Problem/issue detection and classification

by

Frode Nilsen

Thesis submitted in Partial Fulfillment of the
Requirements for the Degree Master of Technology in
Information and Communication Technology

Faculty of Engineering and Science
University of Agder

Grimstad
May 25, 2009

Abstract

This thesis investigates the possibility of using pattern recognition and machine learning to detect online forum posts containing descriptions of problems and issues. In addition we seek to further classify these posts as either informative or non-informative, depending the quality of their content.

Our motivation for this research is the fact that more and more consumers are turning to the Internet when expressing opinions, seeking help or searching for advice about their purchases. As a producer, awareness of this online "word-of-mouth" has become a key factor in brand and reputation control and when dealing with product recall management. However, over the last years, the amount of such online buzz has transcended the point where it's manually manageable. It's simply not possible to survey thousands of online forums by hand anymore. We therefore need systems able to automatically monitor these web sites and detect posts of interest, such as people having problems with a product.

In the thesis, we implement and examine three different algorithms, Naive Bayes, k Nearest Neighbor and Self-Organizing Maps and determine what parameters and features produce the highest accuracy. The prototype is trained and validated using forum posts, but its application is not limited to this type of documents. The contribution of the thesis is to the areas of natural language text processing and classification.

Preface

This report is submitted in partial fullfillment of the requirements of the Master of Science degree in Information and Communication Technology at the University of Agder, Faculty of Engineering and Science. The project is supported by Integraso A/S, who have provided data material, supporting frameworks and guidance during the project period. The project's supervisors have been Ole-Christoffer Granmo from University of Agder and Aleksander M. Stensby and Jaran Nilsen from Integrasco.

I'd like to thank Ole-Christoffer, Aleksander and Jaran for their splendid supervision throughout the project. Your input and support have been invaluable.

Grimstad, May 25, 2009

Frode Nilsen

Contents

Contents **2**

List of Figures **4**

1 Introduction **5**
1.1 Goal and contribution . . . 7
1.2 Target audience . . . 8
1.3 Report outline . . . 8

2 Pattern recognition **10**
2.1 Sample data . . . 10
2.2 Feature selection and extraction . . . 11
2.2.1 N-grams . . . 12
2.2.2 TF-IDF . . . 13
2.3 Supervised vs unsupervised training . . . 14

3 Classification algorithms **15**
3.1 Naive Bayes . . . 15
3.1.1 Bayesian decision theory . . . 15
3.1.2 Naive Bayes classifier . . . 18
3.2 K-Nearest Neighbor . . . 19
3.2.1 Algorithm . . . 19
3.2.2 Distance calculation . . . 20
3.2.3 Selecting k . . . 20
3.3 Kohonen Self-Organizing Maps (MDS) . . . 22
3.3.1 Algorithm . . . 22
3.3.2 Color mapping example . . . 25

4 Design & implementation **27**
4.1 Sample data 27
4.1.1 Sample data bias 28
4.2 The prototype 30
4.2.1 Feature extraction (tokenization) 30
4.2.2 Naive Bayes 31
4.2.3 k Nearest Neighbor 34
4.2.4 Self-Organizing Maps (Kohonen/MDS) 35

5 Results & discussion **41**
5.1 Classifying problem / not problem 41
5.1.1 Naive Bayes 41
5.1.2 K Nearest Neighbor 45
5.1.3 Self-Organizing Map (Kohonen Network) 48
5.1.4 Comparison 53
5.2 Classifying based on informativeness 54

6 Conclusion & further work **56**
6.1 Conclusion 56
6.2 Further work 57

Bibliography **59**

List of Figures

1.1 Online WoM's impact on respondents' purchase decisions[10] . 6

2.1 Simple vector space model for two text documents 12

3.1 k Nearest Neighbor example with different k 20
3.2 k Nearest Neighbor example with too large k 21
3.3 SOM neighborhood radius (a) and factor (b) decay graphs . . 24
3.4 Self-Organizing color map (6 colors) 25
3.5 Self-Organizing color map (hundreds of colors) 25

4.1 Training set generation helper 28
4.2 Examples of pre-classified samples "anchored" in a SOM . . . 37

5.1 Naive Bayes accuracy with different training set sizes 42
5.2 Naive Bayes accuracy per iteration 43
5.3 Naive Bayes accuracy standard deviation with different training set sizes 44
5.4 Naive Bayes accuracy, bigrams vs words 45
5.5 kNN accuracy with different training set sizes and varying k . 46
5.6 Best recorded average accuracy for kNN 47
5.7 10x10 SOM, varying iterations, training set sizes and initial learning rate 49
5.8 10x10 SOM, anchored samples (learning rate 0.1 and 200 training samples) 50
5.9 SOM accuracy with varying number of anchor samples and k . 51
5.10 SOM accuracy from different map sizes 52
5.11 Result comparison Naive Bayes, kNN and SOM 53
5.12 Naive Bayes accuracy on informative/not informative 54
5.13 Naive Bayes accuracy deviation on informative/not informative 55

Chapter 1

Introduction

The ever-growing digital realms of the Internet have over the last decades inarguably had massive impact on how the majority of society go about their daily lives. Even within said decades, we have seen drastic evolutions in how the Internet is being used – and abused. Thanks to the increasing popularity of websites with consumer-generated media (CGM), the general public can quickly and with ease catch up on the latest buzz and express their views to thousands or even millions of others. These sites, such as blogs, discussion boards, user groups, online opinion/review services and a variety of other similar platforms, are in fact the current fastest growing segment of the Internet [9].

With consumers taking their conversations online, effectively measuring, interpreting and acting on this online *Word-Of-Mouth* (WoM) has shown to be a key factor in keeping a competitive advantage for consumer centered brands. Research shows that the Internet savvy consumer to a large extent trust and base his or her purchase decisions on these opinions and reviews. A survey by L. K. Slåttebrekk at BI Norwegian School of Management found that the majority of the respondents let online product and service reviews influence their purchase decisions (see figure 1.1) [10]. Furthermore, a similar study by BIGresearch has found that 17% of US adult Internet users regularly give advice and feedback about products and services online[3].

The challenge in monitoring online WoM is the sheer volume of data to be traversed when trying to find desired information. For a company

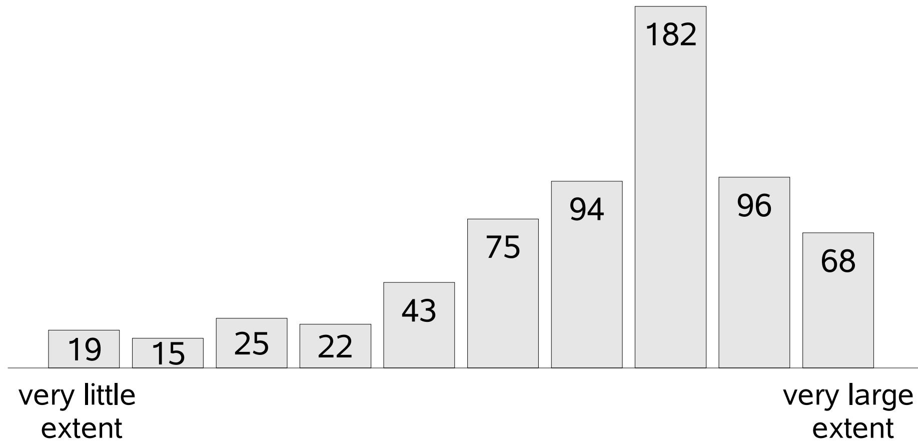


Figure 1.1: Online WoM's impact on respondents' purchase decisions[10]

attempting to learn their online reputation or the consumers opinions about their products, it can be a daunting task. In some cases, a search engine can surely be of great assistance – but as anyone who's been searching for a subject not clearly defined by a word or phrase can testify, even Google does not always return exactly what you intended it to.

Pattern recognition and classification is the act of taking raw data and categorizing it based on the "category" of the pattern [5]. As humans we do this continuously throughout our lives – for example, we can understand written symbols, recognize faces or identify an alarm clock at a nightstand by blindly fumbling around with a hand. Unfortunately, in many areas the amount or nature of the data make it unsuitable for a human to classify efficiently.

Luckily, in some fields, we can also make computers do pattern classification, albeit not with the sophistication of the human mind and senses. For instance, we have email software automatically detecting spam messages or touchscreen PDAs comprehending our handwriting. Newer consumer digital cameras can recognizes faces in the motive, and prototype cars will detect if you're about to doze off while driving.

What if we could also have a classifier help companies with their online reputation management, automatically detecting issues and problems consumers may have with their latest purchase? Their customers, searching for

information about their products, can be given more relevant results by prioritizing texts classified as informative problem descriptions. Storage space requirements can be reduced by discarding texts classified as poor descriptions.

The need for such a classifier is also apparent in the area of Product Recall Management (PRM). PRM is a combination of analysis techniques used to identify products that may contain serious errors and have to be recalled from the market. Obviously, recalling a product can have a substantial financial impact on the producer, and hence it is imperative to quickly detect problems and complaints.

Bug reporting systems for large projects is a similar application area where one might imagine such a classifier being used. An automated classifier could relieve the maintainers of the task of manually deleting non-descriptive bug reports.

Although a variety of subjects can be classified with generic classifiers, there has been no research into detecting issue/problem descriptions and their informativeness. Because the degree of which something is informative is difficult to formally measure, it's not given that a classifier will work as well as with, for example, language classification (where the difference between each class is more distinct). A system with this ability would be quite useful for companies where it is important to detect peoples complaints or issues among vast amount of text.

1.1 Goal and contribution

The following hypothesis is presented: We can use a system based on machine learning and pattern recognition to classify texts as either problem descriptions or not problem descriptions, and in addition classify them as informative problem descriptions or non-informative ones.

For the project we have decided to investigate and compare Naive Bayes, k-Nearest Neighbour and Kohonen Self Organizing Maps (a type of Multidimensional Scaling) on our classification problem. The research questions are as follows:

1. How well are Naive Bayes, Nearest Neighbour and Kohonen SOM able to classify arbitrary forum posts as problem descriptions and not problem descriptions?
2. Using the most efficient and accurate technique from 1., can we make a classifier prototype that is also capable of classifying problem descriptions as either informative or non-informative?

To answer these questions we will perform extensive tests with manually classified samples and measure the accuracy of each classification technique.

Thus, this thesis' contribution to knowledge will be how to make a classifier with the ability to recognize problem descriptions and classify them based on their informativeness. This is a new area of application for pattern recognition of text. In particular, we will determine which one of the three techniques, namely k-Nearest Neighbor, Naive Bayes and Self Organizing Maps (Multidimensional Scaling), are best suited for the task, i.e. classifying as either problem description or not problem description. Additionally, we will learn what amounts of training/pre-classified samples are required to yield a good result for this kind of classification, and what parameters suit the different classifiers best. Lastly, we will investigate to what degree using N-grams can deal with grammar errors in the text without deteriorating the classification itself.

1.2 Target audience

The target audience for this thesis is anyone with an interest in the domain of pattern recognition and classification of text, as well as mining and analysis of Word-of-Mouth on the Internet. The reader is assumed to have basic knowledge of probability theory, computer programming and the Internet in general. The essentials of the general methods and algorithms used in our work however, are introduced and explained in the report.

1.3 Report outline

The remainder of this project report is organized as follows:

Chapter 2 introduces the general concepts of pattern recognition with machine learning.

Chapter 3 explains the three main pattern recognition algorithms we are using: Naive Bayes, k Nearest Neighbor and Self-Organizing Maps.

Chapter 4 describes our implementations of the three algorithms and how we've prepared our experiments.

Chapter 5 presents the results of our experimentation and discusses the different findings.

Chapter 6 concludes our thesis by summarizing the outcomes and suggesting some ideas for further work.

Chapter 2

Pattern recognition

Numerous methods exist in the field of automated pattern recognizers. A trivial example could be a spam filter simply searching for words like "viagra" or "casino" in emails, or a single thermostat categorizing each sensor reading as "too hot" or "too cold". For more complex types of problems, more intricate methods are usually needed to enable a computer to accurately recognize patterns. *Machine learning* is "the concept of incorporating information from training samples in the design of a pattern recognizer" [5]. This mimics the way humans become such expert pattern recognizers – our experience and knowledge of the different patterns and the input from our senses allow us to quickly categorize with great accuracy. Creating a learning machine that can recognize patterns, then, involves using a set of sample data items and implementing a program that can observe features of such data items and compare them to the features of the previously observed samples. Because of this ability to recognize the patterns of different categories or classes, a pattern recognizer is often also called a *classifier*.

