Universität Koblenz-Landau
Fachbereich Informatik

Diplomarbeit

MIC — A System for the Classification of Structured and Unstructured Texts

Gerd Beuster

Wintersemester 2000/2001

This dissertation was submitted in partial fulfillment of the requirements for a Diplom-Informatiker degree at the University Koblenz-Landau.

Supervisors:
Prof. Dr. Ulrich Furbach and
Dipl. inform. Bernd Thomas

The author hereby declares that no third party was involved in writing this dissertation. All sources and texts needed to do this work are listed in the bibliography.

An der Universität Koblenz-Landau eingereichte Diplomarbeit.

Gutachter:
Prof. Dr. Ulrich Furbach and
Dipl. inform. Bernd Thomas

Ich erkläre hiermit, daß die vorliegende Diplomarbeit von mir selbständig verfaßt wurde. Alle benötigten Texte und Quellen sind im Literaturverzeichnis aufgeführt.

Acknowledgments

I want to thank a lot of people who helped me to complete this work. The University Koblenz Artificial Intelligence Research Group, lead by Professor Furbach, provided a stimulating environment and always encouraged me in my scientific endeavors. My co-worker and advisor Bernd Thomas helped a lot during the whole process of writing this thesis, from defining the topic of this thesis to discussing the results and proof reading. I owe to my parents Peter Beuster and Margret Bauer for supporting my interests in science (and for paying my bills). My girlfriend Eva-Maria Kahrau had the patience to go with my through the — sometimes stressful — time of writing my thesis. Oliver Obst, Bernd Thomas and Eva-Maria Kahrau helped by proof reading parts of this thesis.

All these people, and a lot more, made this work possible.

Contents

Part I.

Automatic Text Classification

1. Motivation

The proliferation of computer based communication in all aspects of life leads to an overwhelming amount of data available to the user. Nowadays, nearly all public knowledge is available on the Internet, and can be retrieved within seconds. At the beginning of March 2001, the search engine Google [10] has indexed 1,346,966,000 web pages. This is only a fraction of all documents available on the WWW! For somebody using email in the workplace and privately, it is not uncommon to receive more than a dozen, for some even more than a hundred emails a day. For the first time in history, the knowledge of the world is put to the fingertips of those who have access to the modern means of computer databases and communications. Back in the old days, the location and retrieval of stored information was the main problem in getting informed. The problem of information retrieval lies in a different field now: In order to get informed, one has to filter out the relevant information from all the irrelevant and misleading information under which it is buried. For somebody who works with the Internet, the search for knowledge is nowadays like looking for a needle in a haystack.

Various methods have been proposed and implemented to use information technology as an aid in finding and sorting information. Search engines are an invaluable tool when looking for information on the Internet. Still, the power of automatic tools for information retrieval is limited. The user interface of search engines have limited expressiveness. Common search engines do not really find information, but only limit the huge space of billions of web pages to those web pages which might contain information. In other aspects of modern communication, namely email and Usenet, information retrieval methods are little used.

The approach of traditional search engines is very different from the way a human retrieves information from a library, a book, a newspaper or her personal correspondence. Search engines, as well as systems for the automatic filing of emails, need explicit and strict rules what to look for. The way a human classifies information is rather associative. A human decides which letter to read first, or which article to read from a newspaper, not based on strict rules, but based on association from previous experiences. In this associations, a human heavily relays not so much on the informational content of an article or a letter — in order to gasp the content, he or she would have to read it in the first place — but on *structural* information. This *structural* or meta-information is for example the envelope. Just by looking at the envelope, a human knows whether this is a high-priority letter from his rich aunt, or a bill from the phone company. A human needs little information to decide whether a newspaper article is interesting: he or she will look where the article is placed, what's the headline, perhaps who wrote it, and which pictures

are in the article.

In this thesis, we show how a similar approach can used for automatic, computer-based information retrieval. We show ways to classify information not based on strict rules, but on previous examples. We show how the structural information about a document can be taken into consideration for the classification task.

Text classification is not only useful when trying to find and extract information from global, distributed databases like the Internet. On a local base, many people and companies have large sets of semi-structured data stored in databases and on computer hard disks. In order to retrieve this information, intelligent classification methods are necessary.

This thesis is split into two parts. In the first part, we show how generic methods for text classification can be adapted to tasks where we have additional information provided by the structure of the documents. We show how the performance of generic text classification algorithms can be improved by exploiting structural information from emails and web pages. The second part gives a system description of `MIC`, the text classification system which has been developed for this thesis. MIC implements the classification method from this part of the thesis. In the second part, we also show how MIC can be used on a variety of real world tasks.

1.1. Text Classification Applications

The University Koblenz AI Research Group is involved in several projects developing applications for *intelligent information retrieval*, *machine learning*, and *intelligent web search*. In some of these projects, we need a component for the automatic classification of texts. Numerous methods and implementation for automatic text classification are available. For a comparison of the performance of various text classification methods, see [42]. Generic text classification methods, usually based on statistical methods, use "plain" text as input. In the projects we are working on, we usually do not deal with plain texts, but with documents which contain some additional structural information about a text. An example for these structured texts are web pages. The text contained in web pages is structured by a set of tags. The tags mark structural properties of certain text segments, e.g. indicating that some text is a headline, a paragraph of ordinary text, or a link to another web page. The University Koblenz AI Research Group is actively researching methods to utilize this structure information for information extraction tasks. These work is done by Bernd Thomas. Publications include [36], [37], [39]. A full list of literature can be found on Bernd Thomas' web page [35].

The goal of this thesis is to create a text classification system which can be used within these projects. For this, it has to fulfill two main requirement:

1. Provide text classification algorithms which exploit the special features of structured documents

2. Be flexible enough to be used in a large variety of environments, both as an embedded component in larger systems and as a stand alone text classification program.

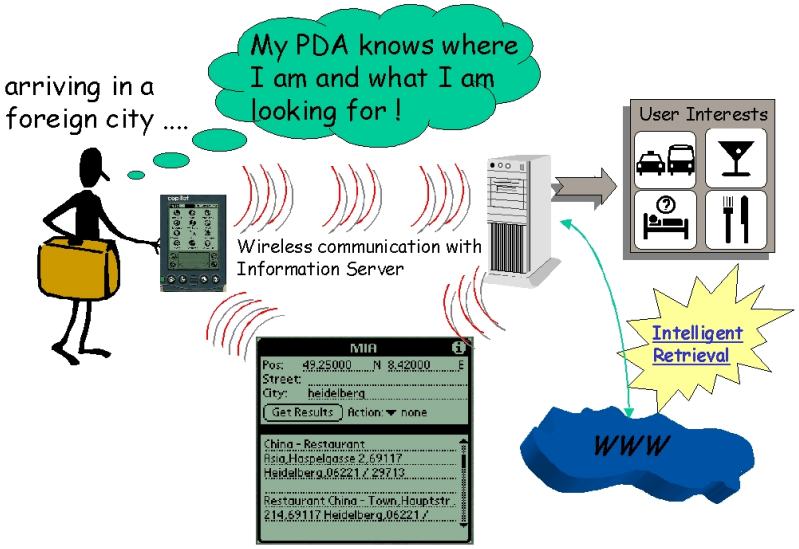

Figure 1.1.: MIA System. (From [38]).

We will use our text classification methods and the system which implements them in three practical scenarios: As a component in the MIA-system, as a component of a system to automatically classify emails, and as a stand alone program.

We are looking for a text classification system which can be used in practical applications. In the next sections of this chapter, we will define the special requirements of the text classification tasks we want to solve.

1.2. The Info System MIA

MIA ("Mobile Information Agents") is an ongoing research project at the University Koblenz-Landau Artificial Intelligence Research Group [1]. The aim of MIA is to create a search engine focusing on the needs of *mobile* users using AI-techniques. When a traveler comes to a place that she does not know, the MIA system shall provide her with relevant information. Some information is relevant, when it meets the *interests* of the user, and the *location* of her. An example might be information about the next restaurant of her favorite cuisine, or the program of the local movie theater.

The MIA projects realizes this by combining hardware and software in a unique way. The hardware components used in MIA are: A server with a direct connection to this Internet. This server does the actual searching. Since MIA targets mobile users, the user connects to the server from a PDA via a wireless Internet connection (usually a data-enabled cellular phone). The current position of the user is either supplied manually, or read out from a GPS-device connected to the PDA. This is shown in figure 1.1.

Beside being based on the local position of the user, there are several other key differences to ordinary search engines:

- Intelligent spidering

  Ordinary search engines keep huge databases of indexed web pages. When queried, they do not search the actual web, but their database. MIA does not spider the web exhaustively in order to create a large database. It spiders intelligently. It "surfs" the web like an ordinary user. Starting from common search engines, it follows links until it finds the information the user is looking for. One consequence of this is that MIA's spidering is too slow for synchronous operation. For an ordinary search engine, one issues a query and gets the answer immediately. MIA works asynchronous. The user defines her search-profile (a collection of the topics she is interested in while on the road) before a trip, and sends updates about her position in regular intervals while on the road. The server uses this information to spider web pages and collect information. Results are stored on the server. The results are transmitted to the user when she requests them.

- Information, not references

  The second big difference to ordinary web search engines is that the results of a MIA-search is not a list of links to web pages which might be relevant, but the actual information the user is looking for. This information is extracted from the web pages found by the MIA spider module. So far, MIA can extract *address information*. When the users' search profile defines she is interested in Chinese restaurants, MIA comes up with a list of addresses of Chinese restaurants in the vicinity of the user.

The MIA search engine has to deal with three kind of information:

1. Search Profile

   The user sets up a search profile. The search profile defines what the user is looking for. At this point, this is a list of keywords, combined with some logical constraints. Examples for a search profile are `Restaurants only Chinese`, or `Sports not Squash`.

2. Kind of Information

   The *kind of information* the user is interested in. Currently, MIA can only retrieve *address information*. When the user searches for `Restaurants only Chinese`, MIA will come up with the addresses of Chinese restaurants. In further versions, we also want to provide information about *event descriptions*, so the user can for example get information about concerts, and *time table information* in order to provide the user with temporal relevant informations, like bus schedules.

3. Geographical information

   MIA focuses on *mobile* users. Its aim is to aid a user who is on the road. The user will define her *search profile*, and the *kind of information* she is interested in before she starts the trip. While on the road, she will send information about her *geographical position* to the search engine. This information is used to guide the

search. MIA will only search for Chinese restaurants in the vicinity of the user's current position.

Within MIA, the *text classification agent* is used to classify texts in regard to the *kind of information*. Namely, it gives a confidence measure judging how likely it is that the web page given to the classifier contains an address.

MIA could do without a classification subsystems. In this scenario, the web pages collected by the spider subsystem are directly passed to the information extraction subsystem, which tries to extract addresses from the web page. Putting a classification subsystem in between the spidering and the extraction subsystem improves the performance of the system, both in terms of its effectiveness and its efficiency:

- Efficiency

  Extracting information from a web page is a very complex and time-consuming task, even when it fails. More than 50% of the web pages gathered by the *spider agent* do not contain any address. By sorting out these pages, we can speed up MIA.

- Effectiveness

  When trying to extract information from a database, we can use different methods. Some of the methods give very accurate results (when they extract something, it is most probable an address), but fail to extract an address from time to time. Other methods are more "loose". They do not disregard addresses, but from time to time they extract something which is not an address at all. When the classifier provides an confidence measure about how likely it is that a page contains an address, we can select the method to use for extraction more accurately.

There are some more requirements: Since the web is huge, the sets of examples and documents which have to be classified are very large. The text classifier shall be able to deal with large datasets gracefully. MIA is a multi agent system. The various components of MIA are independent from each other and communicate in well defined languages via well defined communication interfaces. The classification system must be able to interface with the other agents.

1.3. Email Classification Requirements

For classifying email, the classification program is invoked by the MDA (Message Delivery Agent) when a new email arrives in the system. The classification system stores the emails in different folders, depending on their classification, or forwards them to other email addresses. This is useful in a lot of scenarios.

- Spam

  UCE/UBE[1], also known as Spam, can be bounced before it reaches the mailbox of the user.

[1] Unsolicited Commercial Email / Unsolicited Bulk Email

- Forwarding email to the responsible person

  Larger email systems usually have a set of more or less generic accounts which receive a lot of traffic. These accounts might be `abuse` for email related to abuse of the system, `postmaster` for email related to the mail system in general, `info` to request information from the owner of the system, and `support` to ask support questions. An automatic email classification system can forward mail which is addressed to a generic account to the person who is actually responsible for answering a certain kind of request.

- Splitting up mail in appropriate folders

  Nowadays, a lot of people receive large numbers of emails from various sources. Examples for these sources include friends, coworkers and mailing lists. A lot of users do not want to have all these messages in the same folder. For them, the text classification system can sort the messages into different folders, depending on the source or the topic of the message.

For these scenarios, it is not feasible to work with fixed sets of category. The classification system should be able to learn the categories from examples. This means, the user provides it with a set of example emails whose categories are known, and it shall learn from these examples how to classify new email. In the same way as the structural information of HTML pages is used for the classification of web pages, the structural information of emails should be used in this scenario.

As an email classification system, it should be so reliable that it can be used in a production environment without losing emails due to bugs. We also want to have a general purpose tool for text classification. The text classification system should be usable as a desktop-application for the classification of all kinds of documents. The user should be able to define categories and example sets of documents, and the system shall be able to put new documents into these categories when it is asked to. For such a system, two characteristics are most important: ease of use and flexibility. In order to make the system simple to use, it should have additional user interfaces: A command line interface, and a graphical user interface.

Flexibility can be achieved by various measures. One important feature is not to use a single classification method, but provide a set of classification methods so the user can choose the one most appropriate for her task. Beside providing a number of build-in classification methods, new methods should be simple to integrate into the system.

The text classification system should not only be flexible in regard to the classification method. Before a classification algorithm starts to work on a text, usually various preprocessing steps are applied to the text: Special characters and stop-words are removed, words are converted to all lower-case, ... The system should also be flexible in the selection of these *preprocessing* steps.

One last requirement: Nobody wants to use non-free software. The system should be free under the terms of the GPL [11].

1.4. Conclusion: Special Requirements for MIC

Taking together all the requirement, the text classifier we are looking for shall fulfill the following requirements:

- Can be used as a software agent (→ chapter 6.3)
- Can deal with web pages as well as plain text (→ chapter 3.2)
- Trainable on various tasks (→ chapter 2.1)
- Can handle large datasets (→ chapter 7)
- Learn from examples (→ chapter 2.2.1)
- Make use of special structure of emails (→ chapter 3.2)
- Reliable system (→ chapter 7)
- CLI (→ chapter 6.3)
- GUI (→ chapter 6.2.1)
- Multiple classification methods (→ chapters 2.4–2.6)
- Simple integration of new classification methods (→ chapter 7.3.6)
- Flexible input data manipulation (→ chapter 3)
- Free Software

The rest of this thesis shows how this requirements can be fulfilled. It is split in two parts: The first part gives a formal definition of automatic classification. It shows how the performance of common classification algorithms can be improved by exploiting the structural information of texts, and ends with a comparison of the results of our improved methods to those acquired by generic, off-the-shelf text classification systems. The second part of the thesis describes the system MIC, the application we have developed for automatic text classification.

2. Automatic Classification

The previous chapter describes the process of text classification, and its application to various domains, in a rather informal way. This chapter gives a formal definition of automatic classification. Based on this, the automatic classification methods used within this thesis are introduced. The following chapter show how these general automatic classification methods can be applied to text classification tasks.

2.1. Basic Concepts

We start the formal definitions by defining *automatic classification* and the related concept of *vector representation* of input and output data. *Classification* is the process of assigning *objects* to *categories*. This leads to a first definition of *classification*:

Definition 1 (Classification). *Given a set of objects* O*, and a set of categories* K*, the* classification *of an object from set* O *as belonging to a category from set* K *is defined as a function between elements of the set of objects and elements of the set of categories:*

$$f(o) = k \qquad k \in K, o \in O$$

We do not want to restrict our automatic classification methods to those which only discriminate between a certain kind of objects. We are looking for generic classification methods. For these methods, it does not matter if they are classifying texts or any other kinds of objects. This requires a generic representation of the objects. In this representation, the information necessary to discriminate between the objects must be present. Commonly, one uses *feature vectors* to represent the relevant characteristics of the objects. A feature vector is a vector of numbers. Each characteristic which might contain information relevant for the classification task is represented as a numerical value. We call the vector which corresponds to an object its *feature vector representation*. It is defined in definition 2:

Definition 2 (Feature Vector Representation). *Let* $o \in O$ *be an object from a set of objects, and let* $v_1 \dots v_n \in \mathbb{R}$*. We call the following function* s *the* feature selection function*, and* $v =< v_1, \dots, v_n >$ *the* vector representation *of object* o*:*

$$s(o) = v \quad v =< v_1, \dots, v_n >$$

Since we are not dealing with *text classification* in this chapter, but with *automatic classification* in general, we use an example which has nothing to do with text classification at all. The example task is to classify a vehicle from a set of vehicles as either being a bus, a truck, or a car. The knowledge about the vehicle is limited to three observations:

1. Is the maximum number of allowed passengers above 8?
2. Does the vehicle weight more than 500 kg?
3. Does the vehicle have more than 2 axes?

In our example, one variable represents if the car is allowed to carry more than 8 persons, one variable represents if the car weights more than 500 kg, and one variable represents if the car has more than 2 axes. This set of variables is combined to a *feature vector*.

The transformation of *categories* into a numerical representation is straightforward. We define a mapping between the set of categories and a subset of $\mathbb{N}$:

Definition 3 (Categories). *Let* K *be the set of categories, and* $C = [1, |K|] \subset \mathbb{N}$. *We represent categories by a bijective function* g *with* $g(k) = c \quad c \in C \subset \mathbb{N}, k \in K$

In our example, the three categories could be represented by the integer interval $[0, 2]$, where a possible mapping would be $g(\text{“car”}) = 0$, $g(\text{“bus”}) = 1$, $g(\text{“truck”}) = 2$. Note that the assignment of numbers to category is arbitrary. The only requirement is that the function defines a isomorphic mapping. Therefore it has to be bijective.

By joining definitions 1 to 3, we can define *automatic classification* as the *reduction* of a multi-valued input parameter to an output value:

Definition 4 (Automatic Classification). *Let* O *be a set of objects,* s *a feature selection function on these objects, and* C *a set of representations of categories. Automatic classification is the process of* reducing *vector* $s(o)$ *to* c. *The function* f *with*

$$f(s(o)) = c \qquad o \in O, c \in C$$

is called a classifier.

Output Vectors

In a lot of applications, we do not have only one set of categories, but multiple sets. In the vehicle example, we might not only want to categorize the vehicles according to their types, but also according to their colors. In order to allow multiple classes of categories, we extend the concept of categories. We group together an arbitrary number of category variables into an *output vector*. Thus, the *category variables* themselves are called *output vector components*. We extend definition 4 as follows in order to take the *output vector* concept into consideration:

Definition 5 (Output Vector). *Let* s *be a feature selection function on a set of objects* O*, let* $(C_1, \ldots, C_n)$ *be a set of* n *sets of categories, and let* $f_1 \ldots f_n$ *be* n *classification functions on* $s(o)$*, with*

$$f_i(s(o)) = c_i \qquad o \in O, c_i \in C_i, 1 \leq i \leq n$$

We can combine these classifiers to one classifier f *whose domain is a* vector $\vec{c} =< c_1, \ldots, c_n >$*,*

$$f(s(o)) = \vec{c}$$

Vector $\vec{c}$ *is called an* output vector*.*

Note that the use of the *output vector* format is a mere convenience to group together more than one classification task. The classifiers themselves do not get more (or less) powerful by the use of *output vectors*. In the rest of this part of the thesis, we are not concerned with *output vector* anymore. We will come back to output vectors in the second part of the thesis, where we describe the MIC system.

We have finished the definition of *automatic classification*. In the rest of this chapter, we deal with the classification function $f(o)$. First, a generic categorization of classification methods is given. After that, we introduce the actual classification algorithms which are used within this thesis.

2.2. Classification Methods

We can distinguish between two general approaches used for automatic classification: trainable and non-trainable methods. For trainable methods, the set of categories, and the classification task, are not fixed. Trainable methods are generic methods for classification. They are trained on actual data and categories in order to fulfill a certain classification task. In the training process, examples are presented to the classifier. The classifier extracts information about the structure of the data, and how to classify it, from the training examples. In difference, in non-trainable methods the set of categories, and the way how examples are assigned to categories, are fixed.

2.2.1. Trainable Methods

Within the area of *trainable* methods, we can distinguish even more. One important distinction is between *supervised* and *unsupervised* learning techniques. In *supervised learning* techniques, examples are presented to the classifier together with their classifications. In *unsupervised learning*, classifiers are presented with examples, but with no information about the correct classifications of the examples. In the vehicle example, the training set would consist of a set of vehicles. For unsupervised learning, this is all information the classification algorithm gets. In supervised learning, it gets information about the classification of the vehicles (i.e. whether a vehicle is a car, a bus, or a truck) in the training set.

We define these two kinds of examples:

Definition 6 (Unclassified Examples). *Let* s *be a feature selection function on a set of objects* O. *A subset* E_u *of the set of all feature representations is called* unclassified examples.

$$E_u = \{s(m) | m \in M, M \subseteq O\}$$

Definition 7 (Classified Examples, Reference Classifier). *Let* s *be a feature selection function on a set of objects* O. *Let* C *be a set of categories, and let* r *be a classification function on* $s(o)$. *A subset* E_c *of the set of tuples of feature representations and their classifications is called* classified examples.

$$E_c = \{(s(m), r(s(m))) | m \in M, M \subseteq O\}$$

We call r *the* reference classifier.

When we take learning into account, the classification function changes as follows:

Definition 8 (Learning Classifier). *Let* V *be a set of vector representations of objects. Let* E *be a set of (classified or unclassified) examples, and let* C *be a set of categories. A* learning classifier *is defined as*

$$f(v, E) = c \qquad v \in V, c \in C$$

Definition 9 (Supervised Learning, Unsupervised Learning). *A learning classifier* f *with an example set* E *is* unsupervised learning, *if* E *is a set of unclassified examples.*

A learning classifier f *with an example set* E *is* supervised learning, *if* E *is a set of classified examples.*

Another sophistication in the area of *learning classifiers* lies in the distinction between *incremental learning classifiers*, and *non-incremental learning classifiers*. In *non-incremental learning*, all the training examples are presented in one batch. After the classifier has adjusted itself to this training data, no more learning takes place. In *incremental learning*, the classifier continues to learn even after the first classifications have been performed. We can define incremental learning as refining a hypothesis. The goal of the classification function f_t is the approximation of the reference classifier function f_r. The current configuration of f_t represents one hypothesis about the configuration of f_r. When f_t is confronted with example from f_r which do not match its own classification, it has to refine its hypothesis. So the process of incremental learning can be defined as a search in the space of hypothesises. We will not go deeper into this, because in this thesis, only *non-incremental learning classifiers* are considered.

The tasks we are focusing on in this thesis all require *supervised learning classifiers*. Whether we are classifying web pages, emails or generic documents, we are looking for a system which learns how to classify from examples. Therefore, we are most interested in definition 7. The elements involved in classification based on supervised learning are listed in table 2.1.

T	A set of texts that shall be classified
E	A set of texts used for training
C	A set of categories
$s : T \cup E \rightarrow V_t$	A feature selection function
$f_e : V_t \rightarrow C$	A reference classifier
$f : (V_t, E, f_e) \rightarrow C$	A classification function

Table 2.1.: Elements of Learning Classifiers

For classifiers using supervised learning, the set of example documents is usually quite large. The reliance on examples can be an advantage or a disadvantage of these methods, depending on the situation. It can be an advantage, because learning methods are easily adapted to new classification tasks: Usually, one only has to provide the classifier with a different set of training documents in order to train it on a different task. Relying on training data can be a disadvantage, because the process of training might be a time consuming task. Classifiers using *unsupervised learning* have to discover similarities between documents on their own.

Another disadvantage of learning methods is that most of them are based on statistical analysis methods. A lot of these methods are *black box methods*. The statistical computations that lead to the classification of a document are usually too complex for a human to comprehend. We do not know *why* an object is assigned to the specific category.

2.2.2. Non-trainable Methods

Trainable classifiers use machine learning methods to detect similarities within the set of training examples from one category. Therefore, they are not limited to a specific domain. Whenever there is an adequate *feature selection function*, a trainable classifier can be trained to distinguish between the different classes of objects. This is different for methods which do not rely on training. Non-trainable classifiers are hand-tailored methods for specific tasks, usually based on *expert systems*. In the example of vehicle classification, an expert system has information in the form of rules like "a vehicle with more than two axes is a truck or a bus", and "A vehicle which can carry more than 8 passengers is a bus". The classifier classifies an object by drawing conclusions from these rules. The advantages and disadvantages of non-trainable methods are inverse to those of the trainable methods: Since the rules have to be hand-written, it is more difficult to adapt a non-learning classifier to a new task. On the other hand, the classifier usually does not only *classify* a document, it is also able to *explain* why a document is classified as belonging into a specific category. It can show the rules that lead to the classification of an object.

In this thesis, we will not be concerned with pure non-trainable methods. We will show a method to augment trainable methods with additional non-trained domain knowledge. We discuss our very natural way to combine trainable and non-trainable methods next.

2.2.3. Hybrid Methods

One goal of this thesis is to find methods for the combination of trainable and non-trainable methods. The reason for this comes directly from the tasks we want to solve: In some of the areas where we want to use a classification system, we have heuristics based on non-trainable methods which allow us to classify objects quite well. By combining these non-trainable methods with trainable methods, we expect the accuracy of the classification to improve. We can also expect a classifier which is more adaptable and flexible than a classifier using exclusively non-trainable methods.

Our solution to this problem is to interpret the results of the *non-trainable methods* as additional input features to the *trainable methods*. This fits very well into the definitions of the various components of a classification system laid out in this chapter. Definition 2 defines the input features representation of a text as a vector of real valued components. Definition 3 defines a category as an integer value from an interval. Therefore, we can use the output value of one classifier as a component of the input vector of another classification algorithm:

Definition 10 (Combined Classifier, Hybrid Classifier). *Let* s_1 *and* s_2 *be feature selection functions on a set of objects* O. *Let* f_1 *be a classification function on* s_1, *and let* $s_2(O)$ *be the concatenation of* $f_1(s_1(O))$ *and* $s_1(O)$. *We call the classification function* f_2 *over* s_2 *a* combined classifier.

If the classification algorithms of f_1 *and* f_2 *are different, we call* f_2 *a* hybrid classifier.

Now, all elements of an automatic classification system are defined. The rest of this chapter deals with the automatic classification algorithms actually used. In subsequent chapters, we show how texts are converted into *vector representations*, and evaluate the performance of the algorithms on different text classification tasks.

2.3. Which Classification Methods to use?

There is a large number of classification algorithms for supervised learning classifiers available. The second part of this thesis deals with MIC, the text classification system developed for this thesis. One of the goals in the design of MIC was to be as flexible as possible in regard to the classification algorithms. Still, we have to decide which classification methods to use for our experiments. We base this decision on a comparison of classification methods published in [42]. The following classification algorithms are used: *Neural Network*, *Naive Bayes*, and *Decision Tree*.

Naive Bayes Classifiers are simple yet successful classification methods. They became kind of reference classifiers to measure the performance of other classifiers on various data sets. Naive Bayes Classifiers are used to get general information about the classifiability of a task, and to get reference values to judge the performance of other classifiers.

Classifiers based on Neural Networks score high in the performance measure published in [42]. Neural Networks are especially good at dealing with large sets of categories. For a lot of classifiers, the performance degrades dramatically when the number of categories

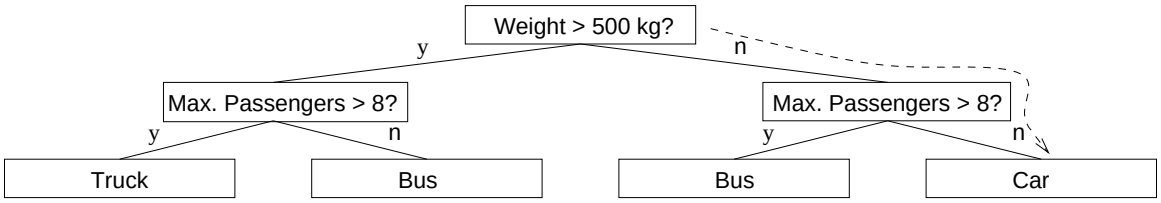

Figure 2.1.: Example of a Decision Tree for vehicle classification example. A vehicle weighting 500 kg with a maximum of 5 passengers is classified as a car.

increases. Neural Networks do not have this weakness. This is an important characteristic, because in at least one of the applications, the email classification task, we have to deal with large sets of categories.

