UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Politiche, Economiche e Sociali
Master's degree in Data Science and Economics

Forest cover type classification using AdaBoost

Gabriele Fusar Bassini - ID 974407

A. Y. 2021-2022

Abstract

This paper implements AdaBoost, an incremental ensemble method that builds classifiers from decision stumps, from scratch, using nothing but NumPy and pandas. The ensemble method is used to classify forest cover types based on the dataset from the Roosevelt National Forest in Colorado, available on Kaggle. Since AdaBoost is a binary classifier, the alogirthm is run for every type and the classifier selects the prediction that maximizes the binary classifier of the class. To evaluate the multiclass classification performance, we use cross validation with zero-one loss for different values of the number T of AdaBoost rounds.

Contents

1 Theoretical background

The aim of this paper is to implement AdaBoost, a binary classifier, on a seven classes dataset. For how the method works, a naive solution can solve the problem and aggregate the results into one prediction after running the algorithm for all the classes through one-vs-all encoding. The performance of the classifier is then tested using external cross-validation.

1.1 AdaBoost

AdaBoost is an ensemble method that aggregates simple base classifiers into a predictor. Despite other ensemble methods, the base classifiers aren't independent and each one retains some information from the previous epochs of training. Let $Y = \{-1, +1\}$ be the classification target and $(x_1, y_1), ..., (x_m, y_m)$ the training set S with m examples, the final predictor is a classifier of the form $\text{sgn}(f)$, where

$$f = \sum_{i=1}^{T} \omega_i h_i$$

and $\omega = (\omega_1, ..., \omega_T)$ is a vector of real coefficients. The base classifiers $(h_1, ..., h_T)$ are decision stumps, i. e. one-level binary decision trees of the form
$h_{i,\tau} : R^d \to \{-1, 1\}$ defined by $h_{i,\tau}(x) = \pm sgn(x_i - \tau)$ where $i = 1, ..., d$ and $\tau \in \mathbb{R}$.
The threshold τ splits the sample into two groups: more intuitively, one where the feature $x_i < \tau$ and $h_i = -1$ and one where $x_i <= \tau$ and $h_i = +1$. Setting $L_i(t) = h_i(x_t)y_t$, where $L_i(t) = 1$ if and only if $h_i(x_t) = y_t$, the weighted training error of h_i with respect to the probability $\mathbb{P}_i$ is

$$\varepsilon_i = \mathbb{P}_i(L_i = -1) = \sum_{t=1}^{m} \mathbb{I}\{L_i(t) = -1\}\mathbb{P}_i(t)$$

Note that ε_i is always less than or equal to 0.5; if it is not computationally, one needs just to flip h_i classification, meaning that in this case when $x_i < \tau$ then $h_i = 1$ and vice versa.
As stated before, the choice of $h_i^* = \arg\min_{h_i \in H} \varepsilon_i$ is performed for every round in T, where T is the number of total rounds. Each time, a new decision stump is created and a new $\omega_i h_i$ adds to f, where

$$\omega_i = \frac{1}{2} \ln(\frac{1 - \varepsilon_i}{\varepsilon_i})$$

. Since the above expression is only defined for $0 < \varepsilon_i < 1$ and
$\mathbb{P}(t) > 0 \quad \forall \quad t \in 1, ...m$ and $\sum_{t=1}^{m} \mathbb{P}(t) = 1$, we have to take into account the possibility of having a special case $\varepsilon_i \in \{0, 1\}$. That would mean that the classification be perfect (or completely wrong). In this case, the algorithm should stop. As the picture below shows, the absolute value of the weight ω_i increases as the error ε_i moves away from 0.5, i.e. the error decreases before 0.5 or increases if it's bigger than 0.5. This will

end up penalizing the worst predictors h_i and emphasizing the effect of the ones characterized by a smaller error.