2.1 Sample data

When making a classifier, one of the first steps is always to gather sample data. Sample data is a set of data items of the same nature as the data items we would ultimately have our classifier to classify. For example, if we wanted to make a pattern recognition system able to classify emails as spam or not

spam, our sample data would be a collection of emails including both spam and not spam.

The sample data is divided into two subsets; one for training or learning the system and one for measuring its accuracy (often referred to as validation). The number of training samples required to train the classifier sufficiently varies with the type of data and pattern recognition system, and typically has to be determined empirically.

The items in the validation subset always have to be manually classified. This enables us to promptly determine how many items were classified correctly by the classifier, and thereby validating, or disproving, that it's able to do what we intended it to. Usually, the items in the training set also have to be manually classified, except for some types of classifiers (see section 2.3). In many areas, we can readily pull sample data from some source where it's already classified because of the nature of the source. For example, if we wanted a pattern recognizer to classify newspaper articles based on their language, we could simply pull a number of articles newspapers of different nationalities into different language categories. In other cases, the process is more time consuming and you have to go through each data item and classify it yourself. Consider for instance a classifier classifying photos based on the number of faces in the motive. It's unlikely that you'd find a data source containing photos categorized by this specific criteria, so you'd have to manually go through a collection of photos and categorize them yourself.

2.2 Feature selection and extraction

Once we've acquired a satisfactory set of sample data, the next step is to determine which *features* are most favorable for that type of data and the classification task at hand. In other words, transform the data items into a representation suitable for the learning algorithm and the classification. A sample's features are its individual measurable properties and characteristics [5]. We want to use the features that are most likely to differ between samples of the different classes. For instance, in character recognition on a touch screen PDA, features could include the horizontal and vertical profiles of characters, number of holes in their shape etc. In speech recognition, length of sounds, relative power or noise ratios are likely features.

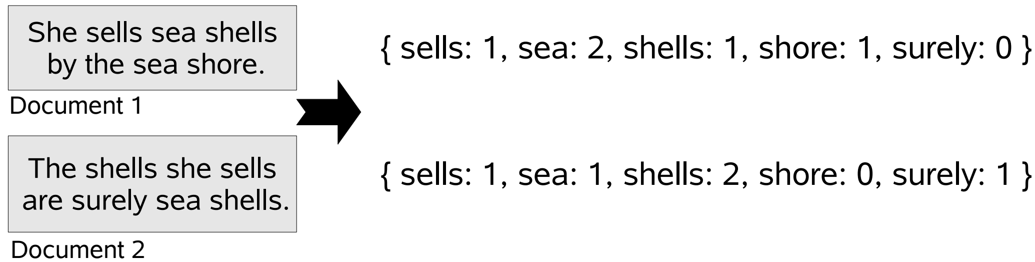


Figure 2.1: Simple vector space model for two text documents

In text classification, presence and frequencies of words or segments of words are often selected as features. Advanced classifiers may even incorporate analysis of grammar and semantics in their selection of features, but for many classification tasks the accuracy is well beyond sufficient even without it.

Observing the features of a data item provides us with a vector where the elements are the measured values of each feature. For example, if we use occurrences of words as our features and observe two text documents, the feature vectors could be extracted as shown in figure 2.1. Notice that common words and punctuation are stripped, and the text is transformed to lowercase. The *feature space* of this set of documents has a dimension of 5.

2.2.1 N-grams

Text from user generated content such as discussion board posts are usually quite prone to spelling mistakes and online lingo and slang. Obviously, this has the potential to confuse the classifier considerably, and ideally we would like a classifier to tolerate some level of noise. One technique for dealing with this problem is the use of N-grams. An N-gram is an N-character long slice of a longer string, with the N denoting the length of the slice. Splitting the entire data items into N-grams, we can use these as features instead of the actual words. Obviously, any spelling errors will still be present in the text, but the total percentage of misspelled features is will be less. As an example, the bigrams extracted from the phrase “don’t panic” would be *_d*, *do*, *on*, *n’*, *’t*, *t_*, *_p*, *pa*, *an*, *ni*, *ic* and *c_*.

In [4], Cavnar et al. apply this method to language classification. The idea is that some combinations of characters occur more often in some languages than other. Their classifier is tolerant of errors, and performs quite well (99.8%). They also try using the classifier for subject classification, with an accuracy of 80%.

2.2.2 TF-IDF

Term Frequency - Inverse Document Frequency (TF-IDF) is a technique used to weight elements of the feature vectors from a collection of data items (often text documents). The weight represents the term's importance to a data item in the collection, and increases proportionally to the number of times the term occurs in the data item, offset by its frequency in the whole collection [7].

Mathematically, we can express TF-IDF for term i in document j with the following fairly straightforward formulas. We let n_{ij} denote the number of occurrences of term i in document j, and $\sum_k n_{k,j}$ the total number of terms in the same document. This gives us the term frequency (TF) for i in j.

$$\mathrm{tf_{i,j}} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

Secondly, we calculate the term's inverse document frequency (IDF). The total number of documents $|D|$ is divided it by the number of documents containing term, $|\{d : t_i \in d\}|$, and the IDF is the logarithm of that quotient. If the dividend is equal, or nearly equal, to the divisor, the IDF converges to 0. In other words, if a term occurs in nearly all the documents in the collection, its IDF value is very low.

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

The finished TF-IDF for term i in document j is simply TF multiplied by IDF. To summarize, high document frequency and low document frequency yields a high TF-IDF.

$$\mathrm{(tf\text{-}idf)_{i,j}} = \mathrm{tf_{i,j}} \times \mathrm{idf_i}$$

We use TF-IDF instead of the document vectors (shown in figure 2.1) to

emphasize important features to a pattern recognizer.

2.3 Supervised vs unsupervised training

In general, we have two types of classifiers based on learning machines: supervised and unsupervised. A fully unsupervised learning system doesn't need pre-classified training samples. In return it's only able to infer the organization or similarity of data, not actually tell you the name of the class to which a sample belong. We can compare this to the way a child can organize LEGO bricks based on their color or shape, but not actually tell you the color or shape's name. Self-organizing maps are examples of a classifier that uses unsupervised learning.

Supervised learning, on the other hand, requires that we during the training phase inform the system about the class or category of each sample. For each sample learned, the system amends its knowledge about that sample's class. If we again compare this to a child and LEGO bricks, it's equivalent of giving the child several bricks, one by one, and saying "This is a red brick", "This is a blue brick" etc. for each one. After a certain amount of samples, a knowledge about each class and its features is built up, and the classifier (in this case the child) is itself able to tell you which class a sample belongs to. Classifiers using K-Nearest Neighbor and Naive Bayes are based on supervised learning.

If we include some aspects of supervised learning into the unsupervised system, we can still get a classifier. For example, after the unsupervised system has determined the organization of data, we can tell it the name of the major segments. For any future samples, the system can determine which major segment is closest to the sample, and give the sample that segment's class name.

Chapter 3

Classification algorithms

As previously mentioned, a variety of different methods for machine learning and pattern recognition exist. In this chapter, we introduce the three we concern ourselves with in this project, namely K-Nearest Neighbor, Naive Bayes and Self-Organizing Maps (Multidimensional Scaling). In general, we can say that each system is comprised of two separate stages: the training stage and the validation stage.

3.1 Naive Bayes

The family of Bayesian classifiers and Bayesian decision theory are rudimentary probabilistic approaches to pattern recognition, in which evidence or observations are used to infer the probability that a hypothesis is true [5]. The phrase Bayes, or Bayesian, comes from the frequent use of Baye's theorem (sometimes called Bayes's Law), derived from the work of reverend Thomas Bayes (1702-1761) [6].

3.1.1 Bayesian decision theory

Bayes theorem relates the conditional and "prior" (marginal) probabilities of two random events. The theorem has proved useful in many modern applications, and can be explained by a discussion of playing cards. We

let $P(H_1)$ and $P(H_2)$ be the probability of drawing a red and black card, respectively. Since there are only red and black cards in a deck of playing cards, the probability of either occurring is exactly 1.

Next, let P(O) be the probability of drawing a face card (a Jack, Queen or King). The joint probability $P(H_1O)$, i.e. the probability of drawing a red face card, can then be found with the following trivial probability calculation.

$$P(H_1O) = P(O|H_1)P(H_1) = \left(\frac{6}{26}\right)\left(\frac{26}{52}\right) = \frac{3}{26}$$

Or, put differently:

$$P(H_1O) = P(H_1|O)P(O) = \left(\frac{6}{12}\right)\left(\frac{12}{52}\right) = \frac{3}{26}$$

Combining these two, we get the equation:

$$P(H_1|O) = \frac{P(O|H_1)P(H_1)}{P(O)}$$

That is, the probability that the card is red, given that it is a face. Similarly for black face cards:

$$P(H_2|O) = \frac{P(O|H_2)P(H_2)}{P(O)}$$

For an experiment with n mutually exclusive outcomes (as is typically the case with text classification), the denominator can be shown to equal 1. Furthermore, even if they aren't mutually exclusive, it stays constant for each outcome, and can hence be left out from each calculation [8].

In Thomas Baye's own words: "If there be two subsequent events, the probability of the second b/n and the probability of both together P/N and it being first discovered that the second event has happened, from hence I guess that the first event has also happened, the probability I am in the right is P/b." [8]

Now, it's somewhat unlikely that any card player immediately would find this very interesting. He already knows this probability is $\frac{6}{12}$, since there are 12 face cards in the deck, and half of them are of each colour. We say that

the *parameters are fixed*, and that the *data is random* (i.e. which card is drawn) [6].

Let's turn these assumptions around, and consider the parameters to be random, coming from a distribution of values, and the data to be known. Say we're out fishing. In advance, the local (very mathematically inclined) fishermen teach us that, at this time of year, the probability of getting a cod is $P(H_1) = 30\%$, and for trout $P(H_2) = 20\%$. We call this our prior knowledge, or *a priori probability*.

Similarly, the fishermen have taught us a characteristic, or a *feature*, the two kinds of fish have: A cod usually has a distinct chin barbell, while the trout does not. In fact, the fishermen have found that, 90% of all cod fish have visible chin barbell, but only 5% of trouts do. We call this the *class-conditional probability* for that feature. Because the fishermen have been fishing all their lives, they've deducted these probabilities from the *known data* (i.e. the fish they've caught).

We're in luck, and get a bite on the first throw. With the fish in our hands, we observe that it has a distinct chin barbell clearly visible. We use our newly learned information to determine what our catch is. How likely is it that it is a cod, based on the size of the chin barbell and the fact that it allegedly is 30% likely to catch a cod? How likely is it that it is a trout based on the same criteria? The priori and class-conditional probabilities for the two fish inserted into Baye's equation gives us:

$$P(cod|barbell) = P(barbell|cod) * P(cod) = 90\% * 30\% = 27\%$$

$$P(trout|barbell) = P(barbell|trout) * P(trout) = 5\% * 20\% = 1\%$$

Notice that, as mentioned earlier, the right hand side denominator ($P(barbell)$) has been excluded from the calculations, because it is simply a constant in both.

The equation gives us the *posteriori probability* for each fish. We choose the fish category that gives the highest value. We have now classified our fish using a Bayesian decision.

3.1.2 Naive Bayes classifier

A Naive Bayes classifier is a classifier that adopts Bayes decision theory to decide which category a data item belongs to[6]. As in the fishing example in the previous section, the decisions are probabilistic – or more specifically based on *maximum a posteriori*. The data is fed through the classifier, just as our catch was evaluated in our imaginary fish classifier, which produces a posterior probability for each category. Finally, the category with highest probability is selected for that particular data item. Expressed as a mathematical formula, this becomes:

$$h_{NB} = \underset{h_j \in H}{\arg\max}\, P(h_j) \prod_i P(o_i|h_j)$$

In the above formula, you may notice that the prior probability is multiplied with the product of many state-conditional probabilities. In the fishing example, we had only one state-conditional probability to multiply with – because we only had one feature. In more realistic applications, there will be several, perhaps thousands of features to incorporate. For instance, in a text classifier, a typical feature set is simply the frequency of the words occurring in the posts. How the state-conditional probabilities for the words can be calculated is described in section 4.2.2.

This multiplication of each feature's state-conditional probability means that we have made the assumption that:

$$P(o_1, o_2, \ldots o_n|h_j) = \prod_i P(o_i|h_j)$$

This is called the Naive Bayes assumption and implies that $o_1, o_2, \ldots o_n$ are conditionally independent. Very often this is not the case at all! For example, in the case of a text classifier, with each word being a feature, the contexts of the words are lost when they're treated as separate entities. This fallacious assumption is where the Naive Bayes classifier gets the "naive" part of its name.

Nevertheless, even in applications such as text classifiers, the Naive Bayes classifier proves to work surprisingly well, regardless of its naivety. In fact, research shows that in many cases, it performs comparably to many signif-

icantly more complex techniques [11]. Its high performance to complexity ratio has been an important factor in the Naive Bayes classifier's success.

