Contextual text classification for parametric mappings in a distributed search engine

Jimmy Pettersson

December 4, 2011

Abstract

The purpose of this thesis was to research the possibility to automatically map queries down to parameters in a distributed search engine. Another objective was to investigate if the mappings could be made good enough to improve search results and user experience.

Two Bayesian classifiers and one classifier using a combination of tf-idf, cosine similarity and the vector space model were implemented and evaluated against each other. The three implementations were all tuned to better fit in with the distributed model of the search engine.

Initial results indicated that one of the simpler Bayesian implementations outperformed the other two implementations. The later parts of the work were dedicated to optimize this implementation, ending up in a blind test that showed an improvement in both search results and user experience.

The final conclusion is that the thesis overall was a success but there is a lot of room for improvement both in and around the current working implementation.

.

Acknowledgments

I would like to thank the following people for making this thesis possible

My examiner at Lund University Pierre Nugues for his guidance throughout this thesis and for taking the time to meet and discuss my work week after week.

Lars Hard and Axel Antonsson at Expertmaker for taking the time to be my supervisors and for their invaluable input.

Everyone Expertmaker for making me feel like a part of the team and for their input on various topics. An extra thank you to the people who shared office with me over the last months for putting up with all my moans and groans during the writing of this report.

Lastly a big thanks my mother Marie, my father Mats and my sister Sandra along with all my friends that have been standing by me for the last five years. You made this possible.

Contents

.

1 Introduction

The field of natural language processing, or NLP for short, has been studied for over 60 years and still presents problems to engineers and academics that seem insurmountable. How do we teach computers to distinguish between different semantic aspects of everyday life? Can a machine give us suggestions based on a set of parameters? Can it mimic the behaviour of a human when answering these questions? There are a wide array of sub-fields within NLP that tries to tackle these problems in different ways and this report will present a proposed solution to map entities based on user queries to a special search engine.

1.1 Boomstep

Boomstep is the result of 5 years of developing a tool called the *Knowledge Designer* at Expertmaker AB (2011), a company based in Malmo, Sweden. Boomstep is a search engine that is builds on artificial intelligence, AI, and harnesses the power of AI to not only display relevant search results but also act as a suggestion machine, answering everyday questions such as

> Where can I eat cheap in Stockholm?

or

> Which breed of dog is best for me if it is my first dog?

The search engine itself consists of a set of smaller specialized search engines, referred to as nodes.

1.2 Vertical search

Going back to the previously stated questions, or queries, there is a node about restaurants in Stockholm as well as a node about dogs. These nodes handle nothing else except the area of expertise that they are supposed to do. This design of the search engine and its nodes is what makes it a so called vertical search engine. A node in a vertical search engine is confined by the data within that node and this makes it possible to work with a very limited subset of lexical content once you have found the right node. A node about sports will have a high frequency on words like *soccer*, *hockey* and *referee*, while a node about dogs will have a low frequency on the same words if they are present at all.

2 Problem

2.1 Background

In order to answer questions of the form *Where can I eat cheap in Stockholm?*, a set of parameters has to be manually set by the user in Boomstep which can be cumbersome if used extensively. The problem I had to solve was to research the possibility of automating this process based on the query the user supplied to the search engine. Given a query consisting of a set of words the algorithm has to map to a set of parameters. The number and type of parameters that can be set for a given node are specific to that node.

Setting the parameters for a node does not only give the search engine the power to give suggestions based on the query but it also yields better search results due to the fact that setting a parameter will filter out the search results that did not match the parameter.

Throughout the implementation phase of the project, I have been working under the assumption that based on the query, the correct node is found by the search engine. Should an incorrect node be found, it is most likely that the mapping will not produce any results, or worse, produce erroneous results.

2.2 Related work

Document classification and categorization dates back to somewhere in the late 1950s. Back then document classification was performed using heuristic methods by letting an expert set up rules and applying these to solve the task of classification.

With the introduction of the Internet, the amount of available data has exploded and the need to automatically be able to classify texts came along with it. Historically, classification can be divided in to three branches.

> Supervised document classification where some external entity (mostly human feedback) provides the data for the correct classification of documents.
>
> Semi-supervised document classification where to some extent an external entity provides the data, and the rest is unsupervised.
>
> Unsupervised document classification where the classification is done without any external influence.

The increasing demand on this kind of technology led to more research in the area and as a result a number of different techniques has been developed over the years. Popular ones like *Naive Bayes*, *Support Vector Machines* and *Nearest Neighbour* has gained acceptance within the community over the years. For an introduction to these techniques see Manning et al. (2008). Newer methods like *Random forests* (Breiman, 2001) are trying to prove their worth, applying a more complex approach to the classification problem.

Finding similar work closely related to the work done in this thesis is not possible since I have been working on a closed source search engine and this is the first attempt at automatically map queries to parameters.

3 The node

A node can be developed by anyone with the help of the *Knowledge Designer* tool from Expertmaker. Often nodes are developed by people or organizations with expertise in a certain field e.g the dogs node was developed by a kennel club in Sweden. This approach is a strength of the search engine to ensure that the data presented is valid and accurate. It is also adds credibility to the suggestions given by the search engine, as it acts on the behalf of an expert or person that is well informed in a particular field.

A node can be either static or dynamic depending on the needs to update the data. The dogs node, containing different breeds of dogs, is static since the need to dynamically add new breeds is low. You want the data to be added manually through the *Knowledge Designer* by an expert in his or her particular field. Some nodes have a higher need to be dynamic, like the news nodes which is constantly updated with news.

A node in Boomstep is a data structure stored in a XML file. It contains data that represents the node, like inputs, outputs, examples as well as meta data. The general structure of the node can be viewed in Fig 1.

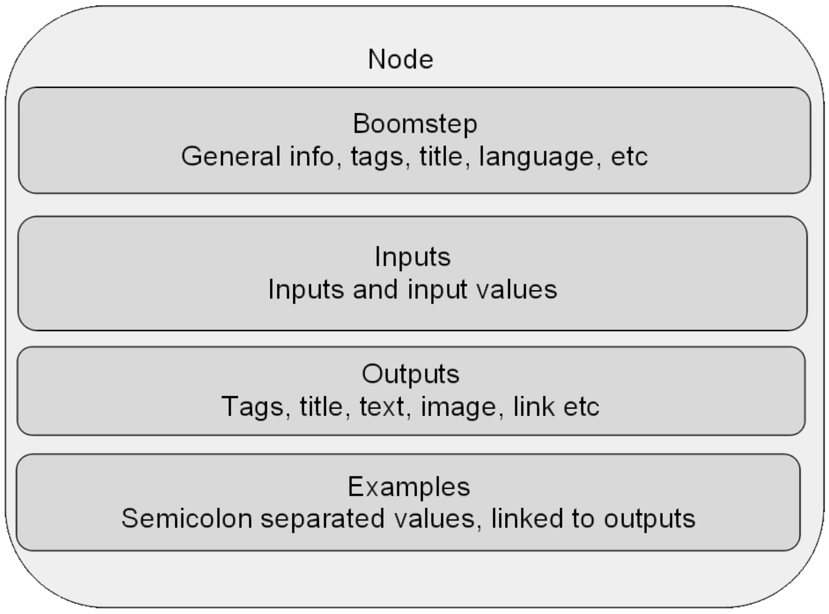
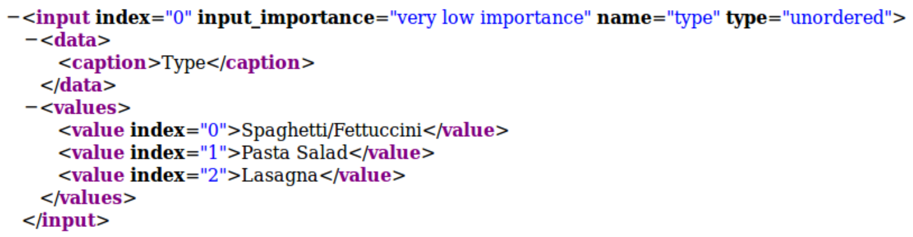

Node
Boomstep
General info, tags, title, language, etc
Inputs
Inputs and input values
Outputs
Tags, title, text, image, link etc
Examples
Semicolon separated values, linked to outputs

Figure 1: Overview of a node and its elements.

