Implementation of Adaboost algorithm from scratch

Project link:
https://colab.research.google.com/drive/1ZNJvkZLs3rxM3bSUpWHwLXtZMVf2tBwm?usp=sharing

1. Introduction

The data selected to learn and predict is the breast cancer data set from the UCI Machine Learning repository that has been loaded into the SKLEARN package. This is a multivariate data set that has the diagnosis of cancer for 539 samples based on 30 features extracted from digitized images of a fine needle aspirate (FNA) of their breast mass. The target is to predict the diagnosis which could be benign or malign, and it has a non-linear distribution.

The Adaboost technique has been chosen because it was designed to tackle non-linear distributions and can be used on linear as well. Adaboost is an ensemble method. It uses basic decision trees that behave a little bit better than flipping a coin, they are called weak learners; thus, each prediction is adjusted iteratively through the significance of the samples and the influence of each weak learner and can come up with better predictions. In this case, the data set will be resampled according to the weight assigned to misclassified samples in a loop until all the weak learners specified have learnt.

This report presents the implementation of the **Adaboost algorithm for binary classification**, where the target has a non-linear distribution. This algorithm is simple to understand and implement, and has a powerful predictivity. Lately, there are variations on the boosting technique, such as XGBoost, which improves its computing performance. XGBoost has been used in some competitions winning ample by performance and accuracy.

2. Learning framework

The machine learning algorithm to construct the Adaboost algorithm has this **learning framework**:

- **Data**: The data chosen is from SKLEARN.DATASETS and belongs to the **Classification** task.
- **Algorithm**: Is the **Adaboost**, that is an ensemble algorithm.
- **Hypothesis**: The hypothesis function assumes that the previous prediction of the classifier has an impact in the new prediction, as described in **Formula F1**. The previous prediction of the weak classifier is c_j and M is the number of weak classifiers in the loop. C_M is the new prediction.

$$\boldsymbol{Formula\ F1}: C_M(X) = sign(\sum_{j=1}^{M} \alpha_j * c_j(X))$$

The impact of each classifier is given by the **alpha** parameter**.** It uses a log function of the errors.

The alpha function is:

$$\boldsymbol{Formula\ F2}: \alpha_j = \frac{1}{2}\log\frac{1 - Error_j}{Error_j}$$

Where:

j = each weak classifier

When the **weak classifier predicts accurately**, then the **error is near zero** and the **alpha** has a relatively **large positive value**, because of the **log** function. Likewise, when the error is 0.5, then the **alpha is zero**; that means **flipping a coin is the same as using a weak classifier**. When the **weak classifier predicts inexactly**, then the **error is close to one** and the **alpha** has a relatively **large negative value**.

- **Loss**: The loss function computes the errors using the exponential.

  The **error** formula uses the weight of each boosting round:

$$\boldsymbol{Formula\ F3}: Error_j = \frac{1}{N}\sum_{i=1}^{N} w_i * Misclassification_i$$

  The **weight** function is:

$$\boldsymbol{Formula\ F4}:$$

$$w_i^{(j+1)} = \frac{w_i^j}{Z_j} * \begin{cases} e^{-\alpha_j} & if\ Prediction\ of\ base\ classifier(x_i) = y_i \\ e^{\alpha_j} & if\ Prediction\ of\ base\ classifier(x_i)\ != y_i \end{cases}$$

  Where:

  $Misclassification = 1\ if\ Prediction\ of\ base\ classifier(x_i) = y_i$, 0 otherwise.

  Z_i = normalization factor used to assure that summatory of weights is equal to one

  i = each sample

  The **weights of incorrectly classified samples** are **increased**, whereas the **weights of correctly classified** ones are **decreased**, that is because of the **exponential** function.

3. Algorithm

3.1 Technical details

The Adaboost algorithm has two phases, the fitting part to create the model and the prediction part which receive new data and return the predicted labels.

The outstanding features for both sections are:

- Fitting section:
  - Resampling of the data based on the weights
  - Calculation of the alpha and loss values.
- Prediction section:
  - Hypothesis computation.

