Forest Cover Type Classification Using AdaBoost

UNIVERSITÀ DEGLI STUDI DI MILANO

Statistical Methods for Machine Learning Course
Experimental project May 2022

Mahdi Sotikhiabani

Abstract

Ensemble methods are a way of combining the predictions of multiple classifiers to get a better answer than simply using one classifier. Combining multiple classifiers can also solve the problems of single classifiers, such as overfitting. We used the popular variant of boosting, called AdaBoost. AdaBoost uses a bunch of weak learners as the base classifier with the input data weighted by a weight vector. We use functions to create a classifier using AdaBoost and the weak learner, decision stumps(which is a single-node tree). This decision stump will later feed our Adaboost Algorithm. The dataset that we are going to apply all of this is “Forest Cover Type Dataset” where each row includes variables like tree type, shadow coverage, distance to nearby landmarks, soil type, local topography, etc. What we will try to do is to predict the cover type of the forest based among 7 different possible classes. The cover type labels have 7 classes, to address this multi-class classification problem, we use a one-vs-all-encoding method where we train seven binary classifiers, one for each of the seven classes. Next, we use a 5-fold cross validation to evaluate the multiclass classification performance for different values of the Adaboost iterations. After evaluating different values for Adaboost rounds 50,150,250,350, and 450, since we don’t see much difference in terms of performance, and considering the computational power, reasonability, and scalability of the solution, we tune our final model with 150 Adaboost rounds and achieve around 71.3% test and 72.7% train accuracy.

Functions needed to implement the full Adaboost Algorithm

In the following, the functions used to implement the Algorithm will be introduced:

```
[2]  #threshold comparison to classify data

     def stump_classify(data_matrix, dim, splitvalue, splitineq):
         filter_array = np.ones((np.shape(data_matrix)[0],1))
         if splitineq == 'below':
            filter_array[data_matrix[:,dim] <= splitvalue] = -1
         else:
            filter_array[data_matrix[:,dim] > splitvalue] = -1
         return filter_array
```

- The first function is the stump_classify(). This function tests if any of the values are below or above the threshold value, and will put everything on one side of the threshold into class -1 and everything else +1.

```
[3]  #finding the best decision stump for our dataset

     def decision_stump(data_array, labelset, D):
         data_matrix = np.mat(data_array)
         labelset = np.mat(labelset).T
         row,col_len = np.shape(data_matrix)
         iterations = 25
         stumps = {}
         best_class = np.ones((row,1))
         min_error = np.inf
         for i in range(col_len):
             minvalue = data_matrix[:,i].min()
             maxvalue = data_matrix[:,i].max()
             step_size = (maxvalue - minvalue)/iterations
             for j in range(0, iterations):
                 for inequal in ['below', 'above']:
                     splitvalue = (minvalue + j * step_size)
                     prediction_values = stump_classify(data_matrix,i,splitvalue,inequal)
                     error_array = np.ones((row,1))
                     error_array[prediction_values == labelset] = 0
                     weighted_error = D.T * error_array

                     if weighted_error < min_error:
                        min_error = weighted_error
                        best_class = prediction_values.copy()
                        stumps['dimension'] = i
                        stumps['split'] = splitvalue
                        stumps['ineq'] = inequal

         return stumps,min_error,best_class
```

- Next function decision_stump() iterates over all the possible inputs to the previous function stump_classify() and finds the best decision stump for the dataset. By best we mean best with respect to the weight vector D. We create an empty dictionary to store the classifier information corresponding to the best choice of the decision stump given the weight vector D. The three nested loops start with going over all the features in the dataset to get the minimum and maximum values to calculate how large the step size should be. The second for loops over these values and the last one switches the inequality between below and above the threshold. The weighted error is the part that will later interact with the Adaboost algorithm. We are basically evaluating the classifier based on the weight vector D.

