Implementing AdaBoost in Python

Mehdi Abbassi
Data Science and Economics
University of Milan
`mehdi.abbassi@studenti.unimi.it`

February 12, 2022

Abstract

AdaBoost (**Ada**ptive **Boost**ing) is a boosting algorithm proposed by Yoav Freund and Robert Schapire. AdaBoost is boosting of weak learners, whose performance just needs to be better than random guessing. We implemented AdaBoost from scratch and studied the effects of number of weak learners, and their minimum performance margin (γ) on the training and validation accuracy. Although weak learners' performance just need to be better than random guessing, we show that for a specific number of learners, each weak learners performance affects the performance of the whole meta-algorithm. Besides, both number of weak learners and their minimum performance margin affects training time. To mitigate this problem we proposed a new hyper-parameter named β that in the cases finding weak learners is costly, reduces minimum performance margin temporarily.

***K*eywords** AdaBoost · Boosting · Machine Learning · Classification

1 Introduction

This projects is about implementing AdaBoost from scratch and evaluating its performance on the *Forest Cover Types* dataset. In the second section we will explain the AdaBoost algorithm. In the third section we will discuss about our implementation of AdaBoost. In the fourth section we will show the results of fine-tuning of hyper-parameters. And finally, in the conclusions we will discuss the complexity of our implementation of the meta-algorithm.

2 Intuition behind AdaBoost

AdaBoost is a boosting algorithm that "assumes the availability of a *base* or *weak learning algorithm* which, given labeled training examples, produces a *base* or *weak classifier*. The goal of boosting is to improve the performance of the weak learning algorithm..."[1]

The only assumption, which is called the *weak learning assumption*[2], is that the weak classifiers are at least a little bit better than random guessing on the examples on which they were trained. Since a boosting algorithm's only tool of learning from the data is through calls to the base learning algorithm, if the base learner is simply called repeatedly, always with the same set of training data, we expect same or almost the same base classifiers to be produced over and over again. Therefore, boosting algorithm should in some way or another manipulate the data that it feeds to the base learner in order to improve the base classifier.[3]

The key idea The key idea behind boosting is to choose training sets for the base learner in such a fashion as to force it to infer something new about the data each time it is called. In order to do so, we choose training sets on which the

[1] S. et Freud. *Boosting: Foundations and Algorithms*. English. Stuttgart: The MIT Press, 2012, p. 4.
[2] Ibid.
[3] Ibid.

performance of the preceding base classifiers are very poor — even poorer than their regular weak performance. So, we can expect the base learner to output a new base classifier which is significantly different from its predecessors.[4]

2.1 Algorithm

Weight distribution AdaBoost proceeds in *rounds* or iterative calls to the base learner. To choose the training sets provided to the base learner on each round, AdaBoost maintains a *distribution* over the training examples. The distribution used on the t-th round is denoted D_t, and the weight it assigns to training example i is denoted $D_t(i)$. Intuitively, this weight is a measure of the importance of correctly classifying example i on the current round. Initially, all weights are set equally, but on each round, the weights of incorrectly classified examples are increased so that, effectively, hard examples get successively higher weight, forcing the base learner to focus its attention on them.[5]

Base learner The base learner's job is to find a base classifier $h_t\colon X \to \{-1, +1\}$ appropriate for the distribution D_t. The quality of a base classifier is measured by its error *weighted* by the distribution D_t:

$$\epsilon_t \doteq \mathbf{Pr}_{i\sim D_t}[h_t(x_i) \neq y_i] = \sum_{i:h_t(x_i)\neq y_i} D_t(i). \tag{1}$$

Thus, the weighted error ϵ_t is the chance of h_t misclassifying a random example if selected according to D_t. Equivalently, it is the sum of the weights of the misclassified examples. Notice that the error is measured with respect to the same distribution D_t on which the base classifier was trained.[6]

