AdaBoost
IT3105

Nordmoen, Jørgen H.
Østensen, Trond

November 20, 2012

Abstract

In this paper we will describe how we implemented AdaBoost, two weak classifiers and what results we got on a number of datasets. We will describe the datasets and why we chose them. We will explain how we divided those datasets into buckets for classification and how that is supported in our implementation. We will end this paper with a discussion of our results trying to explain and illustrate why.

Contents

List of Tables

1 Implementation

In this section we will talk a bit about our implementation. We will describe how it manages to classify new instances, how we generate new classifiers and how it can be extended without changing any implementation.

From our AdaBoost implementations view all we generate are Hypothesis[1] which comes from Generators[2]. These interfaces dictates what all new classifiers must support. A Generator creates a new Hypothesis with the basis in weights and the training set. An Hypothesis only needs to support setting its weight and classifying a new instance[3].

From an Hypothesis point of view all data is of type DataElement[4] which contains a number of attributes specified when it's created. It also contains a classification which it must be able to produce. DataElement uses Java generics to support different types of attributes and classifications which is determined by the filter used. Each Generator must support the same attributes and classifications as the DataElement used, but this is only checked at runtime.

Our data flow starts with reading all lines in a file creating DataElements with String values. Then we apply a filter which must implement the interface Filter[5]. After that is done we shuffle the dataset using a specified random seed to be able to replicate our tests. We then split the dataset into a training set and test set with a split specified on the command-line. After that we start AdaBoost and try to classify the test set.

The meat of our implementation lies in the Generator class which dictates how a specific classifier is created. This is the class which is defined on the command-line and this is the class which can be configured from the command-line.

This class must be able to receive a list of options where it can act on each option according to its own wishes which is done so we can configure each Generator, this is among other things where we pass the depth parameter to the Decision Tree Generator.

Each Generator class uses its own logic to create a classifier of the correct type based on the given data. This means that AdaBoost does not know nor care what a Generator creates as long as it implements the Hypothesis interface. This also means that our implementation is very general. When creating a new classifier type one has to implement a new Generator and a new Hypothesis class, which is all. To support new datasets one only has to implement a new filter to convert from String to a specific type. This was done to decouple the datasets from the classifiers, but it also means that we have to implement less code since each classifier shares the filters.

The filters are tasked with both converting the String values read from a file to the correct format, but it's also tasked with creating buckets for the classifiers. With this we mean that the filter must split the values in each file into proper buckets since our classifiers can't handle large ranges. This is also the reason why we have a filter per dataset. Since we had a hard time creating

[1] no.ntnu.ai.hypothesis.Hypothesis

[2] no.ntnu.ai.hypothesis.Generator

[3] We have created an abstract class for convenience, situated in no.ntnu.ai.classifiers.Classifier

[4] no.ntnu.ai.data.DataElement

[5] no.ntnu.ai.filter.Filter

an algorithm for splitting the ranges in different datasets we chose to have it like this for ease and to be able to control the buckets in more detail.

2 Results

Below is all the results from all our runs. We have used a couple standard values to control the randomness and the split of the datasets.

For each run we have used a split of "0.2" which means that we use 0.2 of our dataset as a test set. We have also used the random seed "42" for all runs this means that in every place where we need a random number the generator for that number is initialized in the same way with the same number each time. This creates reproducibility and ensures that we can be certain that our results are given the same starting point every time.

For each dataset we will point out what is needed to replicate the results.

The classifier options below does not conform to our actual command-line style which is down to space requirement. To use just add the sentence ' –classifier no.ntnu.ai[nbc—dtc].' in front.

2.1 Glass Dataset

This dataset represents the classification of different glasses in a crime scene.

To run, use these options with the classifier options wanted.

Listing 1: Glass dataset general options

```
--global 0.2 42 --file glass.txt --filter no.ntnu.ai.
    filter.GlassFilter
```

NBC #	Training Error	Standard Deviation	DTC #	Training Error	Standard Deviation	Test Error	Classifier option
1	0.339	N/A	0	N/A	N/A	18/43(41%)	NBCGenerator 1
0	N/A	N/A	1	0.181	N/A	17/43(39%)	DTCGenerator 1
5	0.391	0.045	0	N/A	N/A	19/43(44%)	NBCGenerator 5
10	0.371	0.042	0	N/A	N/A	19/43(44%)	NBCGenerator 10
20	0.358	0.034	0	N/A	N/A	20/43(46%)	NBCGenerator 20
0	N/A	N/A	5	0.256	0.043	18/43(41%)	DTCGenerator 5
0	N/A	N/A	10	0.673	0.133	23/43(53%)	DTCGenerator 10 1
0	N/A	N/A	10	0.623	0.124	24/43(55%)	DTCGenerator 10 2
0	N/A	N/A	10	0.265	0.047	18/43(41%)	DTCGenerator 10
0	N/A	N/A	20	0.268	0.042	19/43(44%)	DTCGenerator 20
5	0.430	0.046	5	0.651	0.095	21/43(48%)	DTCGenerator 5 2, NBCGenerator 5
10	0.391	0.044	10	0.646	0.105	17/43(39%)	DTCGenerator 10 2, NBCGenerator 10
20	0.409	0.045	20	0.661	0.093	22/43(51%)	DTCGenerator 20 2, NBCGenerator 20

Table 1: Table showing the results of our classifiers on the Glass dataset

2.2 Page-blocks Dataset

This dataset represents the different blocks of the page layout of a document that has been detected by a segmentation process.