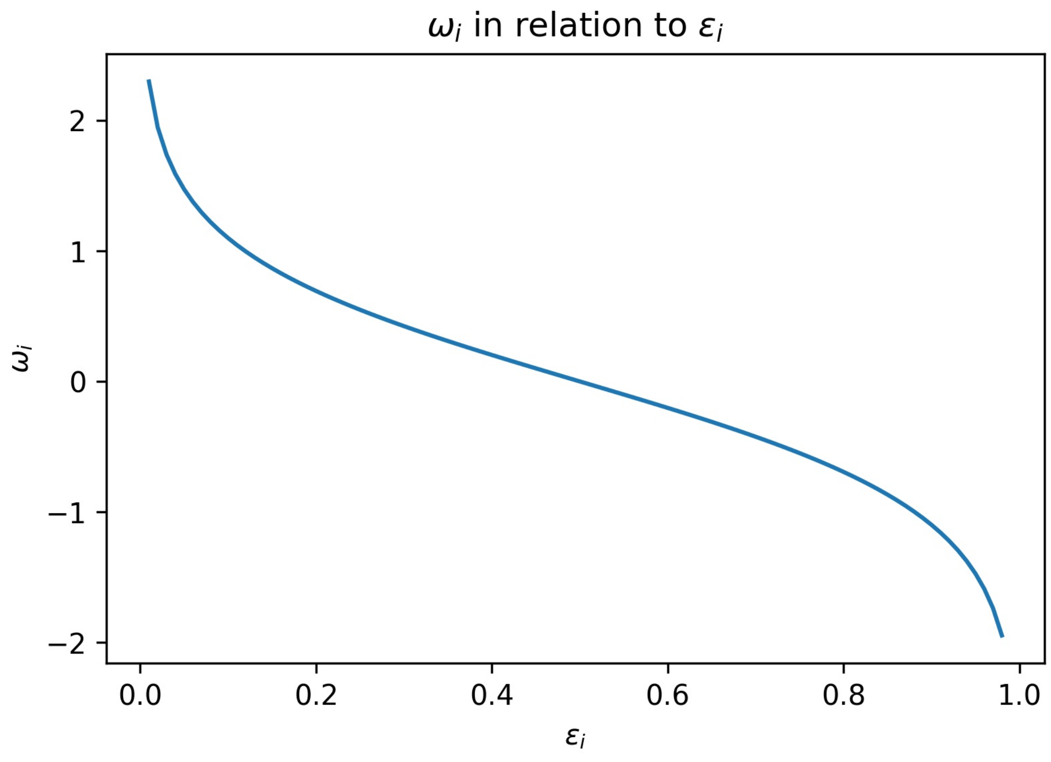

Figure 1: Relation between ω_i and ε_i

Moreover, at every round the probability distribution $\mathbb{P}(t)$ is updated. Initially set as a uniform distribution

$$\mathbb{P}(t) = \frac{1}{m} \quad \forall \quad \mathbb{P}(t) = 1, \ldots, m,$$

they are update every time as

$$\mathbb{P}_{i+1}(t) = \frac{\mathbb{P}_i(t)e^{-\omega_i L_i(t)}}{\mathbb{E}_i[e^{-\omega_i L_i}\ ]} \quad \text{for} \quad t = 1, \ldots, m \quad \text{and} \quad i = 1, \ldots, T-1$$

where

$$\mathbb{E}_i[e^{-\omega_i L_i(t)}\ ] = \sum_{s=1}^{m} e^{-\omega_i L_i}\ \mathbb{P}_i(s)$$

This means that the classifier h_{i+1} retains informations from h_i, and obviously recoursively of the entire preceding process. More precisely, $e^{-\omega_i L_i(t)} > 1$ if and only if $L_i(t) = -1$, that is when the prediction is wrong. Thus, this ensures that the method penalizes the incorrect results by increasing their probability and therefore their weight in building ε_{i+1}. Vice versa, when $L_i(t) = +1$, $\mathbb{P}_i(t)$ is multiplied by a coefficient smaller than one, so it decreases.

1.1.1 Adapting AdaBoost to a multiclass case

In this paper, we use AdaBoost on a multiclass dataset, where forest types are subdivided into seven different clusters. As mentioned before, AdaBoost is usually a binary classifier and its output is $\text{sgn}(f)$. However, f keeps trace, because of how it is computed, of how accurate its prediction is. In fact, since $\lim_{\varepsilon_i \to 0} \omega_i = +\infty$ as Fig.1 attests, there is no bound for f, that increases as the error on h_i decreases. Thus, instead of using $\text{sgn}(f)$ as classifier, we compute the seven classifiers, one for each class, and then select the one that maximizes f. In other words, the forest type will be i that satisfies $argmin_i(f_i)$. In the case of the Roosevelt Forest dataset, $i \in \{1, 7\}$.