3.2 K-Nearest Neighbor

K-Nearest Neighbor is perhaps the most elementary algorithm for machine learning and pattern recognition. The learning stage of the algorithm is often called *lazy learning*, because no actual processing of the training samples is performed during the training stage. One might even say that the algorithm has no distinct training stage, since all we do is assign it a set of pre-classified samples.

3.2.1 Algorithm

To classify a data item X, the algorithm first takes the X's feature vector and calculates the distance from it to every feature vector in the training set. Recall from section 2.2 that the feature vectors can be term frequencies, for example by words or ngrams for text classification, or TF-IDF vectors. The distance between X's feature vector and that of a training set sample represents the similarity between those two data items. The distance can be found in a number of different ways, for example using Euclidean Distance, Manhattan Distance or cosine similarity.

Regardless of the method used, when the distances to each vector in the training set have been found, the training set samples are sorted in ascending order based on their vector's distance to X. The k topmost samples, i.e. the k *nearest neighbors*, determine the classification of X by majority vote. If the majority of the k nearest neighbors are of class C, then X is classified as C and so on.

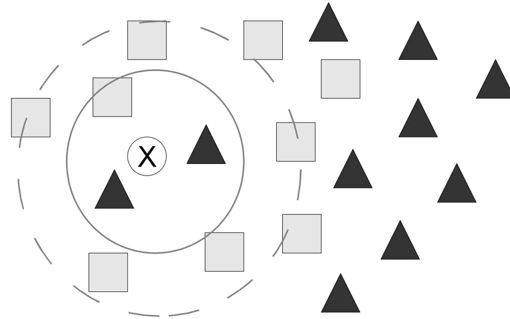


Figure 3.1: k Nearest Neighbor example with different k

3.2.2 Distance calculation

The Euclidean Distance between two vectors $P = (p_1, p_2, \ldots, p_n)$ and $Q = (q_1, q_2, \ldots, q_n)$ is calculated using the following formula.

$$\mathrm{d}_{euclidean} = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2}$$

A different method of calculating the distance is Manhattan Distance. The Manhattan Distance between P and Q from above is calculated as follows:

$$\mathrm{d}_{manhattan} = |p_1 - q_1| + |p_2 - q_2| + \ldots + |p_n - q_n|$$

Similarly, we can also use cosine similarity, in which case the "distance" is the angle between the vectors. A cosine of -1 means the vectors are exactly opposite, 1 means exactly the same and 0 indicates that they are independent (orthogonal). The cosine similarity between our P and Q is found with the following formula.

$$\mathrm{d}_{cosim} = \cos\theta = \frac{P \cdot Q}{|P||Q|} = \frac{p_1 * q_1 + p_2 * q_2 + \ldots + p_n * q_n}{\sqrt{p_1^2 + p_2^2 + \ldots + p_n^2} * \sqrt{q_1^2 + q_2^2 + \ldots + q_n^2}}$$

3.2.3 Selecting k

As with most parameters for a learning system, k usually has to be decided empirically. This means trying a number of different values and seeing which one yields the best results for that particular data and classification task. In

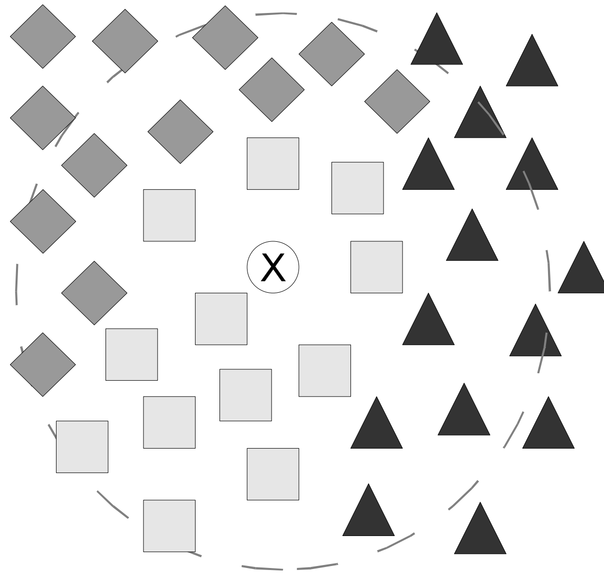


Figure 3.2: k Nearest Neighbor example with too large k

general, it's a good idea to choose a k not divisible by the number of classes, as this could lead to situations where the majority vote ends with a tie. If we choose $k = 1$, the algorithm is simply called Nearest Neighbor.

Generally, we can say that increasing k reduces the effect of irregularities among the neighbors. For example, consider samples in a 2D plane as shown in figure 3.1, and the unlabeled sample X. Using a k-value of 3 (indicated by the solid gray circle) would yield the class "triangle" because of the 2 triangles that for some reason are located among the squares. However, if we increase k and consider a larger neighborhood, for example $k \approx 6$ indicated by the dashed gray circle, it becomes obvious that the X is in a neighborhood dominated by squares. On the other hand, too large k-values make the boundaries between the different neighborhoods more vague, as seen in figure 3.2.

3.3 Kohonen Self-Organizing Maps (MDS)

The examples in figure 3.1 and 3.2 show samples with a feature space of 2, meaning that we can visualize their distribution in a 2D plane. With text classification, the dimensionality of the feature space may well exceed hundreds or thousands, because each distinct term adds one dimension (see figure 2.1). This means that finding the most similar training samples becomes quite computationally intensive. It also means that we can no longer directly visualize the samples and their proximities in a nice 2D plane.

Multidimensional Scaling (MDS) is a set of techniques used to represent multidimensional data in a much lower dimensional space, for example a 2D plane. Self-Organizing Maps (SOM) is one such technique, invented by Finnish professor Teuvo Kohonen and therefore often called "Kohonen networks". When fed with a training set of multidimensional vectors, Kohonen's technique creates a map where the topological relationships (distances) between the training vectors are represented in an arbitrary dimensionality. If we for instance use the feature vectors in a text classification task, it's impossible for us to see their proximities, e.g. similarity, in a 2D map. That would also enable us to more quickly find the k Nearest Neighbors by measuring the distance between coordinates in the map.

One of the most interesting aspects of Self-Organizing Maps is that the learning process is unsupervised. You feed the algorithm a set of vectors, and it figures out their organization and similarities all by itself. Still, as you may recall from section 2.3, you will need labeled samples if you wish the classifier to tell you the exact classification of a data item.

3.3.1 Algorithm

For this project, we only consider multidimensional scaling down to 2D, but SOMs can in theory be of any dimensionality. Before the training can commence we create a x by y matrix of what is called *processing units*. A processing unit is a vector of the same dimension as the vectors in the training set. Then, the learning algorithm proceeds with the following steps.

1. Initialize each processing unit's vector. Every element, or "weight", is

set to a random value. The value is typically within the range of values found in the training set's vectors.

2. Choose a vector at random from the training set, an *input vector*.
3. Calculate the distance (similarity) to each processing unit (as in section 3.2.2). The closest unit is called the Best Matching Unit (BMU).
4. Determine the BMU's neighborhood. The neighborhood is the surrounding processing units, up to a certain radius. The radius usually starts out large, perhaps even spanning the whole matrix, but is reduced over time.
5. Adjust each neighboring unit's weight vector to make it more like the input vector. The magnitude of the adjustment depends on how close it is to the BMU, and how early in the training we are.
6. Repeat from step 2 for a pre-defined number of iterations.

Adjusting the neighbors

Determining the BMU's neighbors and their proper adjustments can be done using the following single formula. Each processing unit's weight vector is transformed by this, and while it may look cryptic at first, it can be explained in fairly simple terms.

$$\mathrm{W}(t+1) = \mathrm{W}(t) + \theta(d,t)\mathrm{L}(t)(\mathrm{D}(t) - \mathrm{W}(t)) \tag{3.1}$$

The essence is that a processing unit's new weight vector, $W(t+1)$, should equal its old weight vector, $W(t)$, plus some adjustment. The added adjustment is a fraction of the difference between the input vector and the old weight vector, $D(t) - W(t)$. The size of this fraction is determined by the two functions $\theta(d,t)$ (the neighborhood factor) and $\mathrm{L}(t)$ (the learning rate factor).

The neighborhood factor ensures two things. Firstly, only processing units within the neighborhood should be adjusted, and as we progress through the training the neighborhood should decrease. We can use an exponential

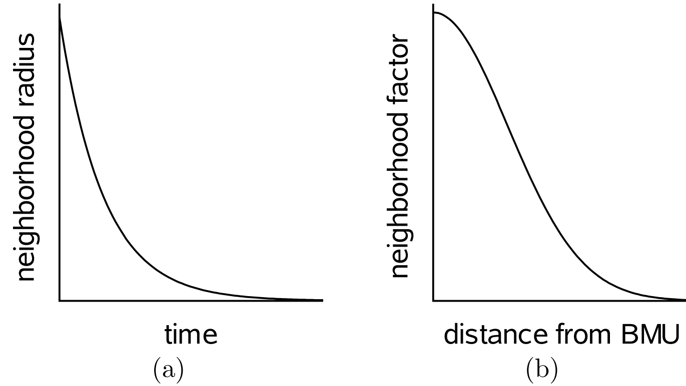


Figure 3.3: SOM neighborhood radius (a) and factor (b) decay graphs

decay function to calculate the neighborhood radius, for instance:

$$\sigma(t) = \sigma_0 * \exp\left(-\frac{t}{\lambda / \log(\sigma_0)}\right) \qquad t = 1, 2, 3 \ldots \tag{3.2}$$

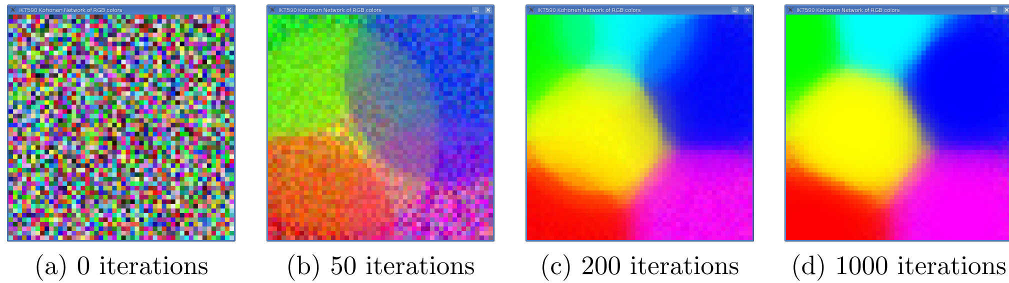
In this formula σ_0 is the pre-defined initial neighborhood radius for our SOM and λ is the total number of iterations. This gives us a nice decaying neighborhood radius over time, as shown in figure 3.3a.

Secondly, the neighborhood factor should ensure that processing units close to the BMU should get a large adjustment compared to units in the outskirts of the neighborhood. We use a Gaussian decay function incorporating the current iteration's radius from equation 3.2 and the input vector's distance d from the BMU:

$$\theta(d, t) = \exp\left(-\frac{d^2}{2\,\sigma^2(t)}\right) \qquad t = 1, 2, 3 \ldots \tag{3.3}$$

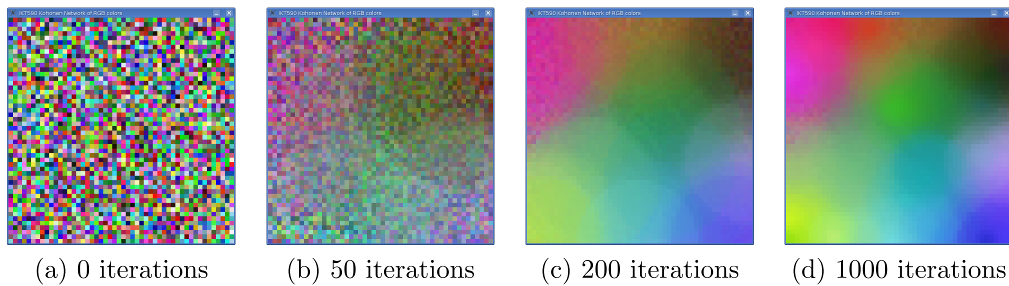
This is the neighborhood factor, or neighborhood function, $\theta(d, t)$ found in equation 3.1.

