Adaboost for forest cover - Statistical methods for machine learning

Bertolotti Francesco

Jun 2019

1 Algorithm Description

1.1 Adaboost

The general idea is to get a predictor from a weighted sum of other predictors, using appropriate weights.

Let D be the joint distribution of $X \times Y = D$, where $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$ and $\mathcal{Y} = \{-1, +1\}$.
Let S be the training set, such that $S = \{(x_t, y_t)\}_{t=1}^{m}$. where each (x_t, y_t) is drawn from D.
Let l be the 0/1 loss functions, such that $l(y, \hat{y}) = \mathbb{1}\{y \neq \hat{y}\}$.
Let $l_S : H \to [0, 1]$ be the training loss, $l_S(h) = \frac{1}{m} \sum_{t=1}^{m} l(h(x_t), y_t)$, for $h \in H$.
Let $l_D : H \to [0, 1]$ be the statistical loss, $l_D(h) = \underset{X,Y \sim D}{\mathbb{E}}[l(h(X), Y)]$.
Let $h_1, h_2, \ldots, h_T$ be predictors, such that $\forall i = 1, 2, \ldots, T : h_t : \mathcal{X} \to \mathcal{Y}$.

Adaboost will output the following predictor f:

$$f(x) = sgn(\sum_{i=1}^{T} w_i h_i(x))$$

.

Adaboost finds the weights w_i such that the training error is low.

$$\begin{aligned} l_S(f) = \frac{1}{m}\sum_{t=1}^{m} l(f(x_t), y_t) = \frac{1}{m}\sum_{t=1}^{m} \mathbb{1}\{f(x_t) \neq y_t\} = \frac{1}{m}\sum_{t=1}^{m} \mathbb{1}\{f(x_t) y_t \leq 0\} = \\ \leq \frac{1}{m}\sum_{t=1}^{m} e^{-f(x_t)y_t} = \frac{1}{m}\sum_{t=1}^{m} e^{-\sum_{i=1}^{T} w_i h_i(x_t) y_t} = \frac{1}{m}\sum_{t=1}^{m}\prod_{i=1}^{T} e^{-w_i h_i(x_t) y_t} = \\ = \mathbb{E}_{t\sim\mathbb{P}_1}[\prod_{i=1}^{T} e^{-w_i h_i(x_t) y_t}] \overset{(*)}{=} \prod_{i=1}^{T} \mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_i h_i(x_t) y_t}] = \\ = \prod_{i=1}^{T} e^{-w_i}\mathbb{P}_i(h_i(x_t) y_t = +1) + e^{w_i}\mathbb{P}_i(h_i(x_t) y_t = -1) = \prod_{i=1}^{T} e^{-w_i}(1 - \epsilon_i) + e^{w_i}\epsilon_i. \end{aligned}$$

now we find w_i such that: $w_i = \underset{w}{argmin}\{e^{-w}(1 - \epsilon_i) + e^{w}\epsilon_i\}$.

$$\frac{d}{dw} e^{-w}(1 - \epsilon_i) + e^{w}\epsilon_i = 0 \implies w e^{w}\epsilon_i = w e^{-w}(1 - \epsilon_i).$$

$$\begin{cases} w = 0 \implies 0 = 0. \\ w \neq 0 \implies e^{w}\epsilon_i = e^{-w}(1 - \epsilon_i). \end{cases}$$

$$e^w\epsilon_i = e^{-w}(1-\epsilon_i).$$

$$\begin{cases} \epsilon_i = 1 \implies e^w = 0 \lightning. \\ \epsilon_i = 0 \implies e^{-w} = 0 \lightning. \\ 0 < \epsilon_i < 1 \implies wln(e) + ln(\epsilon_i) = -wln(e) + ln(1-\epsilon_i). \end{cases}$$

$$wln(e) + ln(\epsilon_i) = -wln(e) + ln(1-\epsilon_i) \implies w = \frac{1}{2}ln(\frac{1-\epsilon_i}{\epsilon_i}).$$

continuing from (4)...

$$\prod_{i=1}^{T} e^{\frac{1}{2}ln(\frac{\epsilon_i}{1-\epsilon_i})}(1-\epsilon_i) + e^{\frac{1}{2}ln(\frac{1-\epsilon_i}{\epsilon_i})}\epsilon_i. = \prod_{i=1}^{T} 2\sqrt{(1-\epsilon_i)\epsilon_i}.$$

Now, call $\gamma_i = \frac{1}{2} - \epsilon_i$.

$$\prod_{i=1}^{T} 2\sqrt{(1-\epsilon_i)\epsilon_i} = \prod_{i=1}^{T}\sqrt{(1-4\gamma_i^2)} = \prod_{i=1}^{T} e^{-2\gamma_i^2} = e^{-\sum_{i=1}^{T}2\gamma_i^2} \leq e^{-2T\gamma^2}.$$

where $\gamma = \underset{i}{min}\{\gamma_i\}$.

