E-Sl@ve

an incremental approach to automated, content-based email classification

Masters Thesis: 505

by:

Christiaan Rudolfs
crudolfs@sci.kun.nl

Katholieke Universiteit Nijmegen (KUN)
dept. Computing Science

Supervisors:
Prof. C.H.A. Koster (KUN)
Dr. Paul Jones (Edmond R&D)

2nd August 2002

Contents

Chapter 1

Introduction

On-line communication, in particular email communication, has an explosive growth. Especially large companies are often flooded with thousands of electronic messages a month. Management of these emails is very important, as this enables a faster processing and easier retrieval of (old) messages.

A way of managing emails is to classify every email into a pre-defined category (mailfolder), which applies to that email. Performing this classification task by hand often is a difficult and time-consuming (thus expensive) job. Therefore an email classification system, which *automatically* classifies incoming emails, becomes very useful.

An email classification system has to be far more powerful than the "filter" functions provided by email packages that file incoming messages according to the sender's name, or a word in the subject line, as these often suggest nothing about the message's content. To effectively categorize email, the system would have to analyze the full text of every message. Furthermore the system must be able to quickly adapt to changes in its *dynamic* email environment, and the user should not endure additional burdens using the system.

1.1 Previous work

Much research has recently been vested in theoretical concerns surrounding the problem of *text classification* (also known as *text categorization*). Many publications are describing issues dealing with this problem. On the other hand, the problem of *automatic email classification* is rather new to scientific research, as systems for categorizing emails into different classes are just now becoming available. Most publications describing *automatic email classification* are dealing with **theoretical** concerns (e.g. comparing classification accuracy for different learning algorithms) surrounding the applicability of *text classification* to the problem of *automatic email classification*. Only a few publications deal with **practical** concerns for *automatic email classification* (e.g. the problem of user-interaction in a real-life situation).

In [6] methods for learning text classifiers are compared, focusing on the kinds of classification problems that might arise in the filtering and filing of personal email messages. An extended version of the *rule-based* learning algorithm *RIPPER* is compared with the traditional IR learning algorithm *Rocchio* ([19]). The extended *RIPPER* algorithm seemed to perform best on various email corpora, although *Rocchio* performed very well also.

In [18] three experiments are presented, comparing a Naïve Bayesian algorithm with bag-valued features against the *RIPPER* rule learning algorithm ([5]) in different email classification tasks. In learning a user's foldering preferences, and learning to detect spam, the Bayesian classifier substantially outperformed *RIPPER* in classification accuracy. In reconstructing the policy of an automated, rule-based email classifier, both systems performed very well, but the Bayesian classifier still showed a small but statistically significant improvement over *RIPPER*.

In [11] it was empirically proved that *co-training* [4] can be applied to email classification. At the same time it was shown that the performance of co-training depends on the learning method it uses. Namely, Naïve Bayes performed very poorly in the experiments while Support Vector Machines ([10]) worked very well. Though, more research is needed to clarify the causes of the poor behaviour of Naïve Bayes in combination with co-training and explore other possibilities (along with feature selection) to improve the performance of Naïve Bayes in the co-training loop.

In [9] a customizable email classification system, Ishmail, has been described that addresses the problem of information overload. Ishmail is unique in that it not only sorts messages into mailboxes, but it orders mailboxes by a combination of user-specified priorities and alarms. In this article, Ishmail's design is diagramed in terms of its functional components and their interactions.

In [22] results are demonstrated of Swiftfile, an email assistant that helps users organize their (personal) email into folders. Using a text classifier that *dynamically* adjusts to the user's mail-filing habits, Swiftfile predicts for each incoming message the three folders that it deems most likely to be choosen by the user as destinations. Swiftfile uses a modified version of AIM [2] for classifying text. AIM is a TF-IDF style text classifier developed at IBM Almaden. Results of their experiments showed that *incremental learning* (2.3) with this AIM classifier performs very well in classifying emails.

1.2 Problem statement

This thesis discusses both theoretical and practical concerns surrounding the applicability of *text classification* to the problem of *email classification.* Using a prototype, called *E-Sl@ve* (3.2), the following issues are explored:

Theoretical issues:

- How applicable is *incremental learning* (2.3) for the *Balanced Winnow* algorithm (2.2), to automatic, content-based *email classification*, and what are the differences with respect to *batched learning.*
- Is it possible to extend or modify the Balanced Winnow algorithm, such that an increased classification accuracy with *incremental learning* is achieved.

Practical issues:

- How to modify E-Sl@ve such that, in a real-life situation, the user only needs little effort in keeping the system accurate.

1.3 Overview

In chapter 2, the domain of *automatic document classification* is introduced, the working of Balanced Winnow is described, and the differences between *batched learning* and *incremental learning* are explained. In chapter 3, the prototype E-Sl@ve is introduced, which provides the core functionality for an email classification system. Using results of experiments, the accuracy of E-Sl@ve in classifying emails is explored. Next, in chapter 4, some (possible) optimalisations to E-Sl@ve are introduced for achieving higher classification accuracy. In chapter 5, *negative relevance feedback* is introduced, which is described as a scenario that enables users in a real-life situation to keep the system accurate with only little effort. Finally, in chapter 6, several issues for future research are mentioned.

Chapter 2

Automatic Document Classification

In document classification, given a text document (e.g. an email) and a collection of potential classes, an algorithm decides which classes the document belongs to, or how strongly it belongs to each class. More formally it can be described as follows (see [12]):

> Given a set of classes (topics) C and examples for each class, construct a *classifier* for each class which, given a document d, computes the *relevance* of document d for the class.

A *classifier* for a class c is thus a function which expresses the *relevance* of documents for class c.

Classifiers have to be *learned*. Many different *learning algorithms* for text classifiers exist, all of them using different techniques in learning the classifiers. In this thesis the *learning algorithm* Balanced Winnow (2.2) is used. Balanced Winnow belongs to the class of *supervised learning* algorithms.

2.1 Supervised learning algorithms

Learning algorithms for document classification (also known as text classification/categorization) bring together techniques from IR (Information Retrieval) and AI (Artificial Intelligence). For an overview of the literature in this field see [21]. In this thesis only one *supervised learning* algorithm (Winnow) is explored.

Supervised learning algorithms use (labeled) **training data** to learn classifiers which classify new texts. Documents in a *corpus*, which consists of a set of "typical" *pre-classified* example documents for each class, form this training data. Each document in this corpus is labeled by one or more classes. A document is considered as a *positive example* for all classes with which it is labeled and as a *negative example* for all classes with which it is not labeled.

Three broad classes of *supervised learning* algorithms can be distinguished:

1. **linear classifiers**
   Learning algorithms for *linear* classifiers classify new documents according to the *score* for each class that is obtained by taking an inproduct of *class profile* (weighted vector of keywords) and *document profile* (2.1.1). Good examples are Rocchio ([19]) and (Balanced) Winnow ([14, 8]).

2. **rule-based classifiers**
   Learning algorithms for *rule-based* classifiers learn by inferring a set of rules from pre-classified documents. A good example is the *Ripper* algorithm ([5, 7]).

3. **example-based classifiers**
   Learning algorithms for *example-based* classifiers classify a new document by finding the k nearest to it in the train set and doing some form of majority voting on the classes of these nearest neighbours (see [10]).

In this thesis Balanced Winnow (2.2) is used. This algorithm trains linear classifiers.

2.1.1 Training linear classifiers

Text classifiers represent a document d by a set of *features*:
$F(d) = \{f_1, f_2, \cdots, f_m\}$, where m is the number of **unique** features in the document. In this thesis a *feature* is represented by a single word. Every feature f has a *strength* in any document d, denoted by $s_d(f)$. Several ways to compute this strength are found in the *Information Retrieval* literature:

- boolean strength: $s_d(f) = 1$ *or* 0, indicating respectively the presence or absence of feature f in d.
- frequency strength: $s_d(f) = n(f, d)$, reflecting the number of times f appears in d.
- square root strength: $s_d(f) = \sqrt{n(f, d)}$, reflecting the square root of the number of times f appears in d.

In E-Sl@ve (3.2), *square root* term strengths are used, because in [8] it was explored that using *square root* term strengths resulted in the best accuracy compared to the other methods of computing term strengths.

A *linear* (text) classifier represents a *document profile* for a document d by a vector of its feature strengths: $\overline{s}_d = (s_d(f_1), s_d(f_2), \cdots, s_d(f_m))$.
A category is represented by a *weighted vector of keywords* (also called *class profile*): $\overline{w}_c = (w_c(f_1), w_c(f_2), \cdots, w_c(f_n))$, where n is the number of features in the domain and $w_c(f_i)$ is the weight of the i-th feature for class c.

The *score* of document d for class c, denoted as $S_c(d)$, is evaluated by computing the dot product of weight vector $\overline{w}_c$ and feature strength vector $\overline{s}_d$:

$$S_c(d) = \sum_{f_i \in F(d)} s_d(f_i) \cdot w_c(f_i)$$

The algorithm classifies a document according to the scores it achieves for all classes. When the score for a class is above a certain *threshold*, then the document is classified as *relevant* for that class. This makes it possible for a document to be classified in more than one class, which is called *multi-classification.* In *mono-classification* the document is assigned to exactly one class.

The task of a learning algorithm for *linear* text classifiers is to find weight vectors (class profiles) which best classify new documents. In the next section, it is explained how *Balanced Winnow* performs this task.

2.2 Balanced Winnow

Balanced Winnow ([8]) is a variant of Littlestone's Winnow algorithm ([14]). Winnow (like Support Vector Machines [10]) classifies documents by learning linear separators (*classifiers*) (2.1.1) in the feature space. Winnow is an *on-line* and *mistake-driven* learning algorithm. It is *on-line* in the sence that a classifier X_c for class c first predicts the *relevance* of a document for class c and then recieves feedback, called *relevance feedback*, on this prediction, which may be used to update the current *hypothesis* (vector of weights) of the classifier. Because this *hypothesis* is only updated when the algorithm has made a wrong prediction (and thus made a mistake), Balanced Winnow is called *mistake-driven.* The current vector of weights represents the *current state* of the classifier.

To *learn* classifiers (which may be interpreted as finding good weight vectors), usually a set of pre-classified documents from a corpus is used as the training data. This is called the *train set.* In a train set each document is labeled by one (*mono-classification*) or more (*multi-classification*) classes. A document is considered as a *positive example* for all classes with which it is labeled and as a *negative example* for all classes with which it is not labeled. The labeling of the documents is used to provide "perfect" *relevance feedback.*

Balanced Winnow has three parameters: a *threshold* θ, and two *update* parameters, a *promotion* parameter α and a *demotion* parameter β. They are choosen as follows:

$$\begin{aligned}&\theta = 1\\&\alpha > 1\\&0 < \beta < 1\end{aligned}$$

The algorithm maintains **two** weights for every feature: w^+ and w^-. The overall weight of a feature is the difference between these two weights, thus allowing for *negative* weights. We have seen (2.1.1) that a document d is denoted as a vector of its feature strengths: $\overline{s}_d = (s_d(f_1), s_d(f_2), \cdots, s_d(f_m))$, where m is the number of unique features in document d and $s_d(f_m)$ is the strength of the m-th feature in d. Now, given a document d, a classifier X_c for class c predicts that this document is *relevant* for that class if:

$$S_c(d) = \sum_{j=1}^{m} (w_c^+(f_j) - w_c^-(f_j)) \cdot s_d(f_j) > \theta$$

in which $w_c(f_j)$ is the weight of the j-th feature in document d for class c. The initialisation of the weights will be discussed later (3.2.3). For now it is important to know that w^+ has an initial value that is 2 times the value of w^-. In case a classifier X_c makes a **wrong** prediction, weight vector $\overline{w}_c$ will be updated. Only the weights of features in $\overline{w}_c$ that **also** occur in the document (the *active* features) are updated. This happens according to the following *update rules*:

1. **Positive example**
   If $S_c(d) < \theta \wedge d \in c$, then for all **active** features, w_c^+ is *promoted* by multiplying it with α and w_c^- is *demoted* by multiplying it with β. This results in an *increasing* overall weight $(w^+ - w^-)$ for all active features, which *promotes* the *positive* example d.

2. **Negative example**
   If $S_c(d) > \theta \wedge d \notin c$, then for all **active** features, w_c^+ is *demoted* by multiplying it with β and w_c^- is *promoted* by multiplying it with α. This results in a *decreasing* overall weight for all active features, which *demotes* the *negative* example.