Decision Tree learning is the third classification algorithm we examine in this chapter. In difference to Neural Networks and Naive Bayes classification, Decision Tree learning does not use statistical methods to learn, but rules. This makes Decision Trees very well suited for the evaluation of the classification process. For the statistical methods, we can only draw vague conclusions about the interactions of the stochastic variables. A Decision Tree gives us a clear set of rules for the classification. In a Decision Tree, a human observers can see directly on which features the classification is based.

2.4. Decision Trees

The way Decision Trees are constructed will give us important insights in how the size of the feature space can be reduced. Chapter 3 will explore why the technique used in Decision Tree learning is important for input feature selection.

A Decision Tree uses the following method to classify an object: One feature is picked from the input feature vector at a time. Depending on the value of this feature, the next feature is picked. Again, its value is checked, and depending on the value of this feature, another feature is picked. This process continues until the algorithm has checked enough features to come up with a classification. The process of feature selection and checking, which finally leads to a classification, can be shown graphically in a tree. This is done in figure 2.1 for a downsized version our example problem: We want to classify a vehicle with the features "does not weight more than 500 kg", and "does not carry more than 8 passengers." The classification algorithm starts at the top node of the tree. The first feature checked for is if the vehicle weights more than 500 kg. Since it does not, the algorithm proceeds to the right child of the top node. Here, it checks whether the vehicle carries more than 8 passengers. It does not, so the algorithms chooses the right child of the node. This is a leaf node. The associated classification is "car", so the vehicle is classified as a car.

More generally, in a Decision Tree, each node of the tree is associated with a feature. The links between the nodes represent the possible values of the feature associated with the parent node. Leaf nodes represent categories. Classification is done the following

way: Start with the top-most node. Check the value of the feature associated with this node. Follow the link annotated with the value of the feature. Loop until you reach a leaf node. The classification associated with the leaf node is the classification of the document.

The algorithm for the traversal of a Decision Tree is shown in algorithm 1.

Algorithm 1 Decision Tree Traversal

Require: object: The object to be classified
Ensure: classification: The classification of the object

```
current_node = top_node
while (not (current_node.is_leaf_node)) do
  current_feature = current_node.feature
  object_feature_value = object.feature_value(current_feature)
  current_node = current_node.link(object_feature_value)
end while
classification = classification(current_node)
```

2.4.1. Construction of a Decision Tree

Decision Trees belong to the *supervised learning methods*. At this point, we examine how a Decision Tree can be trained. We use the standard algorithm for Decision Tree learning given in [27, page 403].

The objects which shall be classified are represented as feature vectors of cardinality n. In the limited vehicle example, the feature vector has two variables. One is "weight $>$ 500 kg", the second is "max. passengers $>$ 8". The domain of both vector components is "yes", or "no". For each training example, these vector components have specific values from their domains. There are three categories: "bus", "car", and "truck".

A Decision Tree is constructed recursively. The vector component which gives the most *information* about the classification of the objects is placed at the top most node. The links from this node represent the possible values of this vector component. The direct child nodes of the node are constructed in the same way the top most node is constructed. At this point, the algorithm does not consider the whole set of objects, but only those where the value of the input feature associated with the parent node has the same value as the link.

In order to set this simple algorithm into action, we need a way to calculate the *information content* of a set of objects. The *information* content of a dataset is the inverse of its *entropy*[1]. It is defined in definition 11.

[1] The *entropy* of a random variable X is defined as (see e.g. [21]):

$$H(X) \equiv \sum_{x \in A_x} P(x) \log \frac{1}{P(x)}$$

Definition 11 (Information Content). *Given a set of categories* $c_1, \ldots c_n$*, and the probabilities for an object from the set of objects to belong to these categories as* $P(c_1), \ldots, P(c_n)$*, the information content of this set of categories can be calculated by the equation*

$$\text{Information\_in\_data} = \sum_x -P(x) \times \log_2 P(x)$$

See [27, pages 360].

In the vehicle classification example, the hypothesises are "the vehicle is a car" (c_1), "the vehicle is a bus" (c_2), and "the vehicle is a truck" (c_3). $P(c_i)$ is the probability that for a randomly drawn vehicle from the set of vehicles hypothesis c_i is true. It is calculated as the number of vehicles for which c_i is true (defined via a reference classifier r and a feature selection function s), divided by the total number of vehicles

$$P(c_i) = \frac{|\{o | o \in O, r(s(o)) = c_i\}|}{|O|}$$

The idea of Decision Tree learning is to split the set of objects by the feature giving the most information recursively. The information gain for each of the features is calculated as given in definition 12:

For each value w of feature ν, a subset of the set of objects is created. This subset contains all the objects for which feature vector component ν has value w. In the vehicle example, when ν is the feature vector "max. passengers $>$ 8", two subsets are created. One contains the objects for which "max. passengers $>$ 8" is true, the other contains the objects for which it is false. The information gain for this feature vector component is the cumulated information content of the subsets, weighted by the sizes of the subsets. The difference between the information content of this subset and the information content of the original set is the *information gain* for this feature.

When this calculation is done for each feature, the feature which gives the most information is selected. The most informative feature is associated with the node. Each of the links is labeled with a possible value of the feature. The nodes connected to the links are computed recursively. For each of the linked nodes, the set of texts is the subset of original texts where the feature has the value associated with the link.

Definition 12 (Information Gain). *Let* $I(O)$ *be the information contained in a set of objects. Let* $O_{\nu_j = w}$ *be the subset of* O *for which feature* ν_j *has value* w*. The* information gain $G(O, \nu_j)$ *when splitting* I *on feature* ν_j *is calculated as*

$$G(O, \nu_j) = I(O) - \sum_{w \in \text{dom}(\nu_j)} I(O_{\nu_j = w}) \cdot \frac{|O_{\nu_j = w}|}{|O|}$$

Algorithms 2 shows how a Decision Tree is constructed.

For a human observer, the classification process of a Decision Tree is very simple to follow, as we see in figure 2.1 on page 24. The most important feature of the input data is located at the top of the tree. The less important ones are located at the levels below.

Algorithm 2 Algorithm for the Construction of a Decision Tree

```
Require: T: set of documents
Ensure: D: decision tree
  D = new node
  if (T.information_content == 0) then
    {There is no information left in the set of objects. We are done at this point.}
    D.is_leaf_node = true
    {Since there is no information left, all objects in the belong to the same category.}
    D.category = T.category
  else
    selected_feature = most_informative_feature(T)
    D.feature = selected_feature
    for all (value = selected_feature.domain) do
      {For each possible value of the selected feature}
      T_subset = T.select_by_value(selected_feature, value)
      {Add childs by recursion over the subsets}
      D.add_child(recursion(T_subset), value)
    end for
  end if
```

Given an object and a Decision Tree, a human observer can tell *why* the algorithm gives a certain classification to an object. This constitutes a significant difference to the classification algorithm introduced next. Artificial Neural Networks operate on a *sub-symbolic* level. They have no explicit representation of the classification process of objects. This difference plays an important role in the next chapter, when we examine the *input feature selection function*. From the way a Decision Tree is constructed, we can get insights how to select features in a better way than standard methods do.

2.5. Neural Networks

Artificial Neural Networks are widely used methods for statistical modeling and automatic classification. Neural Networks are very simple computational structures. Despite their simple structure, ANNs are quite powerful. In a lot of fields Artificial Neural Networks belong to the top performing algorithms (see [42]). This is also true for automatic text classification tasks. Therefore, we include Neural Networks into this thesis.

Artificial Neural Networks are computational models which imitate the behavior of the natural neural networks we find in animals and men. Neural Networks are constructed from two basic components: *Nodes* and *Links*. *Nodes* correspond to the cells in natural neural networks, and *links* correspond to the axons. Nodes are interconnected via links. The functional principle of Artificial Neural Networks is simple:

Every node has some amount of energy. When its energy exceeds a certain threshold, the node *fires*. Its energy propagates via the links to all nodes connected to the firing node, thus changing their energy levels. The computation done in a neuron is simple:

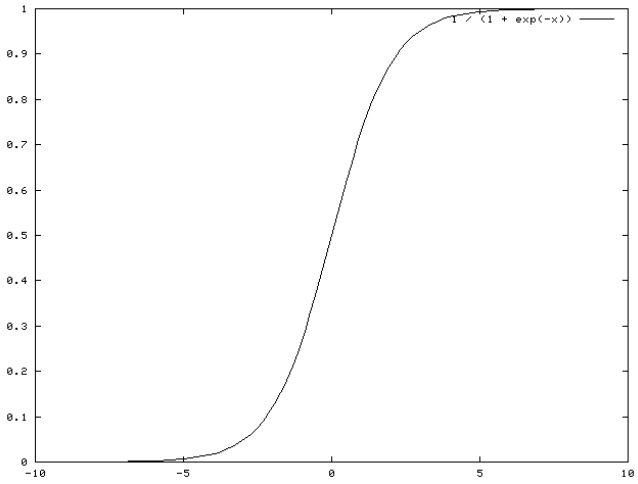

Figure 2.2.: Graph of logistic function

- The current energy level at the input of the node equals the energy output levels of the nodes which connect to the node, weighted by the strengths of the links. Let $y_1 \dots y_J$ be nodes which connect to the node with the strengths $b_1 \dots b_J$. We get equation 2.1 for the nodes input v.[2]

$$v = b_0 + \sum_{j=1}^{J} b_j y_j \tag{2.1}$$

- The output value w of the node is calculate from the input value by an activation function: $w = g(v)$. In natural neural networks, this is a threshold function. When the input activation is above a certain threshold, the output activation is toggled from zero to full activated. The disadvantage of threshold functions for Artificial Neural Networks lies in the problem that threshold functions are not differentiable at all points. Differentiability of the activation function at all points is an essential criteria for finding a mathematical method to train a network, as we will see later in this chapter. Therefore, Artificial Neural Networks do not use threshold functions, but differential functions who are similar to threshold function. These are usually sigmoid functions. Sigmoid functions are bounded, monotonic increasing, and differentiable. A commonly used sigmoid function is the logistic function:

$$g(v) \equiv \frac{1}{1 + \frac{1}{e^v}} \tag{2.2}$$

  The graph of the logistic function is shown in figure 2.2.

[2] In this equation, b_0 is the bias node. The bias node always has full activation. For a discussion of the function of the bias node, see [33, page 36].

Network topology

There are numerous variants of Artificial Neural Networks with different topological orders available. For our automatic classification purposes, we use *back propagation feed-forward networks*. Multi layer backpropagation networks consist of an input layer, where the network receives the feature representation of the input data, and an output layer which gives the results of the network's calculations. In between these layers are a number of hidden layers. It is has been shown in [13] that Neural Networks with one hidden layer are computationally as powerful as networks with more than one hidden layer. Therefore we will refrain from using more than one hidden layer. There are numerous ways how the nodes can be connected. We will use a standard *fully-connected* feed forward network without *short-cuts*. This means, every node in a layer is connected to every node in the next layer, and to no other nodes.

Operating the Network

Figures 2.3 to 2.5 give an example of how the vehicle classification problem can be solved by a neural network.[3] The number of input nodes of the network must be equivalent to the arity of the feature vector, and the number of output nodes must match the number of categories. In this example, the network is already *trained* for the vehicle classification task. In the classification process, the value of each vector component is layed on the corresponding input node. The input values of the hidden nodes are calculated by equation 2.1, and the output activations of the nodes are calculated by equation 2.2 (figure 2.4). The same is applied to the output nodes. Finally, the document is assigned to the category whose output node has the highest output value (Winner Takes All).

2.5.1. Training

A Neural Network has to be trained before it can be used for classification. There are three different classes of training methods for Neural Networks: *supervised learning methods*, *reinforcement learning methods*, and *unsupervised learning methods*. In supervised learning method, a set of preclassified *training data* is used to adjust the weights of the network. After the training phase, the Neural Network can classify objects. When an object is presented to the network, the networks determines the similarity of the object to the categories of objects it had been trained on, and chooses the most similar category. In reinforcement learning, the training phase is different. The Neural Network does not get explicit information which class an object belongs to. The network tries to classify the object, and gets information if the classification was correct or incorrect[4]. The third class of training methods are those that have no feedback at all. The network is completely on its own in finding categories and classifying the data. In general, we can expect best results with supervised learning techniques, followed by the results achieved

[3] Note that these figures contain some simplifications in order to reduce the complexity of the network structure: There are only two input features and two categories (in difference to three in the original example), and the network structure is simplified by ignoring the *bias node*.

[4] This has not to be binary information, it can also be a feedback to which degree a classification was correct.

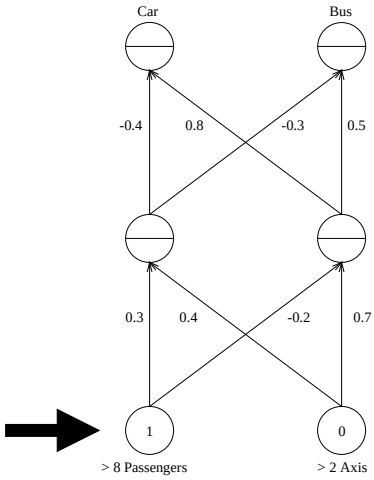

Figure 2.3.: Initial Situation. The vehicle we want to classify has more than 8 passengers and two axes. Therefore, the input node representing "more than 8 passengers" gets an activation value of 1, and the node representing "more than two axes" gets an activation value of 0.

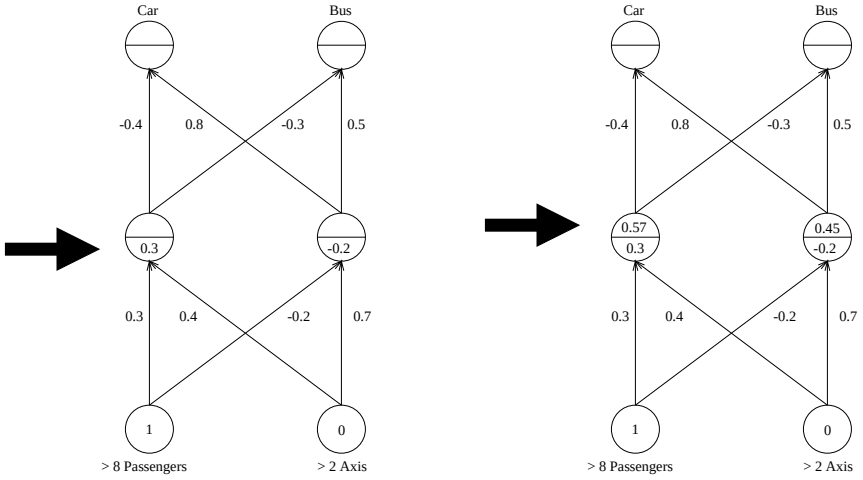

Figure 2.4.: The left graphic shows the network in the second step: The input values of the hidden nodes are calculated as $1 \times 0.3 + 0 \times 0.4$ for the left hidden node, and $1 \times -0.2 + 0 \times 0.7$ for the right hidden node. The right graphic shows the network in the third step: Calculation of the output values of the hidden nodes. The output values are calculated from the input values by the logistic function $g(v) \equiv \frac{1}{1+\frac{1}{e^v}}$.

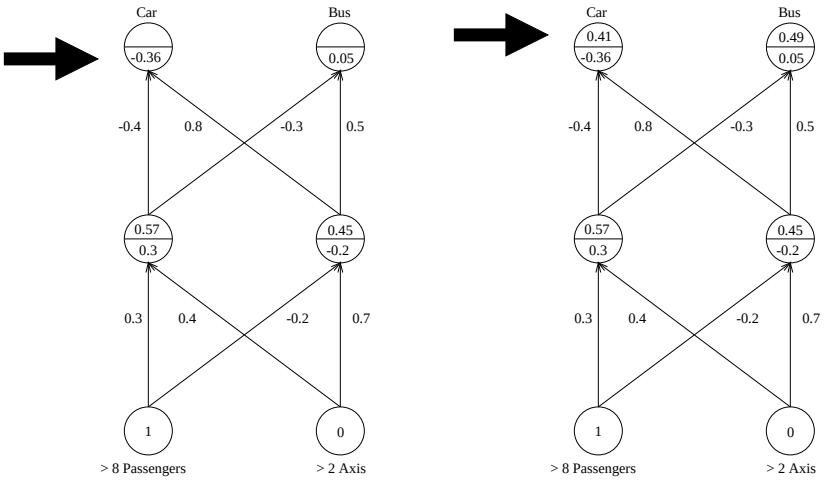

Figure 2.5.: The left graphic shows the network in the fourth step: The input values of the output nodes are calculated as $0.57 \times -0.4 + 0.45 \times 0.8$ for the left output node, and $0.57 \times -0.3 + 0.45 \times 0.5$ for the right output node. The right graphic shows the network in the fifth step: The output values of the output nodes are calculated. The vehicle is classified as belonging to the class which is associated with the output node with the highest activation. In this example, it is classified as belonging in class "bus".

by reinforcement techniques. Unsupervised learning techniques are mainly used when one has no information about the number and kinds of categories. See for example [12]. The only kind of methods we are interested in are supervised learning techniques. Next, we will examine a training algorithm for supervised learning networks.

During the training phase, the *weights* at the links between the nodes are adjusted. Apparently, the results of the network's calculations depend on the values of the connecting weights. How can these weights be trained? The training phase starts with randomized weights. In each training step, the classification error for the training examples is calculated. With this information, a gradient descent in the error surface is performed. In other words, from the experience gained so far, the weights are changed to minimize the classification error on the training examples.

In order to adjust the weights, we have to know how much each of the weights b is responsible for the network's error E: $\frac{\partial E}{\partial b}$. This can be calculated by using the well known chain rule [26, page 150] of calculus:

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial v} \cdot \frac{\partial v}{\partial b} \tag{2.3}$$

In this equation, E is the node's output error. b is the weight in question. z is the output value of the node, t is the correct output value, and v is the node's input. Each term on the right side can be calculated fairly simple:

$$E = \frac{1}{2}(z-t)^2 \tag{2.4}$$

$$\frac{\partial E}{\partial z} = z - t \tag{2.5}$$

$$\frac{\partial z}{\partial v} = z \cdot (1-z) \tag{2.6}$$

$$\frac{\partial v}{\partial b_j} = y_j \tag{2.7}$$

In the last equation, y_j is the output of the hidden nodes connected to this weight. We get the following equation for the derivation of the error in respect to a specific weight:

$$\frac{\partial E}{\partial b} = (z-t) \cdot z \cdot (1-z) \cdot y_j \tag{2.8}$$

This equation holds only for weights connecting hidden nodes to output nodes. For the weights connecting input nodes to hidden nodes, the equation gets a bit more complicate. For the output nodes, the target values t are part of the training data. The target values for the hidden nodes are not directly available. They have to be calculated. We start with the following equation, which is very similar to equation 2.3:

$$\frac{\partial E}{\partial a} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial a} \tag{2.9}$$

E is the node's output error. a is the weight in question. y is the hidden node's output, and u is the hidden node's input. The second and third link in this chain are calculated the same way as for output nodes. The only difference lies in the first part of the chain, the derivation of the error in respect to the hidden node's output. This is the hidden nodes part of the error of all output nodes. We calculate this number by the chain rule again:

$$\frac{\partial E}{\partial y} = \sum_{k=1}^{K} \frac{\partial E}{\partial z_k} \cdot \frac{\partial z_k}{\partial v_k} \cdot \frac{\partial v_k}{\partial y} \tag{2.10}$$

We get

$$\frac{\partial E}{\partial y} = \sum_{k=1}^{K} (z_k - t_k) \cdot z_k \cdot (1-z_k) \cdot b_k \tag{2.11}$$

Thus, the following equation calculates the error of a weight connecting an input node to a hidden node:

$$\frac{\partial E}{\partial a_i} = \left(\sum_{k=1}^{K} (z_k - t_k) \cdot z_k \cdot (1-z_k) \cdot b_k \right) \cdot y \cdot (1-y) \tag{2.12}$$

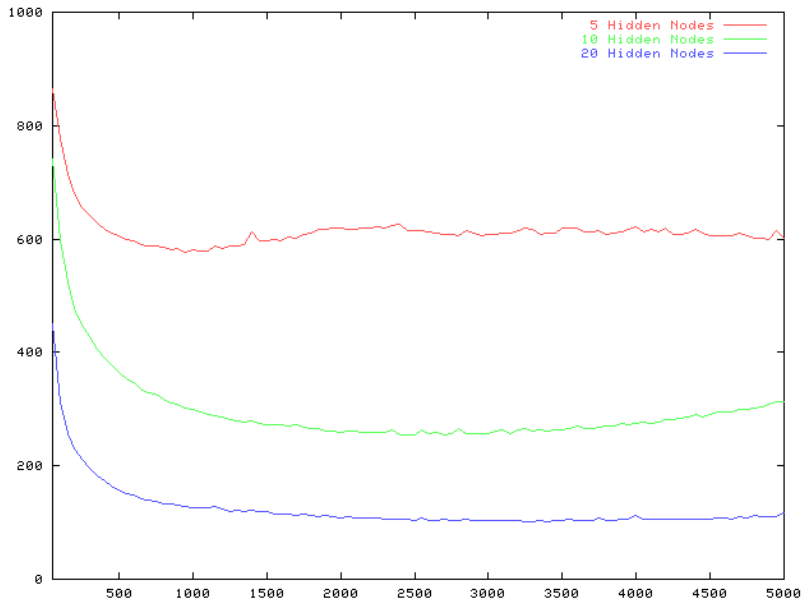

Figure 2.6.: The squared error of the output nodes of a network (y-axis) over 5000 training cycles (x-axis). The three graphs show the performance of networks with 5 (red), 10 (green), and 20 (blue) hidden nodes.

In every learning step n, each weight is changed by the inverse of its contribution to the total error, weighted by a *learning rate* ε:

$$a_{in+1} = a_{in} - \frac{\partial E}{\partial a_{in}} \cdot \varepsilon \tag{2.13}$$

$$b_{in+1} = b_{in} - \frac{\partial E}{\partial b_{in}} \cdot \varepsilon \tag{2.14}$$

The learning rate determines how fast the network's weights are adjusted. A slow learning rate leads to a slow learning process. If the learning rate is set too high, the network does not learn at all.

There is no general rule to decide how many nodes in the hidden layer to use. With too few nodes, the network will not learn well. With too many nodes, the learning process might be slowed down, and there is a risk of *overfitting*. Overfitting means that the Neural Network does not learn the general characteristics of the different categories, but "memorizes" the training data, thus decreases the performance on data which it has not seen in the training process. Figure 2.6 shows the influence of the number of hidden nodes on the performance of a Neural Network. The graph shows the squared error of the output nodes over 5000 cycles of training.[5] The error rate for Neural Networks with only 5 or 10 hidden nodes stay high, because the number of hidden nodes is too small.

[5] For this graphic, the newsgroup dataset from chapter 4.1.3 is trained with 65 input nodes, selected from the ranking of a Decision Tree. The learning rate is set to 0.2.

The latter two problems are not really problems within the context of our applications. In general, speed of training is not an important aspect of the system at this point. In all our applications (see chapter 1), the classifiers are trained *offline*. They are trained before they are used for real tasks. It is acceptable to have a longer training period once, as long as the trained classifiers are fast. Therefore, the time of training does not really matter. Overfitting is not a problem, because there is a very simple solution to this problem: We evaluate the performance of the network during the training phase not on the training data itself, but on a second dataset, on which the network is not trained. We stop the training as soon as the performance of the network drops on this independent dataset, thus preventing overfitting.

2.6. Naive Bayes Classifier

Classifying

Bayes classifiers are based on probability theory. The task of classifying an object is understood as finding the most probable hypothesis, given some evidence. In the vehicle example, the hypothesises are "the object is a car" (c_1), "the object is a bus" (c_2), and "the object is a truck" (c_3). Without looking at the actual object, we can only give unconditional probabilities $P(c_i)$ for the hypothesises. The unconditional probability $P(c_i)$ for a class c_i gives the probability that a randomly drawn object from the set of all objects belongs in class c_i.

In our example, the feature vector has three components, v_1, v_2, and v_3. These features tell about the weight of the vehicle (v_1), the number of axes (v_2) of the vehicle, and the maximal number of passenger (v_3) of the vehicle. For example, "the weight of the vehicle is less than 500 kg" (w_1), "it has only two axes" (w_2), and "it can carry at most 5 passengers" (w_3). Now we calculate the *maximum a posteriori (MAP)* hypothesis given the evidence:

$$c_{MAP} \equiv \text{argmax}_{c \in C} P(c | v_1 = w_1 \wedge \ldots \wedge v_n = w_n) \tag{2.15}$$

Training

The behavior of the classifier depends on the probability distributions of the random variables. In order to train the classifier, we have to calculate the distribution of $P(c|v_1 = w_1 \wedge \ldots \wedge v_n = w_n)$ from the set of training examples. We use *Bayes Theorem* for this calculation. The definition of *Bayes Theorem* is given in definition 13.

Definition 13 (Bayes Theorem). *Let h be a hypothesis, $v =< v_1, \ldots v_n >$ an input feature vector, and $D =< w_1, \ldots w_n >$ the vector feature representation of an object.*

$$P(c|v_1 = w_1 \wedge \ldots \wedge v_n = w_n) = \frac{P(v_1 = w_1 \wedge \ldots \wedge v_n = w_n|c) \cdot P(c)}{P(v_1 = w_1 \wedge \ldots \wedge v_n = w_n)}$$

When we apply *Bayes Theorem* to equation 2.15, we get:

$$c_{MAP} \equiv \text{argmax}_{c \in C} \frac{P(v_1 = w_1 \wedge \ldots \wedge v_n = w_n|c) \cdot P(c)}{P(v_1 = w_1 \wedge \ldots \wedge v_n = w_n)} \tag{2.16}$$

Since $P(v_1 = w_1 \wedge \ldots \wedge v_n = w_n)$ is the same for all hypothesises, we can simplify this equation to

$$c_{MAP} \equiv \text{argmax}_{c \in C} P(v_1 = w_1 \wedge \ldots \wedge v_n = w_n | c) \cdot P(c) \tag{2.17}$$

The calculation of the unconditional probability for a hypothesis is straightforward. Given the total number of examples $|Examples|$, and $examples_j$, the subset of $Examples$ for which hypothesis c_j holds, $P(c_j)$ can be calculated as

$$P(c_j) = \frac{|examples_j|}{|Examples|} \tag{2.18}$$

$P(v_1 = w_1 \wedge \ldots \wedge v_n = w_n | c)$ remains to be calculated. We could solve this equation using the well known chain rule[6] of stochastic calculation:

$$\begin{aligned} P(v_1 = w_1 \wedge \ldots \wedge v_n = w_n) \quad = \quad & P(v_1 = w_1) \times \\ & P(v_2 = w_2 | v_1 = w_1) \times \\ & P(v_3 = w_3 | v_1 = w_1 \wedge v_2 = w_2) \times \\ & . \\ & . \\ & . \\ & P(v_n = w_n | v_1 = w_1 \wedge \ldots \wedge v_{n-1} = w_{n-1}) \end{aligned} \tag{2.19}$$

This possibility is only theoretical, though, because we do not know the conditional probabilities of the chain links. Things get a lot easier, when we assume a conditional independence of $v_1 \ldots v_n$. For conditional independent variables, the following equation holds:

$$P(a \wedge b | c) = P(a|c) \cdot P(b|c) \tag{2.20}$$

Assuming conditional independence, $P(v_1 = w_1 \wedge \ldots \wedge v_n = w_n | c)$ can be calculated as given in definition 14.

Definition 14 (Naive Bayes Classifier). *Let h be a hypothesis, $v =< v_1, \ldots, v_n >$ an input feature vector, and $D =< w_1, \ldots, w_n >$ the vector feature representation of an object.*

$$P(D|c) = \prod_{i=1}^{n} P(v_i = w_i | c) \times P(c)$$

We call this *Naive* Bayes classification, because a conditional independence of the features is usually *not* given. This is definitely the case for text classification. Still, Naive Bayes classification works surprisingly well for a lot tasks (see [7]).