The second factor determining the adjustment in equation 3.1 is the learning rate factor $\mathrm{L}(t)$. The learning rate is a small pre-defined variable which also decreases with time. Its decay is calculated similarly to the neighbor-



(a) 0 iterations (b) 50 iterations (c) 200 iterations (d) 1000 iterations

Figure 3.4: Self-Organizing color map (6 colors)



(a) 0 iterations (b) 50 iterations (c) 200 iterations (d) 1000 iterations

Figure 3.5: Self-Organizing color map (hundreds of colors)

hood radius decay:

$$\mathrm{L}(t) = \mathrm{L}_0 \exp\left(-\frac{t}{\lambda / \log(\sigma_0)}\right) \qquad t = 1, 2, 3 \ldots \tag{3.4}$$

Where L_0 is the initial pre-defined learning rate, and as before, σ_0 is the pre-defined initial neighborhood radius and λ is the total number of iterations.

3.3.2 Color mapping example

One typical introductory example of an SOM is mapping of colors. A color is represented by a three dimensional vector, $(red, green, blue)$ where each element has a value within the range $[0, 255]$. The network is initialized with a completely random color in each processing unit, yielding a map that can be visualized as in figure 3.4a.

Secondly, we generate a training set – for this example, we use the colors

red, green, blue, light blue, pink and yellow. The number of iterations, learning rate and other parameters aren't very important here, so we just try with a learning rate of 0.1 and about 1000 iterations. After 50 iterations of fetching an input vector randomly from the training set and feeding it to the algorithm, the map is transformed into the one shown in figure 3.4b. It's evident that the BMUs for the input vectors have been found, and them and their neighbors have been adjusted slightly to more closely resemble the input vector.

After 150 iterations, the different colors have already gotten fairly distinct borders (figure 3.4c). If watched in real-time, it's easy to see how the amount of adjustment for each iteration is reduced. The last 500 iterations or so only yield minor corrections that are hardly noticeable. This is of course because both the learning rate and the neighborhood radius is lowered for each iteration. When the training is complete at 1000 iterations, the map looks as in figure 3.4d.

Of course, if we wanted, we could use all colors, not just those six. That would give us a much smoother map, such as in figure 3.5, with no distinct borders between a few different colors.

Chapter 4

Design & implementation

Now that we've described the background behind our area of pattern recognition and machine learning, we'll go more into detail on how we implement the different techniques for our experiments. In practice, the techniques are applicable to most areas of pattern recognition, but here we'll outline how we use them in text classification.

4.1 Sample data

Two sets of sample data are needed for our classification tasks.

Problem descriptions / not problem descriptions This set will be used for the first classification task, which is testing which of our three classification methods are best able to recognize problem descriptions.

Good / bad problem descriptions This set is for the second classification task, where we will test whether or not the best classifier is also able to tell the difference between good and bad problem descriptions.

We mentioned in section 2.1 that for some classification tasks, pre-classified samples are already available. We're not that fortunate, so we'll have to find and label the samples ourselves. We use Integrasco's database of forum posts as source for our samples. The database contains millions of entries gathered from a vast variety of Internet forums over several years.

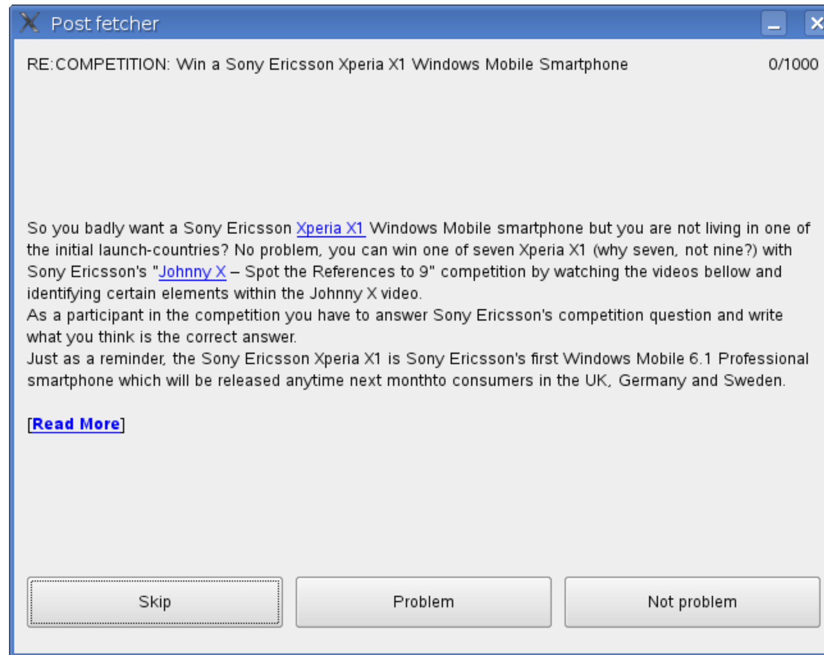


Figure 4.1: Training set generation helper

Generating the sample data is essentially manual work, i.e. basically a matter of fetching a batch of discussion board posts and classifying each and every one by hand – a very tedious task. A simple tool, depicted in figure 4.1, was created to make the work as quick and effective as possible.

For both our sets of sample data, we use a 50-50 ratio between the classes. This means we'll get a 50% accuracy from just guessing the classification. For Naive Bayes, it also means that the prior knowledge (a priori) will be 50% for each class. This does not reflect the real distribution in Integrasco's database, where, for instance, non-problem posts outnumber problem posts by hundreds. Thus, if we were to use our classifier on posts from outside our sample sets, accuracy is likely to be reduced.

4.1.1 Sample data bias

To ease the task of finding forum posts that are problem descriptions, we make use of Integrasco's search API. This allows us to find posts containing

typical problem related terms, such as “problem”, “trouble”, “fail”, “freez”, “doesn’t work”, “wont work”.

However, using search phrases to find samples of a specific classification introduces a particular obstacle. By forcing our problem description samples to contain certain words, they become less representative of real world problem descriptions. After all, a problem description is in real life certainly not required to contain any of the aforementioned terms. We say that the samples are *biased*, reflecting our particular methods and opinions more than reality.

To avoid bias completely is essentially impossible when we’re making the sample sets manually. The samples will always be biased to some degree, at least when there’s no formally defined distinction between the different classes. For example, during the gathering and manual classification of samples, there were several occasions where we were simply unsure whether something should count as a problem description or not.

To deal with the problem of samples being biased by the search terms mentioned above, we have three possible solutions.

Ignore the problem Don’t alter the samples in any way to reduce bias. This will probably yield unrealistically good results, because the search terms will be very strong characteristics of a problem description (i.e. the classifier will learn that those words are almost exclusive to problem descriptions). We call our sample set using this strategy the **NS sample set** (None Stripped).

Remove search terms Erase all search terms from the samples. This is likely to have the opposite effect and produce too pessimistic results, because now we’re forcing the samples to *not* contain certain words – thereby possibly removing too much information. We call the sample set processed like this the **AS sample set** (All Stripped).

Remove conditionally If a sample contains *one* of the search terms remove it, otherwise remove nothing. We believe this provides the most realistic results, seeing as it is a compromise between removing all the forced terms and acknowledging the fact that some problem description may very well have contained these terms without being forced. This sample set we name the **CS sample set** (Conditionally Stripped).

Informativeness of samples

The training and validation samples for the second classification task (informative / not informative) are a set of 500 samples in total. Half the set is samples we consider informative problem descriptions, and the other half is "non-informative". There's an obvious issue with this arrangement, namely the fact that informativeness nearly impossible to measure in any formal way, and certainly not really with two discrete values. Rather, you would typically say that something is not very informative, a little informative, very informative and so on. Additionally, the informativeness is very much in the eye of the beholder – what's informative to me may not be informative to someone else.

Because of this, the samples are very difficult to objectively classify manually. In many cases the samples fall into "gray area" where it's hard to tell whether it's really informative or not. Furthermore, it could be informative within the context of it's original forum thread, but not as a standalone post. All in all, because the classes are so difficult to distinguish, we find it unreasonable to expect the classifier to perform very well based on this sample data.

4.2 The prototype

4.2.1 Feature extraction (tokenization)

For this project we're using two kinds of features, words and n-grams. The first step of our prototype takes care of extracting the features and generating feature vectors for each sample. Programmatically, we use a common interface for both the word tokenizer and n-gram tokenizer, so that we're able to easily swap between the two during the experiments.

During feature extraction, stop words[1] are stripped from the text. We also remove all punctuation, markup codes/characters and other meta data. Also, since we are working with discussion board posts, we make sure to remove quotations.

[1] A list of small and frequently occurring words like *and*, *or*, *in*, *if* [1]

Additionally, two of the classification algorithms want their feature vectors weighted with the TF-IDF scheme. This is generated subsequently to the tokenization, and is independent of whether we're extracting n-grams or entire words. We call this process of tokenization (and, if applicable TF-IDF generation) the *feature extraction stage*. When we generate TF-IDF for a verification sample (to be used in the classification/verification stage), we use the document frequencies from the training samples, and *not* from the verification samples.

4.2.2 Naive Bayes

The Naive Bayes classifier itself has the two obvious stages: the *training/learning stage* and the *classification/validation stage*. The sample data items, tokenized in the feature extraction stage, is passed to the classifier, which then proceeds with its two stages. There is no persistence of data here, meaning that each run of the program re-learns everything and then validates again. The training set is randomized and split into a subset of training samples and a subset used for validation. By re-doing the whole process from scratch like this, we can do many runs to rule out the possibility that one run was just a sporadic "best case". Of course, in a production system, the knowledge learned from a training set is persisted and reused later.

Training stage

The pseudo code for this stage is shown in listing 4.1. A subset of the sample data is fed to this routine, as described above.

Listing 4.1: Pseudo code for the Naive Bayes training stage

```
1  learn_samples(Samples, Categories) {
2
3    foreach(sample in Samples) {
4
5      sample.category.samples += sample
6
7      foreach(term in sample.terms) {
8        sample.category.termcount[term] += 1
```

```
9         total.termcount[term] += 1
10      }
11    }
12
13    foreach(category in Categories) {
14      category.priori = count(category.samples) / count(samples)
15
16      foreach(term in category.termcount) {
17        category.conditional_prob[term] =
18          category.termcount[term] / total.termcount[term]
19      }
20    }
21  }
```

In its bare essence, what this part of the algorithm does, is to calculate the prior probability for each category, and the conditional probability for each feature in each category.

Note that in line 5 and 8 we refer to “a sample’s category” (`sample.category`). While this may seem counter intuitive (the sample already knows its category!), remember that these are pre-categorized samples used to train the classifier.

Verification stage

The verification stage of the program is the part that actually tells us if our classifier is working as it should. It uses the remainder of the sample data after the training component has done its work, and is shown as pseudo code in listing 4.2.

Listing 4.2: Pseudo code for the Naive Bayes verification stage

```
1  classify_samples(Samples, Categories) {
2
3    max_posterior = 0
4    best_category = undefined
5    correct_count = 0, success_rate  = 0
6
7    foreach(sample in Samples) {
```

```
8
9       foreach(category in Categories) {
10        posterior = calculate_posterior(sample, category)
11        if(posterior > max_posterior) {
12          best_category = category
13          max_posterior = posterior
14        }
15      }
16
17      if(best_category == sample.category) correct_count++
18    }
19    success_rate = correct_count / count(samples)
20  }
```

You may have noticed that perhaps the most interesting part of this component, namely the calculation of the posterior probability for each category and sample, is abstracted away with a function call `calculate_posterior` in line 10. It will be explained next.

Recall from section 3.1.1 that we use Bayes theorem to calculate the posterior probability. When we have our prior probabilities and the set of conditional probabilities for each feature in each category, it can be translated into the following formula:

$$P(category|W) = P(category) \prod_{w_i \epsilon W} P(w_i|category) \tag{4.1}$$

In the formula, W is the set of term frequencies of each word in the text to be classified that also appears in the term count from the training component. What this means is that we take the prior probability for the category and multiply it with the conditional probability for each term in that category found in the text.

When this is done for each category, one of them has been found to have the highest posterior probability for that text. The text is then classified as belonging to that category. Finally, the determined category is compared to the actual category of the sample. Again, since we are using the training set,

the true category of each sample is known. If the categories are one and the same, we increase a counter telling us how many successful categorizations we have made. When each verification sample has been attempted classified, we can then calculate an accuracy.

4.2.3 k Nearest Neighbor

As expressed in 3.2, k Nearest Neighbor (kNN) is one of the simplest algorithms for pattern recognition with machine learning. As with Naive Bayes, a subset of the sample data is fed to the training stage, and another to the verification.

For kNN the samples should have TF-IDF weighted vectors. This is taken care of in the feature extraction stage, so the training stage of kNN is therefore concerned solely with storing the sample documents in a list for future use by the classification/verification stage.

The classification/verification stage is naturally a little more intricate, but compared to Naive Bayes and SOM, even this is fairly straightforward. Its essence is shown in listing 4.3.