This promoting and demoting of weights ensures that the classifiers learn from their *mistakes*.

2.2.1 Threshold range

An extension to this algorithm is the *thick-threshold* heuristic (see [8]). In this case the scores for *positive* and *negative* examples are separated as **widely** as possible. The idea is to introduce two separate thresholds: θ^+ and θ^-, such that $\theta^+ > \theta^-$. Now a classifier X_c for class c predicts that a document d is relevant for class c if $S_c(d) > \theta^+$. A document is predicted to be irrelevant if $S_c(d) < \theta^-$. All scores within the range $[\theta^-, \theta^+]$ are considered *mistakes.*

When this heuristic is used, a *positive* example ($d \in c$) is *promoted* when $S_c(d) < \theta^+$ and a *negative* example ($d \notin c$) is *demoted* if $S_c(d) > \theta^-$. In this way the scores for all *positive* examples are widely separated from the scores for *negative* examples. E-Sl@ve uses this heuristic.

2.3 Batched vs. incremental learning

Now that it has been explained how Balanced Winnow works, there remains an important issue unmentioned. This issue concerns the overall *classification process.* This classification process defines the way how classifiers are build, the moments when classifiers are trained and the moments when these classifiers are used to classify **new** (unseen) documents. We consider two different approaches for this process: *batched learning* (as in LCS [1] ([3]) and *incremental learning* (as in *E-Sl@ve* (3.2)). Note that, while explaining these approaches, it is assumed that the learning algorithm Balanced Winnow (2.2) is used.

2.3.1 Batched learning

The *batched learning* approach distinguishes a *training phase* and a *production phase.* The training phase is used to train the classifiers, while the production phase is used to apply the trained classifiers to classify **new** (unseen) documents. The classification process for this approach consists of the following steps:

1. Collect statistics on the train set.
2. Create initial class profiles.
3. Train *iteratively* on all documents of the train set (training phase).
4. Classify new documents (production phase).

[1]Linguistical Classification System, developed at the Katholieke Universiteit of Nijmegen (KUN)

In this approach, training can be done *iteratively* on all the documents in the train set. As Balanced Winnow is sensitive to the ordering of training documents, after each iteration the documents in the train set are shuffled randomly. Iterating can be done for a fixed number of times or until the classifiers do not make any *mistakes* (2.2) on the train set anymore. When the training phase is finished, the classifiers have reached their *final state*. This implies that, during the production phase, the classifiers do not change, which means that **new** documents are **not** used for training.

Each step in this process can not be performed until its previous step has been performed. Therefore this (*batched learning*) approach has some **restrictions** in our situation (that deals with *email classification* in a **dynamic** environment). These restrictions are:

- the required presence of a train set, which must be preserved.
- the required training of a *batch* of documents before the production phase can be started.
- classifiers do not learn (immediately) from new documents.
- classifiers have to be learned from scratch, when additional training with new documents is desired, adding new plus trained documents.

2.3.2 Incremental learning

The global process for *incremental learning* consists of two simple steps:

1. Collect class names.
2. Classify new documents (production phase).

What we see is that, in contrast with *batched learning*, *incremental learning* does not require the presence and preservation of a train set. Initial training is not required either. The only information that must be available from start, are the names of potential classes in which new documents can be classified. The algorithm can therefore directly start classifying new (unseen) documents in the production phase.

In an *incremental learning* situation, classifiers are trained *incrementally* during the production phase, never reaching a *final state*. This means that every *new* **and** *relevant* document that arrives will be used *immediately* (after obtaining *relevance feedback* (5.1)) to update existing classifiers from their current state. In this way, classifiers are trained **one** single document at a time, after which they are ready to classify new documents again. This is a big difference with *batched learning*, where a whole *batch* of documents is trained iteratively from scratch, before the production phase can even be started. Note that training iteratively on a train set is not possible in the *incremental learning* situation (as the train set is not preserved), which makes another difference with *batched learning.*

In a real-life situation, several ways to obtain *relevance feedback* (which is the information about a document that indicates the class for which that document is most relevant, according to an "expert"), are imaginable. In chapter 5 this issue is explored. For now it is only important to notice that recieving feedback on the relevance of a new document is necessary, as *non-relevant* documents do not belong to any of the potential classes and are therefore useless training examples.

In the domain of email classification, *incremental learning* is preferable. The advantages are:

- the production phase can be started right away (no pre-classified documents nor initial training are required).
- classifiers learn immediately from **new** documents, which enables them to adapt to slight changes in the "meaning" of topics (classes) over time (which is useful within the email domain).
- classifiers are trained **one** single document at a time (no batch), which makes periods of training very short and ensures that classifiers are ready for classifying new documents immediately.

Chapter 3

Incremental learning

The email environment is very *dynamic*. Contents of new messages and the user's mail filing habits constantly change. For email classifiers it is important to adapt to these changes, preferably as soon as possible. Some email classification systems adapt to these changes by retraining from scratch on a daily basis (mostly over night) (see [15], [16]). A potential disadvantage of this *batched learning* (2.3) is that the system may not be sufficiently responsive to the above mentioned changes. Therefore, a better way of adapting to changes would be to update existing classifiers from their current state *immediately* after a certain event occurs (which can be the arrival of a new email, the movement of an email from one folder to another, and more..). This is what is called *incremental learning*. In paragraph 2.3 *incremental learning* has been described more thoroughly.

3.1 Different situations

In [22] was demonstrated, that in a **dynamic** email environment *incremental learning* indeed performs better than *periodic (batched) learning*. However, an important detail is that a different algorithm was used. This algorithm can incrementally update classifiers with a single new document, obtaining the same state of the classifiers as it would have obtained by retraining the classifiers from scratch including the new document. The consequence of this is, that in a **static** environment the results of *incremental learning* equal the results of *batched learning*. This does **not** hold for the Balanced Winnow algorithm.

In a **static** environment, Balanced Winnow should perform better in a *batched learning* situation than in an *incremental learning* situation. This assumption can be made for two reasons:

1. in a *batched learning* situation classifiers have statistics on the whole

train set, while in an *incremental learning* situation classifiers only have statistics upto the current state of the system, starting with no statistics at all.

2. in a *batched learning* situation classifiers train *iteratively* on all documents in a train set, while in an *incremental learning* situation classifiers are trained with **one** single document not having the possibility of training *iteratively* on all documents in a train set.

Especially the second point is assumed to have great impact on the results, because training *iteratively* on all documents from a train set ensures that classifiers get to know their class members (and non class members) better. Therefore, in this chapter the performance of *incremental learning* compared to *batched learning* for the Balanced Winnow algorithm is explored.

3.2 E-Sl@ve

For this thesis, a system called E-Sl@ve was developed, which provides the core functionality for an email classification system. E-Sl@ve is coded in the *Java* programming language. The tools, runtimes and APIs that are used, were all provided by the *Java 2 Platform, Standard Edition* [1]. E-Sl@ve learns, according to the *incremental learning* approach (2.3) applied to the Balanced Winnow algorithm (using the *thick-threshold* heuristic (2.2.1)). The system is (currently) only suitable for *mono-classification*, where every document is assigned to exactly one class. No linguistical techniques (e.g. stemming), stoplists or other pre-processing "instruments" are used.

3.2.1 Feature extraction

A text classifier represents a document by its features strengths. E-Sl@ve represents features as single words, and extracts them from a document according to the following criteria:

- a feature should begin with a letter.
- a feature should have a minimum length of two characters.

Email addresses are cut into pieces. For example the email address *christiaan@edmond.nl* is cut into three features (christiaan, edmond, nl). This prevents that, when a person has multiple email addresses within the same domain, these addresses are identified as two different features. For example the same person could also have the email address *christiaan.rudolfs@ed-*

[1]http://java.sun.com/j2se/

mond.nl. When these email addresses are not cut into pieces, these two example email addresses would be identified as two different features, while it belongs to the same person. Whether cutting email addresses into pieces influences the classification accuracy in a positive or negative manner, if it influences the accuracy at all, is not known.

3.2.2 Internal process

E-Sl@ve follows the *incremental learning* approach. In 2.3, the process for this approach has been described already, but this time more details are given.

1. Collect class names.
2. Classify new emails (production phase).
   (a) Classify one new email according to the current state of the classifiers.
   (b) Obtain *relevance feedback* on the email.
   (c) Extend all class profiles (weight vectors) with the terms that occur in the email.
   (d) Incrementally train all classifiers with the single email (Balanced Winnow (2.2)), using it as a *positive* example for the class for which it is relevant (according to the recieved feedback in (b)) and as a *negative* example for all other classes.

Formal description

ad 1):
Obtain potential classes: $\{c_1, c_2, \cdots, c_z\}$, where z is the number of classes.

ad 2(a)):
For each class c_i, there exists a classifier X_{c_i} with weight vector $\overline{w}_{c_i}$. This weight vector is initially empty. A new email e arrives, and will be classified according to the current weight vector $\overline{w}_{c_i}$ for all classifiers X_{c_i}.

ad 2(b)):
Get the class c_x for which email e is relevant (if there is any), according to the label of e (in an experimental environment) or according to the feedback from an "expert" user (real-life situation, see also chapter (5)).

ad 2(c)):
Extend each weight vector $\overline{w}_{c_i}$ with all the features (initial weighted) that occur in e. In this way, weight vector $\overline{w}_{c_i}$ will contain many negative features, which are the features that do not occur in any of the examples for class c_i.

ad 2(d)):
When classifier X_{c_x} makes a mistake according to the *thick-threshold* heuristic (2.2.1), the *active features* (2.2) of weight vector $\overline{w}_{c_x}$ are *promoted.* For all other classifiers X_{c_i} for which $i \neq x$ it holds that, when they make a mistake, the *active features* in weight vector $\overline{w}_{c_i}$ are *demoted.*

Note that step 2(a) to 2(d) are repeated, every time a new message arrives.

3.2.3 Weight initialisation

An important issue is the initialisation of the *Winnow weights* w^+ and w^-. As we have seen already (2.2), these weights are used to compute the *score* of a document d for class c:

$$S_c(d) = \sum_{j=1}^{m} (w_c^+(f_j) - w_c^-(f_j)) \cdot s_d(f_j)$$

When the weights are initialised significantly too low or too high, more documents have to be trained to achieve a certain level of accuracy. The *ideal* initialisation of these weights, in the absence of any knowledge of the correct classes of the documents, should have the property that it assigns to an *average* document for every class the score θ (which is 1).

In our situation (*incremental learning*), it is unknown what an *average* document is, because there are no statistics on the potential classes available. For this reason, the choice was made to modify $s_d(f)$, the strength of feature f in document d, by using a quantity that is normalized with respect to the document length. Formally, the strength $s_d(f)$ is replaced by a *normalized strength*:

$$snorm_d(f) = \frac{s_d(f)}{\sum_{i \in F(d)} s_d(i)}$$

in which $snorm_d(f)$ is the *normalized strength* of feature f in document d, and the other symbols are defined as in 2.1.1.

This modification makes it possible to initialise w^+ to 2θ and w^- to θ. The explanation for this is as follows:

The *average document strength* d_{avg} can be defined as:

$$d_{avg} = \sum_{f \in F(d)} snorm_d(f) = \sum_{f \in F(d)} \frac{s_d(f)}{\sum_{i \in F(d)} s_d(i)} = \frac{\sum_{f \in F(d)} s_d(f)}{\sum_{i \in F(d)} s_d(i)} = 1$$

Consequently this leads to $S_c(d) = \theta$ (=1), because for every $f \in F(d)$ the coefficiënt of the Winnow weights is 1, as $(w^+ - w^-) = (2\theta - \theta) = \theta = 1$. As this is the score for a document that we wanted, it is justified to initialise w^+ to 2θ and w^- to θ.

3.2.4 Determining the relevant class

The algorithm classifies a document according to the scores it achieves for all classes. When the score for a class is above a certain *threshold*, then the document is classified as *relevant* for that class. Therefore it is possible that a document will be classified in more than one class (*multi-classification*). Because this thesis deals with *mono-classification*, a new document should be classified in exactly **one** class. To determine this class, the score $S_c(d)$ of document d is computed for every class c, according to the current state (weight vector) of the classifiers, and then d is assigned to the class for which d obtained the **highest** score.

In 4.3 a different method for determining the relevant class is explored.