1.2 Cross-validation

A crucial aspect of the training phase is to choose a number of trees (hence number of epochs of the algorithm) whose value is as precise as possible during external testing and at the same time avoids overfitting. To find the hyperparameter, as well as to test the reliability of the algorithm, we use external cross-validation. Let S be the entire dataset. We partition S in K folds $D_1, ...D_k$ that we will use as validation sets, or training part. Clearly, the size of each one is m/K. Let then $S^{(k)} = S \backslash D_k$ be the training part. We run the algorithm on each training part $S^{(k)}$. Note that we run the complete algorithm, that includes comparing the seven f_i and finding $argmin_i(f_i)$. We then compute the errors on the testing part of each fold:

$$\hat{\ell}_{D_k}(h_k) = \frac{K}{m} \sum_{(x,y) \in D_k} \ell(y, h_k(x))$$

Then we compute the CV estimate by averaging the errors above:

$$\frac{1}{K} \sum_{k=1}^{K} \ell_{D_k}(h_k)$$

To compute the training error, it is sufficient to repeat the same operation substituting $S^{(k)}$ to the testing part D_k. Cross-validation is a computationally intense operation: the process of building the forest of stumps for every forest type, the choice of i and the testing part are performed for every $k \in K$. That's why in this case we choose $K = 4$.

2 Implementation

In this paper, the implementation of the algorithm lays only on simple Python libraries, namely pandas and NumPy. The entire code takes about an hour to run on an average mid-level consumer computer. It builds several simple functions in order to exploit shortcuts such as list comprehensions and NumPy data structures to save time. Moreover, the code is meant to fit for different datasets, so it carries almost no information - such as the number and the type of the features as well as the number of classes - about the Forest Cover Type Dataset. It includes also a couple of popular libraries - seaborn and Matplotlib - for the sole purpose of plotting data. The library sys allows to stop the process if an exception occours.

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import sys
```

To create different possible thresholds for each feature, a function called indeed **threshold** receives the maximum number of splits *spl* , that in this case will be set to 100 later on, and produces an array of *spl* steps equidistant from each other. It is important to differentiate the process for categorical variabile, that the function divides from the others using the fact that categorical variables have only two possible values. To be as flexible as possible, in this case the only possible spliting value is the mean between the two categories indicator numbers.

```
def threshold(X_feat,spl):
    #check for categorical features
    if len(set(X_feat)) == 2:
        return np.array([sum(set(X_feat)/2])
    else:
        #spl is the number of splits
        step = (min(X_feat)+max(X_feat))/spl
        thr = np.array([round(((index+1) * step),1) for index in
range(spl)])
        return thr
```

Proceeding step by step in building AdaBoost, we write respectively L_i (as an array) and $L_i(t)$ as previously defined in the theoretical section. In the first case, the operations are collapsed into one to speed up the process, assessed that the function **zip** performed slightly better than **enumerate**.

```
def L(X_feat,y,thr):
    return np.array([ -1 * y_t if(X_t < thr) else y_t for X_t,y_t in
zip(X_feat,y) ])
```

```
def L_t(X_t,y_t,thr):
    h_t = -1 if(X_t < thr) else 1
    return h_t * y_t
```

The error ε_i is by definition the sum of P_i where the classification is wrong, i.e. where $L_i(t) = -1$. Once again, while one could use the function **L** written above, the shortened version included in the function **eps** has been proven to be up to twenty times faster. The results converge into the variable *min_eps*. As stressed in the theoretical part, if *min_eps* is equal to one or to zero it means that the classifier is perfect or completely wrong. For the topics we are treating, those case should be more than uncommon. That's why this project has no purpose in investigating the cases more than by just stopping the execution and notify the encountered exception. Moreover, we also proved that there would be no point in *min_eps* being greater than 0.5. In this case, it is sufficient to flip it and update the direction of the classification. In other words, by default **eps** considers the records where the feature value is smaller than the threshold as not belonging to the class and vice versa (hence the classification "direction" *dir_st* set to one). When *min_eps* is greater than 0.5, the classification is turned upside down around the threshold. As a consequence, the error is now the opposite (1 - *min_eps*) and the direction is -1.

```python
def eps(X_feat,y,thr,P):

    #L_i
    h = np.ones(len(y))
    h[X_feat < thr] = -1
    min_eps = np.sum(P[h != y])

    dir_st = 1
    #flip eps and change dir
    if min_eps == 1 or min_eps == 0:
        print("Special case " + str(min_eps))
        sys.exit()
    if min_eps > 0.5:
        min_eps = 1 - min_eps
        dir_st = -1

    return min_eps, dir_st
```

One should then select the threshold that leads to the best results, thus the smallest error. Given the widespread presence of categorical variables (that have only one possibile threshold), the algorithm doesn't perform an initial choice between the threshold belonging to the same feature and directly compares them all in the funcion below. In this case it is is worth to mention that to avoid the use of global variables means also to have a code flexible to changes of hyperparameters and means of validation.

