Forest Cover Type Classification using AdaBoost

Sofia Gervasoni and Luca Paoletti

May 2022

We declare that this material, which we now submit for assessment, is entirely our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of our work. We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by us or any other person for assessment on this or any other course of study.

Contents

1 Introduction

Starting from the Forest Cover Type Dataset, the aim of this project is to train seven binary classifiers (one for each of the classes in CoverType) and use them to make multiclass predictions, with the one-vs-all encoding technique. To do so, we will implement AdaBoost algorithm from scratch and we will run it using Decisions Stumps. We will then use external cross-validation to evaluate the multiclass classification performance (zero-one loss) for different values of the number T of AdaBoost rounds and the hyperparamters tuning to determine the optimal number of Adaboost rounds (T) and the learning rate.

The dataset contains tree observations from four areas of the Roosevelt National Forest in Colorado, including information on tree type, shadow coverage, distance to nearby landmarks, soil type, and local topography. The aim is to correctly predict which of the seven classes each tree belongs to.

To solve this problem of classification and make predictions, we implemented AdaBoost from scratch and we ran it using decision stumps. To implement the algorithm we used the software Python, whereas to clean the dataset and for the pre-processing of data we used the software R.

2 State of art

Today, machine learning is the premise of big innovations and promises to continue enabling companies to make the best decisions through accurate predictions. But what happens when the error susceptibility of these algorithms is high and unaccountable?
In this case, the so called *Ensemble Learning* will help us to improve our base classifier algorithm.

In the next lines we will see what are Ensemble Learning, how they are implemented and in particular we will focus on the method called *AdaBoost Classifier*, the one used for the current project of classification.
Before to underline the approach to Ensemble method, it is important to understand which type of base classifier we will use to make our first *base* predictions. An ensemble method works as boosting algorithm that helps with the improvement of a less efficient one. In this case our AdaBoost algorithm ensemble method will improve the base classifier.

2.1 Decision Stump and Decision Tree

The decision tree is a method of supervised learning used both for regression and classification. Each tree is made up of nodes and leaves. The nodes are the places where data, based on some conditions, are splitted; whereas the leaves are the places where data end after the splitting. In each tree, the nodes are tagged with tests and the leaves are tagged with labels.The aim is to obtain the features for which we obtain the best split, and to do so, we can use Gini function (it refers to Gini impurity) or Scale Entropy (it refers to Information Gain).
A decision stump is a one-level decision tree, that means that it only has one node and two leaves. It is weak learner.
Decision stump is one of the most straightforward classification algorithms.It is a binary classification algorithm with the idea of only focusing on one feature each time and finding a point that can separate data the most. Binary classification is for classifying two categories, such as 0 and 1.
We can write the equation as follows:

$$H = \{h_{a,c} : a \in \mathbb{R}, c \in \{+1, -1\}\}$$

$$h_{(a,c)}(x)\{c = -1 \text{ if } x \leq a, c = 1 \text{ if } x > a\}$$

We need to find one number (a). If the input value is larger than a, we classify it as 1. If it is smaller or equal to a, we classify it as -1. Our goal is to find the best number to classify our training data the most.
We can barely find a point that can perfectly separate the data. Therefore, we need a way to measure the performance. The most elementary way is the calculate the total point that we classify as 'wrong'.

$$E_{in} = \frac{\sum_{n}^{N}[y_n \neq \hat{y}_n]}{N}$$

We can also evaluate the purity of the two subgroups using the **Gini Index**.

$$1 - \sum_{k=1}^{K}(\frac{\sum_{n}^{N}[y_n = k]}{N})^2$$

If two subgroups are entirely pure, the Gini index will be 0.
Last, **Entropy** is another evaluation method of the threshold's performance. Entropy is an information theory metric that measures the impurity or uncertainty in a group of observations. It can be calculated as follows:

$$E = -\sum_{i=1}^{N} p_i \log_2 p_i$$

where p_i is the probability of randomly selecting an example in class i.
As far as Entropy refers to **Information Gain**, we need to introduce also this last metric. We can define information gain as a measure of how much information a feature provides about a class. Information gain helps to determine the order of attributes in the nodes of a decision tree.
The main node is referred to as the parent node, whereas sub-nodes are known as child nodes. We can use information gain to determine how good the splitting of nodes in a decision tree is.
It can help us determine the quality of splitting, as we shall soon see. The calculation of information gain should help us understand this concept better.

$$Gain = E_{parent} - E_{children}$$

The term Gain represents information gain. E_{parent} is the entropy of the parent node and $E_{children}$ is the average entropy of the child nodes.

2.2 Ensemble methods

Ensemble methods are techniques that create multiple models and then combine them to produce improved results. Ensemble methods usually produce more accurate solutions than a single model would.
In the state of art there exist several Ensemble Methods, the most popular are: voting, stacking, bagging and boosting. In the next lines, we will underlined the last one, the Boosting Ensemble Method, because of the usage in the project of the AdaBoost, one of the main ensemble method in the class of boosting technique.
The term 'boosting' is used to describe a family of algorithms which are able to convert base classifier to strong models. The classifier is weak if it has a substantial error rate, but the performance is not random (resulting in an error rate of 0.5 for binary classification). Boosting incrementally builds an ensemble by training each model with the same dataset but where the weights of instances are adjusted according to the error of the last prediction. The main idea is forcing the base classifier to focus on the instances which are hard.

2.2.1 AdaBoost

AdaBoost (adaptive boosting) is one of the main boosting algorithm. This method combines weak learners (in our case decision stumps) to make classification and each of the stumps is made made by taking the previous stump's mistakes into account.
Boosting is an ensemble method, so it combines results of multiple models in such a way that together they will perform much better than each individual. In particular, with boosting techniques, training gives more importance to the data that was miss-classified by the previous models and to decide the output, a weighted voting is needed.
AdaBoost has been designed for solving two-class classifications problems, where the goal is to classify each example in one of the two classes. To pass from two-class to multi-class classification problems with AdaBoosting, the most used techniques reduce the multi-class problems into multiple binary-class problems. Nevertheless it is possible to extend the AdaBoost algorithm to the multi-class case without reducing reducing it to multiple two-class problems.

