) - \sum_{n=1}^{N} g_t(\mathbf{x}_n)(y_n - s_n^{(t-1)}) \\ &= \sum_{n=1}^{N} r_n^{(t)} g_t(\mathbf{x}_n) - \sum_{n=1}^{N} r_n^{(t)} g_t(\mathbf{x}_n) \\ &= 0\end{aligned}$$

$\square$

9.

Initially, for the neurons j and $j+1$ in the hidden layer, we have:

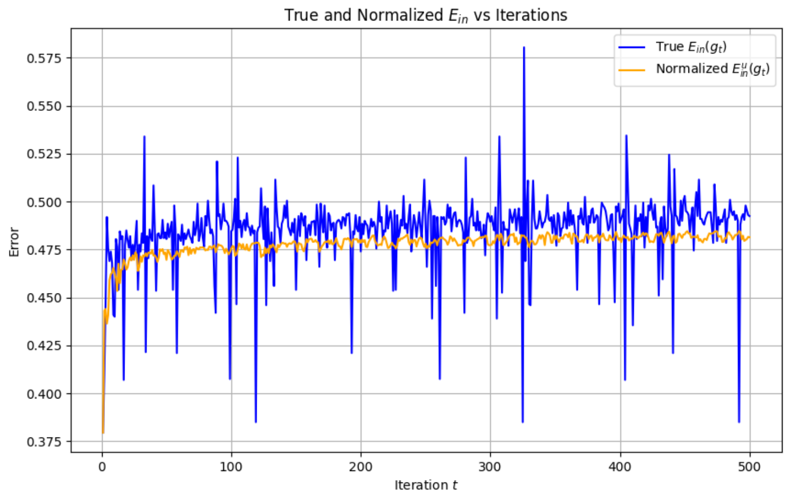
$$z_j = \sum_{i=1}^{d} w_{ij}^{(1)} x_i + b_j^{(1)}, \quad z_{j+1} = \sum_{i=1}^{d} w_{i,j+1}^{(1)} x_i + b_{j+1}^{(1)}$$

and the outputs are:

$$h_j = \tanh(z_j), \quad h_{j+1} = \tanh(z_{j+1})$$

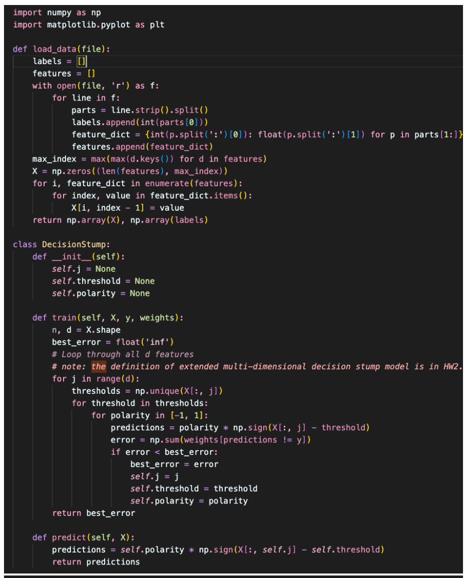
Since all the initial weights $w_{i,j}^{l} = 0.5$

10.

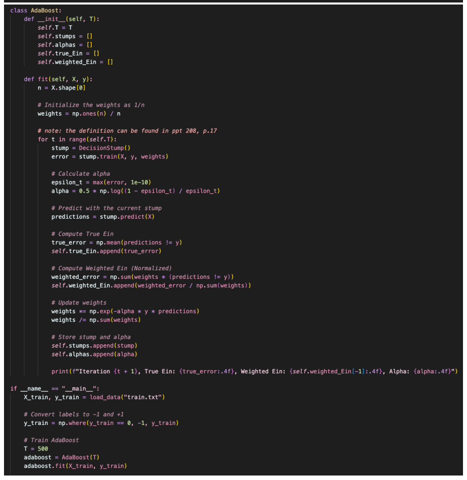


From this plot we can see that, the true E_{in} (the blue curve) fluctuates greatly, this may due to the fact that AdaBoost place emphasis on the misclassified examples by adding more weights on them, since focusing on specific examples can bring instability in the performance of individual weak learners.

The normalized E_{in}^{u} (the orange curve) measures the weighted error. While AdaBoost adjusts weights to focus more on the misclassified examples at each iteration, this weighted error reflects how well the weak learner is performing on those examples.