3.3 E-Sl@ve vs. LCS

In this initial test the results of E-Sl@ve are compared with the results of the Linguistical Classification System (LCS ([3])). The purpose of this test is to compare *incremental learning* (as in E-Sl@ve) with *batched learning* (as in LCS) for the Balanced Winnow algorithm (2.2).

First the corpora used in the experiments are described. Then some measures are defined to determine the success of classification. Finally the setup and results of experiments are described.

3.3.1 Corpora

In this thesis two corpora are used to perform experiments. One corpus (*Reuters mono subset*) is no email corpus, but consists of short newspaper articles, which have a good likeness with email messages. Because it is known that the documents in this corpus are very well pre-classified, this corpus is very useful for running experiments. The other corpus (*Edmond*) **is** an email corpus. This corpus has been created especially for this thesis, which means that no experiences of running experiments on this corpus exist.

Reuters mono *subset* corpus

The *Reuters mono subset* corpus consists of a random selection of 9090 pre-classified documents from the well-known Apte subset of the Reuters 21578 corpus [1]. The documents are short (mono-classified) newspaper articles very unevenly distributed over 66 classes. Because we are not interested in classifying a huge number of documents, a subset of this corpus was created. Important criteria for the subset are: a reasonable number of classes should be taken, the uneven distribution of documents over the classes has to remain intact, and all classes should contain at least 10 example documents. The corpus is called the *Reuters mono subset* corpus. In table 3.1 the statistics for the *Reuters mono subset* corpus are depicted.

Number of documents	3065
Number of classes	15
Total number of words in corpus	404825
Number of unique words in corpus	16169
Average number of words per document	132
Average number of unique words per document	75
Smallest number of documents in a class	12
Largest number of documents in a class	701

Table 3.1: Statistics for the *Reuters mono subset* corpus.

Edmond corpus

The *Edmond* corpus consists of real emails from two running projects within the company Edmond R&D. The class structure and the classification of the emails were manually constructed. The two projects are merged into a single corpus to get a larger document set. In table 3.2 the statistics for this corpus are depicted.

Number of documents	1134
Number of classes	18
Total number of words in corpus	264552
Number of unique words in corpus	15327
Average number of words per document	233
Average number of unique words per document	121
Smallest number of documents in a class	8
Largest number of documents in a class	130

Table 3.2: Statistics for the *Edmond* corpus.

3.3.2 Measures: Precision, Recall, Accuracy

In determining the success of classification, the measures *Precision*, *Recall* and *Accuracy* are used throughout this thesis. These measures are based on several quantities that must be tracked for **every** class during the classification process. The quantities are:

- RS = *R*elevant *S*elected, the number of **relevant** documents, classified as **relevant**.
- RNS = *R*elevant *N*ot *S*elected, the number of **relevant** documents, classified as **irrelevant**.
- NRS = *N*ot *R*elevant *S*elected, the number of **irrelevant** documents, classified as **relevant**.
- $NRNS$ = *N*ot *R*elevant *N*ot *S*elected, the number of **irrelevant** documents, classified as **irrelevant**.

Now we can define *Precision* and *Recall* as follows:

$$Precision = \frac{RS}{RS + NRS}$$

$$Recall = \frac{RS}{RS + RNS}$$

To obtain a single measure for the whole corpus, the average *Precision* or *Recall* is used. Two ways of averaging can be distinguished:

1. **Micro average**
   The *Precision* (and *Recall*) is calculated by summing the quantities over all classes. This average is dominated by the **large** classes (those with many training documents).
2. **Macro average**
   The *Precision* (and *Recall*) is calculated by summing the Precision (and Recall) for every class and then dividing it by the number of classes. This average is dominated by the **small** classes.

In *mono-classification*, **micro-averaged** *Precision* equals **micro-averaged** *Recall*. This is explained as follows. When the algorithm classifies a document d in a class c_i, for which it is not relevant (the algorithm has made a mistake), then NRS_{c_i} is increased by one. At the same time, say that d had to be classified in class c_j, RNS_{c_j} is increased by one. Consequently, the values of NRS and RNS in the above definitions of *Precision* and *Recall* are equal (as the quantities of all classes are summed). The **macro-averaged** *Precision* and *Recall* usually are not equal.

A good measure to indicate the *Accuracy* of the algorithm is the *F1*-measure. This measure is defined such that *Precision* and *Recall* are assigned equal importance:

$$F1 = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

Note that this implies for **micro-averaging** that:

$$F1_{micro} = Precision_{micro} = Recall_{micro}$$

In the rest of this thesis $Accuracy_{micro}$ ($= F1_{micro}$), and $Accuracy_{macro}$ ($= F1_{macro}$) are used to determine the accuracy of classification results.

3.3.3 Experiment: Learning behaviour

Global setup

Several tests on both the *Reuters mono subset* corpus and *Edmond* corpus were performed. For this purpose these corpora were split into a train set and a test set. The pre-classifications of the documents in the corpus is used to provide the "perfect" *relevance feedback*. Documents in the train set are used to train classifiers, while documents in the test set are used to determine the accuracy of the trained classifiers. For both corpora a train set was choosen such that it consists of 75% of the documents in the corpus. The test set consists of the remaining 25% of the documents in the corpus. Tests on several different train sets and test sets were performed to get more reliable results.

Both corpora were partitioned in four parts: $p1$, $p2$, $p3$ and $p4$, each time taking one part as the test set and the other three parts as the train set. In this way, four separate tests for each corpus are acquired:

1. train set = $\{p1, p2, p3\}$, test set = $\{p4\}$
2. train set = $\{p1, p2, p4\}$, test set = $\{p3\}$
3. train set = $\{p1, p3, p4\}$, test set = $\{p2\}$
4. train set = $\{p2, p3, p4\}$, test set = $\{p1\}$

In each test the algorithm trains on increasing parts of the train set (called *epochs*), so that the *learning behaviour* of both systems can be compared. The results of all four tests were averaged to determine the final $Accuracy_{micro}$ and $Accuracy_{macro}$ (3.3.2).

E-Sl@ve specific setup

Because Balanced Winnow is sensitive to the ordering of training documents, and E-Sl@ve (*incremental learning*) does not train iteratively on a batch of documents, all four tests were performed 10 times, each time using a randomly shuffled version of the train set. For all four tests the results were averaged.

The settings that were used for E-Sl@ve during this experiment are depicted in table 3.3.

term strengths	:	*sqrt*
α	:	1.1
β	:	0.9
θ^+	:	1.1
θ^-	:	0.9

Table 3.3: Test settings for E-Sl@ve.

LCS specific setup

In contrast with E-Sl@ve, LCS does not need different shuffled versions of each train set, because the system iteratively trains on all documents out of the train set, internally shuffling the train set after each iteration. Because the results of two runs of the same test may vary (caused by the sensitivity to the ordering of training documents), each test was performed 10 times and the results were averaged.

The settings that were used for LCS during this experiment are depicted in table 3.4.

term strengths	:	*sqrt*
normalize	:	*linear*
term selection	:	off
α	:	1.1
β	:	0.9
θ^+	:	1.1
θ^-	:	0.9
maxiters	:	5

Table 3.4: Test settings for LCS.

Main differences

Note that the main differences between E-Sl@ve and LCS are:

- LCS performs (max.) 5 iterations on the train set (*batched learning*), wile E-Sl@ve performs no iterations at all (*incremental learning*). A situation in which E-Sl@ve performs multiple iterations with **one** document (*Aggressive-Training*) is explored later in this thesis in 4.2.

- LCS collects statistics on the whole train set, while E-Sl@ve only has statistics upto the number of documents that have been processed currently, starting with no statistics at all. Therefore, LCS starts with class profiles that are initially filled with all (initially weighted) features from documents in the corpus (assuming that no term selection is used), while E-Sl@ve starts with class profiles that are initially empty, extending them with features while documents are processed.

3.3.4 Results: Learning behaviour

Reuters mono subset **corpus**

The $Accuracy_{micro}$ (3.3.2) is depicted for both systems in figure 3.1. In a real-life situation, this measure is most indicative, as it denotes the number of “real-time” correctly classified messages. The graph shows us that E-Sl@ve performs well, as it starts a little lower than LCS, but then comes back and performs roughly equal to LCS.

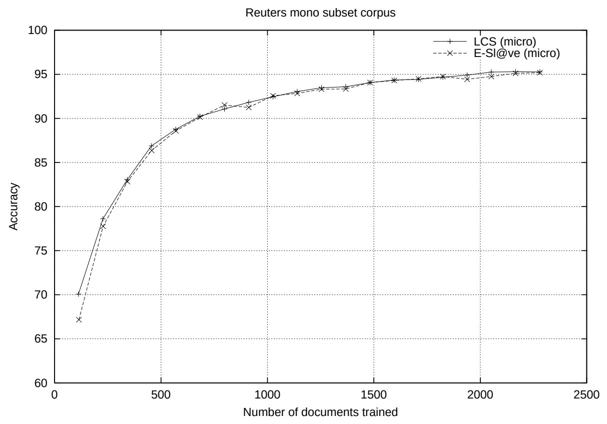

Figure 3.1: E-Sl@ve vs. LCS, Accuracy, *Reuters mono subset* corpus.

The $Accuracy_{micro}$ measure is dominated by the **large** classes (those with many training documents). In order to get an indication of the contribution to the overall classification accuracy for **small** and **large** classes, the **micro-averaged** results have to be compared with the **macro-averaged** results (as these are dominated by the **small** classes).

The results of this *micro/macro* comparison are depicted in figure 3.2. The graph depicted in this figure shows that the $Accuracy_{macro}$ for LCS is initially a lot higher than the $Accuracy_{macro}$ for E-Sl@ve, while for larger numbers of training examples there is only a small advantage for LCS. Therefore, as the **micro-averaged** results of both systems were roughly equal, it might be supposed that on this corpus LCS learns small classes (those with few training documents) a little better than E-Sl@ve does. To be sure about this, a graph is created that shows the averaged accuracy for a class according to its number of training examples. This graph is depicted in figure 3.3. Note that the graph is obtained by counting for all classes, per epoch, the number of training documents, averaging the results (*F1*-measure (3.3.2)) for those classes with an equal number of training documents.

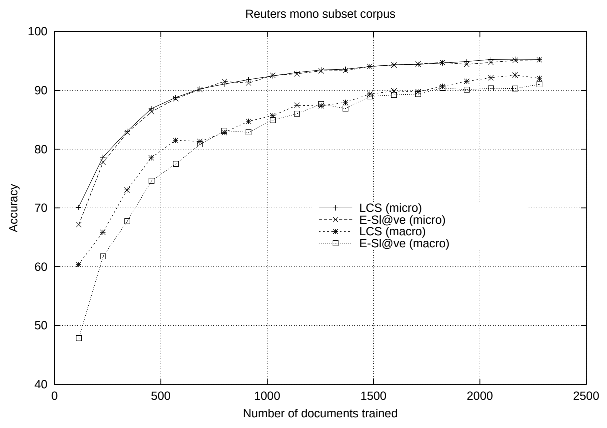

Figure 3.2: E-Sl@ve vs. LCS, micro/macro comparison, *Reuters mono subset* corpus.

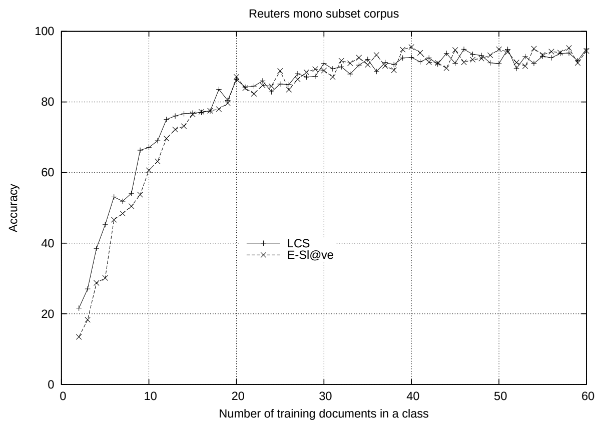

Figure 3.3: E-Sl@ve vs. LCS, class accuracy, *Reuters mono subset* corpus.

As the graph depicted in figure 3.3 shows, LCS indeed learns small classes better than E-Sl@ve does. For classes with more than 20 training examples both systems perform roughly equal.

Edmond corpus

For the *Edmond* corpus, the $Accuracy_{micro}$ for both systems is depicted in figure 3.4. As the graph depicted in this figure shows, LCS performs roughly 5% better at start and ends up with an accuracy that is roughly 3% better compared to E-Sl@ve.