Which means that the training error decreases exponentially in the number of T. Now note that ϵ_i was $\mathbb{P}_i(h_i(x_t)y_t = -1)$, that is, the probability of h_i making a mistake on the training set S where each sample is weighted by $\mathbb{P}_i(t)$. So, we are saying that (for our choice of w_i), the error $l_S(f)$ goes down exponentially on the number of predictors T weighted for $\frac{1}{2} - \epsilon_i$, which means, that the error is low for ϵ_i close to 0 (few errors) or to 1 (many errors). This last statement is particularly obvious in the first term of (10).

Now we have still to clarify the step (3), signed as $(*)$. Indeed, this is not true for all $\mathbb{P}$. But, if we choose:

$$\mathbb{P}_{i+1}(t) = \frac{\mathbb{P}_i(t)e^{-w_ih_i(x_t)y_t}}{\mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_ih_i(x_t)y_t}]}.$$

Then we have that $(*)$ is true. First of all we need to prove that $\sum_{t=1}^{m}\mathbb{P} = 1$. But this is somewhat obiovous since:

$$\sum_{t=1}^{m}\frac{\mathbb{P}_i(t)e^{-w_ih_i(x_t)y_t}}{\mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_ih_i(x_t)y_t}]} = \sum_{t=1}^{m}\frac{\mathbb{P}_i(t)e^{-w_ih_i(x_t)y_t}}{\sum_{t=1}^{m}\mathbb{P}_i(t)e^{-w_ih_i(x_t)y_t}} = 1.$$

And note that:

$$\mathbb{P}_{i+1}(t) = \frac{\mathbb{P}_i(t)e^{-w_ih_i(x_t)y_t}}{\mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_ih_i(x_t)y_t}]}. \implies e^{-w_ih_i(x_t)y_t} = \mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_ih_i(x_t)y_t}]\frac{\mathbb{P}_{i+1}(t)}{\mathbb{P}_i(t)}$$

So, we have:

$$\mathbb{E}_{t\sim\mathbb{P}_1}[\prod_{i=1}^{T}e^{-w_ih_i(x_t)y_t}] = \mathbb{E}_{t\sim\mathbb{P}_1}[\prod_{i=1}^{T}\mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_ih_i(x_t)y_t}]\frac{\mathbb{P}_{i+1}(t)}{\mathbb{P}_i(t)}] =$$

$$\sum_{t=1}^{m}\mathbb{P}_1(t)(\underbrace{\prod_{i=1}^{T}\frac{\mathbb{P}_{i+1}(t)}{\mathbb{P}_i(t)}}_{telescopicproduct}\prod_{i=1}^{T}\mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_ih_i(x_t)y_t}]) = \sum_{t=1}^{m}\mathbb{P}_1(t)(\frac{\mathbb{P}_{T+1}(t)}{\mathbb{P}_1(t)}\prod_{i=1}^{T}\mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_ih_i(x_t)y_t}]) =$$

$$= \sum_{t=1}^{m}(\mathbb{P}_{T+1}(t)\prod_{i=1}^{T}\mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_ih_i(x_t)y_t}]) = \sum_{t=1}^{m}(\mathbb{P}_{T+1}(t)\prod_{i=1}^{T}\mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_ih_i(x_t)y_t}]) = \prod_{i=1}^{T}\mathbb{E}_{t\sim\mathbb{P}_i}[e^{-w_ih_i(x_t)y_t}]).$$