[6] Quite a lot of chain rules... This one is the chain rule of *stochastic*, not of *calculus*, as in the section about Neural Networks. Also, this time the trick is *not* to use the chain rule.

2.6.1. Calculating Feature Probabilities

In order to use the Naive Bayes classifier from definition 14, we have to calculate $P(v_i = w_i|c)$ for every possible value of every feature. This calculation is very similar to equation 2.18, where the unconditional probability of a category is calculated as the number of examples in the category, divided by the total number of examples in all categories.

Let E_c be the set of examples belonging in category c. For each feature v_i, and for each possible value w_i, $P(v_i = w_i|c)$ can be calculated as the number of the examples from E_c for which $v_i = w_i$, divided by the total number of examples in E_c. Equation 2.21 shows how to calculate the feature probability.

$$P(v_i = w_i|c) = \frac{|\{e|e =< v_1, \ldots, v_n >\in E_c, v_i = w_i\}| + 1}{|E_c|} \tag{2.21}$$

The addition of 1 to the numerator is necessary to avoid zero probabilities in the cases where the numerator would be zero otherwise. A zero probability is not desirable, because when inserted into the Naive Bayes classification formula given in definition 14, it would result in zero probability for $P(D|c)$. In this case, all documents in which a feature value appears for which there had been no example in the training set, would be assigned zero probability.

Example

Again, we use the vehicle classification problem as an example for the classification method. It should be noted that Naive Bayes classifiers are not well suited for this kind of examples: Naive Bayes classifiers deal with probabilities. Depending on the characteristics of the object to be classified, it gets more or less *probable* that the object belongs in a category. When classifying vehicles, the rules for classification are rather strict. Therefore, it does make more sense to use a strictly rule based classifier like a *Decision Tree* in such an area.

Back to the example. We assume the training data set yields the following probabilities:

$$\begin{aligned}
P(\text{truck}) &= 0.2 \\
P(\text{bus}) &= 0.3 \\
P(\text{car}) &= 0.5
\end{aligned}$$

$$\begin{aligned}
P(\text{“}< 500 \text{ kg”} = \text{false} \,|\text{truck}) &= 0.9 \\
P(\text{“}< 500 \text{ kg”} = \text{true} \,|\text{truck}) &= 0.1 \\
P(\text{“}> 2 \text{ axes”} = \text{false} \mid \text{truck}) &= 0.3 \\
P(\text{“}> 2 \text{ axes”} = \text{true} \mid \text{truck}) &= 0.7 \\
P(\text{“}> 8 \text{ passengers”} = \text{false} \mid \text{truck}) &= 0.85 \\
P(\text{“}> 8 \text{ passengers”} = \text{true} \mid \text{truck}) &= 0.15
\end{aligned}$$

$$
\begin{aligned}
P(\text{“}< 500 \text{ kg”} = \text{false} \mid \text{bus}) &= 0.8 \\
P(\text{“}< 500 \text{ kg”} = \text{true} \mid \text{bus}) &= 0.2 \\
P(\text{“}> 2 \text{ axes”} = \text{false} \mid \text{bus}) &= 0.25 \\
P(\text{“}> 2 \text{ axes”} = \text{true} \mid \text{bus}) &= 0.75 \\
P(\text{“}> 8 \text{ passengers”} = \text{false} \mid \text{bus}) &= 0.05 \\
P(\text{“}> 8 \text{ passengers”} = \text{true} \mid \text{bus}) &= 0.95
\end{aligned}
$$

$$
\begin{aligned}
P(\text{“}< 500 \text{ kg”} = \text{false} \mid \text{car}) &= 0.4 \\
P(\text{“}< 500 \text{ kg”} = \text{true} \mid \text{car}) &= 0.6 \\
P(\text{“}> 2 \text{ axes”} = \text{false} \mid \text{car}) &= 0.9 \\
P(\text{“}> 2 \text{ axes”} = \text{true} \mid \text{car}) &= 0.1 \\
P(\text{“}> 8 \text{ passengers”} = \text{false} \mid \text{car}) &= 0.85 \\
P(\text{“}> 8 \text{ passengers”} = \text{true} \mid \text{car}) &= 0.15
\end{aligned}
$$

Now we want to classify the probability for a 2-axes vehicle weighting less than 500 kg with maximum 5 passengers. We calculate the probabilities for the three hypothesises “vehicle is a truck” (c_1), “vehicle is a bus” (c_2), and “vehicle is a car” (c_3):

$$
\begin{aligned}
P(D|c_1) &= \prod_{i=1}^{3} P(v_i = w_i|c_1) \\
&= P(\text{“}< 500 \text{ kg”} = \text{true} \mid \text{truck}) \\
&\quad \times P(\text{“}> 2 \text{ axes”} = \text{false} \mid \text{truck}) \\
&\quad \times P(\text{“}> 8 \text{ passengers”} = \text{false} \mid \text{truck}) \\
&= 0.1 \times 0.3 \times 0.85 \\
&= 0.0255
\end{aligned}
\tag{2.22}
$$

$$
\begin{aligned}
P(D|c_2) &= \prod_{i=1}^{3} P(v_i = w_i|c_2) \\
&= P(\text{“}< 500 \text{ kg”} = \text{true} \mid \text{bus}) \\
&\quad \times P(\text{“}> 2 \text{ axes”} = \text{false} \mid \text{bus}) \\
&\quad \times P(\text{“}> 8 \text{ passengers”} = \text{false} \mid \text{bus}) \\
&= 0.2 \times 0.25 \times 0.05 \\
&= 0.0025
\end{aligned}
\tag{2.23}
$$

$$
\begin{aligned}
P(D|c_3) &= \prod_{i=1}^{3} P(v_i = w_i|c_3) \\
&= P(\text{“}< 500 \text{ kg”} = \text{true} \mid \text{car}) \\
&\quad \times P(\text{“}> 2 \text{ axes”} = \text{false} \mid \text{car}) \\
&\quad \times P(\text{“}> 8 \text{ passengers”} = \text{false} \mid \text{car}) \\
&= 0.6 \times 0.9 \times 0.85 \\
&= 0.459
\end{aligned}
\tag{2.24}
$$

From equations 2.22 to 2.24 we get the highest probability for hypothesis c_3. The vehicle is a car.

2.7. Summary

Three classification algorithms are used in this thesis: *Decision Trees*, *Naive Bayes* classifiers, and *Neural Networks*. *Neural Networks* and *Naive Bayes Classifiers* are chosen because they belong to the top-performing classification algorithms. Additionally, *Naive Bayes Classifiers* are kind of reference classifiers because of their simplicity and performance. *Decision Tree Classifiers* are used because they follow a different paradigm than the other two classifiers. They are *rule based* in difference to the *statistical classifiers* *Naive Bayes* and *Neural Networks*. The construction of a Decision Tree will give us insights into how to improve features selection.

3. Input Feature Selection

In this chapter, we show how the generic classification methods from the last chapter can be applied to text classification problems. We define *text* and related concepts, and show how a text can be converted into a *feature vector*. After a summary of commonly used methods to represent texts as *feature vectors*, we introduce the methods specifically devised for this thesis. Most of these special methods take advantage of the additional *structure information* provided with HTML-, and email-documents.

3.1. Basic Concepts

We define the formal elements of text classification bottom-up by first defining *word* in definition 15. Based on this, we define *text* in definition 16, and finally *vector representation of a text* in definition 17.

Definition 15 (Alphabet, Word). *An* alphabet Σ *is a finite, non-empty set. The elements of an alphabet are called* letters.

A word w *over* Σ *is a finite, possibly empty sequence of letters.* $w = (a_1, \ldots, a_n)$ *is also written as* $w = a_1 \ldots a_n$. *The* empty word, *the word with 0 letters, is written as* ε. $|w|$ *is the* length *of word* w, *i.e. the number of letters of* w. *(cited from [4, page 27].)*

Definition 16 (Text). *A* text t *is an ordered set of* words $w_1 \ldots w_n$. *We usually write a* text *as the concatenation of the words. In the latter representation, the words are separated from each other by one or more* whitespace characters. Whitespace characters *are characters from an* alphabet *which is disjunct to the* alphabet *from which the* words *are formed.*

Automatic classification as defined in definition 4 on page 19 takes a vector $< v_1, \ldots, v_n >$ as its input. Therefore, the texts have to be transferred into their vector representation before they can be classified.

Definition 17 (Vector Representation of a Text). *Let* $t \in T$ *be a text from a set of texts, let* $v_1 \ldots v_n \in \mathbb{R}$, *and let* $n \in \mathbb{N}$. *We call the following function* s *the* feature selection function, *and* $v =< v_1, \ldots, v_n >$ *the* vector representation *of the text:*

$$s(t) = v \quad t \in T, v =< v_1, \ldots, v_n >, v_i \in \mathbb{R}, 1 \leq i \leq n$$

Thus the classification function f in regard to a set of categories C can be written as:

$$f(s(t)) = c \qquad t \in T, s(t) =< v_1, \ldots, v_n >, v_i \in \mathbb{R}, 1 \leq i \leq n, n \in \mathbb{N}, c \in C \qquad (3.1)$$

3.1.1. Transforming Texts Into Features

Nearly all generic text classification methods which do not use sophisticated linguistic methods use the same basic method to convert a text into a feature vector (see for example [25, page 183]). Using the frequencies of the words appearing in the example texts as features, $s(t)$ is defined in the following way:

Definition 18 (Word-Feature Selection). *Let $t_1, \ldots, t_n$ be the set of example texts. Let W_i be the set of all distinct words in t_i, and let W be the set of all distinct words in all texts: $W = \cup_{1 \leq i \leq n} W_i$.*

Define a bijective function $wo : [1, |W|] \rightarrow W$ which gives an arbitrary, isomorphic mapping of $[1, |W|] \in \mathbb{N}$ to the words in the training set. Define another function $no : (i, [1, |W|]) \rightarrow \mathbb{N}$ which gives the frequency of a word $wo^{-1}(w), w \in W$, in text t_i.

Let v be the feature vector *with $v =< v_1, \ldots v_w >$. The values of the feature vector components $v_j, 1 \leq j \leq w$ for text t_i is defined by function*

$$v_j = \frac{no(i,j)}{|W_i|}$$

Function wo has to be isomorphic, because in the calculation of the feature vectors, we need a mapping from the words to their index numbers as well as mapping from the index numbers to the words.

Example

We transfer the following text t into its vector representation:

```
In Ulm und um Ulm und um Ulm herum
```

We assume that the five distinct words in the text are indeed the only words in the language. The isomorphic function wo can be defined as $wo(1) =$ "In", $wo(2) =$ "Ulm", $wo(3) =$ "und", $w(4) =$ "um", $w(5) =$ "herum". With this setting, no is defined as $no(1, wo^{-1}(\text{"In"})) = 3$, $no(1, wo^{-1}(\text{"Ulm"})) = 3$, $no(1, wo^{-1}(\text{"und"})) = 2$, $no(1, wo^{-1}(\text{"um"})) = 2$, $no(1, wo^{-1}(\text{"herum"})) = 1$.

The vector feature representation of this text can now be calculated as

$$\begin{aligned} s(t) &= \left\langle \frac{no(1,1)}{|W_1|}, \frac{no(1,2)}{|W_1|}, \frac{no(1,3)}{|W_1|}, \frac{no(1,4)}{|W_1|}, \frac{no(1,5)}{|W_1|} \right\rangle \\ &= \left\langle \frac{1}{9}, \frac{3}{9}, \frac{2}{9}, \frac{2}{9}, \frac{1}{9} \right\rangle \\ &= \langle 0.11111, 0.0.33333, 0.22222, 0.22222, 0.11111 \rangle \end{aligned}$$

Feature Reduction

This method of transferring a text into a feature vector is based on the assumption that the category to which a text belongs is correlated to the relative frequency of the words in the text. This is obviously a simplification. There are several commonly used methods to make the feature representation of a text more accurate. These methods have in common that they reduce the size of the feature vector, either by mapping more than one word to one feature vector component, or by removing the feature vector components for certain words. (See for example [19]).

There are a number of reasons for limiting the number of features. Mapping more than one word onto one feature can make the representation more accurate. For example, in natural languages, there are different spellings for a noun depending on its case. The frequency of a noun is represented more adequately if there is only one feature vector component for the noun, and not one feature vector component for every case of it. Beside the better text-representation, there is a technical reason to reduce the number of features: Using the frequencies of all words as features leads to an immense large number of features, where most of the features (i.e. the words) have little or no significance for classifying. Usually, a large set of features from which most have no meaning decreases the accuracy of the classification. It can also make the computation of the classification of a document significantly slower. For Naive Bayes classification, where the probability of each feature is multiplied, the complexity of the calculation increases linear with the number of features. For Neural Networks, each input node (representing one feature) is connected to every node in the hidden layer. So with every feature added, the number of calculation necessary increases by the number of hidden nodes!

The most common methods for feature space reduction are:

- Convert all words to lower case and remove special characters.
- Remove *stop words*. Stop words are words which appear frequently but provide no information for classifications. Table 3.1 shows a little fragment of the list of German stop words used in MIC.
- Define a minimal and maximal frequency for the words that shall be considered. If a word appears less than the minimal number or more than the maximal number, it is ignored. The effect of this filter is similar to the removal of stop words: We assume that the most and the least frequent words do not contain any information.
- Do *word stemming*. *Word stemming* reduces words to their stems, thus mapping the different spellings of a word depending on its case, temporal form, or mode to a single feature.

The results in chapter 4 show that these common methods already improve the classification of the texts from various domains. To get better results, we have devised some special methods for this thesis. We are especially interested in classifying emails and web pages. These documents are not plain text, but also contain structural information.

```
ab, aber, ähnlich, alle, allein, allem, aller, alles, allg, allgemein,
als, also, am, an, and, andere, anderes, auch, auf, aus, außer, been,
bei, beim, besonders, bevor, bietet, bis, bzw, da, dabei, dadurch, dafür,
daher, dann, daran, darauf, daraus, das, daß, dass, davon, davor, dazu,
dem, den, denen, denn, dennoch, ...
```

Table 3.1.: Fragment of the list of German stop words

```
<html>
  <head>
    <title>A simple HTML page</title>
  </head>
  <body>
    <h1>First Section</h1>
    <p>This is a minimal HTML page</p>
  </body>
</html>
```

Figure 3.1.: A simple HTML page

3.2. Special Methods Devised for MIC

The goals of this thesis are two-folded: One goal is the development of an application that fulfills the text classification requirements given in chapter 1. The second goal is to improve existing text classification methods. We do not improve the automatic classification algorithms themselves. Our improvements take place at the input feature selection level. One aspect of the improvements is a better method to select the features from the set of potential features. The second improvement is a way to preserve structural information when converting texts into feature vectors.

3.2.1. Structured Text

In chapter 1, we state that one of our goals is the exploitation of the additional features *structured text* has in difference to *plain text*. So far, we have talked about methods to classify text files. When converting a text into an input vector, character sequences separated by white spaces are treated as words. This is a sensible approach when we deal with "pure" text. In the area we want to use our text classification system, we do not always deal with plain texts. Two main areas of applications of the methods developed in this thesis is the classification of email and web pages.

Email, as well as web pages, contain structural information. Structured text is not simply an ordered set of words, but an ordered set of words with additional information about the words. A typical example for a structured text is a HTML-page. Figure 3.1 shows a very simple example for a HTML-page. The words embraced in "<" and ">" are *tags*. The word in between the braces is the name of the tag. If the name is preceded by

```
From: bthomas@uni-koblenz.de
To: gb@uni-koblenz.de
Subject: Re: Dokumentation von MIA

klasse,
ich denke wir sollten aber einmal kurz uns gedanken machen wie und wo
wir das in den datei-baum einziehen ... nach dem essen?
```

Figure 3.2.: A typical email message

a "/" it is a *closing tag*, otherwise it is an *opening tag*. All words in between an *opening tag* and a *closing tag* are affected by this tag. Thus, in figure 3.1, the first word, "A", is structured by the list of tags *(html, head, title)*, and the word "minimal" is structured by the tags *(html, body, p)*. Note that these are *ordered* lists of structural tags. *(html, body, p)* is different to *(html, p, body)*.[1] Thus, we come to the following definition:

Definition 19 (Structured Text). *In* structured texts*, each word is associated with an ordered list of zero or more* structure tags*, which themselves are* words *on the same or a different* alphabet.

This definition does not only work for HTML pages, but also for other kinds of structured texts. Next, we apply definition 19 to the simple email message shown in figure 3.2. An email message has at least two distinct parts: A *header* and a *body*. We can assume that the classification results will improve when these elements are treated separately. In general, we can assume that the header information should have more weight than the information in the body. For example, most classification schemes will rely heavily on the sender and recipient of an email message. This information is given in the header. Therefore the names appearing in the header should be treated separately from the names in the body. In an email message, some of the structural tags are obvious while others are not. We see the structural tags `From:`, `To:`, and `Subject:`. Additionally, all of the upper parts of the email, the lines before the empty line, belong to the *header* of the message, whereas all of the lines below the empty line belong to the *body*. By applying definition 19, we get the representation shown in figure 3.3.

Definition 20 gives a method to convert between structured text and plain text.

Definition 20 (Transformation of Structured Text into Plain Text). *Let* t *be a structured text of* n *words together with their tags. Each word* w_i *is affected by* m_i *tags* $(s_{i1} \dots s_{im_i})$*:*

$$t = ((s_{11}, \dots, s_{1m_1}, w_1), \dots, (s_{n1}, \dots, s_{nm_n}, w_n))$$

Let "_" be a separator character *which is not part of the alphabets of the words or the tags. We can transform a structured text into a plain text by concatenating the structure tags and the word of a tuple, where the elements are separated by the whitespace character:*

$$t = (s_{11}\_ \dots \_s_{1m_1}\_w_1 \ \dots \ s_{n1}\_ \dots \_s_{nm_n}\_w_n)$$

[1] Actually, the latter is a violation of the HTML specification.

```
(((header, From:), bthomas@uni-koblenz.de)
 ((header, To:), gb@uni-koblenz.de)
 ((header, Subject:), Re:)
 ((header, Subject:), Dokumentation:)
 ((header, Subject:), von)
 ((header, Subject:), MIA)
 ((body), klasse,)
 ((body), ich)
 ((body), denke)
 ((body), wir)
 ...
)
```

Figure 3.3.: Definition 19 applied to the email message shown in figure 3.2

The method described in definition 20 converts between the structured format and the plain format without information loss. We give an inductive proof:

Proof. Let w be a word together with its structural tags: $v_s = (s_1, \ldots, s_n, w)$.

$n = 0$:
Both the structured version v_s and the plain version v_p are identical $v_s = v_p = (w)$

$n \to n + 1$:
$v_{s_n} = (s_1, \ldots, s_n, w)$, $v_{p_n} = (s_1\_ \ldots \_s_n\_w)$.

$v_{s_{n+1}} \to v_{p_{n+1}}$:
Append _ after s_{n+1} and insert it between s_n_, and w. The result is $v_{p_{n+1}} = (s_1\_ \ldots \_s_n\_s_{n+1}\_w)$.

$v_{p_{n+1}} \to v_{s_{n+1}}$:
Separate element s_{n+1} from $v_{p_{n+1}}$. s_{n+1} can be uniquely identified because according to definition 20, _ is neither part of the alphabet of the tags, nor part of the alphabet of the words. Insert element s_{n+1} as the last but one object in v_{s_n}. This results in $v_{s_{n+1}} = (s_1, \ldots, s_n, s_{n+1}, w)$.

□

The drawback of this method is the explosion of words and thus potential features. Every configuration of feature tags and words gets its own feature. This contradicts all

the methods to improve the feature selection function mentioned before in this chapter, which *reduce* the number of features.

This is alleviated a bit by the methods to select only the most relevant features that are shown next, but still there is no easy solution to this problem. In order to examine the effects of the increased number of features when taking structure into consideration, we run the text classification experiments in three different configurations:

- Treat structured text as plain text.

  The simplest way to deal with the special data formats of HTML files and email is to ignore the additional data and convert them into traditional text files. For HTML pages this means to remove all tags from the documents. Emails are simply treated as plain text, without considering the special meanings of the headers and bodies of the message.

- Use only the tags on the outer level for classification.

  As a compromise between the inclusion of structural features and the goal of not ending up with too much features, we use only the outermost structure tags in combination with the word as the input feature. The definition of this tag selection method is given in definition 21.

- Use only the tags on the inner level for classification.

  This is just like the last reduction method, but now we do not use the outermost tags, but the innermost tags. It is defined in definition 22.

Definition 21 (Outermost Structure Text). *Given a structured text* $\mathfrak{t}$ *as defined in definition 19 (page 43) we call* $\mathfrak{u}$ *the* m-outermost structure text*, if every tags-word-tuple* $(\mathtt{tag}_1, \ldots, \mathtt{tag}_n, \mathtt{word})$ *with* $m < n$ *in* $\mathfrak{t}$ *is replaced by* $(\mathtt{tag}_1, \ldots, \mathtt{tag}_m, \mathtt{word})$ *in* $\mathfrak{u}$.

Definition 22 (Innermost Structure Text). *Given a structured text* $\mathfrak{t}$ *as defined in definition 19 (page 43) we call* $\mathfrak{u}$ *the* m-innermost structure text*, if every tags-word-tuple* $(\mathtt{tag}_1, \ldots, \mathtt{tag}_n, \mathtt{word})$ *with* $m < n$ *in* $\mathfrak{t}$ *is replaced by* $(\mathtt{tag}_{n-m+1}, \ldots, \mathtt{tag}_n, \mathtt{word})$ *in* $\mathfrak{u}$.

When we transform the email message from figure 3.3 (page 44), to the *1-innermost structure text*, we get figure 3.4. When we transfer it to the *1-outermost structure text*, we get figure 3.5.

The notion of *m-innermost*, and *m-outermost structure text* allows to reduce the size of the vocabulary dramatically. Given a vocabulary of w words and t tags, where each word is affected by x tags, the transformation from a structured to a plain text given in definition 20 leads to a vocabulary size of $|t|^x \times |w|$, thus, it increases the size of the vocabulary by factor $|t|^x$. Definitions 22 and 21 reduce the size of the vocabulary for $m < x$ by a factor of $|t|^{x-m}$.

Let's consider an example: We have a text with 100 distinct words, each word is structured by 3 tags. This means, the structured text consists of 100 tuples with four components each: Three tags, and a word. We also assume all of the tags are distinct. If

```
(((From:), bthomas@uni-koblenz.de)
 ((To:), gb@uni-koblenz.de)
 ((Subject:), Re:)
 ((Subject:), Dokumentation:)
 ((Subject:), von)
 ((Subject:), MIA)
 ((body), klasse,)
 ((body), ich)
 ((body), denke)
 ((body), wir)
 ...
)
```

Figure 3.4.: *1-innermost text* of figure 3.3

```
(((header), bthomas@uni-koblenz.de)
 ((header), gb@uni-koblenz.de)
 ((header), Re:)
 ((header), Dokumentation:)
 ((header), von)
 ((header), MIA)
 ((body), klasse,)
 ((body), ich)
 ((body), denke)
 ((body), wir)
 ...
)
```

Figure 3.5.: *1-outermost text* of figure 3.3

we do not take the structural tags into consideration, the set of potential features has 100 elements; the words. The 1-innermost or 1-outermost structure text has 300 potential features: 100 words $\times$ 3 tags. The full text with all structural information has 2700 potential features: 100 words $\times$(3 tags)^3. We see that the reduction of structured text to *1-innermost structured text* or *1-outermost structured text* provides a compromise between keeping the vocabulary small on the one hand, and keeping structural information on the other hand.

3.2.2. Limiting Features by Information Content

The large number of potential features is one of the problems of text classification. It gets worse when we use the conversion from structured text into plain text given in the last section. Various methods have been proposed for limiting the number of features. One commonly used method, which already has been mentioned before, is to remove the features representing words which appear most or least often.[2] We expect these words to contribute only little to the classification task, i.e. we assume they contain very little information. Looking for the words appearing most and least often is only a heuristic method to filter out irrelevant words. A better method is to calculate the information content of every word by definition 12 on page 26. This method also allows us to tell exactly how many features we want to use. If we want n features, we take the n words containing the most information. If we would just remove the most and least frequent words, we can not choose the number of features so easily. We could choose the frequency-borders in a way that we end up with exactly n features, but it is not guaranteed that these features are chosen sensible for the text classification task. As a matter of fact, by reducing the number of features this way, the classification algorithms become very sensible to the number of features. In some areas they work fine, whereas performance breaks down when the number of features drops below a certain number. By choosing the n features containing the most information, the classifiers performance degenerates more gracefully when the number of features is reduced.

This method of feature selection is commonly used, but it is still not optimal. We have developed a better method for information selection. Feature selection based on feature information content becomes non-optimal when there are more than two categories and more than one feature. The reason for this is that the calculation of each potential feature's information content is independent of the other features. Let's assume we have to select two features from a larger set of potential features. We also assume within the potential features, there are two features whose distribution over the set of examples is exactly the same. Let's further assume that these two potential features contain the most information of all potential features. An algorithm which selects features based on their information content will choose these two potential features as the features. This is a non-optimal selection. Since the values of both features are identical, we get no additional information from the second feature. The same information would be present if we used only one of the features. (Which could be chosen randomly.)

[2] These method are for example proposed in [25, page 183]

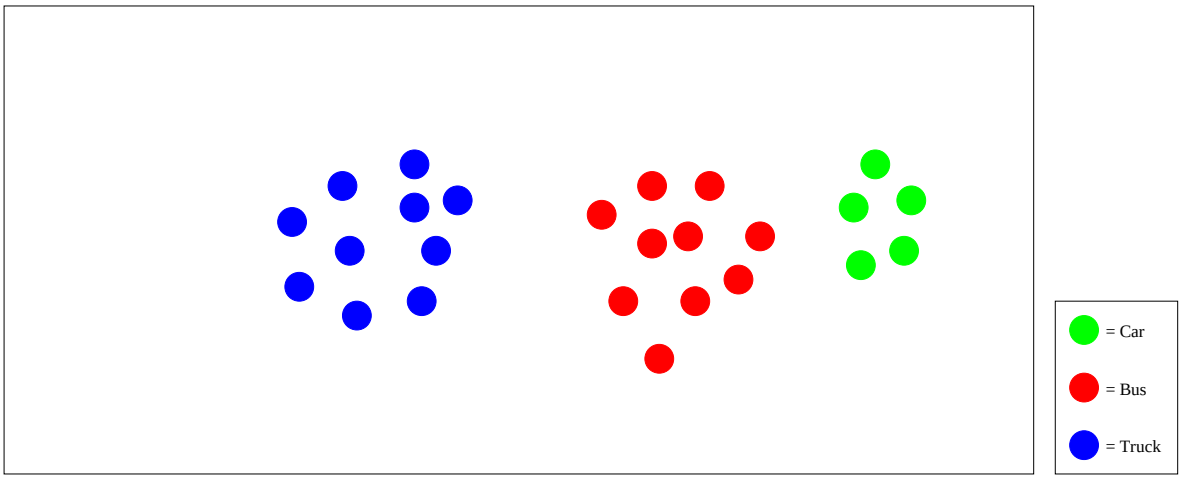

Figure 3.6.: The set of vehicles.

A better choice would be to choose one of the most informative features, and then do a recalculation which takes into consideration the information we got from the chosen feature. For this calculation, we have to split the set of objects into those where the most informative feature is present, and those where it is not present. We calculate the most informative features from both sets. The information gain of these two features is weighted by the number of objects. The result is the total information gain for these potential features. When this is done for every potential feature, we choose from these the one containing most information. This potential feature is the second feature. We continue this process until we have selected as many features from the set of potential features as we wanted.

This is exactly the same calculate that is done when constructing a Decision Tree. Thus, in order to find the best features, we can construct a Decision Tree, and traverse it, collecting the most informative features on the way. In the last chapter of this part of the thesis, where we evaluate the performance of our text classification methods in comparison to other methods, we show the performance of this feature selection method.

Example for Plain Information Selection

We come back to the vehicle classification task as an example. Let's assume we have 24 vehicles: 5 cars, 9 busses, and 10 trucks. The categories are "car" (c_1), "bus" (c_2), and "truck" (c_3). This is shown in figure 3.6. The features are "$>$ 500 kg" (v_1), "$>$ 8 passengers" (v_2), and "$>$ 2 axes" (v_3). Each of the cars, busses, and trucks has the input feature characteristics shown in table 3.2. We want to select the two most informative features from the set of three features. We do this selection by selection on the "plain" information of the features as well as by the Decision Tree method. We will show that the Decision Tree method reduces the information in the set of objects better than the "plain" selection method. This means that the features selected by the Decision Tree method are a better representation for discriminating between the sets of objects.