3.1 The input element

The input element is a specification of an attribute that can be set within the node. The input element contains different attribute properties that tells the search engine how important the input is to the search and if it is an ordered input or not. An ordered input is an input whose attribute values can be ordered in some way like the difficulty of a video game can have the ordered attribute values *easy, normal, hard*, whereas an unordered input can be something like colors. Figure 2 shows the structure of the input with the attribute type *Type* in the pasta node with the attribute values *Spaghetti/Fettuccine*, *Pasta Salad* and *Lasagna* along with some attribute properties.

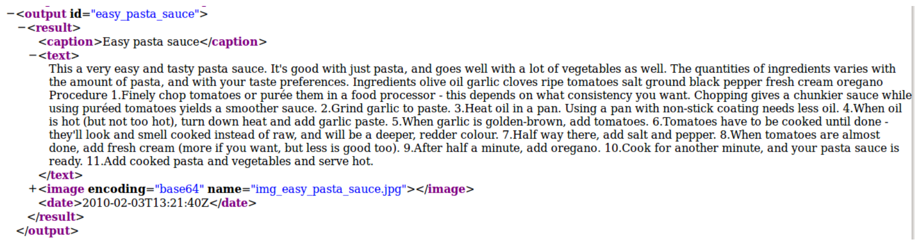


```
-<input index="0" input_importance="very low importance" name="type" type="unordered">
  -<data>
     <caption>Type</caption>
   </data>
  -<values>
     <value index="0">Spaghetti/Fettuccini</value>
     <value index="1">Pasta Salad</value>
     <value index="2">Lasagna</value>
   </values>
 </input>
```

Figure 2: Input element extracted from a XML file.

3.2 The output element

The output element represents the result displayed to the user. It often consists of a title, a body of text and an image. It also contains meta data such as id, the date it was published and tags that helps the search engine to match against the output. Figure 3 shows the structure of the output element for an easy pasta sauce.

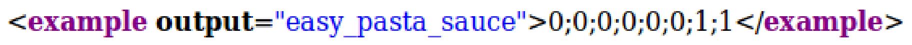


```
-<output id="easy_pasta_sauce">
  -<result>
     <caption>Easy pasta sauce</caption>
    -<text>
       This a very easy and tasty pasta sauce. It's good with just pasta, and goes well with a lot of vegetables as well. The quantities of ingredients varies with
       the amount of pasta, and with your taste preferences. Ingredients olive oil garlic cloves ripe tomatoes salt ground black pepper fresh cream oregano
       Procedure 1.Finely chop tomatoes or purée them in a food processor - this depends on what consistency you want. Chopping gives a chunkier sauce while
       using puréed tomatoes yields a smoother sauce. 2.Grind garlic to paste. 3.Heat oil in a pan. Using a pan with non-stick coating needs less oil. 4.When oil
       is hot (but not too hot), turn down heat and add garlic paste. 5.When garlic is golden-brown, add tomatoes. 6.Tomatoes have to be cooked until done -
       they'll look and smell cooked instead of raw, and will be a deeper, redder colour. 7.Half way there, add salt and pepper. 8.When tomatoes are almost
       done, add fresh cream (more if you want, but less is good too). 9.After half a minute, add oregano. 10.Cook for another minute, and your pasta sauce is
       ready. 11.Add cooked pasta and vegetables and serve hot.
     </text>
    +<image encoding="base64" name="img_easy_pasta_sauce.jpg"></image>
     <date>2010-02-03T13:21:40Z</date>
   </result>
 </output>
```

Figure 3: Output element extracted from a XML file.

3.3 The example element

An example element consists of a set of semicolon separated values, a feature vector, that represents the parameters that has to be set in order to match a given output. The example element are linked to an output element via the id of the output. The relationship between the example elements and the output elements are of many-to-one. An output can have many examples while an example can only have one output. Figure 4 shows what parameters that will match the output element in Figure 3.



```
<example output="easy_pasta_sauce">0;0;0;0;0;0;1;1</example>
```

Figure 4: Example element extracted from a XML file.

The first element in the feature vector in Figure 4 refers to the input element in Figure 2. The value for this element has been specified to 0, indicating that the *Type* should be *Spaghetti/Fettuccine*. In the current version of the search engine, these parameters has to be set manually.

.

4 Document classification

Document classification is a sub field of document retrieval, which itself is a sub field of information retrieval, IR. The field of IR is broad and consists of the study of intra/inter-search of documents as well as searching the Internet. It spans several fields including mathematics, computer science and NLP.

The general idea behind document classification is simple. You have a document you want to classify in some way. It can be a document of a review that is to be classified as positive or negative or it can be a document of a recipe that is to be classified as a recipe for a meat dish, a seafood dish or a vegetarian dish. The human approach of classifying the document would be to read it and use our knowledge and experience to tell which category it belongs to. To automate this process using a computer there are several different methods. What they have in common is that they attempt to mimic the human approach in using other documents they have "read" before drawing a conclusion about the document that is to be classified.

5 Naive Bayes

The Naive Bayes, NB, classifier is a probabilistic classifier that assumes independence of features, meaning that the existence or absence of a feature (word) is independent of the existence or absence of other features (words). This is obviously a simplification of the real world and is counter-intuitive since there is often a high dependency among words. Despite this, the NB classifier performs on par with other more sophisticated classifiers and even outperforms them in certain cases (Zhang, 2004). The NB classifier has been proven to be effective in real world applications such as text classification (Lewis, 1998; McCallum and Nigam, 1998; Rennie, 1999), medical analysis (Friedman et al., 1997), spam filtering (Rish, 2001) and analyzing Arabic texts related to fanaticism (Almonayyes, 2006). A strength of the NB classifier is that it does not require large amounts of training data to perform classification (Forman and Cohen, 2004).

The probability model of a document D being in class C can be written as $P(C|D)$ where D consists of n words $w_0...w_{n-1}$. Given this we can expand the model to $P(C|w_0, ..., w_{n-1})$. Now applying Bayes' Theorem to this model we can write

$$P(C|D) = \frac{P(C)P(D|C)}{P(D)} = \frac{P(C)P(w_0, ..., w_{n-1}|C)}{P(w_0, ..., w_{n-1})}$$

where $P(C)$ is the probability of the given class C. In other words, the probability of a document D being of class C without considering the contents of D. $P(D)$ is the probability of the specific document D occurring. Finally $P(D|C)$ is the probability of that given a class C, the words in D exist in that class. Since the denominator $P(D)$ is independent of C and is fixed over all possible classes, it can be omitted in the calculations and we get

$$P(C|D) = P(C)P(D|C) = P(C)P(w_0, ..., w_{n-1}|C). \tag{1}$$

Omitting $P(D)$ will change the absolute value $P(C|D)$ but will not change the relative values for calculating $P(C_i|D)$ over a set of classes C_i. $P(D|C)$ is calculated by taking the probability of each word $w_0, ..., w_{n-1}$ and calculate the product of these. This is often referred to as the naive step, where the word independence assumption is realized. So we get

$$P(D|C) = \prod_{i=0}^{n-1} P(w_i|C)$$

and applying this to (1) yields

$$P(C|D) = P(C) \cdot \prod_{i=0}^{n-1} P(w_i|C).$$

where

$$P(w_i|C) = \frac{count(w_i)}{\sum_i count(w_i)}$$

where $count(w_i)$ is the number of times w_i appears in C. In other words $P(w_i|C)$ is equal to the number of times w_i occurs in C divided by the total number of words in C. Calculating $P(C_i|D)$ for a set of different classes $C_0...C_m$ will generate a score for each of the classes and the class with the highest score will determine the class of the document D.