Classification/verification

Listing 4.3: Pseudo code for the kNN verification stage

```
1  knn_classify(Samples, TrainingSamples, k) {
2
3    correct_count = 0, success_rate  = 0
4
5    foreach(sample in Samples) {
6      distances = []
7      foreach(trainingSample in TrainingSamples) {
8        distances[trainingSample] =
9          euclideanDistance(sample, trainingSample)
10     }
11     orderAscending(distances)
12     neighbors = distances.subset(0,k)
13
```

```
14      if(sample.realCategory == majorityCategory(neighbors)) {
15        correct_count++
16      }
17    }
18
19    success_rate = correct_count / count(samples)
20
21  }
```

We find the k nearest neighbors in line 11 and 12 by ordering the list of training samples based on their distance in ascending order and then extracting the top *k* elements of the list. Determining the actual classification is done in line 14, where the call to `majorityCategory` determines the class that has majority within the neighbors.

4.2.4 Self-Organizing Maps (Kohonen/MDS)

Self-Organizing Maps is a significantly more complex algorithm than both Naive Bayes and kNN. We've implemented our classifier in three stages: the training stage, a visualization/anchoring stage and the classification/verification stage.

Training stage

The training stage is where the Self-Organizing Map "self-organizes" based on our training set with TF-IDF vectors. This is done using the 6 steps from section 3.3.1, and outlined in pseudo code in listing 4.4.

Listing 4.4: Learning stage of the SOM implementation (pseudo code)

```
1  learn(Samples, MapNodes, iterations) {
2
3    for(iteration = 0 to iterations) {
4
5      sample = getRandomSample(Samples)
6      bmu = undefined, bmuDistance = inf
7
8      for(mapNode in MapNodes) {
9        distance = euclideanDistance(sample, mapNode)
```

```
10        if(distance < bmuDistance) {
11          bmu = mapNode
12          bmuDistance = distance
13        }
14      }
15
16      for(mapNode in MapNodes) {
17        bmuDistance = euclideanDistance(bmu, mapNode)
18        alterMapNode(mapNode, sample,
19            getNeighborhoodFactor(iteration, bmuDistance),
20            getLearningRateFactor(iteration)
21        )
22      }
23    }
24  }
```

This stage alone has a number of different parameters we can adjust to better suit the algorithm to our classification task. The first are the map's dimension and size. Like we have mentioned before, we'll only consider 2D maps in this project. The map size, on the other hand, is one of the parameters we'll investigate. We use square processing units, but other shapes such as hexagons are also normal.

Another parameter we'll test is the number of learning iterations. For each iteration, one sample is taken from the training set and incorporated into the map topology, this enhancing the segmentation of the map (as seeing in the color examples in figures 3.4 and 3.5).

A third parameter is the initial learning rate. The learning rate influences the magnitude of alteration done to each processing unit's weight vector, and decays exponentially. A large initial learning rate yields larger alterations to the weight vectors in the neighborhood, but the rate of decay is steeper than for lower values.

The initial neighborhood radius is a different parameter. In our implementation we use a radius of half the map's width. If one wanted, of course using a different function for the radius decay would be entirely possibly, but we'll limit ourselves to the ones described in section 3.3.1.

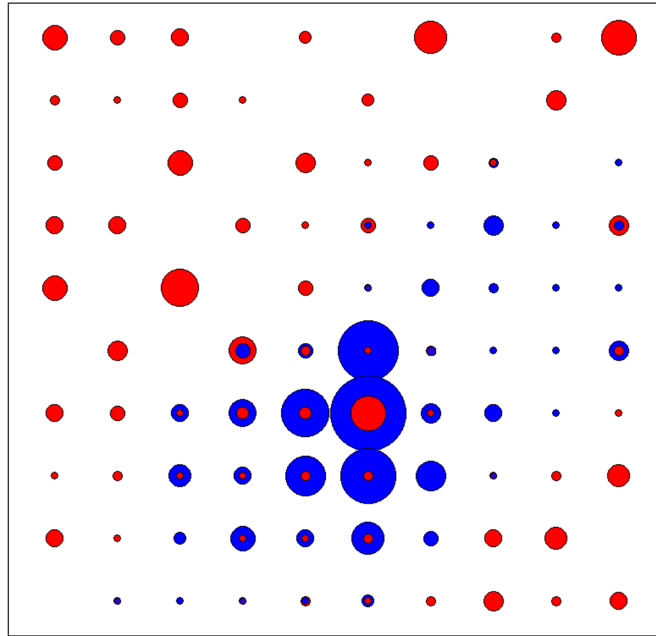


Figure 4.2: Examples of pre-classified samples “anchored” in a SOM

Visualization/anchoring stage

In theory, this stage isn’t really a part of Self-Organizing Maps as such. However, as we pointed out in section 2.3, we have to introduce some elements of supervised learning to the unsupervised SOM to actually be able to perform classification.

At this point we have our map self-organized and ready. We can take a document, generate a TF-IDF vector and then position it in the map, by calculating which processing unit is most similar. This is where we introduce the “supervised” portion of the training. We position several pre-classified training samples in the map, using different colors for each class. This stage we call the *anchoring stage*, because we consider these positioned samples as “anchors” to be used later in the actual classification stage.

In this stage we also take advantage of the fact that we have a “multidimensionally scaled” map, i.e. something we can render into a human readable figure. After having anchored a number of pre-classified samples, we may end up with a layout similar to the one shown in figure 4.2. As this

clearly illustrates, samples of the two different classes are clustered together (the radius of the circles indicate the number of samples anchored in that processing unit). We also note that even though we can clearly see clusters, some stray samples have been positioned in the wrong ones.

Moderately simplified, this stage can be described with the pseudo code in listing 4.5.

Listing 4.5: Anchoring stage of the SOM implementation (pseudo code)

```
1  anchor_samples(Samples, MapNodes) {
2
3    for(sample in Samples) {
4
5      bmuDistance = inf
6      bmu = undefined
7
8      for(mapNode in MapNodes) {
9        distance = euclideanDistance(sample, mapNode)
10       if(distance < bmuDistance) {
11         bmu = mapNode
12         bmuDistance = distance
13       }
14     }
15
16     sample.x = bmu.x
17     sample.y = bmu.y
18
19   }
20   render(MapNodes, Samples)
21 }
```

Classification/verification stage

The last step in the SOM implementation is of course the actual classification, which will verify if (or to what degree) our program manages our classification task. Having the map with anchored, pre-classified sample documents, two simple steps will classify a sample:

1. Position the input sample in the map (find BMU).
2. Use k-Nearest Neighbor to determine which class' neighborhood the BMU is in.

This begs the question: Why go through all the trouble of generating a SOM if we're just going to use kNN anyway? The answer is two-fold. Firstly, the 2D map showing the organization and clustering of data is in many cases useful for presentation purposes. Second, and arguably more important, is the fact that kNN in only 2 dimensions is quicker than kNN in the dimension of the feature vectors.

For example, if the sample data is 600 samples containing 3000 unique words in total, simple kNN would have to calculate the distance to 600 vectors in the 3000th dimension (assuming we don't use any neighbor search optimizations). In the SOM, the kNN step only has to calculate the distance to a small number of vectors (the anchor samples) in 2 dimensions. True enough, we still have to calculate distances in the 3000th dimension to find the BMU in step 1, but only to a small number of vectors (each processing unit). Listing 4.6 outlines how the classification is done in our implementation.

Listing 4.6: SOM classification stage (pseudo code)

```
1  som_classify(Samples, MapNodes, AnchoredSamples) {
2
3    correct_count = 0, success_rate = 0
4
5    foreach(sample in Samples) {
6
7      bmuDistance = inf, bmu = undefined
8
9      for(mapNode in MapNodes) {
10       distance = euclideanDistance(sample, mapNode)
11       if(distance < bmuDistance) {
12         bmu = mapNode
13         bmuDistance = distance
14       }
15     }
16     sample.x = bmu.x
```

```
17        sample.y = bmu.y
18
19        classification = knn_classify_2d(sample, AnchoredSamples, k)
20
21        if(sample.realCategory == classification) correct_count++
22
23    success_rate = correct_count / count(samples)
24 }
```

Chapter 5

Results & discussion

5.1 Classifying problem / not problem

Our initial experiments concern the detection of problem/issue descriptions. To train the classifiers and validate the classifiers, we have a set of 800 samples, of which 400 are problem/issue descriptions and the remaining are not. We seek to determine which classifier that has the highest accuracy and efficiency.

A typical problem description may look something like the following: “My iPhone hangs when entering camera mode. I have tried this on my friend’s phone, and the same problem occurs here. The firmware installed on both phones is version X28923”.

5.1.1 Naive Bayes

Our implementation of Naive Bayes has no parameters to tweak. Our first experiments therefore simply involve finding the most accurate configuration of training set size and feature selection (words/n-grams). We also observe the differences in accuracy between the NS, AS and CS training sets (see section 4.1.1).

We run 50 iterations of training and validation for each configuration, with training and validation sets consisting of different samples each time.

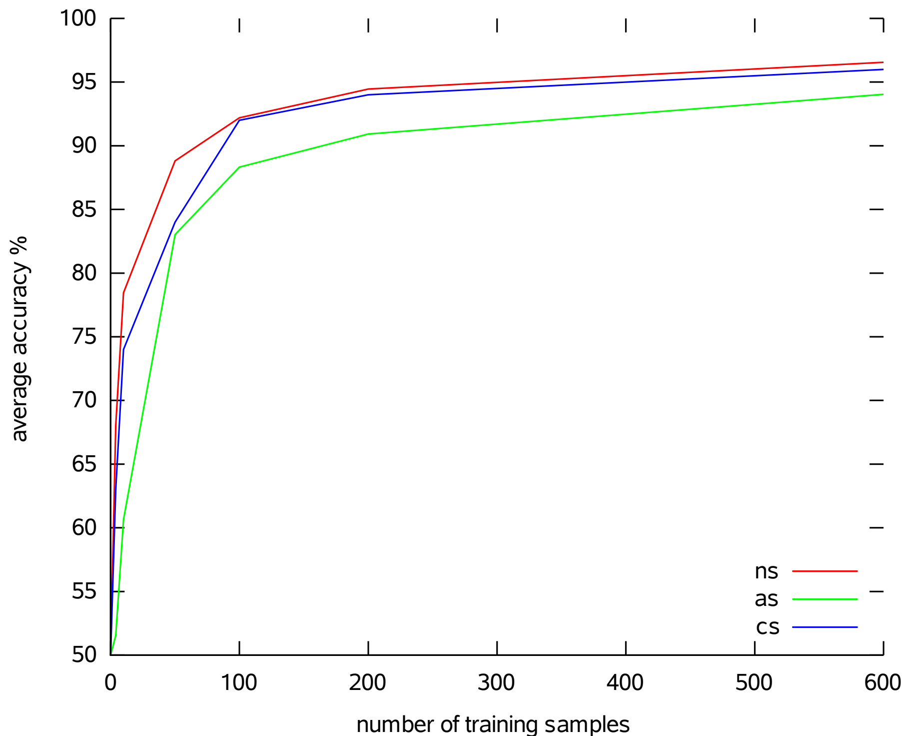


Figure 5.1: Naive Bayes accuracy with different training set sizes

The reason for this is that we want to rule out extremely good or bad results caused by incidentally getting an abnormally favorable set of samples. Additionally, the training and validation with Naive Bayes is relatively fast, so 50 iterations with 800 samples can be completed within reasonable time on our testing machine.

Words as features

Table 5.1 shows the average classifier accuracy using different training set sizes. Since we consider the CS training set the most unbiased, we focus on that. As indicated by the graph in figure 5.1, already at 100 training samples, the average accuracy is close to 90%. Even using only 4 training samples, the accuracy is better than what you'd get by simply guessing (50%).