Algorithm 1 AdaBoost algorithm

```
Data: $(x_1, y_1), \dots, (x_m, y_m)$ where $x_i \in X, y_i \in \{-1, +1\}$.
for $i = 1, \dots, m$ do
    $D_1(i) = \frac{1}{m}$
end
for $t = 1, \dots, T$ do
    Train weak learner using distribution $D_t$.
    Get weak hypothesis $h_t : X \to \{-1, +1\}$
    Aim: select $h_t$ to minimalize the weighted error:
    $\epsilon_t \doteq \mathbf{Pr}_{i\sim D_t}[h_t(x_i) \neq y_i] = \sum_{i:h_t(x_i)\neq y_i} D_t(i)$
    Choose $\alpha_t = \frac{1}{2} ln(\frac{1-\epsilon_t}{\epsilon_t})$
    for $i = 1, \dots, m$ do
        $D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times exp(-\alpha_t \times y_i \times h_t(x_i))$ where $Z_t$ is a normalization factor.
    end
end
Result: $H(x) = sign(\sum_{t=1}^{T} \alpha_t \times h_t(x))$
```

Importance of base classifier Once the base classifier h_t has been received, AdaBoost chooses a parameter α_t as in algorithm. Intuitively, α_t measures the importance that is assigned to h_t. For now, it is enough to observe that $\alpha_t > 0$ if $\epsilon_t < \frac{1}{2}$, and that α_t gets larger as ϵ_t gets smaller. Thus, the more accurate the base classifier h_t, the more importance we assign to it.[7]

Updating weight distribution The distribution D_t is next updated using the rule shown in the algorithm. First, all of the weights are multiplied either by $e^{-\alpha_t} < 1$ for examples correctly classified by h_t, or by $e^{\alpha_t} > 1$ for incorrectly classified examples. Equivalently, since we are using labels and predictions in $\{-1, +1\}$, this update can be expressed more succinctly as a scaling of each example i by $exp(-\alpha_t \times y_i \times h_t(x_i))$.[8]

[4]Freud, *Boosting*, pp. 4–5.
[5]Ibid., p. 5.
[6]Ibid., p. 6.
[7]Ibid.
[8]Ibid.

Next, the resulting set of values is renormalized by dividing through by the factor Z_t to ensure that the new distribution D_{t+1} does indeed sum to 1. The effect of this rule is to increase the weights of examples misclassified by h_t, and to decrease the weights of correctly classified examples. Thus, the weight tends to concentrate on "hard" examples.[9]

Actually, to be more precise, AdaBoost chooses a new distribution D_{t+1} on which the last base classifier h_t is sure to do extremely poorly: It can be shown by a simple computation that the error of h_t with respect to distribution D_{t+1} is exactly $\frac{1}{2}$, that is, exactly the trivial error rate achievable through simple random guessing. In this way, as discussed above, AdaBoost tries on each round to force the base learner to learn something new about the data.[10]

Minimalizing weighted error The weak learner attempts to choose a weak hypothesis h_t with low weighted error ϵ_t. We do not expect that this error will be especially small in an absolute sense, but only in a more general and relative sense; in particular, we expect it to be only a bit better than random, and typically far from zero. To emphasize this looseness in what we require of the weak learner, we say that the weak learner's aim is to *minimalize* the weighted error, using this word to signify a vaguer and less stringent diminishment than that connoted by *minimize*.[11]

The weak learning assumption then amounts to an assumption that the error of each weak classifier is bounded away from $\frac{1}{2}$, so that each ϵ_t is at most $\frac{1}{2} - \gamma$ for some small positive constant γ. In this way, each weak hypothesis is assumed to be slightly better than random guessing by some small amount, as measured by its error. On each round, the weights of incorrectly classified examples are increased so that, effectively, hard examples get successively higher weight, forcing the base learner to focus its attention on them.[12]

Final classifier After many calls to the base learner, AdaBoost combines the many base classifiers into a single *combined* or *final classifier* H. This is accomplished by a simple weighted vote of the base classifiers. That is, given a new instance x, the combined classifier evaluates all of the base classifiers, and predicts with the weighted majority of the base classifiers' predicted classifications. Here, the vote of the t-th base classifier h_t is weighted by the previously chosen parameter α_t. The resulting formula for H's prediction is as shown in the algorithm.[13]

3 Implementing AdaBoost

To implement AdaBoost, we defined two distinct classes for base learning algorithm, named `DecisionStump` and AdaBoost meta-algorithm, named `AdaBoost`. The first class, `DecisionStump`, given a dataset (`X`), labels (`y`), weights (`weights`), and γ (`gamma`), constructs a classifier that performs better than random guessing by at least γ. The second class, `AdaBoost`, given a dataset (`X`), labels (`y`), and number of base classifiers (`n_classifiers`) constructs a set of base classifiers with cardinality of `n_classifiers`.