The NB classifier is commonly criticized, as mentioned above, for its assumption that the words appearing in a document are independent of each other. This criticism boils down to the fact that there is a dependency among words and therefore the NB classifier is claimed to be less accurate than more complex classifiers such as Support Vector Machines, k-nearest neighbours etc. However, the probabilistic classifiers that follows the *maximum a posteriori probability* will correctly classify a document as long as the correct class is more probable than the other classes. This means that the classifier itself does not have to do a "perfect" classification of the correct document, just do a good enough job to push it to the top over the other classified documents. Summing this up, if the overall classifier is performing good enough it can disregard the underlying erroneous assumptions about word independence in its naive probability model.

5.1 Implementation

5.1.1 Training on outputs

I implemented a Bayesian classifier that has been trained on the output elements of the node. The training data were derived from the output elements of the node. The overall idea behind the implementation was to examine the intersection of the top scoring outputs to see if any parameters could be mapped this way.

Step 1 The first step of the process is to take the user submitted query and preprocess it to be able to extract as much useful information as possible from it. The preprocessing consists of splitting the query in to a list of separate words.

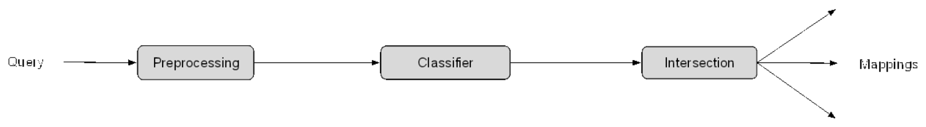


Figure 5: Flow showing the process from input query to outputted mappings.

The list is then scanned for so-called stop words, that is words that provide little to zero lexical content like the words *the* and *I*. If a word is a stop word it is removed from the list. If not, the word is lower cased and trimmed of white spaces. When the preprocessing step is finished the list of words is sent in to the classifier.

Step 2 The classification step of the process takes the preprocessed query and turns it into to a document so that the classifier can compare it to other documents that it has been trained on. As stated, this implementation makes use the the Bayesian classifier described in section 5 but with some modifications. The standard Bayesian classifier only returns the document which has the highest probability among the trained documents to be the most likely match to the query. The classifier in this implementation will return the N most probable documents to the submitted query, and these N documents will be fed in to the next step which is the intersection step. N is an integer that can be specified to the program.

Step 3 The third step of the mapping process is to examine the examples of the N most probable documents. Depending on how much information the node holds the number of examples that will be available for intersection will vary but a minimum of one example per output is guaranteed. All the examples matching the input documents are extracted and then intersected with respect to each input value. Should enough[1] input values intersect for the given examples the input (parameter) with the corresponding input value is going to be considered a correct mapping of that parameter.

Example Fig 5 shows an example of the classification process. Note that this example is constructed to clarify the process of mapping. The search results for the query are most likely not the ones described here and only three examples will be intersected. The search engine is assumed to find the pasta node when the query is supplied.

Consider the query *"the Easy pasta sauce!"*. It contains a stop word, *the*, an upper case letter, a double space and a punctuation at the end. After the preprocessing in step 1 we will end up with a list of words that looks like [*easy, pasta, sauce*]. This list is fed in to the classifier which will return the top three outputs from the classification. Now we go through the example elements connected to the returned outputs and extract the semicolon separated values for each of the example elements. They might look, since this is a constructed example, something like:

[1]A threshold for intersection can be specified in the program

Input	Type	Ingredient	Level	Time	Spicy	No dairy	No nuts	No shellfish
Example #0	0	1	0	3	1	1	1	2
Example #1	0	2	0	2	0	3	0	1
Example #2	0	0	0	1	2	1	2	0

Now it's time for the intersection step and we can easily see that we have 100% overlap on inputs *Type* and *Level* with the respective values 0 and 0. The intersection will return something like [*0:0, 2:0*] indicating that input 0 should have its value set to 0 and input 2 its value set to 0. This corresponds to setting the *type* parameter to *pasta* and the *level* (of difficulty) parameter to *easy*. Summarizing this we have mapped the query *the Easy pasta sauce!* to the the parameters *type* → *pasta* and *level* → *easy*.

Theoretically this approach seems likely to produce good results under the right circumstances however practically it has a few shortcomings. The first shortcoming is the criticism presented in the Bayesian classifier in 5 where the classifier can ignore its underlying flaws as long as the correct classification is made relative to the other documents in the training set. What you get from the Bayesian classifier is indeed a sorted list of documents where the top element is the most likely classification. But when the top N documents is needed for a parameter mapping you have to venture down the list of classification scores and the uncertainty increases with each document in that list. There is no way of explicitly telling how good of a match the fifth document is, only a score indicating that it has the fifth highest score in the classification. In the worst case scenario, this document can be totally irrelevant to the query, it just happened to get the fifth highest score in the classification since there might only have been four documents that would have been considered relevant to a human.

Another shortcoming, which is independent of the classifier, is that it is highly dependent on the set of examples that will be extracted from the classification. The choice of extracting examples from the top N most likely classifications is arbitrary but choosing too few will result in incorrect mappings since it would be more likely to find intersections on parameters that the user were not interested in and choosing too many will introduce a lot of variance that will result in no mappings at all. This could be avoided if there were consistency in the nodes, however the amount of data and number of examples per given input present in an arbitrary node will vary greatly from any other arbitrarily chosen node.

5.1.2 Trained on inputs

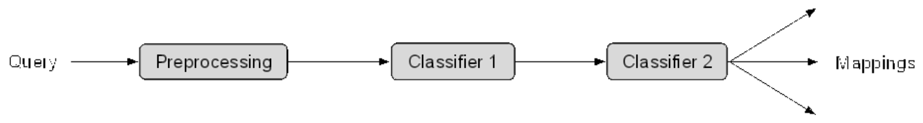


Figure 6: Flow showing the process from input query to outputted mappings.

As a second approach I trained the classifier on the inputs rather than the outputs and this is a more direct approach to the mappings as opposed to the previous method. To clarify what being trained on the inputs means, it is to train the classifier on the words that are present in the *caption* and *values* tags in the input element, see Fig 2. The amount of textual data in an input element is often very sparse but this will be shown to have a smaller impact on the classifier than what could be anticipated due to the design of the search engine.

The implementation consists of two linked classifiers to do the mapping. The first classifier determines which inputs that are relevant to the query. These inputs are the fed to the second classifier which determines which of the input values to map. The first step of the implementation is identical to that of the previous one (query preprocessing).

Step 2 After the query preprocessing the query is turned into a document and fed into the first classifier. This classifier calculates a classification score for each of the node inputs and if the score if above a certain threshold value it is deemed relevant to the query. The inputs that passed through the first classifier is then fed into the second classifier.

Step 3 The second classifier takes the query along with the inputs that came from the first classifier and compares them separately to the input values of each input. Here the standard approach of the Bayesian classifier is utilized and the input value with the highest classification score is the one that gets mapped unless one of two special cases are met. The two special cases are triggered when all of the input values get a classification score of 0.0, meaning that not a single word was matched over all input values. If there is a default input value specified when this happens the program will use the default input value in the mapping. If no default input value is specified then no mapping is performed.