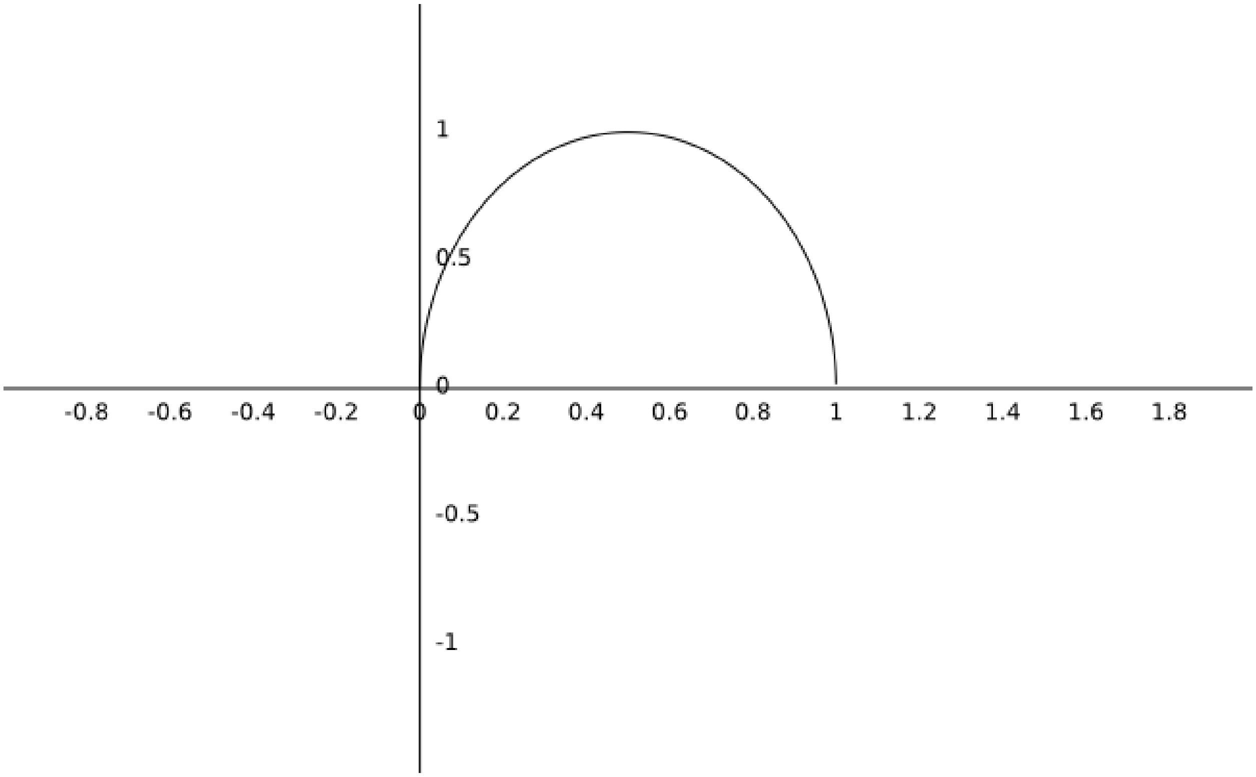

Figure 1: A plot displaying $2\sqrt{(1-\epsilon_i)\epsilon_i}$

.

Now note the followings:

- Adaboost Can use good predictors as a bad one, in fact, bad predictors will have high ϵ_i, and so, a negative weight.
- If the i-th predictor has $\frac{1}{2}$ accuracy over samples weighted with $\mathbb{P}_i$ then it has no use in the classification, since it gets a weight of 0.

1.2 Tree stump

Tree predictors are classifiers particularly suited for problems with discrete or non-euclidean domains, like:

$$\mathcal{X} = \{\underbrace{\{young, adult, old\}}_{age} \times \underbrace{\{yes, no\}}_{smoker} \times \underbrace{\{underweight, normal, overweight, obese\}}_{weight}\}.$$

The general idea is to take tests, functions of the form $f : \mathcal{X} \to \mathcal{Y}$ combining them in a tree where each leaf is labeled with a class $y \in \mathcal{Y}$ and each node is a test. Each sample $x \in \mathcal{X}$ is then routed towards a leaf and classified with the respective label. The training set is used for two reasons:

1. S is used to decide which label to assign to each leaf. Given the leaf l the the label y assigned to l is the majority of labels of samples $(x_t, y_t) \in S$ routed to l.

$$l_y = majority\{y_t | (x_t, y_t) \in S \wedge x_t \text{ routed to } l\}$$

2. S can also be used in the splitting decision (adding a new test to the tree). For example, we could split the leaf with lower accuracy and we could add the test which maximizes the mean of the accuracies of the new leaves (the accuracy is measured in respect to S).

Given any tree, its training error is always less than $\frac{1}{2}$ (in binary classification), since we apply the majority label to the leaf's class. And, any new split can only reduce the training error.

Tree stumps are tree predictors with only one node, in fact, they are just tests.