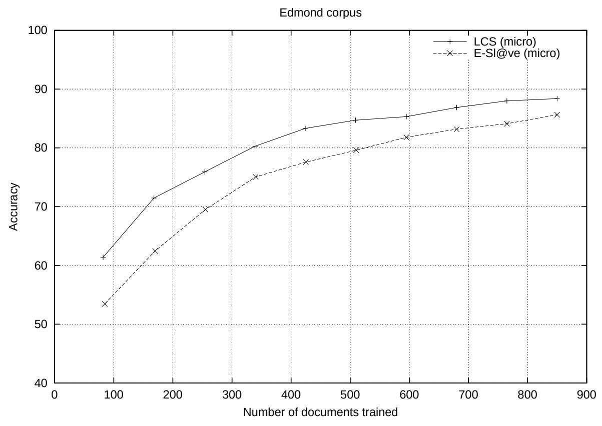

Figure 3.4: E-Sl@ve vs. LCS, Accuracy, *Edmond* corpus.

It is obvious that, looking at the $Accuracy_{micro}$, LCS performs better than E-Sl@ve on this corpus, while both systems performed roughly equal on the *Reuters mono subset* corpus. An explanation for this could be that the *Edmond* corpus contains more "noise" on the pre-classification of its documents compared to the *Reuters mono subset* corpus. This "noise" causes that Balanced Winnow needs more iterations to learn good class profiles. Because E-Sl@ve does not (yet) perform any iterations (and LCS does), this might explain why LCS performs better than E-Sl@ve on the *Edmond* corpus, while both systems perform equal on the *Reuters mono subset* corpus.

Note that the overall lower level of accuracy achieved on this corpus (compared to the *Reuters mono subset* corpus) can be explained by the fact that the *Edmond* corpus is roughly three times smaller than the *Reuters mono subset* corpus (which means that there are less training examples).

As was done for the *Reuters mono subset* corpus, a *micro/macro* comparison has been depicted in figure 3.5. The graph shows that the $Accuracy_{macro}$ for LCS is a lot higher than the $Accuracy_{macro}$ for E-Sl@ve (at start it is even higher than the $Accuracy_{micro}$ for E-Sl@ve). Therefore it might be supposed that (for this corpus) LCS learns small classes a lot better compared to E-Sl@ve. To be sure about this, the graph depicted in figure 3.6 was created (which denotes the averaged accuracy for a class according to its number of training examples). This graph shows that LCS indeed performs a lot better on small classes. Even for larger classes LCS performs significantly better.

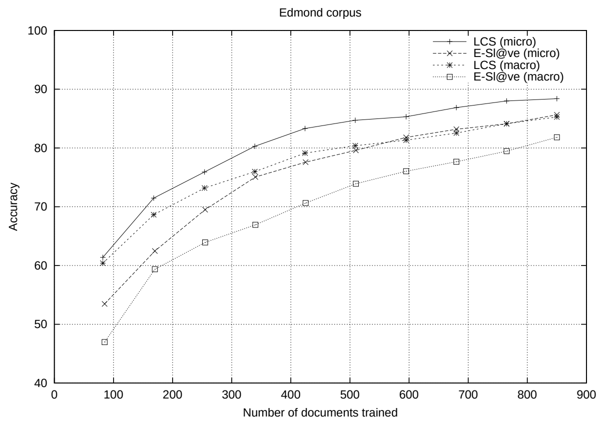

Figure 3.5: E-Sl@ve vs. LCS, micro/macro comparison, *Edmond* corpus.

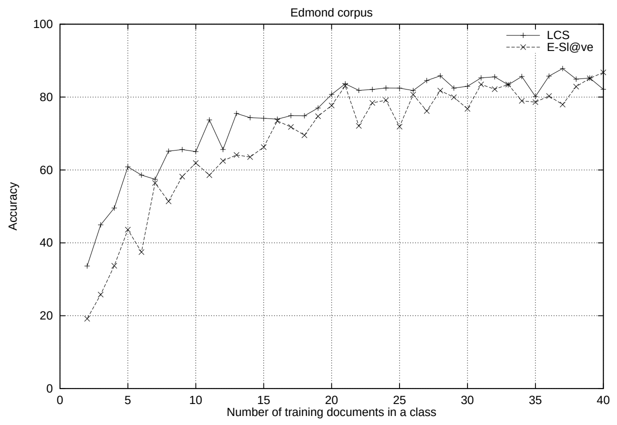

Figure 3.6: E-Sl@ve vs. LCS, class accuracy, *Edmond* corpus.

Conclusion

The results of this experiment show that, for small numbers of training examples, LCS performs better than E-Sl@ve. For larger numbers of training examples, both systems performed roughly equal on the *Reuters mono subset* corpus, while on the *Edmond* corpus LCS performs roughly 3% to 5% better compared to E-Sl@ve.

An explanation for the fact that LCS performs better than E-Sl@ve on the *Edmond* corpus, while on the *Reuters mono subset* corpus both systems perform roughly equal, could be that the *Edmond* corpus contains more "noise" on the pre-classification of its documents compared to the *Reuters mono subset* corpus. This "noise" causes that Balanced Winnow needs more iterations to learn good class profiles. Because E-Sl@ve does not (yet) perform any iterations (and LCS does), this might explain why LCS performs better than E-Sl@ve on the *Edmond* corpus, while both systems perform equal on the *Reuters mono subset* corpus.

In conclusion, it may be stated that the "incremental" Balanced Winnow performs very well.

3.3.5 Experiment: Mistake behaviour

In this experiment the **sort** of *classification errors* of both systems are compared.

In the previous experiment (3.3.3) we have seen that (after a reasonable number of training examples) E-Sl@ve (*incremental learning*) performs roughly as well as LCS (*batched learning*) on the *Reuters mono subset* corpus. This implies that both systems roughly make the same number of mistakes (wrong classifications). Interesting to know is: do both systems make the same sort of classification errors? This experiment explores this issue.

Global setup

Four tests on the *Reuters mono subset* corpus were performed, using the same train sets, test sets, and settings for E-Sl@ve and LCS as in 3.3.3. This time, training on increasing parts of the train set was **not** done, as we are not interested in the learning behaviour.

For each of the four tests, the "Top 6" of best classes for both systems were determined. For this purpose, every test was performed 10 times, each time using a randomly shuffled version of the train set. The "Top 6" consists of those classes that have the lowest average *local ErrorRate*. The *local ErrorRate* for a class c is defined as:

$$ErrorRate_c = \frac{RNS_c + NRS_c}{N}$$

with $N = RS + RNS + NRS + NRNS$ (see 3.3.2).

For all classes the 10 *local ErrorRates* are summed and averaged. The classes with the lowest average *local ErrorRate* form the "Top 6".

3.3.6 Results: Mistake behaviour

The results were satisfying, as both systems appeared to have an equal “Top 6” in all four tests. For one test, the results are depicted in table 3.5.

Top 6	avg. ErrorRate LCS	avg. ErrorRate E-Sl@ve
Cocoa	0.10	0.07
Coffee	0.00	0.00
Gold	0.20	0.07
Iron-steel	0.07	0.13
Nat-gas	0.00	0.10
Sugar	0.00	0.00

Table 3.5: Top 6 classes for E-Sl@ve and LCS.

3.3.7 Performance

To provide an indication of the performance of E-Sl@ve (in training and testing documents), a small test on the *Edmond* corpus was performed for LCS and E-Sl@ve. The same test was performed 10 times, after which the results were averaged. The settings for both systems were the same as in 3.3.3.

The statistics of this test are depicted in table 3.6:

train set	:	850 documents (1518 KByte)
test set	:	284 documents (519 KByte)
total feature space	:	14379 (unique) features
number of classes	:	18

Table 3.6: Statistics of speed performance test.

E-Sl@ve results

The performance of E-Sl@ve is:

Training time: 6.93 sec
Testing time: 1.67 sec

Note that first all documents were cached. Caching all documents (2037 KByte) took roughly 11.70 seconds.

In conclusion, as an average document's size in this corpus is roughly 1.8 KBytes, E-Sl@ve needs 0.0082 seconds to train a message, and needs 0.0058 seconds to test a message.

LCS

The performance of LCS is:

Training time: 2.68 sec
Testing time: 0.95 sec

Caching all documents (2037 KByte) took roughly 3.51 seconds.

In conclusion, as an average document's size in this corpus is roughly 1.8 KBytes, LCS needs 0.0032 seconds to train a message, and needs 0.0033 seconds to test a message.

The results show that the performance of LCS is better compared to E-Sl@ve. Note that LCS is implemented in C++ (executive code), while E-Sl@ve is implemented in *Java* (interpreted code).

3.4 Conclusion

The results of experiments in this chapter showed that the overall performance of *incremental learning* applied to Balanced Winnow is good. For small numbers of training examples E-Sl@ve (*incremental learning*) did not perform as well as LCS (*batched learning*), but for larger numbers of training examples it performed roughly equal to LCS, provide that the corpus contains little "noise".

The last experiment (3.3.5) showed that the global classification errors made by both systems do not differ, as the "Top 6" of best classes were equal for both systems.

According to these results we might conclude that *incremental learning* for Balanced Winnow can be really promising in *automatic email classification*. In the next chapter, an even higher classification accuracy with E-Sl@ve is sought for, by exploring some (possible) optimalisations.

Chapter 4

Optimalisations

In the previous chapter it was shown that the overall classification accuracy of E-Sl@ve is good. In this chapter some (possible) improvements to E-Sl@ve are explored, in order to achieve an even higher accuracy. Respectively *Turbo-Training*, *Aggressive-Training* and *Certainty Based Classification* are introduced.

4.1 Turbo-Training

In a *batched learning* situation (2.3), Balanced Winnow can perform multiple *iterations* on all documents in a train set. In an *incremental learning* situation this is not possible, because each time a new document arrives, the classifiers are trained with this **one** (new) document. Therefore, in the *incremental learning* situation it is even more important than in the *batched learning* situation to get the maximum amount of information out of each document. To achieve this, a **new** heuristic is introduced, which is called *Turbo-Training*.

4.1.1 What is Turbo-Training

With *Turbo-Training* Balanced Winnow (which is *mistake-driven* (2.2)) should learn more from each *mistake*. This may be achieved by speeding up the *promotion* and *demotion* of the weights of *active* terms, according to the *score* of the document. Therefore the *update rules* (2.2.1) are changed as follows:

Positive example
In case document d is labeled *positive* for a class c ($d \in c$), the *active* terms in class c are *promoted* 1, 2 or 3 times according to the score $S_c(d)$:

$$\begin{array}{lcll} S_c(d) & < & \theta^- & \rightarrow \ promote(3) \\ S_c(d) & < & \theta(=1) & \rightarrow \ promote(2) \\ S_c(d) & \leq & \theta^+ & \rightarrow \ promote(1) \ (= \textit{thick-threshold} \text{ heuristic } (2.2.1)) \end{array}$$

in which the function *promote* is defined as follows:

$$\begin{array}{ll} promote(turbofactor): & w^+ := w^+ * \alpha^{turbofactor}; \\ & w^- := w^- * \beta^{turbofactor}; \end{array}$$

Negative example
In case document d is labeled *negative* for a class c ($d \notin c$), the *active* terms in class c are *demoted* 1, 2 or 3 times according to the score $S_c(d)$:

$$\begin{array}{lcll} S_c(d) & > & \theta^+ & \rightarrow \ demote(3) \\ S_c(d) & > & \theta(=1) & \rightarrow \ demote(2) \\ S_c(d) & \geq & \theta^- & \rightarrow \ demote(1) \ (= \textit{thick-threshold} \text{ heuristic } (2.2.1)) \end{array}$$

in which the function *demote* is defined as follows:

$$\begin{array}{ll} demote(turbofactor): & w^+ := w^+ * \beta^{turbofactor}; \\ & w^- := w^- * \alpha^{turbofactor}; \end{array}$$

Note that this heuristic is an alternative to iterative training that conserves incrementality.

4.1.2 Experiment

In this experiment, the results of *Turbo-Training* are compared with the (previously obtained (3.3.3)) results of *normal* training.

The global setup, and settings for both systems, is choosen the same as in 3.3.3. Note that the same order of the training documents in the different shuffled versions of the train set was used, in order to obtain reliable results.