```python
def select_feat(X,y,P,thr):

    data = [(i,index,eps(X[:,i],y,thr[i][index],P)) for i in
range(X.shape[1]) for index in range(thr[i].shape[0])]
    #returns a tuple
    #(best feature number,best eps,best eps value,direction)

    return flatten(min(data, key = lambda t: t[2][0]))
```

Where **flatten** is a simple formatting function defined as:

```
def flatten(data):
    return (data[0],data[1],data[2][0],data[2][1])
```

Eventually, at each round the distribution of probability is updated and the tree composed by feature, threshold, direction and weigth is added to the forest through a dictionary. To be sure that the probability sum is always equal to one, the function **update** simply normalizes it.

```
def update(best_feat, best_eps_ind,best_eps, direction,P,X_train,y,thr):

    w = 1/2 * np.log((1 - best_eps) / best_eps)

    E = np.exp(-1 * w * direction * L(X_train[:,best_feat],y,thr[best_feat]
[best_eps_ind]))
    E = np.array([E[t] * P[t] for t in range(X_train.shape[0])])
    E = np.sum(E)

    P = np.array([(P[t] * np.exp(-1 * w * direction * L_t(X_train
[t,best_feat],y[t],thr[best_feat][best_eps_ind])) / E) for t in
range(X_train.shape[0])])

    #normalize P
    sum_P = np.sum(P)
    P = P / sum_P

    tree = {
      "feature": best_feat,
      "threshold": best_eps_ind,
      "threshold value": best_eps,
      "dir": direction,
      "w": w
    }

    return P, tree
```

The whole operation above can be wrapped into the selection of the feature and subsequent update of P and of the forest with the addiction of a new tree.

```
def add_tree(X_train,y,P,thr,forest):
    best_feat, best_eps_ind,best_eps, direction = select_feat(X_train,y,
P,thr)
    P,tree = update(best_feat, best_eps_ind,best_eps, direction,P,X_train,
y,thr)
```

```
    forest = np.append(forest,tree)
    return forest,P
```

And the operation is performed T times:

```
def build_forest(T,X_train,y,P,thr):
    forest = np.array([])
    for i in range(T):
        forest,P = add_tree(X_train,y,P,thr,forest)
    return forest
```

To sum up, until this point we focused on performing a single binary classification. The nested algorithm creates T trees belonging to the same forest, each one characterized by its own weight in order to compute $f = \sum_{i=1}^{T} \omega_i h_i$ by selecting a certain threshold of a certain feature that minimizes the error $\varepsilon_i = \sum_{t=1}^{m} \mathbb{I}\{L_i(t) = -1\}\mathbb{P}_i(t)$ each time, where $\mathbb{P}_i$ is computed at every epoch in order to enhance the performance as stated before.

We now focus on a broader case in which the classification is not binary. In the case of the Forest Cover Dataset, where there are seven possible categories, we have to deal with two things. The simpler one is to transform the categorical variable y (the type of forest), ranging from one to seven, to a binary one, where +1 means that the record belongs to the class and -1 otherwise:

```
def bool_y(y,tp):
    return np.array([-1 if r!=tp else 1 for r in y])
```

The second problem is to iterate over it, performing the necessary initialization operations, accordingly resetting P. Note that the function **multiforest** passes to **build_forest** the binary array just discussed above.

```
def multiforest(T,X_train,y_train,thr):
    global y_set

    multiforest = []

    for i in y_set:
        P = np.ones(X_train.shape[0])
        P = P / X_train.shape[0]

        this_y = bool_y(y_train,i)
        multiforest.append(build_forest(T,X_train,this_y,P,thr))

    return multiforest
```

The whole code just explained is all we need to build the forest. We now need a

way to evaluate its performance, both inside the training set and the test set. Since we need to tune the hyperparameter T, we will set it to a value greater than the expected in order to be confident to find the best number of epochs. We therefore need to select every stump - taken into account that we create one and only one of them at each round - and add its its classification given by $\omega_i h_i$ to f for each class. In the code, it is in this very moment that the variable *direction* becomes useful. It is important to keep record of f at every epoch in order to compare the results by varying T. The approach in this case is to use a bidimensional matrix and progressively sum the new values where the index is greater or equal to the round number in exam. To save memory as well as to avoid complex data structures, the results and temporary classification are saved into a matrix after every class has been scouted, while the matrix f is soon replaced with the ones of the following one.
After that, the zero-one loss is simple: it is equal to one for every record when the classification is different than the real value, i.e. the difference between them is not zero. The total loss is nothing but the sum of the errors divided by the number of records.

