Time to visualize the full iceberg !

Boosting and AdaBoost clearly explained

A Visual Explanation

Maël Fabien Follow

Feb 14, 2019 · 8 min read ★

Boosting techniques have recently been rising in Kaggle competitions and other predictive analysis tasks. I'll try to explain the concepts of Boosting and AdaBoost as

clearly as possible. The initial article was published on my personal blog : https://maelfabien.github.io/machinelearning/adaboost/

In this article, we'll cover :

- A quick recap of bagging
- Limits of bagging
- Detailed concept of Boosting
- Efficience of boosting in computation
- A code example

I have recently created a dedicated GitHub repository for some tutorials I'm following and/or building. I have also added ML recaps :

https://github.com/maelfabien/Machine_Learning_Tutorials

. . .

I. The limits of Bagging

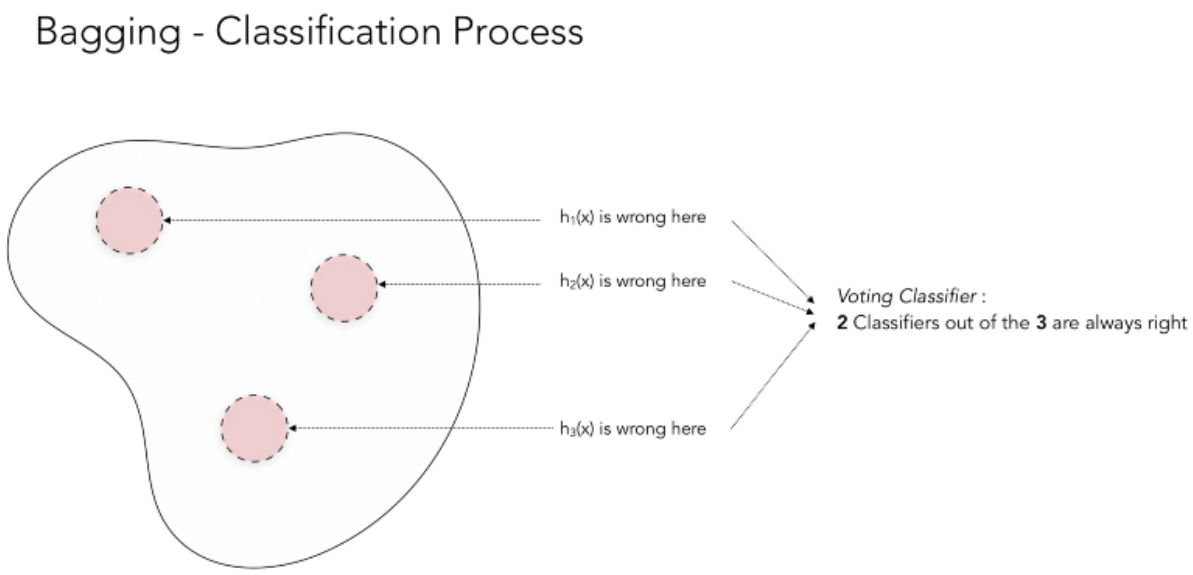
For what comes next, consider a binary classification problem. We are either classifying an observation as 0 or as 1. This is not the purpose of the article, but for the sake of clarity, let's recall the concept of bagging.

Bagging is a technique that stands for "Bootstrap Aggregating". The essence is to select T bootstrap samples, fit a classifier on each of these samples, and train the models in parallel. Typically, in a Random Forest, decision trees are trained in parallel. The results of all classifiers are then averaged into a bagging classifier :