DecisionStump The `DecisionStump` classifier has five attributes: `feature`, `threshold`, `alpha`, `gamma`, and `complexity`. `feature` and `threshold` determine the prediction (the rule is simple: label of all samples whose `feature` is less than `threshold` will be predicted as `-1` otherwise `+1`). `alpha` determines the classifier's importance. `gamma` determines the margin by which the classifier should outperform (or underperform as we see later!) random guessing. `complexity` is just meant for studying the complexity of the learning algorithm.

AdaBoost The `AdaBoost` classifier has also five attributes: `n_classifiers`, `classifiers`, `gamma`, `beta`, and `complexity`. The meta-algorithm calls the base classifier `n\classifiers` times, each time with different `weights`, and stores them in `classifiers`. `gamma` determines the margin by which the classifier should outperform random guessing and `beta` determines if the it fails in the first call to construct the base classifier, by which factor should it reduce `gamma` for the next call. `complexity` is summing all the complexities of the base classifiers.

3.1 AdaBoost

`AdaBoost` has three methods: `__init__`, `fit`, and `predict`. `__init__` initializes the attributes and also sets the seed for `NumPy`'s Random Generator. `fit` does the main task. It calls the base learner iteratively with a given weight discrete distribution (`weights`) and stores the constructed base classifiers in the `classifiers` attribute. Last but not least, `predict` is used to predict the label of the given sample. Since we used the one-vs-all encoding of binary AdaBoost,

[9]Freud, *Boosting*, p. 6.
[10]Ibid.
[11]Ibid.
[12]Ibid.
[13]Ibid.

the real prediction is done outside of the `predict` method of `AdaBoost`, and for this reason we do not use the *sign* function and the output is neither `-1` nor `+1`.

3.1.1 fit

First, we set `complexity` attribute as zero, each time we call the base learner, we will add its complexity to this attribute to have the sum of all calculations needed. Then, we store the number of samples and features in two variables (`n_samples` and `n_features`). We store unique values of features in a dictionary (`thresholds`), so each time we call the base learner we don't need to do the costly operation of finding the unique values. We also always ignore the smallest number in each feature because our prediction rule (`predictions[X_column < self.threshold] = -1` and `+1` for the rest) makes it useless, since it does not apply for none of the samples.

For the order of `gamma` we use Schapire's suggestion in a lecture that we found on YouTube[14], and increase it by factor of `gamma` as we set when we initialize an object of the `AdaBoost` class. And for the `weights`, in the first call, we use a uniform distribution. Finally, we construct a `Python set` to store the base classifiers.

```
        self.complexity = 0

        n_samples, n_features = X.shape

        # saving the unique values of all features in a Python dictionary
        thresholds = {feature: np.sort(np.unique(X[:, feature]))[1:]
                      for feature in range(n_features)}

        # we use Schapire's suggestion factored by 'gamma' when we initiate the DecisionStump
        gamma = self.gamma / np.sqrt(n_samples)

        # initiate 'weights' as a uniform distribution
        weights = np.full(n_samples, (1/n_samples))

        self.classifiers = set()
```

Then we make a copy of `gamma` as `gamma_`, because as we will see, on each call we may need to change the `gamma_` temporarily and later we will need to restore the initial value. Then we iterate `n_classifiers` times, each time construct a (`DecisionStump`) object named `base_classifier` as our base learner with `gamma_` and `complexity`, then train it (`fit`) with given `X`, `y`, and `weights` (we also provide `n_samples thresholds` to avoid computational cost). On each call, we add base learner's `complexity` to `AdaBoost`'s `complexity`.

Since we are looking to construct a base classifier that outperform (or underperform!) random guessing by `gamma_`, we may fail to do so. In this case we multiply `gamma_` by factor of `beta` which is less than one and iterate again this time with smaller `gamma_`. If we succeed constructing `base_classifier` we store the classifier's prediction as `y_predict` and update `weights` accordingly (the weights of incorrectly classified examples are increased so that, effectively, hard examples get successively higher weight, forcing the base learner to focus its attention on them). Last, we reset `gamma_` and add `base_classifier` to `classifiers`.

```
        # the value of 'gamma_' could change, but we want to keep the value of 'gamma'
        gamma_ = gamma

        # iterate n_classifiers times
        for _ in range(self.n_classifiers):
            # initiate a 'DecisionStump' object
            base_classifier = DecisionStump(gamma=gamma_, complexity=self.complexity)

            # train the base classifier
            base_classifier = base_classifier.fit(X, y, weights, n_samples, thresholds)

            # update the 'complexity' of the 'AdaBoost'
            self.complexity = base_classifier.complexity

            # if the base learner was not succeed to construct a base classifier
            if base_classifier.feature is None:
```

