NTNU – Trondheim
Norwegian University of
Science and Technology

Evaluating the use of Learning Algorithms in Categorization of Text

Alf Simen Nygaard Sørensen

Master of Science in Informatics
Submission date: June 2012
Supervisor: Trond Aalberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

The research area of supervised learning have seen a lot of study and improvements in the last couple of decades. We wanted to see if it would be possible to use supervised learning on a real-world case from Difi. They have an ontology with categories and a set of documents coupled to these categories, and they wanted to automate the process of adding new uncategorized documents to the ontology. The amount of documents they want to add is huge and to do this by hand would take lot of time and manpower.

We have tested this by developing a small prototype that used the corpora of labeled documents with some different learning algorithms to see if the results would be satisfactory. We conclude that while the system would indeed make it easier for someone to classify unlabeled documents, it can not work totally autonomously based on the relatively small amount of documents and large amount of categories that are in the ontology.

Norwegian:

Lærende algoritmer brukt på eksempel-dokumenter er et forskningsområde som har hatt ett stort fokus og dermed fått mange forbedringer i de siste årene. Vi ville se om det var mulig å bruke disse på et reelt problemcase fra Difi. Difi har en ontologi med kategorier og ett sett med dokumenter som er koblet opp mot disse kategoriene, og de ville automatisere prosessen med å legge til nye ukategoriserte dokumenter inn i ontologien. Mengden med dokumenter de vil legge til er så stor at hvis man skulle gjøre det for hånd ville det ta veldig lang tid og kreve mye arbeidskraft.

Vi har derfor testet om dette lar seg gjøre ved å utvikle en prototype som bruker denne samlingen med allerede kategoriserte dokumenter til å teste noen forskjellige lærende algoritmer for å se om det er mulig. Vi konkluderer med at selvom systemet nok ville kunne gjøre jobben lettere, så kan den ikke jobbe helt autonomt grunnet den relativt lave mengden med dokumenter og store mengden kategorier som er i ontologien.

Preface

This master's thesis was carried out within the Information Management (IF) group under the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

Trondheim, June 1, 2012

Alf Simen N. Sørensen

Contents

List of Figures

List of Tables

Code listings

Acknowledgements

I would like to thank friends and family for continuous support and encouraging words throughout the process.

I would also like to thank my supervisor, Trond Aalberg for his counseling and guidance which has been invaluable.

1 Introduction

1.1 Background and motivation

The amount of information found on the Internet has increased vastly in the last ten years and information from municipalities have also increasingly been made available on the web. Difi (Agency for Public Management and eGovernment) operates a website called www.norge.no which indexes web-pages with information about the public sector in Norway. They also have a ontology called LOS, which contains tags that are relevant for organizing municipal information. Unfortunately, as of now there are almost no information here, except for some handpicked web-pages, from all the different municipalities that publishes information on their websites. One of Difi's goals are to be able to index this information based on the ontology so that it will be easier for everyone to find relevant information about what they are looking for.

Given that Norway contains 429 municipalities, each with their own web pages with lots of information relevant to their geographic location and with vastly different themes, the amount of information is huge. To be able to solve the gathering and categorization of this information, without having to rely on a huge amount of manpower, the usage of *information retrieval* (IR) techniques would ease the workload. Within IR there is a field called *text mining* which use a learning algorithm on a corpora that's already classified to "teach" the program what classes (we may sometimes use "category" instead of class) documents belong to. Then you can use this to classify new documents that have an unknown class. Since LOS already contains documents that have a defined class, we can use these to teach the algorithm about the classes and thus be able to classify new documents with this later on.

The key problem would then be the relatively low amount of documents per class that we are to learn from. This will impede on the precision and accuracy of the results.

1.2 Problem definition

Based on this background and motivation the goal of this thesis can be summarized as:

> Find out if IR-techniques can be applied to the current amount of information in the ontology so that it would be easier to add new information into the system.

This can then be broken down into several questions we would like to have answered:

1. What techniques could be used and what requirements do we have for these?
2. Are there enough documents in LOS to be able to use these techniques effectively and will we be able to categorize new documents correctly?
   (a) Subsequently, what amount of documents will be required to achieve good enough results so that this kind of system can be used.
3. Are there any pre-made solutions that could be used?

The task at hand is as such not to develop a whole new system or to invent something new to solve this problem, but to see if the methods that already are available are suitable for this specific task.

1.3 Our case

We have mentioned the LOS-ontology several times and we would like to better describe what it is and how it's built up. LOS is a ontology with content relevant to public services like work done at Town Hall etc. It contains 504 keywords, the classes, used on different portals like web pages and other public informations systems. It's structure is built up of 15 main themes, 78 sub-themes and then the 504 classes divided up under the sub-themes. It also contains more than 1500 search words that link to these classes and themes. There are also links to public services (i.e. web pages) that are relevant to the keywords. It is these documents that are linked to that we shall use to teach the system about what kind of document belong to a certain class.

Part of it's structure:

- Arbeid (main theme)
  - Arbeidsliv (sub-theme)
    - Arbeidsavtale (class name)
      - Documents
    - Arbeidsgiver
      - Documents
    - Arbeidsmiljø
      - Documents

        * ...
    - Arbeidssøking og rekruttering
    - ...
- Barn og familie
- ...

More about LOS and it's content can be found in section 2.4.

1.4 Summary results

The results give a clear picture that shows that the amount of documents that are already classified in LOS are not sufficient. The best result we got was 14.8% correctly classified of the test-set and this is not usable at all. Even when we tried reducing the amount of categories by using the other levels of LOS (main theme and sub-theme), we only managed to get up to an average of 51.6% correctly classified. This on the other hand could be used as an indicator to help in the process of manually classifying new documents, but it would not be using the full potential of the ontology since it only uses the 15 top-level categories.

We estimate that the ontology would have to have upwards of 50 documents per category to be able to get good enough results for an automated system. This would mean that they would need (504 categories times 50 documents) 25200 documents in the system. This is a huge increase from the 1105 documents that are in it as of now and it would require a great deal of time and manpower to achieve this.

1.5 Brief overview of the thesis organization

The rest of the thesis is organized as follows:

Chapter 2 Chapter 2 provides the necessary theoretical knowledge to be able to understand the rest of the thesis. It is not meant as a thorough explanation of all details regarding the topics, but will be enough to understand our approach. It also refers to books and articles that goes deeper into the topics if the reader would like to get more information.

Chapter 3 In this chapter we will take a look at some of the different systems that are made for the task we are solving. We will also talk about some of the technologies that we have used in our implementation.

Chapter 4 Here we will show how we implemented our approach and also take a closer look at the algorithms used. Problems we have met during development and how we solved them will be described. Code examples will also be shown and described.

Chapter 5 This chapter will take a look at the results and how good our implementation are compared to other implementations. It will also discuss what needs to be done in future work to improve on the results.

Chapter 6 At last we will draw a conclusion based on our results and take a look at the testing to see what can be improved and what to be researched in the future. It will summarize the thesis as a whole.

Appendix A Contains Java code that are referenced in the thesis when there is a need to explain in more detail how parts of the system works.

Appendix B Describes the content of the enclosed zip-archive with code and folder structure representing LOS that contain the files used to teach the system.

2 Background research

This chapter will give relevant information that is necessary for the reader to understand the basis for this thesis. It contains references to cited papers and books so that one can do more in depth reading if one wants to get an even better understanding of the topics. The goal of this chapter is to be a basic introduction to the relevant topics so that the rest of the thesis is understandable. The basis of the thesis is the broad field of information retrieval (referenced as IR from now on) and within this the field of machine learning. Thus we will have a brief look at these topics here.

2.1 Information retrieval

The term information retrieval has several different definitions and can be anything from searching for a physical book in the library to searching for a specific file on your laptop or just finding out where you left your car keys at home. But to be more specific and relevant to this thesis the definition from the book "Introduction to Information Retrieval" [19] is befitting.

> "Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)."

In essence it can be though of as this: You give some input to a search system containing documents and receive information that is relevant to this input. In our case it would be to input a class that we want to find information about and receiving documents that have relevant information about said class. At least that is all there is for the user of the system, but in the background there could be any number of processes going on at any given time. For the system to be able to give back the relevant information to a query it would have to have indexed a lot of documents based on classes. In a somewhat normal use case this would be done by searching through the documents for occurrences of the class name in the documents and storing this in an index for later retrieval.

2.1.1 Text document indexing and retrieval

The basis for most IR are the indexing and retrieval of text documents. The reason for this are the vast amounts of text documents and textual information stored away in libraries and in later years on the web. And because of the large amounts of information there were a necessity to be

able to develop techniques for extracting relevant information from this in an easy manner. To search through all this information by hand would be an immense task which would take a lot of time and manpower. Since the basis for our case is the text document, lets have a look at what a typical text document would be in our case.

Most would probably describe a document as a physical material object like a book or a piece of paper with text on it. Mentioned by Michael K. Buckland [4] a document was described in 1935 by the International Institute for Intellectual Cooperation, an agency of the League of Nations as:

> "Any source of information, in material form, capable of being used for reference or study or as an authority. Examples: manuscripts, printed matter, illustrations, diagrams, museum specimens, etc."

This fits rather well with the way it is ordinarily described, but in our case the document are not of a similar material form but rather in a digital form as bits on a hard drive. But the main part of the document definition still holds, it contains information, information that is meant to be used for reference or study. The document we are using are a digital collection of words that as a whole defines a web page. It contains information about the look of the web page and also the information visible to the user.

To be able to compare similarities between documents and a query from the user the documents have to be indexed. The most plain and easy way of doing this is with a boolean retrieval system. Documents get indexed by sets of keywords and the search can then use boolean operators like AND, OR, and NOT to retrieve documents containing (or not containing) certain keywords. A much used way of storing the information about what documents contain what keywords are the *inverted file*. An inverted file contains contains all the words and for each word it lists all the documents that contain said word. Thus, if the user wants all the documents containing the word "computer" it is as easy as to just return the list of documents that contain said word. If the user wants documents containing "computer" AND "code" it then merges the two lists for these two words and returns those documents that there are duplicates of in the list. For the OR query it would merge and return all documents that have no duplicates and for the NOT query it would return all documents from list 1 that is not in list 2. A overview of the process of finding documents that fits a query can be seen in Figure 1 on page 7.

This way of indexing is good when you are looking for certain words within a document, but in our case we would like to classify a document based on the combined content of the document. Thus you may search for the class name but miss out on a lot of relevant documents because the class name is

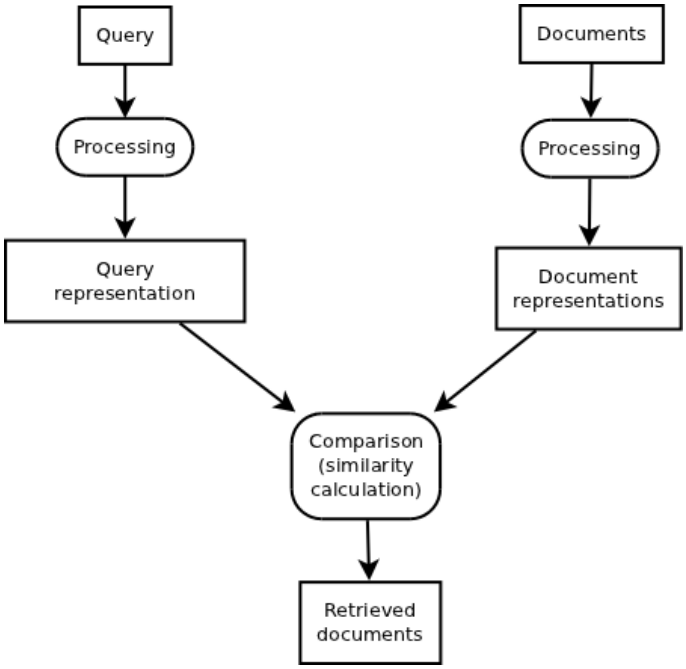


Figure 1: Information retrieval process

not specifically mentioned in the text of all relevant documents. Because of this we would have to have a different approach when classifying documents. Before the era of computers and computer IR-techniques this was done by hand and we will take a look at this now.

2.1.2 Classification by hand

The way such classification has been done since the beginning has been by hand. A person had to either have enough background information about a document from before, or go through the document and try to figure out what class it belonged to. Librarians and other similar professions have been doing this for as long as classification has been around and it is a very manual, labor-intensive work. It is relatively effective as long as the amount of documents are small, or you have small amounts of documents to classify at any given time, but as soon as it increases substantially it will be less and less cost-effective.

The case in our thesis involves trying to classify thousands and thousands of web pages in the most effective way possible, and thus we need the next chapters techniques to achieve this.

2.2 Text mining and machine learning

As mentioned before, the huge amount of documents and information to classify becomes a problem when it is done by hand. The amount of documents also increase at a high rate because we are able to save information much easier now than before when doing it digitally. And somewhere in all these documents lay the information that we would like to get a hold of. As opposed to regular IR where you mainly just look for certain words in a document, in data mining you look for patterns in the information. The book "Data Mining: Practical Learning Tools and Techniques" [34] define data mining as:

> "The process of discovering patterns in data. The process must be automatic or (more usually) semiautomatic. The patterns discovered must be meaningful in that they lead to some advantage, usually an economic one. The data is invariably present in substantial quantities."