$$H_T(x) = 1/T \sum_t h_t(x)$$

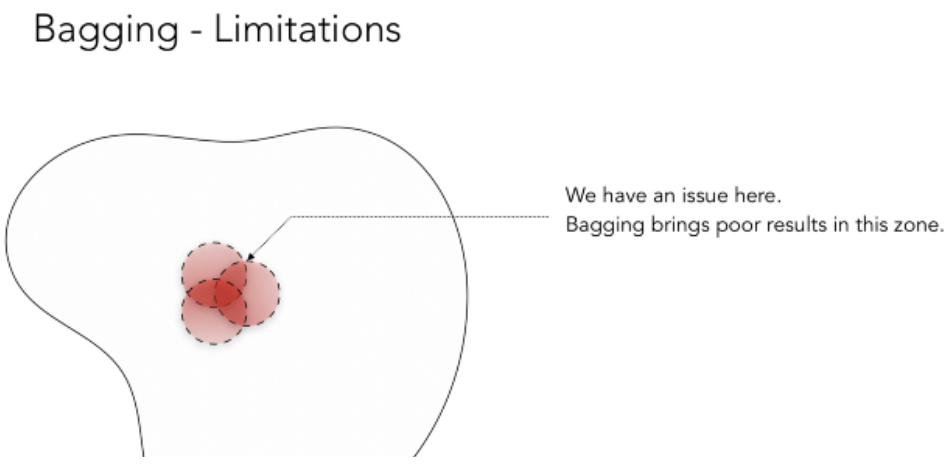
Formula for a Bagging Classifier

This process can be illustrated the following way. Let's consider 3 classifiers which produce a classification result and can be either right or wrong. If we plot the results of the 3 classifiers, there are regions in which the classifiers will be wrong. These regions are represented in red.



Example case in which Bagging works well

This example works perfectly, since when one classifier is wrong, the two others are correct. By voting classifier, you achieve a great accuracy ! But as you might guess, there's also cases in which Bagging does not work properly, when all classifiers are mistaken in the same region.



For this reason, the intuition behind the discovery of Boosting was the following :

- instead of training parallel models, one needs to train models sequentially
- and each model should focus on where the previous classifier performed poorly

. . .

II. Introduction to Boosting

a. Concept

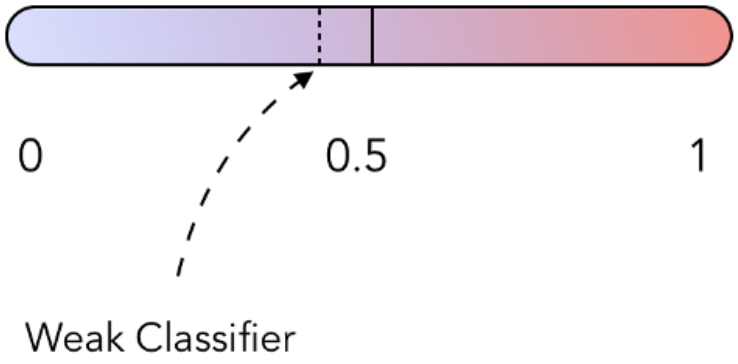
The intuition described above can be described as such :

- Train the model h1 on the whole set
- Train the model h2 with exaggerated data on the regions in which h1 performs poorly
- Train the model h3 with exaggerated data on the regions in which h1 ≠ h2
- …

Instead of training the models in **parallel**, we can train them **sequentially**. This is the essence of Boosting !

Boosting trains a series a low performing algorithms, called weak learners, by adjusting the error metric over time. Weak learners are algorithms whose error rate is slightly under 50% as illustrated below :