# samples	Best	Worst	Avg	Std Dev
600	99%	93%	96%	1.2%
200	99%	89%	94%	1.9%
100	97%	85%	92%	3.0%
50	96%	79%	88%	5.5%
10	96%	51%	78%	14.4%
4	95%	45%	68%	15.1%

Table 5.1: Naive Bayes accuracy with words as features

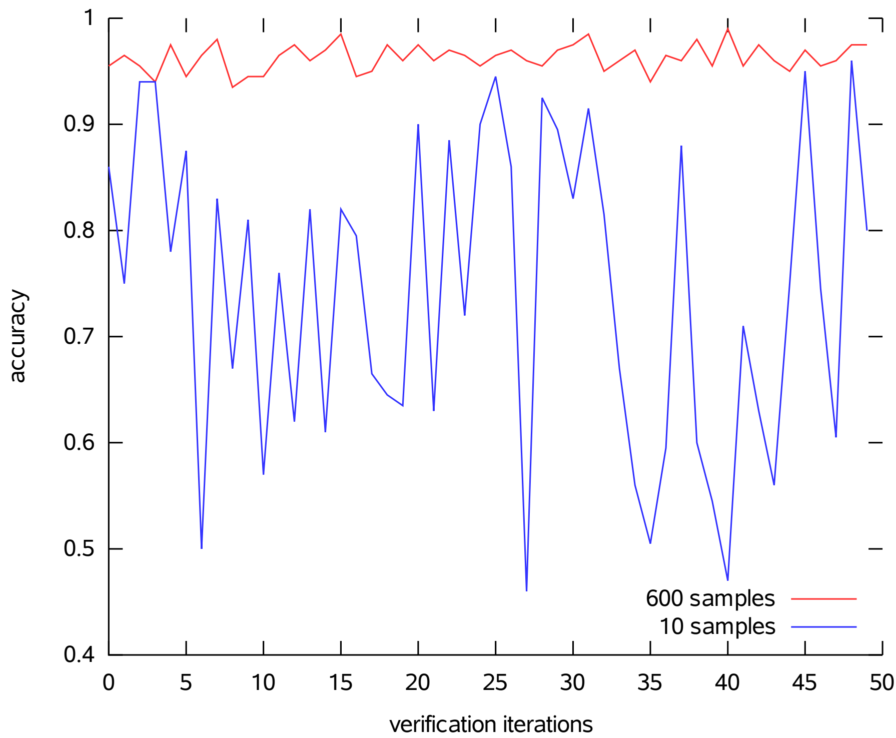


Figure 5.2: Naive Bayes accuracy per iteration

However, if we ignore the overall average accuracy and investigate the accuracy at each training iteration, the numbers tell a slightly different story. Figure 5.2 shows how much less the accuracy deviates from the average with

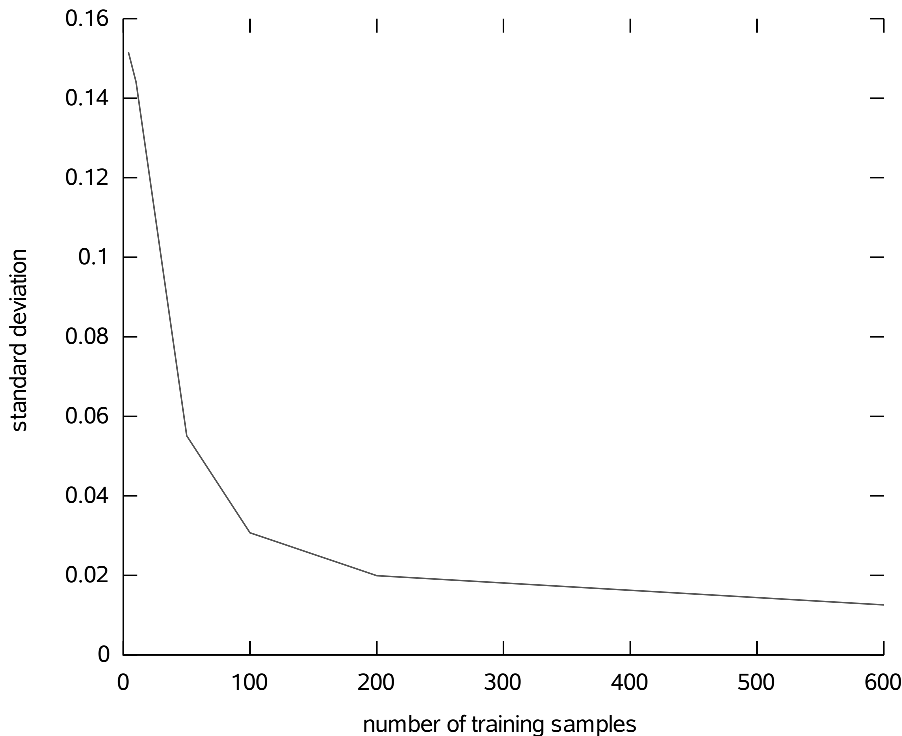


Figure 5.3: Naive Bayes accuracy standard deviation with different training set sizes

600 and 10 training samples.

Our results indicate that although the average accuracy may appear impressive even when using a tiny training set, you need a larger set to reduce the standard deviation. In other words, the smaller training set, the more dependent the classifier is on getting an overly ”good” training set. Figure 5.3 illustrates how the standard deviation is reduced exponentially when the training set size grows.

The average accuracy (figure 5.1) seems to flatten out when we exceed a few hundred training samples. Similarly the standard deviation flattens out around the same area. Because of this, we assume that anything more than our largest training set (600 samples) is unlikely to increase the performance significantly.

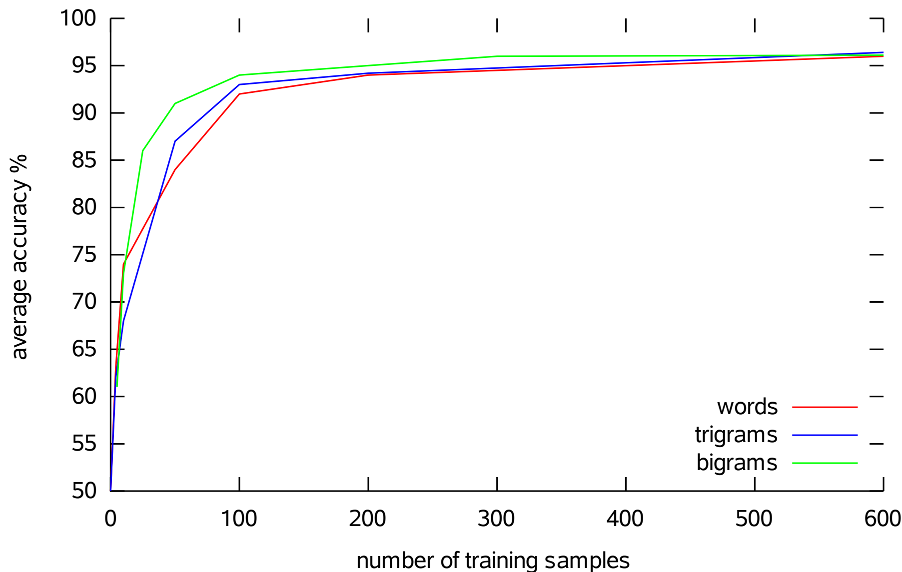


Figure 5.4: Naive Bayes accuracy, bigrams vs words

N-grams as features

Interestingly, both using bigrams or trigrams instead of words produces a marginal increase in the already rather impressive accuracy. However, as can be seen in figure 5.4, the increase is negligible, so in practice all three feature selections perform equally well for our classification task. Standard deviations resemble those found for words in the previous section.

5.1.2 K Nearest Neighbor

As with Naive Bayes, our K Nearest Neighbor implementation doesn't have very many parameters, apart from training set size and type of features. It does, however, have the k, which determines how many neighbors should vote on the classification. In addition to set sizes and features, we will learn which k produces the most accurate votes.

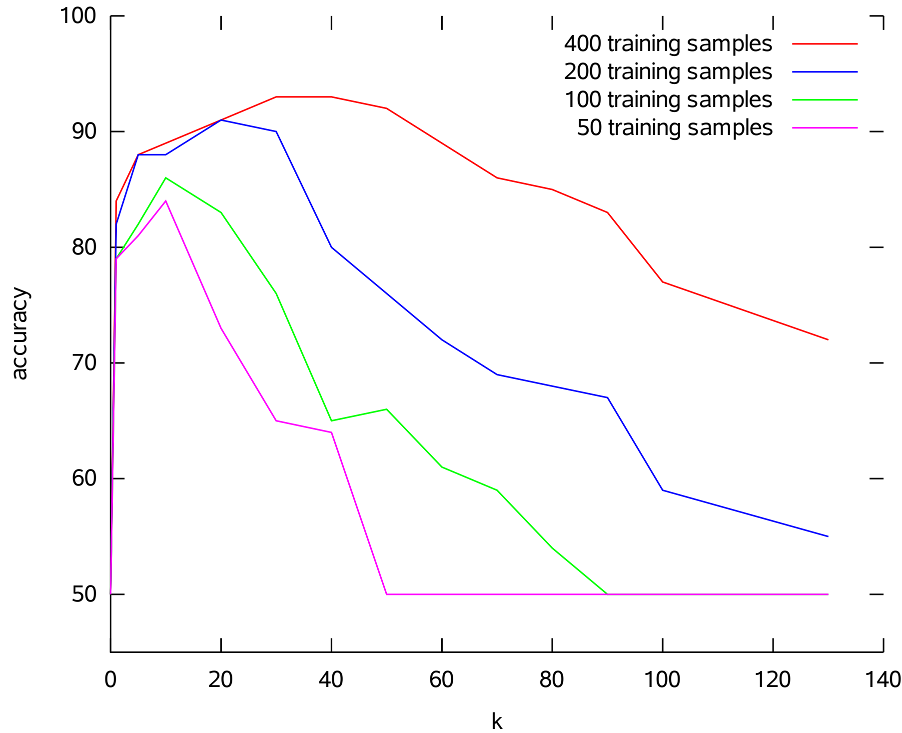


Figure 5.5: kNN accuracy with different training set sizes and varying k

Words as features

Once again, we're focusing on the CS sample set. Plotting the accuracy for different values of k gives us the graphs shown in figure 5.5. Their values indicate clearly that the most ideal k varies with the size of the training set. Please note that while it may look like $k = 0$ gives a high accuracy, all four graphs in reality have an accuracy of 50 when k is zero, which is the same as ignoring the neighbors and and just guessing the classification.

The graphs in figure 5.5 show the accuracy for kNN with training sets of 400, 200, 100 or 50 samples. The highest recorded average accuracy for the 400 set is 93% when k is 30. Similarly, for the 50 sample set, the highest is 83% when i is 10 and so on. Table 5.2 shows the highest recorded accuracies for several more training sets of different sizes.

# samples	Highest avg	k at highest avg
600	94%	50
400	93%	30
200	91%	20
100	86%	10
50	84%	10
10	70%	5

Table 5.2: kNN accuracy with words as features

If we plot the data from table 5.2, we get the graph in figure 5.6. The trends resemble those of Naive Bayes: after a certain amount of samples, the classifier hits its maximum. For Naive Bayes that amount was around 200 samples, for kNN it's found near 400. This indicates that NB is more accurate than kNN when trained with fewer samples. The overall best accuracy for kNN converges around 94% (versus 97% for NB).

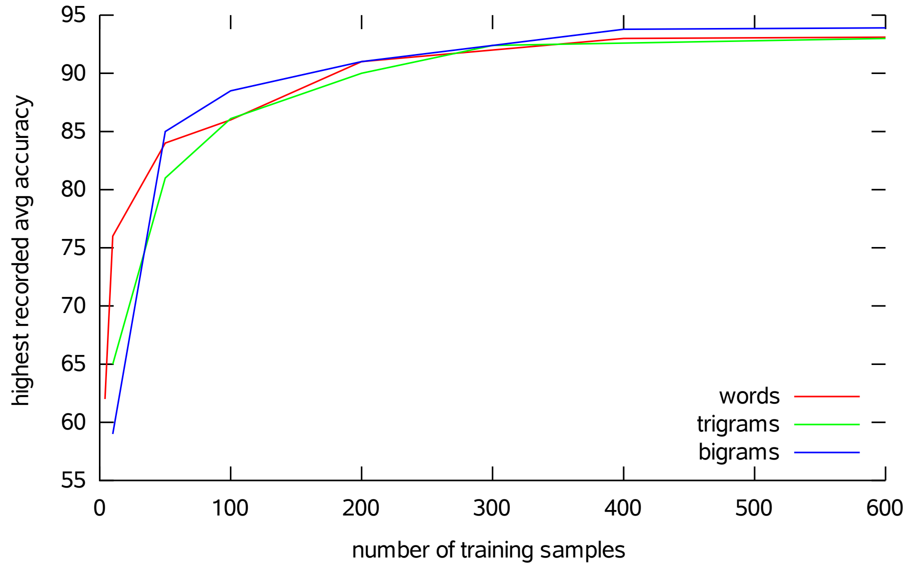


Figure 5.6: Best recorded average accuracy for kNN

N-grams as features

In figure 5.6 we have also plotted the best recorded results using bigrams and trigrams as features. We see that the results are similar to those we found with for Naive Bayes: using n-grams increases the accuracy slightly, but for all intents and purposes negligible.