[14]The video available at `https://youtu.be/L6BlpGnCYVg?t=1702`

```
            gamma_ *= self.beta # decrease ‘gamma_’ by factor of ‘beta’
            continue # try to train a new base classifier with smaller ‘gamma_’

        # assign the predictions
        y_predict = base_classifier.predict(X)

        # update the ‘weights’
        weights *= np.exp(-1 * base_classifier.alpha * y * y_predict)
        weights /= np.sum(weights) # normalise the ‘weights’

        # reset the ‘gamma_’ with the original value of ‘gamma’
        gamma_ = gamma

        # add the constructed base classifier
        self.classifiers.add(base_classifier)
```

3.1.2 predict

In order to predict a label, we use the weighted sum of all base classifiers. In the binary setup, at the end we should use *sign* function to see in the tug-of-war between the two label, which side has more power! But since in this project we used one-vs-all encoding, we will train one `DecisionStump` for each label and at the end we will see which classifier has the more power to choose the corresponding label as the prediction; for this reason the use of *sign* function would hide the power of each boosting classifier.

```
    def predict(self, X):
        return np.sum(base_classifier.alpha * base_classifier.predict(X)
                      for base_classifier in self.classifiers)
```

3.2 DecisionStump

`DecisionStump` has three methods: `__init__`, `fit`, and `predict`. `__init__` initializes the attributes. `fit` constructs a weak classifier in such a way that its prediction score with respect to `weights` outperform random guessing by `gamma`. Finally, `predict` predicts the label based on a simple rule.

3.2.1 fit

In order to construct a classifier, `fit` method iterates over the features, adds one to `complexity`, choose the `X_column` as the column of `X` corresponding to `feature`, and for the `feature` randomly chooses a value from `thresholds_set`, makes a rule-based prediction, calculates `error` considering `weights`, and finally checks if its error’s distance from random guessing is larger than `gamma`, and if so, it calculates `alpha` as its importance in the boosting setup and assigns `alpha`, `feature`, and `threshold` as class attributes and returns the constructed classifier.

The interesting point is that, since we are in binary setup, there is a tug of war between the two labels, and even if we predict the labels in an opposite way, since we have to calculate `alpha` it will assign a negative importance to our base classifier and the result will be exactly the same. In case it fails to find any pair of `feature-threshold` in such a way that the corresponding `error` satisfies the required condition, it returns an empty object (so the boosting algorithm will call it again this time with easier requirements).

```
    for feature in thresholds_set.keys():

        self.complexity += 1 # increase complexity by one

        X_column = X[:, feature] # select the column corresponding to the feature

        thresholds = thresholds_set[feature] # choose the unique values of the selected feature

        threshold = np.random.choice(thresholds) # choose one of the values as our threshold

        y_predict = np.ones(n_samples) # set all predictions as +1
        y_predict[X_column < threshold] = -1 # change those whom apply the rule to -1
```

```
            incorrect = y_predict != y # find the incorrect predictions

            error = np.mean(np.average(incorrect, weights=weights, axis=0)) # calculate the error

            # construct the classifier and return it if the error is in the specified range
            if np.abs(error - 0.5) > self.gamma:
                epsilon = 1e-10
                self.alpha = 0.5 * np.log((1 - error) / (error + epsilon))
                self.feature, self.threshold = feature, threshold
                return self

        # return the empty classifier as a sign of failure
        return self
```

Since the AdaBoost needs only weak classifiers as its base classifier, we don't need to do greedy search, because it will be costly and without any specific gain. In the experiments we found out that the quality of the base classifiers (i.e., their weighted accuracy) are affecting the accuracy of the AdaBoost classifier. To control this, we used the `gamma` as the minimal weighted accuracy for base classifiers.

3.2.2 predict

`predict` method of `DecisionStump` is very simple and straightforward. There is a simple rule: if the value of the `feature` attribute is less than `threshold` attribute, then the prediction will be `-1` and `+1` otherwise.

4 Fine-tuning

In this section we will investigate the effect of different hyperparameters of AdaBoost model on the training and validation accuracy. We have three hyperparameters to investigate: number of base classifiers, gamma and beta. But we will also investigate the complexity of the training process.