Classifier error rate

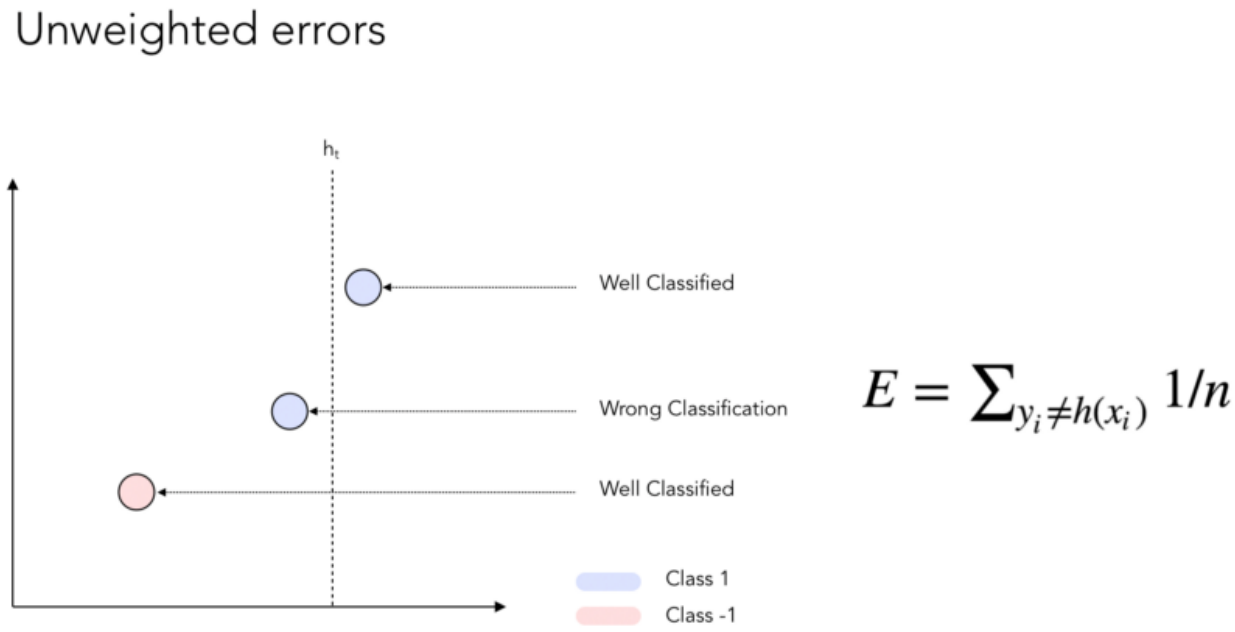


A weak classifier, achieving just under 50% error rate

b. Weighted errors

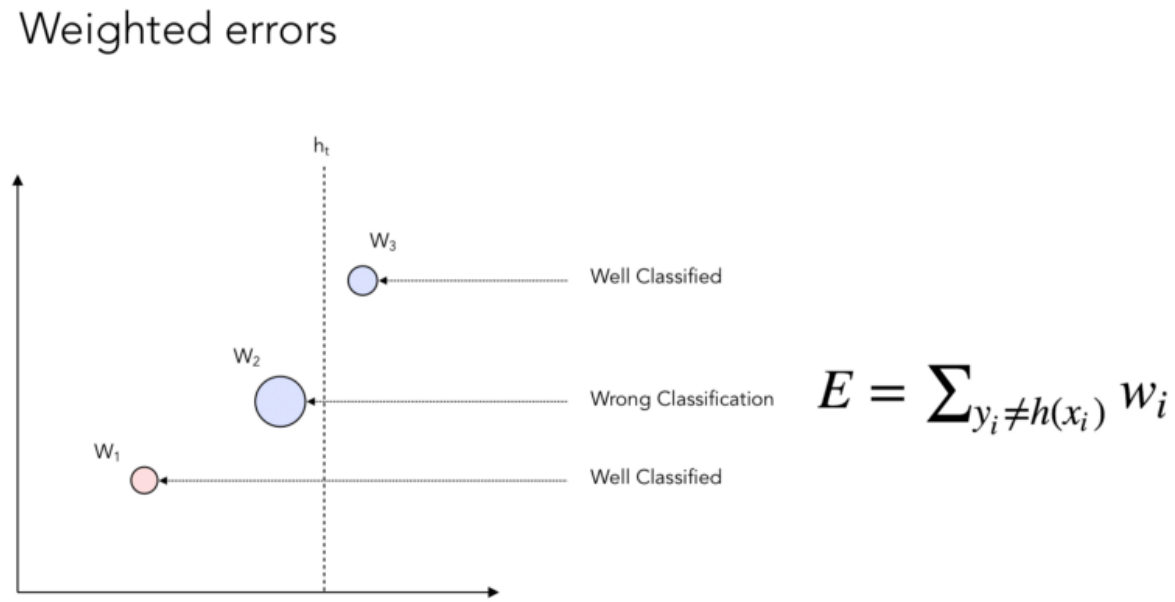
How could we implement such classifier ? By weighting errors throughout the iterations ! This would give more weight to regions in which the previous classifiers performed poorly.

Let's consider data points on a 2D plot. Some of them will be well classified, others won't. Usually, the weight attributed to each error when computing the error rate is 1/n where n is the number of data points to classify.