And it is these patterns that allow us to make predictions about new data. To describe what a pattern is is not straightforward, so we will rather give an example.

The following are the content of one of the web-pages that we get from LOS that has been preprocessed (for information on preprocessing, see 4.3.1):

direktoratet naturforvaltning jakt fangst hopp sideinnhold hopp hovedmeny hopp sidemeny hopp søkefelt hjem nettstedskart english kontrast friluftsliv aktiviteter allemannsretten friluftslivsområder fritidsfiske jakt fangst naturarven verdiskaper norges nasjonalparker nærmiljøsatsing pilegrimsledene stillhet verdiskaping naturbasert reiseliv klima klimaendringer norge klimaeffekter land klimaeffekter ferskvann klimaeffekter havet kysten klimaendringer friluftsliv klimaendringer svalbard arealplanlegging arealbruk klima overvåking effekter klimaendringer naturmangfold naturmangfoldets betydning internasjonalt samarbeid klima database klimaeffekter naturmangfold energi miljøpåvirkning fremmede arter genmodifiserte organismer gmo hav kyst inngrepsfrie naturområder norge inon internasjonalt miljøsamarbeid kartlegging natur kulturlandskap laks sjøørret sjørøye landskap naturindeks norge naturovervåking trua arter naturtyper vann vassdrag verneområder verdien naturmangfold økosystemtjenester forvaltningen arealplanlegging grønnstruktur markaområder naturforvaltning kommunene sikring forvaltning friluftslivsområder strandsonen tilrådninger verneområder tilskuddsordninger publikasjoner brosjyrer dn faktaark dn håndbøker dn notat dn rapporter dn utredninger oppdragsrapporter annet skjema regelverk naturmangfoldloven fjelloven friluftsloven genteknologiloven lakse innlandsfiskloven lov statlig naturoppsyn motorferdselloven plan bygningsloven viltloven handel trua arter cites kart kartkatalog lakseregisteret naturbase rovbase vannmiljø vann nett villreinbase temakart wms kartlag inon kart abonner nyheter ansatte arrangementer høringer innsiktsarkivet kontaktinformasjon ledige stillinger logo direktoratet naturforvaltning media presse nyhetsarkiv offentleg journal offentlige anbud direktoratet naturforvaltning personvernerklæring tema friluftsliv jakt fangst jakt norge fangst fangstredskaper jakthund jaktstatistikk jakttider jegeravgift rapportering jegerprøve opplæringsjakt kvotejakt lisensfelling preparering vilt skyteprøve småviltjakt storviltjakt utdanning jegerprøveinstruktører våpen ammunisjon snarveier friluftsliviskolen st mld nr friluftsliv veg høgare livskvalitet friluftsliv jakt fangst jakt fangst sidene finner informasjon retten drive jakt fangst norge lover regler gjelder jegere fangstfolk jakt fangst viktig del friluftslivet norge forvalter viltressursene våre viktig skjer måte bevarer naturens produktivitet artsmangfold direktoratet naturforvaltning sammen miljøverndepartementet overordnede ansvaret viltforvaltning arbeid knytta utformingen regelverket innen forvaltning viltressursene utøvelsen jakt fangst siste innen jakt fangst sist oppdaterte artikler jegerprøve opplæringsjakt utdanning jegerprøveinstruktører storviltjakt småviltjakt skyteprøve publikasjoner jakt norge hunting in norway jagen in norwegen skitt jakt strategi forvaltning hjortevilt jakt norge hunting in norway jagen in norwegen brosjyre informasjon jakt jakttider flere publikasjoner lover forskrifter fjelloven friluftsloven viltloven sentrale forskrifter viltloven lokale forskrifter viltloven nyheter prøve jakt pil boge god human sikker jakt ettersøkshundene må bedre jegere får jegeravgiftskortet ny utdanning jegerprøveinstruktører nyhetsarkiv eksterne lenker rapporter jakt fangstutbyttet ssb jegerregisteret hjorteviltregisteret se direktoratet naturforvaltning besøksadresse tungasletta postadresse postboks sluppen trondheim tlf faks post postmottak dirnat nettredaktør guri sandvik kontaktinformasjon media presse kontakt nettredaksjonen ledige stillinger ansatte nyhetsbrev

As you can see, it is not really too easy to see what kind of content or what category this belongs too without reading through it thoroughly. This be-

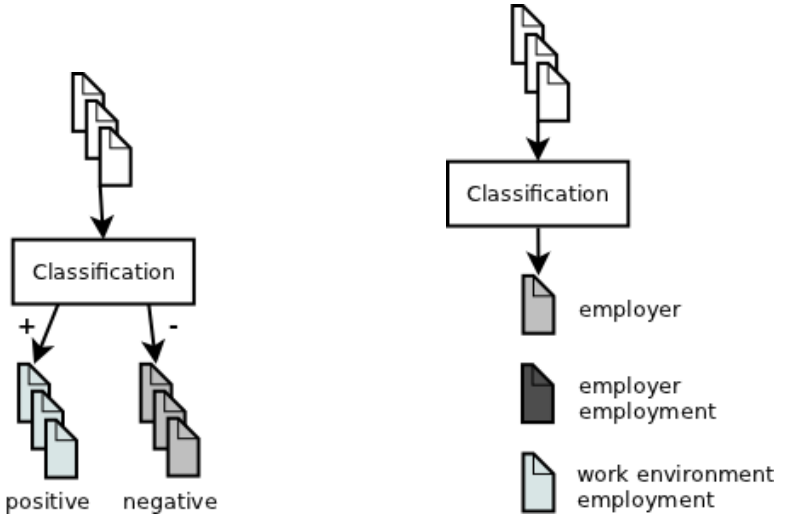
comes a tedious and difficult task to do by hand. One way of finding a pattern in this text is by summing up word-occurrences. When we have done this we can highlight the words that are more common. The result would be this:

Now we can much more easily see a pattern emerge, the words “jakt”, “fangst” and “norge” are words that are very common in this text, and thus we could guess that the document belongs to a category about hunting in Norway. This is rather spot on for the actual category it belongs to in LOS, the actual category is called “viltforvaltning” and is indeed about hunting.

2.2.1 Classification

The classification we are doing can be seen as a sort of subject classification because we want to predict the topic or subject of a web page (or several). There are also a classification type called functional classification and sentiment classification. With the first type you are looking for the role that a web page have. It can be roles like a personal web page, landing page or a course page for a university. The second type looks for the opinions that the web page shows about a certain topic. For example opinions about a political topic or the stance the author of the web page has about religion.

Some classification problems can only be classified into two categories like “positive” or “negative” and this is called binary classification. On the other hand (as it is in our case) there are a lot of classes and this is called multi class classification. Also, the documents we classify may be categorized within just one (single-label classification), or multiple classes (multi-label classification). In our case we are to classify each document to one class and only one. Lastly we can have very clear “boundaries” for a documents class, where it may either be within a class or not, or it can have intermediate states where it may be having 80% in common with one class and 35% with another etc. This is called hard and soft classification. Some examples of



(a) Binary classification

(b) Multiclass, multilabel, hard classification

Figure 2: Classification types

this can be seen in Figure 2 on page 10. We will be using the classification shown in 2b.

The task we set out to do when talking about classification in our case is to create a sort of mapping between the documents in our corpora and a set of class labels, and thus we work with subject classification. This mapping is generally called a model. In general this way of creating a model by using already classified documents are called *supervised learning*. The resulting model can then be used to automatically determine the class of new documents that we have not been assigned any class yet. We can go a little more into detail about the steps involved in this:

1. *Data collection and preprocessing*
   Here we collect all the documents that are already labeled with classes, also called crawling. After this we have to process the documents so that unnecessary information are removed, we "clean" the documents. i.e. this means removing stop words, common words, punctuation symbols and HTML tags and do stemming on the words.
   Lastly we divide the collection of documents into two subsets:

   - *a training set*
     This set will be used to create the model and may also be divided into two subsets; one will be the part that actually creates the model, and the last one used to fine tune the parameters for the

model learning (called a validation subset).

- *a test set*
  This set is used to test if the model are good enough.

2. *Building the model*
   This step is where you actually "learn" or "train" the model with the use of a learning algorithm. It are often an iterative process where you fine tune parameters of the feature selection and the algorithm. Steps to iterate:

   (a) Apply an evaluator that chooses the appropriate features (words)
   (b) Apply the learning algorithm to get a model
   (c) Validate the model with the validation subset from the training set
   (d) Fine tune parameters

3. *Testing the model*
   At this step we apply the model to the test set from step 1 and we compare the predicted classes with the actual classes of the documents. It should be noted that here the classes of the documents are only used for evaluation and not used by the learning algorithm as in step 2.

4. *Classification of new documents*
   When the model are considered good enough it can be used to classify new documents that have no assigned class.

2.2.2 Algorithms

In this section we will give some details about different algorithms that we used and why we choose them over others. There will also be some general information about how the different kinds of algorithms work and how the different kind of types work.

General From the literature we can see some trends with the different kinds of algorithms. With decision trees, the dataset usually needs to be rather large to build a good classifier. Also, decision trees have problems when the amount of classes are very large. In these cases the tree would be too large to be searched efficiently and memory usage would be very high. On the other hand, Bayesian classifiers are much better at small datasets where it usually outperforms other classifiers. Support Vector Machines have shown very good results both in regards to efficiency and the domain it is used [28].

Naive Bayes are usually used as a baseline which to compare other algorithms up against. Thus we also use it here to see how well the other algorithms we have chosen perform compared.

Algorithm details We decided to test algorithms from several different ways of performing the learning. We used [6] and available algorithms in Weka as a basis for choosing which algorithms to test. Since decision trees showed such a poor performance and that the amount of documents we have are so small, we did not test any of these. The different algorithms we have chosen are *Naive Bayes*, *IBk, Random forest* and *SMO*.

Naive Bayes are a well-used classifier when classifying text because of it's easy of implementation and because it is fast. It has been found to not perform very good [35, 37], but we use it as a sort of baseline to see how the other algorithms perform.

Naive Bayes A Bayesian classifier is used to classify text work by correlating words in an already classified text with the class it belongs to and then using Bayesian inference to calculate the class of the new text. The Naive Bayes classifier assumes that all words are conditionally independent of each other given the class (see Figure 3 on the facing page). Meaning that we assume that the words are randomly distributed in the document and not dependent on the documents length or any other words in it. And despite this simplifying assumption it's still on par with a lot of more complex and sophisticated induction algorithms. According to [14] it gives accuracies that are comparable to those from rule-based induction methods in medical domains and that it also performed better than an algorithm with decision-tree induction (C4) in four out of five domains. It has been shown to compete well with more sophisticated classifiers [20] and it is because of this we will try it out on our documents as a sort of baseline for the other algorithms.

When classifying documents you want to know the probability p that a unclassified document D belongs to a given class C from the collection of classes already in the system. This can be written as:

$$p(C|D) = \frac{p(D \bigcap C)}{p(D)}$$

which by Bayes' theorem gives the statement:

$$p(C|D) = \frac{p(C)}{p(D)}\, p(D|C)$$

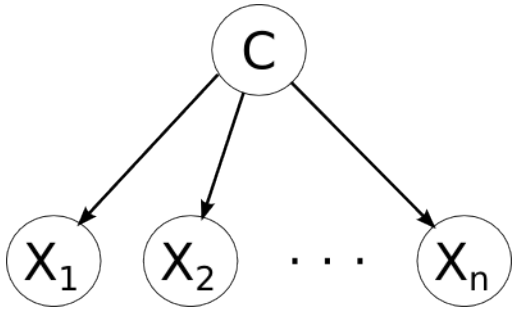


Figure 3: Naive Bayesian classifier as a Bayesian network
It's predictive attrs (X_1, X_2, ... X_n) are conditionally independent given the class attribute (C).