Example Note that the same prerequisites as the previous example applies to this also. Consider the same query, *"the Easy pasta sauce!"*. The preprocessing of the query is identical to the previous example and we will again end up with the list [*easy, pasta, sauce*]. This list of words is turned in to a document and fed in to the first classifier which will give each input (a total of 8 for the pasta node) a classification score based on the list of words.

The first classifier will return a list of the inputs whose score is above a set threshold t. The set of inputs $\{input : input_{score} > t\}$ will be fed in to the second classifier. Assuming the first classifier has correctly classified the query we should now have the list [$input_0$, $input_2$], the same two inputs that had a 100% overlap in the previous example.

The second classifier will now for each of the inputs compare each of the input values to the query and return the most likely input value. So for our two inputs this will look like

> *query* compared to the type values [*Spaghetti/Fettuccine(pasta), Pasta Salad, Lasagna*]

query compared to the level values [*easy, intermediate, hard*]

The second classifier will return the input value with the highest score for each of the inputs that it classified. Again assuming a correct classification was made the input values *Spaghetti/Fettuccine(pasta)* and *easy* will be returned together with information about which input they correspond to and again we have the same mapping as in the previous example.

An obvious shortcoming of this implementation is the lack of training data that comes with it. An input is often described by only one or a couple of words and the same goes for the input values. To minimize the impact of this an additional text file can be specified during the training of the node. The text file makes it possible to add and remove words for each input and input value as well as specify default input values. Granted this does remove the automation of the system but will significantly improve the mapping results.

6 Tf-idf, Cosine similarity and Vector space model

6.1 Tf-idf

Term frequency-inverted document frequency, tf-idf for short, is a technique of weighting words in a set of documents to determine how significant a word is to a particular document. After the weighting scheme was proposed by Salton and Yang (1973) it quickly gained popularity (Hiemstra, 2000). A word with a high tf-idf value implies a strong relationship with the document(s) it appears in. If a query contains a word that has a high tf-idf value then the document(s) it appears in would probably be of interest to the user. The tf-idf value for a word is increased by the number of times the word appears in a document and the value is decreased for every other document the word appears in. As a motivator for using tf-idf when classifying documents consider the following example.

Say we have a training set of documents and want to determine which of the documents in the training set are most relevant, i.e has the highest classification score, to the query *the red car.* Starting out we can simply remove the documents that does not contain all three of the words in the query, assuming there are documents that contain all three words for the sake of argument. If we have a large training set we will be left with a large number of documents after the first elimination. To further separate the remaining documents we can determine the term frequency, the number of times a word appears in a document, for each of the three words. We then calculate the sum of the term frequencies for each document in the training set and sort them by the sums. However, the results will be skewed because the word *the* is the most common in the English language (Fry and Kress, 2006). This will put more weight on documents that happen to make use of the word *the* a lot, and less weight on documents that uses the other more descriptive terms *red* and *car.* This is where the inverse document frequency weighting scheme is applied to diminish the weight of frequent terms over a training set, and to increase the weight of rare terms in the training set.

The tf-idf method is, as implied by its hyphenated name a construction of two parts. The tf part, term frequency, and the idf part, inverted document frequency. The term frequency for a word w in a document D is defined as the number of times w appears in D divided by the total number of words in D. The inverted document frequency is an estimated measure of how important a word is to a document over the set of all documents. Mathematically it is defined as, for a word w

$$idf(w) = \log \frac{|D|}{|\{d : w \in d\}|}$$

where D is the total number of documents and $\{d : w \in d\}$ is the number of documents in which the word w appears. If the word w doesn't appear in any document the denominator will be zero. To prevent division by zero a smoothing of the term $\{d : w \in d\}$ is often applied so that the denominator becomes $1 + \{d : w \in d\}$. To get the tf-idf value for the word w the cross product $tf(w) \times idf(w)$ is calculated. Weighting words with tf-idf is often used together with the *Vector Space Model* proposed by Salton and Mcgill (1986) and *cosine similarity* (Salton, 1989), described below, to classify documents.

6.2 Cosine similarity

Cosine similarity is a method for determining the similarity between two vectors. When it comes to text classification, queries and documents are transformed into vectors for comparison. There are several different methods to transform a text in to a vector. One of the most popular ways of doing this is to use the tf-idf weights explained above to populate the vector. To determine how similar two texts are the angle between the two vectors are calculated. A smaller angle between the vectors implies a higher similarity and the opposite for larger angles.

The cosine similarity between two vectors u and v can be derived from the *Euclidean Dot Product*: $u \cdot v = ||u||||v|| \cos(\theta)$, (Sparr, 1994). It follows that

$$\cos(\theta) = \frac{u \cdot v}{||u||||v||} = \frac{\sum_{i=1}^{n} u_i \times v_i}{\sqrt{\sum_{i=1}^{n} u_i^2} \times \sqrt{\sum_{i=1}^{n} v_i^2}}$$

When comparing texts of different length, which is usually the case for matching queries against trained data, the vectors have to be sorted by the words to align them for comparison as well as padded with zeros at the end to be able to perform the dot product.

6.3 Vector space model

The vector space model is a method of mapping documents onto vectors to be able to represent documents in a computationally efficient way in the computer. The vector space model was first researched and used in the *SMART IR-system* (Buckley, 1985). A representation of a document using the vector space model looks like

$$D = (w_0, ..., w_n)$$

where w is a numerical representation, or weighting, of a word that occurs in the document. The numerical representation in the vector can be obtained using a wide array of different schemes. However the tf-idf weighting scheme described above is one of the most popular at the time of writing. Once a vector space model has been established for a set of documents, it is possible to use vector operations, e.g. cosine similarity, on the documents to compare them with other documents or queries.

Summary As seen above, the methods tf-idf, cosine similarity and the vector space model are not mutually exclusive to each other when it comes to document classification but rather the opposite. When used in conjunction with each other they produce a robust model for classifying documents.

6.4 Implementation

This section describes an implementation that utilizes tf-idf, cosine similarity and the vector space model.

6.4.1 Trained on inputs

This implementation is similar to the one described in 5.1.2. It consists of two linked classifiers that have been trained on the input elements of a node. The first step is again to preprocess the query and feed it in to the first classifier. Step two and three calculates the score for each of the node inputs but uses the Cosine similarity and the tf-idf weighting scheme when doing so instead of the computations described in 5.

Again the shortcomings of this implementation is the lack of training data which can be mitigated by adding an extra file containing training data.

.

7 Notes on implementation

This section will cover some implementation specific elements that either differs from the theory or falls outside of the boundary of document classification but still needs to be mentioned for the sake of the project.

7.1 Parser

At the beginning of the project a complete parser was implemented to structure the data and provide an easy way of accessing the data contained within a node. The parser consists of a central $Node$ object that parses and stores all the elements and element attributes of the node in specific objects such as an *Output* object for the output element along with its attributes. All the objects used for storing data have methods for easy retrieval of the data. Here follows a short explanation of all the parser objects.

- **Node:** The central parser object responsible for parsing and storing the parsed data.
- **Boomstep:** An object that holds the general information of a node such as title, language, description etc.
- **Example:** Holds all the parsed examples, the semicolon separated values, and links them to the right output.
- **Input:** Contains the attributes and data of the input element as well as a list of $InputValue$ objects linked to the input.
- **InputValue:** Contains the data and index of an input value element. Also linked to its parent Input object.
- **Output:** Holds the parsed output texts along with the attributes and tags for each output element.