4.1.3 Results

Reuters mono subset corpus

In figures 4.1 and 4.2, the results of this experiment for the *Reuters mono subset* corpus are depicted. Examining the learning curve of the graph depicted in figure 4.1, it can be seen that *Turbo-Training* speeds up the initial learning process extremely. The graph depicted in figure 4.2 provides an even better view, as it can be seen that *Turbo-Training* increases the accuracy for a class with very few training examples with roughly 10% to 20% (on a test set consisting of 767 documents). When a class contains more than 20 training examples, the results of *Turbo-Training* and *normal* training are roughly equal.

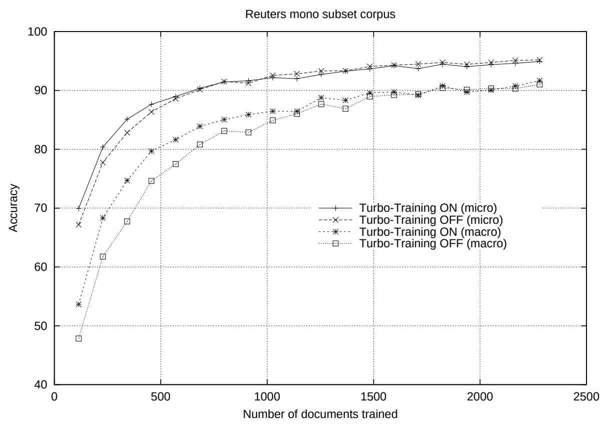

Figure 4.1: Turbo-Training, *Reuters mono subset* corpus.

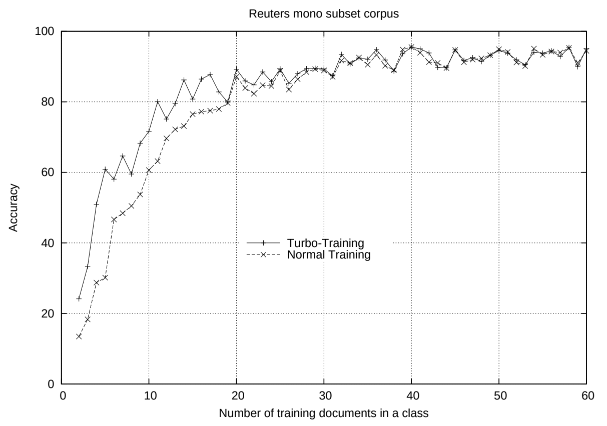

Figure 4.2: Turbo-Training, class accuracy, *Reuters mono subset* corpus.

Edmond corpus

The results for the *Edmond* corpus are depicted in figures 4.3 and 4.4. Because this corpus contains far less training examples compared to the *Reuters mono subset* corpus, *Turbo-Training* speeds up the learning process for the whole range of numbers of training documents (which actually can be seen as the initial range). The graph depicted in figure 4.4 provides a better view, as it shows that *Turbo-Training* increases the accuracy for classes containing a maximum of 17 training examples. When a class contains more than 17 training examples, the results of *Turbo-Training* are roughly equal to the results of *normal* training.

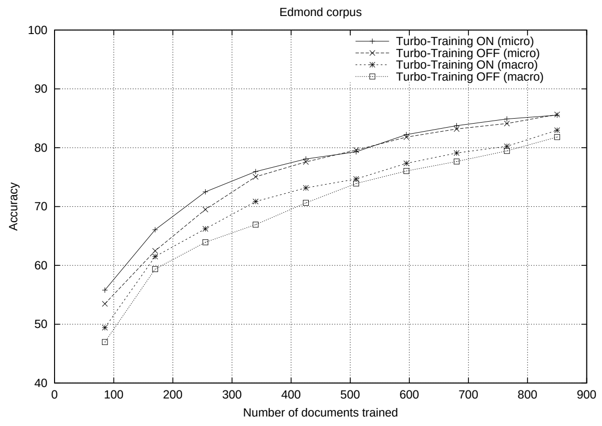

Figure 4.3: Turbo-Training, *Edmond* corpus.

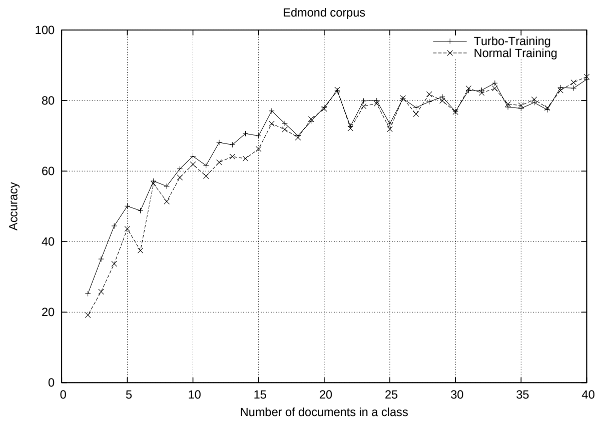

Figure 4.4: Turbo-Training, class accuracy, *Edmond* corpus.

Conclusion

The conclusion for this experiment is very simple: *Turbo-Training* is a perfect alternative to iterative training, which conserves incrementality, as it speeds up the initial learning process by learning classes with small numbers of training examples better.

4.2 Aggressive-Training

In an *incremental learning* situation it is not possible to train *iteratively* on all documents in a train set. But it **is** possible to train iteratively with **one** document. This is exactly what *Aggressive-Training* does. Each email that arrives will be trained *iteratively* until it does not causes a *mistake* anymore. This means that a *negative* example for a class c will be trained *iteratively* until it obtains a score for class c that is below 0.9 ($= \theta^-$). On the other hand, a *positive* example for a class c will be trained *iteratively* until it obtains a score for class c that is above 1.1 ($= \theta^+$).

4.2.1 Experiment

In this experiment, the results of *Aggressive-Training* are compared with the (previously obtained (3.3.3)) results of *normal* training.

The global setup, and settings for both systems, are equal to those in the previous experiment (4.1.2).

4.2.2 Results

Reuters mono subset corpus

The results for the *Reuters mono subset* corpus are depicted in figures 4.5 and 4.6. Examining the learning curve of the graph depicted in figure 4.5, it can be seen that *Aggressive-Training* speeds up the initial learning process a little. The graph depicted in figure 4.6 confirms this, as *Aggressive-Training* increases the accuracy for classes with few training examples. For classes that consist of more than 20 training examples, the results of *Aggressive-Training* roughly equal the results of *normal* training.

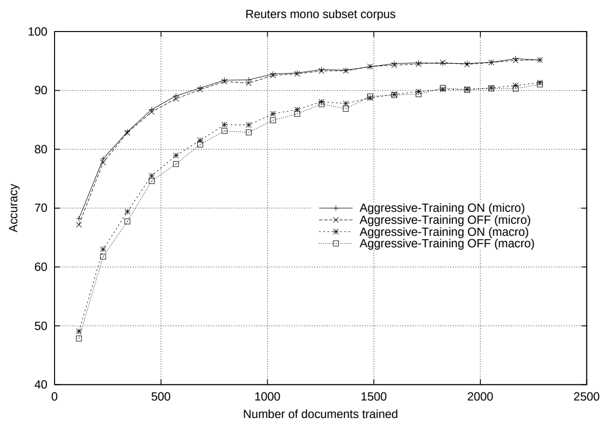

Figure 4.5: Aggressive-Training, *Reuters mono subset* corpus.

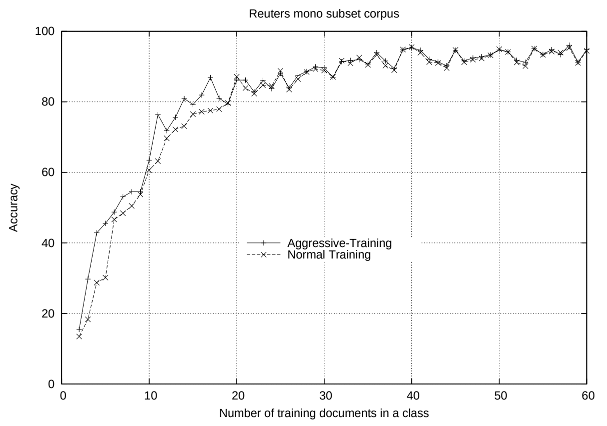

Figure 4.6: Aggressive-Training, class accuracy *Reuters mono subset* corpus.

Edmond corpus

The results for the *Edmond* corpus are depicted in figures 4.7 and 4.8. For this corpus also, a slight speedup in the learning process can be achieved.

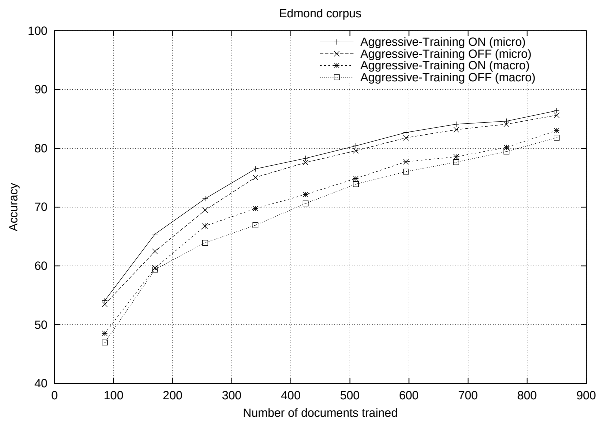

Figure 4.7: Aggressive-Training, *Edmond* corpus.

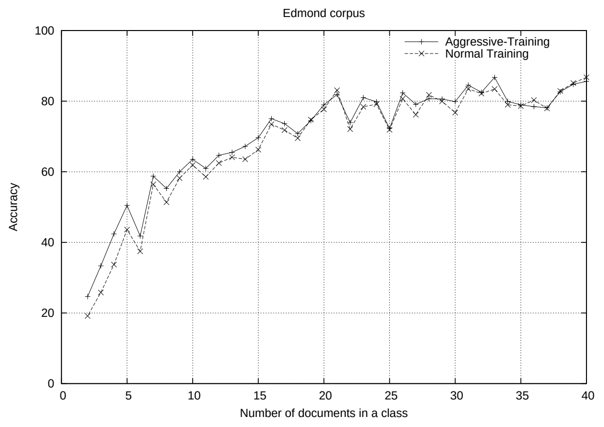

Figure 4.8: Aggressive-Training, *Edmond* corpus.

Conclusion

In conclusion, it can be stated that *Aggressive-Training* speeds up the initial learning process a little, as it learns classes with small numbers of training examples better. Although *Aggressive-Training* does not speed up the learning process as much as *Turbo-Training* does, the *Aggressive-Training* heuristic can be very useful, as it enables the system to *de-train* and *re-train* classes (see 5.2.1). In chapter 5, the *Aggressive-Training* heuristic is used in realising a scenario (*negative relevance feedback*) for a real-life situation.

4.3 Certainty-based classification

This thesis deals with *mono-classification*. This means that a new document must be classified in exactly *one* class. One way of determining this class, is to compute the score $S_c(d)$ of document d for every class c, according to the current state (weight vector) of the classifiers, and then assign d to the class for which d obtained the highest score. This is how most applications work.

However, it is possible that the comparison of the scores for these classes is not such a good measure in determining the relevant class for a new document. This can be explained by the fact that a highest score might be very low for the relevant class according to the *scores history* of previously processed documents for that class, while a less higher score might be very high according to the *scores history* of previously processed documents for other classes.

Therefore a different method in determining the relevant class for a new document is explored. This method uses statistics on the scores of previously processed documents in determining the destination class for a document. Scores of *positive* documents for a class c ($ScPos_c$) and scores of *negative* documents for a class c ($ScNeg_c$) are distinguished:

$$ScPos_c = \{s_1, s_2, \cdots, s_n\}$$

$$ScNeg_c = \{s_1, s_2, \cdots, s_m\}$$

with n the number of scores of previously processed *positive* documents and m the number of scores of previously processed *negative* documents.

Now, when a new document d arrives, for each class c the *yes-probability* and the *no-probability* is computed. The *yes-probability* indicates the probability of d being **relevant** for class c and the *no-probability* indicates the probability of d being **irrelevant** for class c. These probabilities are defined as follows:

$$YesProb_c(d) = \frac{RS_c(S_c(d))}{RS_c(S_c(d)) + NRS_c(S_c(d))}$$

$$NoProb_c(d) = \frac{RS_c(S_c(d))}{RS_c(S_c(d)) + RNS_c(S_c(d))}$$

in which $S_c(d)$ is the score of document d for class c, and RS, RNS and NRS have the same meaning as in 3.3.2 and are defined as follows:

$$RS_c(\gamma) = |ScPos_c[s > \gamma]|$$

$$RNS_c(\gamma) = |ScPos_c[s \leq \gamma]|$$

$$NRS_c(\gamma) = |ScNeg_c[s > \gamma]|$$

in which γ is a threshold.