The probability for a class C is dependent on all features f and can be written as:

$p(C|F_1, ..., F_n)$

which by applying Bayes' theorem gives us:

$p(C|F_1, ..., F_n) = \frac{p(C)\, p(F_1,...,F_n|C)}{p(F_1,...,F_n)}$

this means that the posterior probability equals the prior probability for a class, multiplied by the likelihood for the features given this class, and this result is then divided by the evidence (the probability of the features).
Since the bottom of the fraction is not dependent on the class C and the values of all the features are known, it is a constant and may be removed.
Now we are left with the joint probability model:

$p(C,\, F_1, ..., F_n)$

which after applying the chain rule for repeated applications of the definition of conditional probability gives us:

$p(C) \prod_{i=1}^{n} p(F_i|C,\, F_j)$

where F_j is all other features except F_i, but since the algorithm assumes that all words are conditionally independent of each other given the class, we can remove F_j and ends up with:

$p(C,\, F_1, ...,\, F_n) = p(C) \prod_{i=1}^{n}\, p(F_i|C)$

The specific classifier from Weka that we use are weka.classifiers.bayes.NaiveBayes and are tested with some different options:

1. default options
2. with/without kernel estimation
3. with/without discretization of attributes

IBk IBk is a k-nearest-neighbor algorithm. One of it's main advantages are it's simplicity and ease of implementation. The algorithm works by creating vectors of the documents by using TF/IDF weights of words in the documents. When classifying a document it creates the vectors and then looks for the k-nearest neighbor in the vector space to determine what class it belongs to. It is a quite simple algorithm and it is one of the fastest algorithms we tested (see Table 4 on page 30). For more information read "Instance-based learning algorithms" by Aha and Kibler [1].
The specific classifier in Weka is weka.classifiers.lazy.IBk and we are testing with some different options:

1. default options
2. different number of neighbors used
3. let the algorithm choose number of neighbors

Random Forest One of the tree algorithms we have chosen are the random forest algorithm. It has shown good results in [6], so we would like to see it's performance in our case.
Random Forests work by "growing" a collection of trees from the learning data and then letting them vote for the classes they predict a document to belong to. So each tree generates a random vector that is independent of all the other past random vectors generated by other trees but it is built upon the same distribution. Then the tree is built using the training set and this vector, which results in a classifier. When a large number of such trees are generated, they all vote for which class a document belongs to.
Definition: *A random forest is a classifier consisting of a collection of tree-structured classifiers* $\{h(\mathbf{x}, \Theta_k),\, k = 1, ...\}$ *where the* $\{\Theta_k\}$ *are independent*

identically distributed random vectors and each tree casts a unit vote for the most popular class at input $\boldsymbol{x}$. [3]
The specific classifier from Weka is weka.classifiers.trees.RandomForest and are tested with these different options:

1. default options
2. number of trees
3. number of features to consider

SMO SMO is a Support Vector Machine and stands for *sequential minimal optimization*. It is an algorithm developed by John Platt which is made for training a support vector classifier [25]. SVMs were originally developed by Vapnik in 1982 [32] and it only classified binary problems. The way in which this worked is roughly this:
It generates vectors of the documents in a vector space.
Then it performs class separation, i.e. it's looking for the optimal separating hyperplane between the two classes by maximizing the margin between the classes' closest points. The points that lay on these boundaries are called the support vectors and in the middle of these are the optimal separating hyperplane.
It then reduces the weights of data points on the wrong side of this margin so as to reduce their influence.
If it cannot find a linear separator it projects the data points into a higher-dimensional space via kernel techniques.
Lastly it formulates the task at hand as a quadratic optimization problem to be solved by known techniques.
A figure can be seen in Figure 4 on page 16.

SMO has improved upon this, since the quadratic optimization problem usually needs a lot of storage, the SMO algorithm decomposes the overall problem into a series of the smallest possible QP sub-problems. It then solves the smallest of these at each step analytically, which avoids time-consuming numerical QP optimization. [26]
The specific classifier in Weka is weka.classifiers.functions.SMO and we are testing with some different options:

1. with/without normalization
2. varying the tolerance parameter

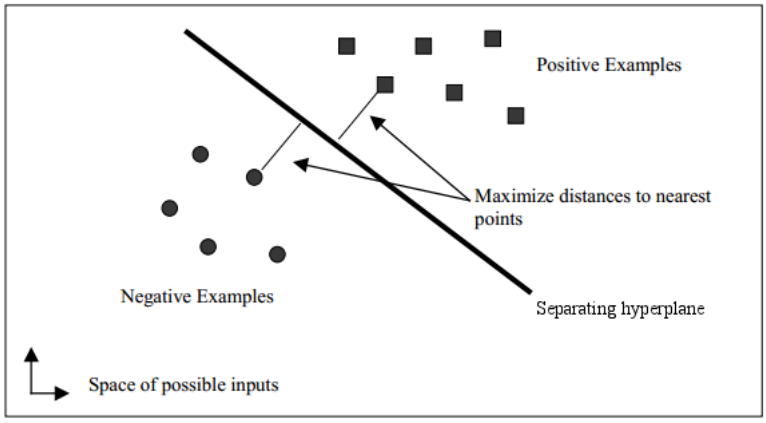


Figure 4: A linear Support Vector Machine

2.3 Our documents

The documents we are classifying in this thesis are web-pages and their syntax etc are well-known and will not be described here. The difference between classifying our documents and the traditional text documents should be noted. Documents used in traditional text classification are usually of a more structured form with consistent styles, as for example a corpora of news articles with well-controlled styles [8]. A web document on the other hand does not have a controlled style or a certain high quality that you expect from a news article. It is of a much more inconsistent form and random in nature.

The diagrams in Figure 5 on page 17 and Figure 6 on page 17 shows the distribution of the document sizes both before preprocessing and after preprocessing, and you can see how they vary a lot in size. It is also notable how large the decrease in size are after the preprocessing are done.

2.4 LOS

The LOS ontology has been mentioned before, and we would like to better describe some details here. We could argue about the definition of LOS as an ontology, because it probably fits better into the description of a thesaurus, but since Difi calls it the LOS-ontology we have also called it that in this paper to avoid confusion. It is a networked collection of controlled vocabulary terms. This mapping is between the three sections main themes, sub-themes and the classes that we are classifying the documents to. This structure can be seen in Figure 8 on page 19. The classes it contains are specific to municipal government services that are available in the public domain and

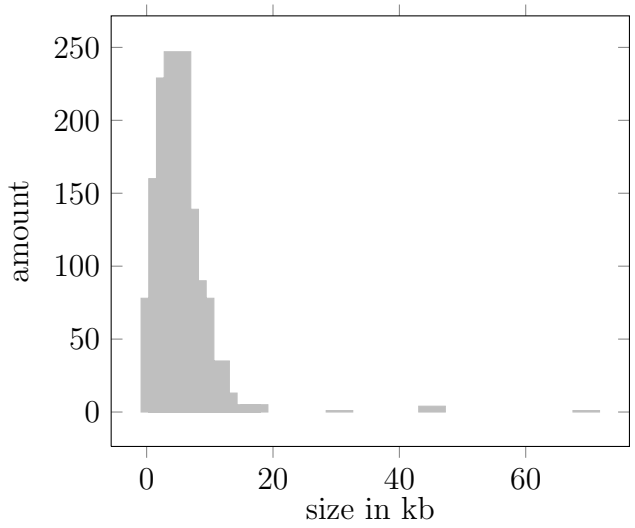


Figure 5: Size distribution of HTML documents

Total size of all the documents before preprocessing are 41.05 MB.

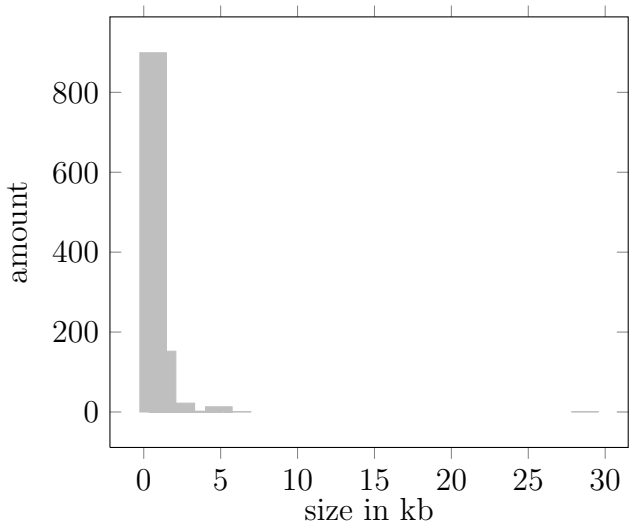


Figure 6: Size distribution of processed documents

Total size of all the documents after preprocessing are 4.32 MB.

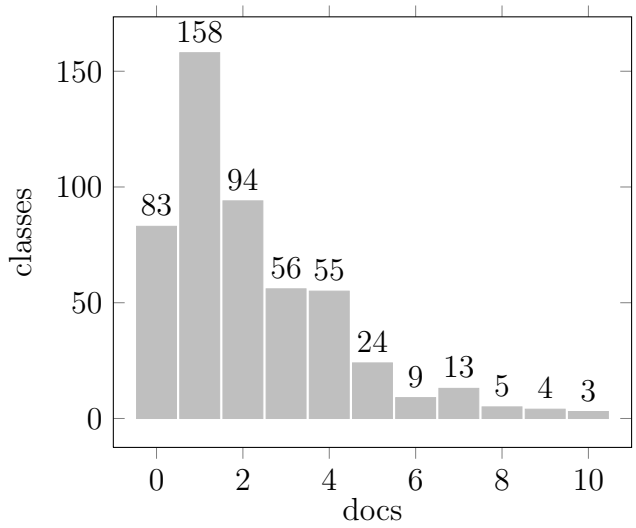


Figure 7: Range of amount of documents in a class

also about rules and regulations that the municipalities enforces. In total it contains 504 classes of which 83 are empty (without any documents) and the rest contains in total 1105 documents. This distribution can be seen in Figure 7.

LOS are created with the goal of making it easier to share information in the public sector. They envision it being used as a standardized menu when you look for services, municipal web-pages can categorize their information about what services they offer and lastly to have an easy way of integrating all such info into the portal norge.no. This way all the relevant information about services that the public sector offers can be available in one central place. Users may not know what search-words to use to find the exact information they want, but with the structure of keywords (categories) that LOS provides it is much easier to navigate until one finds the correct information.

For more information about LOS itself you can visit their web-page[1].

2.4.1 Ontology/thesauri

Los are as mentioned an ontology/thesauri, meaning that it is a collection of knowledge about a certain domain represented as a set of concepts and their relationships. So to be more specific it is a domain ontology. This structural framework is a way of organizing the information about the domain, in this case municipal government services. Since it is a domain ontology, the

[1]http://los.difi.no/

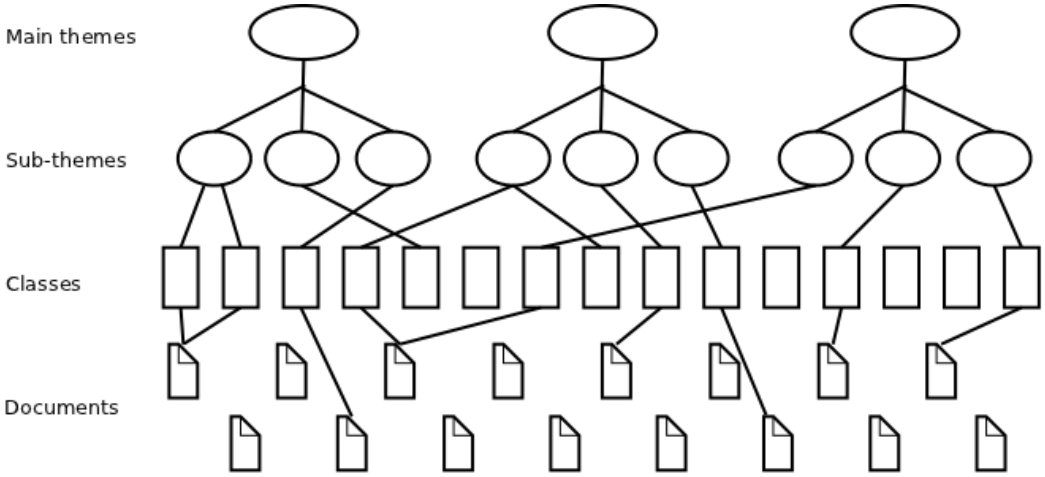


Figure 8: LOS structure

meaning of words that may mean different things depending on the domain should then of course be seen in the context of this ontology's domain. If you were to look at a ontology about "poker" then the word "card" would mean a playing card, as opposed to a ontology about computer hardware were the meaning could be video card. Also, even ontologies on the same domain may have different meanings about certain words depending on what kind of perception of the domain the designers of the ontologies have. The difference in perception may arise from differences in education, language, ideology etc.

At the top there are main themes that encompass large portions of the whole, these are themes like "work", "health", "taxes" and "school", and these contain sub-themes. The sub-themes become a little more explicit and more detailed in what they describe. So for example under the main theme "health" theres sub-themes like "public health", "health care", "mental health" and "patients' rights". These again contain the classes themselves that are a specific area of information. Examples of classes within "health care" are "physiotherapy", "chiropractic", "emergency room" and "hospitals". Each level thus has a relationship to either a higher less specific section or a lower more specific section or both.

3 Technologies

In this chapter we will have a look at some of the scientific work and results that have been done in the field of supervised learning and classification. We will also take a look at some of the different technologies that have been developed as a result of this work. At last we will have some details about the technologies that we have chosen to use when testing the different methods and algorithms that we have discussed in chapter 2. We will take a look at some of the trends and challenges that have been focused upon in research papers and also give a short overview of some systems that are made to automatically categorize web pages with the help of supervised learning.