4.1 Cross-validation

In order to study the effect of hyperparameters, we will use a 5-fold cross-validation. We implemented a binary setup of AdaBoost, but our dataset (*Covertype Data Set*) has multiple classes. To overcome this problem we use a one-vs-all arrangement and train one distinct boosting model for each of the classes and then for a given test sample we choose the class corresponding to the model with highest confidence (or predictive power) as our prediction.

First we need to split our dataset into training and validation sets. We use `Scikit-learn`'s `StratifiedKFold` with a given `random_state` and `n_splits` equals to five. Then we loop over each of five splits; we use $\frac{4}{5}$ of the data as training set and $\frac{1}{5}$ as validation set. Next for each split we need to read and store data samples and labels in different variables. Moreover we loop over the unique values of labels and for each label, we change it to $+1$ and the rest to -1. We train the models, one for each label, and then in the validation time, we choose the label of a classifier that returns the greatest number. To study the `n_classifiers`' effect we use two distinct values for `random_state` and we will report the averages. For the `gamma` and `beta` we use only one `random_state`.

4.2 Number of base classifiers

AdaBoost is made of base classifiers and the whole model is to boost the prediction power of the individual base classifiers. Therefore, it is reasonable to assume that the more the number of base classifiers, the more the predictive power of the boosting algorithm. But there is also the problem of overfitting: the more the number of base classifiers, the more the boosting model fits the training samples. We have to find a optimum value for `n_classifiers` in order to maximize the validation accuracy. Maximizing validation accuracy means that our boosting model generalizes well and neither underfits nor overfits the training samples.

We used 5-fold cross-validation with two distinct value for `random_state`, `gamma` equal to zero, and `beta` equal to one. In the figure 1 we see as number of base classifiers increases, the accuracy of both training and validation sets increase. But after a certain point, around 2000 base classifiers, they start to diverge but both of them still increasing. After 8000 base classifiers the validation accuracy decreases while training accuracy increases and this is a symptom of overfitting.

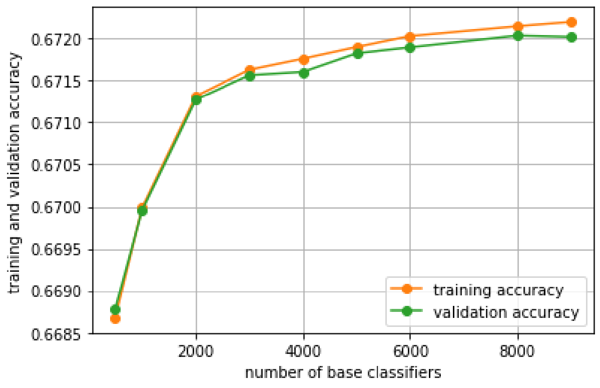

Figure 1: Train and validation accuracy by number of base classifiers.

4.3 Gamma

AdaBoost's base classifiers are weak classifiers, it means that their prediction powers are limited. `gamma` controls how much better they should at least be than a random guessing. AdaBoost boosts the predictive power of its base classifiers, each specified at predicting a region of sample space, so the predictive power of base classifiers affect the predictive power of the whole boosting model. The value we set for `gamma` does not correspond directly to the accuracy margin, but instead it is factored by a special value ($\frac{1}{\sqrt{numberofsamples}}$). Also, increasing `gamma` will increase the complexity of the training process, because finding more accurate base classifier is more costly.

We used 5-fold cross-validation with a single value for `random_state` and `n_classifiers` equal to `100`. This results in a non-smooth graph, yet the general trend is observable. We report the results for `gamma` in range `0` to `19`. The values used for `beta` are from `0` to `0.9` by step size of `0.1`. We report the average of training and validation accuracies for different values of `beta`.

In figure 2 we see in general with increase of `gamma` both accuracies increase and there is no sign of overfitting. It seems that for first four values, zero through three, the accuracies are almost the same, therefore values less than four are not recommended. From four through 12 we see an overall increasing trend. It is also noticed a quasi-flattening after `gamma` equal to 12. In figure 3 we observe the relationship between `gamma` and `complexity`. The trend is not linear and with complexity increases more rapidly for `gamma` in range 12 through 19 than it does in range 0 through 7 or even 8 through 11. While choosing larger value for `gamma` results in a better performance, it also increases the complexity of the training process. To mitigate this problem we have to consider the effect of `beta`.