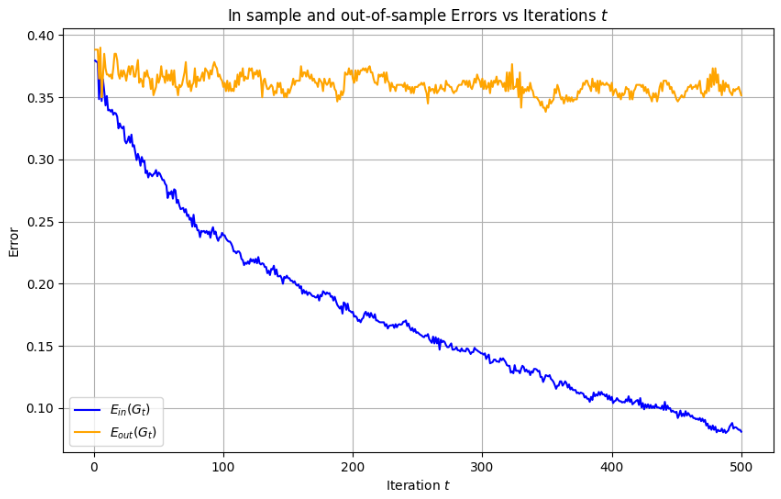


```python
import numpy as np
import matplotlib.pyplot as plt

def load_data(file):
    labels = []
    features = []
    with open(file, 'r') as f:
        for line in f:
            parts = line.strip().split()
            labels.append(int(parts[0]))
            feature_dict = {int(p.split(':')[0]): float(p.split(':')[1]) for p in parts[1:]}
            features.append(feature_dict)
    max_index = max(max(d.keys()) for d in features)
    X = np.zeros((len(features), max_index))
    for i, feature_dict in enumerate(features):
        for index, value in feature_dict.items():
            X[i, index - 1] = value
    return np.array(X), np.array(labels)

class DecisionStump:
    def __init__(self):
        self.j = None
        self.threshold = None
        self.polarity = None

    def train(self, X, y, weights):
        n, d = X.shape
        best_error = float('inf')
        # Loop through all d features
        # note: the definition of extended multi-dimensional decision stump model is in HW2.
        for j in range(d):
            thresholds = np.unique(X[:, j])
            for threshold in thresholds:
                for polarity in [-1, 1]:
                    predictions = polarity * np.sign(X[:, j] - threshold)
                    error = np.sum(weights[predictions != y])
                    if error < best_error:
                        best_error = error
                        self.j = j
                        self.threshold = threshold
                        self.polarity = polarity
        return best_error

    def predict(self, X):
        predictions = self.polarity * np.sign(X[:, self.j] - self.threshold)
        return predictions
```



```python
class AdaBoost:
    def __init__(self, T):
        self.T = T
        self.stumps = []
        self.alphas = []
        self.true_Ein = []
        self.weighted_Ein = []

    def fit(self, X, y):
        n = X.shape[0]

        # Initialize the weights as 1/n
        weights = np.ones(n) / n

        # note: the definition can be found in ppt 208, p.17
        for t in range(self.T):
            stump = DecisionStump()
            error = stump.train(X, y, weights)

            # Calculate alpha
            epsilon_t = max(error, 1e-10)
            alpha = 0.5 * np.log((1 - epsilon_t) / epsilon_t)

            # Predict with the current stump
            predictions = stump.predict(X)

            # Compute True Ein
            true_error = np.mean(predictions != y)
            self.true_Ein.append(true_error)

            # Compute Weighted Ein (Normalized)
            weighted_error = np.sum(weights * (predictions != y))
            self.weighted_Ein.append(weighted_error / np.sum(weights))

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

            # Store stump and alpha
            self.stumps.append(stump)
            self.alphas.append(alpha)

            print(f"Iteration {t + 1}, True Ein: {true_error:.4f}, Weighted Ein: {self.weighted_Ein[-1]:.4f}, Alpha: {alpha:.4f}")

if __name__ == "__main__":
    X_train, y_train = load_data("train.txt")

    # Convert labels to -1 and +1
    y_train = np.where(y_train == 0, -1, y_train)

    # Train AdaBoost
    T = 500
    adaboost = AdaBoost(T)
    adaboost.fit(X_train, y_train)
```