- To summarize, the decision stump function takes the data, the labelset, and weight vector D(probability distribution) and returns the dictionary composed of the classifier information. The decision stump we built is somehow the simplified version of a decision tree(one level decision tree) that’s why we call it the weak classifier.

```
#based on the decision_stump we build, we start implementing the full AdaBoost

def adaboost_training(data_array,labelset,n_iterations= 40):
    weak_learners = []
    row_len = np.shape(data_array)[0]
    D = np.mat(np.ones((row_len,1))/row_len)
    agg_class_est = np.mat(np.zeros((row_len,1)))
    for i in range(n_iterations):
        stumps,error,class_est = decision_stump(data_array, labelset, D)
        alpha = float(0.5*np.log((1.0-error)/max(error,1e-16)))
        stumps['alpha'] = alpha
        weak_learners.append(stumps)
        expon = np.multiply(-1*alpha*np.mat(labelset).T,class_est)
        D = np.multiply(D,np.exp(expon))
        D = D/D.sum()
        agg_class_est += alpha*class_est

        aggErrors = np.multiply(np.sign(agg_class_est) !=
                    np.mat(labelset).T,np.ones((row_len,1)))
        errorRate = aggErrors.sum()/row_len


        if errorRate == 0.0: break
    return weak_learners,agg_class_est
```

- The adaboost_training() takes the dataset, the labelset, and the one and only parameter of the algorithm, the number of iterations which needs to be specified by the user. We want to build a classifier that could make decisions based on the weighted input value. The algorithm will output an array of weak learners(decision stumps in our case). We first create a new Python list to store these values. Next get the length of the rows, the number of data points in our dataset, and create a column vector, D. The vector D is important. It holds the weight of each piece of data. These weights are all equal at first but later on(subsequent iterations) the adaboost algorithm will adjust the weights by assigning higher weights to each misclassified instance and lower weights to correctly classified examples.The final output of the algorithm is an array containing three dictionaries, which contains all the information we need for classification.

```
[13] #we have an array of weak leaner + alpha value for each classifier. now we just need to take the train of weak classifiers from the training function and
     #apply these to an instance

     def adaboost_classifier(classification_data,classifier_array):
         data_matrix = np.mat(classification_data)
         row_len = np.shape(data_matrix)[0]
         agg_class_est = np.mat(np.zeros((row_len,1)))
         for i in range(len(classifier_array)):
             class_est = stump_classify(data_matrix,classifier_array[i]['dimension'],\
                                 classifier_array[i]['split'],\
                                 classifier_array[i]['ineq'])
             agg_class_est += classifier_array[i]['alpha']*class_est

         return (agg_class_est)
```

- Now that we have everything set, we take the train of weak classifiers from the training function and apply it to an instance. The result of each classifier is measured by the alpha values. The weighted results from all of these classifiers are added together. We take the sign of the final weighted sum to reach our final predictions.

```
#test_train split based on the required parameters.

def split_train_test(df,test_size,label_col,random_state=50):

    random.seed(random_state)


    label_col = str(label_col)
    dat_len  = len(df)
    X= df.drop(columns=label_col)
    Y = df[label_col]



    if isinstance(test_size,float):
        test_size = round(test_size*dat_len)

    indices = list(data.index)
    test_indices = random.sample(population=indices, k=test_size)

    X_train = X.drop(test_indices)
    x_test  = X.loc[test_indices]

    Y_train= Y.drop(test_indices)
    y_test = Y.loc[test_indices]


    return X_train.sample(frac=1,random_state=random_state),x_test,Y_train.sample(frac=1,random_state=random_state),y_test
```

- split_train_test is for splitting the dataset to train and test part. The inputs are the test size and the label set and it returns randomly selected train and test data.

```
#since we have 7 classes in the dataset, we need to extent our solution to multi class. this function implements multi-class
#external cross-validation.

def multi_class_adaboost(data,k_folds,T_rounds,labelset,random_state):

    random.seed(random_state)

    X_train,x_test,Y_train,y_test = split_train_test(data,test_size=0.2,label_col=labelset,random_state=random_state)
    indices = np.array_split(list(X_train.index),k_folds)

    Cv_test_acc = np.mat(np.ones(shape=(len(T_rounds),k_folds)))
    Cv_train_acc = np.mat(np.ones(shape=(len(T_rounds),k_folds)))

    for T in T_rounds:
        for k in range(k_folds):
            train_preds= np.mat(np.ones(shape=(np.shape(X_train.drop(indices[k]))[0],len(np.unique(data[labelset])))))
            test_preds = np.mat(np.ones(shape=(np.shape(indices[k])[0],len(np.unique(data[labelset])))))
            for classes in range(len(np.unique(data[labelset]))):
                model,pred = adaboost_training(X_train.drop(indices[k]),np.where(Y_train.drop(indices[k])==classes+1,1,-1),T)
                train_preds[:,classes]=np.multiply(train_preds[:,classes],pred)
                test_est = adaboost_classifier(X_train.loc[indices[k]],model)
                test_preds[:,classes] = np.multiply(test_preds[:,classes],test_est)

            train_predictions = np.argmax(train_preds,axis=1)+1
            training_error =np.where(train_predictions!=np.mat(Y_train.drop(indices[k])).T,1,0).sum()
            Cv_train_acc[T_rounds.index(T),k]=1-(training_error/len(train_predictions))
            test_prediction = np.argmax(test_preds,axis=1)+1
            test_error = np.where(test_prediction!=np.mat(Y_train.loc[indices[k]]).T,1,0).sum()
            Cv_test_acc[T_rounds.index(T),k]=1-(test_error/len(test_prediction))
        print( 'Cv for',T,'rounds is completed')

    return(Cv_train_acc,Cv_test_acc)
```

