Introduction

For the implementations of the naive Bayes, logistic regression and SVM we stayed close to the code shown in the lecture. Thus the implementations of the algorithms were straightforward while optimization and data pre-processing were challenging. Most of our effort went into the data pruning.

In our team one person each focused on one classifier. While we shared insights and the code that we use for the reader, this might account for inconsistencies in the code and different tone in the chapters of this report.

Data transformation and pruning

A big part of our implementation was data preprocessing. We read the tokens from the documents (with the tinyIR library), removed tokens not consisting of letters, stemmed the word and removed stopwords. [1, 2] both list this as the standard approach to pre-processing in document classification. From the resulting corpus of documents we remove those that occured in less then `minOccurrence` and those that are found in more than `maxOccurrenceRate` · `nrDocuments`, where `maxOccurrenceRate` $\in (0, 1]$. The resulting dictionary of words is used to represent each document as a bag-of-words vector. For the weights within the vector we used boolean weights (1 if the word occurs in the document, else 0), tf-idf weights (inspired by [2]) and the word frequency. The latter generally lead to the best results. We also discovered that words extracted from the title of the document contain a high amount of information and lend themselves especially to predicting country codes. The words from the titles were processed like the words from the content.

Because the initial documents have more influence in online algorithms such as our SVM and Logistic Regression, we tried accessing the documents of the training set in random order. The hope was to spread the influence of documents more evenly across the corpus. However this did not lead to significant improvements.

Predicting the industry codes turned out to be very difficult : There were more possible industry codes, and some of them were very rarely assigned. In addition, we observed that around half of the documents did not have any industry code assigned. For Logistic Regression and Naive Bayes, we therefore chose to never assign any industry code. This lead to better results as the precision remained unaffected.

Naive Bayes

We used the formulas for Naive Bayes as shown in the lecture. For each code we calculated the probability $P(word|code)$ for every word in the reduced dictionary. We were then faced with the problem of finding cutoff values for probabilities that determine whether a code is assigned to a document or not.

For each of the three different code-types different cutoffs were evaluated. The advantage of this approach is the independence of the code-types and therefore the possibility of a country-code to be predicted although the probability for this code given a document is much lower than for all the topic-codes. The performance in form of F1-scores is plotted for both country- and topic-codes in figure 1. As mentioned, no industry codes were predicted, and the document title was used instead of the document content to calculate probabilities for the country codes.

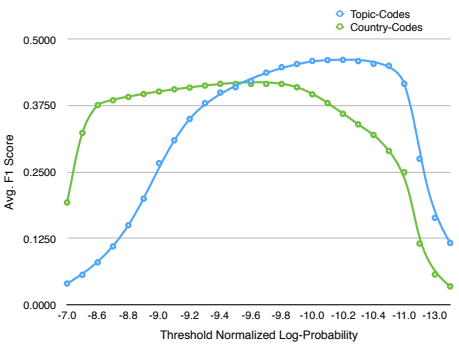

Figure 1: This figure shows the average F1-Score of topic- and country-codes using different thresholds for the normalized log-probability.

For the topic-codes the optimal threshold for the log-probabilities has been evaluated on the full training data to be -10.2 and for the country-codes -9.4. With these thresholds and the above mentioned approach of only considering article titles for training of country-codes and only considering content for training topic-codes we achieved an F1-Average score of 0.424 on the validation set.

The training of the model needs constantly 1.3GB of RAM. The training of the topic-codes took around 160 minutes in total to train the topic-codes (80 minutes) and the country-codes (80 minutes). This was done on a laptop with an intel i7 (2.9GHz) with 8GB of RAM.

Logistic Regression

Logistic regression was done as an online-algorithm with one-vs-all classifiers for each code. After pre-processing and pruning the data as described above, we had to select the `minOcccurence` and `maxOccurence` rate. We tried different values but saw that using `minOccurrence` = 20 and `maxOccurenceRate` = 1 lead to the best results. As in naive bayes, the three code types were treated independently :

Topic codes For topic codes we tried to find a cutoff value such that the average number of codes assigned to a document remained the same in our prediction as in the test set. This was achieved with a "cutoffFinder" construct consisting of two priority queues. With this approach reached an F1-score of around 0.35 for topic codes.

Country codes Again, we had even better results when using the document title instead of the content. Also, the cutoff value was determined manually by trying different values : The cutoffFinder approach lead to worse results since the resulting codes were assigned unevenly across the documents. With our approach we reached a F1-score of around 0.44

Industry codes As mentioned, we never predicted any Industry codes, because the negative impact on precision was too high.

Predicting the three label types separately and combining the predictions into one lead to an F1-score of around 0.36. One attempt at increasing the performance was to lower the learning rate, such that it would increase, for example, after 10 documents instead of 1. Also, different normalization techniques were tried, but none lead to significant improvement in performance. Our final algorithm required approximately 1.9 GB of memory and took around 12 minutes to compute an intel i7 laptop (2.9GHz) with 8GB of RAM.