2.3 External Cross-Validation & Hyperparameters Tuning

The external cross-validation is used to evaluate the effectiveness of a learning algorithm. Assuming the hyperparameters fixed, the aim is to estimate

$$\mathbb{E}[l_D(A)]$$

To do so we can use a technique called K-fold (external) cross-validation.
We start partitioning our entire dataset, S, in K subsets (the so called 'folds') $D_1, \dots, D_K$ of size m/K each (where m is the total number of observations). Let's define $S^{(i)} \equiv S/D_i$, we will have that D_i is our *testing part* of the i-th fold while $S^{(i)}$ is the *training part.*
In order to get clear information in the conceptual part, we will make an example. If we partition $S = \{(x_i, y_1), \dots, (x_{20}, y_{20})\}$ in $K = 4$ subsets

$$D_1 = \{(x_1, y_1), \dots, (x_5, y_5)\} \quad D_2 = \{(x_6, y_6), \dots, (x_{10}, y_{10}\}$$
$$D_2 = \{(x_{11}, y_{11}), \dots, (x_{15}, y_{15})\} \quad D_4 = \{(x_{16}, y_{16}), \dots, (x_{20}, y_{20}\}$$

then $S^{(2)} = \{(x_1, y_1), \dots, (x_5, y_5), (x_{11}, y_{11}), \dots, (x_{20}, y_{20})\}$.
The K-fold CV estimate of $\mathbb{E}[l_D(A)]$ on S, denoted by $l_S^{CV}(A)$, is then computed as follows:
we run A on each training part $S^{(i)}$ of the folds $i = 1, \dots, K$ and obtain the predictors $h_1 = A(S^{(1)}, \dots, h_K = A(S^{(K)})$. We then compute the (rescaled) errors on the testing part of each fold,

$$l_{D_i}(h_i) = \frac{K}{m} \sum_{(x,y) \in \ D_i} l(y, h_i(x))$$

Finally, we compute the CV estimate by averaging these errors

$$l_S^{CV}(A) = \frac{1}{K} \sum_{i=1}^{K} l_{D_i}(h_i)$$

So, intuitively, cross-validation consists in using a limited sample (the training set) to estimate how well the learning algorithm will perform, in general, when used to make predictions based on data different from the ones contained in the training set.
As far as tuning hyperparameters are concerned, we face the problem of finding the best parameters so to obtain a predictor with small risk. This is typically done by minimizing a risk estimate computed using the training data. As Θ may be very large, possibly infinite, the minimization is generally not over Θ, but over a suitably chosen subset $\Theta_0 \subset \Theta$. If S is our training set, then we want to find $\theta^* \in \Theta_0$ so that:

$$l_D(A_{\theta^*}(S)) = \min_{\theta \in \Theta_0} l_D(A_\theta(S))$$

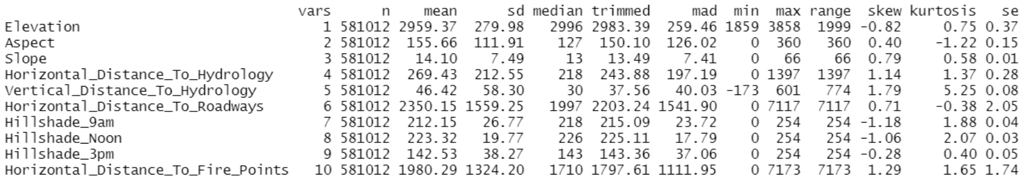
The estimate is computed by splitting the training data in to two subset S_{train} and S_{dev}. The development set S_{dev} is used as surrogate test set. The algorithm is run on S_{train} once for each value of the hyperparameter in Θ_0. The resulting predictors are tested on the dev set. In order to obtain the final predictor, the learning algorithm is run once more on the original training set S using the value of the hyperparameter corresponding to the predictor with smallest error on the validation set.

3 Data

The dataset Forest Cover Type contains 581012 records (each one represents a tree) and 55 columns.
The label set Y is defined as $Y = \{1, 2, 3, 4, 5, 6, 7\}$ and represent by the variable *Cover-Type*, which contains the seven classes of of trees located in the Roosevelt National Forest. There are 54 input (prediction) variables , that represent some characteristics of the trees, location and the soil in which are planted. More in detail, these features are: *Elevation, Aspect, Slope, Horizontal Distance to Hydrology, Vertical Distance to Hydrology, Horizontal Distance To Roadways, Hill shade (9 am), Hill shade (Noon), Hill shade (3 pm), Horizontal Distance to Fire Points, 4 areas of Wilderness and 40 types of soil.*
The goal is to find a classification rule from the training data, so that when we give a new input we can assign it a class label from $\{1, 2, 3, 4, 5, 6, 7\}$.

3.1 Descriptive analysis of variables

Using the *describe* (or *summary*) command in R, we obtained a quick descriptive analysis of the dataset. Looking at the results of this analysis we pointed out that there are no NULL values (for each variable we have 581012 observations), there are no missing values and the variables are all numeric. The first ten variables are continuous variables, whereas the other ones are categorical features.
In the following table we summarized the descriptive statistics for the first 10 variables (continues variables):



```
                                   vars      n    mean      sd median trimmed     mad  min  max range  skew kurtosis   se
Elevation                             1 581012 2959.37  279.98   2996 2983.39  259.46 1859 3858  1999 -0.82     0.75 0.37
Aspect                                2 581012  155.66  111.91    127  150.10  126.02    0  360   360  0.40    -1.22 0.15
Slope                                 3 581012   14.10    7.49     13   13.49    7.41    0   66    66  0.79     0.58 0.01
Horizontal_Distance_To_Hydrology      4 581012  269.43  212.55    218  243.88  197.19    0 1397  1397  1.14     1.37 0.28
Vertical_Distance_To_Hydrology        5 581012   46.42   58.30     30   37.56   40.03 -173  601   774  1.79     5.25 0.08
Horizontal_Distance_To_Roadways       6 581012 2350.15 1559.25   1997 2203.24 1541.90    0 7117  7117  0.71    -0.38 2.05
Hillshade_9am                         7 581012  212.15   26.77    218  215.09   23.72    0  254   254 -1.18     1.88 0.04
Hillshade_Noon                        8 581012  223.32   19.77    226  225.11   17.79    0  254   254 -1.06     2.07 0.03
Hillshade_3pm                         9 581012  142.53   38.27    143  143.36   37.06    0  254   254 -0.28     0.40 0.05
Horizontal_Distance_To_Fire_Points   10 581012 1980.29 1324.20   1710 1797.61 1111.95    0 7173  7173  1.29     1.65 1.74
```

Figure 1: Descriptive statistics

An interesting thing to notice is that the variable *Vertical Distance To Hydrology* takes negative values (as the minimum value is -179). As this variable is defined as a distance (*vertical distance to nearest surface water features*), and distances take only positive values, it does not make so much sense. A possible explanation of this fact is that the nearest hydrology point is below the tree, so it takes negative values because the nearest hydrology point is below sea level.