```
def predict(X_k,y_k,thr,multif):
    #predict every entry at each level T
    global T

    len_y = len(y_k)
    ind = np.arange(0,T)

    #initialize f_multif so that it will be immediately replaced
    f_multif = np.ones((T,len_y)).astype(np.float)*(-5)
    final_pred = np.ones((T,len_y))
    loss = np.zeros((T,len_y))

    type_ind = 1
    for i in multif: #types
        f = np.zeros((T,len_y)).astype(np.float)
        for j in range(T): #stumps
            thr_num = i[j]['threshold']
            feat = i[j]['feature']
            direction = i[j]['dir']
            w = i[j]['w']
            this_thr = thr[feat][thr_num]
            h = np.array([-1 if X_k_t < this_thr else 1 for X_k_t in
X_k[feat]])

            f[ind >= j] = f[ind >= j] + w * h * direction

        for j in range(T):
            for jj in range(len_y):
                if f[j][jj] > f_multif[j][jj]:
                    f_multif[j][jj] = f[j][jj]
                    final_pred[j][jj] = type_ind
        type_ind += 1

    total_loss = np.array([np.sum([(fp - y_k) != 0])/len_y for fp in
final_pred])
    return total_loss
```

Eventually, we need to perform the cross-validation, using the parameter K. Since we finally are dealing with the entire dataset, we can use the global X. We split the dataset into $S^{(k)}$ and D_k accordingly to the theoretical part, generate the possible thresholds with the first function we defined and we are ready to create the forest for every fold, perform the classification and find the training error and the test error.

```
#K = total number of folders, X_len = n of samples, small_k = focus folder
def crossval(small_k):
    global X_len
    global K
    global X
    n = X_len / K
    ind = int(small_k*n)
    S_k_1 = X[0:ind,:]
    y_k_1 = y[0:ind]

    ind = int((small_k+1)*n)
    S_k_2 = X[ind:X_len,:]
    y_k_2 = y[ind:X_len]

    S_k = np.concatenate((S_k_1,S_k_2), axis = 0)

    y_k = np.append(y_k_1,y_k_2)

    ind = int((small_k+1)*n)
    D_k = X[int(small_k*n):ind,:]
    y_D_k = y[int(small_k*n):ind]

    thr = np.array([threshold(S_k[:,i],spl) for i in range(S_k.shape[1])],
dtype=object)

    #create the forest for each type
    multif = multiforest(T,S_k,y_k,thr)

    #predict
    tr_S_k = np.transpose(S_k)
    tr_D_k = np.transpose(D_k)
    train_error = predict(tr_S_k,y_k,thr,multif)
    test_error = predict(tr_D_k,y_D_k,thr,multif)

    return train_error, test_error
```

2.1 Data analysis

This dataset contains tree observations from four areas of the Roosevelt National Forest in Colorado. All observations are cartographic variables (no remote sensing) from 30 meter x 30 meter sections of forest. The dataset includes information on tree type, shadow coverage, distance to nearby landmarks (roads etcetera), soil type, and local topography.

The dataset is available at: kaggle.com/uciml/forest-cover-type-dataset.
It counts 15120 records with 54 features; there are altogether seven different types of cover type.

```
dataset = pd.read_csv('Datasets/forest-cover-type.csv')
dataset.shape
dataset.head()
```

Briefly exploring the dataset, we see that it has only ten quantitative variables and the other 44 are categoricals in the form zero/one. The variables are:

- Elevation
- Aspect
- Slope
- Horizontal distance to hydrology
- Vertical distance to hydrology
- Horizontal Distance to roadways
- Hillshade at different times (9am, noon, 3pm)
- Wilderness area (4 types)
- Soil type (40 types)

The first variable is the id and the last one is the target variable, namely cover type. There are no null values, as tested with

```
dataset.isna().sum()
```

And the target variable is perfectly balanced in the sample. This is even more useful in this case because when we transform the values into the binary form to perform the classification for each cover type we are going to end up with a dataset with the target variable equal to -1 six times more than the ones where it is equal to 1.