The pseudocode of the algorithm is:

```
Section FIT (data)

Input: {(xᵢ, yᵢ)} Ɛ data

For all samples i initialize weightsᵢ = 1/n; where n = number of samples
```

Loop for j = 1 : M (where T is the number of weak learners or stumps) do

Compute error using formula F3

Compute alpha using formula F2

Compute weights using formula F4

Section PREDICT (data)

Input: {(x_i)} Ɛ data

Loop for all samples *i*:

Compute prediction using formula F1

3.2 Implementation

The code in the Jupyter notebook has various blocks. Inside the Block 1 there is the implementation of Adaboost binary classifier class, which has the following processes:

- Initialization of the weights in line 66 of the code.

```
65        # Initialize weights to 1/N
66        weight_loop = np.full(n_samples, (1 / n_samples))
```

Figure 1 Block 1 weights initialization

- Resampling of the data by probability based on the weights in line 76, inside the loop for each weak learner in line 74.

```
74        for idx in range(self.n_stumpLearners):
75            # Resample the data
76            resample_ind = sorted(rng.choice(ind, size=(n_samples,), replace=True, p=weight_loop))
77            # Using the indexes facilitates the matching of the samples with the original labels
78            # This helps to maintain consistency among the data
79            x_resampled = X[resample_ind]
80            y_resampled = y[resample_ind]
```

Figure 2 Block 1 resampling

- Activation of stump or weak learners using a **DecisionTree** structure with **depth one** in line 87, making weak predictions in line 92, and marking the misclassifications comparing the predicted values of the weak learner against the true labels from the data in line 95.

```
87        decisionStumpTree = DecisionTreeClassifier(criterion="entropy",max_depth=1)
88        modelDecisionStump = decisionStumpTree.fit(x_resampled,y_resampled)
89
90        # The modelDecisionStump is our weak classifier that will predict the labels of the resampled data set
91        models_loop.append(modelDecisionStump)
92        predictions = modelDecisionStump.predict(x_resampled)
93
94        # Evaluate the classifications and filter the wrong ones
95        misclassified = np.where(predictions != y_resampled,1,0)
```

Figure 3 Block 1 activation of weak learners

- Computation of **error** in line 98 according to **Formula F3**, **alpha** in line 103 according to **Formula F2**, and **weights** in line 107 and 108 according to **Formula F4**.

```
97          # This section computes the minimum error which uses the weight of the sample and the number of misclassified samples
98          min_err = np.sum(weight_loop*misclassified)/np.sum(weight_loop)
99
100         # Calculate alpha using a verySmallNumber to prevent errors if min_error = zero or min_error = 1
101         # Alpha is the significance of the tree, according to the algorithm it is computed as a 1/2*log([1-min_error]/min_error)
102         tinyNumber = 1e-10
103         alpha = 0.5 * np.log((1.0 - min_err + tinyNumber) / (min_err + tinyNumber))
104         alphas_loop.append(alpha)
105
106         # Update the weights which are going to be used for the training of the next decision stump
107         weight_loop *= np.exp(alpha * np.where(misclassified == 1, 1, -1))
108         weight_loop /= np.sum(weight_loop)
109         weights_loop.append(weight_loop)
```

Figure 4 Block 1 computation of error, alpha and weights

- Finally, the **predictions** in the same block are calculated in line 143 inside a loop where the alpha is giving the significance of the tree in the final classification, according to **Formula F1.**

```
141         # Each decision stump weak learner is multiplied by the corresponding Alpha value
142         for dsl_alpha, dsl_model in zip(self.alphas,self.models):
143             prediction = dsl_alpha * dsl_model.predict(X)
144             predictions.append(prediction)
```

Figure 5 Block 1 calculation of predictions

4. Model evaluation

The data set was split in train and test groups. The training set has the 80% of the samples and the test set has the 20%. Line 6 from Block 5 has the implementation.

```
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2)
```

Figure 6 Block 5 split data

4.1 Data preparation

It was looked for missing values, different scales, and types of data. Also, it was plotted the distribution of the target values.

The data set has no missing values, the data set was re-scaled, and all the features are numerical. The target values follow a non-linear distribution. The implementation is in the Blocks 6, 7, 8 and 9 of the code.

4.2 Experiment, design, and evaluation strategy

The experiment has consisted of a fake data set to follow up the algorithm line by line and when it was ready it was used another data set to run the algorithm. It was tested the resampling, calculation of error, alpha, weights, and predictions. Finally, it was used the XGBoost algorithm to compare the predictive power of the developed model.

The design was to determine the target distribution, the source of data, then establish the training process and select the appropriate function to learn and finally to choose the learning algorithm.

The evaluation strategy was to select metrics to measure the loss and the accuracy of the learning process.

