Automatic Identification of Participants in Discussion Groups

Jakob Carlsson
Väpplingv 7
227 38 Lund
Sweden
dat04jca@student.lu.se

Bobo Wieland
Transtigen 39
262 41 Ängelholm
Sweden
bobo@bitbob.biz

January 23, 2006

Abstract

The idea behind this project was to see if you could create a system that could tell you who wrote a certain piece of text. The method that we used for this was to encode the text as numerical data with an id for each word followed by it's frequency in the text. The numerical data was then fed to an SVM that predicted the author of the text. This report briefly discusses information extraction from the internet and describes the thoughts behind the java application RUU.

1 Introduction

The project idea was to see if there was any possibility to create a system that could tell you who wrote a certain piece of text. It was an interesting area that not too many people had thought of. We early decided that the best way to get data to test the system was to download it from discussion forums on the internet. We also thought about how to implement the system and found that the easiest way would be to use Support Vector Machines (SVM) for the classification of the text.

At this point we started collecting test and training data for the project from discussion forums on the internet. At an early stage we decided that our first goal would be to have a system that could see the difference of two people in a discussion between only these two; our second, and final goal to see the difference between several people in a discussion between these and other people as well. At this point we also started writing our system and we needed a name for it, after a while the system was named RUU (pronounced: Are You You) because it can tell if a text realy is written by a certain person.

1.1 Support Vector Machines

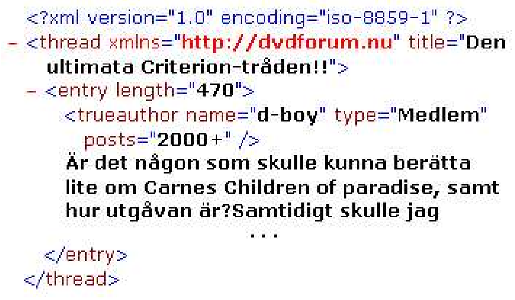
SVM is a method to classify data using vectors and mathematical models. We downloaded LIBSVM [1] and used it in the project because it is quite big to write your own SVM. Since we are beginners on using SVM for data classification we had to read through the beginners guide [3] to get a good grip of how to use SVM.

2 Application Structure

In the following section we will first briefly explain how we gathered and formated our test data (Section 2.1). We will then move on to explain in more detail how our main application - RUU - works (Section 2.2).

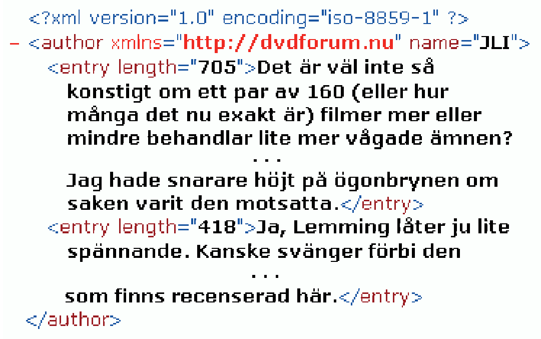
2.1 Information Gathering

We had some different web forums in mind when we started to work on this project but soon decided to use the swedish spoken forum on dvdforum.nu [2]. dvdforum.nu is a web site for movie enthusiasts and has many active members in their forums. It's been online since 1996 and is to some extent a closed



```
<?xml version="1.0" encoding="iso-8859-1" ?>
- <thread xmlns="http://dvdforum.nu" title="Den
    ultimata Criterion-tråden!!">
  - <entry length="470">
      <trueauthor name="d-boy" type="Medlem"
        posts="2000+" />
      Är det någon som skulle kunna berätta
      lite om Carnes Children of paradise, samt
      hur utgåvan är?Samtidigt skulle jag
                        . . .
    </entry>
  </thread>
```