Finally, for each class c a *certainty* measure is obtained, which indicates how certain it is that document d is **relevant** or **irrelevant** for class c:

$$Cert_c(d) = |YesProb_c(d) - NoProb_c(d)|$$

Three different cases have to be distinguished, in order to clarify the interpretation for $Cert_c(d)$:

1. If $YesProb_c(d) = NoProb_c(d)$ then the classifier for class c does not know whether d is **relevant** or **irrelevant** for class c, as $Cert_c(d) = 0$
2. If $YesProb_c(d) > NoProb_c(d)$ then the classifier for class c predicts that d is **relevant** for class c with a certainty of $Cert_c(d)$.
3. If $YesProb_c(d) < NoProb_c(d)$ then the classifier for class c predicts that d is **irrelevant** for class c with a certainty of $Cert_c(d)$.

A new document d is assigned to a class according to the following assigning rules:

- If there are x classes ($x \geq 1$) for which $YesProb_c(d) \geq NoProb_c(d)$ holds, then document d is assigned to one of those x classes for which $Cert_c(d)$ is highest.
- When the situation is such that for all classes $NoProb_c(d) > YesProb_c(d)$ then the algorithm actually tells us that document d is **irrelevant** for **all** classes. When this happens, d is assigned to the class for which $Cert_c(d)$ is lowest, as in that case the algorithm is least sure about his "no-answer".

4.3.1 Experiment

In this experiment, the results of *Certainty Based Classification* are compared with the (previously obtained (3.3.3)) results of *normal* training.

The global setup, and settings for both systems, are equal to those in the previous experiments (4.1.2, 4.2.1).

4.3.2 Results

The results of this experiment are depicted in figures 4.9 and 4.10, respectively for the *Reuters mono subset* corpus and *Edmond* corpus.

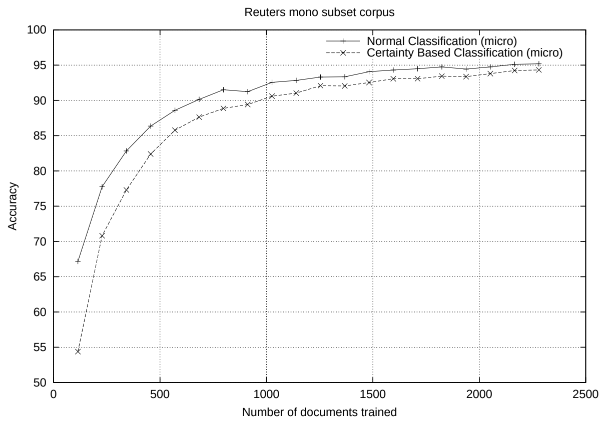

Figure 4.9: Certainty Based Classification, *Reuters mono subset* corpus.

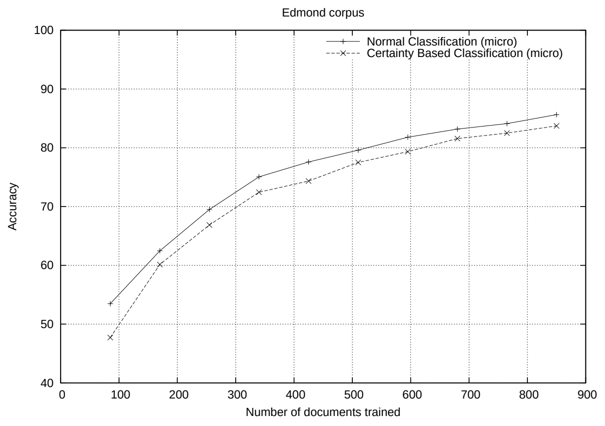

Figure 4.10: Certainty Based Classification, *Edmond* corpus.

According to the results depicted in both graphs, it is obvious that this way of determining the relevant class for a new document performs worse than the "standard" method (where a document is classified in the class for which it obtained the highest score, as described in 3.2.4).

Searching for a cause, it was found that for many of the documents the algorithm has no "certain" answer, because $NoProb_c(d) > YesProb_c(d)$ for all classes c_i. So, for these documents, the algorithm actually answers: "I do not know where to classify this document!". The graph depicted in figure

4.11 shows the percentage of documents in the test set for which E-Sl@ve did not know where to classify them. It can be seen that for small numbers of training examples the percentage of "uncertain" certified documents is higher than for large numbers of training examples. This could be explained by the fact that the *scores history* of classes is less "certain" when few documents are trained.

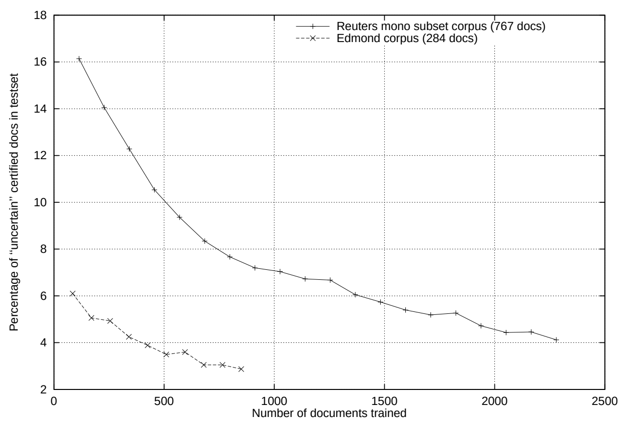

Figure 4.11: Certainty Based Classification, "uncertain" docs.

In conclusion, *Certainty Based Classification* decreases the accuracy, particularly in the initial phase. Therefore, future research on this issue might be useful.

4.4 Conclusion

Several extensions and modifications to E-Sl@ve were introduced. The results were quite satisfying. *Turbo-Training* is a perfect alternative to iterative training, which conserves incrementality, as it speeds up the initial learning process by learning small classes better. *Aggressive-Training* resulted also in a speedup of the learning process, but it does not outperform *Turbo-Training*. Nevertheless, the *Aggressive-Training* heuristic is very useful, as shall be demonstrated in chapter 5. *Certainty based classification* did not perform well, as for many documents the system actually did not know where to classify them, but it might be an interesting issue for future research.

Chapter 5

Negative relevance feedback

Relevance feedback is the information about a document that indicates the class for which that document is relevant, according to the opinion of an "expert". The system needs this feedback in order to be able to (incrementally) train with the document. In an experimental environment, the pre-classification of documents in the corpus can be used as "perfect" *relevance feedback* (as was done in all our previous experiments). In a real-life situation, this is impossible, and therefore *relevance feedback* must be obtained from the user. A difficult point is then to obtain this information such that the user will not find it annoying. In this chapter, *negative relevance feedback* is introduced, which solves this problem elegantly.

5.1 Obtaining relevance feedback

In a real-life situation, when a new email arrives, the global process for E-Sl@ve should be as follows:

1. Classify the new email according to the current state of the classifiers, and file this email automatically in the mailfolder that seems to be most relevant according to this classification.

2. Obtain *relevance feedback* on the email.

3. Incrementally train all classifiers with the single email, according to the *relevance feedback* that was obtained in the previous step.

The bottleneck in this process is the second step. In this step the system actually needs to know whether the email was classified (and thus filed) correctly. In case it was not filed correctly, the system needs to know into which mailfolder it should have been filed. The user has to provide this information (*relevance feedback*) to the system, because the incoming emails are (usually) not labeled...

Two situations can be distinguished after a new email has been classified. A "positive" situation and a "negative" situation:

1. **Positive situation**
   the email has been classified (and thus filed) **correctly**.
2. **Negative situation**
   the email has been classified (and thus filed) **incorrectly**.

In **both** situations, the system needs *relevance feedback* (for incremental learning purposes). The most simple form of obtaining *relevance feedback*, is to prompt the user every time a new email has been filed, and ask him explicitly for *relevance feedback* (which has been simulated in all our previous experiments). This means that, in case the email has been filed correctly, the user has to confirm this, otherwise the user has to indicate into which mailfolder the email should have been filed. It is clear that this **explicit** form of obtaining *relevance feedback* imposes an increased burden and increased cognitive load, as was explored in [20].

The system that has been described in [22] (Swiftfile) uses a more subtle method. It provides three shortcut buttons above each message, which represent the "top 3" classes for which the email seems relevant according to the system. The shortcut buttons can be used to move a message quickly to the specific mailfolder (class). Important to notice is that Swiftfile does not file messages automatically, but that it only provides shortcut buttons, which enables the user to file the message. Implicitly this means that, for every message, the user still has to tell the system for which class (mailfolder) the message is most relevant (by clicking on a shortcut button).

5.2 Negative relevance feedback

E-Sl@ve files messages automatically. Messages that are filed into the wrong mailfolder, will be detected by the user after some time. It is reasonable to assume that the user will move this message to the correct mailfolder. This movement should be detected by the system, because it provides *relevance feedback* on the classification of the email. Actually the user tells the system: "Hey, this email should not be filed here, it should be filed there!". Because this (implicit) feedback is provided only in a "negative" situation (in which the system has filed a message into the wrong mailfolder), it is called *negative relevance feedback*.

In this chapter, it is explored whether an acceptable level of accuracy can be achieved with E-Sl@ve in a real-life situation, when the user only has to provide *negative relevance feedback.* If this **is** possible, this would imply that the user only needs little effort in keeping the system "accurate", as he only has to move misclassified emails to the correct mailfolder (which should occur rarely, after an acceptable level of accuracy is achieved).

The only problem with this scenario is, that in this way E-Sl@ve does not recieve *relevance feedback* in a "positive" situation, as correctly filed messages will never be moved to another mailfolder. Fortunately, this can be solved by slightly modifying the learning process for E-Sl@ve.

5.2.1 Incremental learning process

E-Sl@ve needs *relevance feedback* immediately after a new message has been filed (see the process in 5.1). As was mentioned before, it is no good option to let the user provide this information for every message. Therefore, E-Sl@ve provides its own *relevance feedback.* This is realised by assuming (blindly) that E-Sl@ve always classifies a (new) message initially correctly, using the classification results as the *relevance feedback.* More formally, the process is as follows:

1. Classify message d into class (mailfolder) c_x, which is the class for which d seems to be relevant according to the current weight vector $\overline{w}_{c_i}$ for all classifiers X_{c_i}.
2. Obtain *relevance feedback*: assume (blindly) that class c_x (step 1) is the class for which d is relevant.
3. Incrementally train all classifiers X_{c_i} with message d, using it as a *positive* example for class c_x and as a *negative* example for all other classes c_i, for which $i \neq x$.

This process ensures that E-Sl@ve obtains *relevance feedback* immediately after a new message has been classified, without the need for any interaction with the user, which enables the system to train immediately with this message. For messages that are classified initially into the correct class (step 1), this works fine. Only a problem occurs, when a message is classified into the **wrong** class (step 1), because then the system obtains the **wrong** *relevance feedback* (step 2), and therefore trains with this message (step 3), using it as a *positive* example for the **wrong** class and as a *negative example* for the **correct** class (and all other classes). Fortunately, this "damage" can be repared when the user detects the message was filed into the wrong mailfolder, and moves the message to the correct mailfolder (providing *negative relevance feedback*).

Say that the user moves an email message d from class (mailfolder) c_x to class c_y. This means that, according to the user, E-Sl@ve initially made a mistake in classifying message d. Consequently, class c_x has been trained *positive* with an **irrelevant** example, and class c_y has been trained *negative* with a **relevant** example. Therefore E-Sl@ve has to **de-train** class c_x for message d and has to **re-train** class c_y for message d. This is realised as follows:

1. **De-train**
   Message d is trained as a *negative* example for class c_x (according to the *thick-threshold* heuristic (2.2.1)). This is done *iteratively*, until the score $S_{c_x}(d)$ reaches a value below θ^-. In other words: message d is **demoted** for class c_x until it provides a score below θ^-.

2. **Re-train**
   Message d is trained as a *positive* example for class c_y (according to the *thick-threshold* heurstic (2.2.1)). This is done *iteratively*, until the score $S_{c_y}(d)$ reaches a value above θ^+. In other words: message d is **promoted** for class c_y, until it provides a score above θ^+.