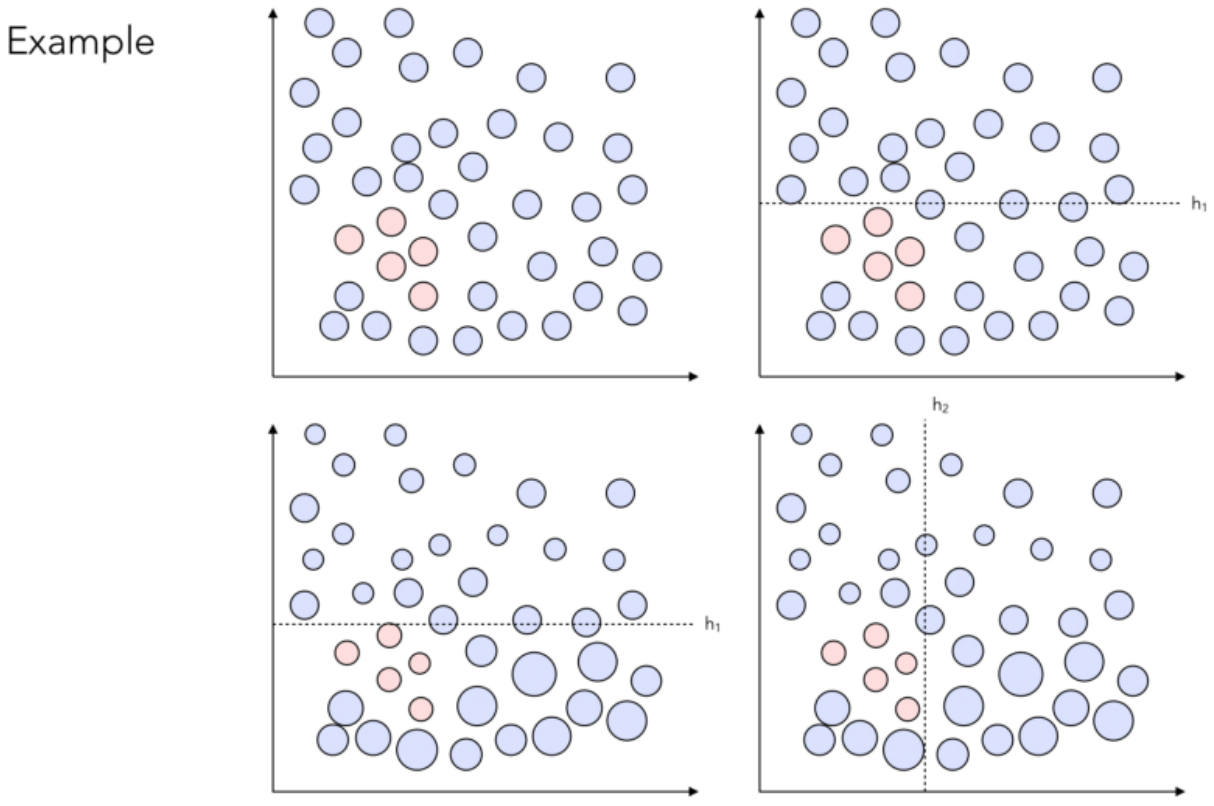
Unweighted Errors

Now if we apply some weight to the errors :



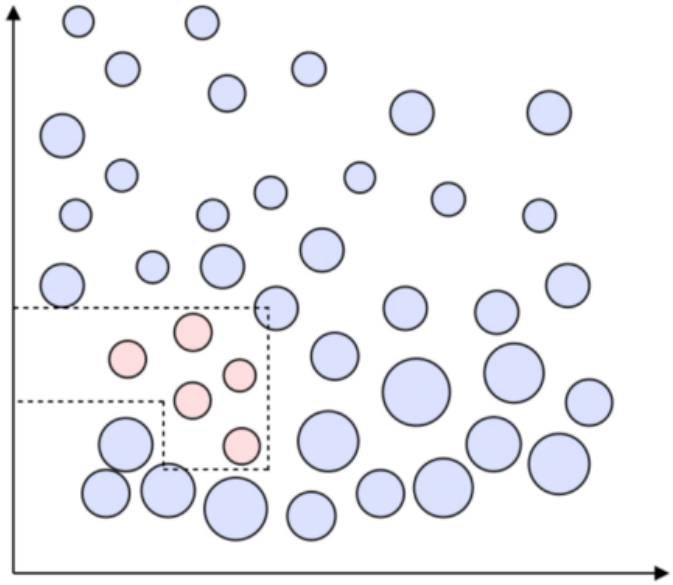
Weighted Errors

You might now notice that we give more weight to the data points that are not well classified. Here's an illustration of the weighting process :



Example of weighting process

In the end, we want to build a strong classifier that may look like this :



