Forest Type Classification Using AdaBoost

Ricardo Murillo Rapso
University of Milan

May 31th 2020

Abstract The following assignment uses decision stumps with AdaBoost meta-algorithm using to train several classifiers on the Forest Cover Type Dataset with a one vs all approach. A total of seven different binaty classifiers are trained, based on each of the categories of Forest Type Cover that are available on the database, then the classification performance for different values of the number of rounds in AdaBoost is studied using external cross-validation to evaluate its accuracy. It is found in general, that the accuracy is better, using a large number iterations of AdaBoost, usually, 10. Beyond that, the accuracy usually gets worse.

Keywords: Machine learning, AdaBoost, Decision Stumps, One vs all-encoding.

Contents

1 Introduction

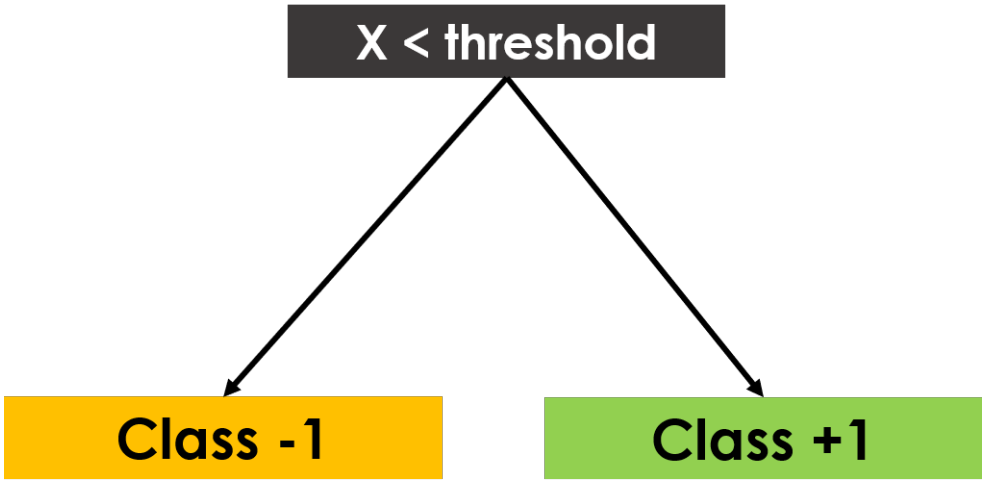
Ensemble Methods are meta-algorithms that that try to improve the capabilities of learning algorithms in order for them to generate better predictors without having to change the current set of training data. Adding randomization and then combining predictors has an average effect on the performance of the learning algorithm, which corresponds to reducing variance and increasing bias. The most popular Ensemble Methods meta-algorithms are Bagging, Boosting and Random Forest. [Lantz, 2015] describes bagging meta-algorithm as the one that consists on generating an additional number of training sets, by using a bootstrap sampling on the original training data, Random Forest, uses this same bagging principle, but to generate extra diversity on the decision tree models, and booting consists on a meta-algorithm that boosts the performance of weak learners for them to attain the performance of stronger ones. Boosting is an algorithmic paradigm that grew out of a theoretical question and became a very practical machine learning tool [Shalev-Shwartz and Ben-David, 2013]. Adaboost, is a type of Boosting meta-algorithm. It was proposed by Freund and Schapire in 1997, and whose idea is that of generating weak learners that iteratively learn a larger portion of the difficult-to-classify examples by giving more weight to frequently misclassified examples[Lantz, 2015]. This assignment consists on the application of the AdaBoost meta-algorithm to a decision stump classifier using the Forest Type Cover dataset. Following this section, there will be a description of the algorithm, of the Forest Type Cover Dataset, the actual methodology applied and a description of the results.

2 Description of the algorithms used

2.1 Decision Stumps

Decision tree learners in machine learning, are the ones making use of a tree structure (roots, trunks, branches and leaves) to model the relationships among the features and the potential outcomes. In much the same way, a decision tree classifier uses a structure of branching decisions, which channel examples into a final predicted class value [Lantz, 2015]. A decision stump is a special case of this structure of machine learning models consisting of a one-level decision tree. Figure 1 shows and example of a Decision Stump, where there exist only two classes. Forming part of one or another class in this case comes from being under of below a certain threshold established. After the model is created, many decision tree algorithms output the resulting structure in a readable format. This provides tremendous insight into how and why the model works or doesn't work well for a particular task. Algorithms for the construction of decision trees usually work in a top-down manner, choosing at each step the variable that best divides the set of elements.