Figure 1: Example of Criterion.xml



```
<?xml version="1.0" encoding="iso-8859-1" ?>
- <author xmlns="http://dvdforum.nu" name="JLI">
    <entry length="705">Det är väl inte så
      konstigt om ett par av 160 (eller hur
      många det nu exakt är) filmer mer eller
      mindre behandlar lite mer vågade ämnen?
                        . . .
      Jag hade snarare höjt på ögonbrynen om
      saken varit den motsatta.</entry>
    <entry length="418">Ja, Lemming låter ju lite
      spännande. Kanske svänger förbi den
                        . . .
      som finns recenserad här.</entry>
  </author>
```

Figure 2: Example of JLI.xml

domain since most discussions concerns movie related subjects. It suited our needs perfectly since we needed a long forum thread with many posts for our tests and as many posts as possible from forum members active in that particular thread.

After finding a suitable test thread, Den ultimata Criterion-tråden!! (The ultimate Criterion[1] thread!!) with >1100 posts over a period of four years, we created a simple PHP script to parse the thread data. The script simply looked at the HTML source and split the text by the tag pattern of the code. After stripping out all HTML tags the script saved the data in a convenient XML format as Criterion.xml (Figure 1).

By counting the number of posts from different users in Criterion.xml we could easily single out our test subjects. We will reference our four test subjects in the remaining of this rapport by their screen names - d-boy, JLI, ola-t and von Krolock.

When we knew the screen names of our test subjects it was easy to modify our existing PHP script to loop through the threads at dvdforum.nu, catching all posts by either subject and append the post to a specified XML file; one for each of our five subjects with their screen names as file names (Figure 2). At this point we also decided not to use posts consisting of fewer than 250 characters.

While being an easy and simple way of gathering data it wasn't the most efficient one. We started our loop counter at the then most recent thread id and went backwards in time from that point on, looping through each thread, existing or non-existing, and each page of each thread (in case it was spread out over many pages due to it's number of posts). After letting the script run for 24-hours straight and looping through approximately 50.000 threads we hoped that we had gathered enough data and aborted the information gathering.

After doing some labour-some manual edits to our test files, removing the signature each subject put last in all of their posts that unfortunately was impossible to remove automatically (at least with our simple PHP script), our information gathering was complete.

2.2 RUU

RUU is our main java application that converts our XML files to files that can be used with SVM. It does not, however, simply convert from one format to the other but tries to format the text in the XML files to maximize the final svm prediction rate by apllaying some simple rules.

RUU does two passes over the supplied data, first building a dictionary of tokens and the in the second pass generating the output.

We will now explain in more detail four parts of the application; The Sink (Section 2.2.1), The Tokenizer (Section 2.2.2), the Dictionary (Section 2.2.3) and the svmFileCreator (Section 2.2.4).

2.2.1 Sink

RUU uses a SAX parser to parse the XML files. We choose to use a SAX parser rather than a DOM parser since we, at the time of the decision, didn't know how large our XML files where to be and if

[1]The Criterion Collection is a line of authoritative consumer versions of "classic and important contemporary films" on DVD (and on Laser Disc pre DVD era). The quality of these releases - from picture and sound to packageing and included extras - are always top-notch.

they would be well-formed or not. In contrast to a DOM parser a SAX parser reads the XML files a bit at the time. This means that the whole XML document will never be in the computers memory (which could cause problems if the documents are huge) and it also guaranties that until the parser encounters an error in the XML syntax it will parse the data. A DOM parser would abort the attempt to read the XML document immediately.

The second reason to use SAX rather than DOM was the simplicity of our XML files and the knowledge that we would only use the parser to read the data - not to manipulate it.

The Sinks main purpose in our application is to handle the data sent from the SAX parser. In it's basic form it had three important methods for handling XML data; one for handling data sent when a XML tag is opened, one for handle data when a tag is closed and one for handling character data.

The actions the Sink takes is different depending upon what stage of the process RUU are in. In the first pass that RUU does over the files the Sink sends the data from all the documents to the Dictionary. In the second pass it sends the data to the svmFileCreator instead. In both cases all character data is processed by the Tokenizer.