Strong Classifier

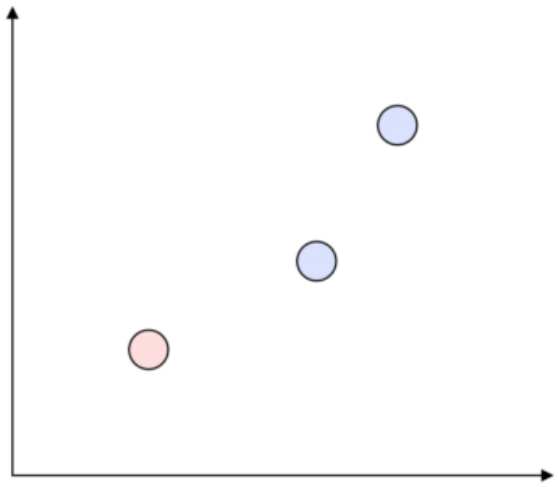
c. Tree stumps

One question you might ask, is how many classifiers should one implement in order to have it working well ? And how is each classifier chosen at each step ?

The answer lies in the definition of so-called tree stumps ! Tree stumps defines a 1-level decision tree. The main idea is that at each step, we want to find the best stump, i.e the best data split, that minimizes the overall error. You can see a stump as a test, in which the assumption is that everything that lies on one side belongs to class 1, and everything that lies on the other side belongs to class 0.

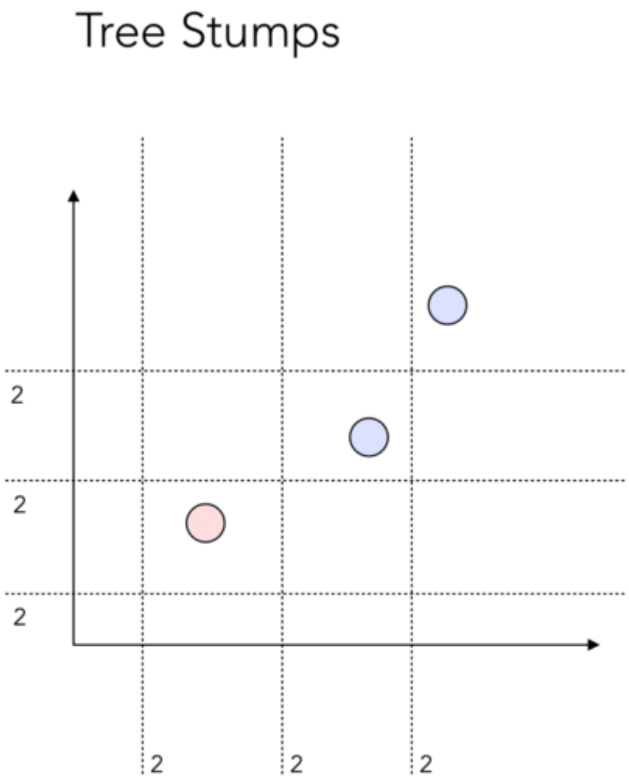
There are a lot of combinations possible for a tree stump. Let's see how many combinations we face in our simple example ? Let's take a minute to count them.

Tree Stumps



3 data points to split

Well, the answer is… 12 ! It might seem surprising, but it's rather easy to understand.



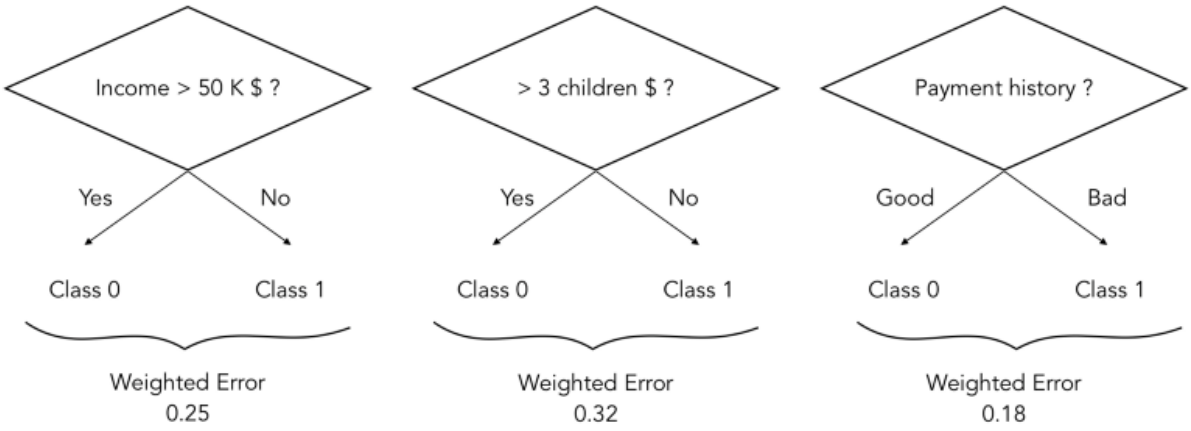
12 Stumps

There are 12 possible “test” we could make. The “2” on the side of each separating line simply represents the fact that all points on one side could be points that belong to class 0, or to class 1. Therefore, there’s 2 tests embedded in it.