SVM

The algorithm shown in the lecture is the well-known pegasos algorithm[4, 3]. For the implementation we trained one SVM per category occurring in the training data in an all-vs-one approach. Table 1 shows the results of various approaches. The data was preprocessed as introduced before with the use of homogeneous coordinates (bias term) and without. The listed papers suggest not using a bias with the normal formulation of the pegasos algorithm or to adjust it if doing so, as it undermines the convergences guarantee. As it only provided a very small change, we therefore did not include a bias. For the parameter λ the optimal choice seems to be around $10^{-4} - 10^{6}$. 10^{-5} is the point where both precission an recall are optimal, higher values have better precission, lower better recall, but always at the cost of the other parameter. This seems consistent when changing other parameter of the algorithm. Thus further tests were only run with those λ values.

Reducing the dictionary size helped to reduce the run time of the algorithm and slightly increased the score. So the final version (bold) uses only tokens extracted from titles, but all of them.

When it comes to the different weighting schemes for bag-of-words, tf-idf is clearly the best, as suggested in the literature [2]. Closely followed by absolut word-couns and not so close by boolean weights.

λ	`minOccurrence`	`maxOccurrenceRate`	Dictionary,Weights	Resulting # Words	Run Time	Memory Consumption	Avg. Precession	Avg. Recall	Avg. F1-Score
1	1	0.2	tokens, count-weights	109995	123 min	8 GB	0.34	0.09	0.14
1	3	0.2	tokens, count-weights	39249	28 min	3.4 GB	0.35	0.09	0.14
0.1	3	0.2	tokens, count-weights	39249	28 min	3.7 GB	0.35	0.09	0.14
0.01	3	0.2	tokens, count-weights	39249	19 min	3.4 GB	0.64	0.19	0.27
0.001	3	0.2	tokens, count-weights	39249	15 min	3.5 GB	0.88	0.41	0.52
0.0001	3	0.2	tokens, count-weights	39249	14 min	3.4 GB	0.91	0.63	0.71
0.00001	3	0.2	tokens, count-weights	39249	17 min	3.4GB	0.86	0.73	0.76
0.0001	10	0.2	tokens, count-weights	17521	7 min	3 GB	0.91	0.63	0.71
0.0001	20	0.2	tokens, count-weights	11397	5 min	3 GB	0.91	0.63	0.71
0.0001	20	0.2	tokens, boolean weights	11397	5 min	2.9 GB	0.91	0.59	0.68
0.0001	20	0.2	tokens, tf-idf weights	11397	5 min	3.1 GB	0.93	0.65	0.73
0.0001	30	0.2	tokens, boolean weights	8785	4 min	3.1 GB	0.91	0.59	0.68
0.0001	30	0.2	tokens, count-weights	8785	4 min	3.1 GB	0.91	0.63	0.71
0.0001	30	0.2	tokens, tf-idf weights	8785	4 min	3.1 GB	0.92	0.66	0.73
0.00001	30	0.2	tokens, tf-idf weights	8785	4 min	3.1 GB	0.86	0.74	0.77
0.00001	**30**	**1**	**tokens, tf-idf weights**	**8808**	**4 min**	**3.1 GB**	**0.87**	**0.74**	**0.78**
0.0001	40	0.2	tokens, boolean weights	7311	3 min	3.1 GB	0.91	0.60	0.68
0.0001	40	0.2	tokens, tf-idf weights	7311	3 min	3.1 GB	0.92	0.65	0.73
0.0001	40	0.2	tokens, count-weights	7311	3 min	3.1 GB	0.91	0.63	0.71
0.0001	20	1	tokens, count-weights	11420	5 min	3.1 GB	0.92	0.62	0.71
0.0001	0	1	titles, count-weights	22871	8 min	3.1 GB	0.92	0.61	0.69
0.0001	0	1	titles, tf-idf weights	22871	8 min	3.1 GB	0.92	0.62	0.70

Table 1: Test Results for the SVM. All tests were done on a Machine with 32 GB ram and a high java heap size, thus java never deallocated memory. On another machine with less memory the memory consumption of the SVM code took only around 2 GB for most cases. The fact that this was tested on a stronger machine is also the reason why training of the SVM is faster than other approaches. The bold line shows what is used for submission and what is the current default.

It should be noted that we struggled with a bug in our SVM implementation until shortly before the deadline. Otherwise it would probably have been possible to even turn it further and maybe implement features like RBF kernels that perform very well in the literature.

References

[1] Thorsten Joachims. "Text categorization with support vector machines: Learning with many relevant features". In: *European conference on machine learning*. Springer, 1998, pp. 137–142.

[2] Arzucan Özgür, Levent Özgür, and Tunga Güngör. "Text categorization with class-based and corpus-based keyword selection". In: *International Symposium on Computer and Information Sciences*. Springer, 2005, pp. 606–615.

[3] Shai Shalev-Shwartz et al. "Pegasos: Primal Estimated sub-GrAdient SOlver for SVM". In: ().

[4] Shai Shalev-Shwartz et al. "Pegasos: Primal estimated sub-gradient solver for svm". In: *Mathematical programming* 127.1 (2011), pp. 3–30.