Question 1
Adaboost Algorithm Implementation in the Code:
Step 1: Data Preparation
The code starts by loading the MNIST dataset and selecting only the samples labelled as 0 and 1.

Step 2: Label Encoding
The label 0 is changed to -1 to conform to the convention of AdaBoost where weak learners typically predict class labels as either -1 or 1.

Step 3: Data Flattening
The matrix representing the images is flattened to convert each image into a one-dimensional array.

Step 4: Dimensionality Reduction
Principal Component Analysis (PCA) is performed to reduce the dimensionality of the flattened data, resulting in a matrix with dimensions 10665x5. This reduces computational complexity and focuses on the most important features.

Step 5: AdaBoost Algorithm Implementation
AdaBoost, or Adaptive Boosting, is a machine learning ensemble meta-algorithm used to improve the performance of weak learners.

Algorithm Steps:
Initialization: Each training instance is initially assigned an equal weight.

Iterative Training:

For each iteration:

> A weak learner (e.g., decision stump) is trained on the training data. The weak learner focuses on the instances that were misclassified in the previous iteration.
> The error of the weak learner on the training set is computed.
> The weight of the weak learner's vote in the final classification is computed based on its error rate.
> The weights of the training instances are updated, giving more weight to the misclassified instances.

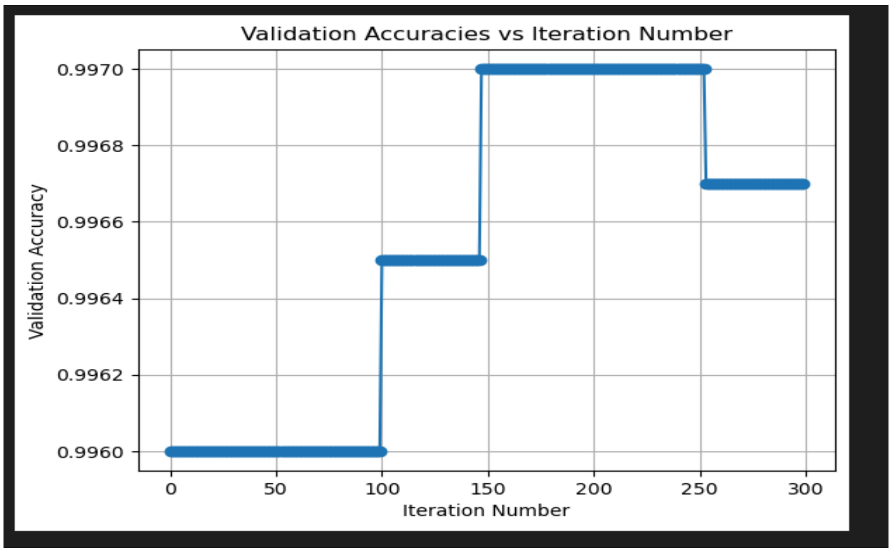
Final Model Construction:
The weak learners are combined into a strong classifier by assigning weights to their predictions based on their performance.
The final model is a weighted combination of the weak learners' predictions.

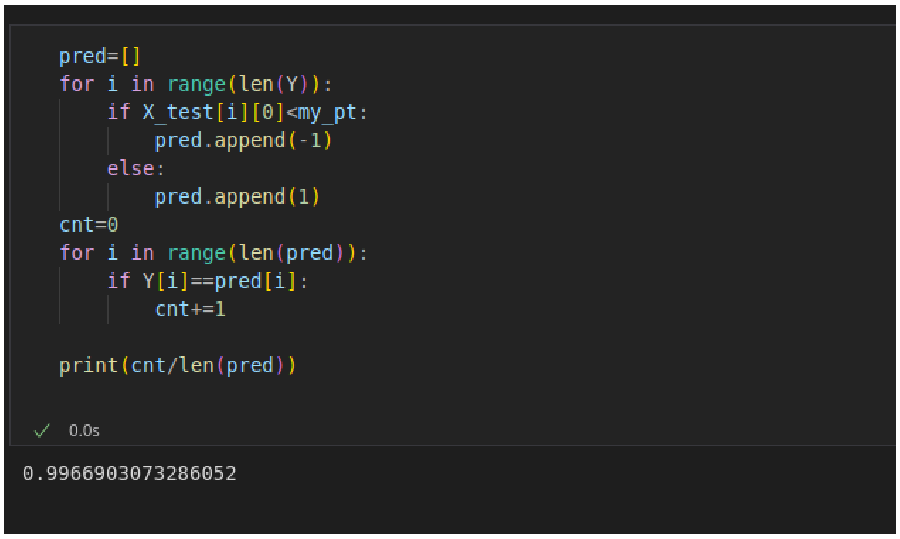
Prediction:

To make predictions with the AdaBoost model:
Each weak learner predicts the class label of a given instance.
The final prediction is determined by combining the individual weak learner predictions, weighted by their respective vote weights.

Advantages:
AdaBoost is effective in improving the accuracy of weak classifiers.
It is less susceptible to overfitting compared to individual weak learners.
Can handle complex decision boundaries.

Tree vs accuracy graph





```
pred=[]
for i in range(len(Y)):
    if X_test[i][0]<my_pt:
        pred.append(-1)
    else:
        pred.append(1)
cnt=0
for i in range(len(pred)):
    if Y[i]==pred[i]:
        cnt+=1

print(cnt/len(pred))
```

✓ 0.0s

```
0.9966903073286052
```

Gradient Boosting Algorithm Implementation:

Step 1: Data Preparation

- The code begins by loading the MNIST dataset and extracting samples labeled as 0 and 1.
- To prepare for regression, where the goal is to predict continuous values, the label 0 is mapped to -1.

Step 2: Data Preprocessing

- The input images are flattened into one-dimensional arrays to facilitate processing.
- Principal Component Analysis (PCA) is applied to reduce the dimensionality of the data to 5 dimensions. This reduces computational complexity while retaining the most relevant information.

Step 3: Initialization

- Residuals (or gradients) are initialized as the difference between the actual labels and the initial predictions.
- Variables to store the best split feature and value are defined.

Step 4: Gradient Boosting Iteration

- For each boosting iteration:
    - The algorithm searches for the best split feature and value to minimize the sum of squared residuals (SSR). This is typically done by exhaustively evaluating all possible splits and selecting the one that minimizes SSR.
    - Once the best split is found, the predictions are updated based on this split.
    - The residuals are then updated based on the new predictions. This step ensures that subsequent weak learners focus on the errors made by the previous ones.
    - The mean squared error (MSE) on the validation set is computed for each iteration and stored in mse_values for later analysis.

Step 5: Model Evaluation

- The code plots the mean squared error (MSE) on the validation set versus the number of boosting iterations. This visualization helps assess the model's performance and identify the optimal number of iterations.

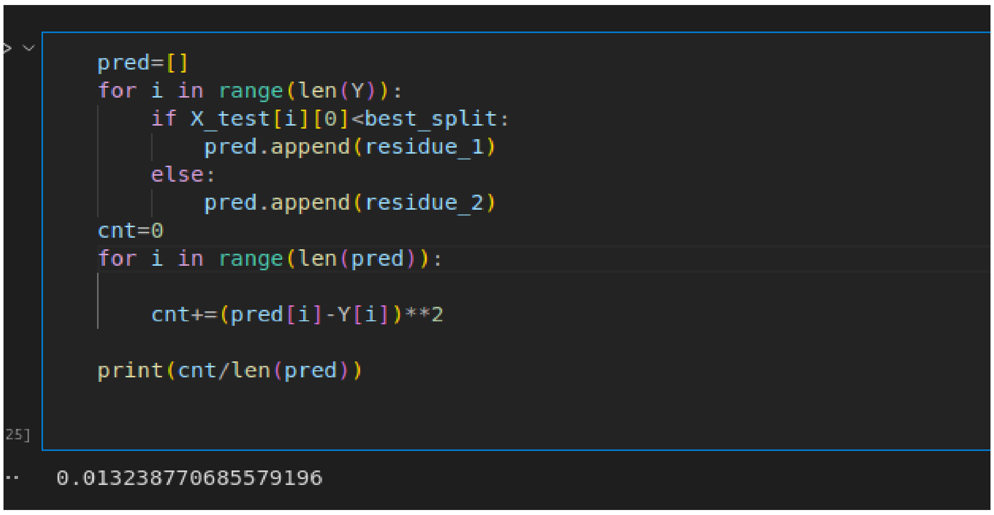
Step 6: Testing

- Finally, the code prepares the test data in the same manner as the training data.