The parser also has support for stemming words and adding synonyms to the parsed data but these features were not used in the final implementation. The stemming of words were not used due to the real time criterion of the program. The user submitted query has to be stemmed in order to be viable for document classification and the standard stemmers in the nltk package (2011) were too slow to meet the real time requirements. The choice of not using synonyms was made because the *synonyms module* in the nltk package did not perform well and caused erroneous mapping to occur in the implementations.

When the parsing of a node is done, it is written to a file on disk using Python's *pickle module* making it easy and fast to read into memory and keeping the data structure intact.

7.2 Logarithmic probabilities

From section 5, we have that

$$P(C|D) = P(C) \cdot \prod_{i=0}^{n-1} P(w_i|C).$$

However, the values of $P(w_i|C)$ is often small, magnitude 10^{-2} or less. When multiplying these values together the risk of arithmetic underflow in the computer is big and the resulting score will be output as 0.0 instead of something like $1.4353... \cdot 10^{-129}$. This is machine dependent and the number of representable bits will differ but this needs to be taken into account when implementing the scoring algorithm. To avoid the possibility of arithmetic underflow the logarithm of $P(C|D)$ is calculated using the logarithm law $\log(AB) = \log A + \log B$ which gives us

$$\log(P(C|D)) = \log(P(C)) + \sum_{i=0}^{n-1} \log(P(w_i|C)).$$

The logarithm law does not only preserve the relative scores among documents but also speeds up the calculations since it eliminates the need to multiply floating point numbers which is more time consuming than taking the logarithm of the elements and adding them together.

7.3 UML diagrams

Below are the UML diagrams for the classifier and parser implementations. The classifier UML diagram shown is for the Bayesian trained on inputs. The other classifier UML diagrams are practically identical with exception to names.

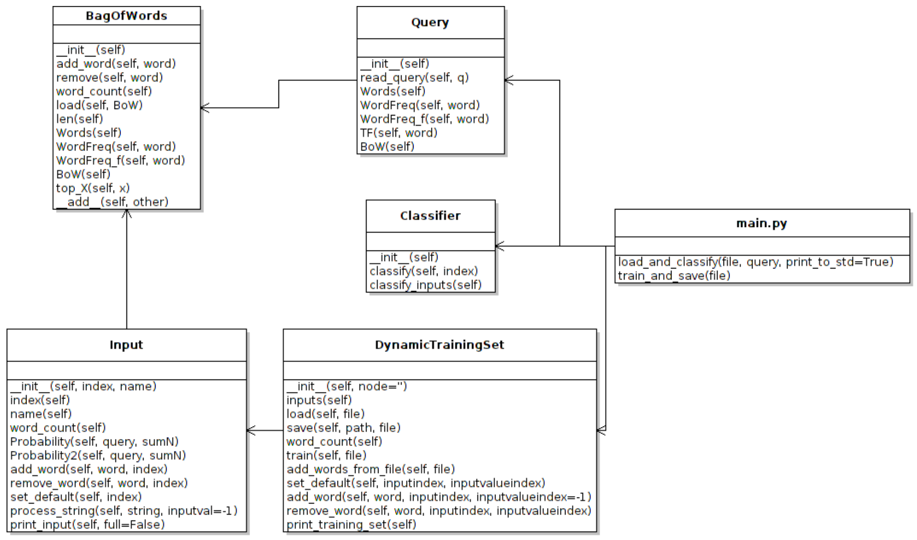


Figure 7: UML showing the structure of an implementation of a classifier.

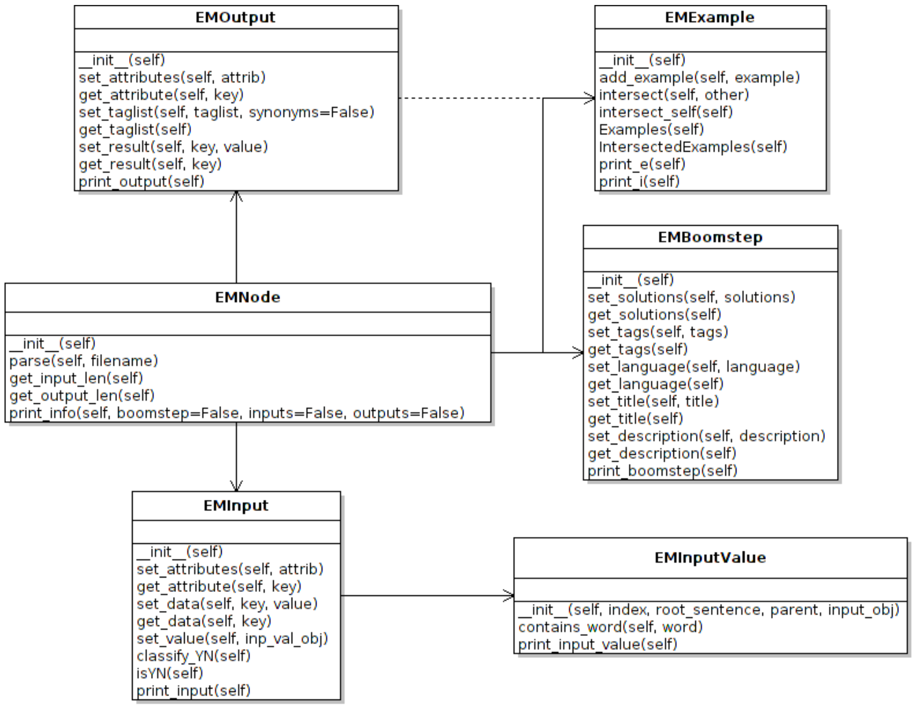


Figure 8: UML showing the structure of the parser implementation.

.

8 Experimental setup

8.1 Initial tests

To quickly evaluate the different classification implementations, a simple test suite was set up. The test suite consisted of 20 example queries that had been manually annotated with the expected outcome of the query when fed in to each of the classifiers, see Appendix A. This initial test laid the foundation for which one of the methods that would be focused for the remainder of the project. The tests were all conducted on the exact same training sets where applicable and they were also tweaked in between tests, meaning that the initial results below are the highest score the implementation received during all tests.

The Bayesian classifier that had been trained on outputs was the first one to be tested. Due to the nature of the implementation there is no clear score on the initial test suite. The number of possible combinations between the number of outputs taken into account and the intersection threshold made it hard to determine the highest score. However, one conclusion that can be drawn from this implementation is that it is the one that is most error prone among all implementations. Higher scores were achieved for a lower threshold on the intersection overlap but this also led to some erroneous mappings which has to be considered a severe shortcoming.

The Bayesian classifier that had been trained on the inputs was the next one to be tested. For the initial tests this implementation got 19 out of the 20 example queries correct after the addition of extra training data in an external text file. This was the highest score for the initial test out of the three implementations.

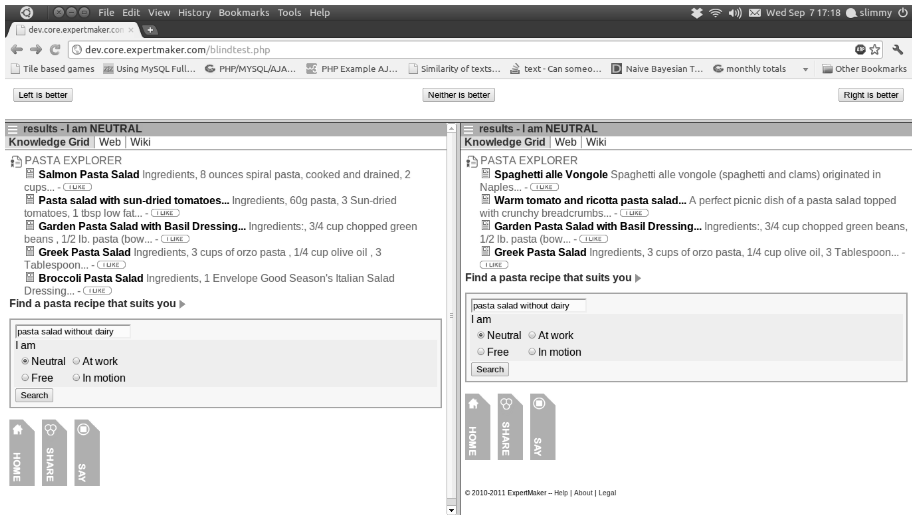
The implementation combining tf-idf, cosine similarity and the vector space model was the last one to be tested. This implementation scored 8 out of 20 on the initial tests, with the exact same training data as the Bayesian classifier that was trained on the inputs.