We start with the "plain" information selection method. The information contained

	> 500 kg (v_1)	> 8 passengers (v_2)	> 2 axes (v_3)
Car (c_1)	No	No	No
Bus (c_2)	No	Yes	No
Truck (c_3)	Yes	No	Yes

Table 3.2.: Input features of the objects in the example

in the set of objects D is calculated as

$$\begin{aligned} I(D) &= \sum_{i=1}^{3} P(c_i) \times -\log(P(c_i)) \\ &= P\left(\tfrac{5}{24}\right) \times -\log\left(\tfrac{5}{24}\right) \\ &\quad + P\left(\tfrac{9}{24}\right) \times -\log\left(\tfrac{9}{24}\right) \\ &\quad + P\left(\tfrac{10}{24}\right) \times -\log\left(\tfrac{10}{24}\right) \\ &= 0.46 \end{aligned} \tag{3.2}$$

We calculate the information of the subsets created by splitting the total set of objects in subset $D_{v_i=\mathsf{true}}$, for which $v_i = \mathsf{true}$, and in subset $D_{v_i=\mathsf{false}}$, for which $v_i = \mathsf{false}$. We start with $I(D_{v_i=\mathsf{true}})$:

$$\begin{aligned} I(D_{v_1=\mathsf{true}}) &= \sum_{i=1}^{3} P(c_i|v_1 = \mathsf{true}) \times -\log(P(c_i|v_1 = \mathsf{true})) \\ &= P\left(\tfrac{0}{10}\right) \times -\log\left(\tfrac{0}{10}\right) \\ &\quad + P\left(\tfrac{0}{10}\right) \times -\log\left(\tfrac{0}{10}\right) \\ &\quad + P\left(\tfrac{10}{10}\right) \times -\log\left(\tfrac{10}{10}\right) \\ &= 0 \end{aligned} \tag{3.3}$$

$$\begin{aligned} I(D_{v_2=\mathsf{true}}) &= \sum_{i=1}^{3} P(c_i|v_2 = \mathsf{true}) \times -\log(P(c_i|v_2 = \mathsf{true})) \\ &= P\left(\tfrac{0}{9}\right) \times -\log\left(\tfrac{0}{9}\right) \\ &\quad + P\left(\tfrac{9}{9}\right) \times -\log\left(\tfrac{9}{9}\right) \\ &\quad + P\left(\tfrac{0}{9}\right) \times -\log\left(\tfrac{0}{9}\right) \\ &= 0 \end{aligned} \tag{3.4}$$

$$\begin{aligned} I(D_{v_3=\mathsf{true}}) &= \sum_{i=1}^{3} P(c_i|v_3 = \mathsf{true}) \times -\log(P(c_i|v_3 = \mathsf{true})) \\ &= P\left(\tfrac{0}{10}\right) \times -\log\left(\tfrac{0}{10}\right) \\ &\quad + P\left(\tfrac{0}{10}\right) \times -\log\left(\tfrac{0}{10}\right) \\ &\quad + P\left(\tfrac{10}{10}\right) \times -\log\left(\tfrac{10}{10}\right) \\ &= 0 \end{aligned} \tag{3.5}$$

Now, we calculate the information content of the document sets $D_{v_i=\mathsf{false}}$.

$$
\begin{aligned}
I(D_{v_1=false}) &= \sum_{i=1}^{3} P(c_i|v_1 = false) \times -\log(P(c_i|v_1 = false)) \\
&= P\left(\tfrac{5}{14}\right) \times -\log\left(\tfrac{5}{14}\right) \\
&\quad + P\left(\tfrac{9}{14}\right) \times -\log\left(\tfrac{9}{14}\right) \\
&\quad + P\left(\tfrac{0}{14}\right) \times -\log\left(\tfrac{0}{14}\right) \\
&= 0.2830543
\end{aligned} \tag{3.6}
$$

$$
\begin{aligned}
I(D_{v_2=false}) &= \sum_{i=1}^{3} P(c_i|v_2 = false) \times -\log(P(c_i|v_2 = false)) \\
&= P\left(\tfrac{5}{15}\right) \times -\log\left(\tfrac{5}{15}\right) \\
&\quad + P\left(\tfrac{0}{15}\right) \times -\log\left(\tfrac{0}{15}\right) \\
&\quad + P\left(\tfrac{10}{15}\right) \times -\log\left(\tfrac{10}{15}\right) \\
&= 0.2764346
\end{aligned} \tag{3.7}
$$

$$
\begin{aligned}
I(D_{v_3=false}) &= \sum_{i=1}^{3} P(c_i|v_3 = false) \times -\log(P(c_i|v_3 = false)) \\
&= P\left(\tfrac{5}{14}\right) \times -\log\left(\tfrac{5}{14}\right) \\
&\quad + P\left(\tfrac{9}{14}\right) \times -\log\left(\tfrac{9}{14}\right) \\
&\quad + P\left(\tfrac{0}{14}\right) \times -\log\left(\tfrac{0}{14}\right) \\
&= 0.2830543
\end{aligned} \tag{3.8}
$$

Finally, we can calculate the information content for splitting on each feature v_i:

$$
\begin{aligned}
I(D_{v_1=true} \wedge D_{v_1=false}) &= \frac{1}{|D|} \cdot (|D_{v_1=true}| \cdot I(D_{v_1=true}) \\
&\qquad + |D_{v_1=false}| \cdot I(D_{v_1=false})) \\
&= \frac{1}{24} \cdot (10 \cdot 0 + 14 \cdot 0.2830543) \\
&= 0.1722939
\end{aligned} \tag{3.9}
$$

$$
\begin{aligned}
I(D_{v_2=true} \wedge D_{v_2=false}) &= \frac{1}{|D|} \cdot (|D_{v_2=true}| \cdot I(D_{v_2=true}) \\
&\qquad + |D_{v_2=false}| \cdot I(D_{v_2=false})) \\
&= \frac{1}{24} \cdot (9 \cdot 0 + 15 \cdot 0.2764346) \\
&= 0.1802834
\end{aligned} \tag{3.10}
$$

$$
\begin{aligned}
I(D_{v_3=true} \wedge D_{v_3=false}) &= \frac{1}{|D|} \cdot (|D_{v_3=true}| \cdot I(D_{v_3=true}) \\
&\qquad + |D_{v_3=false}| \cdot I(D_{v_3=false})) \\
&= \frac{1}{24} \cdot (10 \cdot 0 + 14 \cdot 0.2830543) \\
&= 0.1722939
\end{aligned} \tag{3.11}
$$

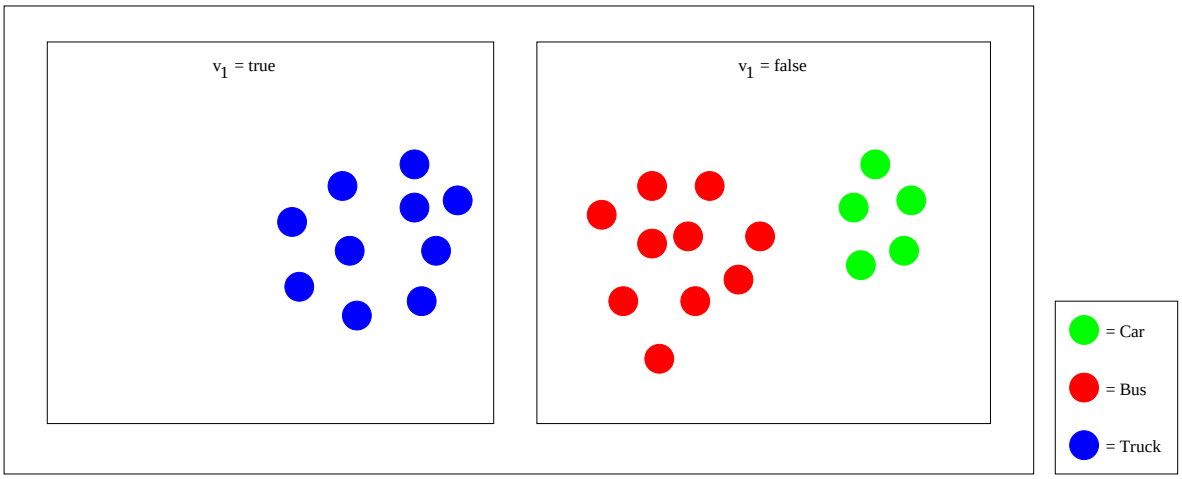

Figure 3.7.: The set of vehicles after splitting on v_1.

We select the feature which reduces most information from the set of objects In this case, both for splitting on feature v_1 and v_3, only 0.1722939 bit information is left in the set of objects. For v_2, 0.1802834 bit information is left in the set of objects. Therefore, "plain" information selection selects features v_1 and v_3. Figure 3.7 shows the set of vehicles after it has been split on feature v_1. Next, we calculate the information left in the set of documents after they have been split on the values of v_1 and v_3:

$$\begin{aligned} I(D_{v_1=true} \wedge D_{v_1=false} \wedge D_{v_3=true} \wedge D_{v_3=false}) = \\ \frac{|D_{v_1=false \wedge v_3=false}|}{|D|} \times I(D_{v_1=false \wedge v_3=false}) \\ + \frac{|D_{v_1=true \wedge v_3=false}|}{|D|} \times I(D_{v_1=true \wedge v_3=false}) \\ + \frac{|D_{v_1=false \wedge v_3=true}|}{|D|} \times I(D_{v_1=false \wedge v_3=true}) \\ + \frac{|D_{v_1=true \wedge v_3=true}|}{|D|} \times I(D_{v_1=true \wedge v_3=true}) \end{aligned} \tag{3.12}$$

Since for all objects from the set of objects, v_1 and v_3 have identical values, equation 3.12 can be reduced to

$$\begin{aligned} I(D_{v_1=true} \wedge D_{v_1=false} \wedge D_{v_3=true} \wedge D_{v_3=false}) = \\ \frac{|D_{v_1=false}|}{|D|} \times I(D_{v_1=false}) \\ + \frac{|D_{v_1=true}|}{|D|} \times I(D_{v_1=true}) \end{aligned} \tag{3.13}$$

Equation 3.13 is identicial to equation 3.9. So the information left when splitting on v_1 and v_3 is 0.1722939, which is the same information content as splitting on v_1 or v_3 alone. We get no further information by using a second feature. This is shown in figure 3.8.

Example for Selection by Decision Tree

Now, we use the Decision Tree approach to select the features. The calculation to determine the first feature is the same as in "plain" information selection. Since v_1 and v_3

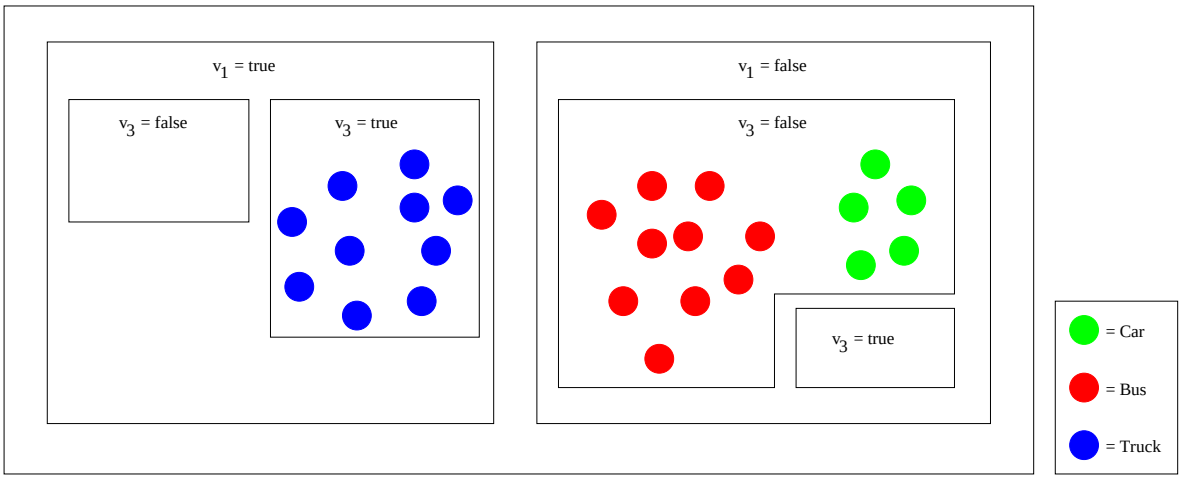

Figure 3.8.: The set of vehicles after splitting on v_1 and v_3.

reduce the information to the same value, we can choose one of them. We choose v_1. In order to determine the second parameter, we calculate the best feature for the subsets $D_{v_1=true}$ and $D_{v_1=false}$ of D which are created by splitting on v_1. $I(D_{v_1=true})$ has been calculated in equation 3.3. It is 0. All objects for which $v_1 = true$ belong to the same category. There is no information left in this set of objects. The information in the set of objects for which $v_1 = false$ has been calculated in equation 3.6. It is 0.2830543. We calculate how much information is reduced by splitting this set on feature v_2 or v_3. For v_3, this has already been calculated in equations 3.12 and 3.13: We gain no further information by splitting on v_3. For v_2, the information gain is

$$\begin{aligned} I(D_{v_1=false \wedge v_2=true}) &= \sum_{i=1}^{3} P(c_i|D_{v_1=false \wedge v_2=true}) \\ &\quad \times -\log(P(c_i|D_{v_1=false \wedge v_2=true})) \\ &= P(\tfrac{0}{9}) \times -\log(\tfrac{0}{9}) \\ &\quad + P(\tfrac{9}{9}) \times -\log(\tfrac{9}{9}) \\ &\quad + P(\tfrac{0}{9}) \times -\log(\tfrac{0}{9}) \\ &= 0 \end{aligned} \tag{3.14}$$

$$\begin{aligned} I(D_{v_1=false \wedge v_2=false}) &= \sum_{i=1}^{3} P(c_i|D_{v_1=false \wedge v_2=false}) \\ &\quad \times -\log(P(c_i|D_{v_1=false \wedge v_2=false})) \\ &= P(\tfrac{5}{5}) \times -\log(\tfrac{5}{5}) \\ &\quad + P(\tfrac{0}{5}) \times -\log(\tfrac{0}{5}) \\ &\quad + P(\tfrac{0}{5}) \times -\log(\tfrac{0}{5}) \\ &= 0 \end{aligned} \tag{3.15}$$

Figure 3.9 shows the set of vehicles after splitting on v_1 and v_2. In none of the sets is any information left. By using the features from the Decision Tree selection method,v_1

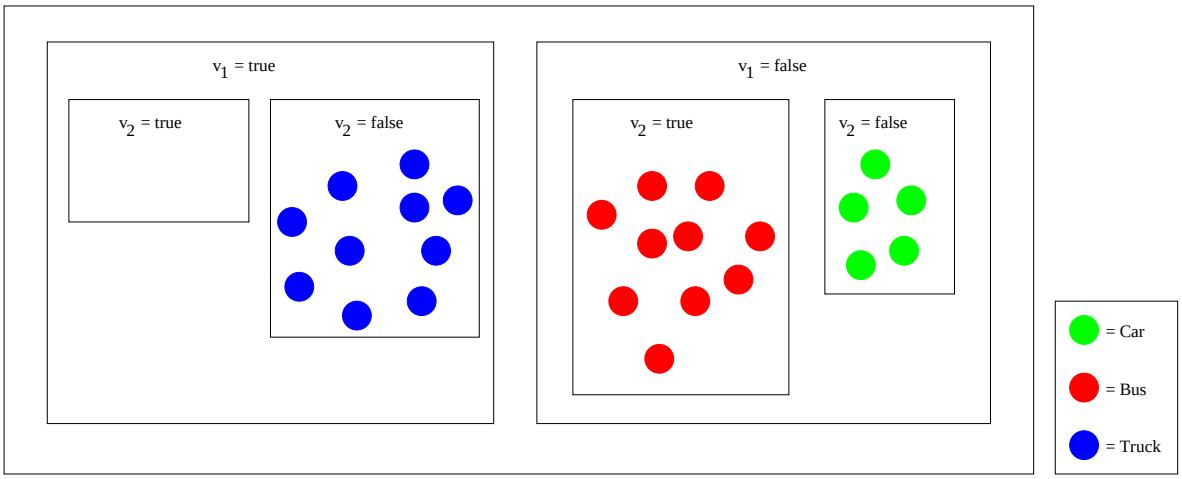

Figure 3.9.: The set of vehicles after splitting on ν_1 and ν_2.

and ν_2, we were able to reduce the information content in the subsets to 0. When using the features from the "plain" information selection method, ν_1, and ν_3, there is 0.1722939 bit information left in the subsets. Therefore, Decision Tree feature selection provides us with better features.

In the next chapter, the performance of the methods shown in this chapter, in combination with the classification algorithms from chapter 2, is measured on different data sets.

4. Performance Evaluation

In this chapter, we evaluate the performance of the classification algorithms from chapter 2 in combination with the input feature selection methods from chapter 3.

The documents used for the evaluation of the classifiers stem from two different sources. One part is taken directly from the applications in which we want to use the classification system. These are real world datasets which give direct information about the applicability of the classifiers to the target domains. The second source of data are publicly available datasets which serve as a general measure for the quality of text classification systems. Results are compared to the performance of existing third party systems.

Two indicators are commonly used in the performance evaluation of classification systems; the *recall rate*, and the *precision*. The recall rate tells how many of the objects falling into one of the categories have been identified. The precision rate tells how many of the objects which are recalled have been classified correctly. *Recall* and *Precision* are defined below.

Definition 23 (Recall). *Let* D *be a set of feature representations of texts. Let* $D_r \subseteq D$ *be a subset for which the reference classifier* r *defines a classification. Let* $D_t \subseteq D$ *be a subset for which the test classifier* t *gives a classification. The* recall rate *of the test classifier is defined as*

$$\text{recall} = \frac{|D_t|}{|D_r|}$$

Definition 24 (Precision). *Let* f_t *be the test classifier,* f_r *the reference classifier, and* D_t *the set of feature representations of texts for which the test classifier gives a classification. We define the* precision rate *of* f_t *as*

$$\text{precision} = \frac{|\{t | t \in D, f_t(t) = f_r(t)\}|}{|D_t|}$$

In our implementation of the classification algorithms, the recall rate is always 100%. The classifiers *always* assign a category to a text. There is no way that our classifiers refuse to classify a text. For this reason, we show only the *precision rates* in the results.

4.1. Documents to Classify

4.1.1. MIA

One of the areas of practical applications for our text classification system is the MIA-project, developed at the University Koblenz Artificial Intelligence Research Group. In chapter 1.2, we explained the usage of a classifier within the MIA system. Here, the classification task is to tell if web pages contain (real world) addresses. The categories are "page contains an address" and "page does not contain an address". Getting evaluation data for the web page classification task is straight forward. We collect data from the real, running MIA application by grabbing the results from the spider agent.

We searched for the 21 cities Bayreuth, Halle, München, Berlin, Hamburg, Potsdam, Bonn, Rostock, Chemnitz, Kiel, Saarbrücken, Dortmund, Koblenz, Stuttgart, Erfurt, Köln, Wiesbaden, Essen, Leipzig, Frankfurt and Mainz, and for 9 topics: "Sport", "Restaurants", "Kultur" (culture), "Hotels", "Bibliotheken" (libraries), "Bahnhöfe" (train stations), "Zeitungen" (newspapers), "Vereine" (associations/clubs) and "Firmen" (companies). The MIA spider agent collected 973 pages. From these 973 pages, 347 pages (35.66%) contain an address, and 626 pages (64.34%) do not contain an address. Table 4.1 shows some more details about the pages.

All classification algorithms are *supervised learning* algorithms. In order to be trained, they need the correct classifications of the web pages. Classifying all pages manually is a monotonous, time-consuming and error-prone task. In order to avoid it, we use a highly accurate classifier as the reference classifier. A non-trainable classifier with special domain knowledge is used. It has access to a database containing the zip codes and names of German cities. When a web page contains the zip code of a city followed by its name, it is classified as containing an address. We assume that the classification of the reference classifier is always correct, i.e. we do not take noise into consideration.

4.1.2. Emails

Chapter 1.3 shows the context of the email classification scenario. The reason for optimizing the system on email classification was a proposed cooperation with an ISP. An ISP has a number of generic support accounts with names like `abuse`, `admin`, `postmaster`, `info` and `root`. These accounts receive hundred of email messages a day. This ISP wants to forward email addressed to its support accounts automatically to the person responsible for the specific problem mentioned in the email.

We do not have access to original data. Therefore, we use a dataset from a different area of email classification to test the performance of MIC. This is the personal email archive of the supervisor of this thesis. The original dataset contains 2250 emails, sorted in 37 categories. The categories correspond to the topics of the emails. With this dataset, we evaluate how well a text classification system can aid a user in filing her personal correspondence. The email dataset poses some problems. One problem is the large variety in the number of emails in each of the categories. While one category contains only 7 emails, others contain up to 494 emails. Additionally, the categories are not always clearly distinguishable. There are two categories covering the same topic.

City	Pages Collected	Contains Addresses?		City	Pages Collected	Contains Addresses?	
		Yes	No			Yes	No
Bayreuth	37	6	31	Koblenz	63	32	31
Berlin	39	8	31	Köln	34	3	31
Bonn	39	8	31	Leipzig	25	2	23
Chemnitz	32	4	28	Mainz	45	14	31
Dortmund	34	3	31	München	47	16	31
Erfurt	57	26	31	Potsdam	42	11	31
Essen	35	4	31	Rostock	67	36	31
Frankfurt	34	3	31	Saarbrücken	32	1	31
Halle	18	1	17	Stuttgart	71	40	31
Hamburg	112	81	31	Wiesbaden	72	41	31
Kiel	38	7	31				

Table 4.1.: "Pages Collected" is the total number of pages collected for the city. "Contains Address" and "Does not Contain an Address" are the number of pages that do (not) contain an address.

The only difference between the two categories is that one contains older messages, and the other contains recent messages. We can not be sure that ambiguous cases, where an email falls in between two categories, or fits in more than one category, have been handled in a consistent way. Additionally, we do not know if the manual classification was always correct in the first place.

Beside the full dataset, we use a second, smaller subset of the emails. The smaller dataset contains 363 emails in 8 categories. We do this, because we want to compare the performance of our classification system against other systems. One of these is SERpersonalBrain. SERpersonalBrain is a commercial software package. The only version available for no charge supports only a limited set of categories and documents. It is limited to 10 categories, and 100 documents per category.

Since these are non-anonymized personal mails, we can not provide this dataset to the public. Therefore, it is not included on the CD.

4.1.3. Newsgroups

In order to compare the performance of our classification algorithms to other classifiers, we run tests on a dataset from the Internet which is commonly used for the comparison of text classification algorithms.

The bow library [24] comes with a large example dataset of articles from newsgroups. Articles from 20 newsgroups have been collected: `alt.atheism`, `comp.graphics`, `comp.os.ms-windows.misc`, `comp.sys.ibm.pc.hardware`, `comp.sys.mac.hardware`, `comp.windows.x`, `misc.forsale`, `rec.autos`, `rec.motorcycles`, `rec.sport.baseball`, `rec.sport.hockey`, `sci.crypt`, `sci.electronics`, `sci.med`, `sci.space`, `soc.religion.christian`, `talk.politics.guns`, `talk.politics.mideast`,

`talk.politics.misc`, and `talk.religion.misc`. 1000 articles have been collected from each of the newsgroups. We use a striped down version of this dataset, which is also available on the Internet [23]. It contains 100 messages in each of the newsgroups, totaling 2000 messages.

4.2. Naive Bayes Classifier for Texts

In chapter 2.6.1, the calculation of the feature probabilities for the Naive Bayes classifier is explained. The calculation method from chapter 2.6.1 assumes binary features: Either a feature is present in the representation of an object, or it is not present. However, the text representation introduced in chapter 3 is richer. The feature vector components do not simply indicate if a word is present or not. They give the frequency of a word in the text. The algorithm for the calculation of feature probabilities can be adapted to make use of the richer representation. For each distinct word w_x in at least one of the texts in the training dataset, we calculate the probabilities of a randomly drawn word from class c_y being word w_x, $P(w_x|c_y)$. This is calculated as the total number of words w_x in texts of class c_y, divided by the total number of words in all texts. 1 is added to the numerator in order to avoid zero probabilities in the cases where the word does not appear in any document of class c_y. Zero probabilities are not desirable, because in case a testing document contains one of the words which have zero probability, the probability of the text belonging in the category would become zero, too.

Definition 25 (Naive Bayes Word Probabilities). *Let* D *be a set of texts, and* c *a category. Let* n *be the total number of distinct words in* D*, and let* w *be a word. Let* $|w_c|$ *be the total number of words* w *in texts from category* c*. The probability for a randomly word drawn from category* c *to be word* w *is calculated as*

$$P(w|c) = \frac{|w_c| + 1}{|D| + n}$$

4.3. Feature Selection

Chapter 3 shows the input feature selection methods used in this thesis. We can distinguish between four kinds of parameters:

- Parameters restricting the number of input features.
- Parameters defining how input features are selected from a larger set of potential features.
- Parameters influencing if and how more than one word is mapped to one (potential) input feature.
- Parameters influencing the way how structural information is presented to the classifier.

Setting the Number of Input Features

The number of input features influences the classification in two ways: When we use too few input features, the classifier might not be able to perform well because of lack of information. If we use too much input features, the classification algorithm might *overfit*. Chapter 4.5.1 examines the influence of the number of input features. In the simulations in chapters 4.5.2 to 4.5.4, the number of input features is equal to the number of nodes in the Decision Tree. There are as many features used as necessary to contain all the information from the training dataset.

Selecting Input Features

Two methods are used to select the features. The first method is the information content of the words. The words containing most information are selected as input features. The second method does not use the "plain" information content of the words, but the information content of the words in relation to the other words selected as input features. The second method uses a Decision Tree to find the most informative features. This method is described in chapter 3.2.2. A comparison of both methods is shown in chapter 4.5.1. In the following runs of the classification system, the advanced decision tree based feature selection method is used.

Improved Feature Selection

These are the generic methods to improve input feature selection described in chapter 3.1.1. In chapters 4.5.2 to 4.5.4, all runs are performed with plain feature selection. Each word is mapped to one (potential) input feature, and with improved feature selection using the following methods:

- Convert text to lower case

  This is one of the general methods to reduce the size of the feature space. All words are converted to lower case.

- Convert quoted printable characters

  The quoted printable format allows to transfer 8 bit characters (like German umlauts) with a 7 bit encoding. This option reverts the quoted printable characters to their original 8 bit representation.

- Convert HTML special characters

  HTML also has a special format to convert 8 bit characters in 7 bit format (See [3]). Just like for quoted printable characters in emails, this option converts HTML escaped characters back to their original form.

- Remove special characters from text

  This is also a generic method for feature-reduction: All non-alphanumeric characters remaining after quoted printable and HTML special characters have been converted are removed.

- Remove stop words

Remove stop words. Right now, we use a list of German stop words which is partially shown in table 3.1 on page 42.

Transferring Structure Into Features

One aim of this thesis is to extend text classification algorithms for utilizing structural data. The example datasets vary in the kinds of structural information they provide. The datasets of email messages and newsgroup articles are very similar in their structure. Email messages, as well as news articles, consists of a header and a body. The header contains additional structural elements. These elements are (among others) the name of the sender, the name of the receiver, a subject line, and references to other emails / newsgroup articles. Since the proliferation of the MIME-standard for emails (see [6]), emails do not only contain plain text, but often also all kinds of binary attachments. These attachments usually do not contribute to the classification process, because the "word strings" in them are rather meaningless, and since the binary parts are usually a lot larger than the text parts of a message, the classification process can be distracted from those words which are really important. Even more important, the parsing of large binary parts of a message takes a lot of time. Therefore, binary parts are removed from emails. This option is only of interest if we are classifying emails.

The MIA dataset differs from the other datasets. A HTML page contains a header and a body, too, but in difference to the other datasets, the header section of a HTML document contains less information and is not as rigid structured as the headers of email messages and newsgroup articles. In difference to the other datasets, the body, i.e. the actual text, of a HTML document contains structure tags, too.