1.3 Adaboost with stumps

Let us suppose that $\mathcal{Y} = \{-1, +1\}$, so that, we are in a binary classification setting. The algorithm works as follow:

```
Let S be the training set.
Let T be the number of boosting rounds.
Let F be set of possible tests (stumps).
Let P_1 be the uniform probability distribution over S.
For i = 1,2,...,T:
  choose a test f_i in F.
  compute the e_i training error weighted with P_i, for f_i.
  compute the predictor's weight w_i.
  compute the new probability distribution P_i+1.
Return ((w_1,..,w_T),(f_1,...,f_T)).
```

Now the only part that is unclear is about how to choose the next test. In fact, this is not important, Adaboost works even with randomly chosen tests. But, this affects the number of iteration required to convergence.

So, i tried the following strategies for the stump generation:

1. Fully random. the next test is chosen from a pre-generated set of possible tests.

2. A variable is chosen randomly and the best test is searched from all tests related to that variable. For the sake of speed if a test has an error that differs of 0.001 from $\frac{1}{2}$ is directly chosen.

3. The set of possible tests is shuffled and then the best is searched sequentially. Again, for the sake of speed if the test error differs from $\frac{1}{2}$ for more than 0.001 then is directly chosen.

1.4 Cross-Validation

Cross-validation, given a learning algorithm A is a good method to approximate as closely as possible the statistical error. The idea is to run A more times over different training set and test set. The mean of different test errors should be close to the statistical error.

More precisely, given the dataset S. split S in k partitions $S_1, S_2, \ldots S_k$ where each of them has size of $\frac{|S|}{k}$ and return:

$$\frac{1}{K} \sum_{i=1}^{i=k} l_{S_i}(A(S \setminus S_i)).$$

Often learning algorithms may have hyperparameters, so we need to test essentially different algorithms $A_1, \ldots, A_r$. We may use cross-validation to find the one with the smallest meaned test error. However, it is possible that we have found one A_l that is good on the dataset but not on the real statistical data, in this case, we are overfitting in respect to the hyperparameters and our test error doesn't mean too much. So to protect ourselves from this rather unfortunate case we split S in two S_1 and S_2. S_1 will be used normally with the cross-validation, meanwhile, the purpose of S_2 is to have unseen data to test our best algorithm.

2 Forest Cover dataset

The dataset is composed of 15120 records. Each record has 56 attributes where the first one is a unique identifier and the last one the class (or cover type), which ranges from 1 to 7. Of course, these two attributes will be excluded from the classification. Our input data are both binary and integers and we have variables as Elevation, Aspect, Slope, etc. Lastly, all classes are the $\sim 14\%$ of the whole dataset, so, there are not over represented classes.

3 Experiment description

3.1 Tests generation (stumps)

So we have 54 attributes: $\mathcal{X} = \mathcal{X}_1, \dots, \mathcal{X}_{54}$ and $\mathcal{Y} = \{-1, 1\}$, where each class is considered in a setting one vs all, so that, we can use binary predictors. Each base predictor is from $\mathcal{F} \subset \{\{1, \dots, 54\} \times \mathbb{N} :\rightarrow \{True, False\}\}$. For example, the test $f_{(14,42)}(x)$ would check if $x_{14} \leq 42$.

For each attribute $\mathcal{X}_i$, I have collected its value appearing in the training set. For instance, for a binary attribute, I would get $\{0, 1\}$. Given this set, I have calculated the percentile, for each 0.001%, so that, for a list like $[1, 2, 3, \dots, 10000]$ I take only $[0, 10, 20, \dots, 10000]$.

In conlusion, for an attribute $\mathcal{X}_i$ with seen values $\in [1, 10000]$, I generate the list of tests $[f_{(i,0)}, f_{(i,10)}, f_{(1i4,20)}, \dots, f_{(i,10000)}]$. This choice is just for the sake of not having too many tests.

3.2 The test choice

Since Adaboost benefits of strongly erroneous or accurate predictors, it makes much sense to search for these types of predictors. So, when we want to know how a predictor behaves at the i-th step of Adaboost we just need to compute.

$$\epsilon_f = \mathbb{P}_i(f(x_t)y_t = -1)$$

The far the predictor strays from $\frac{1}{2}$ the better it is. Again, the strategies adopted for the search are: fully random, shuffled best and random attribute best test.