To run, use these options with the classifier options wanted.

Listing 2: Page-blocks dataset general options

```
--global 0.2 42 --file page-blocks.txt --filter
no.ntnu.ai.filter.PageBlocksFilter
```

NBC #	Training Error	Standard Deviation	DTC #	Training Error	Standard Deviation	Test Error	Classifier option
1	0.071	N/A	0	N/A	N/A	76/1095(6%)	NBCGenerator 1
0	N/A	N/A	1	0.011	N/A	47/1095(4%)	DTCGenerator 1
5	0.158	0.088	0	N/A	N/A	62/1095(5%)	NBCGenerator 5
10	0.134	0.075	0	N/A	N/A	67/1095(6%)	NBCGenerator 10
20	0.110	0.058	0	N/A	N/A	65/1095(5%)	NBCGenerator 20
0	N/A	N/A	5	0.020	0.010	50/1095(4%)	DTCGenerator 5
0	N/A	N/A	10	0.598	0.263	75/1095(6%)	DTCGenerator 10 1
0	N/A	N/A	10	0.485	0.200	61/1095(5%)	DTCGenerator 10 2
0	N/A	N/A	10	0.021	0.008	47/1095(4%)	DTCGenerator 10
0	N/A	N/A	20	0.021	0.006	50/1095(4%)	DTCGenerator 20
5	0.111	0.023	5	0.533	0.234	66/1095(6%)	DTCGenerator 5 2, NBCGenerator 5
10	0.095	0.007	10	0.596	0.267	56/1095(5%)	DTCGenerator 10 2, NBCGenerator 10
20	0.100	0.004	20	0.678	0.179	53/1095(4%)	DTCGenerator 20 2, NBCGenerator 20

Table 2: Table showing the results of our classifiers on the Page-blocks dataset

2.3 Yeast Dataset

This dataset represents the cellular localization sites of proteins in yeast.
To run, use these options with the classifier options wanted.

Listing 3: Yeast dataset general options

```
--global 0.2 42 --file yeast.txt --filter no.ntnu.ai.
    filter.YeastFilter
```

NBC #	Training Error	Standard Deviation	DTC #	Training Error	Standard Deviation	Test Error	Classifier option
1	0.414	N/A	0	N/A	N/A	132/297(44%)	NBCGenerator 1
0	N/A	N/A	1	0.347	N/A	152/297(51%)	DTCGenerator 1
5	0.585	0.089	0	N/A	N/A	150/297(50%)	NBCGenerator 5
10	0.576	0.064	0	N/A	N/A	148/297(49%)	NBCGenerator 10
20	0.556	0.051	0	N/A	N/A	152/297(51%)	NBCGenerator 20
0	N/A	N/A	5	0.506	0.088	156/297(48%)	DTCGenerator 5
0	N/A	N/A	10	0.796	0.127	187/297(62%)	DTCGenerator 10 1
0	N/A	N/A	10	0.763	0.122	164/297(55%)	DTCGenerator 10 2
0	N/A	N/A	10	0.535	0.076	145/297(48%)	DTCGenerator 10
0	N/A	N/A	20	0.544	0.059	146/297(49%)	DTCGenerator 20
5	0.584	0.088	5	0.796	0.080	141/297(47%)	DTCGenerator 5 2, NBCGenerator 5
10	0.600	0.055	10	0.788	0.126	141/297(47%)	DTCGenerator 10 2, NBCGenerator 10
20	0.598	0.051	20	0.833	0.094	164/297(55%)	DTCGenerator 20 2, NBCGenerator 20

Table 3: Table showing the results of our classifiers on the Yeast dataset

2.4 Nursery Dataset

This dataset represents the classification of applications for a nursery school.

To run, use these options with the classifier options wanted.