Figure 1: Decision Stump



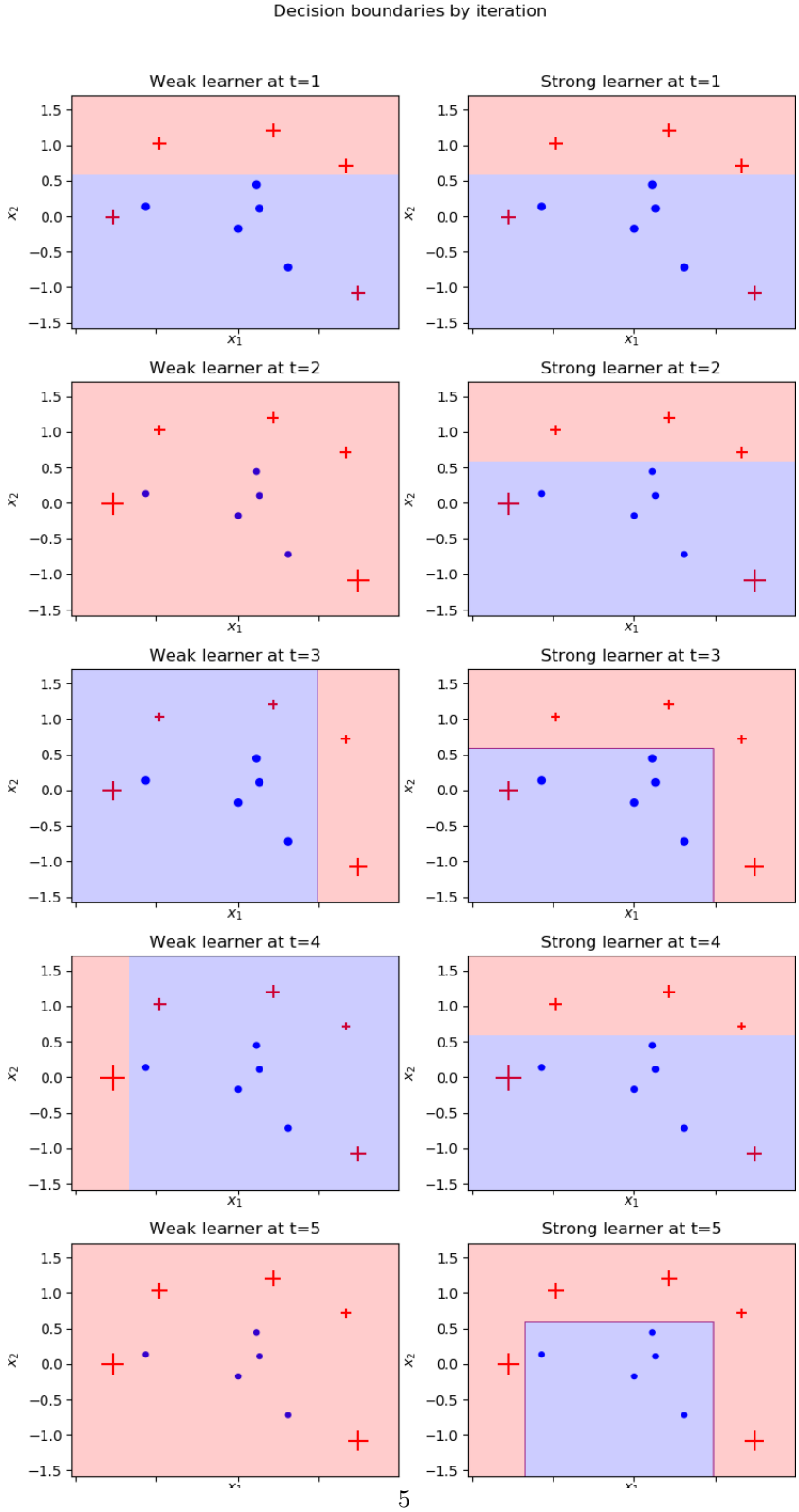
2.2 AdaBoost

AdaBoost, or adaptive boosting, is based on the idea that having a group of experts to make decisions is better than having only one. The group of experts is known as an assembly and this represents a strong classifier, that is, a classifier with very good precision. On the other hand, weak classifiers,are the ones that show less precision. For this matter a weak classifier is considered a classifier that performs slightly better than random[Mohri et al., 2018]. Boosting algorithms, construct a strong classifier using only a training set and a set of weak learning classifiers. This algorithms are currently among the most popular and most successful algorithms for statistical learning [Rudin et al., 2007].

The name of AdaBoost comes from the abbreviation of adaptive boosting and refers to a different weighting strategy for each of the examples, during training. Those examples that were incorrectly classified in one iteration receive a higher weighting in the next iteration. On the contrary, the examples that were correctly classified in that iteration will receive less importance in the next. This strategy allows AdaBoost to focus, iteration after iteration, on those examples that have not yet been correctly classified by the weak classifier assembly.