- It makes predictions on the test data using the trained model and calculates the mean squared error (MSE) between the predicted and actual labels. This metric quantifies the model's performance on unseen data.

Conclusion:

The provided code implements a gradient boosting algorithm from scratch for regression tasks, specifically applied to the MNIST dataset. By iteratively refining predictions based on the residuals of the previous iterations, the algorithm constructs a strong ensemble model. The model's performance is evaluated using mean squared error (MSE), both on a validation set during training and on a separate test set for final assessment. This detailed breakdown highlights the steps involved in building and evaluating a gradient boosting model for regression.

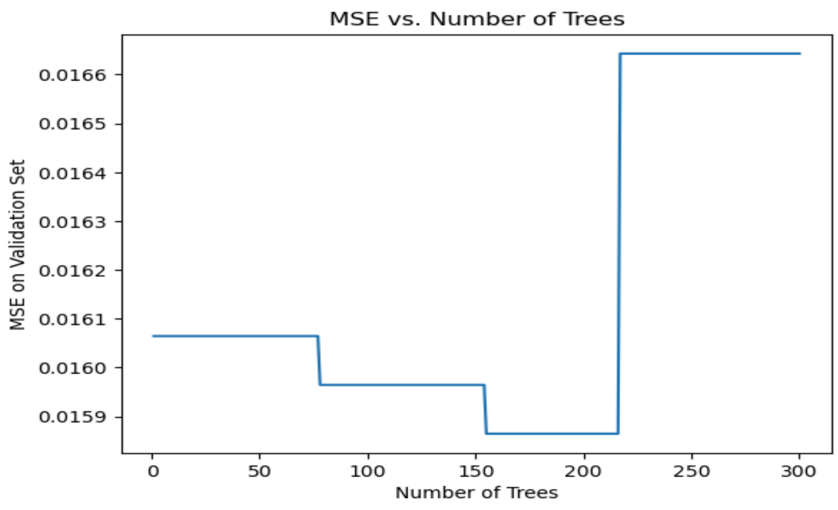


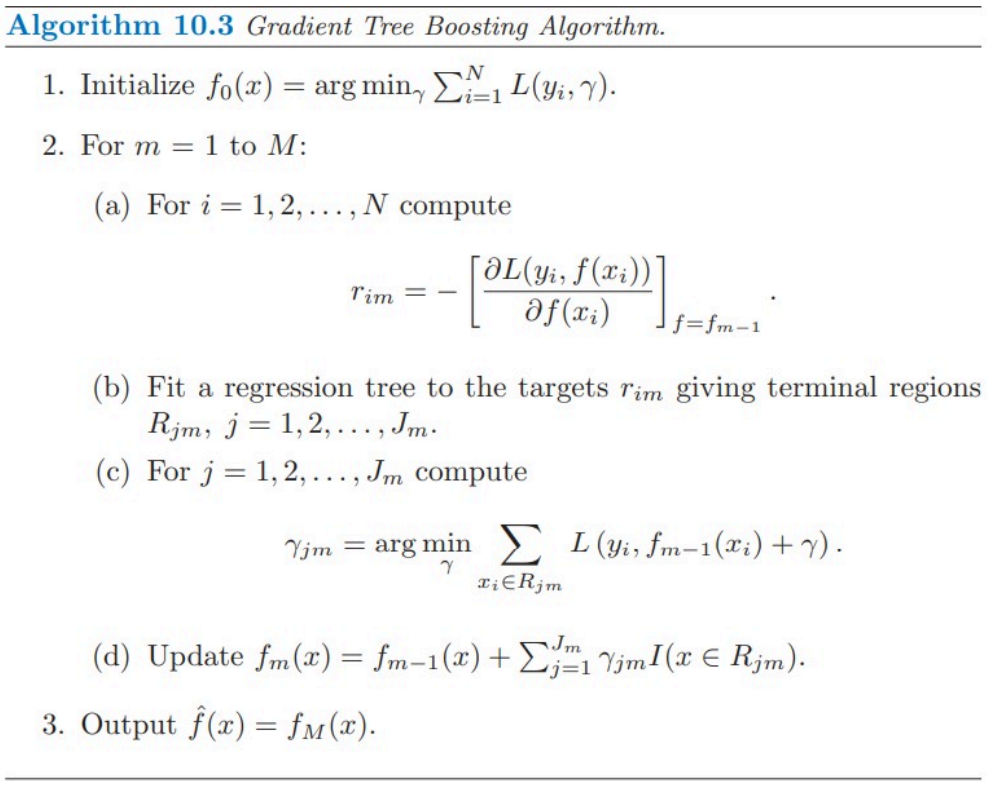
```
pred=[]
for i in range(len(Y)):
    if X_test[i][0]<best_split:
        pred.append(residue_1)
    else:
        pred.append(residue_2)
cnt=0
for i in range(len(pred)):

    cnt+=(pred[i]-Y[i])**2

print(cnt/len(pred))
```

```
0.013238770685579196
```