- Since we have 7 classes in our dataset, with this function we extend our solution to multi class classification and we evaluate the performance of our solution with 5-fold cross-validation for different values of the number of adaboost rounds(50,150,250,350,450).

Using Adaboost on a Forest Cover Type Dataset

```
[16] data = pd.read_csv('covtype.csv')
     data = data.sample(frac = 0.020)
     forest = data.copy()
```

- We start by importing the dataset. We take 20% of the dataset randomly to implement Adaboost and do our analysis. The reason we take a smaller portion of the dataset is to make our analysis easier in terms of the computation time.

```
forest.info()
```

```
---  ------                              --------------  -----
 0   Elevation                           11620 non-null  int64
 1   Aspect                              11620 non-null  int64
 2   Slope                               11620 non-null  int64
 3   Horizontal_Distance_To_Hydrology    11620 non-null  int64
 4   Vertical_Distance_To_Hydrology      11620 non-null  int64
 5   Horizontal_Distance_To_Roadways     11620 non-null  int64
 6   Hillshade_9am                       11620 non-null  int64
 7   Hillshade_Noon                      11620 non-null  int64
 8   Hillshade_3pm                       11620 non-null  int64
 9   Horizontal_Distance_To_Fire_Points  11620 non-null  int64
 10  Wilderness_Area1                    11620 non-null  int64
 11  Wilderness_Area2                    11620 non-null  int64
 12  Wilderness_Area3                    11620 non-null  int64
 13  Wilderness_Area4                    11620 non-null  int64
 14  Soil_Type1                          11620 non-null  int64
 15  Soil_Type2                          11620 non-null  int64
 16  Soil_Type3                          11620 non-null  int64
 17  Soil_Type4                          11620 non-null  int64
 18  Soil_Type5                          11620 non-null  int64
 19  Soil_Type6                          11620 non-null  int64
 20  Soil_Type7                          11620 non-null  int64
 21  Soil_Type8                          11620 non-null  int64
 22  Soil_Type9                          11620 non-null  int64
 23  Soil_Type10                         11620 non-null  int64
 24  Soil_Type11                         11620 non-null  int64
 25  Soil_Type12                         11620 non-null  int64
 26  Soil_Type13                         11620 non-null  int64
 27  Soil_Type14                         11620 non-null  int64
 28  Soil_Type15                         11620 non-null  int64
 29  Soil_Type16                         11620 non-null  int64
 30  Soil_Type17                         11620 non-null  int64
 31  Soil_Type18                         11620 non-null  int64
 32  Soil_Type19                         11620 non-null  int64
 33  Soil_Type20                         11620 non-null  int64
 34  Soil_Type21                         11620 non-null  int64
 35  Soil_Type22                         11620 non-null  int64
 36  Soil_Type23                         11620 non-null  int64
 37  Soil_Type24                         11620 non-null  int64
 38  Soil_Type25                         11620 non-null  int64
 39  Soil_Type26                         11620 non-null  int64
 40  Soil_Type27                         11620 non-null  int64
 41  Soil_Type28                         11620 non-null  int64
 42  Soil_Type29                         11620 non-null  int64
 43  Soil_Type30                         11620 non-null  int64
 44  Soil_Type31                         11620 non-null  int64
 45  Soil_Type32                         11620 non-null  int64
 46  Soil_Type33                         11620 non-null  int64
 47  Soil_Type34                         11620 non-null  int64
 48  Soil_Type35                         11620 non-null  int64
 49  Soil_Type36                         11620 non-null  int64
 50  Soil_Type37                         11620 non-null  int64
 51  Soil_Type38                         11620 non-null  int64
 52  Soil_Type39                         11620 non-null  int64
 53  Soil_Type40                         11620 non-null  int64
 54  Cover_Type                          11620 non-null  int64
dtypes: int64(55)
memory usage: 5.0 MB
```