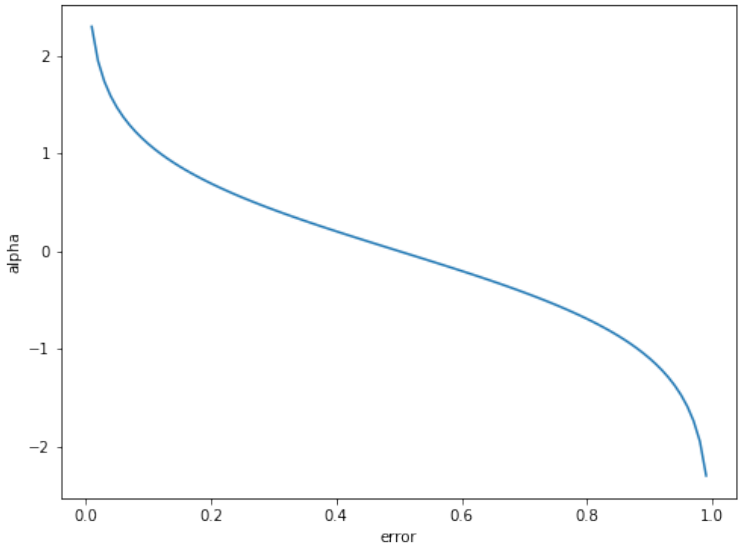
To explain a little how AdaBoost works, consider the following binary classification problem with ten random training examples. In Figure 2 we can see that the training examples belong to a problem with two dimensional data so it can be represented a plane. However, AdaBoost is general enough to work in larger dimensions, as it can be seen after. For this example a random set of 10 points for which half are classified below the thresehold and the other half over the threshold is used to compute an AdaBoost classifier in five iterations.

Figure 2: AdaBoost Example



The left column of Figure 2, represents each of the different iterations, which represent each of the different weak classifiers, and the column at the right, represents the strong classifier that would be obtained at that particular iteration. We can see the first weak classifier, which in this case is a horizontal line, makes a good job classifying the negative, blue cases, but not so well classifying the positive cases at t=1. In this case, as it is only one iteration, the weak and strong learners are the same.
When t=2, we can see that the weak learner is equal to saying that all the points are positive, for which, it gets all the positive right and all the negatives wrong, this weak classifier in particular is as good as a random uniform distribution classifier would be, hence is the weakest learner possible for this algorithm. For this case also, there is no significant contibution to the strong learner.
For t=3, it can be noted that the strong learner has improved up to correctly classifying all but one of the data points, this lead by the fact that the weak classifier gets 7 out of 10 of points correctly classified. Moving to t=4, it can be noted that there is a setback on the accuracy of the model. Finally when t=5, the strong classifier composed by the weighted weak learners, correctly classifies all the data points. The strong classifier that results from creating an assembly with weak classifiers. The way we use these weak classifiers is through a majority decision, as explained below. When we want to classify a new example, we ask each of our previous weak classifiers their opinion. If the majority thinks that the new example is positive, then the decision of the strong classifier will be that it is a positive example. If, on the contrary, the majority think that the example in question is negative, the opinion of the strong classifier will then be that it is a negative example.
It is important to mention that each weak classifier will have an error assigned during training. This error will depend on the number of training examples classified incorrectly. Also, depending on their assigned error, each classifier will have a respective level of confidence. The less error, the greater the confidence we will have in your opinion, and vice versa. This means that during the calculation of the final decision of the strong classifier, some weak classifiers will have greater influence than others, as their opinion will be worth more than the opinion of others. Figure 3, shows the relation between a coefficient alpha and the error. Alpha can be interpreted as the weight each of the weak learners will have in the making of the strong classifier. It is to note that low error leads to a greater alpha, but also a large amount of error, can lead to a large magnitude of alpha, which can also be useful if the weak learner is interpreted as the negative.
The graphic example used here to illustrate the operation of AdaBoost lets us see how intuitive this meta-algorithm is. In general, hundreds of weak classifiers are required for non-trivial problems to achieve a minimum acceptable performance. From the theoretical point of view it is said that AdaBoost is robust to over-training, this means that we can train our model for many iterations, without worrying about an overfit. Overfitting is a common problem in several supervised learning methods. When they train for many iterations, it can happen that our algorithm learns to perfectly classify the examples we use during training but, at the same time, it becomes bad to classify the new examples we provide, that is, it does not have a good ability to generalize. This is definitely bad because the main objective of a classifier is precisely that it has a great capacity to generalize.

Figure 3: Alpha and Error



AdaBoost is a meta-algorithm because the weak classifiers that comprise it can be generated with any machine learning algorithm for classification[Mohri et al., 2018], for example, the logistic regression algorithm, decision trees, the perceptron or even simple straight, flat or hyper-flat. In the example illustrated above, straight lines were used. Something important that is always sought when choosing weak classifiers is that they are really simple and easy to train. In general, the only condition that is required for AdaBoost to work is that weak classifiers have an error of less than 0.5 in their classification, that is, they are slightly better than throwing a coin in the air. Hence the name of weak classifier. At present, with the popular deep neural networks that have become, AdaBoost has fallen relatively disused, but its applications to image processing [Wang, 2012], and because it is such an intuitive and robust method makes it well worth knowing.