Note that all other classes c_i, for which $i \neq y$, are also trained with message d, using it as a *negative* example for these classes. However, in most cases this should not be necessary, because the score of message d for all those classes should be below θ^- already, as d was never trained as a *positive* example for those classes.

5.3 Non-delayed negative relevance feedback

In the best case in a real-life situation, *negative relevance feedback* is provided immediately. This means that the user detects and moves a misclassified message immediately, even before the arrival of a new message. This may not be very realistic, but it provides a first indication of how *negative relevance feedback* performs.

5.3.1 Experiment

In this experiment, the results of **non-delayed** *negative relevance feedback* are compared with the (previously obtained) results of *Aggressive-Training* (see 4.2).

The experiment was performed on both the *Reuters mono subset* corpus and *Edmond* corpus, using the labeling of documents as the "perfect" *relevance feedback*. The global setup (train sets, test sets and parameter settings for E-Sl@ve) is the same as in 3.3.3. To obtain reliable results, the same order of training documents in the different shuffled versions of the train set was used.

Simulation setup

To simulate the situation of **non-delayed** *negative relevance feedback*, the learning process for E-Sl@ve was as follows:

1. Classify message d into class c_x, which is the class for which d seems to be relevant according to the current state of all classifiers X_{c_i}.
2. Incrementally train all classifiers, using d as a *positive* example for class c_x and as a *negative* example for all other classes. (Assume blindly that the results in step 1 are correct.)
3. Obtain *relevance feedback* (by checking the label of message d), which provides the information: d is relevant for class c_y.
4. **if** $c_x \neq c_y$ **then**: **de-train** class c_x for message d and **re-train** class c_y for message d.

The results of *Aggressive-Training* (4.2.2) were obtained in a situation which simulates that the user provides *relevance feedback* **explicitly** (as was the case for all our previous experiments). The learning process for E-Sl@ve was as follows:

1. Classify message d into class c_x, which is the class for which d seems to be relevant according to the current state of all classifiers X_{c_i}.
2. Obtain *relevance feedback* (by checking the label of message d), which provides the information: d is relevant for class c_y.
3. Incrementally train all classifiers, using d as a *positive* example for class c_y and as a *negative* example for all other classes.

Note that in both simulations, an email is trained *iteratively* until it does not cause a *mistake* anymore (which is called *Aggressive-Training...*). This means that a *positive* example for a class c will be trained *iteratively* until it obtains a score for class c that is above θ^+. A *negative* example for a class c will be trained *iteratively* until it obtains a score for class c that is below θ^-.

5.3.2 Results

The results of this experiment are depicted in figures 5.1 and 5.2, respectively for the *Reuters mono subset* corpus and *Edmond* corpus. The results show that *negative relevance feedback* performs roughly equal to *Aggressive-Training*, and at some points it performs even better. This is striking, because the opposite was assumed.

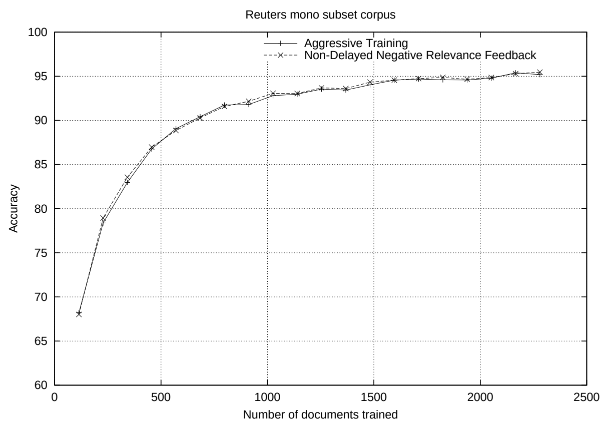

Figure 5.1: Non-Delayed Negative Relevance Feedback, *Reuters mono subset* corpus

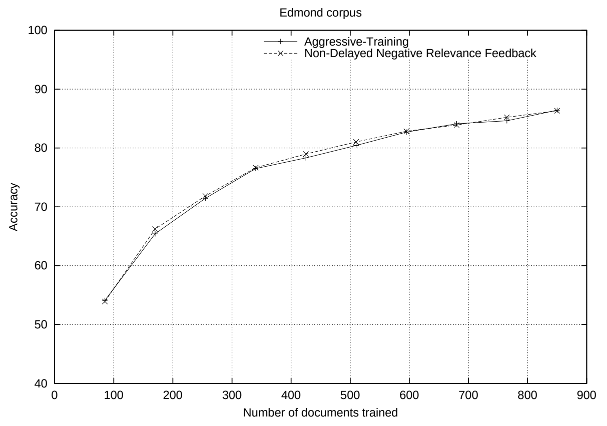

Figure 5.2: Non-Delayed Negative Relevance Feedback, *Edmond* corpus

In a real-life situation it is not likely that a misclassified message will be detected and moved immediately after it has been filed. For this reason, it is too early to conclude that the *negative relevance feedback* scenario works fine in a real-life situation. The only conclusion for now is, that **de-training** and **re-training** of classes, work extremely well.

5.4 Delayed negative relevance feedback

In a real-life situation, it is not reasonable to assume that the user will provide *negative relevance feedback* immediately. Usually there will be a delay between the moment the system files a message d (into the wrong mailfolder), and the moment the user moves message d into the correct mailfolder. This delay could effect the *Accuracy* of the system, as in the meanwhile new messages arrive that can be filed (and thus trained) incorrectly due to the currently (and temporarily) "instable" state of the classifiers. For this purpose, an experiment was performed to see whether the **delayed** *negative relevance feedback* scenario decreases the *Accuracy*.

5.4.1 Experiment

In this experiment, the results of **delayed** *negative relevance feedback* are compared with the (previously obtained) results of **non-delayed** *negative relevance feedback* (see 5.3.1).

The global setup (all settings, train sets and test sets) for this experiment is equal to the global setup for the previous experiment.

Simulation setup

To simulate the situation of **delayed** *negative relevance feedback*, each message d that has been misclassified is assigned a "delay-value", denoted as δ_d. This delay-value is randomly choosen in the range $[\delta^-, \delta^+]$, with ($\delta^- \leq \delta^+$, $\delta^- \geq 0$). If $\delta_d = 0$, it is assumed that *negative relevance feedback* on d is provided immediately, else it is assumed that it takes δ_d more messages to be processed first, before *negative relevance feedback* is provided on d.

To describe the simulation more formally, a train set $T = \{d_1, \cdots, d_n\}$, is defined (n denoting the number of messages in the train set) and a function $Time(d_i)$ which determines the number of messages that have been processed since message d_i was processed.

The process is as follows:

1. Classify message d_i into class c_x, which is the class for which d_i seems to be relevant according to the current state of all classifiers X_{c_z}.
2. Incrementally train all classifiers, using d_i as a *positive* example for class c_x and as a *negative* example for all other classes.
3. Obtain *relevance feedback* (by checking the label of message d_i), which provides the information: d_i is relevant for class c_y.
4. **if** $c_x \neq c_y$ **then**: randomly assign a delay-value δ_{d_i} to d_i.
   (with $\delta^- \leq \delta_{d_i} \leq \delta^+$).
5. **for** all messages d_j that were assigned a delay δ_{d_j} **do**
   if $Time(d_j) \geq \delta_{d_j}$ **then**: use message d_j for **de-training** and **re-training** of classes.

Note that when range $[d^-, d^+]$ is choosen as $[0, 0]$, the situation of **non-delayed** *negative relevance feedback* is obtained. Note also that there never need to be more than d^+ messages queued (which are messages d for which $Time(d) < \delta_d$ holds).

In this experiment, several tests were performed, using the following ranges:
$[d^-, d^+] = [0, 0]$ (equals **non-delayed** *negative relevance feedback*)
$[d^-, d^+] = [0, 10]$
$[d^-, d^+] = [0, 20]$
$[d^-, d^+] = [0, 50]$

5.4.2 Results

The results of this experiment are depicted in figures 5.3 and 5.4. It can be seen that a wider delay range causes a more decreased accuracy of the system, particularly for small numbers of training documents. For large numbers of training documents (> 800), the results seem to converge.

From the results of this experiment we may conlcude that it indeed **is** possible to obtain a very acceptable level of accuracy with E-Sl@ve when the user only has to provide *negative relevance feedback*, provide that *negative relevance feedback* is given on **all** messages that have been misclassified.

5.4.3 Lazyness

In real life, it could happen that some misclassified messages are never moved to the correct mailfolder. When this happens, most of the times, it is caused by the "lazyness" of users. Consequently, the system will be a little "confused", as it is actually trained with incorrect information (which is never corrected). To explore the effect of this confusion on the accuracy of the

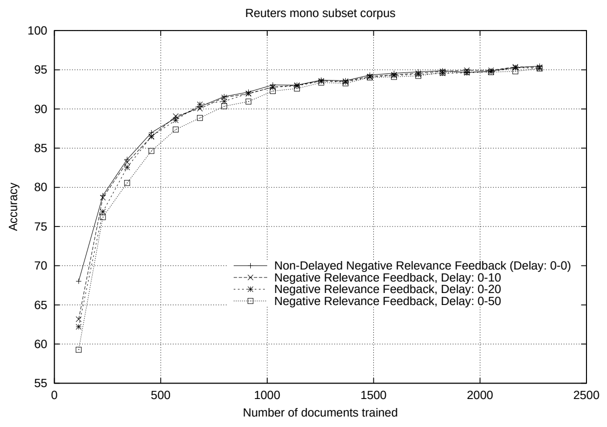

Figure 5.3: Delayed Negative Relevance Feedback, *Reuters mono subset* corpus

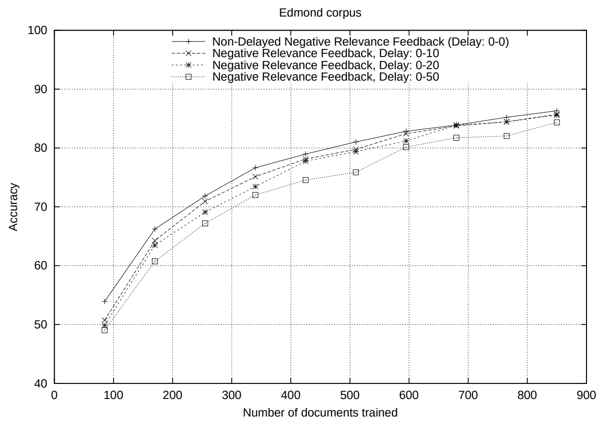

Figure 5.4: Delayed Negative Relevance Feedback, *Edmond* corpus

system, an additional experiment is performed which simulates a **delayed** *negative relevance feedback* scenario in which it is assumed that a certain percentage of misclassified messages will never be moved to the correct mailfolder (and therefore will not be used for **de-training** and **re-training** classes).

In figures 5.5 and 5.6 the results are depicted for situations in which 0%, 10% and 20% of the number of misclassified messages is assumed to be never moved to the correct mailfolder. The delay range was set on $[0, 20]$.

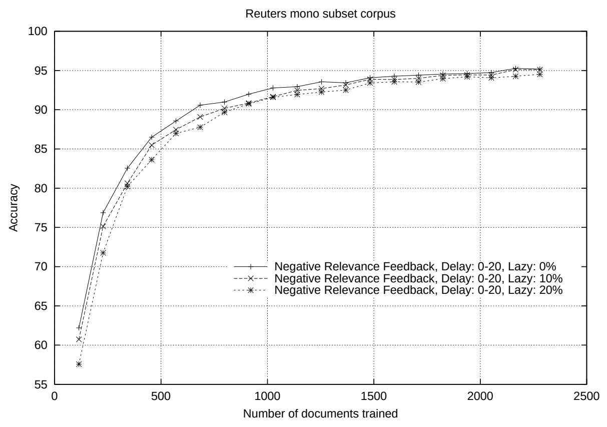

Figure 5.5: Delayed Negative Relevance Feedback, *Reuters mono subset* corpus

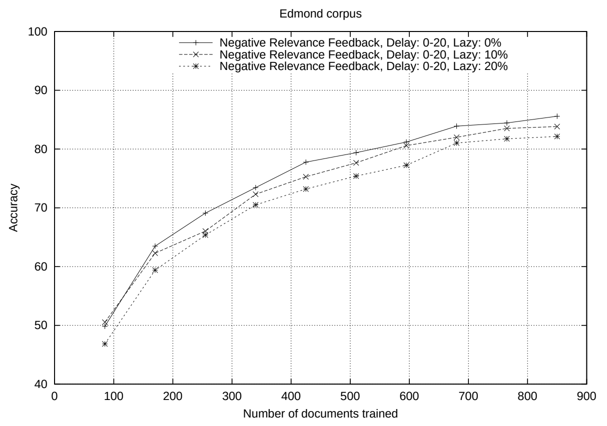

Figure 5.6: Delayed Negative Relevance Feedback, *Edmond* corpus

The graphs show that, as was supposed, a higher “lazy” percentage results in a little decreased accuracy. Fortunately, in real life, the user shall not benefit by not moving misclassified messages to the correct folder, which therefore makes it reasonable to assume that this occurs rarely.