2.2.2 Tokenizer

A regular tokenizer breaks a character stream into tokens - separate words - and sentences [4]. Our tokenizer breaks the string of words supplied by the Sink into tokens, but sentences are not generally taken into account. Before storing a token our tokenizer turns all regular characters into lower case and on top of this scans the input string for specific patterns and possibly add some of three special tokens;

1. #SMILEY# - Some regular ascii smilies - i.e. :) or ;-(- are recognized and are replaced with this token.

2. #NET_SHORT# - The most regular internet short forms for different expressions - such as lol (laughing out loud) or isf (swedish for i så fall (in that case)) - adds this token to the list. It does not, as with #SMILEY#, replace the old token.

3. #NON-CAPITAL# - While sentences are of no interest to us in general, the Tokenizer does check ff the first character of a sentence is in lower case and if that is the case adds this token to the list.

This special treatment of the input string is done since it will, supposedly, help SVM to predict who is who more accurate. To explain our thoughts behind this we have to see ourselves as long users of the internet. We've grown accustom to the way people express themselves and it feels naturally to categorize peoples use of words and symbols.

Some users use a lot of smilies in their post - some use non. Some use a lot of internet abbreviations - some, again, use non.

Also, to abuse the use of capitalized letters means, in net-language, to shout. And excessive shouting is often followed by excessive amounts of angry replays to the point where the original poster learns to fear Caps-Lock. This is why we treat the few capitalized words as a non-capitalized word since it won't be a net users regular way of writing.

Finally, many - but far from all - regular forum posters have the bad habit to, more often than not, forget to start sentences with a capitalized letter. If this is because of laziness or fear of Caps-Lock, we won't elaborate further on #SMILEY#.

2.2.3 Dictionary

In the first pass that RUU does over the XML files all tokens returned by the Tokenizer is sent to the Dictionary. The Dictionary gives each token a unique label - starting with 1 for the first token, 2 for the second and so on - and keeps track of the total number of times - it's frequency - a token is used throughout the XML documents.

We've also added the functionality to store bigrams (word pairs) instead of unigrams (single tokens), or to store both bigrams and unigrams at the same time, in the Dictionary. Since bigram and n-gram predictions themselves can be used to check authorship of texts this seemed a resonable thing to do. However - later tests showed us that using bigrams instead of unigrams gave a great (huge!) performance hit and a much worse final results, so we won't say much more on this matter.

Number of subjects	Method used	T-limit	C / g(e-5)	CV rate	Correct/ Tot(Valid)
2	Unigrams	1	128 / 12.21	94.84	71/83
2	Bigrams	1	32 / 12.21	92.46	42/83
2	Uni+Bi	1	32 / 12.21	94.05	42/83
2	Unigrams	2	32 / 48.83	94.84	70/110(83)
3	Unigrams	2	512 / 03.05	87.09	80/110
4	Unigrams	2	512 / 03.05	81.10	101/242
4	Unigrams	2	512 / 03.05	81.10	118/1090(242)

Table 1: Final results

2.2.4 svmFileCreator

After the Dictionary is populated RUU parses the XML files a second time. This time the Sink keeps track of the author of the entries (forum posts) in our test files giving them unique labels, once again starting at 1 for the first author, 2 for the second (and so on and so forth)...

After character data has been tokenized the tokens for that particular user and entry is stored along with the frequency for each of the tokens. In this case the frequency is the number of times the token has appeared in the current entry and not the total frequency stored in the Dictionary.

When the Sink encounters the end element of an entry the svmFileGenerator is used to generate an SVM suitable representation of the stored data for the entry in question. This data is appended to a train file, later to be used by SVM to build a prediction model.

When all test files are parsed and the train file is completed, the procedure is repeated once more for the XML file that should be used to test our SVM model. When the Sink encounters a entry in this file that has not been written by one of our test subjects it - depending on what we've chosen - either completely discards the entry or generates a new unique label for it (the Sink does not keep track of other authors of our test files so two entries posted by he user bitbob will not get the same label - in fact we increment the label value by one each time we come across a post that is not from one of our test authors)).