- Our dataset contains 11620 rows of data and 55 columns. We have 7 different classes of output(cover-type) and we are trying to predict the final class based on the below variables.

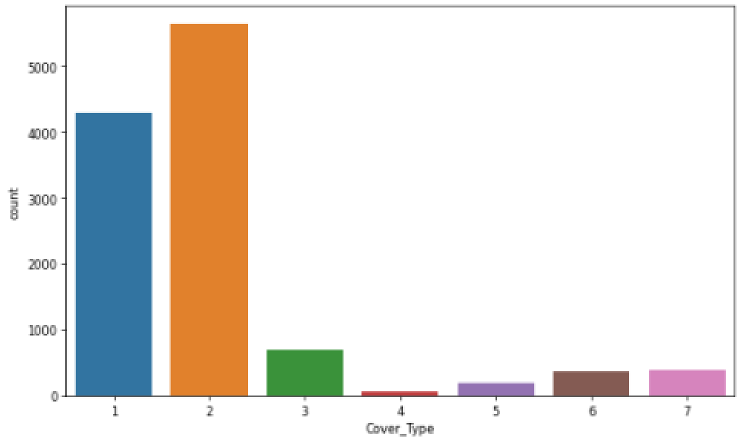
```python
[20] forest.isna().sum()
```

```
Elevation                             0
Aspect                                0
Slope                                 0
Horizontal_Distance_To_Hydrology      0
Vertical_Distance_To_Hydrology        0
Horizontal_Distance_To_Roadways       0
Hillshade_9am                         0
Hillshade_Noon                        0
Hillshade_3pm                         0
Horizontal_Distance_To_Fire_Points    0
Wilderness_Area1                      0
Wilderness_Area2                      0
Wilderness_Area3                      0
Wilderness_Area4                      0
Soil_Type1                            0
Soil_Type2                            0
Soil_Type3                            0
Soil_Type4                            0
Soil_Type5                            0
Soil_Type6                            0
Soil_Type7                            0
Soil_Type8                            0
Soil_Type9                            0
Soil_Type10                           0
Soil_Type11                           0
Soil_Type12                           0
Soil_Type13                           0
Soil_Type14                           0
Soil_Type15                           0
Soil_Type16                           0
Soil_Type17                           0
Soil_Type18                           0
Soil_Type19                           0
Soil_Type20                           0
Soil_Type21                           0
Soil_Type22                           0
Soil_Type23                           0
Soil_Type24                           0
Soil_Type25                           0
Soil_Type26                           0
Soil_Type27                           0
Soil_Type28                           0
Soil_Type29                           0
Soil_Type30                           0
Soil_Type31                           0
Soil_Type32                           0
Soil_Type33                           0
Soil_Type34                           0
Soil_Type35                           0
Soil_Type36                           0
Soil_Type37                           0
Soil_Type38                           0
Soil_Type39                           0
Soil_Type40                           0
Cover_Type                            0
dtype: int64
```

- Here we check whether there exists a null value in our dataset.

```
[19] figure(figsize=(10, 6), dpi=60)
     sns.countplot(x="Cover_Type", data=forest)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f51a4514790>
```



- Our labels are not equal to each other. We have an unbalanced dataset but this won't affect our analysis since we are already using one vs rest predictor for our classification problem.

```
[21] cv_train,cv_test = multi_class_adaboost(forest,k_folds=5,T_rounds=[50,150,250,350,450],
                                             labelset='Cover_Type',random_state=50)

Cv for 50 rounds is completed
Cv for 150 rounds is completed
Cv for 250 rounds is completed
Cv for 350 rounds is completed
Cv for 450 rounds is completed
```