3.1 Related work

3.1.1 Research about supervised learning and classification

There has for a long time been a high focus on trying to properly and easily extract patterns and relations from the World Wide Web in the research-community. There has been an abundance of different methods proposed and tested with the goal of achieving this. When classifying content on the web it can help in improving crawling, assist in developing large web directories and to analyze content and structure of websites. In the beginning of research about classification the focus has been on the use of already present information in already classified documents to be able to learn what documents belong to certain classes. This is called supervised learning and has proven to work very well with large corpora of labeled documents.

In the later years (last decade) there has been a bigger focus on also using the unclassified documents as a resource directly into the learning process. By using the unlabeled documents into the learning process the accuracy of the results will increase [30], especially when the amount of already labeled documents are small. This methodology has been called semi-supervised learning.

Supervised learning [16] Within supervised learning there are a lot of different learning algorithms such as Support Vector Machines, neural nets, logistic regression, Bayes and naive Bayes, memory-based learning, random forests, decision trees, bagged trees etc, and they have all proved themselves through the years on different tasks. There have been a good amount of progress over the last decade and some of the newer methods give performance that some of the older algorithms from 15 years ago would struggle to obtain [6]. Also, a good algorithm will most probably not perform well on all

Research paper	Classes	Documents per class (avg.)
[18, 13]	135	108
[27]	90	106
	33	65
[8]	20	2000
[9]	10	188
[17]	2	7352
[6]	2	2500
[2]	varying from 2 to 26	varying from 50 to 5500
[28]	varying from 10 to 135	varying from 105 to 960

Table 1: Amount of classes and documents used in research

tasks it is set to solve. There may be great variances in the performance of algorithms on different sets of problems. Thus, when trying to solve a task, several different methods should be tested to get a good result.

Semi-supervised learning [7] Because supervised learning work well on large corpora of labeled documents and this is usually not the case in practical learning scenarios, semi-supervised learning have gotten high focus the last decade. In most practical learning scenarios you have only a small amount of labeled documents and a huge amount of unlabeled documents. Semi-supervised learning tries to take advantage of this scenario by using the unlabeled documents as a resource in the learning process. One of the biggest problems in supervised learning are that there is a sparsity of data available in the labeled documents. With semi-supervised learning you reduce this and improve the accuracy.

A way to get extra information from the unlabeled documents are to use clustering of words in the documents for an improved accuracy [30]. These clusters will then contain several documents that have the same word representations and can then be combined into the prediction from the supervised part of the learning.

While the unlabeled documents can help to improve the accuracy of the supervised learning, there is still some difficulties when the amount of labeled documents are very low [36]. Thus they propose to cluster both the labeled and unlabeled documents with guidance from the labeled data, then to label some of the unlabeled documents based on these clusters and at last build new classifiers by supervised learning based on these expanded datasets. An example can be seen in Figure 9 on page 23. It is rather similar to unsupervised learning, but in this case the labeled data are used to aid in the

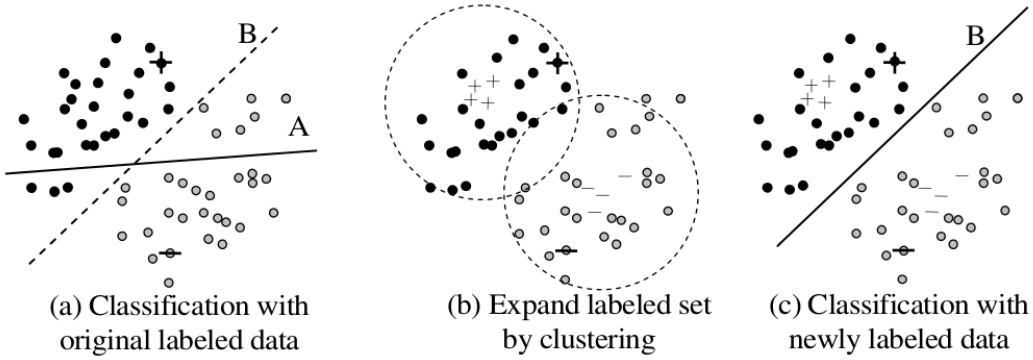


Figure 9: CBC Clustering
The black and gray points are the two actual classes in the unlabeled data. The big + and - are the originally labeled data, and the small + and - are documents that are considered labeled with high confidence by the clustering.

clustering so that latent class labels are easier to spot, and some of the unlabeled documents can be considered as labeled to the clusters they come into with high confidence.

3.1.2 Automatic classification systems

There are a lot of different types of automatic classification systems, but we have focused on the ones that employ supervised learning because this fits the thesis problem. Some systems don't use learning but instead use pure string matching and searches for the ontology term itself, or use part-of-speech tagging or other methods.

MnM [33] is a tool for annotating web resources with semantic information. The tool is accessible through a web browser and provides options for using your own ontologies through it's API and also tools for information extraction from the web-pages. The five main activities that MnM facilitates are:

- *Browse.*
  Here you can choose the specific set of the ontology that you want to use.

- *Markup.*
  If you have no documents that already are marked with categories, you can mark documents here.

- *Learn.*
  This part uses learning algorithms to learn from the marked up documents to create a learning model.

- *Test.*
  In the test activity you can test the created learning model with a testing corpus to see if it scores well enough in recall and precision.

- *Extract.*
  The last part extracts information from new and untagged documents to find the category it belongs to.

These are the main parts of the process that Difi would like to perform.

Melita [10] is a tool that is made to interact with the process of using machine learning to create a model from already categorized documents. The system that learns the documents are called *Amilcare*. It has an interface where you can see the proposed categories for a document and also highlighting of the words or parts of the document that triggered it. In the early stages of categorization, when there are not enough documents to make the system very sure of what category a document belongs to, the user can double-check the proposed categories and select the one that is correct. When the system has got enough documents to learn from it will be able to select categories without the user having to double-check.

Rainbow project [29] is a collection of more or less independent web mining projects that are done by the *knowledge engineering group* at the University of Economics in Prague. Unlike the other systems, the rainbow project are only usable within the domain that these researchers have agreed upon and can thus not be used with your own specialized ontology.

Thresher [12] is a system that focuses more on letting the user teach the system the relations between the web-pages and the categories. The user highlights content in the document that are relevant to a category and the system calculates the *tree edit distance* between the DOM subtrees of these examples to generate a model of the documents. It can then automatically categorize new documents. Thresher thus needs a lot more user interaction than the other systems, and are not able to use a pre-existing ontology or a corpus of already categorized documents as easily as the other ones.

3.2 Technologies considered

When we set out to develop the prototype, we researched some of the options when it came to gathering the web-pages that the learning should be performed on and also at what API's with learning algorithms that were easily

accessible and quick to implement. In this chapter we present some of the most important findings.

3.2.1 WebSphinx

WebSphinx is a Java library for developing web crawlers that are highly customizable [21]. It offers a number of features including a graphical workbench where you can visualize websites and their links amongst other things. The part we used were the Java class library for writing web crawlers. Some of it's features are multi threading, object model for representing pages and links, HTML parsing, pattern matching and HTML transformations.

We used it in the beginning with the collection of links extracted from the LOS sql database we got from Difi, but found out it was easier to scrape los.difi.no myself and use jsoup (section 3.3.2 on page 27) to save the web page. This way we also got a lot more web pages than using the sql database Difi first gave us. This resulted in us going from 450 documents to 1105, which is a significant improvement. The code for this can be found in Code listing 8 in the appendix, Section A.

Although we did not need this library when we were collecting the documents we needed, this library will fit nicely if Difi were to write a system that should crawl municipal websites. There are also a lot of other open source crawlers written in Java, such as Heritrix, JSpider, Bixo, Crawler4j etc. More can be found at [31].

3.2.2 Java Machine Learning Library (Java-ML)

Java-ML is collection of machine learning algorithms written in Java. It is meant to be used and implemented by programmers and research scientists in their code and as such it has no GUI that enables any visual experimentation with the algorithms. An overview of it's main algorithms and features can be seen in Table 2 on page 26.

The library are built around two key elements, the *dataset* and the *instance*, and these are represented by two core interfaces. These again have different implementations for different kinds of datasets and the algorithms you use. A quick look at using the library in Java code are presented in Code listing 1. This example takes in a dataset called iris.data in line 1, then defines a clustering algorithm to be used in line 2 and at last uses this algorithm to get the clusters the data would create from the data in line 3.

Clustering	Classification	Feature selection
K-means-like (7) Self organizing maps Density based clustering (3) Markov chain clustering Cobweb Cluster evaluation measures (15)	SVM (2) Instance based learning (4) Tree based methods (2) Random Forests Bagging	Entropy based methods (4) Stepwise addition/removal (2) SVM_RF Random forest Ensemble feature selection
Data filters	**Distance measures**	**Utilities**
Discretization Normalization (2) Missing values (3) Instance manipulation (11)	Similarity measures (6) Distance metrics (11) Correlation measures (2)	Cross-validation/evaluation Data loading (ARFF and CSV) Weka bridges (2)

Table 2: Java-ML main algorithms and features

Code listing 1: Java-ML code example

```
1 Dataset data = FileHandler.loadDataset(new
    File("iris.data"), 4, ",");
2 Clusterer km = new KMeans();
3 Dataset[] clusters = km.cluster(data);
```

We considered Java-ML as a proper contender for what library to use for the machine learning techniques, but Weka had some properties that gave it an edge and this will be elaborated on in the next chapter.

For a more in-depth look at Java-ML, all the API documentation can be found at their web-page[2].

3.3 Technologies in our prototype

Here we take a look at some of the main technologies that we use when testing the machine learning methods and also software that we envision used in a system developed for this purpose.

[2]http://java-ml.sourceforge.net/api/

3.3.1 WEKA

Weka is similar to the Java-ML library mentioned before as it is a collection of machine learning algorithms used for data mining and it is written in Java. A difference is that it also comes with a graphical user interface that can be used for testing data preprocessing, classification, etc., together with the usual API you can use while writing Java applications.

Weka was developed by a machine learning group at *The University of Waikato* in New Zealand and their goal is to "build state-of-the-art software for developing machine learning techniques and to apply them to real-world data mining problems". Out from this goal sprung the Weka software, a software now used both by specialists within a particular field, researchers and scientists, and also within teaching [11]. An example of the impact Weka have done in the field of data mining and machine learning is the *SIGKDD Service Award* it was awarded from *ACM's Special Interest Group on Knowledge Discovery and Data Mining* in 2005 [24]. This award are given to individuals or teams for outstanding service contributions to the field of knowledge discovery in data and data mining.

The reason for choosing Weka is that I could use the GUI to test some things beforehand, that it had been given acclaim as mentioned above and also that it is used in several papers. Also, I have been reading the book *Data Mining: Practical Machine Learning Tools and Techniques* [34] and it are showing that it is powerful and could be used for exactly what I wanted to do.

3.3.2 jsoup

jsoup is a Java library for working with HTML documents and it provides an API for extraction and manipulation of the data within the document. It uses a documents DOM in combination with jQuery-like methods to achieve this. Some of it's main usage applications are:

- scrape and parse a HTML document from URL, local file or a string
- find and extract data from HTML documents by traversing the DOM
- manipulate the HTML elements, attributes and text

If we were to try to parse the HTML by developing our own methods it would be like inventing the wheel all over again, and we would waste a lot of time probably ending up with a sub-par solution compared to jsoup. It is therefore a good choice to use such a library for this kind of task.

My usage of jsoup can be seen in Code listing 8 in Appendix A.

4 Prototyping

This chapter will give an overview of the overall design that we have envisioned to be used at Difi. We will also describe how the parts we have tested can be used and incorporated into the system.

4.1 Problems and decisions

One of the main problems we have encountered along the way have been the rather small amount of documents per class in the ontology. At this moment there are only 1105 documents distributed across 504 classes, and to make matters worse 83 of these classes are even empty. Because some classes are empty a supervised learning system will not be able to learn these classes and the resulting classification will never be able to classify any new classes into these either. The tests (which can be read about in chapter 5 on page 37) also support this.

4.1.1 Memory and time restrictions

The way the algorithms are built and the fact that the system are going to be processing large amounts of data when building a classifier the system that should run this would need a good amount of RAM. Depending on the amount of words that are set to be kept in the filter, the algorithm used and the amount of documents to process, this will vary. Some examples of RAM-usage with our amount of docs are shown in Table 3.

As you can see, the amount of RAM needed are quite large and would only increase when more documents are added to the system. Therefore any computer that works with this would probably benefit greatly from having at least 8Gb of RAM and the bare minimum would be 4Gb as long as it weren't working on anything else at the same time.

Also, the time each algorithm uses to build the classifier from the data and also the time it uses when classifying documents varies quite a bit and should be taken into account when choosing which one to use in a system. Time

Algorithm	RAM usage
NaiveBayes	3174 MB
IBk	494 MB
RandomForest	1965 MB
SMO	2940 MB

Table 3: RAM usage with default settings

Algorithm	Build classifier	Classify per document (avg)
NaiveBayes	64 sec	6.1 sec
IBk	9 sec	0.017 sec
RandomForest	4274 sec (71 min, 14 sec)	0.004 sec
SMO	410 sec	0.846 sec

Table 4: Time usage building classifier

usage may be critical if the system should rebuild the classifier each time a new document are added or if it should just do the rebuilding every night when it has more time to do so. The time used to classify new documents are not too great and should not present a big problem as long as it's not in the amounts of tens of thousands at a time.