3 Description of the database

3.1 Forest Cover Type Database

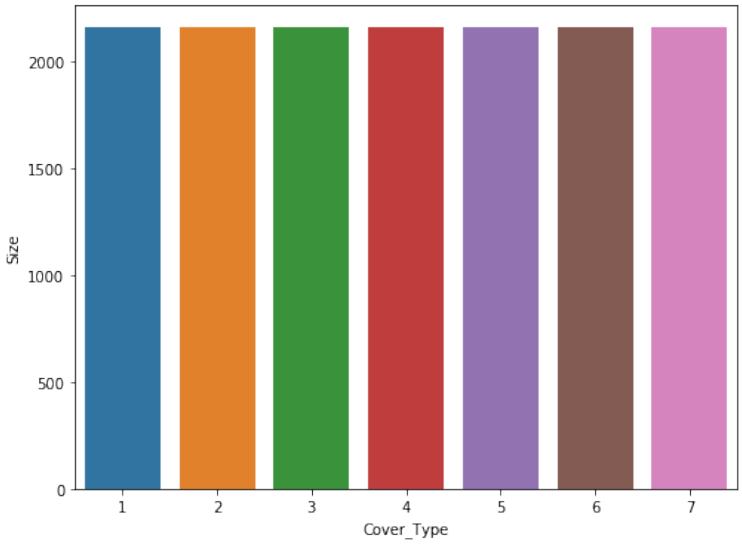
The Forest Type Cover Database taken from[Bache and Lichman, 2013], comes from a study that includes wilderness areas located in the Roosevelt National Forest of northern Colorado in the United States. Each of the total 15120 observations on the database, and that represents a total of 55 features, corresponds to a 900m2 squared patches that were determined by the United States Forest Service(USFS). Independent variables were derived from data originally obtained from US Geological Survey (USGS) and USFS data (Blackard and Denis, 2000). Table 1 resumes the variable names, data types, unit of measurement and a short description of each of the variables.

Table 1: Variable Description

Name	Data Type	Measurement	Description
ID	Quantitative	1 to 15120	Index Variable
Elevation	Quantitative	meters	Elevation in meters
Aspect	quantitative	azimuth	Aspect in degrees azimuth
Slope	quantitative	degrees	Slope in degrees
Horizontal_Distance_To_Hydrology	Quantitative	meters	Horz Dist to nearest surface water features
Vertical_Distance_To_Hydrology	Quantitative	meters	Vert Dist to nearest surface water features
Horizontal_Distance_To_Roadways	Quantitative	meters	Horz Dist to nearest roadway
Hillshade_9am	Quantitative	0 to 255 index	Hillshade index at 9am, summer solstice
Hillshade_Noon	Quantitative	0 to 255 index	Hillshade index at noon, summer solstice
Hillshade_3pm	Quantitative	0 to 255 index	Hillshade index at 3pm, summer solstice
Horizontal_Distance_To_Fire_Points	Quantitative	meters	Horz Dist to nearest wildfire ignition points
Wilderness_Area (4 binary columns)	Qualitative	0 (absence) or 1 (presence)	Wilderness area designation
Soil_Type (40 binary columns)	Qualitative	0 (absence) or 1 (presence)	Soil Type designation
Cover_Type (7 types)	Integer	1 to 7	Forest Cover Type designation

Figure 4 shows the distribution of the seven different classes of Cover Types. It can be noted that they are all equally distributed with 2160 observations each, totaling the 15120 observations of the base.

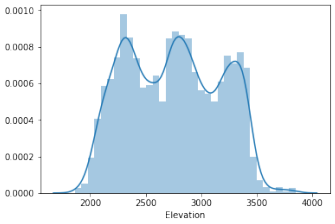
Figure 4: Distribution by Cover Type



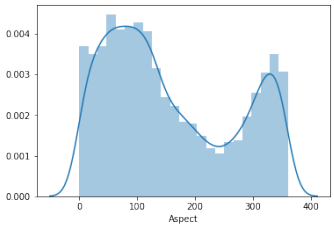
The skewness for a normal distribution is zero, and any symmetric data should have a skewness near zero. Negative values for the skewness indicate data that are skewed left and positive values for the skewness indicate data that are skewed right. By skewed left, it means that the left tail is long relative to the right tail. Similarly, skewed right means that the right tail is long relative to the left tail. Figure 5 shows the skewness for all the numeric variables of the database. Some of the numeric variables are heavily skewed, and this could possibly lead to bias in the estimation, so it would be wise to correct them on a later stage.