The categorical features refer to the belonging of the tree to a Wildness Area (there are 4 wildness areas) and the Type of Soil (there are 40 soil types). These categorical features are encoded with the *One Hot Encoding* technique, so they take value 1 if the tree belongs to the wildness area/type of soil, or 0 if it does not.
Speaking about the response variable, Cover Type, there are different number of observations for each of the seven classes. Most of the observations are in the second (48.76%) and in the first (36.46%) class. The other classes have lower numbers of observation,

in particular the third class contains 6.15% of the total observations, followed by the seventh class (3.53%), the sixth class (2.99%), the fifth class (1.63%) and the fourth class (0.47%). The dataset in this case results to be unbalanced and this could lead to lower effectiveness of the algorithm, especially when predicting minority classes. Having unbalanced classes could be misleading when it comes to evaluate the accuracy of an algorithm, because we could obtain high accuracy without actually making useful predictions. In this case confusion matrices perform better since they better summarize the performance of a classification algorithm.
Given the fact that we have an unbalanced dataset (the classes are not represented equally), measuring the performance of our model with accuracy could lead to misleading results. With imbalanced classes, it's easy to get a high accuracy without actually making useful predictions. So,we could follow two paths:

1. Re-sample the dataset using under-sampling or over-sampling techniques. The former technique removes samples from over-represented classes whereas the latter adds more samples from under-represented classes.
2. Use metrics obtained from confusion-matrix (e.g. F1 weighted, precision weighted, recall weighted, Matthews Correlation Coefficient), in this way we can summarize the performance of a classification algorithm, even if the dataset is unbalanced.

We decided to follow the second path, leaving the dataset unbalanced.

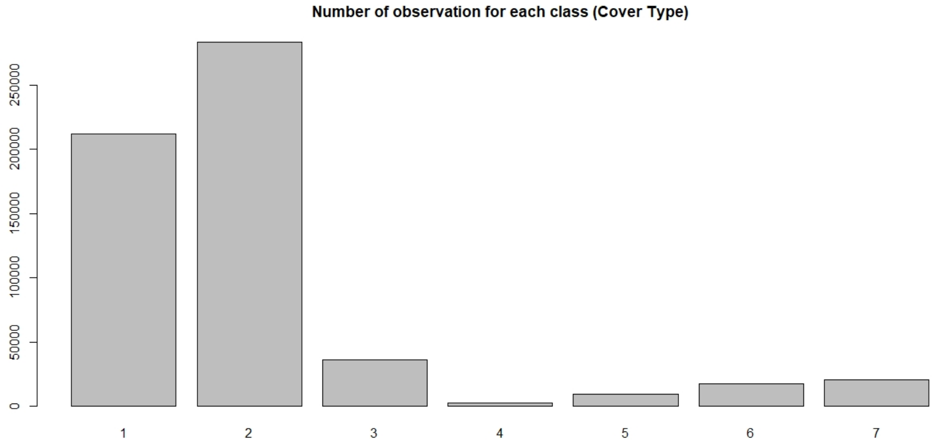


Figure 2: Number of observation for each class (Cover Type)

3.2 Outliers

To detect the presence of outliers we used the interquartile rule and the related visual tool (the box plot). Following this rule we defined as extreme outliers the values that are below the lower limit and the values that are above the upper limit. We considered extreme outliers the observations which are more than 3 times the interquartile range below the first quartile or above the third quartile.

$$\text{lower limit} = Q1 - (3 \times IQR)$$

$$\text{upper limit} = Q3 + (3 \times IQR)$$

Where $Q1$ represents the first quartile, $Q3$ represents the third quartile and IQR represents the interquartile range ($Q3 - Q1$).
Each of the 7 classes contains different types of trees, so when it comes to detect the presence of outliers in the 10 continuous variables we should group the observations by the CoverType.

We pointed out that there are 8963 extreme outliers in total. The following table summarizes the distribution of the outliers in the continuous variables grouped by CoverType.

Variable Name	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Elevation	0	0	0	0	0	0	360
Aspect	0	0	0	0	0	0	0
Slope	96	63	0	0	0	0	33
Horizontal Distance To Hydrology	0	377	0	0	41	0	0
Vertical Distance To Hydrology	2293	3511	0	0	0	7	4
Horizontal Distance To Roadways	0	0	0	0	0	0	0
Hillshade 9am	411	610	0	0	0	3	12
Hillshade Noon	430	296	0	0	12	1	89
Hillshade 3pm	0	0	0	0	0	0	0
Horizontal Distance To Fire Points	0	19	0	0	295	0	0

Most of the outliers are in the class type 1 and 2, but it is mainly due to the fact that most of the observations are concentrated in these two classes. If we look at the proportion of outliers for each class it is possible to notice that the class with the higher number of outliers (proportion) is the fifth (with 3.67% of outliers on the total), followed by the second (with 1.72% of outliers on the total) and the first (with 1.52% of outliers on the total) classes. Classes 3 and 4 do not present any outlier. These results are graphically represented with the barplot in Figure 3. As is it possible to notice from the table above, the variables with the higher number of outliers are *Vertical Distance To Hydrology* and *Hillshade 9am*. So, we took a look at the boxplots of these two variables to see how the outliers are distribute and how severe they are.