4.3 Evaluation results

The metrics used were accuracy, measures of false positives, false negatives, true positives and true negatives, and F1-score. Block 13 lines 19 to 23 on Figure 7, Block 16, and Block 17 on Figure 8 have those implementations. It is observed that both algorithms reach over 80% of accuracy. However, being a prediction of a disease is controversial to have false negatives and false positives as well. In this case, there are 19 false positive cases, those numbers represent people, and the ML model is suggesting they are healthy when is not true.

Adaboost Scratch	Accuracy training set	85.93%
Adaboost Scratch	Accuracy testing set	83.33%
Adaboost Scratch	F1-Score testing set	87.90%
XGBoost	Accuracy testing set	93.86%
XGBoost	F1-Score testing set	94.96%

```
19  acc_train = accuracy(y_train, y_pred_train)
20  accuracy_train_list.append(acc_train)
21  y_pred_test = ada_clf_model.predict(X_scaled_features_test)
22  acc_test = accuracy(y_test, y_pred_test)
23  accuracy_test_list.append(acc_test)
```

Figure 7 Block 13 implementation Accuracy

```
# Block 16: Model Evaluation
# The confusion matrix says how many cases are true_positive, true_negative, false_positive and false_negative.
# It compares the ground truth against the outcomes of the model
# In this case the confusion matrix states the comparison using the TEST DATA SET:
# 26 True Positive cases: the model predicted they were malign, and the target was really malign.
# 00 False Positive cases: the model predicted they were malign, but the target said they were benign.
# 19 False Negative cases: the model predicted they were benign, but the target said they were malign.
# 69 True Negative cases: the model predicted they were benign, and the target was really benign.
confusion_matrix(y_test, y_pred_test)
```

```
array([[26, 19],
       [ 0, 69]])
```

```
# Block 17: Model Evaluation
# To compute the F1 score. This is another metric to evaluate the model.
# F1 score is the harmonic mean of precision and recall.
# Precision measures how many true cases were predicted against the false positive cases (the model said it was malign but it wasn't).
# Recall measures how many true cases were predicted against the false negative cases (the model said it was benign but it wasn't ).
from sklearn.metrics import f1_score
f1_score_test = f1_score(y_test, y_pred_test)
print(f'F1-score test set: {f1_score_test:.2%}.')
```

```
F1-score test set: 87.90%.
```

Figure 8 Block 16: confusion matrix Block 17: F1-Score

Additionally, it was used two plots, ROC curve that is representing a normal learning with 80% of AUC, and a comparison between the testing and training set accuracy which also shows a normal learning without underfitting or overfitting.

It was measured the processing time, both algorithms Adaboost scratch and XGBoost are in the order of milliseconds. Times are summarised in Figure 9 and Figure 10.

```
Adaboost Scratch Time between times: 0 days, 0 hours, 0 minutes and 0 seconds
start 2020-10-10 11:59:45.687236, end 2020-10-10 11:59:45.912271
```

Figure 9 Adaboost scratch processing time (100 weak learners)

```
XGBoost Time between times: 0 days, 0 hours, 0 minutes and 0 seconds
start 2020-10-10 11:59:47.276153, end 2020-10-10 11:59:47.370713
```

Figure 10 XGBoost processing time (100 weak learners)

5. Conclusion

ML process is based on prior knowledge and it is approximately correct. It is critical to have a well-defined task, a performance metric and data to train; thus, the ML algorithm will determine the best fit to the data from the space of possible hypothesis.

The benefits of Adaboost are its predictive power and the performance. In this case study the results were shown in the Evaluation section.

The drawback is trees algorithms are prone to overfit sooner. And generally, noise in the label data, such as mismatch or labelling errors cause bad predictions.

The implemented Adaboost algorithm only works on binary classifications. This is a huge difference compared with ML libraries as XGBoost. Packages available on scikit-learn provide with wide functionality, such as multiclass classification, regression tasks, parameters to tune and optimise the performance of the model. This implementation has a wide room to improve, here the main ones:

- Add regularization functions to measure the complexity of the model and tune it as a parameter.
- It can be used the GINI index based on the weight to select the best split to build the tree.
- The algorithm could split the training data into training and validation; thus the model ought to finish when the validation reaches a level of no more improvement in the accuracy and the error instead of going down starts to going up.
- Modify the interpretation of the targets so it can learn to predict more than two labels.
- Deal with missing values, one way could be giving the option to calculate the average, or to fill forward or backward with the nearest known value.
- Recode the verbosity of the class to give more understanding of what the model is doing.
- Changing the number of samples and features produces differences in the performance of the algorithm. ML libraries present a more stable behaviour in front of different type of data sets.
- Change the loss function with regression tasks to RMSE (root mean squared error).