4.2 Overview of system

The system would be working as autonomously as possible, guided by the user for some operations. For example; if the system classifies a document that gets a very low score, it would be marked as "need human evaluation" and the user could then see the different categories that the system predicted and their scores. Then the user could decide which one of them that are the correct one. This way, if the collection has categories with a low coverage (small amount of docs) the system would predict lower and more similar scores for several categories and the user could then inspect and decide the correct one.

Some of it's main operations:

- **Monitoring the LOS-ontology.**
  It should be monitoring the LOS-ontology for new additions that may be added manually by users. When such an addition is found it should add this new document to the learning model and recalculate the model accordingly when needed.

- **Have an input-option for single documents that users want to have classified.**
  This way the user can give the system a single document in the form of a URL, and have it added to the system for classification. When it is done, it should be added into the ontology in its appropriate class.

- **Have an input-option for a whole domain.**
  When Difi wants to add a whole new domain or municipality's domain

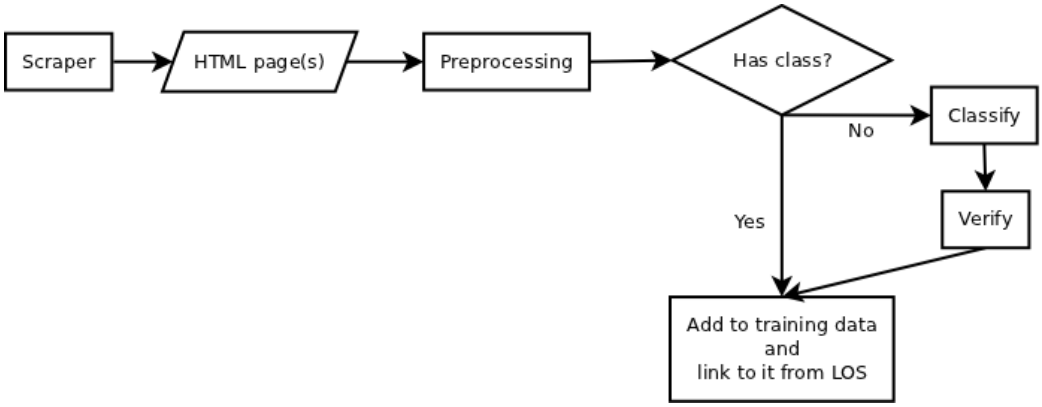


Figure 10: Overview of the system workflow

to the ontology, they could just give the domain-URL to the system and have it download, preprocess and classify all it's pages and then add them to the ontology.

A graphical overview of the workflow of the system can be seen in Figure 10.

4.3 System parts we have implemented for testing

This section will give an overview of the parts of this system that we have implemented and tested. Since this was only supposed to be a prototype, we have not created any GUI or made it very user-friendly. The file that selects what kind of algorithm to use and what options to use are "ClassifyController.java" found in the enclosed zip archive. The code only outputs results to the standard output stream.

4.3.1 Preprocessing

Preprocessing will have to be performed on all documents that are gathered from the web. This is both to ensure that the feature selection will have appropriate words to use and that file-sizes will be kept at a minimum. There are as mentioned in section 2.3 a lot of extra information in a HTML-document that are needed for the browser, but are superfluous for us in the classification-task. The preprocessing-part of the system are where this extra information are removed.

The preprocessing are done to remove certain parts that are either not relevant or that are not giving any positive effect on the results. Parts that are removed are:

- **HTML tags**
  The HTML tags are not relevant for the classification and may therefore be removed.

- **Stop words / common words**
  Stop words are words that are very common and considered to not give any positive effect on the results, and can thus be removed. A list of the stop words used can be found in the file structure provided which are described in Appendix B on page 65.

- **Punctuation symbols**
  Punctuations symbols like "?", "!" etc, are not relevant for the learning process and may be removed.

You can see the decrease in file size after preprocessing are done in the diagrams in Figure 5 on page 17 and Figure 6 on page 17. The total size of all 1105 documents before preprocessing are 41.05 MB and after the preprocessing it has decreased to 4.32 MB. The preprocessed size is only 10.5% of the original size and this is a great reduction in size. Since the system has to keep the documents locally it is imperative to keep the file size as low as possible and the preprocessing helps with this.

The code for this can be found in the provided zip described in Appendix B on page 65.

4.3.2 Feature selection

The feature selection are handled by a Weka filter called StringToWordVector which converts strings into a set of attributes that represents word occurrence in a document (the string provided from the preprocessing). The weight w_j of a word t_j in a document d_i is in the simplest way calculated as:

$$w_j = \begin{cases} 1, & if\, the\, word\, is\, in\, the\, web\, page \\ 0, & otherwise \end{cases}$$

This gives equal importance to words that occur a lot in many documents and to words that are more rare and probably more important. Thus we use some of the StringToWord built-in options of the StringToWordVector filter to achieve TF/IDF (term frequency–inverse document frequency) weights for each word. TF/IDF is a product of the local term importance (TF) and the global term importance (IDF):

$$w_{ij} = TF(t_j, d_i) \cdot IDF(t_j)$$

TF(t_j, d_i) is the number of time the word t_j occurs in the document d_i, and the document frequency DF(t_j) shown next is the number of documents that the word $t_j$occurs in at least once.

$$DF(t_j) = \sum_{i=1}^{N} \begin{cases} 1, & if\, d_i\, contains\, t_j \\ 0, & otherwise \end{cases}$$

The IDF(t_j) is then calculated from this and the total number of documents N:

$$IDF(t_j) = log(\frac{N}{DF(t_j)})$$

This reflects how important a word is to a document in a collection of documents as a whole. Meaning that if the word occurs in many documents, then it will be given a lower weight, but if it only occurs in a single document it will be given a larger weight. We also normalize the word frequencies for a document according to the document lengths. This way a long document that contains a lot of one word should not get precedence just because it is a long document with frequent use of this word because of it's length.

This filter uses what's called a *bag of words* representation of a document. It takes single words found in the collection of documents as features and ignore the sequence of the words in a document and only focuses on the statistics of the isolated word. There are some other ways one could choose to get feature representations about the words. One could use the word positions in the document, use n-grams representation where you look at word sequences of length n, use phrases, terms or hypernyms etc. All of these methods have been used in lots of different ways on different problems, but there have not been shown any significant differences in performance or any big advantages when using them in text categorization [27, 22].

4.3.3 Classification as a whole

Here we will describe the whole process of learning a model of the documents in a corpus, and then classifying new documents with this model.

The code for this and section 4.3.2 can be seen in code listing 9 on page 57. The testing system creates an instance of the class *WebPageClassifier* which contains a set of Instances called *m_data* which are going to contain the training data. It has a boolean called *m_UpToDate* that ensures that every time new data are added to the training data in *m_data*, when it is going to classify a document, it has to perform the feature selection by the filter

and also rebuild the classifier. It also contains the filter *m_Filter* and the classifier in *m_Classifier*.

When an instance of this is created, it takes in a specified classifier and it's settings and uses these. Then it defines a *FastVector* with 2 attributes, the document content (text) and the class it belongs to. The class attribute has default-values which are all the classes that LOS contains. After this it defines the name of the dataset, its attributes (the *FastVector*) and a initial capacity.

The next step is to update the model with the data from all the documents and this is done by going through all the preprocessed files in the file structure and for each file run the method *updateModel* with the document content and class. This method then runs the method *makeInstance* on the content of the file. *makeInstance* defines the type of attribute that the content should be and the value of it (the content of the file), and what dataset it should belong to (*m_data*). Then it returns the instance so that *updateModel* can set it's class value and add the instance to the dataset *m_data*. At last it sets the boolean *m_UpToDate* to false so that the system knows that it has to perform the filter and rebuild of the classifier the next time it runs a classification on a document.

Now when the system has been filled with the documents that are to be learned the system may use the method called *buildClassifier* to build the classifier (learn the documents and classes). This method first ensures that the filter has it's options set as described in section 4.3.2 by running the method *setFilterOptions*. Then it filters the data with the filter (performs feature selection) and lastly builds the classifier with the filtered instances. Now the system has learned the documents and classes so that it are able to predict classes for new documents.

The last part is then to classify new documents. The method *classifyContent* takes in the content of a document and in our case also the actual class it belongs to (for test purposes). If the system has added new documents into the collection it will run the *buildClassifier* method to rebuild the classifier as described above. Then it creates a copy of the structure if the data has string or relational attributes and "cleanes" string types (i.e. doesn't contain references to the strings seen in the past) and all relational attributes. The content of the document to be predicted are then added to the cleaned test-set and the test-set is then filtered by the filter. When this is done the last step is to use the *classifyInstance* method of the classifier with the filtered test-set as input and get the indexes of the predicted classes.

The output would be the predicted percentages of the different classes it could be a part of. Some classes may get almost the same percentages and this could mean that the document could belong to all of them. In these

cases the user should verify the results manually.

5 Testing and results

5.1 The testing

As mentioned we will test each algorithm mentioned in section 2.2.2 with different options to see if we can find the optimum settings for the best results. All the algorithms are tested with their default settings as a baseline to compare the other settings against.

The tests consist of two parts, the first is to test the classifier against a test set described in section 5.2, and to perform a cross-validation on the learned data set. The cross-validation was performed with 5 runs with 5 folds each and with different seeds for the randomizing of the data. Each run of the validation creates a copy of the test data that it then randomizes the features of each document instance. Then it creates an evaluation instance and runs 5 folds on this randomized data. Each fold then creates a training set and a test set from the randomized data. Then it builds the classifier on the training set and lastly runs an evaluation of the classifier on the test set. For each run it then shows it's results and we can then average over these 5 runs for our results. The code for this can be seen in the function *performEval()* in code listing 9 on page 57.

We also tried running a cluster-based classification from Weka, but it ran for 140 hours without returning any results. The reason for it using way too much time is because of the 421 clusters it was supposed to make, and with each new document added it would recalculate a lot of the clusters.

Also, we have mentioned some methods of semi-supervised learning, but have not tried any of them out for ourself. This is because we have not had access to any more documents that are already categorized in LOS to work with and the relative small amount of 169 documents we have as a test-set would not really be enough. The "unclassified" documents would need to be categorized for us to be able to check if the classification improved (the categories would only be used to check for improvement, and the documents would be treated as if they were unclassified), and we would not have enough time to sit and manually classify thousands of documents ourself. After all, the reason for testing this is to be able to automate that process.

5.1.1 Test options

NaiveBayes We tried adding the kernel estimation option, but it used too much ram and we could not get any results on whether this could improve the results or not. We also tried it with the supervised discretization option to process the numeric attributes. This turned out to be getting very bad

results, it classified wrong on all test documents and showed comparably bad results in the cross-validation.

IBk First we tried to vary the amount of neighbors used by the algorithm manually and saw no improvement when choosing more neighbors, it actually got worse. Then we choose the option to let the algorithm choose itself within an interval we set. It turned out that the algorithm found that just 1 neighbor gave the best results, which we also could see from testing each on our own.

RandomForest We started out by varying the amount of trees it builds from the standard of 10, and then increased it twofold each time. We also increased the amount of features it should consider.

SMO We tried it with and without normalization (in case there are a great variance in some of the variables) and also to vary the tolerance parameter from the default of 0.001 and up to 1.0 with the steps 0.001, 0.005, 0.01, 0.05, 0.1, 0.5 and 1.0.

5.1.2 The test system

The tests have been run on a computer with a Intel i7 CPU running normally at 2.2GHz but it automatically overclocks upwards to 3.3GHz if it's not too hot. It has usually been working in the range of 2.4 to 2.8 GHz during the testing and it has only used one core. We have also given the program as much ram as we have to give (4Gb) to use. This has been more than enough for most of the tasks, although some of the algorithm options have seemed to use so much ram, so fast, that we could not complete them in testing. Also, some of the algorithms uses a lot of time to complete, and the validation runs the algorithms 5*5 times, this results in a lot of time. The algorithms have thus not gotten all their options tested. This is not such a big problem, since we still show how bad they perform because of the low amount of training material, and we were not out to try to find the perfect options.

5.2 The test set

We created the test set by selecting at random one document from every class that has 3 or more documents in it. This resulted in a test set containing 169 documents from a good portion of the data set (18%). We could not take out much more documents since the amount of documents we were learning from were rather small already. If we remove too many documents the quality of

the learning would suffer, and as we can see in the results, this was probably a good call.