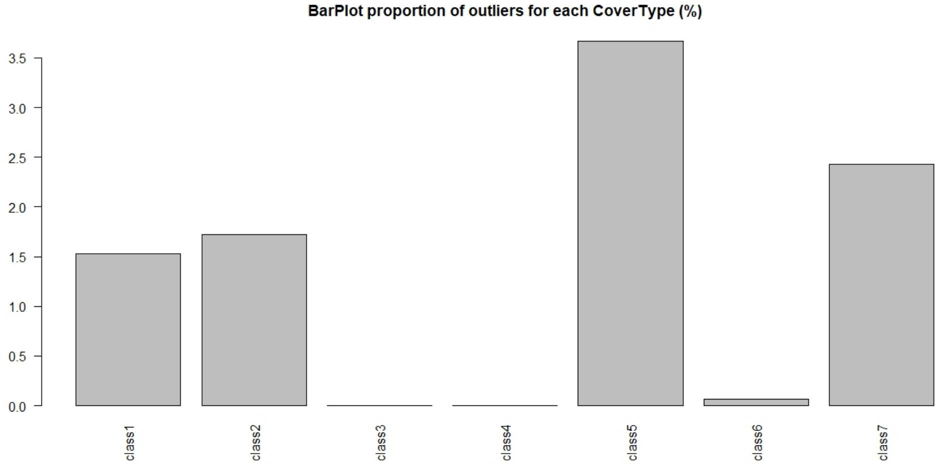


Figure 3: BarPlot: Proportion of outliers for each class (%)

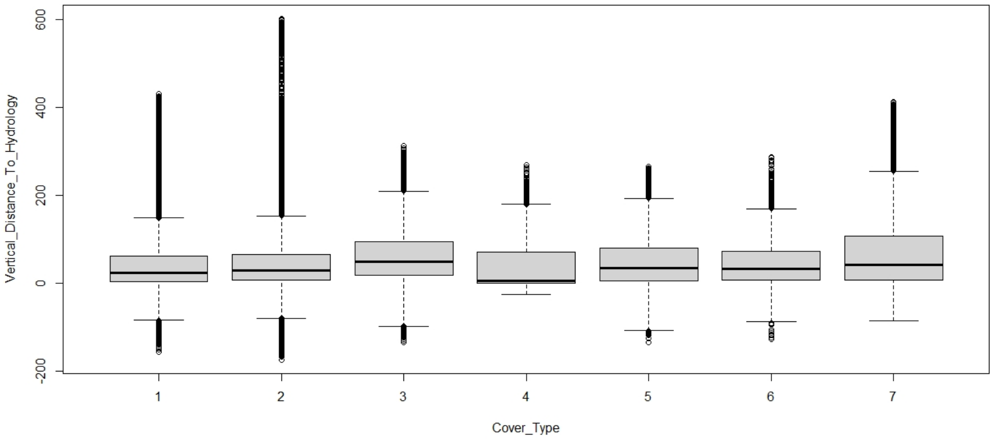


Figure 4: BoxPlot Vertical Distance To Hydrology

As we can notice from Figure 4, most of the outliers are in the first and second class type. Moreover, for these two classes the outliers seem largely out from the whiskers.

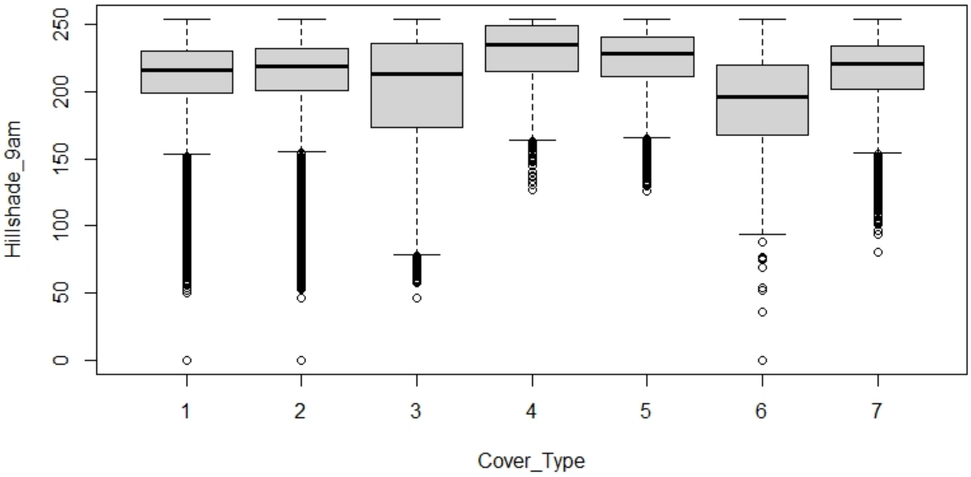


Figure 5: Boxplot Hillshade 9am

In Figure 5 we can notice how, also in this case, most of the outliers are for the first and second class type. This is mainly due to the fact that most of the observations are in these two classes.

We decided to proceed dropping all the records containing extreme outliers, obtaining a new dataset with 572627 observations. With this operation we lost 8385 observations (1.44 % of observations), which is not a relevant quantity given the large number of observations that we have in this dataset.

3.3 Distributions and standardization

Taking a look at the distributions of the continuous variables (Figure 6) it is possible to notice that the variables are not normally distributed and in particular we are dealing with out-of-scale quantities. Scaling is particularly useful when dealing with distance-based algorithms, because they are using distances between data points to determine their similarity. On the other hand, it has not a big effect on tree-based algorithms as the decision tree splits a node on a feature that increases the homogeneity of the node. As we will deal with tree-based algorithms (Decision Stumps), we decided not to rescale the variables, as this worsens the accuracy of our predictions.