```
dataset.groupby('Cover_Type').count().iloc[:, 0:1]
```

One may prefer to have a look also visually. In this case, the code below outputs the Fig.2.

```
ax = sns.countplot(x="Cover_Type", data=dataset)
fig = ax.get_figure()
fig.savefig("count.png")
```

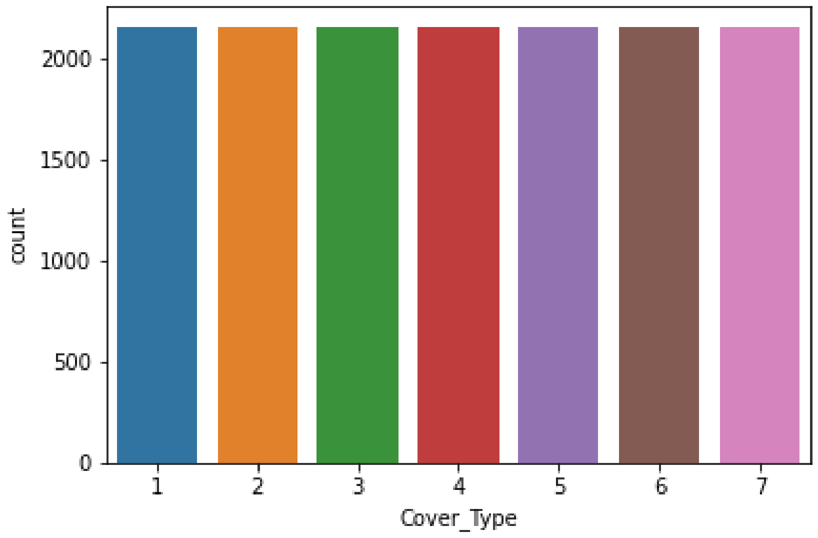

Figure 2: Distribution of the classes among the dataset

2.2 Results

We are finally ready to launch the code. After splitting the dataset into X and y, we proceed setting the parameters $T = 400$ and $K = 4$ (the number of folds for the cross-validation). We also set the maximum number of possible thresholds for every feat to 100.

```
X = dataset.drop(columns=['Id', 'Cover_Type']).values
y = dataset['Cover_Type'].values

y_set = set(y)
X_len = X.shape[0]

T = 400
K = 4
#number of possible splits
spl = 100
```

```
train_error = []
test_error = []

for i in range(K):
    train_error.append([])
    test_error.append([])

for i in range(K):
    tr_error, ts_error = crossval(i)
    train_error[i] = np.append(train_error[i],tr_error)
    test_error[i] = np.append(test_error[i],ts_error)

train_error = np.transpose(train_error)
test_error = np.transpose(test_error)
```

```
av_train_error = [np.mean(train_error[i]) for i in range(T)]
std_train_error = [np.std(train_error[i]) for i in range(T)]
av_test_error = [np.mean(test_error[i]) for i in range(T)]
std_test_error = [np.std(test_error[i]) for i in range(T)]

#plot
step = 40
T_ar = np.arange(T)

train_column = [str(np.round(av_train_error[st],5))+"±"+
str(np.round(std_train_error[st],5)) for st in T_ar[step::step]]
test_column = [str(np.round(av_test_error[st],5))+"±"+
str(np.round(std_test_error[st],5)) for st in T_ar[step::step]]

d = {'Training error': train_column, 'Test error': test_column}
error_df = pd.DataFrame(data=d,index=T_ar[step::step])
print(np.round(av_train_error[step::step],5))
print(np.round(std_train_error[step::step],5))
error_df
```

As the following table shows, the test error stabilizes around 30% after 300 rounds. As expected, the training error is smaller and keeps decreasing in the following rounds. The result is emphasized by Fig.3.

T	Training error	Test error
40	0.27782±0.00466	0.32804±0.04186
80	0.26422±0.0059	0.31567±0.03008
120	0.25714±0.00436	0.31144±0.02636
160	0.2506±0.00479	0.31197±0.02816
200	0.24658±0.00628	0.30787±0.02775
240	0.24427±0.00602	0.30979±0.02968
280	0.24224±0.00527	0.30886±0.02879
320	0.24094±0.00513	0.30675±0.02745
360	0.23785±0.00513	0.30734±0.02854
400	0.23618±0.00495	0.30681±0.02779