3.3 Measures

I have measured three common metrics: accuracy, precision, and recall. Each one of the previous can be computed easily by computing the followings:

- true positives:
$$tp = |\{(x, y) \in S : h(x) = 1 \land y = 1\}|.$$
- false positives:
$$fp = |\{(x, y) \in S : h(x) = 1 \land y = 0\}|.$$
- true negatives:
$$tp = |\{(x, y) \in S : h(x) = 0 \land y = 0\}|.$$
- false negatives:
$$fn = |\{(x, y) \in S : h(x) = 0 \land y = 1\}|.$$

where:

- Accuracy is the rate of correct predictions against all predictions.
$$accuracy = \frac{tp + tn}{tp + fn + fp + tn}$$
- Recall is the rate of true labels classified as true over all true labels.
$$recall = \frac{tp}{tp + fn}$$
- Precision is the rate of true labels classified as true against all true predictions.
$$recall = \frac{tp}{tp + fn}$$

3.4 Bringing it all together

How does all works:

1. Get the list of all records S.
2. Generate from S the possible tests $\mathcal{F}$.
3. Split the dataset in two S_1 and S_2, such that, $|S_1| = \frac{1}{3}|S_2|$.
4. Choose a strategy from $\{\text{random}, \text{shuffled best}, \text{random attribute best test}\}$.
5. Choose a class $c \in \{1, 2, 3, 4, 5, 6, 7\}$.
6. Generate the one-vs-all version of S_1, call it S_{vs}.
$$S_{vs} = \{(x, 0)|(x, c') \in S \wedge c' \neq c\} \cup \{(x, 1)|(x, c) \in S\}.$$
7. Now partition S_{vs} in $S_{vs_1}, S_{vs_2}, S_{vs_3}, S_{vs_4}$.
8. Train the predictor $h_{c,strategy,S_{vs_j}}$ with Adaboost and $\mathcal{F}$ over $S_1 \setminus S_{vs_j}$.
9. Test the prdictor $h_{c,strategy,S_{vs_j}}$ over S_{vs_j}.
10. Repeat from 8. for all $j \in \{1, 2, 3, 4\}$.
11. Repeat from 5. for all $c \in \{1, 2, 3, 4, 5, 6, 7\}$.
12. Mean all the test error obtained for ceratain strategy, call it $\epsilon_{strategy}$.
13. Repeat from 4. for all *strategy* $\in \{\text{random}, \text{shuffled best}, \text{random attribute best test}\}$.
14. find $bestStrategy = \underset{strategy}{argmin}\{\epsilon_{random}, \epsilon_{shuffledbest}, \epsilon_{randomattributebesttest}\}$.
15. Train again a predictor h with adaboost, but this time over all S_1 with the *bestStrategy*.
16. Compute the test error of h on S_2.

4 Experiment results

Now we will compare results obtained for different strategies and different classes. The best strategy between the one listed is *random attribute best choice* which has reached a meaned (from different classes) test error of 0.922 using 1001 stump predictors.

lets start comparing results for the most hard class (class 2):

Random.

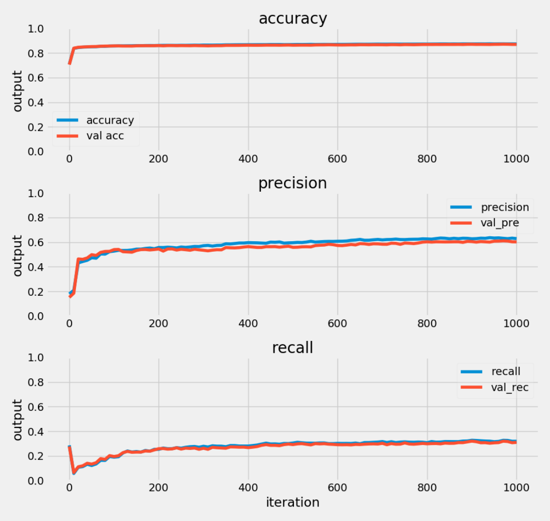

Figure 2: A plot displaying the results for random choice strategy for class 2 with 4 partitions of cross-validation.

Random attribute best test.

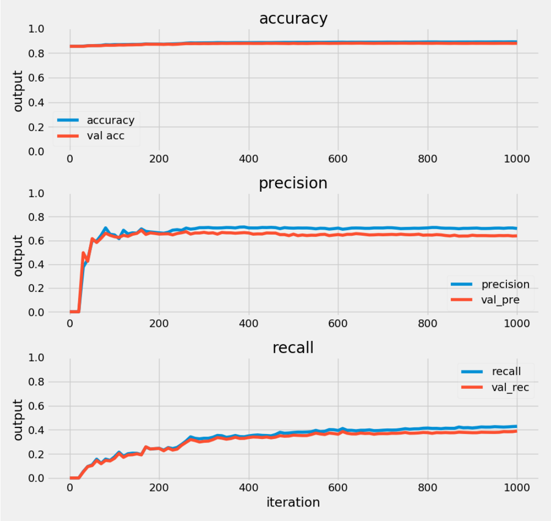

Figure 4: A plot displaying the results for random attribute best test strategy for class 2 with 4 partitions of cross-validation.