Given the results on the initial tests, the remainder of the thesis were focused on the Bayesian classifier that had been trained on the inputs. Most of the work on the classifier were spent on fine tuning thresholds, refactoring and optimizing the source code and implementing support for training and classifying multiple nodes concurrently.

8.2 Online blind test

To test the Bayesian classifier trained on the inputs more in depth and in a live environment, we set up an online blind test. The test was set up so that a user could type in a query about pasta recipes and the query would then be piped to the search engine in two ways. The first was without the classifier and the second one with the classifier enabled. The results returned from the search engine, with and without the classifier, would then be presented side by side to the user where he/she could vote on which result was better, or vote for a tie between the results. To make it a blind test, the side by side showing of the results were randomized in each run so that the user had no knowledge on which side had used the classifier.

An email was sent out to a group of friends and colleagues asking them to take the blind test. The email had information about the pasta node, giving the users hints on what type of queries the search engine expected. The test was ran during a weekend in August 2010 and a total of 100 queries were collected as data along with which result the users had voted on for each query. Figure 9 shows the website of the blind test.


File Edit View History Bookmarks Tools Help
Wed Sep 7 17:18 slimmy
dev.core.expertmaker.com/blindtest.php
Tile based games | Using MySQL Full... | PHP/MYSQL/AJA... | PHP Example AJ... | Similarity of texts... | text - Can someo... | Naive Bayesian T... | monthly totals | Other Bookmarks
Left is better
Neither is better
Right is better
HOME
SHARE
SAY

results - I am NEUTRAL
Knowledge Grid | Web | Wiki
PASTA EXPLORER
Salmon Pasta Salad Ingredients, 8 ounces spiral pasta, cooked and drained, 2 cups... - I LIKE
Pasta salad with sun-dried tomatoes... Ingredients, 60g pasta, 3 Sun-dried tomatoes, 1 tbsp low fat... - I LIKE
Garden Pasta Salad with Basil Dressing... Ingredients:, 3/4 cup chopped green beans , 1/2 lb. pasta (bow... - I LIKE
Greek Pasta Salad Ingredients, 3 cups of orzo pasta , 1/4 cup olive oil , 3 Tablespoon... - I LIKE
Broccoli Pasta Salad Ingredients, 1 Envelope Good Season's Italian Salad Dressing... - I LIKE
Find a pasta recipe that suits you
pasta salad without dairy
I am
[✓ Neutral] At work
Free In motion
Search

results - I am NEUTRAL
Knowledge Grid | Web | Wiki
PASTA EXPLORER
Spaghetti alle Vongole Spaghetti alle vongole (spaghetti and clams) originated in Naples... - I LIKE
Warm tomato and ricotta pasta salad... A perfect picnic dish of a pasta salad topped with crunchy breadcrumbs... - I LIKE
Garden Pasta Salad with Basil Dressing... Ingredients:, 3/4 cup chopped green beans, 1/2 lb. pasta (bow... - I LIKE
Greek Pasta Salad Ingredients, 3 cups of orzo pasta, 1/4 cup olive oil, 3 Tablespoon... - I LIKE
Find a pasta recipe that suits you
pasta salad without dairy
I am
[✓ Neutral] At work
Free In motion
Search

© 2010-2011 ExpertMaker – Help | About | Legal

Figure 9: Screenshot of the blind test website

9 Results

9.1 Blind test

The following section will display and discuss the results related to the online blind test conducted.

9.1.1 Analyzing the data

There are four different cases defined when analyzing the data. These are explained with respect to the labels in the figures.

- **Raw:** The raw test data, nothing has been post-processed.
- **Nonsense:** Queries only containing nonsense and non-related queries has been removed, e.g blank queries or queries like *asdf*.
- **Spell:** Queries with misspelled keywords has been removed.
- **Nonsense and spell:** A intersection of the two cases above.

9.1.2 Results

Figures 10, 12, 11 and 13 shows the results of the blind test. The results can be divided in to two groups. One group where tie scores are taken into account and one where the tie scores have been excluded.

Figure 10 shows the results for the four different cases with tie scores taken into account. Already with the raw data, close to 50% of the users prefers the results given from the classifier. About 30% of the users prefers the standard system and the rest thinks it's a tie.

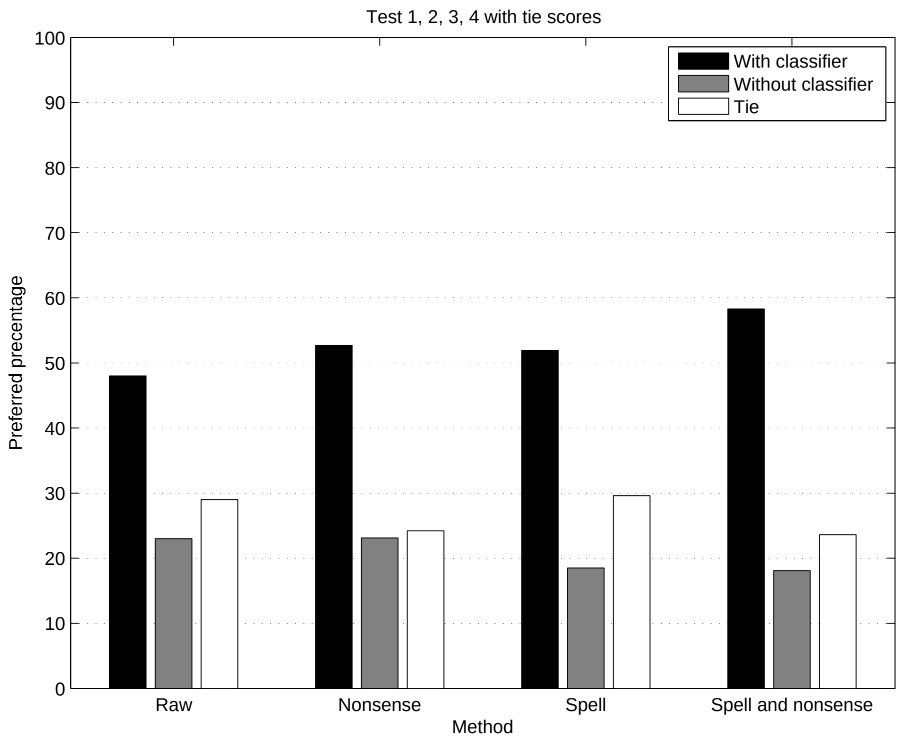


Figure 10: Results of test 1-4 with tie scores included.

Figure 11 shows the development of preferred percentage over the four cases taking tie scores into account. A clear trend is that when the search engine is supplied with valid queries that are spelled correctly, making it possible for the classifier to perform mapping, more users tend to prefer the results given by the classifier.