At each iteration t, we will choose ht the weak classifier that splits best the data, by reducing the overall error rate the most. Recall that the error rate is a modified error rate version that takes into account what has been introduced before.

d. Finding the best split

As stated above, the best split is found by identifying at each iteration t, the best weak classifier ht, generally a decision tree with 1 node and 2 leaves (a stump). Suppose that we are trying a predict whether someone who wants to borrow money will be a good payer or not :



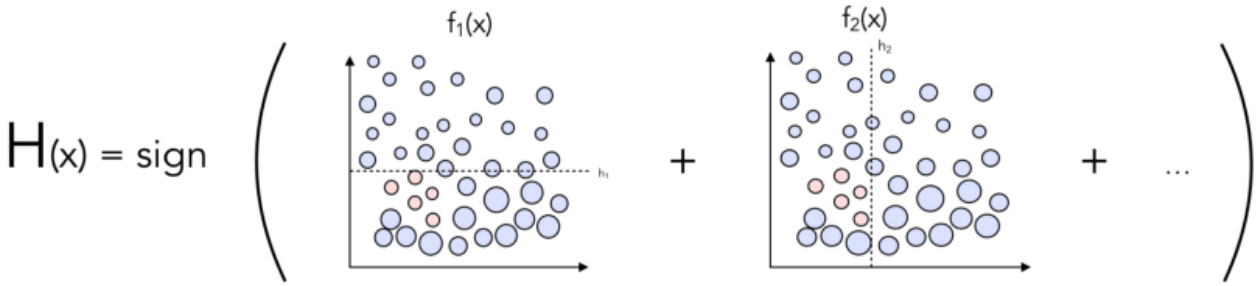
Identifying the best split

In this case, the best split at time t is to stump on the Payment history, since the weighted error resulting of this split is minimal.

Simply note that decision tree classifiers like these ones can in practice be deeper than a simple stump. This will be a hyper-parameter.

e. Combining classifiers

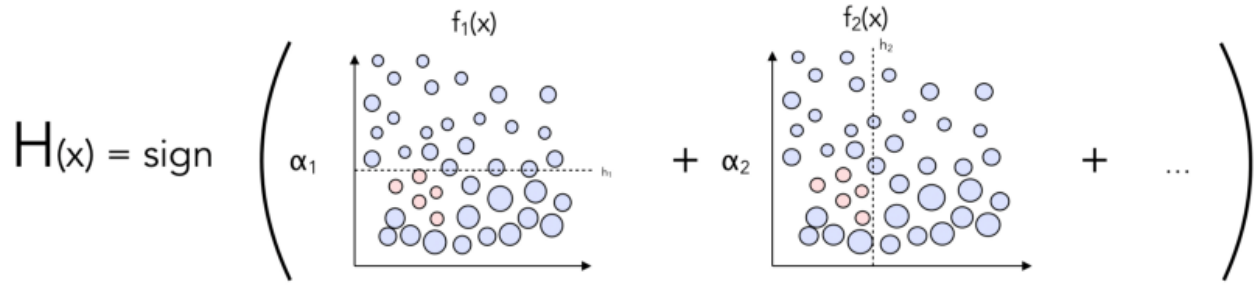
The next logical step is to combine the classifiers into a Sign classifier, and depending on which side of the frontier a point will stand, it will be classified as 0 or 1. It can be achieved this way :



Combining classifiers

Do you see any way to potentially improve the classifier ?

By adding weights on each classifier, to avoid giving the same importance to the different classifiers.



AdaBoost

f. Wrapping it up

Let's wrap up in a small pseudo-code what we covered so far.

Step 1 : Let $w_t(i) = \frac{1}{N}$ where N denotes the number of training samples, and let T be the chosen number of iterations.