Shuffled Best.

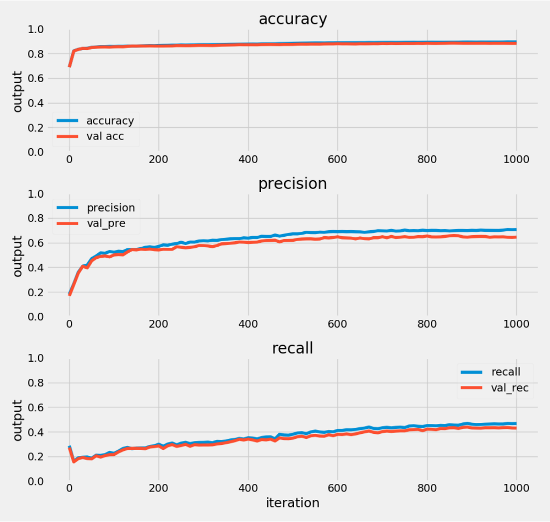

Figure 3: A plot displaying the results for shuffled best test strategy for class 2 with 4 partitions of cross-validation.

Here the best class accuracy barely reaches 90%, And recall is extremely low, which means that actually, we are not able to classify well the data. Instead, the 90% seems even too good, and probably the predictor classifies most of the label as *false*. This is no surprise since the large majority of them are *false* (about $\sim 85\%$).

As we can see the *random* strategy is the worst behaving one, and despite being the fastest, even after 1000 Adaboost's iteration it doesn't catch up with the others. Meanwhile *Random attribute best test* seems to converge faster, in number of iteration, than others.

Now lets see for the most easy class (class4):

Random.

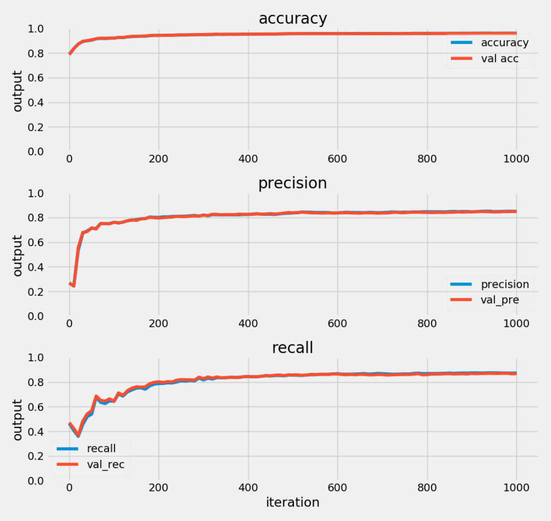

Figure 5: A plot displaying the results for random choice strategy for class 4 with 4 partitions of cross-validation.

Random attribute best test.

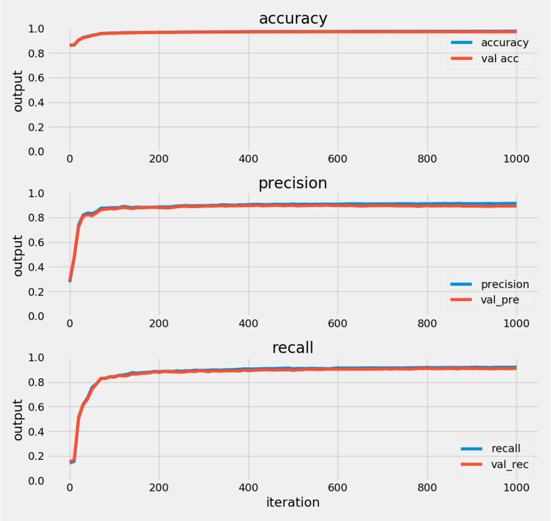

Figure 7: A plot displaying the results for random attribute best test strategy for class 4 with 4 partitions of cross-validation.

Shuffled Best.

