Type Classification of Semi-Structured Documents*

Markus Tresch, Neal Palmer, Allen Luniewski

IBM Almaden Research Center

Abstract

Semi-structured documents (e.g. journal articles, electronic mail, television programs, mail order catalogs, ...) are often not explicitly typed; the only available type information is the implicit structure. An explicit type, however, is needed in order to apply object-oriented technology, like type-specific methods.

In this paper, we present an experimental vector space classifier for determining the type of semi-structured documents. Our goal was to design a high-performance classifier in terms of accuracy (recall and precision), speed, and extensibility.

Keywords: file classification, semi-structured data, object, text, and image databases.

1 Introduction

Novel networked information services [ODL93], for example the World-Wide Web, offer a huge diversity of information: journal articles, electronic mail, C source code, bug reports, television listings, mail order catalogs, etc. Most of this information is semi-structured. In some cases, the schema of semi-structured information is only partially defined. In other cases, it has a highly variable structure. And in yet other cases, semi-structured information has a well-defined, but unknown schema [Sch93]. For instance, RFC822 e-mail follows rules on how the header must be constructed, but the mail body itself is not further restricted.

The ability to deal with semi-structured information is of emerging importance for object-oriented database management systems, storing not only normalized data but also text or images. However, semi-structured documents are often not explicitly typed objects. Instead they are, for instance, data stored as files in a file system like UNIX. This makes it difficult for database management systems to work with semi-structured data, because they usually assume that objects have an explicit type.

Hence, the first step towards automatic processing of semi-structured data is *classification*, i.e., to assign an explicit type to them [Sal89]. A classifier explores the implicit structure of such a document and assigns it to one of a set of predefined types (e.g. document categories, file classes, ...) [GRW84, Hoc94]. This type can then be used to apply object-oriented techniques. For example, type-specific methods can be triggered to extract values of (normalized and non-normalized) attributes in order to store them in specialized databases such as an object-oriented database for complex structured data, a text database for natural language text, and an image database for pictures.

Such a classifier plays a key role in the Rufus system [SLS+93], where the explicit type of a file, assigned by the classifier, is used to trigger type-dependent extraction algorithms. Based on this extraction, Rufus supports structured queries that can combine queries against the extracted attributes as well as the text of the original files. In general, a file classifier is necessary for any application that operates on a variety of file types and seeks to take advantage of the (possibly hidden) document structure.

Classifying semi-structured documents is a challenging task [GRW84, Sal89]. In contrast to fully structured data, their schema is only partially known and the assignment of a type is often not clear. But as opposed to completely unstructured information, the analysis of documents can be guided by partially

*Research partially supported by Wright Laboratories, Wright Patterson AFB, under Grant Number F33615-93-C-1337.

Proceedings of the 21st VLDB Conference Zurich, Swizerland, 1995

available schema information and must not fully rely on a semantic document analysis [Sal89, Hoc94]. As a consequence, much better performing classifiers can be achieved. However, neither the database nor the information retrieval community have come up with comprehensive solutions to this issue.

In this paper, we present a high performance classifier for semi-structured data that is based on the vector space model [SWY75]. Based on our experiences with the Rufus system, we define high performance as:

- *Accuracy.* The classifier must have an extremely low error rate. This is the primary goal of any classifier. For example, to be reliable for file classification an accuracy of more than 95% must be achieved.
- *Speed.* The classifier must be very fast. For example, to be able to classify files fed by an information network, classification time must be no more than 1/10 of a second.
- *Extensibility.* The classifier must easily be extensible with new user-defined classes. This is important to react to changing user demands. The ability of quick and incremental retraining is crucial in this context.

The remainder of the paper is organized as follows. In Section 2, we review the basic technology of our experimental vector space classifier and compare the accuracy of several implementation alternatives. In Section 3, we introduce a novel confidence measure that gives important feedback about how certain the classifier is after classifying a document. In Section 4, we show that finding a good schema is crucial for the classifier's performance. We present techniques for selecting distinguishing features. In Section 5, we compare the vector space classifier with other known classifier technologies, and conclude in Section 6 with a summary and outlook.

2 An Experimental Vector Space Classifier

In this section, we introduce our experimental vector space classifier (VSC) system. The goal was to build an extremely high-performance classifier, in terms of accuracy, speed, and extensibility for the *classification of files by type*. UNIX was considered as a sample file system. The classifier examines a UNIX file (a document) and assigns it to one of a set of predefined UNIX file types (classes). To date, the classifier is able to distinguish the 47 different file types illustrated in Figure 1. These 47 types were those that were readily available in our environment. Note that the classifier presented here can be generalized to most non-UNIX file systems. In the sequel, we briefly review the underlying *vector space model* [SWY75]. We focus on issues that are unique and novel in our particular implementation.