As before, when encountering the end element of an entry the svmFileGenerator comes into play; generating an SVM suitable representation of the data that gets saved to disc.

With the two files created - the train file for building the SVM model and the test file to test the model on - RUU is done and it is time to let easy.py give the CPU a run for it's money.

2.3 easy.py

Throughout the project we've followed the guide lines given in the paper A Practical Guide to Support Vector Classification [3] and their suggestion to use easy.py for fast and easy, as well as good, results.

easy.py is a python script that comes with the LIBSVM package. It simplifies the use of SVM by automatically fine tune your files and finding suitable values for constants used by the RBF kernel function. It creates the SVM model and and runs the model on a test file, if specified.

All of our result data is data given by easy.py

3 Results

While generating train and test files in RUU, we've tried some different approaches, just to see what gave the best result. In most of our early tests we've used only two of our test subjects; JLI and von Krolock (since we had most data from these two). After getting some initially poor final results we decided to skip the test on Criterion.xml and use a subset of JLI's and von_Krolock's test files instead. We'd gotten a fairly good CV rating[2] on them (between 80- and 95%) so it felt like a good

[2]CV stands for Cross Validation and the CV rating is the probability that SVM guesses right when it uses parts of the training file as test data (used when easy.py tries to find the optimal constants)

place to start. We cut out 25% of each file and put randomly each entry of the data in a new test file.

After sorting out a small error (causing a major drop i prediction accuracy) in our code we could see a fairly good result with a prediction rate approximately 10% lower than our CV value.

From this point we concentrated on tweaking our variables in RUU for best results. As mentioned before we had an idea that maybe using bigrams instead of unigrams would give a better result. This wasn't the case at all and we noticed a huge performance hit as well as really poor results.

We also, at one point, tried to raise the frequency of each token by a power of 2 or even 3. While not taking SVM much longer to process it gave a small decrease in prediction accuracy (minus 1- or 2%). So, as the with the case with bigrams, we soon discarded this idea.

The final modification we tried was to put a lower limit on how many times, in total a token had to be used in the text to be considered. Trying different values we decided to use 2 as the lower limit, making RUU discard all tokens that was only mentioned once in the text. This will include non-standard miss-spellings, unusual names and other rare character combinations.

Table 1 shows some of our test results. The left side of the tables is values we've decided in RUU (T-limit being the lower limit of tokens mentioned in the previous paragraph) and the right hand side is the values easy.py calculated for us; C and g being the constants used by the RBF kernel function and CV being the cross validation accuracy. The last column is the final result of applying the train model on the test file. Correct is the number of correctly predictions and Tot being the total number of predictions made (or total number of rows in the test file). However, when supplied with a test file that had entries from others than our test subjects SVM allways predict each row as one of the known subjects. In these cases the number of valid entries (entries written by any of our test subjects used in the test) is written in parenthesis.

4 Conclusion

One of the most important things that came up during the project was that it wouldn't be possible to use this implementation in a real-time system, but that was never our intention with RUU. To have this in a real-time system we must have a model created before and a direct link between RUU and the SVM so that we don't need to create any files or run an external program in any part of the process. Another thing is that it takes a lot of CPU and time to create the model and when finished the model is quite large so you don't want to create a new model to often.

Finally; our test results show us that the more people that are involved in a disscusion the more our final results will suffer. Trying to separate and keeping track of hundreds of people at once probably will prove to be impossible simply using SVM.

Acknowledgements

We would like to thank Richard Johansson for all the support and help with the project. Special thanks to Chih-Jen Lin for fast response and invaluable help and advice.

References

[1] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[2] dvdforum.nu. www.dvdforum.nu. A swedish discussion forum on `http://www.dvdforum.nu`.

[3] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. *A Practical Guide to Support Vector Classification*. `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

[4] Pierre Nugues. *An Introduction to Language Processing with Perl and Prolog*. Springer, 2005.