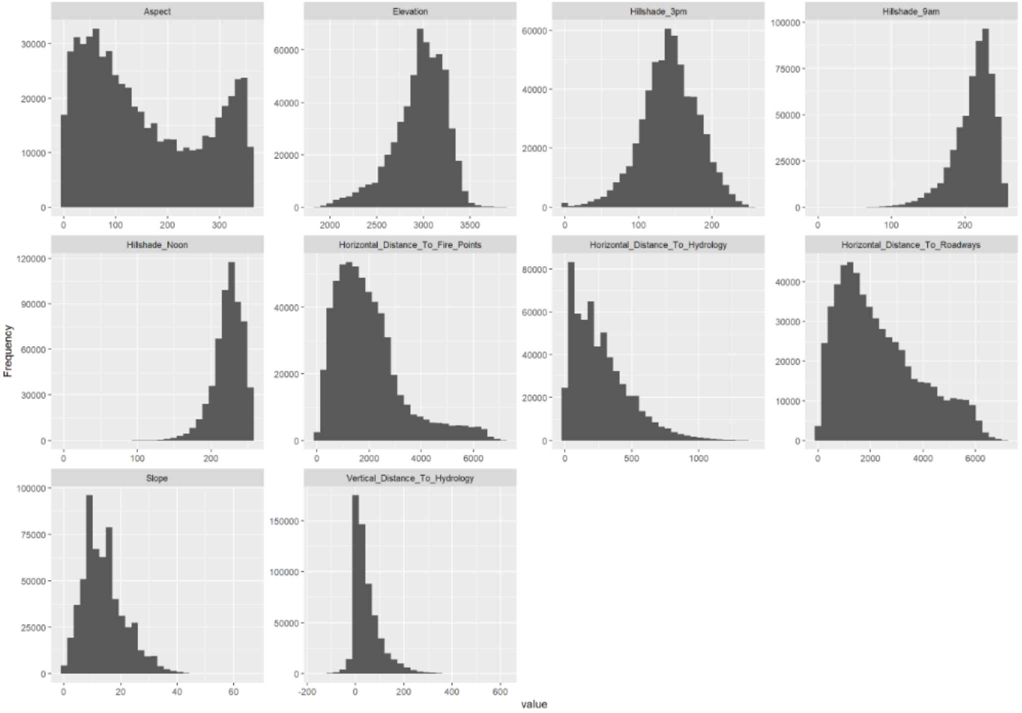


Figure 6: Distributions

3.4 Correlation

We now proceed analyzing the relationship between the ten continuous variables.We calculated the correlation between the continuous variables and we plotted them with the following heatmap (Figure 7).

From Figure 7 it is possible to notice that there is a strong negative correlation (-0.79) between *Hillshade 9am* and *Hillshade 3pm*. This makes sense because the sun at 9am is on the opposite side to the position of the sun at 3pm, so the shade of the hill at 9am will be the opposite than the one at 3pm. There is no correlation between *Hillshade 9am* and *Hillshade Noon*, whereas there is a strong positive correlation (0.6) between *Hillshade 3pm* and *Hillshade Noon*.
It is also interesting to notice that the correlation between CoverType and the continuous variables is very weak. There is almost no correlation between CoverType and Aspect, Hillshade 9am and Hillshade 3pm. The variable most correlated with CoverType is Elevation, with which it has a negative correlation (-0.27): the last classes (eg. class types 6 and 7) have lower values for the variable Elevation (we expect that the trees in the last classes are lower than the ones in the first classes).

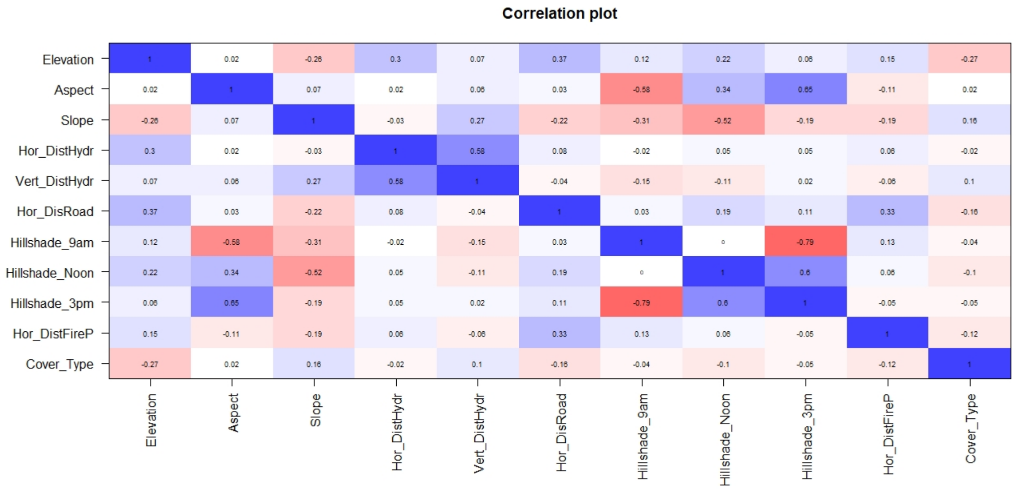


Figure 7: Correlation Plot (continuous variables)

4 Implementation and methodology

In order to create a classification algorithm from scratch, we will introduce different classes of object and different functions with different functionalities.
As we know, the main algorithm in this project is the AdaBoost classifier, but, like other ensemble methods, it needs a base classifier to start from: in this case the base classifier is a decision stump, a.k.a. Decision Tree with depth equal to 1.

In the following subsection we will explain how we created the 2 algorithms:

- Decision Stump
- AdaBoost

4.1 Decision Stump function

As we have previously mentioned, Decision Stump algorithm is the base classifier that lets the AdaBoost work, in fact, in our case, we created a class object *Decision Stump* with one simple function, the *predict* function.
At every round of the Adaboost, the predict function will create two leaves from the root according to the threshold that the ensemble method extracts. If the column value is lower than the threshold, the row will be predicted as -1, otherwise it will be predicted as 1. This type of algorithm is very very simple and sometimes could not work so well because of the depth of the decision tree. Despite of that, the issue was in part solved by the AdaBoost that will improve the classification task.

With this type of algorithm we can only predict a binary response variable and in order to do that, for each of the 7 classes, we will make a one vs all prediction. What does that mean? Basically, for each class, we predict the tested class as 1 and all the other as -1. In our example we will have this differentiation among the 7 cover type's classes (e.g. $1 = 1, -1 = [2, 3, 4, 5, 6, 7]$).

4.2 AdaBoost function

The core part of the coding part is the *AdaBoost* class. We have already explained how it works and consequently we will not deep dive the theoretical concepts in this section. As for the *Decision Stump*, also in this case an *AdaBoost* class has been created; the difference with the base classifier class is the presence of one more function, the *fit* function, and the presence of one parameter that *Adaboost* takes as input.