5.1.3 Self-Organizing Map (Kohonen Network)

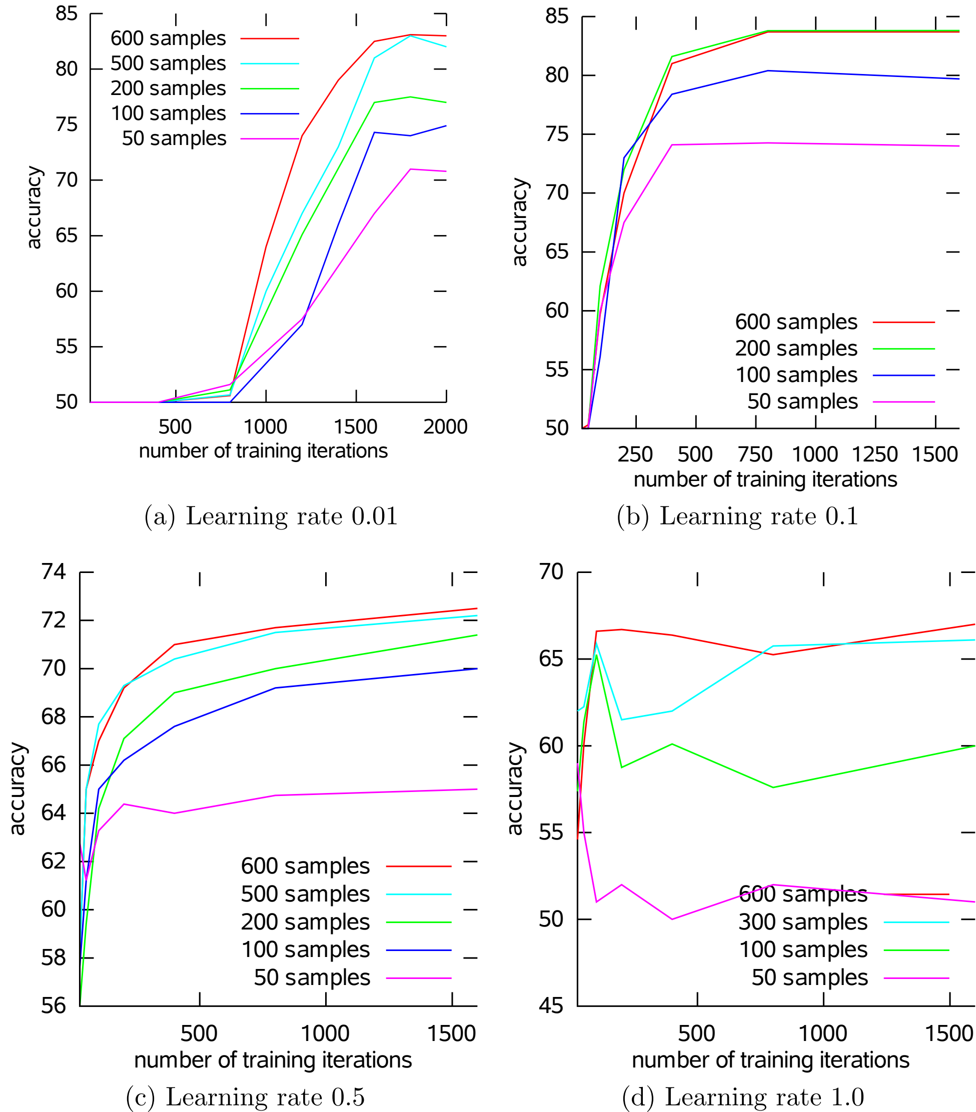
The Self-Organizing Maps technique introduces a plethora of different parameters compared to Naive Bayes and kNN. In addition to the type of feature selection and training set size, the SOM (as described in 3.3.1) can be generated with different number of learning iterations, initial learning rate, map size and so on. Furthermore, we can vary the number of anchor documents in the anchoring stage, and the classification/validation can be done with different values of k (recall that we are using kNN in this stage). Because of the number of different combinations of these parameters, and the fact that generating a map takes a considerable amount of time compared to kNN and Naive Bayes, we'll take some shortcuts.

We'll only consider words as features. Because SOM and kNN are both based on comparing TF-IDF vectors, we can (rightfully, as the results will show) assume that their results will be similar. Since choosing n-grams over words had limited impact on kNN's accuracy, it's unlikely affect the accuracy differently here. Additionally, as we have mentioned earlier, we limit ourselves to 2 dimensional maps with square processing units.

Sample set sizes, learning rate and iterations

As always, one of the most interesting parameters is the number of training samples required to get a good accuracy. We start off with a SOM size of 10x10 and investigate how the accuracy varies with different sample set sizes, learning iterations and initial learning rate. Figure 5.7 shows the recorded results for 4 different learning rates. Note that the results here were produced with a k of 40 during the validation stage.

With respect to training set size, the results show that bigger is better,



(a) Learning rate 0.01

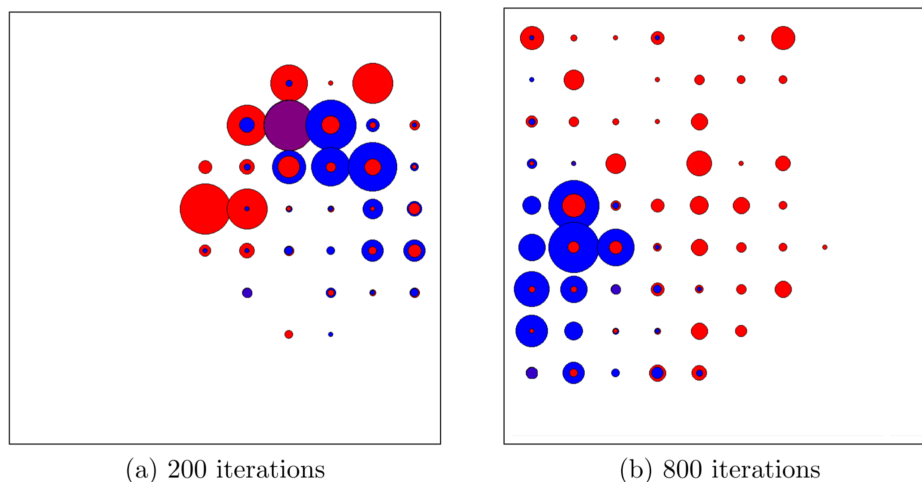
(b) Learning rate 0.1

(c) Learning rate 0.5

(d) Learning rate 1.0

Figure 5.7: 10x10 SOM, varying iterations, training set sizes and initial learning rate

but anything more than 200 samples is usually a waste. With the exception of when the learning rate is very small (5.7a), increasing the training set from 200 to 600 only improves the accuracy with 1%.



(a) 200 iterations

(b) 800 iterations

Figure 5.8: 10x10 SOM, anchored samples (learning rate 0.1 and 200 training samples)

If we consider the number of training iterations and learning rate, we see that a small initial learning rate requires more iterations and vica versa. With an initial learning rate of 0.01, at least 1000 iterations are needed to push the accuracy over 60%. A higher learning rate, for example 0.5, gives us a steeper graph initially where 60% is surpassed already by the 50th iteration (5.7c). The snapshots in figure 5.8a and 5.8b show how the anchor samples would typically organize themselves after 200 and 800 iterations, respectively.

On the other hand, the overall best accuracy belongs to the lower learning rates. We find the best combination of training set size, learning rate and iterations with a learning rate of 0.1, 200 training samples and about 1000 iterations. This produces a classification accuracy of just under 85%. Neither upping the amount of samples, using more iterations or smaller learning rate increases the accuracy significantly from that point.

Anchor samples and k

It's reasonable to think that the amount of anchor samples should have a substantial influence on our choice of k for the kNN in the validation/classification stage. If we have few anchor samples, a large k would probably include too many neighbors in the vote and produce inaccurate results. From

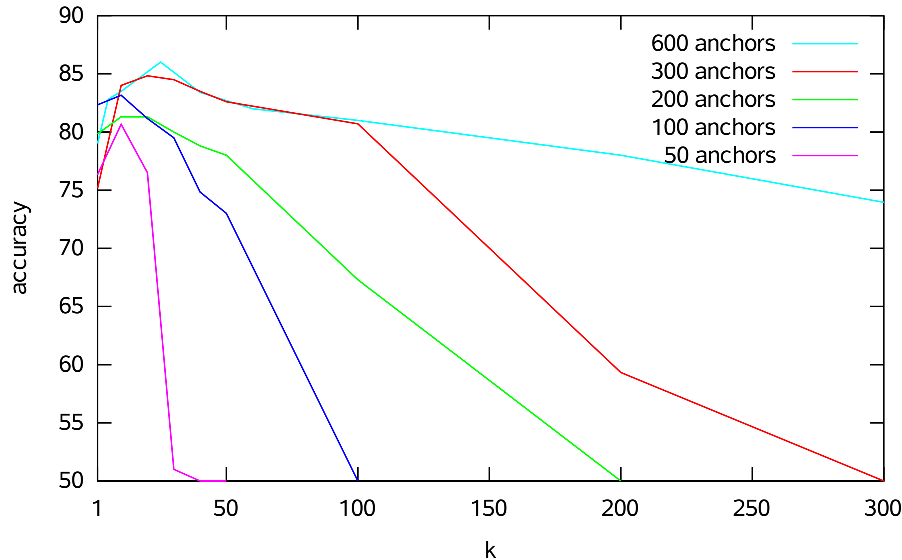


Figure 5.9: SOM accuracy with varying number of anchor samples and k

a performance perspective, we always want to have as few anchor samples as possible, but using too few will impede the accuracy. The results with different number of anchor samples and a varying k is shown in figure 5.9. The graphs have several noteworthy elements.

First off, it's evident that even using just Nearest Neighbor (i.e. $k = 1$) gives us a decent classifier. Even with only 50 anchor samples, the accuracy is above 75% when k is 1. If we increase k to 10, 50 anchor samples actually yields 80% accuracy, which is fairly impressive.

The overall impression of the graphs is that more anchors equals higher accuracy. This does, however, not seem to be the case for the smallest values of k. We attribute this to the fact that smaller k are more prone to irregularities in the samples (section 3.2.3). With SOMs being very time consuming to generate, we had to reduce the number of training/validation runs. While we could easily so 50 runs for each parameter combination with Naive Bayes, with SOM we had to reduce the number to 5. The results per run will therefore deviate more and may sometimes produce a slightly misguiding average accuracy.

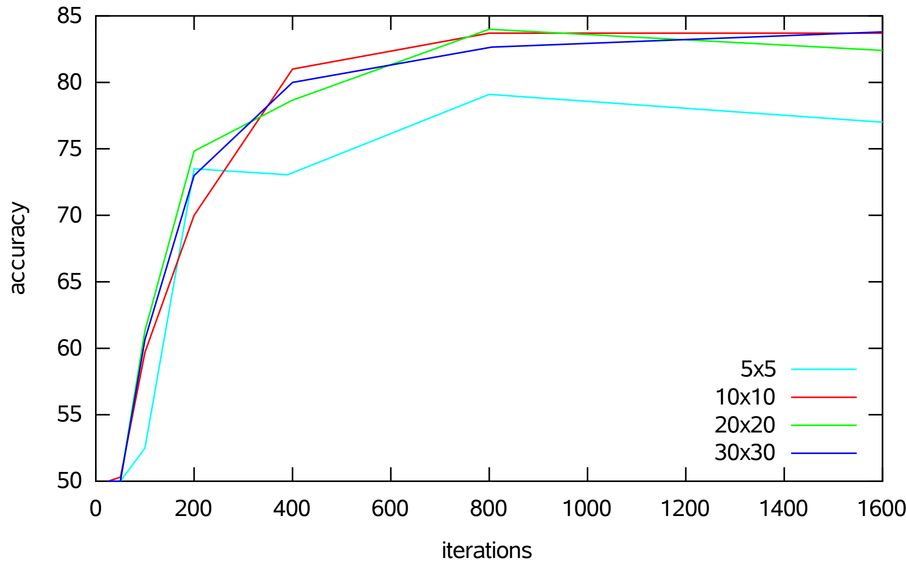


Figure 5.10: SOM accuracy from different map sizes

We also observe that the overall best accuracy does not seem to improve much from what we found using $k = 40$ in the previous tests (just under 85%). True enough, with 600 anchors, we see a peak ever so slightly over 85%, but the improvement is negligible. More interesting though, is the fact that 300 anchors seem to perform comparably with 600.

Map size

We can adjust the SOM's resolution and thereby produce a more fine-grained map than the 10x10 used in the previous experiments. An increased resolution will in theory give more accurate results, but obviously also require more time to self-organize. The graphs in figure 5.10 indicate that increasing the map size beyond 10x10 provides no increase in overall accuracy for our classification task. The number of iterations before the accuracy converges to its maximum also remains the same.