5.3 The results

In the testing there are two parts, the real-world scenario of classifying our test-set and also the results from the WEKA evaluation. From both of these we will look at the percentage of correctly classified documents, but in the evaluation there are also one other measure that are of high interest, the kappa statistic. The kappa statistic are a measure that tells you how much better the result would be than if one would just randomly guess what class a document belongs to. This number is computed as:

$\frac{P(A)-P(E)}{1-P(E)}$ [5]

P(A) is the result from the classification and P(E) is the result that was expected from the classification. I.E. P(E) represents the amount of times we would expect the system to agree to the result by chance. The resulting number can range from -1 to 1 where a value of 1 would mean perfect agreement, a 0 would mean it is equal to chance and a -1 would mean a perfect disagreement. In our case, the closer the kappa statistic are to 1, the better the calculated model. If the result would be close to 0 it would mean that the calculated model would be bad and equal to someone just picking categories randomly.

5.3.1 Main results

NaiveBayes The best results came from using the default options, with neither kernel estimation or supervised discretization. Out of the 169 documents in the test set it classified 157 with the wrong class, giving it only 12 correct classifications. That equals to only 7.1% correct and is not a very good result. On the cross-validation it was a little lower with an average of 5.4% correctly classified instances. We can also see that the kappa statistic is very close to zero, which means that it is close to random guessing. The full overview of one of the evaluation runs can be seen here:

Code listing 2: Validation result for Naive Bayes with default options

```
=== Setup run 3 ===
Classifier: weka.classifiers.bayes.NaiveBayes
Dataset: WebpageClassification
Folds: 5
Seed: 3

=== 5-fold Cross-validation run 3===
Correctly Classified Instances          50               5.3419 %
Incorrectly Classified Instances       886              94.6581 %
Kappa statistic                          0.05
Mean absolute error                      0.0037
Root mean squared error                  0.0595
Relative absolute error                 94.6926 %
Root relative squared error            133.7309 %
Total Number of Instances              936
```

IBk The best results came once again from the default options where it only looked at 1 neighbor. This was also shown when we used the option to let the algorithm choose the number of neighbors itself by doing some evaluation of the training data. Out of the 169 documents in the test set it classified 160 with the wrong class giving it only 9 correct classifications. This gives it a little bit of a worse performance than the NaiveBayes, 5.3% in this case. On the cross-validation it also got a slightly lower result with an average of 5.2% correct. As with NaiveBayes, this also got a very low kappa statistic score close to zero. The full overview of one of the evaluation runs can be seen here:

Code listing 3: Validation result for IBk with default options

```
=== Setup run 2 ===
Classifier: weka.classifiers.lazy.IBk -K 1 -W 0 -A
    "weka.core.neighboursearch.LinearNNSearch -A
    \"weka.core.EuclideanDistance -R first-last\""
Dataset: WebpageClassification
Folds: 5
Seed: 2

=== 5-fold Cross-validation run 2===
Correctly Classified Instances          49               5.235  %
Incorrectly Classified Instances       887              94.765  %
Kappa statistic                          0.0491
Mean absolute error                      0.0038
Root mean squared error                  0.0484
Relative absolute error                 96.9611 %
Root relative squared error            108.7455 %
Total Number of Instances              936
```

RandomForest With the default options of the RandomForest algorithm we got similar results as the previous ones. It classified 157 out of the 169 wrong, giving it 12 correctly classified, or 7.1%. The validation results came in at something equal to the others with an average of 6.5% correct.

Then we started to vary the amount of trees it builds and the number of features it considers. From all the testing it turned out that setting the amount of features to 1 gave the best results in every variance of the amount of trees we chose. We then increased the amount of trees twofold each time. 20 trees gave 4 more correctly classified, 40 trees gave 7 more and 80 trees or more gave a whole 13 more. But beyond the amount of 80 trees the results did not improve any more. The result is that we had an impressive 25 correctly classified instances from the test-set. This amounts to 14.8% and is quite much higher than any of the other algorithms.

The validation had a slightly lower result with an average of 8.3% correctly classified and the kappa statistic has improved slightly, but are still not very good.

One of the validation results from the best options with 80 trees and 1 features considered:

Code listing 4: Validation result for RandomForest with 80 trees and 1 feature

```
=== Setup run 4 ===
Classifier: weka.classifiers.trees.RandomForest -I 80 -K 1 -S 1
Dataset: WebpageClassification
Folds: 5
Seed: 4

=== 5-fold Cross-validation run 4===
Correctly Classified Instances          84               8.9744 %
Incorrectly Classified Instances       852              91.0256 %
Kappa statistic                          0.0861
Mean absolute error                      0.0038
Root mean squared error                  0.049
Relative absolute error                 95.7871 %
Root relative squared error            110.1826 %
Total Number of Instances              936
```

SMO The SMO algorithm impressively classified a whole 17 (10.1%) documents correctly with it's default options. In the validation it got a decent 7.4% correct which is better than most of the other algorithms even after they have been tweaked by using different options. But once again the kappa statistic are low, almost on the same level as NaiveBayes and IBk. When we tried it with normalization and to vary the tolerance parameter, it got slightly worse (2 more wrongly classified) up to 0.1 where it slightly improved

(only 1 more wrongly classified). But then on 0.5 it got worse again with 4 more wrongly classified and on 1.0 it got considerably worse with every document classified wrong. Thus, the default options turned out to get the best results. One of the validation results follows:

Code listing 5: Validation result for SMO with default options

```
=== Setup run 3 ===
Classifier: weka.classifiers.functions.SMO -C 1.0 -L 0.001 -P
    1.0E-12 -N 0 -V -1 -W 1 -K
    "weka.classifiers.functions.supportVector.PolyKernel -C 250007
    -E 1.0"
Dataset: WebpageClassification
Folds: 5
Seed: 3

=== 5-fold Cross-validation run 3===
Correctly Classified Instances          63               6.7308 %
Incorrectly Classified Instances       873              93.2692 %
Kappa statistic                          0.0631
Mean absolute error                      0.004
Root mean squared error                  0.0445
Relative absolute error                100.0224 %
Root relative squared error            100.0365 %
Total Number of Instances              936
```

Overview One can see in the figure 11 that all of the algorithms performed fairly similarly. Especially in the evaluation results, all of the algorithms were within 3.1% of each other. On the real world test with the test set on the other hand, the variance were a bit bigger. With a difference of 9.5% from the worst performing algorithm IBk to the best performing RandomForest, we can see that there is a difference in how the real world data works with different algorithms. So with the documents that are in the ontology, the RandomForest algorithm are clearly the one to choose.

5.3.2 Extended testing

Seeing both from the literature (Table 1 on page 22) and from our tests, the amount of documents per class in the system does not result in any good classification. There are simply to few documents and too many classes. Because of this we wanted to see if we could get some better results if we used the two other levels (main themes and sub-themes) of the LOS-ontology's class structure (Figure 8 on page 19) when classifying.

If we would use the sub-theme level we would have a lot fewer classes (79 versus the 421 that contains documents, 504 in total), and this would be a reduction of as much as 81.24%. At the main theme level, the decrease in classes would be substantial, going from 421 to only 15 classes. This

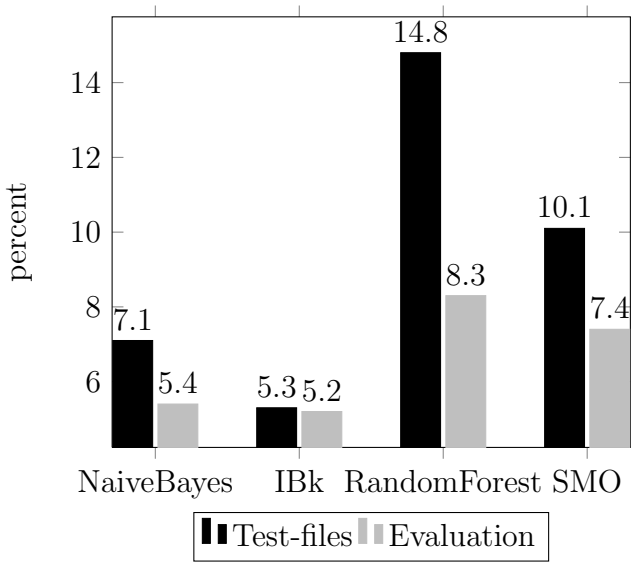


Figure 11: Distribution correctly classified in testing

equates to a reduction of as much as 96.44%, and it is a huge reduction. And when reducing the amount of classes we of course increase the amount of documents per class and thus the probability for a correct classification. When reducing to 79 classes the amount of documents per class increases from an average of 2.63 per class to an average of 13.99 documents per class. This is a decent improvement, but nowhere near the amount used in testing by most literature. It is only when we reduce the number of classes to only the 15 main theme classes that we see a number of average documents per class that are somewhere near what is used in the literature. This results in an average of 73.66 documents per class and are within the numbers we have seen. A comparison of these numbers can be seen in Figure 12 on page 45.

We wanted to see what kind of improvements we could get from the reductions in classes by using the algorithm that performed best in our earlier tests, the RandomForest algorithm, with the settings that performed the best. We ran the learning on the documents with the new levels and performed the cross-validation we used in the previous tests to get the results.

When we reduced the number of classes to 79 we achieved an average of 34.23% correctly classified, which is a good step up from the 8.3% we saw when using all 421 classes earlier. Here you can see one of the validation results:

Code listing 6: RandomForest Validation results with 79 Classes

```
=== Setup run 2 ===
Classifier: weka.classifiers.trees.RandomForest -I 80 -K 1 -S 1
Dataset: WebpageClassification
Folds: 5
Seed: 2

=== 5-fold Cross-validation run 2===
Correctly Classified Instances         321               34.2949 %
Incorrectly Classified Instances       615               65.7051 %
Kappa statistic                          0.329
Mean absolute error                      0.0195
Root mean squared error                  0.1093
Relative absolute error                 78.422  %
Root relative squared error             98.0375 %
Total Number of Instances              936
```

Lastly we ran the test with only the 15 main themes as classes and achieved an average of 51.60% correctly classified. The last validation result:

Code listing 7: RandomForest Validation results with 15 Classes

```
=== Setup run 5 ===
Classifier: weka.classifiers.trees.RandomForest -I 80 -K 1 -S 1
Dataset: WebpageClassification
Folds: 5
Seed: 5

=== 5-fold Cross-validation run 5===
Correctly Classified Instances         484               51.7094 %
Incorrectly Classified Instances       452               48.2906 %
Kappa statistic                          0.478
Mean absolute error                      0.0821
Root mean squared error                  0.2196
Relative absolute error                 66.4749 %
Root relative squared error             88.361  %
Total Number of Instances              936
```

5.3.3 Discussion

These results are not good enough to be able to use this kind of system on the LOS-ontology as it is as of now.

One of the reasons for the bad result is the low degree of separation between the categories in LOS. Firstly, the collection has a rather low amount of documents to be able to learn the differences between categories and the categories used are rather fuzzy and ambiguous. Many classes overlap and are not very well-defined or separated. Because of this the algorithms computes that a document has equal similarity with several classes. Although this

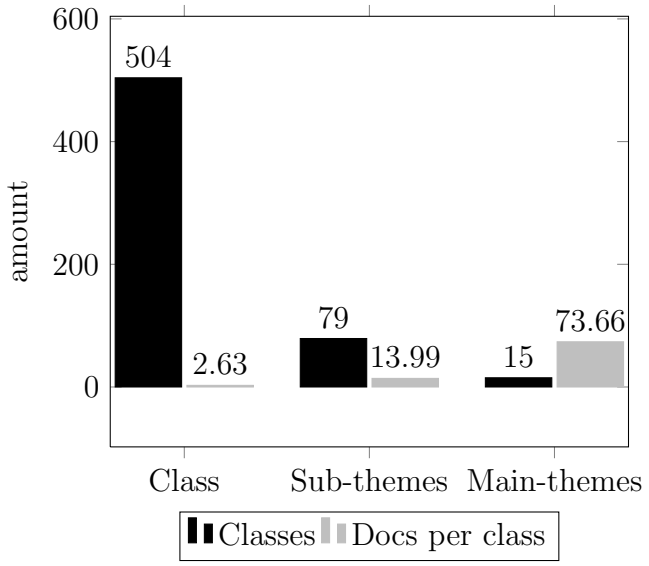


Figure 12: Reduction in classes

may not really be wrong either, because the document may very well contain information relevant for several classes, the cut-off value for how high of a percent a document needs to have in common with a class to be considered belonging to this class may be hard to define.

When we took a look at the results from the test-data we could see that this problem of ambiguity shows. The classes that get a lot of correct classification are either classes with a lot of documents or they are quite unique in the words they use. Although only 13% of the classes that got correctly classified over 50% of the time had 5 or more documents, of all the classes only 3.6% had 5 or more documents. So, having more documents does improve the chances of getting correctly classified. We could also see that most of the other classes that got correctly classified over 50% of the time had content that was quite unique, so that despite them having fewer documents to learn from, they got good results. Classes like “Viltforvaltning”, “Kystforvaltning, “Alkoholsalg” and “Lotteri” got correctly classified in over 75% of the tests.

The extended testing showed a great improvement because of the increase in documents per class and further solidifies the stance that we need more documents to get good enough results.