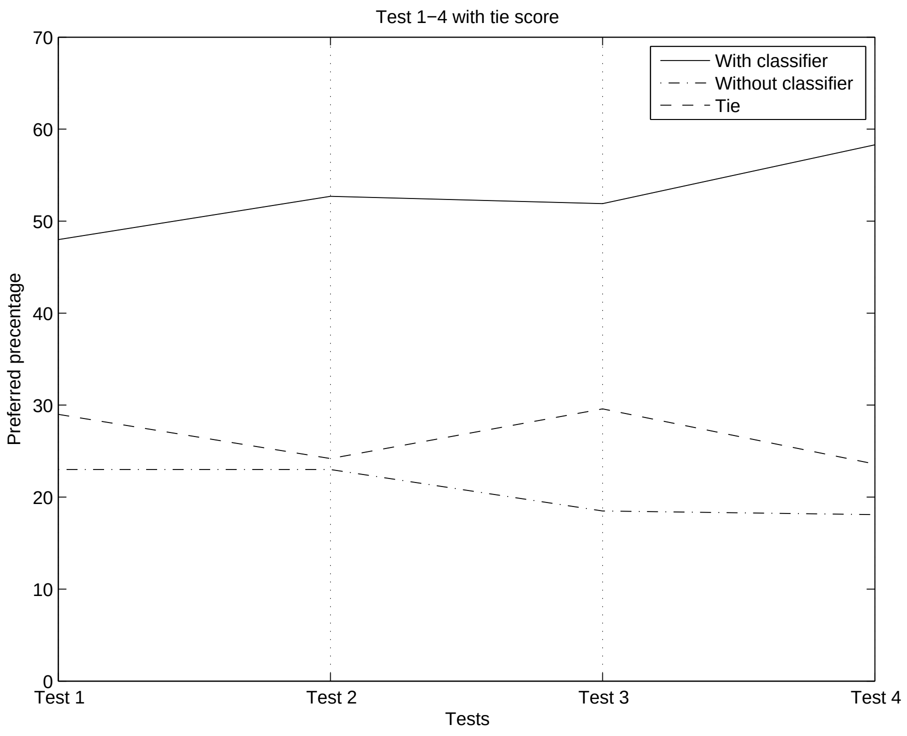


Figure 11: Development of results over the four tests. Tie scores included.

Figure 12 shows the results for the four different cases excluding the tie scores. Without the tie scores the contrasts between the results are clearer. When put head to head around 68% of the users prefers the classifier on the raw data and about 76% prefers the search engine using the classifier when the data has been rid of nonsense and spell checked.

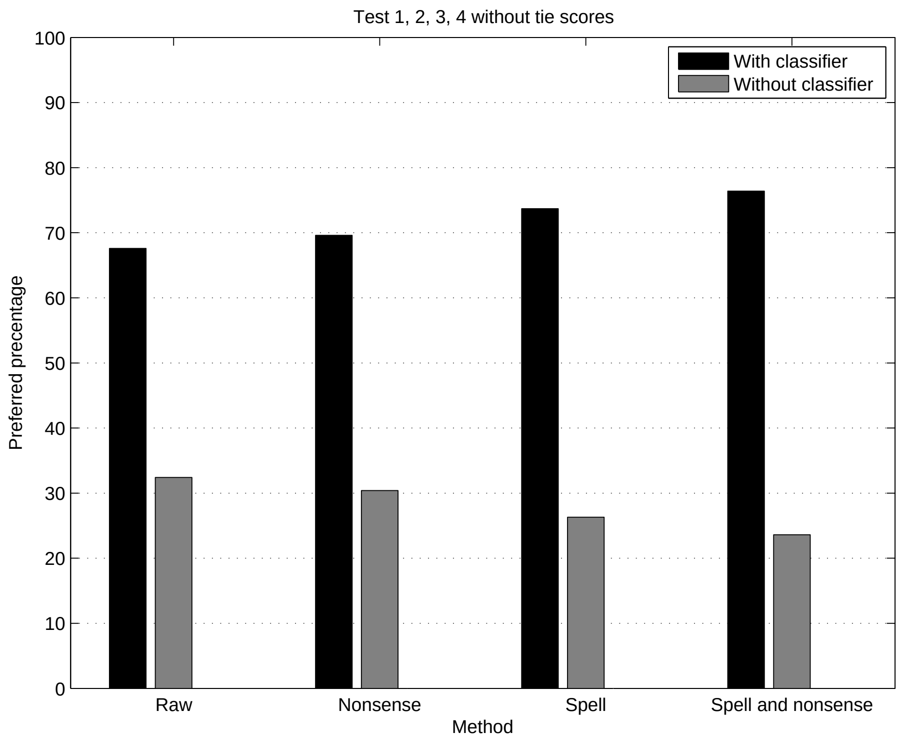


Figure 12: Results of test 1-4 with tie scores excluded.

Figure 13 shows the development of preferred percentage over the four cases excluding the tie scores and again the trend is clear that when mapping is possible the users tend to prefer the classifier.

Once the four cases had been tested and analyzed the 100 queries were manually annotated with the correct parameters that the classifier were expected to to map. If there was any vagueness as to which parameter that should be mapped given the query it was excluded from the statistics. The majority of the queries were single parameter mappings with some queries that had up to five parameters to be mapped. Among the 100 queries, a total of 141 parameters were set by the classifier with the following statistics.

1 out of 141, *.7%*, parameters were mapped wrongly.

7 out of 141, *4.9%*, parameters were not mapped when they were supposed to.

133 out of 141, *94.3%*, parameters were correctly mapped.

The single parameter that were wrongly mapped were because of the training data and not because of the classifier itself. The query was *pasta with no seafood*, which was interpreted as a user allergic to shellfish asking for a pasta dish. The *no shellfish* parameter had not been trained on the word *seafood* making it an

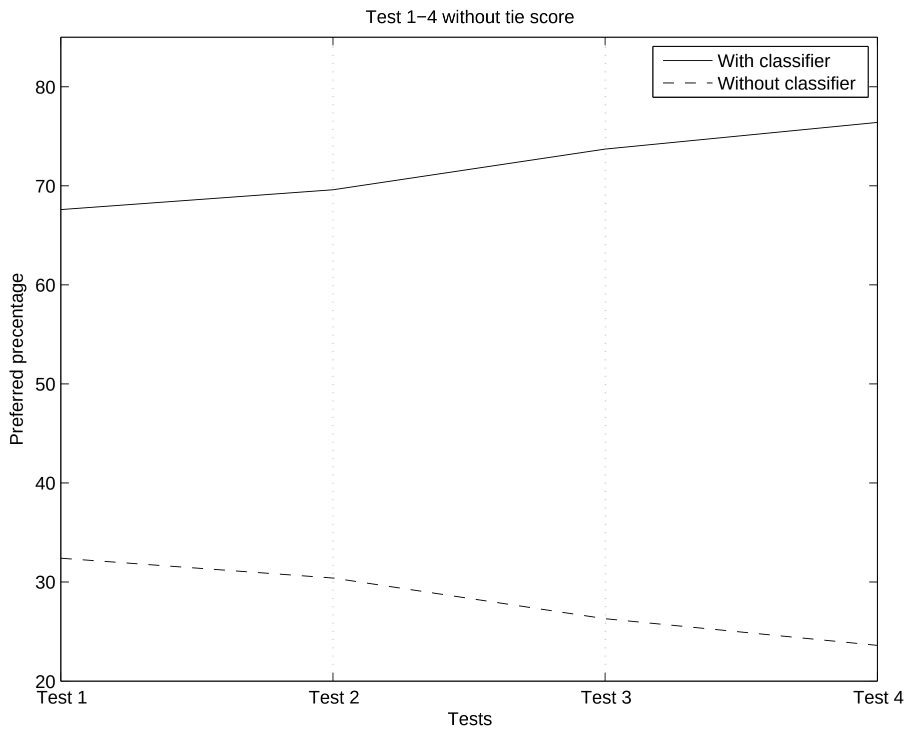


Figure 13: Development of results over the four tests. Tie scores excluded.

incorrect mapping according to the manually annotated data. On a side note it could also be interpreted as a user asking for any pasta dish that does not contain seafood in which case there should be no parameter mapping at all since the search engine cannot filter on exclusion of parameters only on inclusion of parameters.