Let's follow the order of the script, **fit** function. The objective of this function is to fit the model, already split into train and test set, with a Decision Stump as base classifier. First we initialize the weights to $w_i = \frac{1}{N}$; after that, we start the iteration through classifiers. At the beginning we mentioned the number of AdaBoost rounds, the number of classifier to use, and in this function we repeat the process until the T round is reached. The process is the following: we make a greedy search to find the best threshold and feature, how do we do that? We consider each column and for every unique value of the column selected, we make some basic prediction of 1 and -1 whether the record is lower or greater than the threshold (unique value). Once we have done that, we define an error as $error = \sum w_i[\hat{y}_i \neq y_i]$ and, if the error is greater than 0.5 we invert the prediction and the error becomes $1 - error$. Why? Since this is a greedy research of the best configuration and and as far as we have only a binary classification, if the error is greater than 0.5 means that we missclassified more than 50% of the records; inverting the polarity (from 1 to -1) we also have more correctly classified records and a lower error rate than before.
At the end of each round, we store a new threshold and a new feature only if the error of the round before was higher. When the error is defined as the best on for that Decision Stump, we upgrade the weights (w_i), we normalize them to one and we move to the next base classifier (next round). At the end of round T we finish our fitting method and we can proceed with the next function, the *prediction* function, that will need the final weight w_i and the final alpha α_i (all metrics explained before).

The **prediction** function is the last function of our algorithm, it has the objective to make the final prediction of our binary classification.
The function takes as input only the test set, and uses the previously stored greedy prediction, α_i and w_i in order to make the final prediction $\hat{y}_i$.
As far as this classifier works with a *sign* function, we will have 7 final continuous $\hat{h}$, one for each 'one vs all' classifier. Of course we can evaluate our model for every binary prediction, but, the aim of the project is a multiclass classification, and in order to solve this problem, we will predict our final $\hat{y}$ as the sign of the greatest final continuous prediction ($\hat{h}$).

With this final result, we have completed the AdaBoost classifier from scratch using a Decision Stump as base classifier.

4.3 Hyperparamters tuning and External Cross Validation

Before submitting the results of the project, we need to define hyperparameters (in this case we will see only the T rounds of AdaBoost and Learning Rate as hyperparamters) and to apply external cross validation to get more accurate findings.

4.3.1 Hyperparamters tuning

The T **rounds of Adaboost** are the number of base classifiers that we want to tune before to make prediction on the test set. The idea is that, as we increase T, the evaluation metrics improves. Unfortunately we don't have machines with such a computing power to fulfill our desire, but, as far as the next plot is concerned we can see, after the 100th round of Adaboost the evaluation metrics decline. We tuned this hyperparameter fixing temporarily the learning rate = 0.5.We will use $T = 100$ round of Adaboost to train our model, as it is the value that provides the best results in terms of evaluation matrix. In Figure 8 we displayed how the evaluation metrics behave given different values for T.

The second hyperparameter tuned is the **Learning Rate**, that controls the loss function used for calculating the weight of the base models. We pointed out that the learning rate that improve our metrics is = 0.6, as it is possible to notice from Figure 9, where we displayed how the evaluation metrics change given different learning rates.

There is a trade-off between the learning rate and the number of rounds, the higher the learning rate the fewer estimators we need.
We finally decide to set the test size = 0.2 (train size = 0.8). This latter parameter is defined as constant during the entire project.

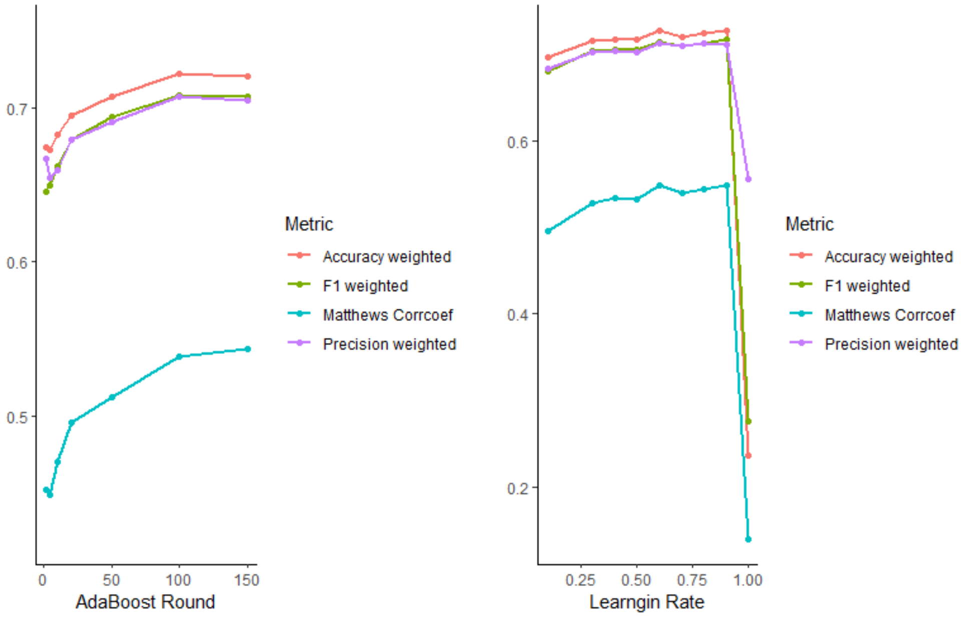


Figure 8: Hyperparameters tuning

4.3.2 External Cross Validation

During the implementation of external cross validation we assume the hyperparamters (T rounds) as fixed and the goal is now to test the model on different training and validation set in order to see whether the model fits well despite the random sampling of the initial training set.
At each k folder of the cross validation, we will have an efficiency metric as output (in our case will be the accuracy) and at the end we will find the average accuracy as quality metric of the trained model. Theoretically, once we predict the response feature of the test set, we will find a similar accuracy to the average one, found with the external cross validation.
It is fundamental to choose correctly the number of splits K, because choosing a wrong K could result in a mis-representative idea of the skill of the model. To balance the bias-variance trade-off associated with the choice of K in K-fold cross-validation, it has been shown empirically that using $K = 5$ (or $K = 10$), test error rate estimate does not suffer neither from excessively high bias nor from very high variance.