Definitions 21 and 22 on page 45 define the n-*innermost (-outermost) text*. Each word is associated with the n innermost (outermost) tags affecting it. We evaluate the performance of the classifiers on the document sets for $n = 1$ and $n = 2$ both for the innermost and outermost structure text.

4.4. Existing Systems

The performance of our methods is compared to the performance of already existing systems for automatic text classification. An extensive list of text classification system is available from the web pages of CALD (Center for Automated Learning and Discovery) at CMU [22]. We have examined those systems from the list which are available for free (or at least free demo versions are downloadable). Other systems (like CLEMENTINE [34]) have not been taken into consideration. These systems are:

RoC: The Robust Bayesian Classifier

RoC [30] has been developed at the Knowledge Media Institute of Open University (UK). It is a Windows 9x/NT desktop software which allows the classification of texts via a GUI. The only algorithm supported by RoC is the Robust Bayesian Classifier. RoC is royalty free for non-commercial use. Source code is not available.

MLC++

MLC++ [32] is a C++-library developed at Stanford University and SGI for supervised automatic classification. MLC++ is provided in source code, but the license allows to use it only for research projects. It is based on the LEDA [9] library which provides additional data types. LEDA is commercial software. We had not been able to compile MLC++, and there are no binary versions available.

SERpersonalBrain V01.02

SERpersonalBrain is a commercial product from SER Systems AG. A demo version is available at [31]. SERpersonalBrain is a text classification and retrieval system for Desktop-PC users. Via graphical user interface, the user can give examples for document classes to the system. After a training phase, SERpersonalBrain categorizes new documents according to the training data. The classification method is fixed. No information is given about the classification method used. The demo-version is limited to 10 categories and 100 documents per category.

Rainbow

Rainbow is a text classification frontend for the bow library. The bow library [24] has been developed by Andrew Kachites McCallum from the Center for Automated Learning and Discovery at Carnegie Mellon University. Bow is a library for statistical text analysis, language modeling and information retrieval programs. Rainbow is a command-line program supporting text classification. Rainbow supports multiple text classification algorithms and several ways to manipulate the input data. Rainbow is free software. For the large set of text classification methods available, and its ease of use, Rainbow will serve as a reference classification system when we evaluate the results of our methods.

SVM^{light}

SVM^{light} [14] is a project originally developed at University of Dortmund, and now hosted at GMD. This command-line program implements *Support Vector Machines* (SVMs). We did not include SVM^{light}into the performance comparison, because Support Vector Machine classification is one of the classification algorithms provided by Rainbow, too.

In chapter 1 we formulated a number of requirements for a text classification system which can be used for our classification tasks. Table 4.2 shows how well the various already existing classifiers fit our needs. The evaluation of existing text classification systems leads to the conclusion, that we have to develop a new system. This is MIC ("MIA Classifier"). MIC fulfills all of the requirements from table 4.2. The technical aspects of MIC are described in the second part of this thesis. Although the existing text classification systems can not be used for our purposes, we do use them to compare the performance of our improved classification methods to the results produced by these classifiers.

	RoC	MLC++	SER	SVM^{light}	Rainbow
Can be used as a software agent	–	?	–	–	√
Trainable on various tasks	√	√	√	√	√
Learn from examples	√	√	√	√	√
Make use of emails & HTML pages	–	–	–	–	–
CLI	?	?	–	√	√
GUI	√	?	√	–	–
Multiple classification methods	–	√	–	–	√
Extensible	–	?	–	–	√
Flexible input data manipulation	?	?	–	–	√
Free Software	–	–	–	–	√

Table 4.2.: Overview: Capabilities of existing text classification systems

4.5. Results

4.5.1. Restricting the Number of Input Features

First, we examine the parameters influencing the number of input features. We use the full email dataset in combination with standard feature selection. The experiments are run with the Naive Bayes classification algorithms. The number of input features is varied between 5 and 50. In one set of experiments, the features are extracted from a Decision Tree. In the other set, the "plain" information content of the words is used to decide which of the potential features to use. These methods are described in chapter 3.2.2. The precision of the classification algorithms is shown in figure 4.1. The figure shows that input feature selection based on a Decision Tree needs about 5 input features less, in order to achieve the same precision. We also see for both feature selection mechanisms the deterring effect when too many features are used. Performance decreases when a certain number of features is exceeded. The reason for this deterioration is the same as for overfitting of a Neural Network (see chapter 2.5): By adding too much features, the algorithm gets distracted from the features which really contain the information relevant for the classification task. It turns to features which have no relevance for the real classification task, but coincidently correlate with some of the categories in the training dataset.

4.5.2. MIA Dataset

Table 4.3 shows the performance of the different algorithms and feature selection methods on the MIA dataset. For the Neural Network classifier, we see how the inclusion of structural information improves the classification process. While its accuracy is only 73.8758 % when not using structural information, the classification algorithm achieves between 84.7966 % and 87.5803 % when including structural information.

It is surprising that for the Naive Bayes classifier, the standard feature selection performs better than all the improved feature selection methods. We see this behavior

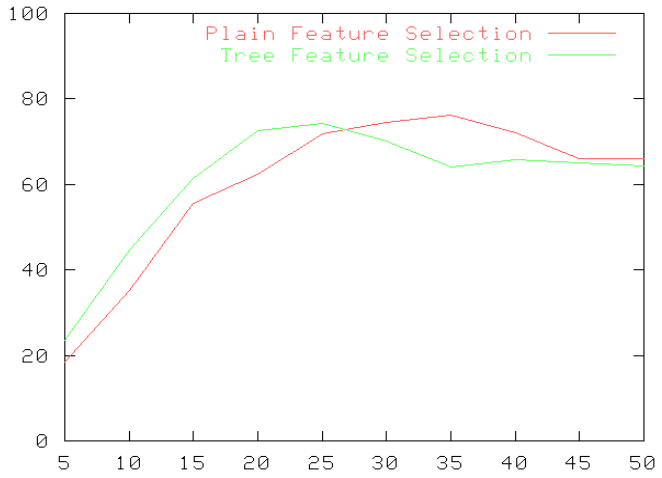

Figure 4.1.: Classification precision of a Naive Bayes classifier using 5–50 input features. The red line shows the performance of a classifier using plain information content to select features, the green line shows a classifier using Decision Tree based information content to select features. The data for this figure can be found on the CD in directory `data/results/nr_input_features/`.

for no other classifier. For an analysis of it, we have a look at table 4.4. It shows the most informative words used by standard feature selection, by improved feature selection (see page 58) and by 2-outermost feature selection. The most informative word list from the improved feature selection shows that the algorithm has not been too successful in finding generic features to identify address information. This is not a surprise, because the feature selection mechanism only counts the frequency of words.

It is quite obvious why "fax" was selected as the most informative word, because in a typical address information, a fax-number is included very often. The third most informative word, "040" contains less information, because it is more specific: "040" is the telephone area code of Hamburg. Again, a phone number is usually part of an address, but this feature identifies only addresses in Hamburg. The second most informative word, "blind", is clearly a wrong generalization. It has been selected, because in the training data, a lot of the pages containing an address have been created by the same company. This company uses images named "blind.gif" on their pages. By choosing this keyword, the algorithm has only learned that web pages designed by this company usually contain address information. The same is true for the words "gmbh
", "kcs", and "info@kcs", because the name of said company is "KCS GmbH". For all other potential features, like city names, the algorithm would need advanced linguistic techniques, in order to know that a word is the name of a city.

The advanced feature selection method, which takes the structure of the web page into consideration (see page 58), leads to a somewhat better feature selection. The most informative word list shows that this selection scheme gives better features: Beside the

	Naive Bayes	Decision Tree	Neural Network
Standard Feature Selection	80.5139%	87.152%	84.7966%
Improved Feature Selection	84.3683%	87.7944%	73.8758%
1-Innermost Tags	83.5118%	84.7966%	86.5096%
2-Innermost Tags	85.2248%	86.9379%	84.7966%
1-Outermost Tags	82.0128%	89.2934%	86.0814%
2-Outermost Tags	83.2976%	89.2934%	87.5803%

Table 4.3.: MIC's performance on MIA dataset. The data for this figure can be found on the CD in directory `data/results/mia_dataset/`.

word "fax", which had already been on the most informative word list for plain feature selection, it now contains the words "tel", "email", and "internet". We see that these words indicate an address only if a break tag "`<br>`" is in front of them. On web pages, it is a common method of formatting to end each line of an address with a break. Therefore, this word selection is more accurate. For the plain texts, "tel", "email", and "internet" do not belong to the most informative words, because they appear too often in a non-address contexts. With the additional structural information of the `<br>`-tag, the algorithm can identify those appearances of the words where they are most likely part of an address.

The Decision Tree classifier and the Neural Network classifier perform somewhat better than the Naive Bayes classifier. The reason for this lies in the input feature selection. As we have seen in the discussion about the most informative word lists, pages containing address information can be identified by looking for certain keywords. When the word "fax" appears on a page, it is very likely that this page contains an address. The Decision Tree classifier uses binary input features. It gets the information if a word is contained on a page, or not. The Neural Network and the Naive Bayes classifier get somewhat different information: For them, the input feature is the relative frequency of a word on a page. For the Naive Bayes classifier, this makes it harder to classify a page, because the influence of a word on the probability of a hypothesis is linear to the number of appearances of the word. Therefore, the algorithm is mislead by the different frequencies of the keywords. Neural Networks do not suffer this problem, because the influence of the network's input (i.e. the feature vector elements) is non-linear.

For the classification of HTML pages, we would expect better results when taking the structure of the page is taken into consideration, because HTML pages contain a lot of structural information. In practice, the performance increases only little when using structural information. An explanation for this lies in the hybrid structure of HTML. The structure of HTML tags is not very rigid. A lot of tags are not indicators for the kind of information they are tagging, but for the way the information shall be displayed on a screen. HTML mixes tags indicating structural properties of texts, like the `<h1>` tag to indicate head lines, with formatting tags like `<font>`, which set a specific font. We can not expect to take advantage of these tags. Additionally, a lot of HTML pages are written in a very sloppy way, where tags are used semantically wrong as well as

	Standard Features	Improved Features	Outermost-2 Features	Rainbow
1	Fax:	fax	br_a_internet	fax
2	·	blind	a__32	tel
3	BGCOLOR=#FFFFFF	040	_br_040	br
4	GmbH 	mailto	_br_tel	kr
5	040	gmbh 	style__8381e1	middot
6	6076	1367	style__1814b8	blind
7	KCS	6076	_br_email	sungen
8	EMail:	kcs	_br_6076	hotel
9	ALT=¨Hamburg	topsite	_br_1367	mailto
10	CELLSPACING=0>	info@kcs	_br_fax	topsite

Table 4.4.: Most informative words for the classification of the MIA dataset using standard feature selection, improved feature selection (see page 58), and 2-outermost feature selection. The right column shows the most informative words used by Rainbow

syntactical wrong.

We can expect this to change when structural languages with a better separation of description of context and description of how the context shall be displayed, like XML [41], proliferate.

Comparison to Rainbow

The performance of Rainbow on the MIA dataset is shown in table 4.5. The best performing algorithm / input feature selection method combination of MIC classifies 89.2934 % of the pages correctly. This is slightly better than the performance of Rainbow, where the best classification result is 88.04 %. It is interesting to note that Rainbow's Naive Bayes classifier performs better than MIC's Naive Bayes classifier. MIC's Naive Bayes classifier with improved feature selection (see page 58) achieves a precision of 84.3683%. The reason for this lies in the different methods used by MIC and Rainbow to improve the quality of the features. Rainbow filters out more non-alphanumeric characters than MIC. The lists of most informative words used by Rainbow and MIC are shown in figure 4.4. In difference to the improved word list used by MIC, Rainbow uses both the words "tel", and "fax" for classifying pages, whereas MIC only uses "fax". Rainbow filters out all special characters, whereas in MIC some special characters remain in the words. This is due to the MIC's treatment of structural tags. MIC's structural feature selection method build upon the improved feature selection methods. For the structural methods, it is necessary to keep special characters like "<", and ">" in the text in order to distinguish tags from words. This is not really a problem, because when including the structural information into the feature selection, MIC outperforms Rainbow.

Naive Bayes	88.04%
TFIDF/Rocchio	87.05%
K-nearest neighbor	87.84%
Maximum Entropy	86.64%
Expectation Maximization	86.93%
Support Vector Machines	87.67%
Fuhr's Probabilistic Indexing	86.85%
Shrinkage with Naive Bayes	87.71%

Table 4.5.: Performance of Rainbow's algorithms on MIA dataset

4.5.3. Email Dataset

Emails only have two structural tags: The outer one indicates if the section of the email is the header or the body. For the head elements, there is an additional tag which shows to which header (for example `From:`, `To:`, or `Subject:`) the line belongs. Therefore the *n-innermost-selection* for all n is identical with the *1-innermost-selection*. The same applies to the *n-outermost-selection*.

We used two datasets in these simulations. One contains the full set of emails. The other contains a reduced set of only 363 emails in 8 categories. This smaller dataset has been created in order to compare the performance of our methods to the performance of the commercial software SERpersonalBrain. Only an evaluation version of SERpersonalBrain is available without charge. This evaluation version is limited to a maximum of 10 categories with up to 100 document in each category. MIC's results are shown in table 4.6.

An examination of the most informative words lists shows why the classification of the emails does not improve when we include structural information from the mail headers. Table 4.7 shows the most informative word lists for the different feature selection methods. The number one most informative words on all lists look a bit strange on first sight: For the plain (no structure) feature selection list, this word is "ro". For the word list using structural information, it's the word "header_o", i.e. the word "o" in the header of a message. These strange entries stem from the way a Unix mail client stores status information about email. The client adds a line "`Status:`" to the header, followed either by "RO", for old mail which had been read, or "O" for old mail that has not been read. If no status line is present, the mail arrived newly in the system. Thus, the most informative feature of a mail is whether it has been read, or not. In terms of data mining, we can extrapolate that the user tends to read mails from only a few categories, whereas other mail gets stored unread. The status information is only available because our classification algorithm works on archived mail. If the classification algorithm would work within the MDA as intended, this information would not be present.

SER Dataset			
	Naive Bayes	Decision Tree	Neural Network
Standard Feature Selection	64.6707%	77.2455%	76.0479%
Improved Feature Selection	71.8563%	78.4431%	77.8443%
1-Innermost Tags	70.0599%	76.0479%	67.6647%
1-Outermost Tags	71.2575%	78.4431%	74.8503%
Full Email Dataset			
	Naive Bayes	Decision Tree	Neural Network
Standard Feature Selection	52.0436%	68.0291%	70.0272%
Improved Feature Selection	63.851%	75.931%	70.5722%
1-Innermost Tags	59.2189%	62.2162%	67.6658%
1-Outermost Tags	64.396%	71.4805%	69.4823%

Table 4.6.: MIC's performance on email SER dataset and full email dataset. The data for this figure can be found on the CD in the directories `data/results/full_email_dataset/` and `data/results/ser_email_dataset/`.

Comparison to SERpersonalBrain

As mentioned in chapter 4.1.2, only an evaluation version of SERpersonalBrain is available without charge. It is restricted in the numbers of categories and documents. In order to evaluate the performance of this program, we created the `SER Mail` subset of the full email dataset. The SER subset contains 363 mails in 8 categories. Table 4.6 shows the performance of our algorithms on this dataset. 196 of these mails have been used for training, and 167 for testing. SERpersonalBrain classified 118 mails correctly, 6 wrong, and 42 not at all. This is a recall-rate of 70.6%, and a precision of 95.1%. The top performing algorithm / feature selection combination from our experiments classifies 78.44% of the emails correctly. Thus, our classification methods outperform SERpersonalBrain on this dataset.

Comparison to Rainbow

Table 4.8 shows the performance of Rainbow on this dataset. The performance of all algorithms provided by Rainbow is basically the same. Just as for the other testing datasets, MIC significantly outperforms Rainbow. The performance of Rainbow's classification algorithms is comparable to MIC's Naive Bayes classifier with standard feature selection. There are two reasons why Rainbow achieves only the performance of MIC's worst combination of classification algorithm and input feature selection: The first reason are the algorithms. Just like for the MIA dataset, in this classification task not the frequency of a word on a page is really relevant, but if a word appears on a page at all. In chapter 4.5.2 we explain why Neural Networks and Decision Trees are better suited for these kind of classification tasks than Naive Bayes classifier.

The second reason lies in Rainbow's input feature improvements. For the MIA classification task, Rainbow's more restrictive way of removing all non-alphanumeric characters

	Improved Features	Outermost-1 Features	Rainbow
1	ro	header_o	ro
2	edu	header_edu	cade
3	ijcar	to	die
4	ki	the	ich
5	the	header_vm	der
6	uka	header_d	und
7	adnan	header_uka	uiowa
8	floc	adnan	cs
9	chalmers	header_ijcar	zu
10	nil	i	pschmitt

Table 4.7.: Word list for email classification using improved feature selection and 1-outermost feature selection. The right column shows the most informative words used by Rainbow.

Naive Bayes	53.42%
TFIDF/Rocchio	53.26%
K-nearest neighbor	53.62%
Maximum Entropy	53.85%
Expectation Maximization	53.10%
Support Vector Machines	52.41%
Fuhr's Probabilistic Indexing	53.03%
Shrinkage with Naive Bayes	53.17%

Table 4.8.: Performance of Rainbow's algorithms on full email dataset

had been an advantage compared to MIC's less restrictive approach. Here, MIC's approach does better. The list of most informative words in table 4.7 also shows the most informative words used by Rainbow. Compared to MIC's improved feature selection, the word "edu", which is the second most informative word for MIC's feature selection, is not used by Rainbow.

4.5.4. News Dataset

For the news dataset, the same argument applies as for the email dataset. It is interesting to examine how features are selected in this example. In a way, there is no real classification task involved. The classification, i.e. the name of the newsgroup, is provided in the header, thus it is part of the input data. The dataset is of interest anyway, because it allows us to study if and how this information is detected and used by the classification methods.

	Naive Bayes	Decision Tree	Neural Network
Standard Feature Selection	81.5763%	75.8444%	51.7912%
Improved Feature Selection	83.2139%	92.4731%	82.088%
1-Innermost Tags	84.1351%	94.5752%	91.2999%
1-Outermost Tags	87.2057%	94.1658%	85.6704%

Table 4.9.: MIC's performance on newsgroup dataset. The data for this figure can be found on the CD in directory `data/results/newsgroups_dataset/`.

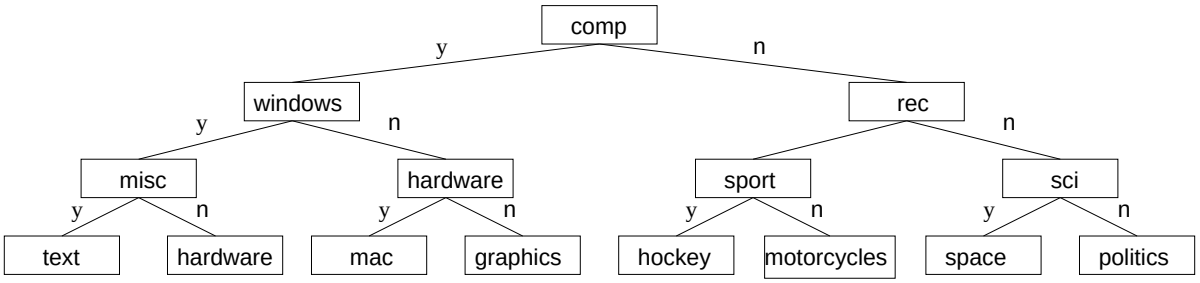

Figure 4.2.: Top of decision tree for newsgroup classification using improved feature selection (see page 58).

The Decision Tree classifier allows to get exact information about the classification process. Table 4.10 shows the most informative words for the different feature selection methods. Decision Tree classifier with standard feature selection decided on the classifications in a very reasonable way: The classification is mainly based on the newsgroup name. The notable exceptions are "christian@aramis.rutgers.edu" and "Approved:" as the first and third most informative word. The newsgroup `soc.religion.christian` is moderated. Therefore, all articles in it contain the header line `Approved: christian@aramis.rutgers.edu`. It is a better keyword than the name of the newsgroup itself, because the name of the newsgroup is mentioned in other newsgroups, too. The other categories are identified by the name of the newsgroups. Although this is a very good strategy, the classification is correct only for 75.8444% of the documents. This feature selection method has two weaknesses. It does not know if the newsgroup name comes indeed from the `Newsgroups:` header. It is also possible that the name of a newsgroup is mentioned in the body of an article, but the article did not got posted to this newsgroup. The second problem poses articles which got cross-posted to multiple newsgroups. In cross-posted articles, the `Newsgroups:` line contains multiple newsgroups names, separated by commas. In these cases, the relevant newsgroup's name is not a separate feature, but combined with the names of other newsgroups. Therefore, the classification algorithm can not identify it.

This second problem does not occur for improved feature selection (see page 58). Figure 4.2 shows the top of the decision tree for this feature selection method. One aspect of the improvement is that all special characters are removed from the text. This includes

	Standard Features	Improved Features	Outermost-1 Features
1	christian@aramis.rutgers.edu	comp	header_comp
2	rec.motorcycles	rec	header_sci
3	Approved:	sci	header_rec
4	soc.religion.christian	politics	header_politics
5	rec.sport.hockey	windows	header_windows
6	rec.sport.baseball	sport	header_sport
7	comp.windows.x	approved	header_electronics
8	References:	space	header_hardware
9	sci.electronics	hardware	header_approved
10	comp.sys.mac.hardware	mideast	header_space

Table 4.10.: Ten most informative words for the newsgroup dataset using standard, improved, and 1-outermost feature selection.

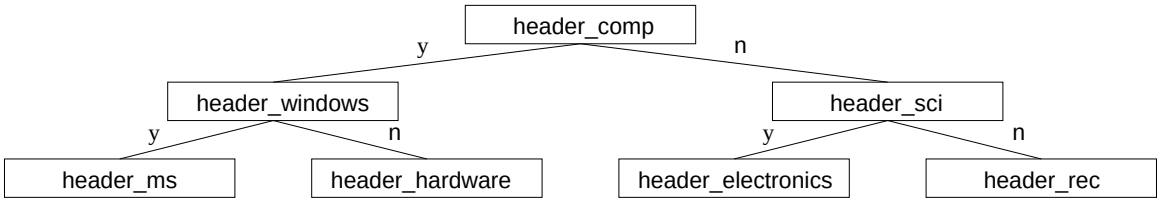

Figure 4.3.: Top of decision tree for newsgroup classification using 1-outermost feature selection.

commas. Therefore, the newsgroup names are no longer concatenated. A new problem arises from this. The removal of special characters also removes periods. The names of the newsgroups are teared apart. The algorithm does not identify `rec.sports.baseball` as one feature, but as the three features `rec`, `sports`, and `baseball`. This difference leads to a radically different Decision Tree. The Decision Tree for the plain feature selection methods had been very similar to *decision lists*: The tree is very unbalanced. On each node, the positive branch leads directly to a leaf node, i.e. a classification. With improved feature selection, we get a "real" Decision Tree, who's hierarchical structure corresponds to the hierarchical structure of the newsgroups. The first node splits on the top level domain "comp". This separates all documents who are in one of the newsgroups below `comp.ALL` from the rest. At the branch with the texts not belonging in `comp`, `rec.ALL` is split from the rest. The next split in the `rec.ALL` branch occurs on the word `sport`, splitting the messages from `rec.sport.ALL`, from the rest. This continues in the next split on the word `hockey`. By this split, the messages from `rec.sport.hockey`, and from `rec.sport.basketball` are identified (the latter because they do not contain the word `hockey`).

The improved feature selection is still not perfect. The separate treatment of the ele-

Naive Bayes	77.16%
TFIDF/Rocchio	78.06%
K-nearest neighbor	78.36%
Maximum Entropy	78.24%
Expectation Maximization	77.96%
Support Vector Machines	77.78%
Fuhr's Probabilistic Indexing	78.28%
Shrinkage with Naive Bayes	78.42%

Table 4.11.: Performance of Rainbow's algorithms on newsgroup dataset

	MIC	Rainbow
MIA dataset	89.2934%	88.04%
Email Dataset	75.931%	53.85%
Newsgroup Dataset	94.5752%	78.42%

Table 4.12.: Comparison of the best performing algorithms of MIC and Rainbow on the datasets.

.

ments of newsgroup names makes this approach even more vulnerable to the first problem mentioned: The algorithm has no way to distinguish between `comp` in a newsgroup name, and `comp` somewhere in the body of the message. This problem is solved when we additionally take the structure of the article into consideration. Figure 4.3 shows the top of the decision tree for 1-outermost feature selection. Now, the hierarchy of the decision tree matches the hierarchy of newsgroups.

Comparison to Rainbow

The performance of the different algorithms of Rainbow on the newsgroups dataset is shown in table 4.11. Some of the classification methods provided by Rainbow have not been included, because Rainbow crashes when trying to use them. Rainbow achieve the best performance on the *newsgroups* dataset for the `Shrinkage with Naive Bayes` classifier with an accuracy of 78.42%. Again, MIC outperforms Rainbow. Using Decision Tree learning with 1-innermost-tag feature selection, MIC classifies 94.5752% of the documents correctly. The comparison becomes more adequate when we compare the same classification algorithms. The only algorithm used both by MIC and Rainbow is Naive Bayes. For Naive Bayes classification, MIC using 1-outermost feature selection achieves an accuracy of 87.2057%, while Rainbow only achieves 77.16%.

Table 4.12 shows the best performance achieved by MIC and by Rainbow on each of the datasets. MIC performs better on all of them. Additional, a subset of the email dataset has been compared to SERpersonalBrain. Here, MIC's best result is a precision of 78.4431%, whereas SERpersonalBrain only achieves 70.6%.

5. Conclusions

In chapters 2 and 3, we developed a formal framework for the automatic classification of texts. Based on these definitions, automatic text classification has been improved in two ways: We showed how a subset of input features can be selected from a larger set of potential features while preserving as much information as possible. Our approach, based on Decision Trees, preserves more information than commonly used methods, like selection by plain information content and filtering out most and least frequent words.

Secondly, we developed a input feature representation which preservers the structural information of texts. Our method only affects the input feature representation of the texts. It can be used with the same standard classification algorithm used for the classification of unstructured texts.

In chapter 4, we measured the performance of our methods and compared it to standard methods and third party text classification systems. There is no one-matches-all algorithm for automatic text classification. Depending on the problem, a different combination of automatic classification algorithm and feature selection function works best. On all datasets used for testing, MIC outperformed the third-party text classification systems.

The methods developed for MIC have been shown to perform better than standard text classification methods. There is still room for improvements. So far, the transformation between plain and structured text drops the attributes of the structure tags. A further field of research is to look for methods to preserve this information. The feature reduction method based on Decision Trees seems to limit the number of features too much. We can expect better results by combining different feature selection methods.

In HTML pages, structural descriptive tags are mixed which tags describing the appearance of parts of text. This makes not well suited for structural treatment. When real structural descriptive languages like XML proliferate, we expect the significance of classifiers using structural information to increase.

In this part of the thesis, we developed a working methodology for the classification of structured text. In the next part, we show the software system developed in order to set these methods into practice.

Part II.

MIC

The first part of this thesis is about the automatic classification of structured and unstructured texts. We show our approaches and compare them to the results obtained from traditional text classification systems. In the second part of the thesis, we describe MIC, the text classification system developed for this work. MIC has been developed for a couple of reasons. We need a reference implementation in order to test the concepts from chapters 2 and 3, and to measure the results shown in chapter 4. The second reason for developing MIC is that we need a real, working application to solve the tasks described in chapter 1.

This part of the thesis is split into two chapters. The next chapter describes the usage of MIC in the different scenarios described in chapter 1. Chapter 7 gives a description of the internal structure of MIC. Appendix A shows in detail how MIC is set up for the classification of email.

6. Using MIC

In chapter 1, we describe three applications for our text classification system:

- A stand-alone program for general text classification purposes.
- Part of the specialized Internet search engine MIA, where it has to classify web pages.
- A system for automatic email classification.

In this chapter, we show how MIC is applicable to these three scenarios. We start with the usage of MIC as a stand-alone application. In difference to the areas where MIC is embedded in other software, the standalone application is the only scenario where the graphical user interface is used. The GUI gives a direct, visual view on the elements of MIC.