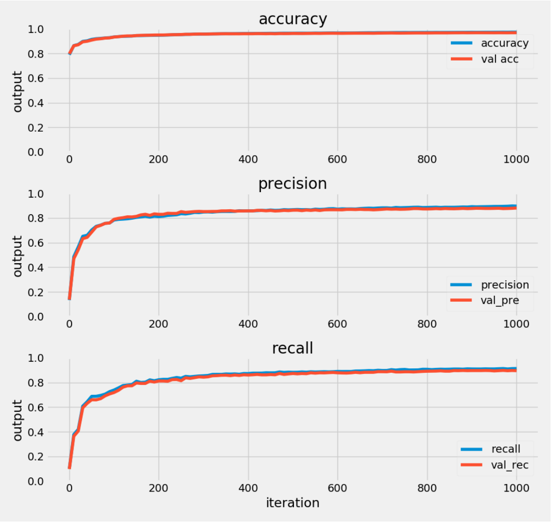

Figure 6: A plot displaying the results for shuffled best test strategy for class 4 with 4 partitions of cross-validation.

Here, the previous problem doesn't seem to appear. Even if the majority of labels is false, the predictor has learned well how to distinguish class 4 from the other.

The previous reasoning applies also in this case, and the best strategy seems to be *random attribute best test* Here the best class accuracy reaches $\sim 97\%$.

Finally, we have ascertained that the strategy which reaches the best results and faster is *random attribute best test*. And now, we show the values obtained when this algorithm is trained on the whole validation data and tested on completely unseen records:

Class 1.

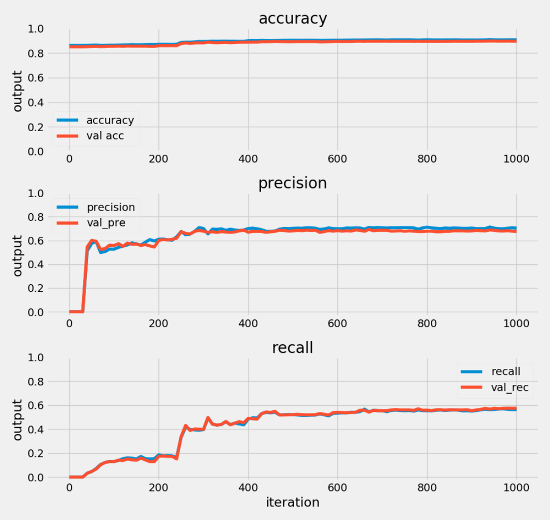

Figure 8: A plot displaying the results for random attribute best test strategy for class 1 without cross-validation.

Class 2.

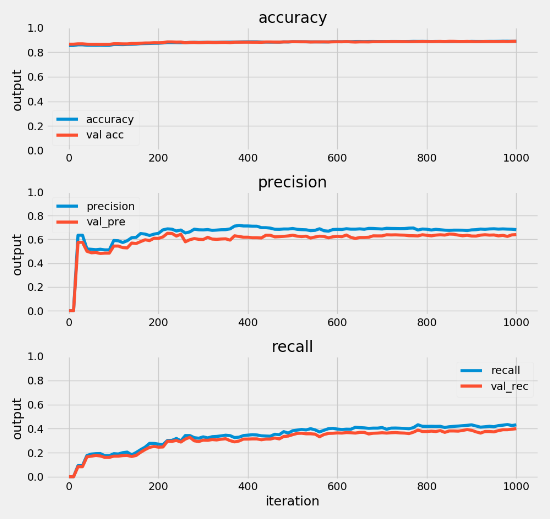

Figure 9: A plot displaying the results for random attribute best test strategy for class 2 without cross-validation.

Class 3.

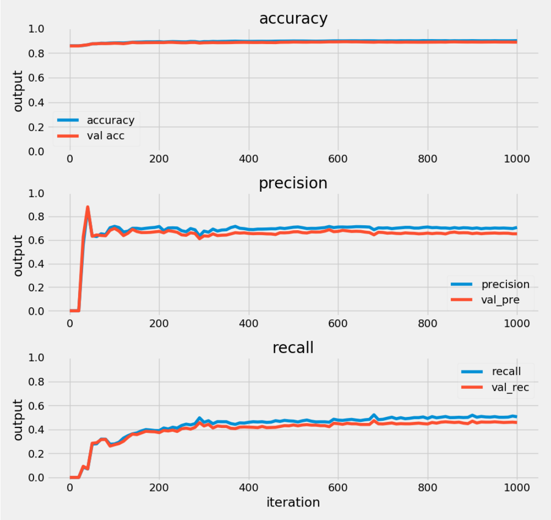

Figure 10: A plot displaying the results for random attribute best test strategy for class 3 without cross-validation.