Step 2 : For t in T :

a. Pick h^t the weak classifier that minimizes ϵ_t

$$\epsilon_t = \sum_{i=1}^{m} w_t(i)[y_i \neq h(x_i)] \quad (2)$$

b. Compute the weight of the classifier chosen :

$$\alpha_t = \frac{1}{2} ln \frac{1-\epsilon_t}{\epsilon_t} \quad (3)$$

c. Update the weights of the training examples w_{t+1}^i and go back to step a).

Step 3 : $H(x) = sign(\alpha^1 h^1(x) + \alpha^2 h^2(x) + \ldots + \alpha^T h^T(x))$

If you'd like to understand the intuition behind w_i^{t+1}, here's the formula :
$w_{t+1}(i) = \frac{w_t(i)}{Z} e^{-\alpha^t h^t(x) y(x)}$

Pseudo-Code

The key elements to remember from it are :

- Z is a constant whose role is to normalize the weights so that they add up to 1 !
- αt is a weight that we apply to each classifier

And we're done ! This algorithm is called **AdaBoost**. This is the most important algorithm one needs to understand in order to fully understand all boosting methods.

III. Computation

Boosting algorithms are rather fast to train, which is great. But how come they're fast to train since we consider every stump possible and compute exponentials recursively ?

Well, here's where the magic happens. If we choose properly αt and Z, the weights that are supposed to change at each step simplify to :

$$\sum_{positive} = \frac{1}{2} \text{ and } \sum_{negative} = \frac{1}{2}$$

Weights after choice of α and Z

This is a very strong result, and it does not contradict the statement according to which the weights should vary with the iterations, since the number of training samples that are badly classified drops, and their total weights is still 0.5 !

- No Z to compute
- No α
- No exponential

And there's another trick : any classifier that tries to split 2 well classified data points will never be optimal. There's no need to even compute it.

IV. Let's code !

Now, we'll take a quick look at how to use AdaBoost in Python using a simple example on a handwritten digit recognition.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import train_test_split
from sklearn.model_selection import learning_curve

from sklearn.datasets import load_digits
```

import hosted with ♡ by GitHub view raw

Let's now load the data :

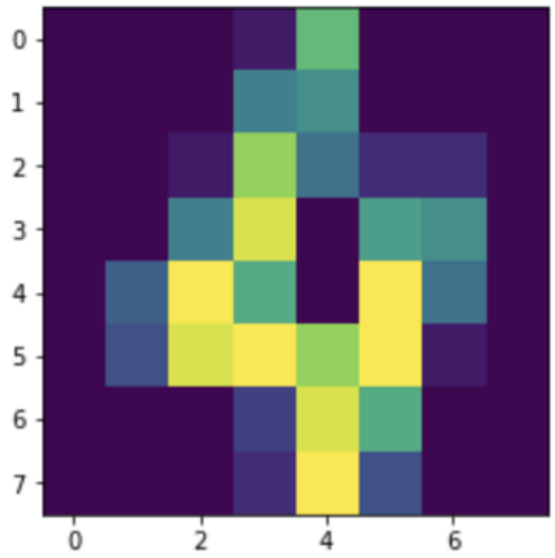
```
dataset = load_digits()
X = dataset['data']
y = dataset['target']
```

data hosted with ♡ by GitHub view raw

X contains arrays of length 64 which are simply flattened 8x8 images. The aim of this dataset is to recognize handwritten digits. Let's take a look at a given handwritten digit :

```
plt.imshow(X[4].reshape(8,8))
```

illu hosted with ♡ by GitHub view raw



8x8 image of a handwritten "4"

If we stick to a Decision Tree Classifier of depth 1 (a stump), here's how to implement AdaBoost classifier :

```
reg_ada = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1))
scores_ada = cross_val_score(reg_ada, X, y, cv=6)
scores_ada.mean()
```

depth hosted with ♡ by GitHub view raw

And it should head a result of around 26%, which can largely be improved. One of the key parameters is the depth of the sequential decision tree classifiers. How does accuracy improve with depth of the decision trees ?