5 Results

Here are the results obtained with the algorithm developed.
As far as the classification problem is concerned we summarized our outcomes starting from a confusion matrix. In our case, the classification matrix sums up on the diagonal

the correctly classified observations, while on the other cells the miss-classified ones; in particular, each row represents the actual classes while the columns represent the predicted labels.
In order to read and describe the matrix in a more comprehensive way, we calculated some evaluation metrics that help with the understanding of the results. In particular we defined the accuracy (weighted) as

$$accuracy(weighted) = \frac{\sum_{k=1}^{K} \frac{TP}{Total_{row_k} \times w_k}}{K \times W}$$

the Macro F1 score as

$$MacroF1Score = 2 \times (\frac{MacroAevragePrecision \times MacroAverageRecall}{MacroAveragePrecision^{-1} + MacroAverageRecall^{-1}})$$

the Macro Average Precision as

$$MacroAveragePrecision = \frac{\sum_{k=1}^{K} Precision_k}{K}$$

where

$$Precision_k = \frac{TP_k}{TP_k + FP_k}$$

the Macro Average Recall as

$$MacroAverageRecall = \frac{\sum_{k=1}^{k} Recall_k}{K}$$

where

$$Recall_k = \frac{TP_k}{TP_k + FN_k}$$

and finally the Matthews Correlation Coefficient as

$$MCC = \frac{c \times s - \sum_{k}^{K} p_k \times t_k}{\sqrt{(s^2 - \sum_{k}^{K} p_k^2)(s^2 - \sum_{k}^{K} t_k^2)}}$$

To simplify the definition, it is necessary to consider the following intermediarte variables:

- $c = \sum_{k}^{K} C_{kk}$ the total number of elements correctly predicted
- $s = \sum_{i}^{K} \sum_{j}^{K} C_{ij}$ the total number of elements
- $p_k = \sum_{i}^{K} Cki$ the number of times that class k was predicted (column total)
- $t_k = \sum_{i}^{K} C_{ik}$ the number of times that class k truly occured (row total)

While the firsts metrics vary from 0 (worst) to 1 (best), the latter describe how random our prediction has been. In particular, as every correlation metrics, a value of -1 defines an opposite prediction, a value of 0 defines a random guessing prediction and finally a value of 1 defines a perfect prediction.
Our results for these metrics appear in the table below

Metrics	**Values**
Accuracy weighted	0.727
F1 weighted	0.714
Precision wighted	0.712
Matthwes Correlation coefficient	0.548

Finally, we presented a plot that describes perfectly the prediction we made, how many correct prediction (HIT), and which wrong predictions were made in which classes. The plot displays the performance on the test set of the algorithm developed. Each class is represented with a different colour and the portion of bar with the wording ”Hit” corresponds to the observations correctly classified. As it is possible to see from the graph the first three classes and the seventh class are mostly correctly classified, whereas the fifth class is the worst classified. The fourth and sixth classes are the also fairly well classified.

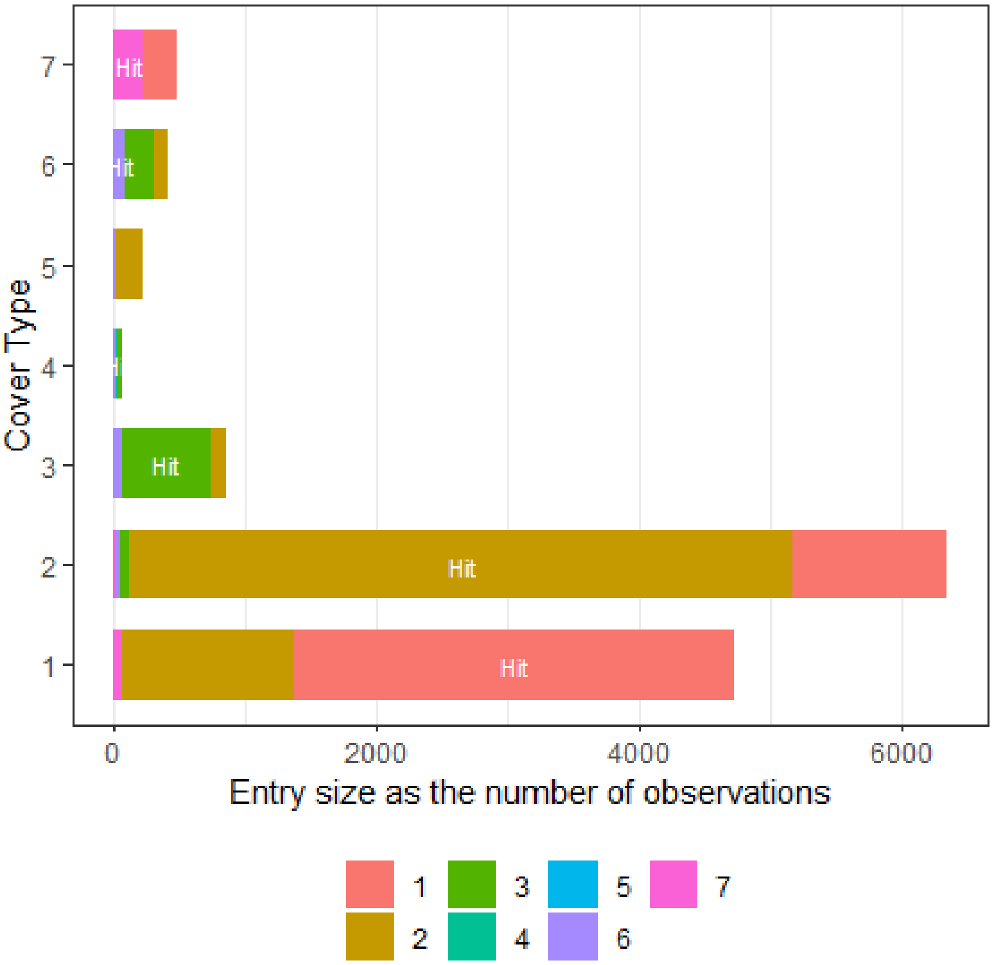


Figure 9: Barplot Confusion matrix

6 Conclusions

To sum up, we are satisfied with the results obtained running the AdaBoost algorithm developed from scratch, as most of the observations are correctly classified in the seven classes. The cause of the misclassification of some observation in the less frequent classes can be traced back to the unbalancing of classes, nevertheless the weighted evaluation metrics show good performances.