Listing 4: Nursery dataset general options

```
--global 0.2 42 --file nursery.txt --filter no.ntnu.ai.
    filter.NurseryFilter
```

NBC #	Training Error	Standard Deviation	DTC #	Training Error	Standard Deviation	Test Error	Classifier option
1	0.097	N/A	0	N/A	N/A	255/2592(9%)	NBCGenerator 1
0	N/A	N/A	1	0.0	N/A	76/2592(2%)	DTCGenerator 1
5	0.234	0.117	0	N/A	N/A	249/2592(9%)	NBCGenerator 5
10	0.193	0.094	0	N/A	N/A	249/2592(9%)	NBCGenerator 10
20	0.161	0.094	0	N/A	N/A	249/2592(9%)	NBCGenerator 20
0	N/A	N/A	5	0.0	0.0	76/2592(2%)	DTCGenerator 5
0	N/A	N/A	10	0.381	0.066	847/2592(32%)	DTCGenerator 10 1
0	N/A	N/A	10	0.358	0.081	451/2592(17%)	DTCGenerator 10 2
0	N/A	N/A	10	0.0	0.0	76/2592(2%)	DTCGenerator 10
0	N/A	N/A	20	0.0	0.0	76/2592(2%)	DTCGenerator 20
5	0.229	0.075	5	0.380	0.077	291/2592(11%)	DTCGenerator 5 2, NBCGenerator 5
10	0.194	0.075	10	0.349	0.079	313/2592(12%)	DTCGenerator 10 2, NBCGenerator 10
20	0.269	0.037	20	0.347	0.085	338/2592(13%)	DTCGenerator 20 2, NBCGenerator 20

Table 4: Table showing the results of our classifiers on the Nursery dataset

2.5 Pen Digit Dataset

This dataset represents the classification of digits written by different writers.
To run, use these options with the classifier options wanted.

Listing 5: Pen digit dataset general options

```
--global 0.2 42 --file pen-digits.txt --filter no.ntnu.ai
    .filter.PenDigitsFilter
```

NBC #	Training Error	Standard Deviation	DTC #	Training Error	Standard Deviation	Test Error	Classifier option
1	0.119	N/A	0	N/A	N/A	251/2199(11%)	NBCGenerator 1
0	N/A	N/A	1	0.0	N/A	252/2119(11%)	DTCGenerator 1
5	0.314	0.106	0	N/A	N/A	248/2199(11%)	NBCGenerator 5
10	0.329	0.081	0	N/A	N/A	239/2119(10%)	NBCGenerator 10
20	0.330	0.078	0	N/A	N/A	230/2119(10%)	NBCGenerator 20
0	N/A	N/A	5	0.0	0.0	252/2119(11%)	DTCGenerator 5
0	N/A	N/A	10	0.761	0.057	1126/2119(55%)	DTCGenerator 10 1
0	N/A	N/A	10	0.615	0.127	398/2119(18%)	DTCGenerator 10 2
0	N/A	N/A	10	0.0	0.0	252/2119(11%)	DTCGenerator 10
0	N/A	N/A	20	0.0	0.0	252/2119(11%)	DTCGenerator 20
5	0.289	0.086	5	0.664	0.063	172/2119(7%)	DTCGenerator 5 2, NBCGenerator 5
10	0.351	0.049	10	0.612	0.0121	196/2119(8%)	DTCGenerator 10 2, NBCGenerator 10
20	0.347	0.073	20	0.656	0.091	174/2119(7%)	DTCGenerator 20 2, NBCGenerator 20

Table 5: Table showing the results of our classifiers on the Pen-digits dataset

3 Discussion

One important thing we found when examining the results were that the datasets are very important for the effect the boosting algorithm has. If a dataset has a low number of instances and/or a very heavily skewed proportion of instance-classes boosting, at least using our implementation of Naive Bayesian and Decision Tree Classifiers, is not very effective.

Another important factor for the effect of boosting is the depth of the decision trees the DTC is allowed to construct, if these are deep enough to allow a dataset to be split on all of its attributes it will construct over-fitted trees that only have a marginal improvement from boosting in the cases were the data is ambiguous or unknown.

One of the problems that we encountered when using boosting is that the weight of a classifier would not reflect what it could actually explain. With this we mean that a first classifier might get quite a lot correct and therefor get a high weight, the next classifier will then only care about the instances that the previous classifier got wrong and won't care about the rest. This might mean that the second classifier were able to answer all the instances the first got wrong, but miss classify the rest and thus get a low weight. This would manifest it self when we try to classify the test set, the first classifier would suggest a wrong answer, but because it has a high weight it will win over the second. Even though that test instance might be one of the instances the first had gotten wrong.

This seemed to be a problem for a couple of the other groups and one suggestion was to partition the training set randomly using the weights, possibly with Fitness proportionate selection[6]. When generating a new classifier the classifier would only see a subset of the actual training set, but the weight of the classifier would not be effected by the set of training instances which it did not see. This could mean that the classifiers would only get a weight proportional to the actual explanation strength of that classifier. Unfortunately we did not have time to try and implement this, but it could possibly solve some of the problems that we experienced during boosting.

[6] https://en.wikipedia.org/wiki/Roulette_wheel_selection