Figure 5: Distribution of numerical features

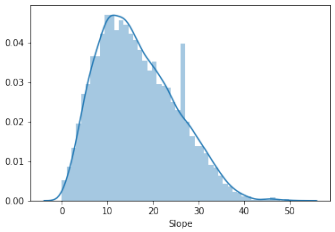
(a) Elevation



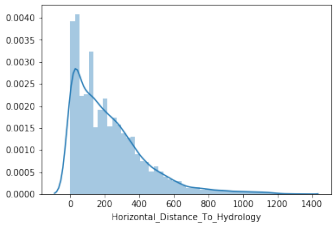
(b) Aspect



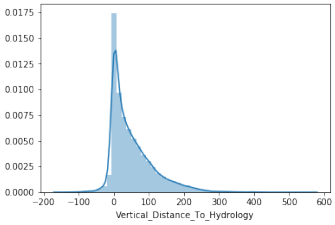
(c) Slope



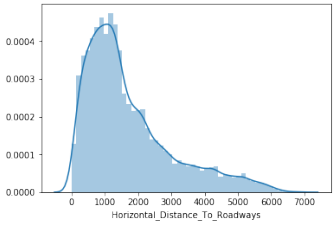
(d) Horizontal distance to hydrology



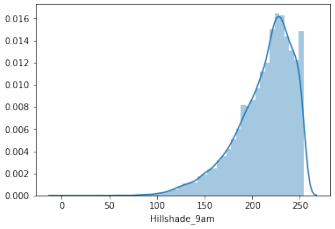
(e) Vertical distance to hydrology



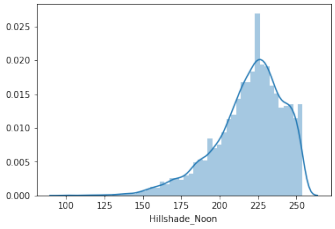
(f) Horizontal distance to roadways



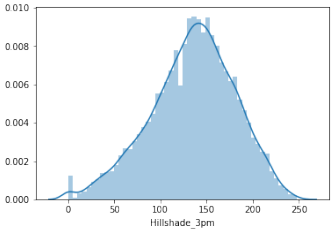
(g) Hillshade at 9 a.m.



(h) Hillshade at noon



(i) Hillshade at 3 p.m.



(j) Horizontal distance to firepoints

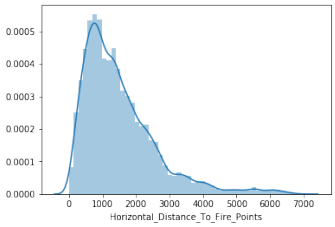
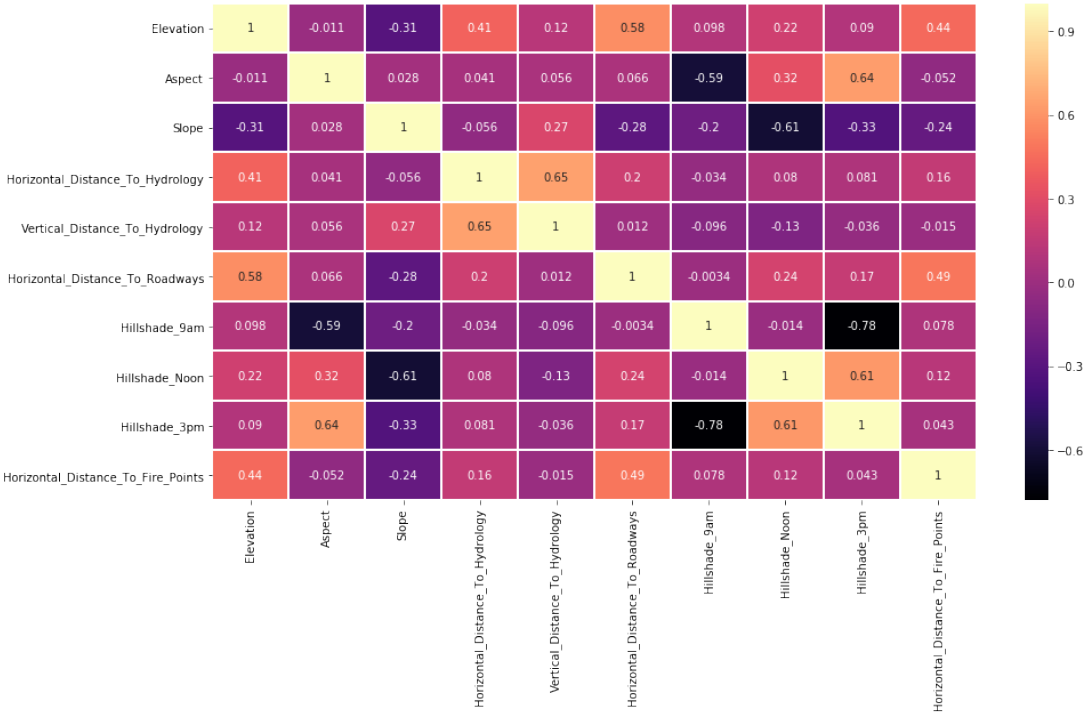


Figure 6 shows a heatmap of correlations between the numeric variables of the base. From here it is to noticed that some variables are strongly correlated between them, such as the Hillshade_non and Hillshade_3pm, Slope and Hillshade_noon, Hillshade_noon and Hillshade_9am, and Vertical_Distance_to_Hidrology and Horizontal_Distance_to_Hidrology.