- Now we use the multi-class-adaboost function to see the difference between different rounds of adaboost and decide the final number of T_rounds which is the only parameter we need for finalizing our prediction.

```
[14] np.mean(cv_test,axis=1)

matrix([[0.70718475],
        [0.71417718],
        [0.71428471],
        [0.71471528],
        [0.71568348]])
```

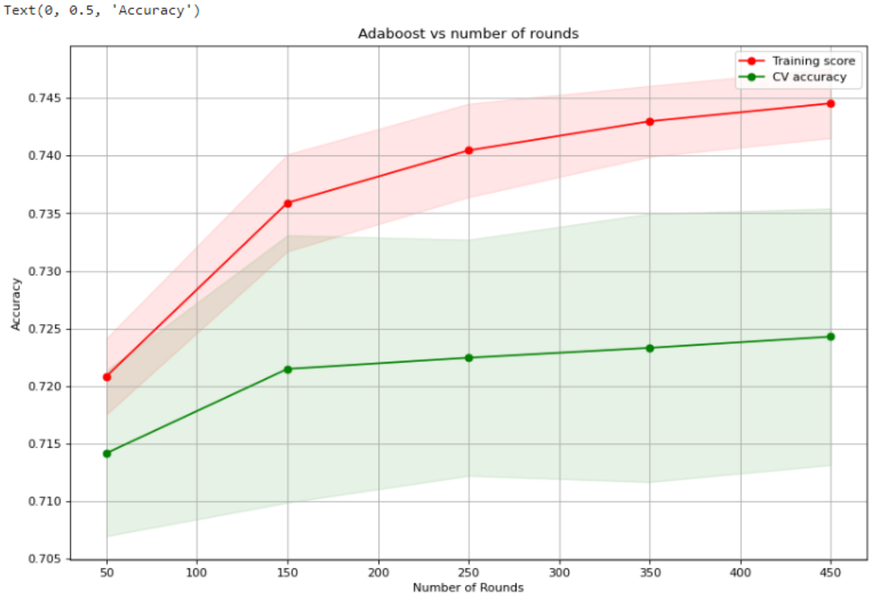
```
[15] np.mean(cv_train,axis=1)

matrix([[0.7156572 ],
        [0.72870045],
        [0.73308404],
        [0.73585404],
        [0.73819376]])
```

- This is the accuracy we achieved for 5 different values of adaboost rounds that we tried (50,150,250,350,450), as we can see there is not much difference in terms of improvement in the accuracy as we are increasing the number of rounds. Considering the time it takes to run the code, we tune the final model with 150 rounds where we achieve a reasonable test and train accuracy.

```
[28] train_score_mean = np.mean(np.array(cv_train),axis=1)
     train_score_std = np.std(np.array(cv_train),axis=1)
     test_score_mean = np.mean(np.array(cv_test),axis=1)
     test_score_std = np.std(np.array(cv_test),axis=1)
     T_rounds = np.array([50,150,250,350,450])
     plt.figure(figsize=(12, 8), dpi=80)
     plt.title('Adaboost vs number of rounds')
     plt.grid()
     plt.fill_between(T_rounds, train_score_mean - train_score_std,
                      train_score_mean + train_score_std, alpha=0.1,
                      color="r")
     plt.fill_between(T_rounds, test_score_mean - test_score_std,
                      test_score_mean + test_score_std, alpha=0.1, color="g")
     plt.plot(T_rounds, train_score_mean, 'o-', color="r",
              label="Training score")
     plt.plot(T_rounds, test_score_mean, 'o-', color="g",
              label="CV accuracy")
     plt.legend()
     plt.xlabel('Number of Rounds')
     plt.ylabel('Accuracy')
```

```
Text(0, 0.5, 'Accuracy')
```