Moreover, we did not meet any problems of over or under-fitting since the evaluations metrics obtained for the training and the test set resulted very close to each other (and the train and test errors are not high). That last result was obtained also thanks to the fact that in each subset of the dataset, training or validation set, the classes were equally distributed as their original distribution was in the complete dataset. That is a proof of the good dataset's splitting method we used in order to define training validation and test sets. In particular, we can see how the majority classes (1 and 2) are always more present in the different subsets, while the other classes still do not appear as the first two.

In our opinion, one of the main limits of this algorithm is the fact that we are using a decision stump as 'weak learner' of Adaboost. The decision stump is a tree with depth equal to 1 and as a consequence, could rarely create good split from the roots; in addition, the decision stump can only classify binary label while we are facing a multiclass classification problem. In fact, the tough part of the project was to move from a binary classification problem to a multiclass classification problem and, despite the fact that we successfully solved that issue, we are still using a binary classifier as 'Decision Stump' to make mutliclass predictions.

Comparing our algorithm built from scratch result with the scikit learn's one, we obtained similar values (or also better results from our algorithm). We show the results obtained (on the test set) from the two algorithms in the following table:

Metrics	**Our algorithm**	**ScikitLearn's algorithm**
Accuracy weighted	0.727	0.630
F1 weighted	0.714	0.627
Precision weighted	0.712	0.632
Matthews Correlation coefficient	0.548	0.407

The barplot in Figure 10 provides a visual representation of the confusion matrix obtained running the ScikitLearn's algorithm. As in the case of our algorithm, it works particularly well for the majority classes, whereas for the minority classes the classification is not that good. This algorithm performs particularly badly for the fourth class, where all the observations are misclassified, whether, as far as our algorithm built from scratch concerned, the same class is definitely better classified. Also for the seventh class our algorithm performs better than the one obtained from ScikitLearn's ones. So, in general, we could say that our algorithm built from scratch works fairly well and in most of the cases produces reliable predictions.

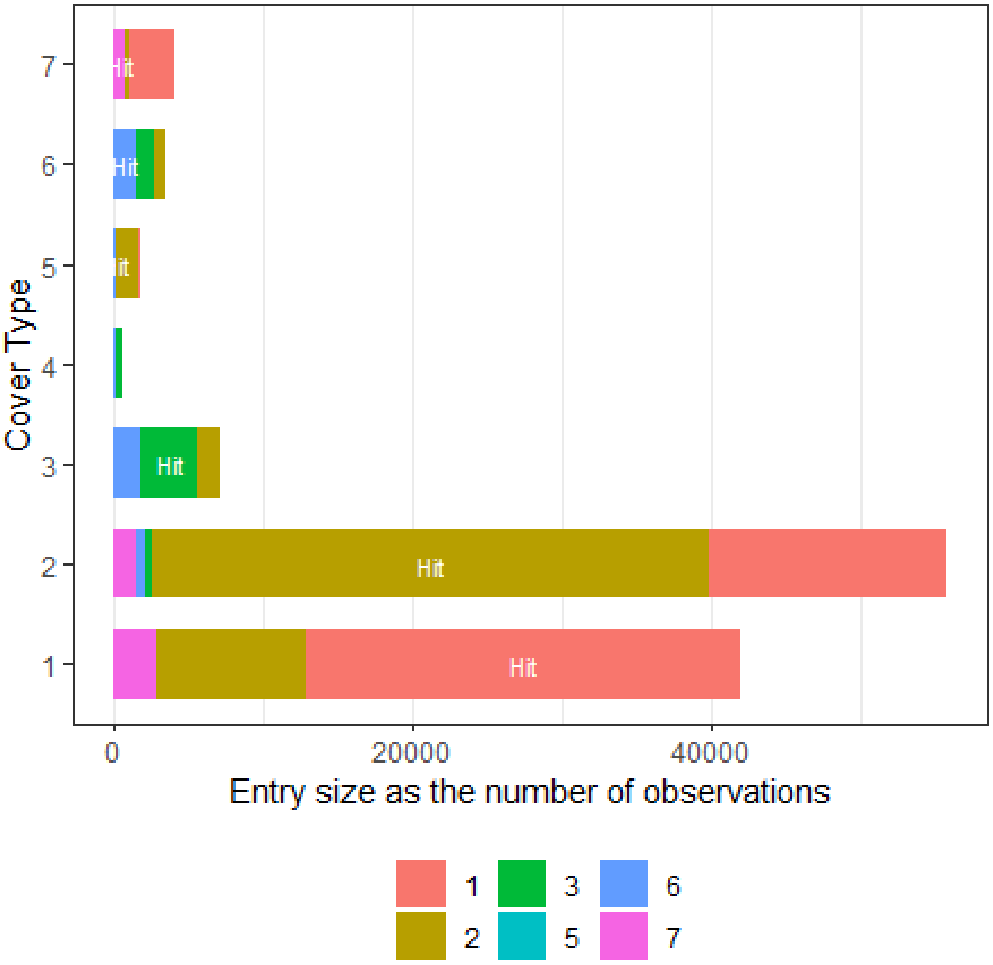


Figure 10: Barplot Confusion matrix ScikitLearn

References

[1] Metrics for multi-class classification: an overview
Margherita Grandini, Enrico Bagli, Giorgio Visani
August 14, 2020

[2] Boosting: Foundations and Algorithms
Robert E. Schapire, Yoav Freund
October 1,2012

[3] Multi-class AdaBoost
Ji Zhu , Hui Zou , Saharon Rosset and Trevor Hastie
Statistics and Its Interface Volume 2 (2009) 349–360

[4] Comprehensive Guide to Multiclass Classification Metrics
To be bookmarked for LIFE: all the multiclass classification metrics you need neatly explained
Bex T.
Jun 9, 2021

[5] Machine Learning — Statistical Methods for Machine Learning course notes
Professor Nicolò Cesa-Bianchi
March-May 2022