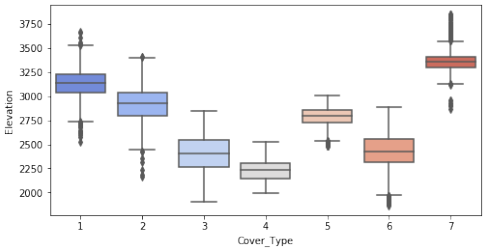
Figure 6: Correlation heatmap between numeric variables



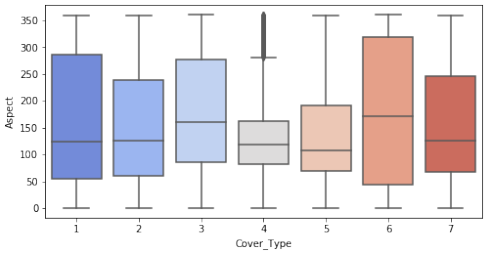
The distribution of these numeric variables is contrasted by the Cover Type in Figure 4, where it can be seen that there are variables such as Elevation, Aspect, or Horizontal_distance_to_hidrology that are "well behave" since there is variability between them and the Cover_Type, but there are others such as Vertical_distance_to_hidrology, in which it practically the behavior is the same for each of the Cover_Types.

Figure 7: Frequency distribution of numeric variables by Cover Type

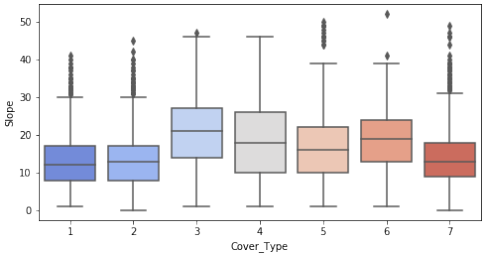
(a) Elevation



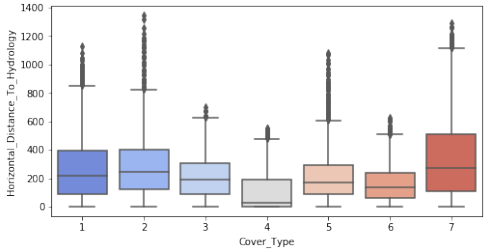
(b) Aspect



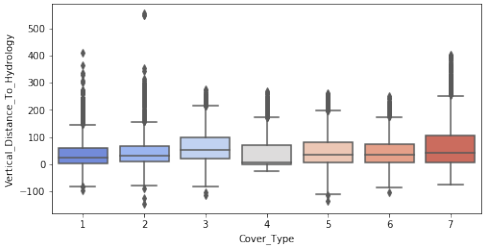
(c) Slope



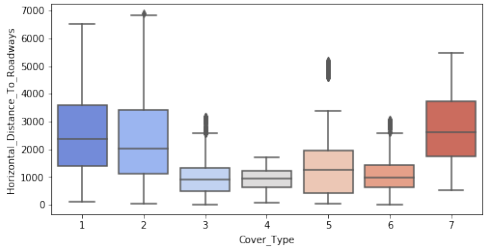
(d) Horizontal distance to hydrology



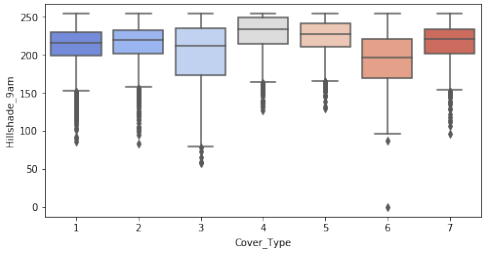
(e) Vertical distance to hydrology



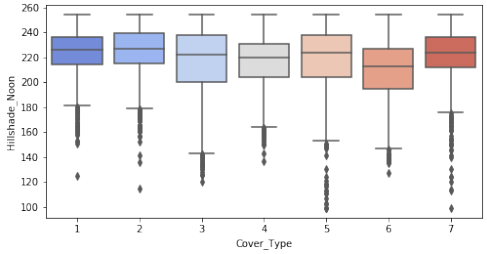
(f) Horizontal distance to roadways



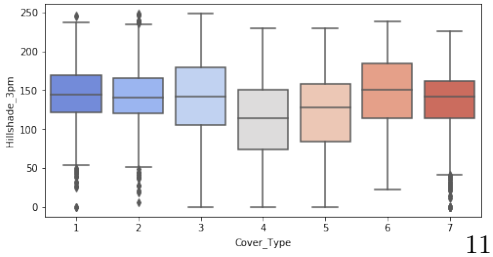
(g) Hillshade 9.a.m.