6 Conclusion

In this thesis we tried to find out if there were possible to use supervised learning to help in classification of documents into the classes of LOS. We wanted to see what kinds of precision we could get in the classification of new documents and also estimate how many documents that would be needed. In this chapter we will discuss our findings and testing results.

6.1 Our results

As we have seen in the results in Section 5, the percentages of correctly classified documents on each of the classifiers are very low. The worst performing came in at a low 5.3% and the best performing at an albeit higher but still low 14.8%. Also, the kappa statistic, that ranges from 0.0491 to 0.0861, shows that they do not really perform much better than if it tried to just guess the classes at random.

When trying to decrease the number of classes and increasing the number of documents per class by using the higher-level categories of LOS we see a great increase in both percentage correct and in the kappa statistic. With 79 classes the amount of correctly classified documents increase to 34.29% and the kappa statistic increase to 0.329 which is a great increase from the previous results. When we go even further and only use the 15 top level categories of LOS the amount of correctly classified documents increase to 51.71% and the kappa statistic to 0.478. Both of these are good increases and shows that with more documents per category the algorithm can give a much better result.

But when we look at the percentage of increase between the different levels it levels out a bit. The first leap from the original 2.625 documents per class up to 13.987 is a 432.84% increase. This resulted in the increase from 8.3% correctly classified up to 34.29% and this is an increase of 313.13%. Here both the increase in documents per class and the resulting correctly classified are huge and follow each other rather well with only a difference of 32.10%. The last leap from 13.987 documents per class up to 73.666 is an increase of 426.67%, almost the same as with the first one. On the other hand, the increase in correctly classified documents go from 34.29% to 51.71% and is only an increase of 50.8%. Here, the increase in documents are still huge, but the increase in correctly classified are rather low. The difference this time is 157.44%. I.E. the increase in correctly classified documents does not follow the increases in the amount of documents per class.

We would probably see some differences in this behavior if we would actually get more documents labeled as opposed to just decreasing the amount

of categories. We would probably not get such a drastic difference so quickly when increasing the amount of documents per category if we were adding new documents. But this still shows that we at least need more documents for the percentages of correctly classified to reach a usable level. The number of correctly classified documents would have to be in the range of 80% for this to be usable by Difi, and there is a long way to get there. To be able to reach these numbers there are two possible things to do. One can add more documents, which is the number one thing to do to achieve this goal. And secondly, one can try to decrease the amount of categories. The last part are probably not something Difi wants to do since they feel that the different categories are required. But seeing as a lot of categories are overlapping each other quite a lot and that the result of this are that the learning are having a hard time to discern the different categories, something should be done about this too.

From what we have seen in the literature (Table 1 on page 22) and from our testing we would estimate that with such a high amount of categories as LOS has (504), they would need at least 50 documents per class on average to be able to get any good results. This would result in them having to have upwards of 25200 documents in LOS. This is a huge increase of 2180.54% from what they have now, and would take quite a lot of manual labour to achieve. On the other hand, they could use supervised learning to give them a sort of indication of what categories a document may belong to so that the manual job would take less time.

6.2 Future work

To be able to get better results with this low amount of documents per category some methods try to use the documents that are to be classified as an information source as well. In "CBC: Clustering Based Text Classification" [36], they use clustering of both the labeled documents together with the documents that are unlabeled. They then consider the unlabeled documents that are very close to the centroids of classified documents as having such a high certainty of belonging in that class that they then label them and use them in the training portion to make a new model.

"Using Unlabeled Data to Improve Text Classification" [23] also uses the unlabeled documents as a way of improving the results. They initially use for example a Naive Bayes classifier to model the labeled documents only. Then they estimate the labels of the unlabeled documents and use both the newly estimated documents and the labeled documents to build a new classifier. This method showed a 30% reduction in classification error over just learning the labeled documents when there were only 15 documents per

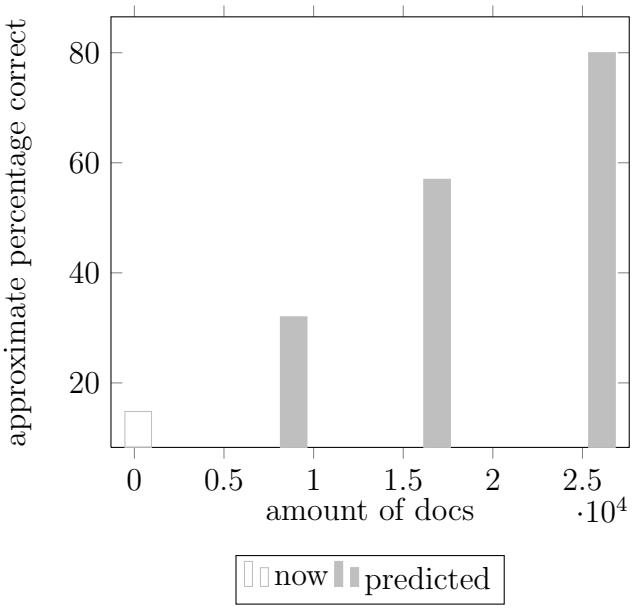


Figure 13: Estimated needed amount of docs

class.

6.3 Summary

As we have seen in the literature, supervised learning are most certainly a viable option when you want to automate categorization. A lot of papers show good results [15, 6] and the sheer amount of papers that are written on the subject show that it is a subject that are thoroughly studied and continually improving.

But as the situation stands today with regards to the LOS ontology and the use of supervised learning, it is not possible to get any good results with the standard methods. The small amount of documents and the large amount of classes in LOS makes it impossible to get a good model that gives good results when classifying new documents. As we saw when we tried to use the higher-level categories of LOS and thus reducing the amount of classes and increasing the amount of documents per category, the percentage of correctly classified documents increased a lot. It did not however increase enough to be able to be used. Thus we envision that as the system stands today, using supervised learning to categorize new documents does not give good enough results to be able to be used totally automated without human intervention. On the other hand, it can be used as an indicator that automatically can

categorize documents that have a very high probability itself, and let the user manually categorize documents where the system are not sure enough. This way it can ease the burden of the manual labour of categorizing a lot of documents.

Bibliography

[1] D. Aha and D. Kibler. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

[2] E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine learning*, 36(1):105–139, 1999.

[3] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[4] M.K. Buckland. What is a "document"? *JASIS*, 48(9):804–809, 1997.

[5] J. Carletta. Assessing agreement on classification tasks: the kappa statistic. *Computational linguistics*, 22(2):249–254, 1996.

[6] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168. ACM, 2006.

[7] O. Chapelle, B. Schölkopf, A. Zien, et al. *Semi-supervised learning*, volume 2. MIT press Cambridge, MA:, 2006.

[8] C. Chekuri, M.H. Goldwasser, P. Raghavan, and E. Upfal. Web search using automatic classification. In *Proceedings of the Sixth International Conference on the World Wide Web*. Citeseer, 1997.

[9] J. Chen, H. Huang, S. Tian, and Y. Qu. Feature selection for text classification with naïve bayes. *Expert Systems with Applications*, 36(3):5432–5435, 2009.

[10] F. Ciravegna, A. Dingli, D. Petrelli, and Y. Wilks. User-system cooperation in document annotation based on information extraction. *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, pages 122–137, 2002.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[12] A. Hogue and D. Karger. Thresher: automating the unwrapping of semantic content from the world wide web. In *Proceedings of the 14th international conference on World Wide Web*, pages 86–95. ACM, 2005.

[13] T. Joachims. A probabilistic analysis of the rocchio algorithm with tfidf for text categorization. Technical report, DTIC Document, 1996.

[14] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, San Mateo, 1995. Morgan Kaufmann.

[15] R.D. King, C. Feng, and A. Sutherland. Statlog: comparison of classification algorithms on large real-world problems. *Applied Artificial Intelligence an International Journal*, 9(3):289–333, 1995.

[16] SB Kotsiantis, ID Zaharakis, and PE Pintelas. Supervised machine learning: A review of classification techniques. *FRONTIERS IN ARTIFICIAL INTELLIGENCE AND APPLICATIONS*, 160:3, 2007.

[17] D.D. Lewis. An evaluation of phrasal and clustered representations on a text categorization task. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 37–50. ACM, 1992.

[18] D.D. Lewis and M. Ringuette. A comparison of two learning algorithms for text categorization. In *Third annual symposium on document analysis and information retrieval*, volume 33, pages 81–93. Citeseer, 1994.

[19] C.D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[20] A. McCallum and K. Nigam. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48, 1998.

[21] Rob Miller. Websphinx: A personal, customizable web crawler. `http://www.cs.cmu.edu/~rcm/websphinx/`, 2012. [Online; accessed 28-febr-2012]".

[22] D. Mladenic. Text-learning and related intelligent agents: a survey. *Intelligent Systems and their Applications, IEEE*, 14(4):44–54, 1999.

[23] K.P. Nigam. *Using unlabeled data to improve text classification*. PhD thesis, Citeseer, 2001.

[24] Gregory Piatetsky-Shapiro. Winner of sigkdd data mining and knowledge discovery service award ... `http://www.kdnuggets.com/news/2005/n13/2i.html`, 2005. [Online; accessed 29-febr-2012]".

[25] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schoelkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998.

[26] J. Platt et al. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[27] S. Scott and S. Matwin. Feature engineering for text classification. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, pages 379–388. Citeseer, 1999.

[28] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[29] V. Svátek, M. Labskỳ, and M. Vacura. Knowledge modelling for deductive web mining. *Engineering Knowledge in the Age of the Semantic Web*, pages 337–353, 2004.

[30] J. Turian, L. Ratinov, and Y. Bengio. Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

[31] `http://java-source.net/`. Open source crawlers in java. `http://java-source.net/open-source/crawlers`, 2012. [Online; accessed 24-may-2012].

[32] VN Vapnik. Estimation of dependences based on empirical data, 1982. *NY: Springer-Verlag*.

[33] M. Vargas-Vera, E. Motta, J. Domingue, M. Lanzoni, A. Stutt, and F. Ciravegna. Mnm: Ontology driven semi-automatic and automatic support for semantic markup. *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, pages 213–221, 2002.

[34] I.H. Witten, E. Frank, and M.A. Hall. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2011.

[35] Y. Yang and X. Liu. A re-examination of text categorization methods. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 42–49. ACM, 1999.

[36] H.J. Zeng, X.H. Wang, Z. Chen, H. Lu, and W.Y. Ma. Cbc: Clustering based text classification requiring minimal labeled data. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 443–450. IEEE, 2003.

[37] T. Zhang and F.J. Oles. Text categorization based on regularized linear classification methods. *Information retrieval*, 4(1):5–31, 2001.

A Code

Code that crawls the LOS-website, creates the folder structure according to the LOS-structure and then saves the web pages belonging to each class.

Code listing 8: CrawlLOS.java

```
1  package Useful;
2
3  import org.jsoup.Jsoup;
4  import org.jsoup.nodes.Document;
5  import org.jsoup.nodes.Element;
6  import org.jsoup.select.Elements;
7
8  import java.io.*;
9  import java.util.ArrayList;
10
11 public class CrawlLOS {
12     public static void traverseAndMakeFoldersFromLosWeb(String
           rootUrl, int level, String folderpath) {
13         ArrayList<String> temaerUrls = new ArrayList<String>();
14         ArrayList<String> temaerNames = new ArrayList<String>();
15         try {
16             Document doc = Jsoup.connect(rootUrl).get();
17             Element body = doc.body();
18             Elements main = body.select("#main");
19             Elements elements = null;
20             if(level <= 3) {
21                 elements = main.get(0).select(".elements");
22             }
23             else {
24                 elements = main.get(0).select(".netresource");
25             }
26
27             for(Element e : elements) {
28                 Elements tema = e.select("a");
29                 temaerUrls.add(tema.get(0).attr("href"));
30                 temaerNames.add(tema.get(0).text());
31             }
32         } catch (IOException e) {
33             e.printStackTrace();
34         }
35         if(level <= 3) {
36             int count = 0;
37             for(String url : temaerUrls) {
38                 File folder = new File(folderpath +
                       temaerNames.get(count) + "/");
39                 if(!folder.exists()) {
40                     folder.mkdir();
```

```
41                  }
42                  String spaces = "";
43                  if(level == 2)
44                      spaces += "-> ";
45                  if(level == 3)
46                      spaces += "--> ";
47                  System.out.println(spaces + "Folder: " +
                        temaerNames.get(count));
48                  traverseAndMakeFoldersFromLosWeb("http://los.difi.no/"
                        + url, level + 1, folderpath +
                        temaerNames.get(count) + "/");
49                  count++;
50              }
51          }
52          else {
53              int count = 0;
54              for(String url : temaerUrls) {
55                  try {
56                      File file = new File(folderpath,
                            temaerNames.get(count));
57                      if(!file.exists()) {
58                          System.out.println("File: " +
                                temaerNames.get(count) + " url: " +
                                url);
59                          Document doc = Jsoup.connect(url).get();
60                          BufferedWriter out = new
                                BufferedWriter(new
                                OutputStreamWriter(new
                                FileOutputStream(file),"UTF8"));
61                              //new BufferedWriter(new
                                    FileWriter(folderpath + "/"
                                    + temaerNames.get(count)));
62                          out.write(doc.html());
63                          out.close();
64                      }
65                      count++;
66                  } catch (IOException e) {
67                      System.out.println("Error on file: " +
                            temaerNames.get(count) + " url: " + url);
68                      e.printStackTrace();
69                  }
70              }
71          }
72      }
73      public static void main(String[] args) {
74          traverseAndMakeFoldersFromLosWeb("http://los.difi.no/struktur/",
                1, "/media/Master/LOS/");
75      }
76  }
```