```
score = []
for depth in [1,2,10] :
    reg_ada = AdaBoostClassifier(DecisionTreeClassifier(max_depth=depth))
    scores_ada = cross_val_score(reg_ada, X, y, cv=6)
```

```
        score.append(scores_ada.mean())
```

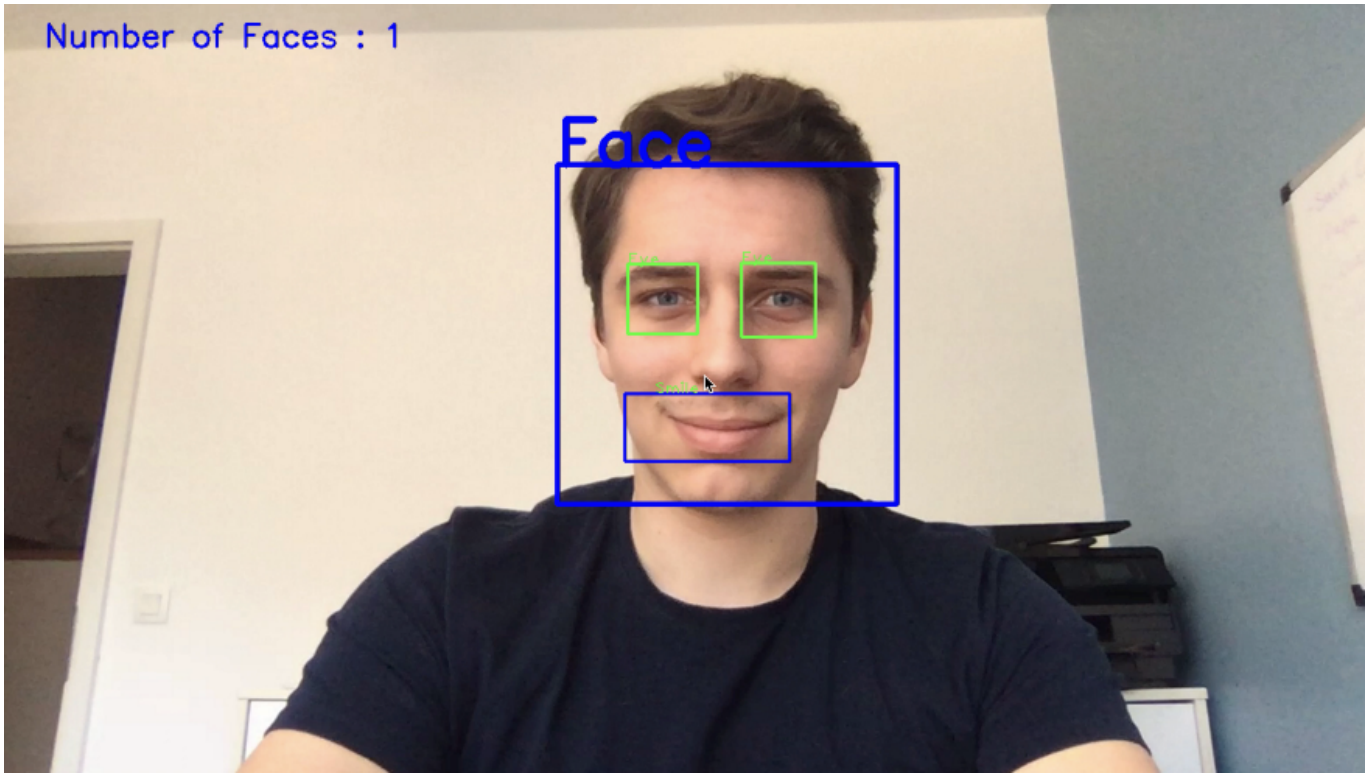
depth_full hosted with ♡ by GitHub view raw

And the maximal score is reached for a depth of 10 in this simple example, with an accuracy of 95.8%.

IV. Conclusion

It has been discussed whether AdaBoost overfits or not. Lately, it has been proven to overfit at some point, and one should be aware of it. AdaBoost can also be used as a regression algorithm.

AdaBoost is vastly used in face detection to assess whether there is a face in the video or not. I'll do another article on this topic soon ! In a further article, we'll also cover gradient boosting :)



I hope that this article introduced clearly the concept of AdaBoost and that it does now seem clear to you. Don't hesitate to drop a comment if you have any question or remark.

. . .

References :

[1] https://www.coursera.org/lecture/ml-classification/learning-boosted-decision-stumps-with-adaboost-bx5YA

[2] https://ru.coursera.org/lecture/ml-classification/learning-boosted-decision-stumps-with-adaboost-bx5YA

[3] https://www.youtube.com/watch?v=UHBmv7qCey4