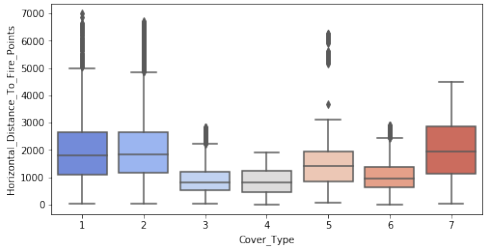
(h) Hillshade noon



(i) Hillshade 3p.m.



(j) Horizontal distance to fire points



4 Application

4.1 Experiment and results

The Cover_Type variable, which represents the seven different types of covers, was used to get the seven different target variables that will be used as classifiers of each of the classes. First the database is split between two arrays, one containing only the data points (called X), and the other containing only the data labels. A different array corresponding to each label of the Cover Type variables are created.
After this, the arrays are splitted into training and tests sets with the proportion of 70% to training and 30% to testing. Again this is done for each of the corresponding new dummy labels.

Table 2: Accuracy of classifiers

Cover_Type	Number of Iterations				
	t= 3	t=5	t=7	t=10	t=15
1	87.47 %	87.47 %	87.63 %	87.70 %	87.76 %
2	87.20 %	87.20 %	87.20 %	85.28 %	86.77 %
3	86.38 %	86.38 %	87.10 %	87.24 %	87.17 %
4	95.37 %	95.73 %	96.06 %	96.06 %	96.10 %
5	86.71 %	86.71 %	88.69 %	89.38 %	90.41 %
6	86.21 %	86.84 %	86.54 %	85.22 %	86.67 %
7	94.74 %	94.74 %	94.78 %	95.11 %	95.44 %

Table 3: Precision of classifiers

Cover_Type	Number of Iterations				
	t= 3	t=5	t=7	t=10	t=15
1	97.59 %	97.59 %	97.86 %	94.60 %	97.28 %
2	99.92 %	99.92 %	99.92 %	99.39 %	95.68 %
3	95.69 %	95.69 %	97.19 %	97.00 %	94.34 %
4	97.25 %	96.37 %	97.06 %	97.17 %	97.25 %
5	98.30 %	98.30 %	98.69 %	97.37 %	98.38 %
6	91.91 %	92.72 %	92.30 %	90.11 %	92.41 %
7	96.83 %	96.83 %	96.83 %	96.83 %	97.15 %

For each of the seven dummy variables we train several different AdaBoost classifiers, corresponding to a different number of iterations of the algorithm, with 3, 5, 7, 10, and 15 iterations, 35 different predictors are calculated. For each of the predictors we calculate the accuracy of the classifier and the confusion matrix, which is then used to describe the accuracy of our predictors. Table 2and Table 3 indicate respectively the accuracy and precision for each Cover Type Classifier according to the number of iterations. It is to notice also that the Classifiers 4, and 7 show the largest levels of accuracy, meanwhile Classifiers 1, 2, 4, and 7 show the largest levels of precision. In general, the accuracy for all the classifiers seems to reach its top at the most iterations, but in the case of the precision, Classifier 2 and 3 loose a significant amount of it, when going from 10 to 15 iterations. Also in general, it seem

that AdaBoost, worked as expected, and was able to produce decent classifiers for most of the model specifications. Also, it was noted that the size of the parameter t of the AdaBoost Classifier is relevant on the regarding the predictive power of the model.

5 Conclusions

Throughout the realization of this project it was possible to reach the following conclusions:

- The one vs all encoding implementation of AdaBoost works as a powerful algorithm to create classifiers based on multi categorical models.
- Even thought the numerical variables, most numerical variables from the model where far from normal, and showed a large amount of colinerarity, AdaBoost managed to provide a good set of classifiers.
- AdaBoost can succesfully be used to tackle other problems such as non-linearity.
- The model seems in general, to improve after each iteration.

6 Annex

6.1 Python Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import os
import array as arr
import seaborn as sns
from typing import Optional
import warnings
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_gaussian_quantiles
from sklearn import datasets
from sklearn import preprocessing
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import confusion_matrix

##Getting the data

data= pd.read_csv('forest-cover-type.csv')


#Checking for missing values


data.isnull().sum()

#Data exploring

data.describe()

class_dist=data.groupby('Cover_Type').size()
class_label=pd.DataFrame(class_dist,columns=['Size'])
plt.figure(figsize=(8,6))
sns.barplot(x=class_label.index,y='Size',data=class_label)

#Histograms

cont_data=data.loc[:,'Elevation':'Horizontal_Distance_To_Fire_Points']

binary_data=data.loc[:,'Wilderness_Area1':'Soil_Type40']

Wilderness_data=data.loc[:,'Wilderness_Area1': 'Wilderness_Area4']

Soil_data=data.loc[:,'Soil_Type1':'Soil_Type40']

for i, col in enumerate(cont_data.columns):
```

```
    plt.figure(i)
    sns.distplot(cont_data[col])