Class 4.

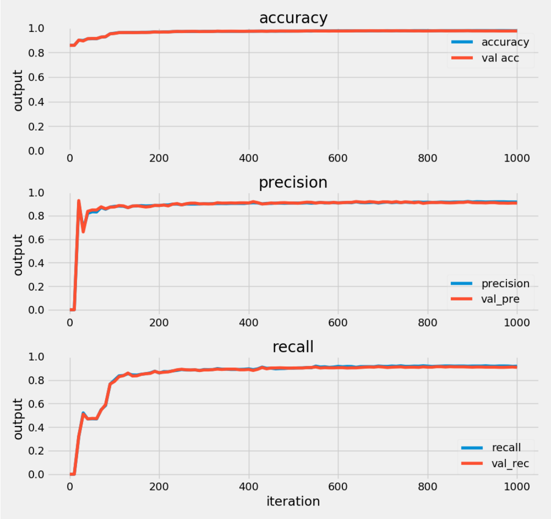

Figure 11: A plot displaying the results for random attribute best test strategy for class 4 without cross-validation.

Class 5.

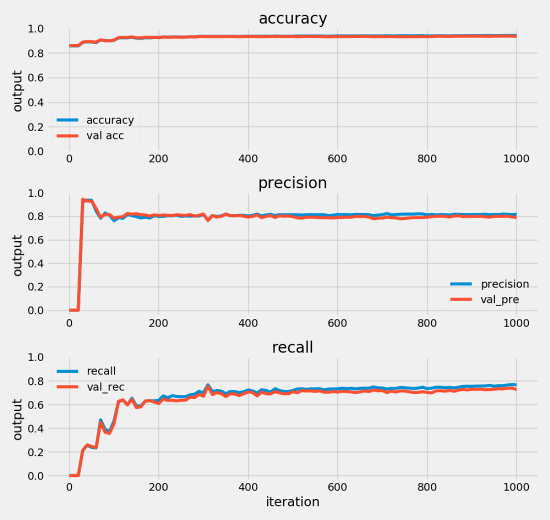

Figure 12: A plot displaying the results for random attribute best test strategy for class 5 without cross-validation.

Class 6.

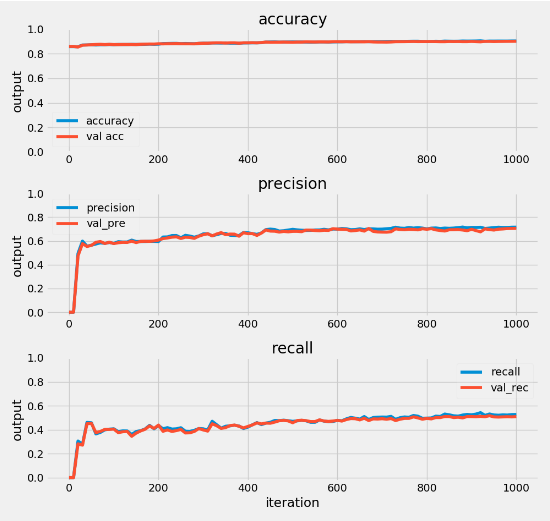

Figure 13: A plot displaying the results for random attribute best test strategy for class 6 without cross-validation.

Class 7.

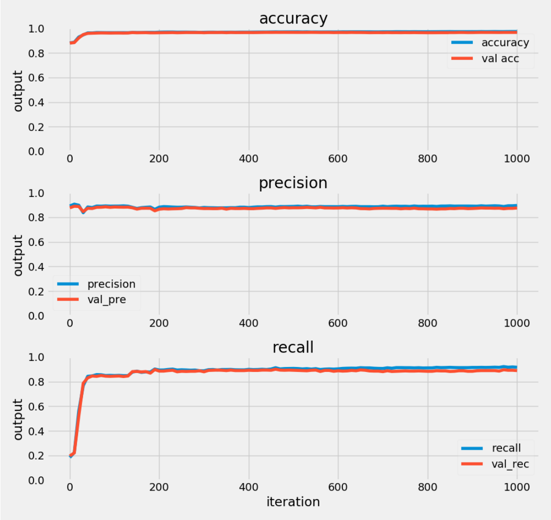

Figure 14: A plot displaying the results for random attribute best test strategy for class 7 without cross-validation.

In conclusion, even with more training data classes $1, 2, 3$ and 6 seems a bit problematic, meanwhile, classes $4, 5$ and 7 behave very well.