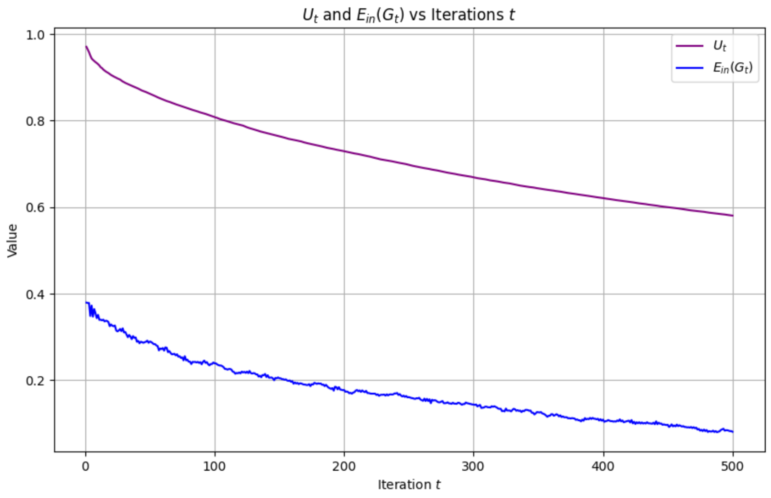
11.



From the above figure, we can see that the blue curve (which is the in sample error), shows that E_{in} decreases steadily as the number of iterations t increases. When we got to $T = 500$, the error is quite low, which matches our expectation since the target of AdaBoost is to iteratively minimize the training error by giving the misclassified exampels higher weights. Therefore, the model becomes increasingly fitted to the training data.

For the orange curve (which is the out-of-sample error), shows that even though the curve fluctuates, after we got to $T = 500$, it did not decrease like E_in does. This implies that our model is not generalizing well, and this result is as expected, since as we make the model more fitted to the training data, overfitting must happen.

12.



The purple curve (which represents the sum of weights U_t) is steadily decreasing as the iteration number decreases. We can see that initially, this value matches the definition that all weights are $\frac{1}{n}$, and therefore making the sum to be 1.

The decreased result also matches our expectation, since if the weak learners consistently achieve errors $\epsilon_\tau < 0.5$, the value U_t keeps decreasing. Moreover, as the value of ϵ_τ becomes smaller, the decreasing behavior should slow down, and this can also be seen in our plot.

From the blue curve, we can see that $E_{in}(G_t)$ is also steadily decreasing. This shows that U_t and $E_{in}(G_t)$ have similar behavior as the number of iteration increases. The reason why $E_{in}(G_t)$ is decreasing is that when we add more weak classifiers to our model, it becomes stronger, hence having the ability to fit the examples better, and thus reduces the error value.

13.

From the reply of ChatGPT, it said that:

For 3 input bits (x_1, x_2, x_3), the hidden neuron could compute:

$$\begin{aligned} h_1 &= x_1 \oplus x_2 \\ h_2 &= h_1 \oplus x_3 \end{aligned}$$

To see how many neurons is required to implement the XOR function for $d = 3$, we first consider the implementation of the XOR function for $d = 2$, which is:

$$h_1 = x_1 \oplus x_2$$

From the property that XOR can be implemented using AND and OR gates, which is:

$$A \oplus B = (\bar{A} \cdot B) + (A \cdot \bar{B})$$

we can obtain h_1 by using 2 neurons, which are the two terms in the following equation:

$$h_1 = (\bar{x}_1 \cdot x_2) + (x_1 \cdot \bar{x}_2)$$

Using this result and getting back to the XOR function for $d = 3$, we have:

$$\begin{aligned} x_1 \oplus x_2 \oplus x_3 &= (x_1 \oplus x_2) \oplus x_3 \\ &= (\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2) \oplus x_3 \\ &= \overline{(\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2)} \cdot x_3 + (\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2) \cdot \bar{x}_3 \\ &= (\overline{\bar{x}_1 \cdot x_2} \cdot \overline{x_1 \cdot \bar{x}_2}) \cdot x_3 + (\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2) \cdot \bar{x}_3 \\ &= (x_1 + \bar{x}_2) \cdot (\bar{x}_1 + x_2) \cdot x_3 + (\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2) \cdot \bar{x}_3 \\ &= x_1\bar{x}_1x_3 + x_1x_2x_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_2x_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 \\ &= x_1x_2x_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 \end{aligned}$$

By this result, we can see that four neurons are required for the case $d = 3$. Hence, we have $2 - 2 - 1$ for two input bits, and $3 - 4 - 1$ for three input bits, if we observe the result, we can find that we need:

- one term for all of the input bits are not inverted (i.e. $x_1x_2x_3$ in this example)

- d terms for each input bit not inverted, and others inverted (i.e. $x_1\bar{x}_2\bar{x}_3,\ \bar{x}_1x_2\bar{x}_3,\ \bar{x}_1\bar{x}_2x_3$)

Therefore, to implement a XOR function on d input bits, we need $d+1$ neurons in the hidden layer, which shows the insufficiency for using $d-1$ neurons.