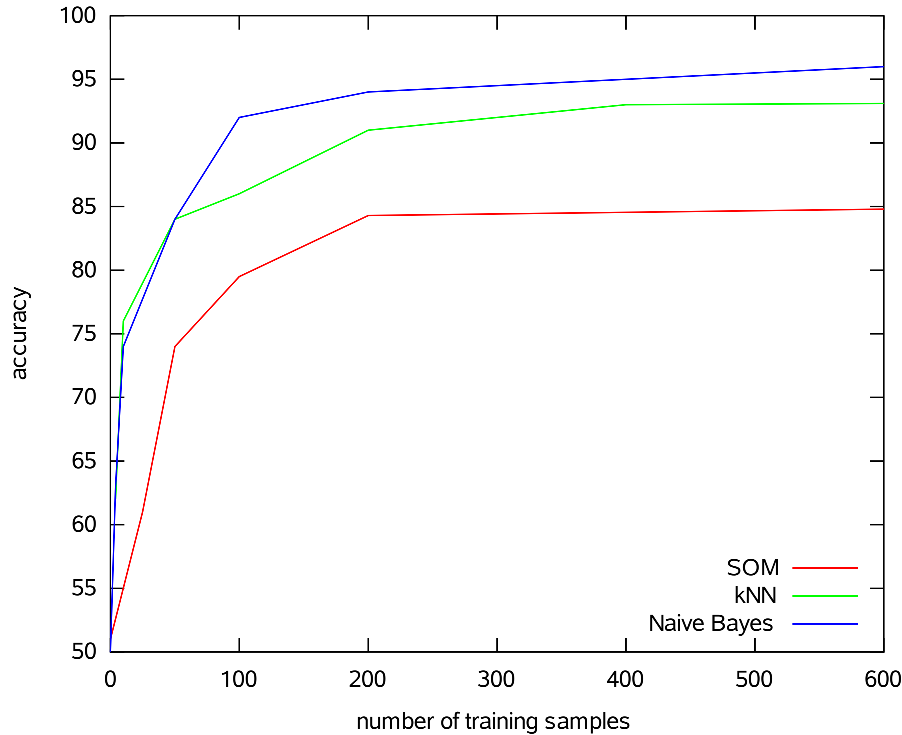


Figure 5.11: Result comparison Naive Bayes, kNN and SOM

5.1.4 Comparison

Based on the results seen in this section, we can illustrate the comparison between the three techniques as seen in figure 5.11. In summary, we see that Naive Bayes produces the most accurate classification, with kNN close behind, while SOM turns out to be slightly less accurate. Because SOM and kNN both are based on measuring distances between the samples' TF-IDF vectors, one would perhaps think that their performances would be more similar. However, some information is lost in the process of scaling the multidimensional feature vectors down to 2D [2], so it's unrealistic to expect SOM to perform as accurately as kNN.

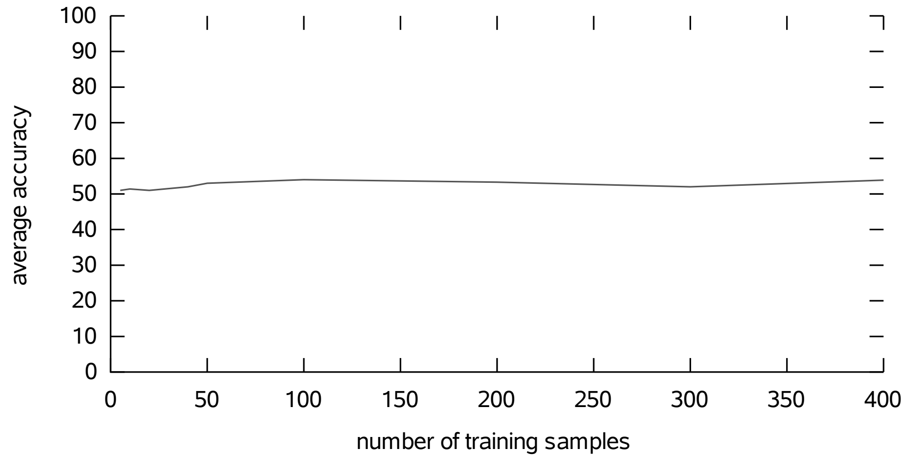


Figure 5.12: Naive Bayes accuracy on informative/not informative

5.2 Classifying based on informativeness

The second part of our hypothesis was that we could also use a classifier to determine whether a problem description was informative or not informative. The classifier used to test this would be the one with the highest accuracy from our first classification task (problem/not problem, section 5.1). This marginally turned out to be Naive Bayes.

As we mentioned in section 4.1.1, the sample data for this classification task is potentially very confusing to the classifier. Regardless, our Naive Bayes classifier is more than happy to try. In figure 5.12 we've plotted the results with different numbers of training samples. Unfortunately, but unsurprisingly, we can ascertain that the results are well under par, even when utilizing 400 training samples (the remaining 100 being used for the validation). The average accuracy steadily remains between 50 and 55% - barely any better than guessing. Moreover, we see that the increase in training set size hardly yields any perceptible improvement on accuracy. The classifier has simply not been able to learn what distinguishes informative and non-informative forum posts.

In section 5.1.1, we saw that even though average accuracy may not improve much from increasing the amount of training samples, it may reduce

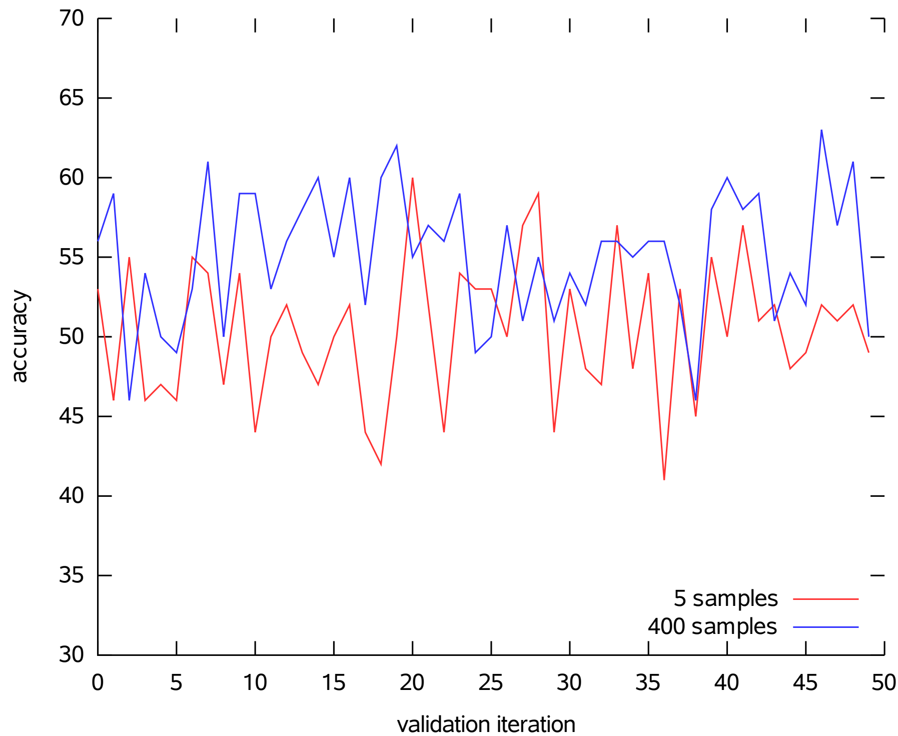


Figure 5.13: Naive Bayes accuracy deviation on informative/not informative

the standard deviation between the different runs that make up the average. Recall that we do several runs with different training and validation samples each time. A smaller standard deviation means that the results are more consistent for each such run. The graphs in figure 5.13 show the individual accuracy results for each run.

Even at a glance, it's clear that the deviation is not noticeably reduced from 5 training samples to 400. A closer look at the numbers reveal that the standard deviations are 4.3 and 4.0 for 5 and 400 training samples, respectively.

Chapter 6

Conclusion & further work

6.1 Conclusion

In this thesis we have developed a prototype application for detecting problem descriptions among online forum posts. The prototype implements three different algorithms for pattern recognition with machine learning, namely k Nearest Neighbor, Naive Bayes and Self-Organizing Maps. We have run extensive tests with each one to determine which one and what parameters will yield the highest accuracy.

Furthermore, we have investigated whether we could also extended the prototype to classify problem descriptions as either informative or non-informative. For this second classification task we used the algorithm that most accurately detected problem descriptions.

All three algorithms have been shown to perform fairly well for the first task (detecting problem descriptions), given the right training, configuration and parameters. Naive Bayes proved to be the winner, reaching an impressive classification accuracy of 96%. K Nearest Neighbor was only marginally worse, with a maximum recorded accuracy of 94%. Being that Self-Organizing Maps rely heavily on the many of same calculations as kNN, we expected them to perform comparably. However, the accuracy of our SOM topped out at 85%, indicating that some information is lost to the inherent reduction of feature space dimensionality. Regardless, the SOM did

excellent in illustrating the clustering of samples, as seen in figure 4.2. Overall, these results should be of great use in situations where it's important to get an overview of people's issues or complaints with a product.

The results of classification based on informativeness were not as satisfactory as the first. This was essentially something we feared already from the point where the training samples were gathered. It was simply incredibly difficult to objectively and formally determine whether a sample should be considered informative or not informative. The determining factors were usually contextual or subjective: for example a forum post could be informative when seen in the context of the forum thread but not on its own.

Our fears were reflected in the classification accuracy. At best, 56% were consistently classified correctly – hardly better than simple guessing. In other words, the classifier in its current form is not able to classify informative and non-informative samples.

6.2 Further work

The three algorithms admittedly performed beyond expectation for the first classification task. This is a testament to the fact that "naive" classifiers can often perform extremely well even in cases where one would think the features are dependent and need contextual data to convey their importance (such as when classifying text based on its "meaning"). Nevertheless, we've discussed a number of possible improvements that should be researched from here on.

Combining the algorithms Even though Naive Bayes proved to perform better than kNN and SOM, it safe to assume that all three have strength and weaknesses. A possible improvement to our classifier prototype would be to let all three have a vote in the classification of a sample. They're all in practice probabilistic, meaning that instead of a classification, they can produce the probability of each class being the correct one. Let's imagine that Naive Bayes tells us that the probability of a sample being a problem description is 60% and 40% that it isn't. At the same time kNN informs us that 75% of the sample's k nearest neighbors are problem descriptions and 25% are not. We can then, for

example, choose to trust kNN, because it seems to be more "certain" than Naive Bayes. Alternatively, we can take the average probability from each classifier or simply use a majority vote.

Improving feature extraction So far we've used words or n-grams as features for our classifiers, and observed that the differences in accuracy were essentially negligible. What we haven't tried, is to use mixed order n-grams. Mixed order n-grams means for example using bigrams and trigrams at the same time, or even together with the whole words.

The second classification task was not as successful as we'd hoped. With such a depressing accuracy, it's obvious that more research is in order. What's important to note is that even though our Naive Bayes classifier was unsuccessful in this classification task, we can't thereby deem it unsuited. The main issue is probably not the nature of the classifier and its naivety, but rather the poor distinction between the two classes in our training set. As we have mentioned before, it's more appropriate to have a linear scale to measure informativeness, rather than two discrete classes.

Bibliography

[1] "Swansea university, library jargong glossary," http://www.swan.ac.uk/lis/HelpAndGuides/LibraryJargon/.

[2] W. Basalaj, "Proximity visualization of abstract data," 2001, http://www.pavis.org/essay/index.html.

[3] BIGreasearch. Simultaneous media usage study 10. Http://globaltechforum.eiu.com/index.asp?layout=rich_storydoc_id=11201.

[4] W. B. Cavnar and J. M. Trenkle, "N-gram-based text categorization."

[5] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*. John Wiley & Sons, Inc., 2001. [Online]. Available: http://www.amazon.com/Pattern-Classification-2nd-Richard-Duda/dp/0471056693

[6] D. T. Larose, *Data mining, Methods and models*. John Wiley & Sons, Inc., 2006.

[7] Y. Liao and V. R. Vemuri, "Using text categorization techniques for intrusion detection," *University of California*, 2002, http://www.usenix.org/event/sec02/full_papers/liao/liao_html/text_ss02.html.

[8] D. Mack, "Reverend bayes' useful contribution," *Potentials, IEEE*, vol. 12, no. 1, pp. 41–42, Feb 1993.

[9] About consumer-generated media (cgm). Nielsen Online. [Online]. Available: http://www.nielsen-online.com/resources.jsp?section=about_cgm

[10] L. K. SlÃttebrekk, "Background information on e-wom'ers," bI Norwegian School of Management.

[11] Y. Yang and X. Liu, "A re-examination of text categorization methods," in *SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval.* New York, NY, USA: ACM, 1999, pp. 42–49.