The choice of removing misspelled queries instead of correcting them and add them to the data set is justified by the fact that there is no explicit *right* or *wrong* to the test itself. The test is only a reflection of what users perceive to be the correct answer. The result of a correctly mapped query was sometimes receiving a vote against the classifier and some missed mappings were sometimes receiving a vote favoring the classifier.

Take as an example the somewhat generic (to the pasta node) query *pasta salad*. I personally think that the "standard" pasta salad is a Caesar salad and if I'm presented with results that shows the Caesar salad at the top rather than some other salad I'm probably more inclined to vote for that result as opposed to the other result. However someone else might think that a Greek salad is the "standard" salad and would vote on that instead of the Caesar salad that I would vote on. By this reasoning, the misspelled queries were removed and not corrected because there is no way of knowing what the user would have voted on if the query was correctly spelled.

9.2 Discussion

At the start of the thesis, the initial problem was to investigate whether of not it was possible to perform automatic parameter mappings given a query. The data was *very* sparse and document classification on sparse data is challenging. However, the most reasonable approach to the problem was to try out document classification and see how it performed. Initial tests with a simple Bayesian implementation, not the linked Bayesian classifiers described in this report, did not perform well on the sparse data. Not only did the initial implementation not meet the real time requirements since it tried to classify each of the inputs, relevant to the query or not. It did also consistently perform erroneous mappings.

With the introduction of the linked Bayesian classifier and the ability to add synonyms to the training set, an increase in both speed and performance was observed. After some tweaking of the threshold values for classifying relevant inputs and the addition of synonyms the implementation was tried out in a blind test where up to 76% of the users preferred the search engine utilizing the classifier along with 94.3% of the parameters being mapped correctly which is a result that I am satisfied with.

10 Conclusions

To draw a simple conclusion about this thesis is to say it is possible to perform automatic parameter mappings given a query, at least to some extent. The fact that the simplest solution also proved to be the best performing one is perhaps not surprising given the initial conditions, but from a personal point of view it would have been interesting to try out a more "up to date" method of classification like *Random forests*.

There is however a lot of improvements that can be made with the current implementation. For starters the threshold that is set to determine relevant inputs could be made dynamic depending on the characteristics of the node. Another improvement would be to have the system automatically suggest the addition of synonyms based on user statistics, e.g if enough users have used a search term X and a certain percentage have set the parameter Y when using the term X the system should suggest an addition of the synonym X to the training set of Y. This could also work the other way when enough users change a mapped parameter, the system should suggest a removal of a synonym from the training set. These improvements are not realizable until enough user data has been gathered for a retraining of the system.

Another improvement the system would benefit from, that probably falls outside the area of this thesis, is spell correction. For the classifier to be able to perform mappings the words has to be spelled correctly. As an example, during the blind test the word *spaghetti* was misspelled in over 65% of the cases (with 4 different spellings, including the correct one).

As a last proposed improvement, maybe not the most critical one, would be to introduce semantics as a feature in the classifier. At the time of writing it is not hard to "trick" the classifier with double negatives etc, but this requires the user to actively try to make the classifier perform the wrong mappings.

.

A Initial test suite

This is the text file used in the initial test for all the implementations. Each line describes a query and the expected output from the classifier. Query and expected output are separated by \$. The output is in the form of

$$ix_m = y_n$$

where x_m is the mth input and y_n is the nth input value for that input. Multiple mappings are separated by &. The file contains the following lines.

```
spaghetti $ i0=0
pasta salad $ i0=1
lasagna $ i0=2
chicken $ i1=2
spicy $ i4=2
lasagna without nuts $ i0=2&i6=1
pasta salad with seafood $ i0=1&i1=3
advanced spaghetti dish $ i0=0&i2=2
quick lasagna dish $ i0=2&i3=0
pasta salad without shellfish $ i0=1&i7=1
fettuccine with chicken and without nuts $ i0=0&i1=2&i6=1
fettuccine with chicken and no nuts $ i0=0&i1=2&i6=1
quick pasta salad with meat $ i0=1&i1=1&i3=0
spicy lasagna without shellfish $ i0=2&i4=2&i7=1
vegetarian spaghetti without lactose $ i0=0&i1=0&i5=1
a quick lasagna with meat and no nuts $ i0=2&i1=1&i3=0&i6=1
mild seafood dish without dairy and no nuts $ i1=3&i4=0&i5=1&i6=1
mild seafood dish without dairy and without nuts $ i1=3&i4=0&i5=1&i6=1
chicken pasta salad with nuts and with shellfish $ i0=1&i1=2&i6=0&i7=0
quick and advanced spaghetti dish with meat that is mild without dairy, ...
... with nuts and no shellfish $ i0=0&i1=1&i2=2&i3=0&i4=0&i5=1&i6=0&i7=1
```

.

References

AB, E. (2011). Expertmaker.com.

Almonayyes, A. (2006). Multiple explanations driven Naive Bayes classifier. Technical report, Kuwait University.

Breiman, L. (2001). Random forests. *Machine Learning*, 45:5–32. 10.1023/A:1010933404324.

Buckley, C. (1985). Implementations of the SMART information retrieval system. Technical report, Cornell University.

Forman, G. and Cohen, I. (2004). Learning from little: Comparison of classifiers given little training. In Boulicaut, J.-F., Esposito, F., Giannotti, F., and Pedreschi, D., editors, *Knowledge Discovery in Databases: PKDD 2004*, volume 3202 of *Lecture Notes in Computer Science*, pages 161–172. Springer Berlin / Heidelberg.

Friedman, N., Geiger, D., and Goldszmidt, M. (1997). Bayesian network classifiers. *Machine Learning*, 29:131–163. 10.1023/A:1007465528199.

Fry, E. B. and Kress, J. E. (2006). *The Reading Teacher's Book Of Lists*. Lavoisier.

Hiemstra, D. (2000). A probabilistic justification for using tfidf term weighting in information retrieval. *International Journal on Digital Libraries*, 3:131–139. 10.1007/s007999900025.

Lewis, D. (1998). Naive (Bayes) at forty: The independence assumption in information retrieval. In Nédellec, C. and Rouveirol, C., editors, *Machine Learning: ECML-98*, volume 1398 of *Lecture Notes in Computer Science*, pages 4–15. Springer Berlin / Heidelberg. 10.1007/BFb0026666.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, 1 edition.

McCallum, A. and Nigam, K. (1998). A comparison of event models for Naive Bayes text classification. Technical report, Carnegie Mellon University.

nltk package (2011). nltk.org.

Rennie, J. D. M. (1999). Improving multi-class text classification with Naive Bayes. Technical report, Carnegie Mellon University.

Rish, I. (2001). An empirical study of the Naive Bayes classifier. Technical report, T.J. Watson Research Center.

Salton, G. (1989). *Automatic Text Processing*. Addison-Wesley.

Salton, G. and Mcgill, M. J. (1986). *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA.

Salton, G. and Yang, C. (1973). On the specification of term values in automatic indexing. Technical report, Cornell University.

Sparr, G. (1994). *Linjar Algebra*. Studentlitteratur.

Zhang, H. (2004). The optimality of Naive Bayes. Technical report, University of New Brunswick.