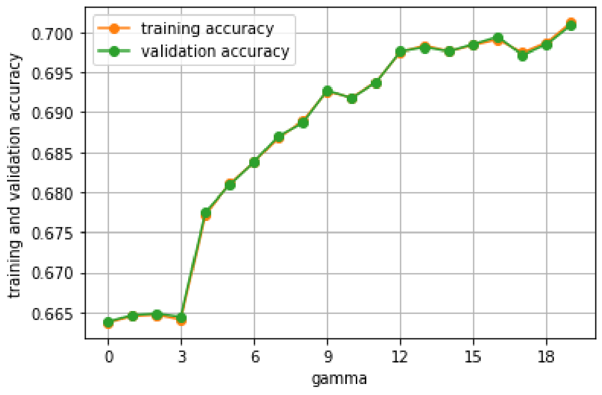

Figure 2: Train and validation accuracy by gamma with 100 base classifiers.

4.4 Beta

When AdaBoost calls a base classifier, it specifies the minimum margin that the base classifier should overperform a random guessing by that. Sometimes constructing such a base classifier is either too difficult and costly or impossible.

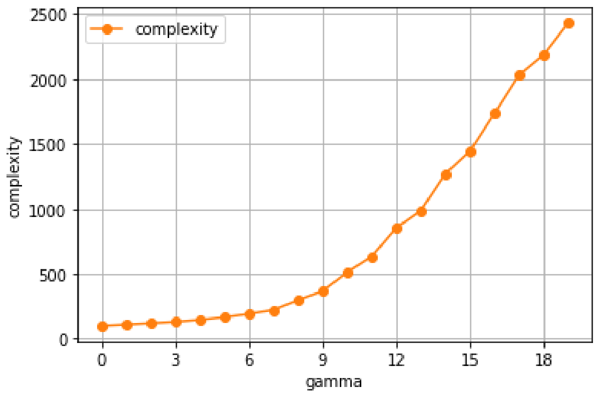

Figure 3: Complexity by gamma with 100 base classifiers.

`beta` helps to overcome this problem. When the training algorithm tries all features (and one value for each feature chosen randomly) and does not succeed in constructing a base classifier that satisfies the required margin of accuracy, the AdaBoost algorithm recalls the base learner algorithm but this time with previous margin factored by `beta` and this process continues until the base learner constructs a base learner. After constructing a base classifier, the margin of accuracy resets as original value and the effect of `beta` neutralizes. For example, if the margin of accuracy is 0.07 and `beta` is equal to 0.9, after one round if the base learner does not succeed, the margin will become 0.063 and in case it does not succeed again, it will become 0.0567 and so on. When the base learner constructs the base classifier, the margin of accuracy will become 0.07 as it was in the first place.

As it is evident in figure 4 that demonstrates the relationship between `beta` and training and validation accuracies, the trend is upward and linear. If we see the figure 5 that shows the relationship between `beta` and `complexity`, we see that the trend in upward but exponential. This shows that reducing `beta` from the side close to 1, for example reducing `beta` from 0.9 to 0.8 or 0.7, reduces accuracy linearly but it decreases `complexity` exponentially.

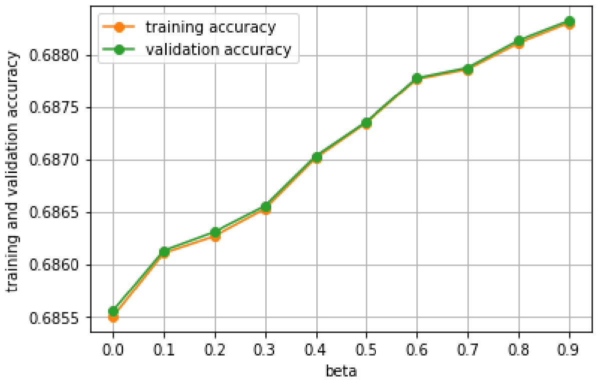

Figure 4: Train and validation accuracy by beta with 100 base classifiers.

4.5 Gamma and beta

To achieve the best performance we need to find a high validation accuracy with a reasonable complexity. Note that there is a trade-off between validation accuracy and complexity. For a specific number of base classifiers, we have two independent variables, i.e., `gamma` and `beta`, and two dependent variables, i.e., `complexity` and validation accuracy. We want to set `gamma` and `beta` in a certain way to achieve the mentioned objective.