Code listing 9: WebPageClassifier.java

```
1  package Classifying;
2
3  import weka.classifiers.Classifier;
4  import weka.classifiers.Evaluation;
5  import weka.core.*;
6  import weka.filters.Filter;
7  import weka.filters.unsupervised.attribute.StringToWordVector;
8
9  import java.io.File;
10 import java.io.FileOutputStream;
11 import java.io.ObjectOutputStream;
12 import java.io.Serializable;
13 import java.text.SimpleDateFormat;
14 import java.util.ArrayList;
15 import java.util.Calendar;
16 import java.util.Collections;
17 import java.util.Random;
18
19 public class WebPageClassifier implements Serializable {
20     private static final long serialVersionUID = 1L;
21     // The root folder of the training data files
22     private static final String rootF = "/media/Master/LOS/";
23     // Training data
24     private Instances m_data = null;
25     // if it needs to rebuild the classifier because there have
           been added new instances to the training data or not
26     private boolean m_UpToDate;
27     // Filter
28     private StringToWordVector m_Filter = new
           StringToWordVector();
29     // Classifier
30     private Classifier m_Classifier = null;
31
32     /**
33      * Construction of empty training set
34      * Adds the classes to the training data
35      * @throws Exception
36      */
37     public WebPageClassifier(Classifier classifier, String[]
           classifierOptions) throws Exception {
38         m_Classifier = classifier;
39         m_Classifier.setOptions(classifierOptions);
40         String datasetName = "WebpageClassification";
41
42         FastVector attributes = new FastVector(2);
43         attributes.addElement(new Attribute("Content",
               (FastVector) null));
```

```
44
45          // Class attributes
46          File root = new File(rootF);
47          ArrayList<String> classes = new ArrayList<String>();
48          for(String f1 : root.list()) {
49              File folder1 = new File(rootF + "/" + f1);
50              for(String f2 : folder1.list()) {
51                  File folder2 = new File(rootF + "/" + f1 + "/"
                        + f2);
52                  for(String folder : folder2.list()) {
53                      if(!classes.contains(f2 + " - " + folder)) {
54                          classes.add(f2 + " - " + folder);
55                      }
56                  }
57              }
58          }
59          FastVector classValues = new FastVector(classes.size());
60          for(String c : classes) {
61              classValues.addElement(c);
62          }
63          attributes.addElement(new Attribute("Class",
                classValues));
64
65          // Create a dataset with initial capacity of 100, plus
                set the index of class.
66          m_data = new Instances(datasetName, attributes, 100);
67          m_data.setClassIndex(m_data.numAttributes() -1);
68      }
69
70      /**
71       * Updates the model with the given training page.
72       * Makes an instance of the data, sets it's class and adds
             it to m_data.
73       * Also sets m_UpToDate to false so that it will rebuild
             the classifier before classifying new documents.
74       * @throws Exception
75       */
76      public void updateModel(String content, String classValue)
            throws Exception {
77          // Convert content string into an instance
78          Instance instance = makeInstance(content, m_data);
79
80          // Add the class value to the instance
81          instance.setClassValue(classValue);
82
83          // Add teh instance to the training data
84          m_data.add(instance);
85
86          m_UpToDate = false;
```

```
87      }
88
89      /**
90       * sets the options for the filter, i.e.:
91       * adds TD/IDF, normalizes vectors based on document length
92       * @throws Exception
93       */
94      public void setFilterOptions() throws Exception {
95          // Initialize filter
96          //TF_IDF
97          m_Filter.setIDFTransform(true);
98          m_Filter.setTFTransform(true);
99          // Normalize document lengths
100         m_Filter.setNormalizeDocLength(new
                SelectedTag(StringToWordVector.FILTER_NORMALIZE_ALL,
                StringToWordVector.TAGS_FILTER));
101         m_Filter.setInputFormat(m_data);
102     }
103
104     /**
105      * Method that sets the options of the filter, filters the
             data with this filter and then build the classififer.
106      * Sets m_UpToDate to true so that it will not need to
             perform this task when it is already done and no new
             data is added.
107      * @throws Exception
108      */
109     public void buildClassifier() throws Exception {
110         setFilterOptions();
111
112         Instances filteredData = Filter.useFilter(m_data,
                m_Filter);
113
114         m_Classifier.buildClassifier(filteredData);
115
116         m_UpToDate = true;
117     }
118
119     /**
120      * Classifies a given document
121      * @param content Content of the document
122      * @param actualClass the class that the document belongs to
123      * @throws Exception
124      */
125     public void classifyContent(String content, String
            actualClass) throws Exception {
126         // Check if a classifier already has been built
127         if(m_data.numInstances() == 0) {
128             throw new Exception("No classifier available");
```

```
129         }
130
131         if(!m_UpToDate) {
132             System.out.println("Model not up to date, building
                    new classifier");
133
134             buildClassifier();
135         }
136
137         Instances testSet = m_data.stringFreeStructure();
138
139         Instance instance = makeInstance(content, testSet);
140
141         // Filter the instance
142         m_Filter.input(instance);
143         Instance filteredInstance = m_Filter.output();
144
145         // Get the index of the predicted class value
146         double predicted =
                m_Classifier.classifyInstance(filteredInstance);
147
148         double [] dist =
                m_Classifier.distributionForInstance(filteredInstance);
149         ArrayList<MyClass> myClasses = new ArrayList<MyClass>();
150         int count = 0;
151         for(double d : dist) {
152             myClasses.add(new MyClass(count, d));
153             count++;
154         }
155         Collections.sort(myClasses);
156
157         System.out.println("\nContent classified as: " + (int)
                predicted + ": " +
158                 m_data.classAttribute().value((int) predicted)
                        + "\nActual class:              " +
159                 actualClass + "\n    Probability: " +
                        dist[(int) predicted]);
160
161         if(!actualClass.equals(m_data.classAttribute().value((int)
                predicted))) {
162             System.out.println("WRONG! other possibilities:\n");
163             count = 0;
164             for(MyClass m : myClasses) {
165                 count++;
166                 System.out.println('"' +
                        m_data.classAttribute().value(m.getId()) +
                        '"' + " with prob: " + m.getPercent());
167                 if(m.getPercent() == 0.0 || count == 30) {
168                     break;
```

```
169                     }
170                 }
171             }
172     }
173
174     /**
175      * Converts the content of a file into an instance
176      * @param content The content of the file
177      * @param data the data already in the system (m_data)
178      * @return an Instance of the content
179      */
180     public Instance makeInstance(String content, Instances
            data) {
181         Instance instance = new Instance(2);
182
183         Attribute contentAtt = data.attribute("Content");
184         instance.setValue(contentAtt,
                contentAtt.addStringValue(content));
185
186         instance.setDataset(data);
187
188         return instance;
189     }
190
191     /**
192      *
193      * @param root The string for the root folder of the files
194      * @param processed if it should look for files that ends
                with .processed or not
195      * @return ArrayList with type FilePathAndClass with all
                the filepaths
196      */
197     public ArrayList<FilePathAndClass>
            getAllFilePathsFromFolderStructure(String root, boolean
            processed) {
198         ArrayList<FilePathAndClass> filePaths = new
                ArrayList<FilePathAndClass>();
199
200         File _root = new File(root);
201         if(_root.exists()) {
202             for(String f1 : _root.list()) {
203                 File folder1 = new File(root + "/" + f1);
204                 for(String f2 : folder1.list()) {
205                     File folder2 = new File(root + "/" + f1 +
                            "/" + f2);
206                     for(String f3 : folder2.list()) {
207                         File folder3 = new File(root + "/" + f1
                                + "/" + f2 + "/" + f3);
208                         for(String file : folder3.list()) {
```

```
209                             if(processed) {
210                                 if(file.endsWith(".processed"))
                                        {
211                                     filePaths.add(new
                                            FilePathAndClass(f2 + "
                                            - " + f3, root + "/" +
                                            f1 + "/" + f2 + "/" + f3
                                            + "/" + file));
212                                 }
213                             }
214                             else {
215                                 if(!file.endsWith(".processed"))
                                        {
216                                     filePaths.add(new
                                            FilePathAndClass(f2 + "
                                            - " + f3, root + "/" +
                                            f1 + "/" + f2 + "/" + f3
                                            + "/" + file));
217                                 }
218                             }
219                         }
220                     }
221                 }
222             }
223         }
224         return filePaths;
225     }
226
227     /**
228      * runs a 5 run, 5 fold evaluation of the learned data
229      * @throws Exception
230      */
231     public void performEval() throws Exception {
232         int runs = 5;
233         int folds = 5;
234
235         Classifier classifier = (Classifier)
                Utils.forName(Classifier.class,
                m_Classifier.getClass().getName(),
                m_Classifier.getOptions());
236         setFilterOptions();
237         Instances filteredData = Filter.useFilter(m_data,
                m_Filter);
238
239         for(int i = 0; i < runs; i++) {
240             int seed = i + 1;
241             Random random = new Random(seed);
242
243             Instances randData = new Instances(filteredData);
```

```
244             randData.randomize(random);
245             if(randData.classAttribute().isNominal()) {
246                 randData.stratify(folds);
247             }
248
249             Evaluation evaluation = new Evaluation(randData);
250             for(int n = 0; n < folds; n++) {
251                 System.out.println("Fold " + n);
252                 Instances train = randData.trainCV(folds, n);
253                 Instances test = randData.testCV(folds, n);
254
255                 Classifier classifierCopy =
                        Classifier.makeCopy(classifier);
256                 classifierCopy.buildClassifier(train);
257                 evaluation.evaluateModel(classifierCopy, test);
258             }
259
260             // output evaluation
261             System.out.println();
262             System.out.println("=== Setup run " + (i + 1) + "
                    ===");
263             System.out.println("Classifier: " +
                    classifier.getClass().getName() + " " +
                    Utils.joinOptions(classifier.getOptions()));
264             System.out.println("Dataset: " +
                    m_data.relationName());
265             System.out.println("Folds: " + folds);
266             System.out.println("Seed: " + seed);
267             System.out.println();
268             System.out.println(evaluation.toSummaryString("===
                    " + folds + "-fold Cross-validation run " + (i +
                    1) + "===", false));
269             System.out.println();
270
271         }
272     }
273
274     /**
275      * Gets the time
276      * @return String on the format yyyy-MM-dd HH:mm:ss
277      */
278     public static String now() {
279         final String DATE_FORMAT_NOW = "yyyy-MM-dd HH:mm:ss";
280         Calendar cal = Calendar.getInstance();
281         SimpleDateFormat sdf = new
                SimpleDateFormat(DATE_FORMAT_NOW);
282         return sdf.format(cal.getTime());
283     }
284
```

```
285     /**
286      * Writes the learned model to a file for later usage
287      * @param path The place it should be stored
288      * @param modelFileString Name of the file
289      * @throws Exception
290      */
291     public void writeModelToFile(String path, String
            modelFileString) throws Exception {
292         File f = new File(path + modelFileString);
293         if(f.exists()) {
294             f.renameTo(new File(path + modelFileString +
                    ".old"));
295             f.delete();
296         }
297         FileOutputStream modelOutFile = new
                FileOutputStream(path + modelFileString);
298         ObjectOutputStream modelOutObjectFile = new
                ObjectOutputStream(modelOutFile);
299         modelOutObjectFile.writeObject(this);
300         modelOutObjectFile.flush();
301         modelOutFile.close();
302     }
303
304     /**
305      * Creates a string with the classifier used and all it's
             options
306      * @return String
307      */
308     public String getClassifierString() {
309         String options = " options:";
310         for(String s : m_Classifier.getOptions()) {
311             options += " " + s;
312         }
313         return m_Classifier.getClass().toString() + options;
314     }
315 }
```

B Enclosed ZIP Archive

This appendix describes the contents of the ZIP archive enclosed with this thesis. The archive is available through the DAIM system at `http://daim.idi.ntnu.no`

The folder structure of the archive is as following:

- **LOS**
  The folder structure of `http://los.difi.no/struktur/`
  - Main categories
    - Sub categories
      - Classes
- **LOStest**
  The test-set
- **src**
  Source files for the prototype
  - **Classifying**
    The parts that perform classification
  - **Crawl**
    The part that crawls the los-webpage
  - **PreProcessing**
    The parts that perform preprocessing
- **stop.txt**
  Text-document that lists all stop words I have used