Experiment No. 4

Title - Implement Boosting Algorithm

Theory -

Boosting is an ensemble machine learning technique that builds a strong learner by sequentially training multiple weak learners. The key principle behind boosting is to focus on correcting the mistakes of preceding models, emphasising the misclassified instances to improve overall prediction accuracy. Here's an overview of the boosting algorithm:

Steps

1. Initialisation:

- Assign equal weights to all data points in the training set.
- Choose a base (weak) learner, often a simple model like a decision stump (a one-level decision tree).

2. Iterative Training:

- Train the base learner on the training set with the assigned weights.
- Calculate the model's error, typically using misclassification error or another chosen metric.
- Adjust instance weights:
  - Increase the weights of misclassified instances to focus more on them in the next iteration.
  - Decrease the weights of correctly classified instances to reduce their impact.
  - Re-assign weights to data points based on the error made by the current model.

3. Sequential Learning:

- Build a new model, which focuses more on the previously misclassified instances.
- Iterate this process for a predefined number of iterations or until a stopping criterion is met.

4. Model Combination:

- Aggregate the predictions of all models with different weights, usually using a weighted majority vote for classification or weighted averaging for regression.

Key Concepts

- Weighted Data Points: Instances that were misclassified by the previous models are given higher weights to force the subsequent models to focus on these errors.
- Sequential Correction: Each new model attempts to correct the mistakes made by the ensemble of previously trained models.
- Adaptive Learning: Boosting is adaptive as subsequent models learn from the errors made by earlier models, improving overall performance.

Popular Boosting Algorithms

- AdaBoost (Adaptive Boosting): It adjusts weights of misclassified samples to improve performance.
- Gradient Boosting Machines (GBM): It builds models in a sequential manner by minimising loss functions.
- XGBoost (Extreme Gradient Boosting): An optimised implementation of gradient boosting with regularisation and parallel processing.

Mathematics -

- **Sample Weight Update**

$$D_{t,i} = D_{t-1,i} \cdot e^{-\eta_t \cdot y_i \cdot h_t(\mathbf{x}_i)} \cdot D_{t,i}$$

$D_{t,\,i}$ - weight of the i^{th} training example at iteration t .

$D_{t\text{-}1,\,i}$ - previous weight of the same training example.

η_t - Learning rate at iteration t.

y_t - true label of the i^{th} example.

$h_t\ (x_i)$ - prediction of the weak learner at iteration t for the i^{th} example.

- **Error of Weak Learner**

$$\epsilon_t = \sum_{i=1}^{n} D_{t,i} \cdot y_i \cdot h_t(\mathbf{x}_i) - \sum_{i=1}^{n} y_i \cdot h_t(\mathbf{x}_i)$$

ϵ_t - weighted error of the weak learner at iteration t.

- **Contribution of Weak Learner**

$$\alpha_t = \frac{1}{2} \cdot \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

α_t - represents the contribution weight of the weak learner at iteration ***t .***

Implementation

```
class Boosting:
    def __init__(self,dataset,T,test_dataset):
        self.dataset = dataset
        self.T = T
        self.test_dataset = test_dataset
        self.alphas = None
        self.models = None
        self.accuracy = []
        self.predictions = None

    def fit(self):
        # Set the descriptive features and the target feature
        X = self.dataset.drop(['stroke'],axis=1)
        Y = self.dataset['stroke'].where(self.dataset['stroke']==1,-1)

        # Initialize the weights of each sample with wi = 1/N
        Evaluation = pd.DataFrame(Y.copy())
        Evaluation['weights'] = 1/len(self.dataset) # Set the initial weights w = 1/N

        # Run the boosting algorithm by creating T "weighted models"

        alphas = []
        models = []

        for t in range(self.T):
            Tree_model = DecisionTreeClassifier(criterion="entropy",max_depth=1)
            model = Tree_model.fit(X,Y,sample_weight=np.array(Evaluation['weights']))

           # Append the single weak classifiers to a list
            models.append(model)
            predictions = model.predict(X)
            score = model.score(X,Y)

            # Add values to the Evaluation DataFrame
            Evaluation['predictions'] = predictions
            Evaluation['evaluation'] = np.where(Evaluation['predictions'] == Evaluation['stroke'],1,0)
            Evaluation['misclassified'] = np.where(Evaluation['predictions'] != Evaluation['stroke'],1,0)
            # Calculate the misclassification rate and accuracy
            accuracy = sum(Evaluation['evaluation'])/len(Evaluation['evaluation'])
            misclassification = sum(Evaluation['misclassified'])/len(Evaluation['misclassified'])

            # Caclulate the error

            err = np.sum(Evaluation[‘weights'] *
                                                  Evaluation['misclassified'])/np.sum(Evaluation['weights'])

            # Calculate the alpha values
            alpha = np.log((1-err)/err)
            alphas.append(alpha)

            # Update the weights wi --> These updated weights are used in the sample_weight parameter
            # for the training of the next decision stump.
            Evaluation['weights']  *= np.exp(alpha*Evaluation['misclassified'])

            #print('The Accuracy of the {0}. model is : '.format(t+1),accuracy*100,'%')
            #print('The missclassification rate is: ',misclassification*100,'%')

        self.alphas = alphas
        self.models = models

    def predict(self):
        X_test = self.test_dataset.drop(['stroke'],axis=1).reindex(range(len(self.test_dataset)))
                                                                                        Y_test      =
self.test_dataset['stroke'].reindex(range(len(self.test_dataset))).where(self.dataset['stroke']==1,-1)

        # With each model in the self.model list, make a prediction
        accuracy = []
        predictions = []

        for alpha,model in zip(self.alphas,self.models):
                    prediction = alpha*model.predict(X_test) # We use the predict method for the single
decisiontreeclassifier models in the list
            predictions.append(prediction)

self.accuracy.append(np.sum(np.sign(np.sum(np.array(predictions),axis=0))==Y_test.values)/
len(predictions[0]))
        self.predictions = np.sign(np.sum(np.array(predictions),axis=0)
```

Benefits of Boosting

- Increased Accuracy: Boosting tends to produce highly accurate models by iteratively focusing on difficult instances.
- Versatility: It can be used with various base models and is effective in both regression and classification problems.
- Robustness: Boosting reduces bias and variance, leading to robust models that generalise well on unseen data.

Conclusion

Boosting is a powerful technique in machine learning, often employed to enhance the predictive performance of models by combining multiple weak learners into a robust and accurate ensemble model.