- This plot shows how the accuracy improves with respect to the number of iterations.

```
[16] adaboost_round = 150

     X_train,x_test,Y_train,y_test = split_train_test(data,test_size=0.2,label_col='Cover_Type',random_state=50)
     train_preds= np.mat(np.ones(shape=(np.shape(X_train)[0],len(np.unique(data['Cover_Type'])))))
     test_preds =  np.mat(np.ones(shape=(np.shape(x_test)[0],len(np.unique(data['Cover_Type'])))))
     for classes in range(len(np.unique(data['Cover_Type']))): # one VS Rest classifier
         model,pred = adaboost_training(X_train,np.mat(np.where(Y_train==int(classes)+1,1,-1)),adaboost_round)
         train_preds[:,classes]=np.multiply(train_preds[:,classes],pred)
         test_est = adaboost_classifier(x_test,model)
         test_preds[:,classes]=np.multiply(test_preds[:,classes],test_est)

     train_predictions = np.argmax(train_preds,axis=1)+1 # because indexing starts from 0
     training_error = np.where(train_predictions != np.mat(Y_train).T,1,0).sum()
     test_prediction = np.argmax(test_preds,axis=1)+1
     test_error = np.where(test_prediction != np.mat(y_test).T,1,0).sum()
     print('test accuracy:',1-(test_error/len(x_test)))
     print('training accuracy:',1-(training_error/len(X_train)))
```

```
test accuracy: 0.713855421686747
training accuracy: 0.7270869191049913
```

- This is our final results in terms of training and test accuracy tuned with 150 numbers of T_rounds.

```
[27] model # decision_stumps
```


```
[{'alpha': 1.6834253960754564,
  'dimension': 0,
  'ineq': 'below',
  'split': 3659.6000000000004},
 {'alpha': 1.1304050922181703,
  'dimension': 0,
  'ineq': 'below',
  'split': 3196.4},
 {'alpha': 0.6553514319139033,
  'dimension': 0,
  'ineq': 'below',
  'split': 3350.8},
 {'alpha': 0.29629809749152025,
  'dimension': 3,
  'ineq': 'above',
  'split': 106.24},
 {'alpha': 0.31134438479423915,
  'dimension': 0,
  'ineq': 'below',
  'split': 3119.2},
 {'alpha': 0.35595545848532095,
  'dimension': 52,
  'ineq': 'below',
  'split': 0.0},
 {'alpha': 0.2316672748318274,
  'dimension': 1,
  'ineq': 'above',
  'split': 201.6},
 {'alpha': 0.26583195547542376,
  'dimension': 12,
  'ineq': 'below',
  'split': 0.0},
 {'alpha': 0.1906404750243307,
  'dimension': 9,
  'ineq': 'below',
  'split': 1994.1599999999999},
 {'alpha': 0.20837572882711178,
  'dimension': 0,
  'ineq': 'below',
  'split': 3273.6000000000004},
 {'alpha': 0.22100912858359145,
  'dimension': 4,
  'ineq': 'above',
  'split': 9.800000000000011},
 {'alpha': 0.16473177630468655,
  'dimension': 0,
  'ineq': 'below',
  'split': 3119.2},
 {'alpha': 0.207056628659189, 'dimension': 51, 'ineq': 'below', 'split': 0.0},
 {'alpha': 0.14315763399616266,
  'dimension': 11,
  'ineq': 'above',
  'split': 0.0},
 {'alpha': 0.14425588157886288,
  'dimension': 42,
  'ineq': 'below',
  'split': 0.0},
```


- This is the list containing dictionaries we discussed before. The dictionaries presented in the list contain all the important values we used. Alpha value, the dimension, “below or above” and the split value

Conclusion

We started by creating our weak learner, the decision stump. This Decision-Stump provided adaboost the best matrix having the smallest error according to the probability distribution, and after adaboost calculates alpha values using this smallest error, (epsilon). Adaboost then adjusts the probability distribution by using the best estimated matrix from Decision Stump and the alpha value to feed the Decision Stump once again. After defining the fundamental functions of Adaboost we had to address the problem of multi-class classification since our solution was originally for binary classification. We used a one-vs-all-encoding method for multi-class classification and a 5-fold cross validation to evaluate the multi-class classification performance for different values of the Adaboost iterations(50,150,250,350,450). Finally after observing the results for different rounds, since we didn’t observe a big difference between the different values that we tested, we built the final model with only 150 Adaboost rounds and achieved 71.3% test and 72.7% train accuracy.

Acknowledgment

- I used chapter 7 ”Improving classification with the AdaBoost meta-algorithm” of the book “Machine Learning in Action” authored by Peter Harrington to implement the Adaboost algorithm from scratch.