5.5 Conclusion

In this chapter an elegant solution to the problem of obtaining *relevance feedback* in a real-life situation has been provided: the *negative relevance feedback* scenario (5.2). *Negative relevance feedback* ensures that the user only needs little effort in keeping the system accurate, as he only needs to move misclassified messages to the correct mailfolder (which should occur rarely, after an acceptable level of accuracy is achieved).

Results of experiments that simulate the *negative relevance feedback* scenario show that, even in the presence of "lazy" users, a very acceptable level of accuracy can be achieved. Therefore it might be concluded that E-Sl@ve could become a useful and valuable addition to any (*Java*-compliant) email-client.

Chapter 6

Conclusion

Looking at the results of experiments performed in this thesis, the overall conclusion is that E-Sl@ve could become a useful and valuable addition to any *Java*-compliant email-client.

The core of E-Sl@ve, an "incremental" Balanced Winnow (learning algorithm), has (empirically) proved to be very accurate in classifying emails (and short newspaper articles). Initially comparing E-Sl@ve to LCS, a system that uses Balanced Winnow in a "batched" fashion, the results of E-Sl@ve were, after a reasonable number of training examples, roughly as good as the results of LCS. Only for small numbers of training examples, E-Sl@ve performed worse than LCS. According to these results, E-Sl@ve seemed already promising in *automatic email classification*, but an even higher accuracy with E-Sl@ve was sought for by exploring some (possible) optimalisation.

Three (possible) optimalisations for E-Sl@ve were explored. Two of those slightly change the training heuristic of Balanced Winnow: *Turbo-Training* and *Aggressive-Training*. The third provided a different heuristic in classifying a new message: *Certainty Based Classification*.

The results of *Certainty Based Classification*, which ensures that new messages are classified according to a "certainty", were not satisfying. The main reason for this was that, particularly for small numbers of training examples, classifiers were extremely "uncertain" about their prediction.

On the other hand, the results of *Turbo-Training* and *Aggressive-Training* were quite satisfying. *Turbo-Training*, an alternative for iterative training which conserves incrementality, resulted in a strong speedup of the initial learning process, as classes consisting of only few training examples were learned much better. *Aggressive-Training*, which ensures that a new message is trained iteratively until the algorithm predicts the correct class for this message, resulted in a slight speedup of the initial learning process.

Although *Aggressive-Training* did not perform as well as *Turbo-Training*, it is useful in realising a scenario suitable for a real-life situation (*negative relevance feedback*).

E-Sl@ve (incrementally) learns from (new) messages according to the feedback that is provided on messages that have been classified. In a real-life situation, this *relevance feedback* must be obtained from the user in order to remain accurate. A difficult point is then to obtain *relevance feedback* such that the user will not find it annoying. In this thesis an elegant solution to this problem has been provided, named *negative relevance feedback*. *Negative relevance feedback* ensures that the user only needs little effort in keeping the system accurate, as he only needs to move misclassified messages to the correct class (mailfolder). Results of experiments (which simulated a real-life situation) have shown that, using *negative relevance feedback*, a high level of accuracy can be achieved.

6.1 Future research

The ideas for the issues mentioned in this section were all acquired during the production of this thesis, but there was no time left to explore them.

6.1.1 Term selection

Most classes depend only on a small subset of indicative features and not on all the features that occur in documents that belong to that specific category. Therefore, it seems plausible to discard "noisy" features for every class, as it improves efficiency and possibly also the accuracy of the classifier. Some classification systems (like LCS, see [12]) have a feature selection preprocessing stage. In an *incremental* approach this is not possible, because the class profiles are build "on-the-fly", adding new features as incoming documents are processed. Therefore, a proposal for a new term selection technique that could be used for incremental training (with Balanced Winnow) is introduced.

Motion-based term selection

As in [13] is shown, the Winnow k-steps strategy does not work well. Therefore another strategy, based on the number of promotions and demotions (together called *motions*) of a feature, is proposed.

This technique uses a *UC* ratio, which is defined as follows:

$$UC_f = \frac{|promo_f - demo_f|}{promo_f + demo_f}$$

in which $promo_f$ is the number of promotions for feature f and $demo_f$ the number of demotions for that feature. This ratio is an indicator of the *uncertainty* (see [17]) in the contribution of this feature to the score. Apart from the case in which $promo_f = demo_f = 0$, this value more or less decreases from 1 to a small number.

Example approaches for selecting terms (**explicitly** or **implicitly**) are:

- **Explicit**
  - discard all terms for which $UC < k$ holds, in which k is a certain threshold.
  - select the top k terms per class with the highest UC.
- **Implicit**
  adapt the score computation of Balanced Winnow, as follows:

$$S_c(d) = \sum_{j=1}^{m} (w_c^+(f_j) - w_c^-(f_j)) \cdot s_d(f_j) \cdot UC_{f_j} > \theta$$

Note that for the **explicit** approaches something "smart" has to be done, as features that have been discarded could become important again in the future.

6.1.2 Threshold range

Recent research with LCS [1] explored the effect of different values for θ^- and θ^+ on the classification accuracy of Balanced Winnow. Performing an experiment in which for several different combinations of θ^- and θ^+ the classification accuracy was determined, it was found that using $\theta^- = 0.6$ and $\theta^+ = 3.0$ resulted in an increased accuracy of roughly 3% compared to results that were already very good.

Because LCS uses Balanced Winnow in a "batched" fashion, it is useful to perform a similar experiment for E-Sl@ve (which uses Balanced Winnow in an "incremental" fashion) to check whether this results in an increased accuracy also.

[1] Linguistical Classification System, developed at the Katholieke Universiteit of Nijmegen

A quick test using $\theta^- = 0.6$ and $\theta^+ = 3.0$, yielded the results as depicted in figures 6.1 and 6.2.

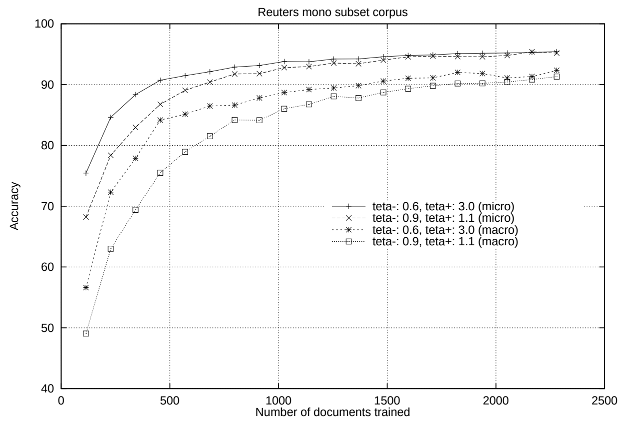

Figure 6.1: Theta test, *Reuters mono subset* corpus

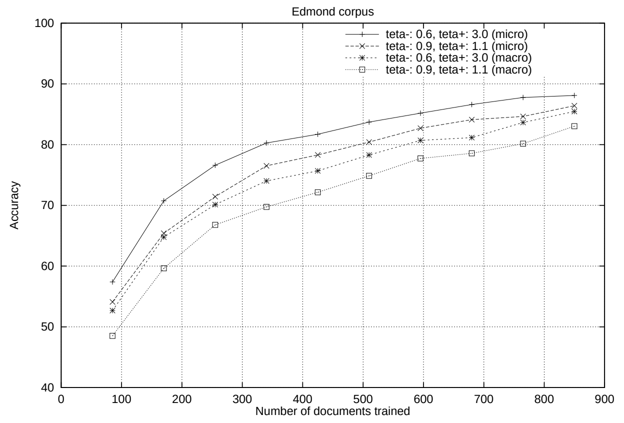

Figure 6.2: Theta test, *Edmond* corpus

From the results is clear that modifying the thresholds increases the overall learning behaviour quite extremely. Note that the *Aggressive-Training* heuristic (4.2) was used for this test.

6.1.3 Certainty Based Classification

As the results of experiments in this thesis showed, *Certainty Based Classification* did not perform well. The main reason for this was that many documents were certified “uncertain”. In order to make this classification heuristic work, more research has to be done. For example, the "uncertain" documents could be examined and removed from the train set in order to see if accuracy grows. Also adaptions to the *yes-probability* and *no-probability* could be sought for, in order to achieve more reliable certainty measures.

6.2 Further work

E-Sl@ve should be adapted to a popular (*Java*-compliant) email-client such as Netscape Messenger.

Acknowledgements

I would like to thank the company Edmond R&D that generously offered me a place for writing this thesis.

Special thanks to:

- Prof. C.H.A Koster and Dr. Paul Jones, who generously assisted me during this thesis.
- My parents
- Marieke Linders, for her love and support.

Bibliography

[1] Apté C., Damerau F. Automatic learning of decision rules for text categorization. *ACM Transactions on Information Systems*, 12(3):233–251, january 1994.

[2] Barret R. and Selker T. AIM: A new approach for meeting information needs. *Technical Report, IBM Research*, october 1995.

[3] Beney J. The LCS Profiling System User Manual. version 1.2, may 2000.

[4] Blum A., Mitchell T. Combining labeled and unlabeled data with co-training. *In Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, pages 92–100.

[5] Cohen W.W. Fast effective rule induction. *Machine Learning: Proceedings of the Twelfth International Conference*, 1995.

[6] Cohen W.W. Learning Rules that Classify E-mail. *In Proceedings of the 1996 AAAI Spring Symposium on Machine Learning and Information Access*, pages 18–25, 1996.

[7] Cohen W.W. Learning with Set-valued Features. *In Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.

[8] Dagan I., Karov Y., Roth D. Mistake-driven learning in text categorization. *In Proceedings of EMNLP-97, 2nd Conference on Empirical Methods in Natural Language Processing*, 1997.

[9] Helfman J. Isbell C. Ishmail: Immediate Identification of Important Information. *In Proceedings of ECIR 2002*, 1995.

[10] Joachims T. Text categorization with Support Vector Machines: learning with many relevant features. *In Proceedings of ECML-98, 10th European Conference on Machine Learning*, version 1.2, may 2000.

[11] Kiritchenko S., Matwin S. Email Classification with Co-Training. october 2001.

[12] Koster C.H.A. IR2 dictaat: Full-Text Information Retrieval. march 2002.

[13] Koster C.H.A., Ragas H. Four text classification algorithms compared on a dutch corpus. *In Proceedings of SIGIR-98, 21st ACM International Conference on Research and Development in Information Retrieval*, augustus 1998.

[14] Littlestone N. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[15] Maes P. Agents that Reduce Work and Information Overload. *Communications of the ACM*, 37(7):31–40, july 1994.

[16] Payne T.R., Edwards P. Interface Agents that Learn: An Investigation of Learning Issues in a Mail Agent Interface. *Applied Artificial Intelligence*, 11:1–32, 1997.

[17] Peters C., Koster C.H.A. Uncertainty-based noise reduction and termselection in text categorization. *ECIR 2002*, april 2002.

[18] Provost J. Naïve Bayes vs. Rule-Learning in Classification of Email. *In Proceedings of ECIR 2002*, 2000.

[19] Rocchio J.J. Relevance feedback in Information Retrieval. *The Smart Retrieval system - experiments in automatic document processing*, pages 313–323, 1971.

[20] Ryen et. al. The Use of Implicit Evidence for Relevance Feedback in Web Retrieval. *In Proceedings of ECIR 2002*, march 2002.

[21] Sebastiani F. Machine Learning in Automated Text Categorization. *Technical report IEI-B4-31-1999, Instituto di Elaborazione dell'Informazione, Consiglio Nazionale delle Ricerche, Pisa, IT, 1999. Submitted for publication to ACM Computing Surveys.*, 2000.

[22] Segal R.B. and Kephart J.O. SwiftFile: An intelligent assistant for organizing email. *In AAAI 2000 Spring Symposium on Adaptive User Interfaces*, 2000.