In figure 6 we see the relationship between `gamma`, `beta` and validation accuracy and complexity. As we expected, the higher `gamma` and `beta`, the higher validation accuracy (since the boosting algorithm constructs better base classifiers); on the other hand, the higher `gamma` and `beta`, the higher complexity. As it is seen in the figures, validation accuracy is distributed more homogeneously while `complexity` has changed drastically when we approach maximum values for `gamma` and `beta`.

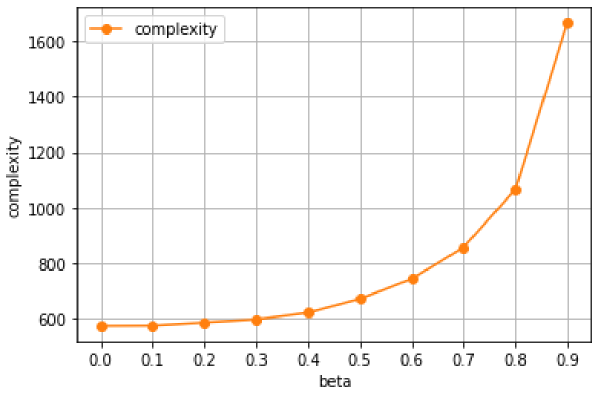

Figure 5: Complexity by beta with 100 base classifiers.

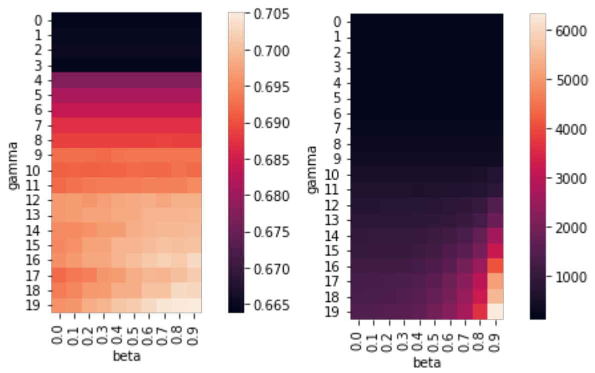

Figure 6: Validation accuracy (left) and complexity (right) by beta and gamma with 100 base classifiers.

4.6 Complexity and validation accuracy

To achieve our objective, i.e., having high validation accuracy with a reasonable complexity, we draw a graph that shows the relation between the two variable. In figure 7 (left) we see the training and validation accuracies for different complexities. As we see in the beginning with a small increase in the complexity we can increase the validation accuracy by a considerable amount, while after a certain value, the increase in validation accuracy slows down and finally plateaus. We are not interested in cases that achieve a lower validation accuracy with respect to configurations with lower complexities, therefore we remove such a points and we achieve the figure on the right. The optimal values for `gamma` and `beta` would be those corresponding to complexity around 1500 and validation accuracy around 0.705, that corresponds to `gamma` equal to 16 and `beta` to 0.6 or 19 and 0.4 respectively.

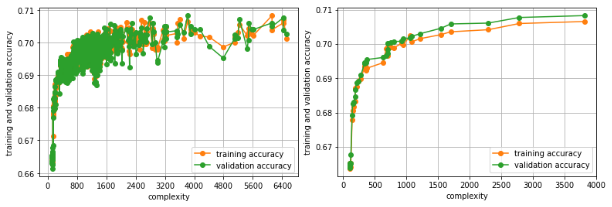

Figure 7: Validation accuracy by complexity of the training process

5 Conclusion

AdaBoost made from base classifiers that are weak in themselves but constructed in such a fashion that each one is try to predict the samples that the rest failed predicting them so far. This makes it impossible to train base classifiers in parallel. Having more base classifiers increases the predictive power of the whole boosting model up until a certain point, and after it the whole model overfits on the training data. Also, the more the predictive power of base classifiers, the more the predictive power of the boosting model. The only problem with the predictive power of the base classifiers is that training them is costly in terms of calculation, therefore we need to find a balance between the cost of training and the validation accuracy.

6 Declaration

I declare that this material, which I now submit for assessment, is entirely my 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 my work. I 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 me or any other person for assessment on this or any other course of study.

References

[1] Robert E. Schapire, Yoav Freund *Boosting: Foundations and Algorithms. English.* Stuttgart: The MIT Press, 2012