6.1. Using MIC as a Standalone Program

6.2. General Purpose Text Classification

Modern personal computers come with gigantic storage space. A new computer nowadays (March 2001) has at least 15 GBytes of hard disk storage. Even when using the computer only for personal matters, a user stores large collections of text data on her computer. Data from the Internet in form of web pages, Usenet messages and personal email gets stored, as well as data from other sources, like CD-ROMs with newspaper archives, e-books and encyclopedias. The hierarchical file systems of modern operating systems have little means for helping the user to manage these data collections. Database systems allow fast and consistent data storage and retrieval, but are little used in non-professional environments, because databases need a well thought layout and regular maintenance. A personal user hardly puts the necessary efforts in creating and maintaining a database system. The intrinsic inhomogeneity of data from various sources makes it practically impossible to use a database as a generic storage medium for all kinds of information.

A solution to this problem is the use of advanced tools for the retrieval of data stored on an ordinary file system. A commonly used approach is keyword search. For example, the program glimpse [8] uses an index of all words in all files in order to do quick keyword searches. The use of keywords has limitations. In a lot of cases, it is not possible to give explicit keywords, but one can point to examples for the kind of texts one is looking for.

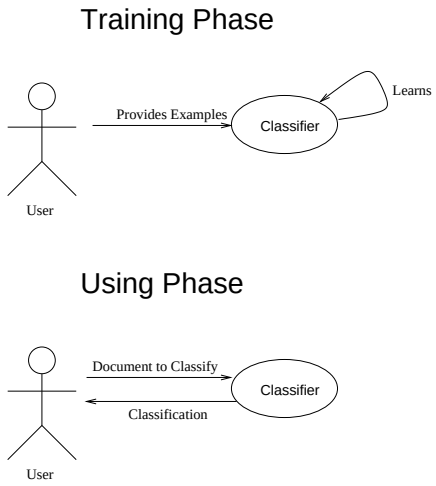

Figure 6.1.: Use case diagram for generic text classification tasks

View	Classification System Element
Document Set View	Texts, Sets of Texts, Classifications
Output Vector View	Output Vectors, Output Vector Components
Classifier View	Classifiers
Statistics	Classification

Table 6.1.: Views and their corresponding elements of the classification system

In these cases, our automatic classification system MIC can be used. MIC provides a GUI where one can define categories and example documents for the categories. After the system has trained on the examples, it can be used to find similar documents. Figure 6.1 shows a use case diagram for generic text classification tasks.

When we describe the usage of MIC as a standalone program, we will use the newsgroup article classification example [23]. 100 documents from each of 20 newsgroups are given. The dataset is randomly splitted in two halves. One half is used for training. The other half is the testing dataset. The task for the system is to categorize this second dataset correctly.

6.2.1. GUI Elements

The GUI follows the common structure of a KDE application. It is divided into three parts: The top part contains the menu bar. The menu bar has common elements for all KDE applications. Foremost, the menu bar allows to load and save files. The icons on the tool bar, located by default under the menu bar, give direct access to the most important functions. The largest part of the screen is occupied by the *view* of the object. The view is individual for each application. In this chapter, we describe the views used in MIC. Figure 6.2 shows the menu bar and the tool bar. Under the tool bar, it shows

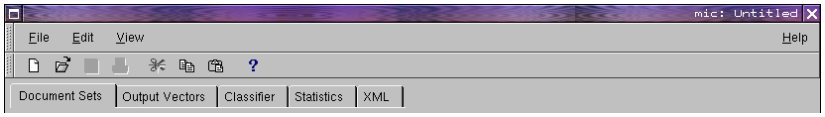

Figure 6.2.: MIC's menu bar, tool bar, and tab bar

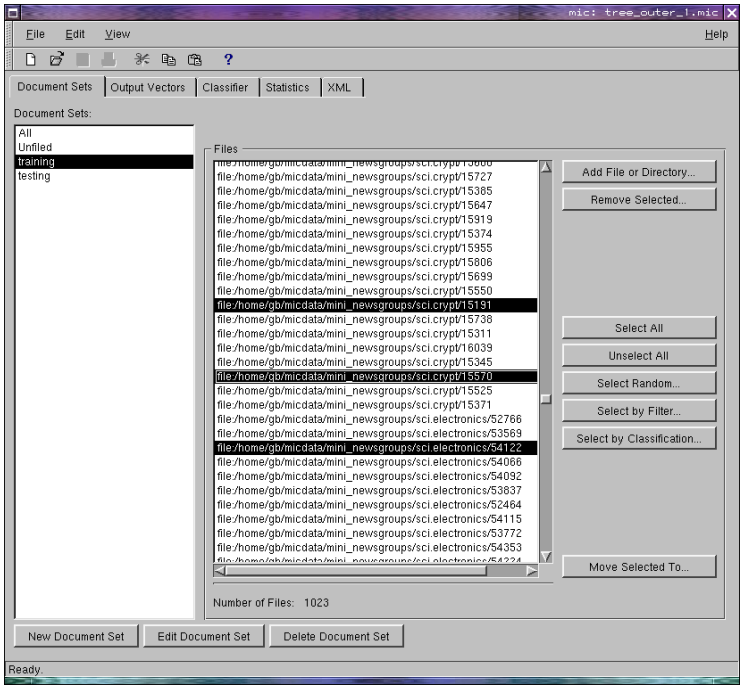

Figure 6.3.: Document Set View

the tab bar which is the main navigation element in the MIC view.

In chapters 2 and 3, the following elements of a text classification system have been identified: Texts (page 39), sets of texts (examples, page 21), classifiers (page 19), classifications (page 19), output vectors (page 20), output vector components (page 19), and feature selection functions (page 18). MIC's GUI reflects these formal elements. It is divided into five *views*. Each view gives access to one or more elements of the text classification systems. The tab bar shown in figure 6.2 is used to switch between the views. Table 6.1 shows which elements of the classifier are accessed via which view.

Document Set View

The document set view manipulates texts and document sets. Document sets can be created and deleted. Texts can be assigned and removed from document sets. Usually, at least two document sets are needed for a text classification run: One document set contains the texts used for training the classifier. The second document set contains the texts for testing. These are the text which shall be classified. In order to make the creation and manipulation of text sets more simple and flexible, different methods are

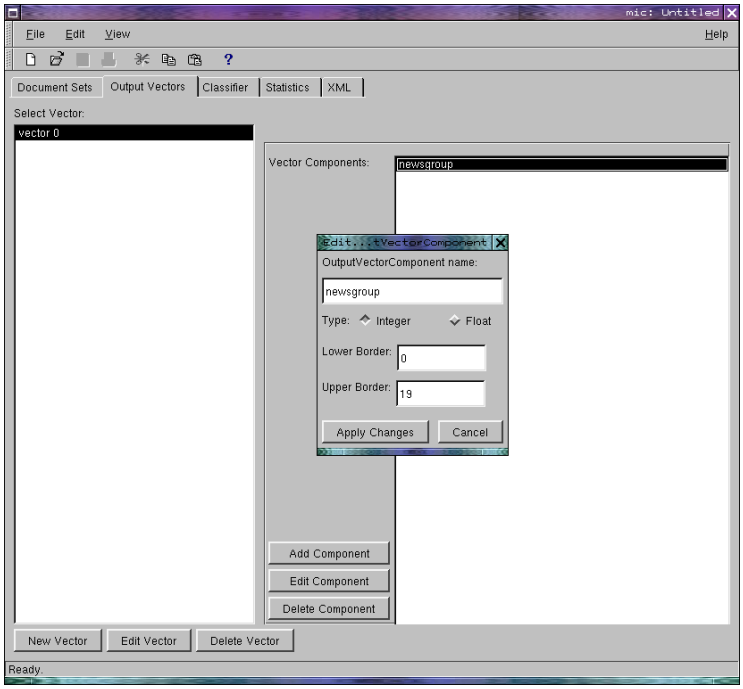

Figure 6.4.: Output Vector View

available to select and move texts between document sets. Texts can be selected by a pattern in their file name, they can be selected by their classification and they can be selected randomly.

There are 2000 texts in the newsgroups example. Two document sets, "training" and "testing" are created. The texts are randomly assigned to the two document sets.

Output Vectors View

The output vector view, shown in figure 6.4, manipulates the output vector components (categories) and output vectors. Output vector components are defined as intervals, in which each number of the interval represents a category (definition 3 on page 19). MIC allows to define categories not only as integer values, but also as floating point values. This allows the categorization in non-discrete categories. At this point, none of the currently implemented classifier handles floating point valued categories.

Since the newsgroup articles from the example are classified only in regard to their newsgroup, the output vector has just one component. The domain of the vector component is the integer interval $[0, 19]$, representing the 20 newsgroups.

Classifier View

The classifier view provides the interface for the interaction with the various classification methods. The four elements on top of the view, shown in figure 6.5, are common for all

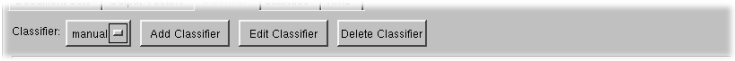

Figure 6.5.: Common elements of classifier views

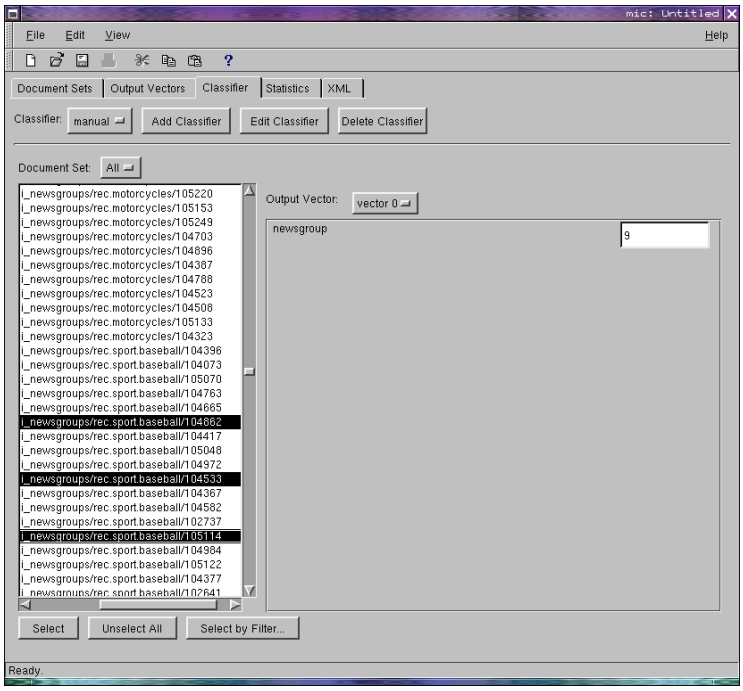

Figure 6.6.: Classifier View: Manual Classifier

classification methods. These elements allow to add, delete, and change the classification methods. The lower main part of the view is provided by the actual classification method. There are four classification methods available.

Manual Classifier

In the manual classifier, the classifications of the texts are done manually. The interface for the manual classifier is shown in figure 6.6. In the left part of the screen, single or multiple texts are selected. Like in the document set view, the texts can be selected manually or by matching filename patterns. In the right part of the screen, the classification of the selected texts is set.

Usually, a manual classifier serves as the reference classifier (see definition 7 page 21) when another classifier is trained.

In the newsgroup example, the articles from each of the newsgroups are stored in a subdirectory carrying the name of the newsgroup. All texts from one newsgroup can be selected by pattern matching on the filenames, where the name of the newsgroup is

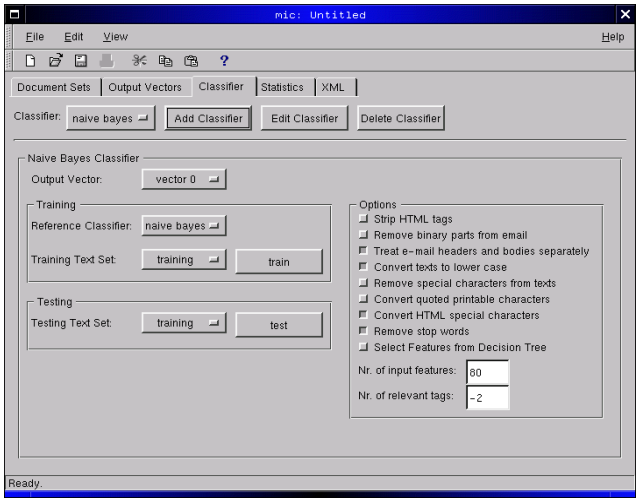

Figure 6.7.: Classifier View: Naive Bayes Classifier

used as the pattern. When the selection is done, the selected texts are assigned to the corresponding category.

Naive Bayes Classifier, Decision Tree Classifier

Figure 6.7 shows a screenshot of the Naive Bayes classifier view. This view and the view for the Decision Tree classifier are very similar. They are split into two parts. The parameters of the classification algorithm are set in the left part of the screen. Since both classification methods are learning classifiers, they have several parameters in common. These are the selectors for the output vector, the reference classifier, the training document set and the testing document set. The training set selection defines which document set is used for training, and the testing set selection defines which document set is used for testing. The reference classifier gives the correct classification of the texts in the training sets when the algorithm is trained (see definition 7 on page 21). Usually, the reference classifier is a manual classifier.

On the right side of the screen, the feature selection can be manipulated. Changing these parameters changes the feature selection function as described in chapter 3. The methods for limiting the number of input features as described in chapter 3.2 are available only for the Naive Bayes classifier. For Naive Bayes classifier, one can choose how many input features to use and how they shall be selected. Either, features can be selected by the "plain" information content of the words or by the information content extracted from a Decision Tree. The Decision Tree classifier always uses as many features as possible.

There are two modes of operation for the classification function. In *training* mode, initiated by pushing button "training", the classifier is trained from the training text set. In *testing* mode, initiated by pushing button "testing", the classification function is applied to the set of documents in the testing text set.

Neural Network classifier

The Neural Network classifier is similar to the Naive Bayes, but there are additional parameters specific to this classifier. The number of hidden nodes for the network can be selected, and the learning rate. (See chapter 2.5 for details about selecting the number of hidden nodes and the learning rate.)

For the newsgroup example, we choose a Neural Network classifier with 50 hidden nodes and a learning rate of 0.2. We choose all the standard feature improvement methods, plus the outermost structure information. For the newsgroups messages, this structure information is about whether a word is in the header of the article or in the body.

When the network is trained, the documents from the testing set can be classified. When this is done, the user has two choices how to proceed. One opportunity is to switch back to the document set view. Here, the user can check the classification of single documents or a group of documents from the testing set. For example, she could decide to select all texts from the testing set who have been assigned to newsgroup `alt.atheism`, and move them to a new document set for further processing. Alternatively, the user can switch to the *statistics view*, where some statistical measures about the accuracy of the classification can be calculated.

Statistical View

This view shows some statistics about the performance of a classifier. The selection boxes on the left side of the screen are similar to those used in the *neural network classifier* and *Naive Bayes classifier*. In these boxes, the reference classifier and the test classifier are set. The output vector box selects the target domain. The *precision* and the *recall* of the testing classifier are calculated as given in definitions 23 and 24 on page 54. The results are written in the text box on the right screen. The results of consecutive calculations (for example, when classifiers are compared), are appended to the text box. A screenshot of the statistical view is shown in figure 6.8.

XML

The last element of the tab bar gives direct access to the XML representation of all objects of the current document. This view is mainly for debugging purposes.

6.3. Using MIC as an Agent

In order to use MIC embedded in an email classification system and within MIA, we need a programmable user interface instead of a GUI. A variety of methods is available for interconnecting software components. Since MIC is based on the KDE desktop environment, the most obvious choice would be to access MIC via the KDE component architecture KParts [17]. We did not choose this approach, because it allows only to communicate in between KDE applications, but both of our target applications, MIA and the email classification system, have not been developed in the KDE framework. MIA does have a very flexible, agent based component system. Therefore, we implemented an agent interface in MIC.

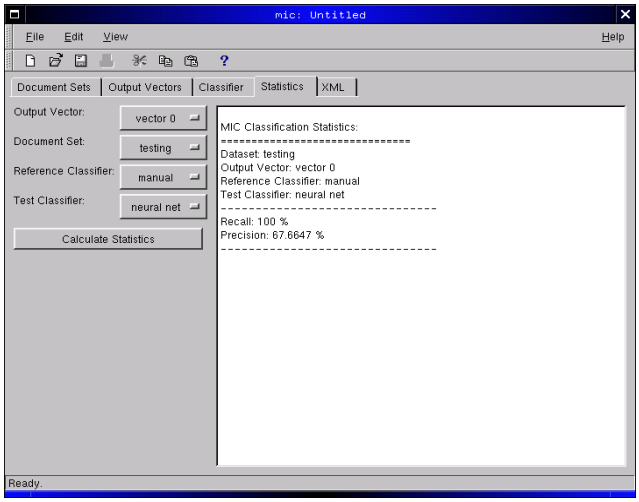

Figure 6.8.: Statistics View

6.3.1. Agent Interface

For the agent interface, MIC needs a communication protocol and a communication channel to exchange information with other agents. The MIC agent component supports two communication channels. The first channel is the program's standard input and output. This channel allows to access MIC via a *Command Line Interface (CLI)*. Unless standard input and output are redirected, MIC reads commands from the keyboard and writes them to the text console. This mode allows a human user to use MIC interactively. Using command line redirection for non-interactive access to MIC causes problems, because Unix pipes are not suited for bidirectional communication.[1] In a true agent based application, it should be possible to run the various agents in different hosts on a network. When more than one agent wants to talk to another agent, there should be locking mechanisms preventing interference. All this is not possible when the command line and standard input/output are the only means to communicate for an agent. The proliferation of the Internet established communication mechanisms that are able to handle these requirements. MIC uses the standard protocol(s) TCP/IP as the communication channel for agent based communications.

In the future, the standard agent language KQML [5] will be used for all communication between the MIA agents and other agents. Since the KQML interfaces for MIA are not defined yet, MIC uses a much simpler, proprietary communication protocol for now. Commands are sent to MIC as one line instruction. These instructions consist of a command and a number of arguments. The commands are tightly connected to the internal structure of the MIC system. Commands consist of two parts. The first part denotes the *class* which is called, the second part the *method* in the class. Following

[1] This is for example discussed in [18, page 344–345]

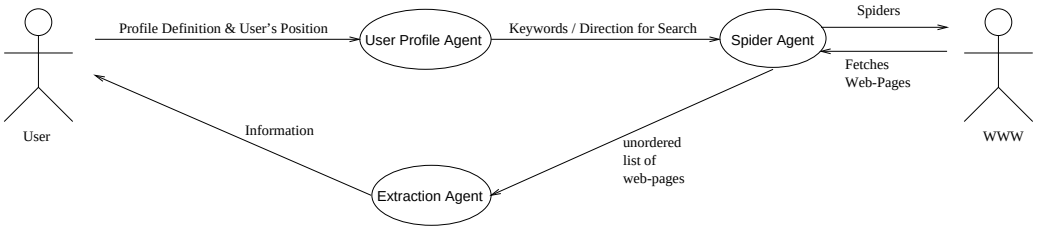

Figure 6.9.: Use case diagram of the interactions between the MIA-agents, the WWW, and the user. This diagram is simplified. There are more agents in the actual system who act in a more complex manner.

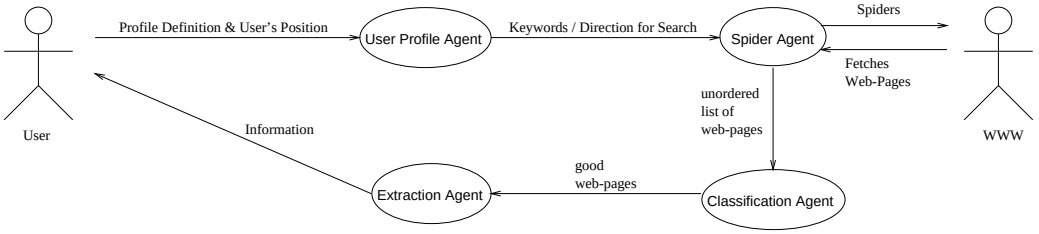

Figure 6.10.: MIA with a classification agent.

parameters are passed as functional parameters to the method. We will have an in-depth look in MIC's structure in the next chapter. Therefore, we will not give a detailed list of commands at this point. It should be noted that MIC is nearly fully scriptable. All relevant methods from all classes can be accessed via the agent interface. In this chapter, we show how MIC is embedded in the other applications.

6.3.2. Using MIC as a Part of MIA

The aim of MIA is to provide a mobile user with relevant information. One part of the MIA system extracts addresses from web pages. When MIC knows the user is looking for Chinese restaurants and found the web pages of a restaurant in the vicinity of the user, the *extraction agent* tries to extract this address information from the web page. The web pages are collected by the *spider agent* which crawls through the WWW. This scenario is shown in figure 6.9. Figure 6.10 shows how the *classification agent* is situated in between the spider agent and the extraction agent. The classification agent improves the performance of the system as described in chapter 1:

- Efficiency

  Extracting information from a web page is a very complex and time-consuming task, even when it fails. More than 50% of the web pages

gathered by the *spider agent* do not contain any address. By sorting out these pages, we can speed up MIA.

- Effectiveness

  When trying to extract information from a database, we can use different methods. Some of the methods give very accurate results (when they extract something, it is most probable an address), but fail to extract an address from time to time. Other methods are more "loose". They do not disregard addresses, but from time to time they extract something which is not an address at all. When the classifier provides an confidence measure about how likely it is that a page contains an address, we can select the method to use for extraction more accurately.

Two steps are necessary to incorporate MIC into MIA. First, we have to set up and train a classifier. Second, the classifier must be put into the MIA system.

The training step can be done offline via the GUI. For training, the dataset of classified web pages described in chapter 4.1 is used. This is a set of 973 texts gathered by the MIA spider module. There are two categories: Either a text contains an address, or it does not contain an address. The pre-classification for the learning algorithm is provided by a reference classifier. This has been the non-trainable *zip + city* algorithm, which is described in the chapter 4.1.1. We choose the best-performing classifier and feature selection to be trained on these data. This is a Decision Tree classifier with improved feature selection. The results of the training are stored in a file.

Within MIA, MIC is used as an agent. MIA only needs a few commands to communicate with MIC: On startup, MIC has to load the file with the trained classifier. After this, MIC waits to get the command to assign a classification to a text and returns the classification.

MIC is accessed as a client/server application in this scenario. The server process is started, and the trained classifier is loaded. The server listens to a socket. Whenever a new web page is retrieved by the spider agent, it connects to the MIC server and requests the classification of the text. The server classifies the text and returns the result. Only if the web page is classified as containing an address, it is passed to the information extraction agent.

6.3.3. Using MIC as a Part of an Email Classification System

One practical application of MIC is the automatic classification of email. Figure 6.11 shows a use case diagram for the classification agent in a mail-system.

There are already numerous systems available for the automatic filing of emails. These systems filter email by pattern matching. For example, emails are filtered by a pattern in the `From:` or `Subject:` line. These systems are mainly used to separate personal mail from bulk mail, like messages from mailing lists. They fail when the categories are loose and not strictly defined in term of patterns. A common example of such a set are UCE/UBE[2], also known as Spam. No user wants to see all the unsolicited advertisement

[2] Unsolicited Commercial Email / Unsolicited Bulk Email

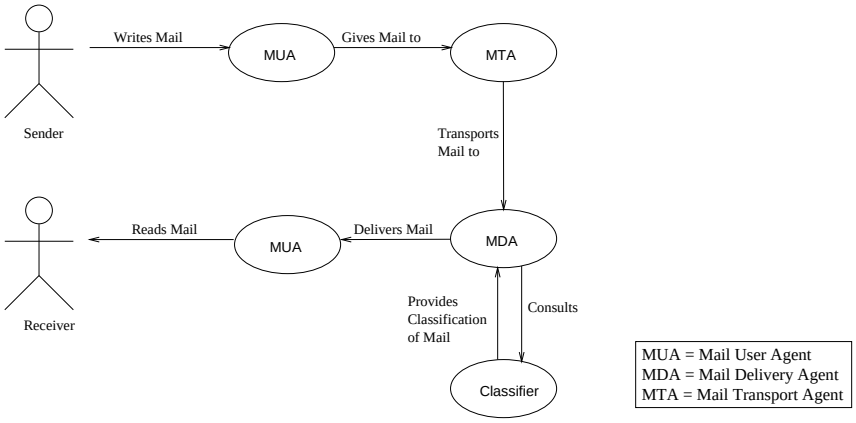

Figure 6.11.: Use case diagram of a mail-system with a classifier agent.

emails. These emails can not be filtered out easily by matching for specific patterns, because those who send it take active counter measures against keyword based filtering.

MIC is better suited to filter out these kind of email, because its methods take the whole message into consideration and not only some keywords.

There are other areas where MIC's methods are better suited than simple pattern matching methods. All bigger Internet entities have a number of accounts via which a user can gets in touch with the staff. Common names for these accounts are `abuse`, `admin`, `root`, `usenet`, or `postmaster`. A lot of work is saved when mail send to these generic account is automatically forwarded to the person who is responsible to answer requests of a certain type. Since the user will usually not use predefined forms in her email, but freely written text, strict pattern matching is not very well suited to do this classification.

Just like in the MIA application, MIC is trained offline for classifying email before it is incorporated into the MDA. The email dataset from chapter 4.5.3 is used for training. The trained classifier is saved to a file. Just like in the MIA scenario, the actual classification of the email is done in a client/server environment. A MIC server process is started, and the trained classifier is loaded. Whenever a new email arrives, the MDA calls a small client application which passes the email to MIC, and stores the email according to MIC's classification. The details of this process are explained in appendix A.

7. MIC System Description

This chapter shows the internal structure of MIC. Since MIC is object oriented, the structure of the chapter follows the object oriented paradigm of UML (Unified Modeling Language) [2]. UML is the standard of object oriented software design. An UML description consists of a set of diagrams of different types. The diagrams describe different aspects of the software project. In this chapter we make use of four UML diagram types: Use-case models, finite state machines, class diagrams and sequence diagrams.

We start by transforming the formal elements of a text classification system, as developed in the first part of the thesis, into objects. The objects are grouped together in a hierarchy. The main part of this chapter describes the objects. We start with the base classes, which define common elements of all MIC objects. A detailed look on the different objects follows.

UML does not give instructions how to transform a real-world system into an object oriented model. We use another concept from software engineering, which is not part of UML, for this transition. The formal elements of the text classification system are modeled in an Entity-Relationship model(ER model). This ER model is transfered into relational tables. After some normalization, these tables are transfered into objects.