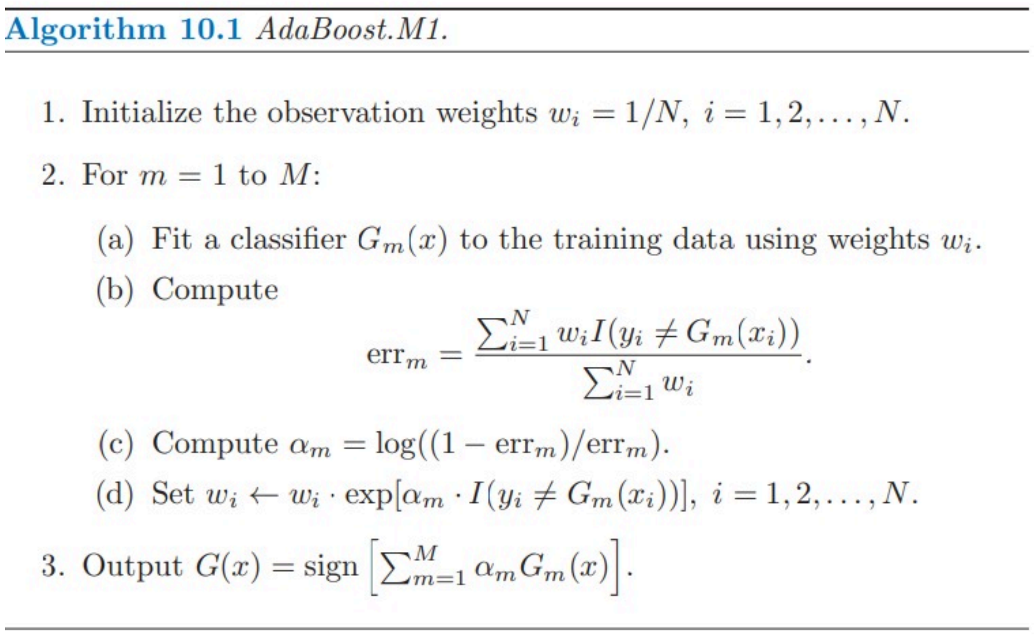




Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

```
1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to $M$:
    (a) For $i = 1, 2, \dots, N$ compute
            $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
    (b) Fit a regression tree to the targets $r_{im}$ giving terminal regions
        $R_{jm},\ j = 1, 2, \dots, J_m$.
    (c) For $j = 1, 2, \dots, J_m$ compute
            $$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
    (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.
```

Used this method in my code which is taken from Element of statistical learning



Algorithm 10.1 *AdaBoost.M1.*

```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
        $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
    (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.
```

Used this method in my code which is taken from Element of statistical learning