Information Retrieval Project 1, Group 11

Michael Merki, Julien Romero, Markus Greiner
November 2016

Installation

See README.md how to install

Preprocessing / Tokenizing

We started with the provided tokenizer that is part of ReutersRCVParser. For the 50'000 training documents, this resulted in a vocabulary size of 9.2 Mio tokens for the given 150 K words.

The performance of the classifiers during runtime is strongly influenced by the vocabulary size. So the goal was to reduce vocabulary size without negative influence on precision.

In a second step, the following tokens were matched using regular expressions and replaced numbers, dates, ordinals, punctuation and underlines with standard tokens:

```
// regular expressions defined statically
val rDate = ("^\\d+[/-]\\d+[/-]\\d+$".r, "<DATE>")
val rUSPhone = ("^\\d{3}\\W\\d+{3}\\W\\d{4}$".r -> "<USPHONE>")
val rNumber = ("^[-]?\\d+([.,]\\d+)*$".r -> "<NUMBER>")
val rTwoNum = ("^\\d+[-/=]\\d+$".r -> "<TUMBER>")
val rOrdinal = ("^\\d+(th|1st|2nd|3rd)$".r -> "<ORDINAL>")
val rPunct = ("[,;.:]$".r -> " <PUNCT>")      // if it is like "end.", should
return "end <PUNCT>"
val rLine = ("--+".r -> "")                   // underlines like -----------
```

In addition, the list of English stop words of the nltk[1] was taken and all of these words replaced by the single token <stop>.

These two measures reduced the vocabulary by 135 words.

In a further step, the provided Porter stemmer was applied, reducing the vocabulary by over 25'000 words.

During a final step, the terms in the training set with frequency > 30'000 were manually inspected, and these words were added to the stop-word list:

```
val HighFreqWords = List(
  //"share",     // -> 32117,
  //"bank",      // -> 30300,
  //"million",   // -> 45624,
  "would",     // -> 35039,
  //"percent",   //  -> 53790,
  "year",      //  -> 40026,
```

[1] http://www.nltk.org/book/ch02.html

```
  //"market",    // -> 35931,
  "said",       // -> 155872,
  "new"         // -> 33655
)
```

All these stop-words were converted to static regular expressions to speed up matching.

Overview vocabulary size
50'000 documents, total 9'208'748 tokens.

Tokenizing	Size vocabulary
Simple white-space tokenizer	150'125
Replace number-pattern matching Replace stop-words	149'990
Porter Stemmer Replace most frequent words	124'695

Finally, removing all rare words (which appear once or twice) reduced a lot the size of the vocabulary, which allowed us to improve computation time.

Challenges during development

The software was developed in a team of three, in parallel on intelliJ IDEA (2 users) and Eclipse (1 user), each with the Scala plugin.

A first challenge was to synchronize the build environment from a central build.sbt file to either the necessary project files of intelliJ (in folder .idea) or eclipse. The IDE specific projects had to be regenerated from the build.sbt numerous times.

Scala version incompatibility was a further challenge. It took several attempts to find a scala version that supports the given tinyir library and the breeze libraries. In the end, we opted for re-compiling tinyir under Scala 2.11.5 and provide the jar file.

Heap overflow was the most time-consuming issue. What could have been very precise single-line Scala code in many cases was turned into foreach() or map() constructs just in order to monitor the progress. Several helper classes were used for this (timer.scala, java.util.logging).

Long execution times and heap issues continue to be an important issue until the end. During testing, we often worked with strongly reduced zip files of the training, validation and test data with only a few thousand documents each.

Naïve Bayes Classifier

Our first approach of Naïve Bayes Classifier was based on simply precomputing class probabilities P(c) as well as conditional word probabilities P(w|c) for each class (and term).

We decided to implement a one-vs-all approach where we learn for each topic a classifier that can be applied to unseen documents to decide whether it contains that topic or not. In order to achieve this goal we had to also precompute P(w|not c) and P(not c). Applying this training step to the whole training set containing 50’000 documents (124698 terms / 696 categories) leaded to heap problems.

In order to reduce the needed heap space we introduced a smarter way of preprocessing. Instead of precomputing all the conditional word probabilities we only determine the term frequencies for each document and in addition for each category a list of documents belonging to this category (both stored in normal Scala Maps). While labeling unseen documents we then on the fly compute the conditional word probabilities based on this information. With this approach we have a slightly slower process of labeling new documents but on the other hand have the advantage of not running into heap problems while keeping everything in memory. In order to come around problems with zero probabilities we applied the Laplace Smoothing.

Applying our Naïve Bayes Classifier for all the training documents resulted in the following averaged precision, recall and F1 score:

Precision: 0.6947626720108807
Recall: 0.675971500225758
F1: 0.6464305694437176

We also tested the impact of applying an additional filter step and removing all the terms with a collection frequency smaller than 5. Because of a resulting lower F1 score we removed this filter step in the final solution.

Logistic Regression Classifier

For the Logistic Regression Classifier we introduced a more efficient way of keeping the information about the term-frequencies for each document by introducing a new class (RCVDataSet) which stores the term-frequencies as Vectors (Class DenseVector of breeze). RCVDataSet works on the indeces of the documents and labels respectively, so after testing these have to be converted back to document IDs and label names again (DocIndex, ClassIndex).

Our Logistic Regression Classifier is an implementation of gradient descent logistic regression described in the lecture slides. For every class we train a one-vs-all classifier by randomly picking one data point from the training set in each round and applying a gradient step. For this gradient step we used the squared loss instead of the log likelihood as mentioned in the lecture material.

Applying 10’000 iterations over 20 training rounds with a learning rate of 0.005 leaded to the following average precision, recall and F1 score:

Precision: 0.8751904761904755
Recall: 0.4326508944272101
F1: 0.5426678964740116

Note: In order to decide whether an unseen document belongs to a class or not we used a threshold of 0.5.

In order to improve this result we tried to apply the hint for imbalanced classes given in the project description. But this didn’t increase F1 score why we skipped this extension for the final run.

SVM Classifier

We tried several algorithms to compute the SVM classifier. The large amount of data and the bag of word vectors size was a computation challenge.

To train our model, we used the early stopping heuristic: we keep training until the validation F1 stops increasing.

First, we chose to start with an online version, to be able to stream data into the algorithm. We implemented the Online Convex Problem version of SVM (OCP). We obtained the following results:

Precision: 0.8324802425188552
Recall: 0.7808960530062035
F1: 0.7848252064349883

Secondly, we implemented the Pegasos algorithm. It allowed us to add a batching system. However, the results were not improved and the computation was longer than OCP. We obtained the following results:

Precision: 0.9057903403219517
Recall: 0.7196410873440543
F1: 0.7752180051080029

We thought that the problem might be represented with a non-linear function so we implemented the kernelized version of Pegasos. However, the computation time was too long.

So we tried to approximate the kernel algorithm with Random Fourier Features but as the bag of words approach creates high dimensions vectors, the solution did not fit in memory.