7.1. Concepts

7.1.1. Entity-Relationship-Model of the data

For an object oriented model of the text classification system, we have to identify the data elements processed in a text classification task and their relationships. In chapters 2 and 3, the following elements of a text classification system have been identified:

- Single texts (definition 16 page 39)
- Sets of texts (examples, definition 6 page 21)
- Text classification functions (definition 4 page 19)
- Feature selection functions (definition 2 page 18)
- Classifications of texts (definition 1 page 18)
- Output vectors (definition 5 page 20)
- Output vector components (definition 3 on page 19)

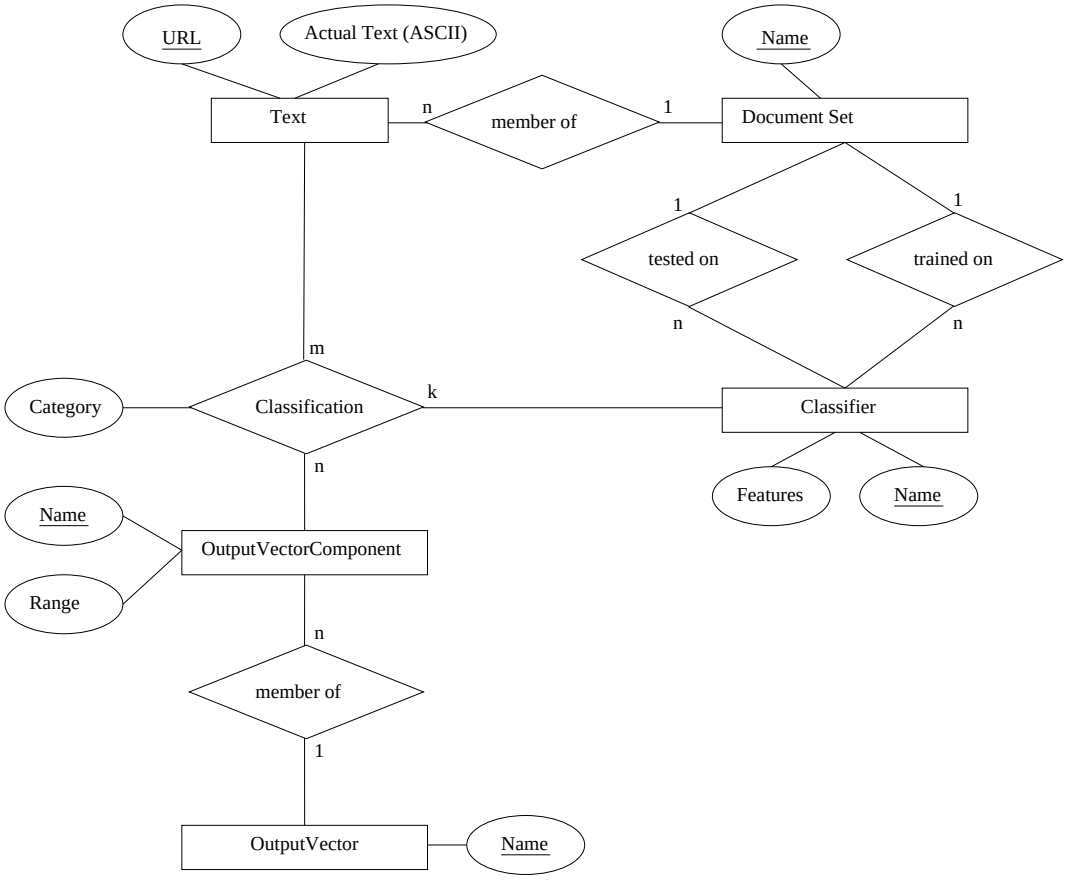

Figure 7.1.: Entity-Relationship diagram of MIC's data structures

Figure 7.1 shows an entity relationship model based on this table. There are numerous variants of entity relationship models around, which differ in some syntactical detail. We use the notation given in [40]. Some of the table's elements correspond directly to entities of the ER-diagram. *Texts* are entities with two attributes. The first attribute is the URL of the text, which gives a unique key for the entity. The second attribute is the actual text itself. *Text*s are aggregated into *document sets*. *Document sets* have a name as there only attribute and they have a $1 : n$ relation to texts; one document set can contain multiple texts, but each *text* is a member in only one *document set*. *Text classifiers*, have relations to the *document set*s. One document set is used for training, and one document set is used for testing.

According to definition 3 on page 19, sets of categories are represented as intervals of integer values. The borders of this interval are stored in the range attribute of the output vector component entity.

In definition 4 on page 19, text classification is defined as a function from the feature representation of a text to a classification. This translates into a relation between a *text*, an *output vector component*, and a *classifier* in the ER-diagram.

From the ER-diagram, it is a small step to the object design. First, the ER-diagram is transformed into a relational database model. This is a straight-forward task which is described in every text book on database design. (See for example [40, page 134].) The relational tables are shown in table 7.1. Each table of the relational database corresponds to an object of the system. Before we can transform the tables into objects, they have to be *normalized*. Normalization is a standard method in database design in order to avoid *anomalies*. An *anomaly* occurs when a data base gets inconsistent because of an operation, or if an operation can not be performed because the database design is too restrictive. In the relational tables in table 7.1, *insertion anomalies* and *deletion anomalies* can occur. An insertion anomaly occurs when a new text (with URL and actual text) shall be defined without assigning it to a document set. With table 7.1 this is not possible, because both the URL and the name of the document set are key attributes of table *Document Set / Text*. The deletion anomaly is symmetric to the insertion anomaly: When a document set is deleted, also all the texts in this document set must be deleted.

Anomalies are avoided by transferring the tables into normal forms. This is done — just as the transformation from the ER-model to the relational tables — by a standard method. Table 7.2 shows the normalized form of the tables. From this table, the main classes of the MIC system are derived.

7.2. Base Classes

In object oriented design, objects are arranged in a hierarchy. The hierarchy of MIC classes is shown in figure 7.2. Objects in lower positions in the hierarchy inherit the properties from the objects at the upper positions. The base class of all MIC objects contains the functionality which is common to all MIC objects. For MIC, these are two properties: In order to load and save a setup to a file, an object must be able to

Document Set / Text	Name (Document Set) URL Actual Text (ASCII)
Classifier / Document Sets / Features	Name (Classifier) Name (Document Set (Testing)) Name (Document Set (Training)) Features
Output Vector / Output Vector Component	Name (Output Vector) Name (Output Vector Component) Range (Output Vector Component)
Classification / Text / Classifier / Output Vector Component	*URL (Text)* Name (Classifier) Name (Output Vector Component) Category

Table 7.1.: Tables created by transforming entity-relation-model shown in figure 7.1

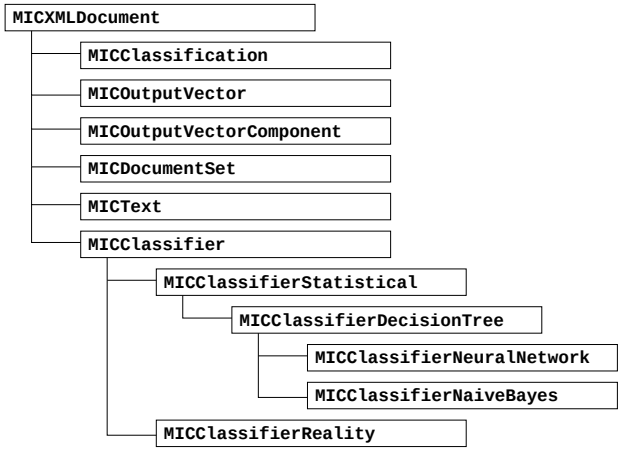

Figure 7.2.: Class Hierarchy

Document Set / Text	Name (Document Set) URL
Text	URL Actual Text (ASCII)
Classifier / Document Sets / Features	Name (Classifier) Name (Document Set (Training)) Name (Document Set (Testing)) Features
Output Vector / Output Vector Component	Name (Output Vector) Name (Output Vector Component)
Output Vector Component	Name Range
Classification / Text / Classifier / Output Vector Component	URL (Text) Name (Classifier) Name (Output Vector Component) Category

Table 7.2.: Table 7.1 transfered to second normal form.

Object	Corresponds to Table
`MICDocumentSet`	Document Set / Text
`MICText`	Text
`MICClassifier`	Classifier / Document Sets / Features
`MICOutputVector`	Output Vector / Output Vector Component
`MICClassification`	Classification / Text / Classifier / Output Vector Component

Table 7.3.: Correspondence of MIC classes and relational tables.

write its current state into a text representation which can be stored in a file. The second property has to do with the agent interface of MIC. In order to use MIC non-interactively, all classes need an interface for batch processing.

7.2.1. XML

The superclass of all MIC classes is class `MICXMLObject`, which itself inherits from class `Qbject`, the base class of all objects derived from the QT library [29]. As the name suggests, the base class defines an interface for loading and storing instances of the class in XML format. Since this interface is inherited by all MIC objects, all objects can save and restore themselves from an XML document. By this, a defined way exists to transfer instances of MIC classes in and out of MIC. For now, MIC objects can be loaded or saved on storage media. In future version, the interface will be extended for the exchange of objects with other applications. XML has been chosen as a flexible, well accepted standard format for structured information. The DTD for MIC documents is shown in appendix C.

7.2.2. Batch Interface

Beside loading and storing the instance of the object as an XML document, `MICXMLDocument` provides a second common interface. This interface is method `batch`. It handles accessing the objects in *batch mode* and `agent mode`. In chapter 6, the methods of accessing MIC are described: Via the Graphical User Interface, and via a the batch / agent interface. The batch / agent interface allows a direct call to methods of the classes via textual strings. These strings can either be provided interactively or in an agent environment by another agent. The batch interface transfers commands into API-calls transparently. The syntax of a command is

```
<Classname>.<methodname> <first parameter> <second parameter> <...>
```

The details of the syntax and semantics of the batch interface are discussed in section 7.5.1, which describes class `MICBatch`. For now, it is only important to note that all classes inheriting from `MICXMLDocument` are scriptable.

7.3. Classes Details

This chapter gives details about the classes. For each of them, the purpose of the class, the batch interface, and the GUI is shown. The DTD for the XML structure of documents is given in appendix C.

7.3.1. The Class `MICText`

Class `MICText` provides the interface to the actual text. As shown in table 7.2, texts are identified via their URL. Using the URL has two advantages: We can be sure to have a unique identifier, even in a global namespace. Additionally, the URL abstracts from the

actual location and method of access. It does not matter if the text is stored on the local hard disk, a web page, a FTP server, or a news server.

The batch interface to `MICText` is minimal. It provides the following calls

MICText::batch		
Command	Parameters	Description
new	*URL*	Creates a new instance of `MICText` and associates it with URL *URL*.
setDocumentSet	*URL, documentSet*	Assigns text `URL` to document set *documentSet*

GUI

`MICText` has not a GUI element of its own. It is embedded in the GUI elements for editing document sets and manual classifications.

7.3.2. The Class `MICDocumentSet`

Class `MICDocumentSet` corresponds to relational table *Document Set / Text* in table 7.3. It defines sets of `MICTexts`. Usually, at least two document sets are used. One defines the training set for the classifier and one the testing set. `MICDocumentSet` provides flexible means to sort lists of texts by common criteria, like their classification, or a pattern in their filename. `MICDocumentSets` are identified by their *name*.

MICDocumentSet::batch		
Command	Parameters	Description
new	*name*	Creates a new document set with name *name*.

GUI

The elements of the GUI have been introduced in chapter 6. The view-element `MICViewElementDocumentSets` provides access to the `MICTexts`, and the `MICDocumentSets`. A screenshot is shown in figure 6.3 on page 78. The left side of the screen shows the document sets. It allows to create new document sets, change the name of existing document sets and delete them. The right side of the window shows the `MICTexts`. Texts can be loaded, removed and assigned to document sets. It is possible to recurse directories when adding texts. Beside manual selection, files can be selected based on patterns in their filenames, randomly, or by their classification. Figure 7.3 shows a Finite State Automaton (FSM) of `MICViewElementDocumentSet`. The FSM is shown in UML notation [2, pp. 248].

7.3.3. The Class `MICOutputVectorComponent`

`MICOutputVectorComponent` stores elements of an output vector, i.e. the set of categories in regard to a specific topic. The categories are represented as intervals of integer or floating point values, as defined in definition 3 on page 19.

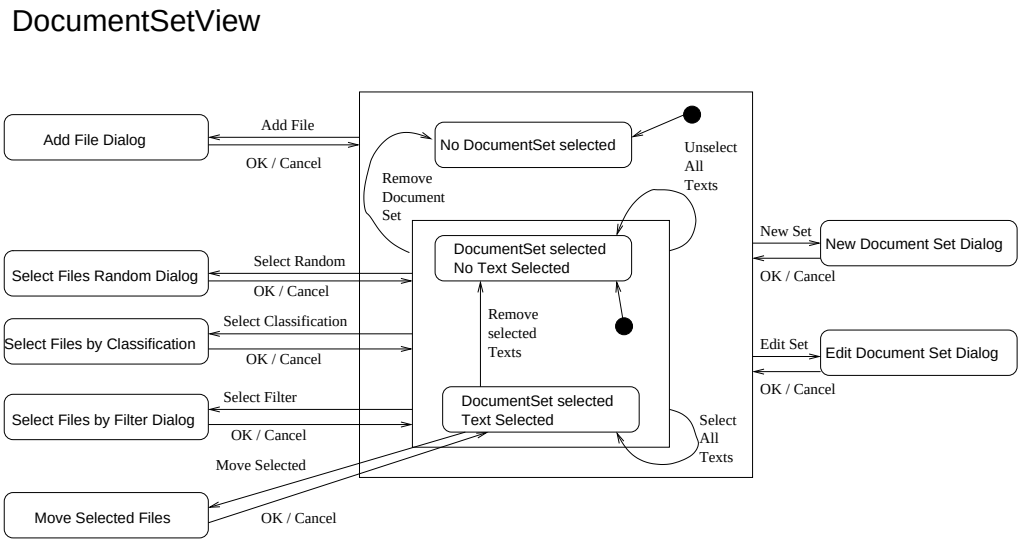

Figure 7.3.: FSM diagram of `MICViewElementDocumentSet`

MICOutputVectorComponent::batch		
Command	Parameters	Description
new	*name*	Creates a new output vector component with name *name*.
setOutputVector	*name*	Associates output vector component with output vector *name*.
setTypeInt	*type*	Set type of the output vector to *integer* or *floating point*.
setBordersTypeInt	*lower, upper*	Set the borders of the category-interval to [*lower*, *upper*].

GUI

The GUI of the output vector component is combined with the GUI of the output vector. It is described in the section about the output vectors.

7.3.4. The Class `MICOutputVector`

The output vector groups output vector components. Since the association between an output vector component and its output vector is stored with the output vector component, `MICOutputVector` — similar to `MICDocumentSet` — only stores its name.

MICOutputVector::batch		
Command	Parameters	Description
new	*name*	Creates a new output vector with name *name*.

OutputVectorView

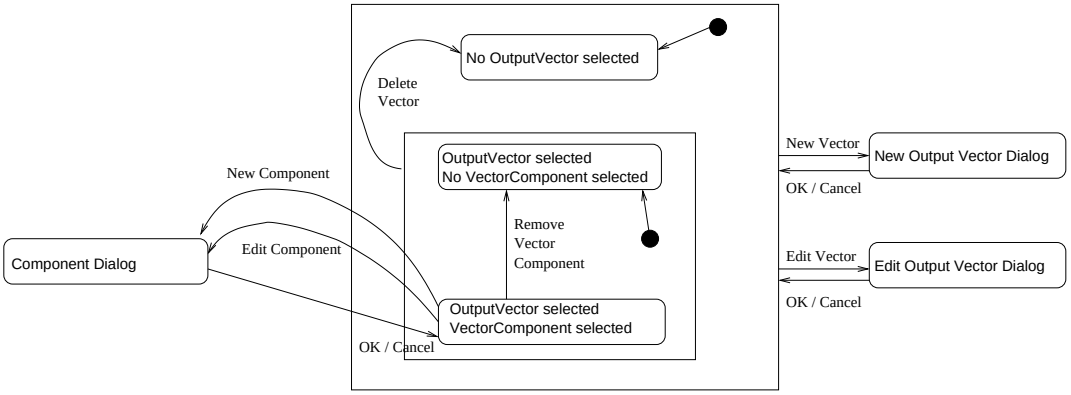

Figure 7.4.: FSM diagram of `MICViewElementOutputVector`

`GUI`
`MICViewElementOutputVectors` give access to the `MICOutputVectors` and `MICOutputVectorComponents`. A screenshot is shown in figure 6.4 on page 79. Output vectors and output vector components can be added, edited and deleted. Every output vector component is part of an output vector. Figure 7.4 shows a FSM of this view element. In this diagram, `Component Dialog` refers to a dialog in which the user can set the domain for an output vector component and its data type. MIC supports both integer valued and float valued output vector components, but so far the implemented classifiers only support integer-valued output vector components.

7.3.5. The Class `MICClassification`

`MICClassification` stores the classification of a text in regard to a specific classifier and output vector component. It is a relation between three elements.

Batch Interface

MICClassification::batch		
Command	Parameters	Description
new	*text, output vector component, classifier, classification*	Assigns classification *classification* to *text* in regard of *output vector component* and *classifier*

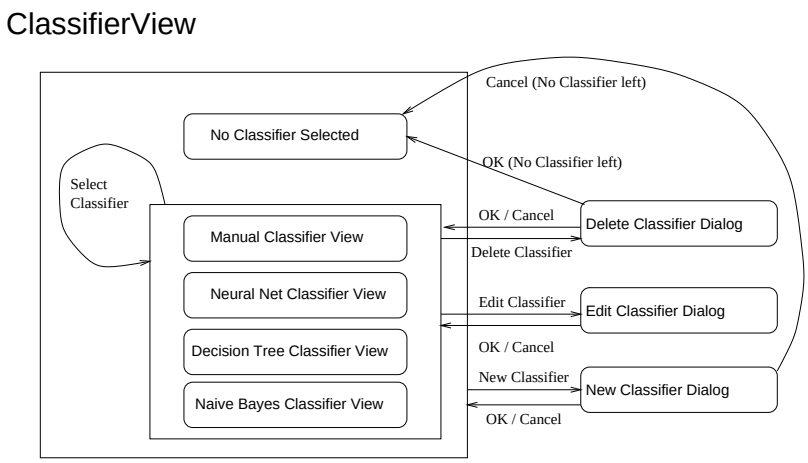

Figure 7.5.: FSM diagram of `MICViewElementClassifier`

GUI

This class has no GUI element of its own. `MICViewElementDocumentSet`, where the texts and sets of texts are manipulated, visualizes this class, too. In `MICViewElementDocumentSet`, texts can be selected by their classification in regard to a classifier and an output vector.

7.3.6. The Class `MICClassifier`

The classes which implement the various classification algorithms have a special status within MIC. For these classes, it is important to integrate them into MIC in a way which makes adding new classes simple. An advanced user can add new classes to the system without too much hassle. We use two levels of inheritance in the structure of the classifier classes. The abstract interface to the classifiers is defined in class `MICClassifier`, which inherits from `MICXMLDocument`. It does not do any classification. It serves just a base class for the real classifiers.

MICClassifier::batch		
Command	Parameters	Description
new	*classifier*	Create a new classifier of type *classifier*.

GUI

The class in charge of visualizing `MICClassifier` is `MICViewElementClassifier`. This view allows access to the classifiers. A screenshot is shown in figure 6.7 on page 81. The view itself provides a two-splitted window. It only controls the upper part of this window itself. In it, classifiers can be selected, created, edited and removed. The lower part of the window belongs to the currently selected classifier. The control of the lower part of the window is the responsibility of the selected classifier. These are discussed with the

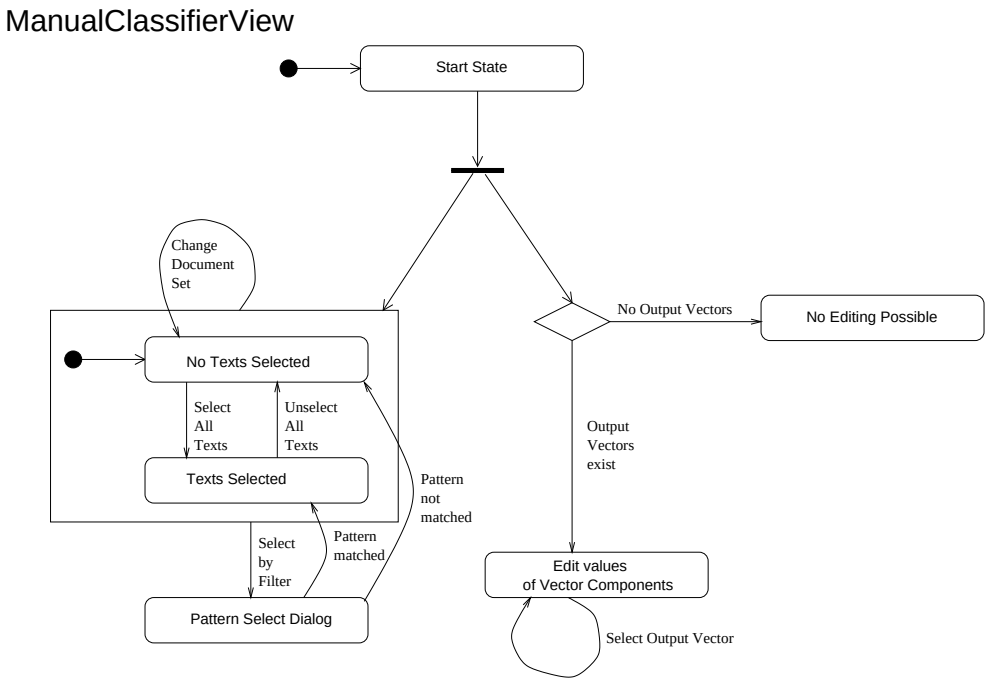

Figure 7.6.: FSM diagram of `MICClassifierRealityWidget`

respective classifiers next. A FSM describing the upper part of the window is shown in figure 7.5. In the `New Classifier Dialog` and the `Edit Classifier Dialog`, the user creates new classifiers by assigning a name and a type to it. The name of the classifier shows up in the classifier selection box. The type of the classifier decides in which of the four states `Manual Classifier View`, `Neural Net Classifier View`, `Decision Tree Classifier View` and `Naive Bayes Classifier View` the FSM switches when the classifier with this name is selected. These sub views, which are shown in the lower part of the window, are described next.

7.3.7. The Class `MICClassifierReality`

This class implements the manual classifier. It inherits from `MICClassifier`. Since it does no classification on its own, this class is fairly empty.

MICClassifierReality::batch		
Command	Parameters	Description
none		

Since this is the manual classifier, there is no need to access it. In order to manually set a classification, one creates an instance of this classifier and associates with a classification, output vector, and a text by creating an instance of `MICClassification`.

GUI

`MICViewElementClassifierReality` visualizes `MICClassifierReality`. This view allows the manual classification of texts. The left part of the screen shows the texts in a document set. The right side shows an output vector. By selecting texts on the left side, and setting the values of the output vector components on the right side, the user assigns classifications to texts. She is aided by three buttons on the bottom of the screen. These buttons allow the selection and unselection all texts in the current document set and selection based on file name patterns. Figure 7.6 shows the FSM of this view element.

7.3.8. The Class `MICClassifierStatistical`

`MICClassifierStatistical` inherits from class `MICClassifier`. One important aspect of text classification is the conversion from a text into an *input feature vector* as defined in definition 2 on page 18. Feature selection is not implemented in its own class, but as part of class `MICClassifierStatistical`.

According to definition 18 (page 40), a text is transfered into a feature vector by using a subset of the distinct words in all texts as components of the input feature vector. The value for each vector component is calculated as the relative frequency of the word in the text. Calculation of the word frequencies is done in class `MICClassifierStatistical`. It provides a method `countWordsInText()` which calculates the frequency of the words in the text. All feature selection methods described in chapter 3 are based on altering the word frequency count. They either map more than one word to a feature vector component or transfer structured text into plain text. All these manipulations are located in method `countWordsInText()`. A number of switches turns on or off certain input feature manipulation methods. A list of these methods is given in chapter 4.3 on page 57. `MICClassifierStatistical` also lays the grounds for learnable classifiers, by adding hooks for a *training*, and a *testing* dataset. The actual classification methods in the child classes reimplementing methods `train()` and `test()` with implementations of their classification algorithms.

MICClassifierStatistical::batch		
Command	Parameters	Description
setOutputVector	*output vector*	Define the output vector on which the classifier is trained.
setStripHTMLTags	*strip tags*	Toggle whether to remove HTML tags from the text.
setRemove-BinariesFrom-Email	*remove binaries*	Toggle whether to remove binary parts when classifying email messages.
setSplitEmail-HeaderAndBody	*structure*	Set if email shall be treated as plain text or as structured text.

setConvertTo-LowerCase	*convert*	Toggle whether to convert all words to lower case.
setRemove-SpecialCharacters	*remove*	Toggle whether to remove special characters (i.e. everything non-alphanumerical) from the text.
setConvert-QuotedPrintable	*convert*	Toggle whether to convert quoted printable characters to their ISO-8859-1 representation.
setConvertHTML-SpecialCharacters	*convert*	Toggle whether to convert special characters from HTML representation to ISO-8859-1 representation.
setRemoveStop-Words	*convert*	Toggle whether to remove (German) stop words from the text.
setReference-Classifier	*reference*	Set reference classifier to *reference*.
setTrainingSet	*training set*	Set training set to *training set*.
setTestingSet	*testing set*	Set testing set to *training set*.
printMost-InformativeWords		Print a list of the most informative words.
setRelevantTags	*number*	Set the number of relevant tags to *number*.
setNrInput-Features	*number*	Set the number of input features to *number*.
train	*train*	Train classifier on training set.
test	*test*	Test classifier on testing set.

GUI

`MICClassifierStatistical` has no GUI element, because it is not used directly. Only instances of the descendant classes of `MICClassifierStatistical` are used.

7.3.9. The Class `MICClassifierDecisionTree`

This class implements the Decision Tree classifier. It is also used by its descendants, `MICClassifierNeuralNetwork` and `MICClassifierNaiveBayes`, to calculate the most informative words by extraction from a Decision Tree as described in chapter 3.2.2. The Decision Tree classifier is implemented by reimplementing methods `train()`, and `test()` from `MICClassifierStatistical`.

MICClassifierDecisionTree::batch		
Command	Parameters	Description
train	*train*	Train classifier on training set
test	*test*	Test classifier on testing set
setKindOf-Information-Selection	*type*	Set information selection method to either *plain* or *Decision Tree*.

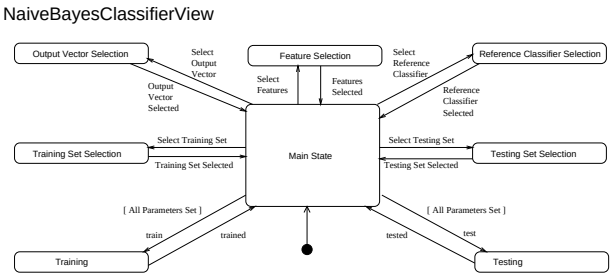

Figure 7.7.: FSM diagram of `MICClassifierNaiveBayesWidget`

GUI

The GUIs of `MICClassifierDecisionTree`, `MICClassifierNaiveBayes` and `MICClassifierNeuralNetwork` vary only in details. We will explain the GUI only once for class `MICClassifierNaiveBayes`.

7.3.10. The Class `MICClassifierNaiveBayes`

`MICClassifierNaiveBayes` implements the Naive Bayes classifier. Just like for `MICClassifierDecisionTree`, methods `train()` and `test()` from the parent class are reimplemented.

Since `MICClassifierNaiveBayes` does not have any special parameters, the batch interface is limited to calls to methods `train()` and `test()`.

MICClassifierNaiveBayes::batch		
Command	Parameters	Description
train	*train*	Train classifier on training set.
test	*test*	Test classifier on testing set.

GUI

The GUI-elements for the classes derived from `MICClassifierStatistical` are very similar. Therefore, we discuss only `MICClassifierNaiveBayesWidget`, the visualization class for `MICClassifierNaiveBayes`. A screenshot is shown in figure 6.7 on page 81. The FSM for this dialog is shown in figure 7.7. This view element has four components:

Output Vector Selector In this box, the output vector is selected. It determines the categories.

Training Document Set Selector Here, the user decides which document set is used for training the classifier.

Reference Classifier Selector Defines the reference classifier. The reference classifier gives the correct classifications for the texts in the training document set.

Training Push Button Trains the classifier.

Testing Document Set Selector Sets the document set used for testing the classifier.

Testing Push Button Tests the classifier.

Feature Selection Options A bar of radio-boxes defining how the input features are calculated.

7.3.11. The Class `MICClassifierNeuralNetwork`

This class implements the Neural Network classifier. The neural network itself is encapsulated in auxiliary class `MICNeuralNetwork`. The Neural Network classifier has some more options than the Naive Bayes classifier and the Decision Tree classifier. The user can set the number of hidden nodes, and the learning rate of the network. The additional parameters can be set via the batch interface.

MICClassifierNeuralNetwork::batch		
Command	Parameters	Description
train	*train*	Train classifier on training set.
est	*test*	Test classifier on testing set.
setNrHiddenNodes	*nodes*	Set the number of hidden nodes of the Neural Network.
setLearningRate	*rate*	Set learning rate to *rate*.

GUI

See GUI section of `MICClassifierNaiveBayes`

7.4. KDevelop Template Classes

The general structure of the GUI-subsystem of MIC is given by the templates provided by KDevelop [16], which have been used in the development of MIC. KDevelop is as an IDE for building KDE applications. By default, a KDE program consists of three components: the *document*, the *view* and the *application*. In MIC, these are the classes `MICDoc`, `MICView` and `MICApp`.

7.4.1. The Class `MICDoc`

An instance of `MICDoc` is an aggregation of sets of instances of the previously described classes. It contains sets of `MICDocumentSets`, `MICTexts`, `MICClassifications`, `MICOutputVectors`, `MICOutputVectorComponents`, and `MICClassifiers`. A `MICDoc` is a configuration of the classification system. Class `MICDoc` gives access to all these other classes, and it has method to load and store them.

Batch Interface

MICDoc::batch		
Command	Parameters	Description
newDocument		Create a new (empty) document.

closeDocument		Remove document from memory.
openDocument	*URL*	Load document *URL*.
saveDocument	*URL*	Save document to *URL*.
getDocumentAsXML		Write document in XML format to stdout.