#Distribution plots

data['Cover_Type']=data['Cover_Type'].astype('category') #To convert target class into category

for i, col in enumerate(cont_data.columns):
    plt.figure(i,figsize=(8,4))
    sns.boxplot(x=data['Cover_Type'], y=col, data=data, palette="coolwarm")

#Heatmap
plt.figure(figsize=(15,8))
sns.heatmap(cont_data.corr(),cmap='magma',linecolor='white',linewidths=1,annot=True)

#Alpha Error

alpha = lambda x: 0.5 *np.log((1.0-x)/x)
error = np.arange(0.01, 1.00, 0.01)

plt.figure(figsize=(8,6))
plt.xlabel('error')
plt.ylabel('say')
plt.plot(error, alpha(error))
plt.show()

# Decision stump used as weak classifier
class DS():
    def __init__(self):
        self.polarity = 1
        self.feature_idx = None
        self.threshold = None
        self.alpha = None

    def predict(self, X):
        n_samples = X.shape[0]
        X_column = X[:, self.feature_idx]
        predictions = np.ones(n_samples)
        if self.polarity == 1:
            predictions[X_column < self.threshold] = -1
        else:
            predictions[X_column > self.threshold] = -1

        return predictions

# AdaBoost Classifier

class AB():
```

```
def __init__(self, n_clf=5):
    self.n_clf = n_clf

def fit(self, X, y):
    n_samples, n_features = X.shape

    # Initialize weights to 1/N
    w = np.full(n_samples, (1 / n_samples))

    self.clfs = []
    # Iterate through classifiers
    for _ in range(self.n_clf):
        clf = DS()

        min_error = float('inf')
        # greedy search to find best threshold and feature
        for feature_i in range(n_features):
            X_column = X[:, feature_i]
            thresholds = np.unique(X_column)

            for threshold in thresholds:
                # predict with polarity 1
                p = 1
                predictions = np.ones(n_samples)
                predictions[X_column < threshold] = -1

                # Error = sum of weights of misclassified samples
                misclassified = w[y != predictions]
                error = sum(misclassified)

                if error > 0.5:
                    error = 1 - error
                    p = -1

                # store the best configuration
                if error < min_error:
                    clf.polarity = p
                    clf.threshold = threshold
                    clf.feature_idx = feature_i
                    min_error = error

        # calculate alpha
        EPS = 1e-10
        clf.alpha = 0.5 * np.log((1.0 - min_error + EPS) / (min_error + EPS))

        # calculate predictions and update weights
        predictions = clf.predict(X)

        w *= np.exp(-clf.alpha * y * predictions)
```

```
            # Normalize to one
            w /= np.sum(w)

            # Save classifier
            self.clfs.append(clf)

    def predict(self, X):
        clf_preds = [clf.alpha * clf.predict(X) for clf in self.clfs]
        y_pred = np.sum(clf_preds, axis=0)
        y_pred = np.sign(y_pred)

        return y_pred

     def __init__(self):
        self.stumps = None
        self.stump_weights = None
        self.errors = None
        self.sample_weights = None

def _check_X_y(self, X, y):
            assert set(y) == {-1, 1},
        return X, y


#parameters for the model

cover = list(range(1,8))
T= (3, 5, 7, 10, 15)
coverdummies = pd.DataFrame(pd.get_dummies(data['Cover_Type']))

X= data.drop(columns=['Id', 'Cover_Type']).values
X = X.astype('int')

##Model implementation

for i in cover:
    def accuracy(y_true, y_pred):
        accuracy = np.sum(y_true == y_pred) / len(y_true)
        return accuracy
    y= coverdummies[i].values
    y= y.astype('int')
    y[y==0] = -1
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=5)
#Adaboost classification with 5 weak classifiers
    for j in T:
        clf = AB(n_clf=j)
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        acc= accuracy(y_test, y_pred)
```

```
print(acc)
print(i)
print(j)
print(confusion_matrix(y_test, y_pred))
```

7 Bibliography

References

[Bache and Lichman, 2013] Bache, K. and Lichman, M. (2013). Forest cover data.

[Lantz, 2015] Lantz, B. (2015). *Machine Learning with R.* Packt Publishing Ltd., Birmingham, UK.

[Mohri et al., 2018] Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2018). *Foundations of Machine Learning.* The MIT Press.

[Rudin et al., 2007] Rudin, C., Schapire, R. E., and Daubechies, I. (2007). Analysis of boosting algorithms using the smooth margin function. *The Annals of Statistics*, 35(6):2723–2768.

[Shalev-Shwartz and Ben-David, 2013] Shalev-Shwartz, S. and Ben-David, S. (2013). *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press.

[Wang, 2012] Wang, R. (2012). Adaboost for feature selction and its relation with svm, a review. *Physics Procedia*, 25(1):800–807.