```
plt.title('AdaBoost Accuracy')
plt.errorbar(T_ar[step::step],av_train_error[step::step],
std_test_error[step::step],color='#fa4',label="Train error",marker='o')

plt.errorbar(T_ar[step::step],av_test_error[step::step],
std_test_error[step::step],color='#5cc',label="Test error",marker='o')

plt.ylabel('Error')
plt.xlabel('Number of trees')

plt.legend(loc='lower left')

plt.savefig("accuracy.jpg",dpi=300)
plt.show()
```

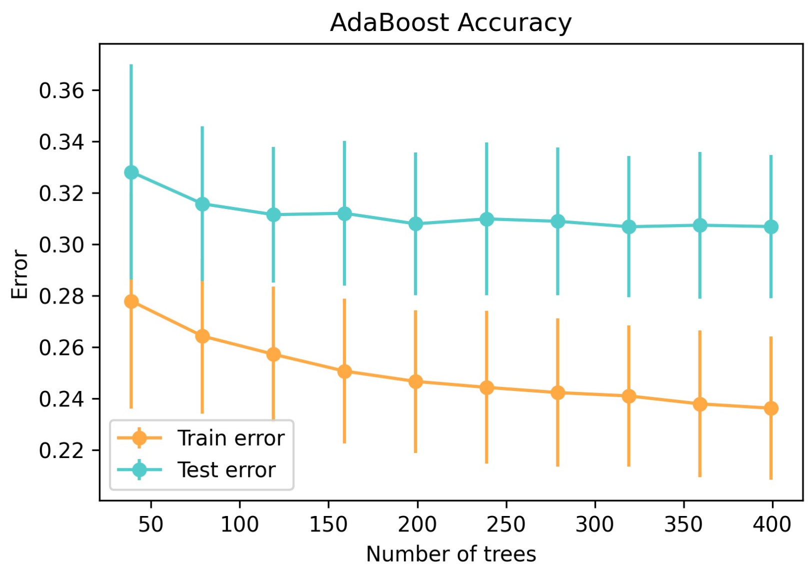

Figure 3: Accuracy

If we mean to dig a little more into the analysis, that is out of the purpose of the paper, an interesting insight would be to test if some cover types are easier to identify than others. In that case, one should just insert the code below into the **predict** function and add *type_loss* to the output returned by it. Then, we would have to catch it into **crossval** and we are able to perform the same operations such as averaging just as we did before with *train_error* and *test_error*. Note that, in this case, we find the percentage of records classified correctly instead of the loss to highlight where the classifier performed better.

```
arr1,arr2,type_loss = np.zeros(len(y_set)),
np.zeros(len(y_set)),np.zeros(len(y_set))
    for i in y_set:
        arr1 = np.copy(y_k)
        arr1[arr1 != i] = 0
        arr1[arr1 == i] = 1
        arr2 = np.copy(final_pred[T-1])
        arr2[arr2 != i] = 0
        arr2[arr2 == i] = 1

        arr3 = [a1 * a2 for a1,a2 in zip(arr1,arr2)]
        if (len(y_k[y_k == i]) != 0):
            type_loss[i-1] = np.sum([a1 * a2 for a1,a2
in zip(arr1,arr2)])/len(y_k[y_k == i])
```

The table below shows the results: the classification for the type four and seven is very good, while the second type of cover is not easy to spot for the classifier. The deviation in the test error of the fourth row is due to randomness, for there were no entries of that kind in one of the folds. Clearly, if we had to proceed in this analysis we wolud have to deal with it.

T	**1 - tr. error**	**1 - test error**
1	0.677±0.035	0.613±0.049
2	0.55±0.084	0.434±0.15
3	0.642±0.052	0.552±0.056
4	0.942±0.003	0.655±0.384
5	0.864±0.016	0.782±0.068
6	0.723±0.028	0.605±0.082
7	0.934±0.013	0.816±0.183

3 Conclusions

This project implements the ensemble method AdaBoost on the Roosevelt National Forest in Colorado dataset composed of 15120 records divided into seven classes and 54 features in a reasonable time. The number of rounds for the algorithm to converge is around T = 300. The performance could use some improvement, but still classifies about 70% of the entries correctly. Of these, the main part comes from the seventh and the fourth type of cover, while the second type was poorly distinguished from the others.