GUI

`MICView` provides the GUI of `MICDoc`. It arranges the GUI elements of the objects contained in a `MICDoc` via a tab bar. This tab bar is shown in figure 6.2 on page 78.

7.4.2. The Class `MICApp`

`MICApp` runs the graphical user interface. It is is in charge of the menu bar, and tool bar, which are shown in figure 6.2 on page 78. It also handles loading and saving `MICDocs`.

7.5. Auxiliary classes

Beside these core elements, there is a number of additional classes. We only document the most important ones here. These are `MICBatch` and `MICStatistics`. `MICBatch` passes the batch processing commands to the batch interfaces of the respective classes. `MICStatistics` provides some statistical calculations on the performances of the classifiers.

7.5.1. The Class `MICBatch`

`MICBatch` handles the batch processing of the MIC system. It is for the batch interface what `MICApp` is for the GUI interface: It passes method calls to the right instances of the objects.

When MIC is stared, a command line parameters determines the mode of operation. Either the GUI is invoked via `MICApp`, or the CLI / agent interface via `MICBatch`. When MIC is invoked as an agent, `MICBatch` controls the application. It reads commands from either standard input or a socket. The commands are passed to the batch interface of the appropriate class. The result of the batch process is written back to standard out or the socket. Since all processes can connect to a socket, `MICBatch` also authenticates the client. This prevents unauthorized clients from issuing commands to MIC.

Appendix A shows a practical example of the use of the agent interface: MIC is used as an email classification agent in the MDA of a Unix-like system.

7.5.2. The Class `MICStatistics`

Description

MICStatistics calculates the recall rate and the precision of a classification as defined in definitions 23 and 24 on page 54. In order to calculate these values, the user has to set the output vector, the document set, the reference classifier, and the testing classifier. In

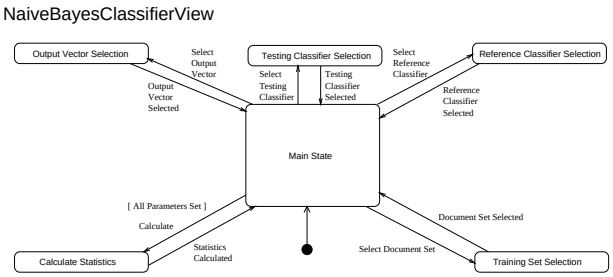

Figure 7.8.: FSM diagram of `MICViewElementStatistics`

the calculation, the classification of the testing classifier on the document set in regard to the output vector is compared to the classification by the reference classifier.

Batch Interface

MICStatistics::batch		
Command	Parameters	Description
calculateStatistics		Calculate precision and recall rate.
setDocumentSet	*name*	Set document set to *name*.
setOutputVector	*name*	Set output vector to *name*.
setReferenceclassifier	*name*	Set reference classifier to *name*.
setTestClassifier	*name*	Set test classifier to *name*.

GUI

`MICViewElementStatistics` provides the GUI for `MICStatistics`. A screenshot is shown in figure 6.8 on page 83. This GUI element is quite simple. The document set, output vector, reference classifier, and testing classifier can be selected, and the statistics can be calculated. Results are concatenated to the text box on the right side of the screen. The FSM of this view is shown in figure 7.8.

7.6. Class Diagram

Finally, we group together all classes and create a class diagram. It is shown in figure 7.9. The main structure is observable from the diagram: There are classes for each of the tables identified in the entity-relationship diagram. The various elements of the classification system are held together by class `MICDoc`. `MICDoc` contains all the texts, classifications, classifiers, output vectors, output vector components and document sets for a certain classification task. On the other side of the diagram, we see the elements of the GUI. `MICView` is managing the view element classes. When the GUI is used, MIC is accessed via `MICApp`. In batch processing, `MICBatch` handles the commands. The batch interface does not connect to the GUI.

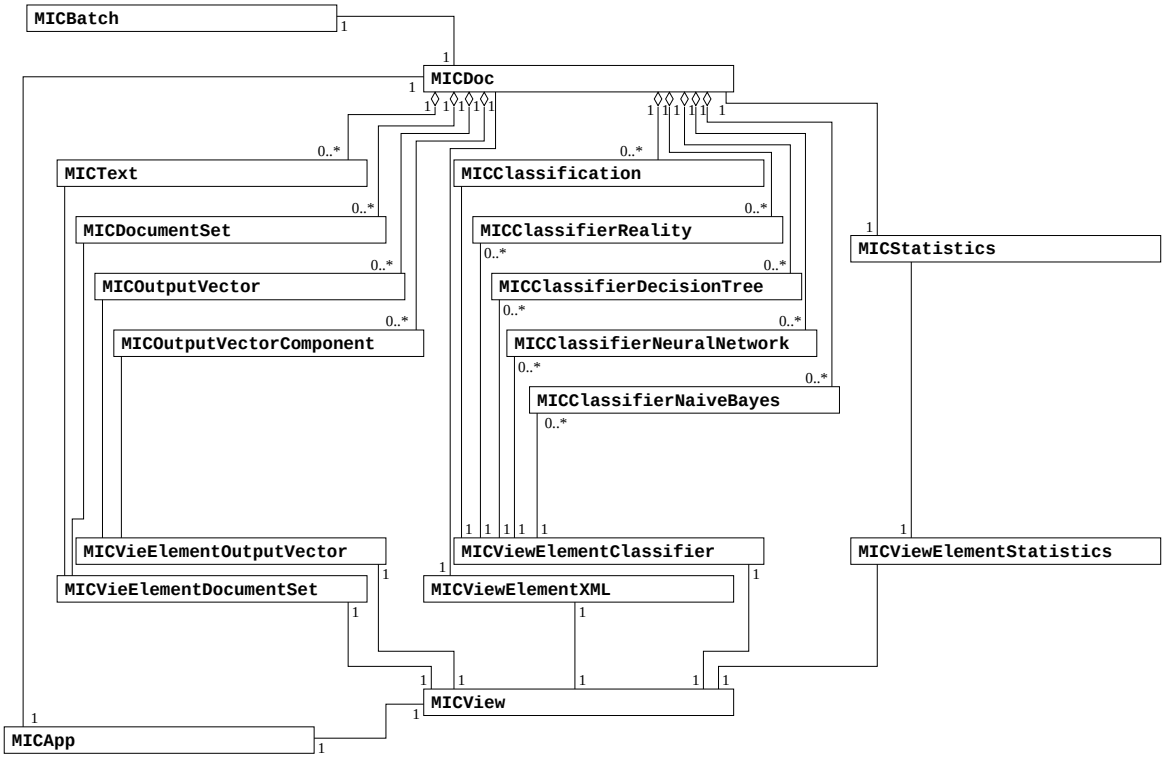

Figure 7.9.: Class diagram for MIC

7.7. About the Platform

We base the application on widely available and reliable tools. MIC has been developed on GNU/Linux [20] systems (although it should compile on any Unix-like system) in C++, using the QT [29] and KDE [15] libraries. (GNU)-C++ combines an object oriented paradigm with a good compiler and great auxiliary tools for developing, debugging and profiling applications. The QT library provides a rich set of functionality for general purpose C++ programming. It is especially suited for the development of applications with decent GUIs. KDE is one of the de facto standard desktop environments for Unix-like systems. All software used for the development of MIC is licensed under the GPL [11]. So is MIC itself.

8. Conclusions About MIC

MIC has been used successfully on various classification tasks. There is no other free software tool around which provides the flexibility of MIC, both in terms of the classification methods and the user interfaces. The classification methods implemented in MIC have been shown to outperform all other freely available text classification systems.

In its current implementation, MIC is mainly a research tool. The design of the classification system and the GUI focus on maximal flexibility in the selection of classification algorithms and input feature selection methods. In further versions, the results of this thesis will be incorporated into MIC. The set of options in MIC can be restricted to those which guarantee best performance in different scenarios. This way, MIC will become easier to use without losing its classification power.

MIC has been designed within the KDE framework, which is one of the de facto standards for the desktop environments of Unix-like systems. With its object and agent oriented design, MIC is well prepared for the proliferation of component models in which components from different sources interact in one desktop environment. The flexibility of MIC, its ability to adapt to different environments, has been shown. MIC can become a useful component in intelligent document management environments for desktop systems, as well as in client/server applications.

Appendix

A. Classifying Email: A Practical Example

In chapter 6.3.3, the batch interface is used for email classification. We give a demonstration of the batch interface by showing how MIC is accessed in this scenario. In the email classification task, MIC is used as a client/server application. A MIC server is running permanently on a machine connected to the net. The server is trained on the classification task. Whenever a new email arrives, the client is invoked by the MDA (Mail Delivery Agent). The client passes the email to the server. The server responds with the classification of the email, and the client files the email in the appropriate category.

Server

The server is set into agent mode by the command line option `-socket`. When started with this option, it does not show the GUI, but listens to a TCP port. TCP sockets are means of communication between processes running on the same or different hosts. MIC chooses a random port for this. Since the client must know where the server is running, the name of the server and the port number is written to a file in the user's home directory. Via this socket, bidirectional communication with the MIC server is possible. All batch commands listed in chapter 7 are applicable. As long as the operation system takes no special measures to restrict access to the ports (e.g. by using a firewall), everybody can connect to MIC. When running in a non-trusted environment like the Internet, it would pose a serious security risk if everybody could issue commands to the MIC server. The client has to authenticate before it is allowed to issue commands to the server. The authentication mechanism is similar to the MIT-MAGIC-COOKIES used in X: Upon startup, the server generates a random string. This string is stored in configuration file `.mic_socket_setup` in the home directory of the user. The name of the machine and the port MIC is running on are also written in this file. In order to contact the server, the client must not only know the name and port of the server, but also send this authentication string. This way, only clients who have access to the contact information file, which resides in the user's home directory, can send commands to the server.

The classifier for classifying emails is trained offline and saved to a file. When started, the server has to load this file. This is done by a call to method `openDocument()` of class `MICDoc`. Let's assume the file is stored under the name `email_classifier.mic`. The following command is send to MIC:

```
MICDoc.openDocument "email\_classifier.mic"
```

Alternatively, the name of the MIC-file with the trained classifier can be passed via

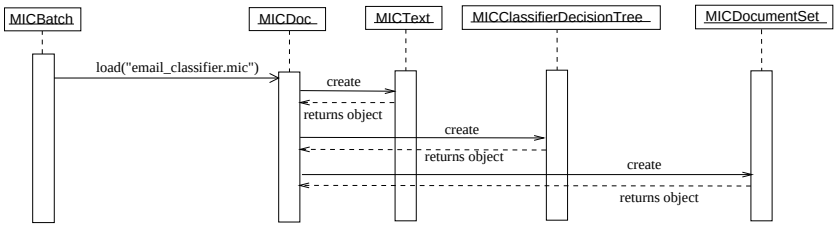

Figure A.1.: Sequence Diagram of processing a request to open a document

```
/home/gb/tmp/new_mail
/home/gb/tmp/mailablage0
/home/gb/tmp/mailablage1
/home/gb/tmp/mailablage2
/home/gb/tmp/mailablage3
/home/gb/tmp/mailablage4
/home/gb/tmp/mailablage5
/home/gb/tmp/mailablage6
/home/gb/tmp/mailablage7
```

Figure A.2.: Example `.mice_setup` file.

.

the command line. Figure A.1 shows how this command is processed by MIC. The first part of the command tells `MICBatch` that this is a command for `MICDoc`. `MICDoc` loads the file. After loading, it parses the file. Whenever it finds an XML tag indicating an object, it calls the constructor of the corresponding object. The XML sequence specifies the object. This way, all the objects of the system are created.

Since `MICDocuments` can contain more than one classifier the classifier must be named "mailclassifier", and the output vector component for mail classification must be named "mailfolder". The categories must be numbered ascending, starting with `0`.

Client

The client is named `mice` ("MIC Email classifier"). The source code for `mice` is shown in appendix B. Mice reads mail from the command line. It requests a classification of the mail from the server and writes it to the appropriate mail folder. In order to do this, it reads its configuration information from file `.mice_setup` in the user's home directory. The first line of `.mice_setup` gives the name for a temporary file. Both the client and the server must have access to this file. The following lines give the names and paths of the mailfolders. The second line gives the name and path where mails belonging in category 0 shall be stored. The third line gives the name and path where mails belonging in category 1 shall be stored and so on. An example for a configuration file is shown in figure A.2.

Mice is invoked by the MDA. Usually, `procmail` [28] is used for this. The entry in

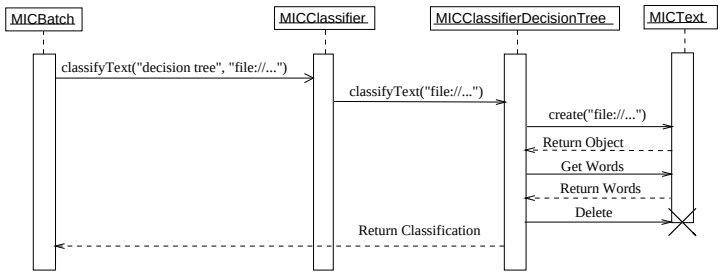

Figure A.3.: Sequence Diagram of processing a request to classify a text

the procamil configuration file `.procmailrc` is quite simple:

```
# :0 :
| mice
```

By this command, all incoming mail is piped to `mice`. MIC is ready for classification. Whenever an email arrives, it is passed to `mice`. `mice` stores the email in the temproray file given in `.mice_setup`. It asks classifier "mailclassifier" to classify this file in regard to output vector component "mailfolder": The following command issues the classification request to MIC:

```
MICClassifier.classifyFile "mailclassifier" "mailfolder"
                                    "file:///home/gb/tmp/new\_mail"
```

MIC returns the classification as an integer value from the domain of the output vector component. Figure A.3 shows how this command is processed by MIC. From the first part of the command name, `MICBatch` concludes that this is a batch command for class `MICClassifier`. The request is passed to `MICClassifier`. `MICClassifier` determines that this command is for the instance of `MICClassifierDecisionTree` called "mailclassifier". It passes the request to this class. This class creates a `MICText` object with the text that shall be classified. It requests the words in the text, and classifies the text based on the word frequencies. The `MICText` object is deleted, and the classification returned.

B. Source Code of mice

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <assert.h>
#include <netinet/in.h>
#include <unistd.h>
#include <errno.h>
#include <pwd.h>
#include <netdb.h>
#include <string.h>

#define STRINGLENGTH 128
#define MAX_CATEGORIES 50
#define FALSE 0
#define TRUE !FALSE
// This file store the setup of the socket interface when MIC is used
// as a socket. It contains the port number MIC is listening to, and
// is store din the user's home directory
#define SOCKET_SETUP_FILE ".mic_socket_setup"
#define MICE_SETUP_FILE ".mice_setup"

//
// Global variables
//
unsigned nrCategories = 0;
// Stores names of the mailfolders
char category[MAX_CATEGORIES][STRINGLENGTH];
// Mails are written to this file for classification
char tmp_file[STRINGLENGTH];

/*****************************************************************************
 *
 * Establish connection to MIC server. A connected socket is returned
 *
 *****************************************************************************
 */

int establishConnection(){
```

```
struct hostent *server_hostent;
struct passwd *pw;
char *home_dir;
unsigned port;
char hostname[STRINGLENGTH];
char cookie[STRINGLENGTH];
struct sockaddr_in server_addr;
int sockfd;
char port_string[STRINGLENGTH];
FILE *CONFIGF;

// Get IP & port of server
pw = getpwuid(getuid());
if(pw){
  home_dir = (char *) calloc((strlen(pw->pw_dir)
                              + strlen(SOCKET_SETUP_FILE) + 1),
                             sizeof(char));
  strcpy(home_dir, pw->pw_dir);
  *(home_dir + strlen(pw->pw_dir)) = '/';
  strcpy(home_dir + strlen(pw->pw_dir) + 1, SOCKET_SETUP_FILE);
}
else{
  fprintf(stderr, "Can't locate user home directory.\n");
  assert(FALSE);
}
CONFIGF = fopen(home_dir, "r");
if(!CONFIGF){
  fprintf(stderr, "Can't open %s.\n", home_dir);
  assert(FALSE);
}
// Read cookie (used for authentification).
fscanf(CONFIGF, "%s", cookie);
if(!hostname){
  fprintf(stderr, "Can't read cookie from %s.\n", home_dir);
  assert(FALSE);
}
// Read hostname
fscanf(CONFIGF, "%s", hostname);
if(!hostname){
  fprintf(stderr, "Can't read hostname from %s.\n", home_dir);
  assert(FALSE);
}
// Read port
fscanf(CONFIGF, "%s", port_string);
if(!port_string){
  fprintf(stderr, "Can't read port from %s.\n", home_dir);
  assert(FALSE);
}
port = atoi(port_string);
```

```
  server_hostent = gethostbyname(hostname);
  if(!server_hostent){
    fprintf(stderr, "Error getting hostname entry for host %s.\n",
            hostname);
    assert(FALSE);
  }

  if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
    fprintf(stderr, "Error getting socket.\n");
    assert(FALSE);
  }
  server_addr.sin_family = AF_INET;
  server_addr.sin_port = htons(port);
  server_addr.sin_addr = *((struct in_addr *) server_hostent->h_addr_list[0]);
  bzero(&(server_addr.sin_zero), 8);

  if(connect(sockfd, (struct sockaddr *) &server_addr,
             sizeof(struct sockaddr)) == -1){
    fprintf(stderr, "Error connecting socket.\n");
    assert(FALSE);
  }
  // Connect. We have to authenticate
  send(sockfd, cookie, strlen(cookie), 0);
  send(sockfd, "\n", strlen("\n"), 0);
  if(recv(sockfd, NULL, 0, 0) == - 1){
    fprintf(stderr, "Could not authenticate with server.\n");
    assert(FALSE);
  }

  return sockfd;
}

/*****************************************************************************
 *
 * Read configuration
 *
 *****************************************************************************
 */

void readConfiguration(){

  struct passwd *pw;
  char *home_dir;
  FILE *CONFIGF;

  pw = getpwuid(getuid());
  if(pw){
    home_dir = (char *) calloc((strlen(pw->pw_dir)
                                + strlen(SOCKET_SETUP_FILE) + 1),
```

```
                                sizeof(char));
    strcpy(home_dir, pw->pw_dir);
    *(home_dir + strlen(pw->pw_dir)) = '/';
    strcpy(home_dir + strlen(pw->pw_dir) + 1, MICE_SETUP_FILE);
  }
  else{
    fprintf(stderr, "Can't locate user home directory.\n");
    assert(FALSE);
  }
  CONFIGF = fopen(home_dir, "r");
  if(!CONFIGF){
    fprintf(stderr, "Can't open %s.\n", home_dir);
    assert(FALSE);
  }

  fscanf(CONFIGF, "%s", tmp_file);
  while((nrCategories < MAX_CATEGORIES) &&
        (fscanf(CONFIGF, "%s", category[nrCategories++]) != EOF));
  nrCategories--;
  fclose(CONFIGF);
}


/*************************************************************************
 *
 * Write mail from stdin to tmp_file
 *
 *************************************************************************
 */

void writeMailToTmpFile(){
  FILE *TMPF;
  int nextChar;
  TMPF = fopen(tmp_file, "w");
  if(!TMPF){
    fprintf(stderr, "Can't open %s for writing.\n", tmp_file);
    assert(FALSE);
  }
  while((nextChar = getc(stdin)) != EOF)
    if(fputc(nextChar, TMPF) == EOF){
      fprintf(stderr, "Error writing %s.\n", tmp_file);
      assert(FALSE);
    }
  fclose(TMPF);
}

/*************************************************************************
 *
 * Write mail to category
```

```
 *
 ************************************************************************
 */
void writeMail(int classification){

  FILE *IN, *OUT;
  int nextChar;

  IN = fopen(tmp_file, "r");
  if(!IN){
    fprintf(stderr, "Can't open %s for reading.\n", tmp_file);
    assert(FALSE);
  }
  OUT = fopen(category[classification], "a");
  if(!OUT){
    fprintf(stderr, "Can't open %s for writing.\n", category[classification]);
    assert(FALSE);
  }
  while((nextChar = fgetc(IN)) != EOF)
    if(fputc(nextChar, OUT) == EOF){
      fprintf(stderr, "Error writing %s.\n", tmp_file);
      assert(FALSE);
    }
  fclose(IN);
  fputc('\n', OUT);
  fclose(OUT);
}

/************************************************************************
 *
 * main
 *
 ************************************************************************
 */

int main(int argc, char *argv[]){

  int sockfd;
  char resultString[STRINGLENGTH];
  char tmpString[STRINGLENGTH];
  int tmp;
  int classification;
  int readResult;

  // Read configuration and write mail to temporary file
  readConfiguration();
  writeMailToTmpFile();
  // Ask MIC server to classify it
  sockfd = establishConnection();
```

```
  snprintf(tmpString, STRINGLENGTH,
           "MICClassifier.classifyFile \"mailclassifier\" \"mailfolder\" \"file:%s\"\n",
           tmp_file);
  send(sockfd, tmpString, strlen(tmpString), 0);
  tmp = 0;
  // Receive classification
  do{
    readResult = recv(sockfd, (void *) &resultString[tmp], 1, 0);
    if(readResult == -1){
      fprintf(stderr, "Error reading socket.\n");
      assert(FALSE);
    }
  }while(readResult && (tmp < STRINGLENGTH - 1)
         && (resultString[tmp++] != '\n'));
  resultString[tmp - 1] = '\0';
  classification = atoi(resultString);
  // Write mail
  writeMail(classification);
  // Delete temporary file
  unlink(tmp_file);

  return 0;
}
```

C. DTD

The following table shows the DTD for the XML-representation of `MICDoc`.

```
<!DOCTYPE mic [
   <!ELEMENT mic (outputVector+, outputVectorComponent+,
                  classification+, classifier+, text+, documentset+)>
   <!ATTLIST outputVector name CDATA #REQUIRED
             id ID #REQUIRED>
   <!ATTLIST outputVectorComponent name CDATA #REQUIRED
             id ID #REQUIRED
             outputVectorRef IDREF #REQUIRED
             lowerborder CDATA #REQUIRED
             upperborder CDATA #REQUIRED
             type CDATA #REQUIRED>
   <!ATTLIST classification textref IDREF  #REQUIRED
             outputVectorComponentRef IDREF #REQUIRED
             classifierref IDREF #REQUIRED
             value CDATA #REQUIRED>
   <!ATTLIST classifier name CDATA #REQUIRED
             id ID  #REQUIRED
             class CDATA  #REQUIRED>
   <!ATTLIST text url CDATA #REQUIRED
             id ID  #REQUIRED>
             documentsetref IDREF  #REQUIRED>
   <!ATTLIST documentset name CDATA #REQUIRED
             id ID  #REQUIRED>
]>
```

Bibliography

[1] Gerd Beuster, Bernd Thomas, and Christian Wolff. MIA — An Ubiquitous Multi-Agent Web Information System. In *Proceedings of International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce, MAMA*, 2000. http://www.uni-koblenz.de/~bthomas/WORK/PAPERS/mama2000/final.pdf.gz.

[2] Rainer Burkhardt. *UML — Unified Modeling Language: Objektorientierte Modellierung für die Praxis.* Addison-Wesley-Longman, Bonn, 1997.

[3] World Wide Web Consortium. HTML 4.01 Specification W3C Recommendation, December 1999. http://www.w3.org/TR/html4/.

[4] Katrin Erk and Lutz Priese. *Theoretische Informatik — Eine umfassende Einführung.* Springer, Berlin, Heidelberg, New York, 2000.

[5] Tim Finin, Jay Weber, Gio Wiederhold, et al. DRAFT: Specification of the Kqml. http://www.cs.umbc.edu/kqml/kqmlspec/spec.html.

[6] Internet Engineering Task Force. RFC 1341: MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies. http://www.ietf.org/rfc/rfc1341.txt.

[7] Nir Friedman. Bayesian Network Classifiers. *Machine Learning*, 29:131–163, 1997.

[8] Glimpse. http://glimpse.cs.arizona.edu/.

[9] Algorithmic Solutions Software GmbH. LEDA. http://www.algorithmic-solutions.com/as_html/products/products.html.

[10] Google. http://www.google.com.

[11] GNU General Public License. http://www.gnu.org/copyleft/gpl.html.

[12] Geoffrey E. Hinton and Terrence J. Sejnowski, editors. *Unsupervised Learning: Foundations of Neural Computation.* Computational Neuroscience Series. MIT Press, 1999.

[13] K. Hornik, M. Stinchombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[14] T. Joachims. Making large-Scale SVM Learning Practical. *Advances in Kernel Methods - Support Vector Learning*, MIT-Press, 1999.

[15] KDE. http://www.kde.org.

[16] KDevelop. http://www.kdevelop.org.

[17] KParts. http://developer.kde.org/documentation/tutorials/kparts/.

[18] Tom Christiansen Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., Bonn, Cambridge, Paris, Sebastopol, Tokyo, 1996.

[19] Daiv D. Lewis. Feature Selection and Feature Extraction for Text Categorization. In *Speech and Natural Language: Proceedings of a workshop held at Harriman, New York*, pages 212–217, San Mateo, CA, 1992. Morgan Kaufmann.

[20] Linux. http://www.linux.com.

[21] David J.C.: MacKay. Introduction to Information Theory, 1997. http://wol.ra.phy.cam.ac.uk/mackay/itprnn/1997/l1/node23.html.

[22] CALD. List of Classification Software. http://www.cs.cmu.edu/~cald/software.html.

[23] Andrew Kachites McCallum. Mini Version of 20 Newsgroups Dataset. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-11/www/naive-bayes/mini_newsgroups.tar.gz.

[24] Andrew Kachites McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. http://www.cs.cmu.edu/~mccallum/bow, 1996.

[25] Tom M. Mitchell. *Machine Learning*, chapter Bayesian Learning. The McGraw-Hill Companies, Inc., New York, 1997.

[26] Neunzert, Eschmann, Blickensdörfer-Ehlers, and Schelkes. *Analysis 2.* Springer, Heidelberg, New York, London, Paris, 1991.

[27] David Poole, Alan Mackworth, and Rand Goebel. *Computational Intelligence — a logical approach.* Oxford University Press, New York, Oxford, 1998.

[28] Procmail. http://www.procmail.org.

[29] QT. http://www.troll.no/qt/.

[30] Marco Ramoni and Paola Sebastiani. RoC: The Robust Bayesian Classifier. http://kmi.open.ac.uk/projects/bkd/.

[31] SER. SERpersonalbrain. http://www.serware.de/de/serware/download/download-index.html.

[32] SGI. MLC++. http://www.sgi.com/Technology/mlc/.

[33] Murray Smith. *Neural Networks for Statistical Modeling.* Thompson Computer Press, 1993.

[34] SPSS. CLEMENTINE. http://www.spss.com/clementine/.

[35] Bernd Thomas. Bernd Thomas' Homepage. http://www.berndthomas.de.

[36] Bernd Thomas. Intelligent Web Querying With Logic Programs. In *Proceedings of the Workshop on Inference Mechanisms in Knowledge-based Systems, preceeding the national german AI conference KI '98*, Bremen, 1998. http://www.uni-koblenz.de/home/bthomas/WORK/PAPERS/ki-ws98/paper.pdf.gz.

[37] Bernd Thomas. Anti-Unification Based Learning of T-Wrappers for Information Extraction. In *Workshop on Machine Learning for Information Extraction, preceeding Sixteenth National American Conference on Artifical Intelligence (AAAI-99)*, 1999. http://www.uni-koblenz.de/home/bthomas/WORK/PAPERS/aaai99/paper.pdf.gz.

[38] Bernd Thomas. MIA — An Ubiquitous Multi-Agent Web Information System, Presentation given at International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce, MAMA. http://www.uni-koblenz.de/home/bthomas/WORK/TALK/Mama2000/mama2000/index.htm, 2000.

[39] Bernd Thomas. Token-Templates and Logic Programs for Intelligent Web Search. *Journal of Intelligent Information Systems*, 14:241–261, 2000. http://www.uni-koblenz.de/home/bthomas/WORK/PAPERS/jiis/jiis_paper.pdf.gz.

[40] Gottfried Vossen. *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme.* Addison-Wesley, Bonn, Paris, 1994.

[41] XML. http://www.xml.org.

[42] Y. Yang. An evaluation of statistical approaches to text categorization, 1999.

Index