VSCs are created for a given classification task in two steps:

1. *Schema definition.* The schema of the classifier is defined by describing the names and features of all the classes one would like to identify. The features, say $f_1 \ldots f_m$, span an m-dimensional feature space. In this feature space, each document d can be represented by a vector of the form $v_d = (a_1, \ldots, a_m)$ where coefficient a_i gives the value of feature f_i in document d.
2. *Classifier training.* The classifier is trained with training data – a collection of typical documents for each class. The frequency of each feature f_i in all training documents is determined. For each class c_i, a centroid $v_{c_i} = (\overline{a}_1, \ldots, \overline{a}_m)$ is computed, whose coefficients $\overline{a}_i$ are the mean values of the extracted features of all training documents for that class.

Given a trained classifier with centroids for each class, classification of a document d means finding the "most similar" centroid v_c and assigning d to that class c. A commonly used *similarity measure* is the *cosine metric* [vR79]. It defines the distance between document d and class centroid c by the angle α between the document vector and the centroid vector, that is,

$$\mathrm{sim}(d, c) = \cos\alpha = \frac{v_d \cdot v_c}{|v_d||v_c|} .$$

Building centroids from training data and using the similarity measure allows for very fast classification. To give a rough idea, an individual document can be classified by our system in about 40 milliseconds on an IBM RISC System/6000 Model 530H. In Section 2.2, we will compare the accuracy of the cosine metric with common alternatives.

2.1 Defining the Classifier's Schema

A feature is some identifiable part of a document that distinguishes between document classes. For example, a feature of LaTeX files is that file names usually have the extension "`.tex`" and that the text frequently contains patterns like "`\begin{...}`".

Features can either be boolean or counting. Boolean features simply determine whether or not a feature occurred in the document. Counting features determine how often a feature was detected. They are useful to partially filter out "noise" in the training data. Consider for example CSource files, having a lot of curly

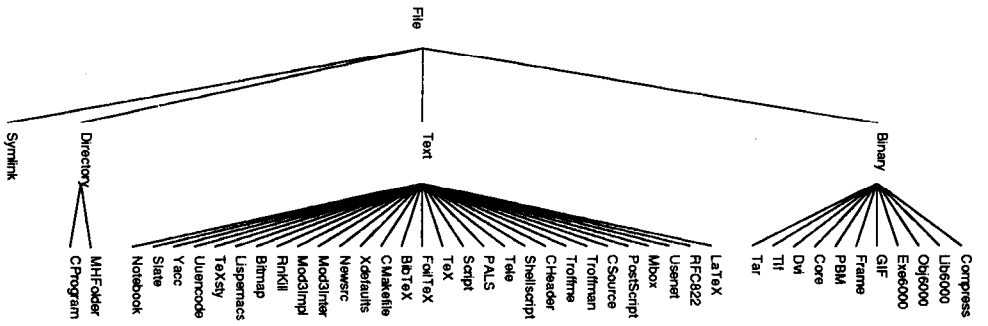

Figure 1: Class hierarchy of the experimental file classifier

braces. Many other file types have curly braces too, but only a few. Hence, counting this feature instead of just noting its presence would differentiate CSOURCE files from others.

In the remainder of this paper, we assume boolean features only. Using counting features is part of future work. In general, counting features in a VSC must be normalized before using, like proposed for example by the non-binary independence model [YMP89].

To define the schema of the file classifier, four different types of features are supported. They can be used to describe patterns that characterize file types:

- A **filenames** feature specifies a pattern that is matched against the name of a file;
- A **firstpats** feature specifies a pattern that is matched against the first line of a file;
- A **restpats** feature specifies a pattern that is matched against any line of a file;
- An **extract** feature specifies that there exists an extraction function (e.g. a C program) to determine if the feature is present.

The feature types **filenames**, **firstpats**, and **restpats** are processed by a pattern matcher. For performance reasons, this is a finite state machine specially built from the classifier schema. Patterns can either be string literals or regular expressions. The regular expressions supported are similar to the regular expressions of the UNIX "ed" command. The feature type **extract** is used to define file properties that cannot be described by regular expressions. For instance, **extract** features can be programmed to check whether a document is an executable file or a directory.

Any feature can be defined as **must**, which means that its occurrence is mandatory. If such a feature is not present in a given file, the file cannot be a member of that class. Notice that the converse is not true: the presence of a **must** feature does not force a type match.

Example 1: Figure 2 shows an excerpt of a sample classifier schema, defining classes for POSTSCRIPT pictures, LATEX documents, MHFOLDER directories, and COMPRESS files.

A **filenames** feature specifies that names of POSTSCRIPT files usually end with the extension "**.ps**", names of LATEX files with "**.tex**", and names of COMPRESS files with "**.z**" or "**.Z**". They are all defined as regular expressions, as indicated by the keyword "**regexp**".

A **firstpats** feature is defined for POSTSCRIPT files. It is a regular expression, saying that the first lines of these files always begin with "**%!**". This pattern is given with a **must** keyword, i.e., it must be present in POSTSCRIPT files.

restpats features specify that POSTSCRIPT files usually contain the two string literals "**%EndComments**" and "**%Creator**" and that LATEX files often contain the string literals "**\begin{**", "**\end{**", or "**{document}**".

Extraction functions exist for classes MHFOLDER and COMPRESS. Notice, that the implementation of extraction functions is not part of the classifier schema. However, by naming convention, they are implemented by C functions called "**ex_MHFolder**" and "**ex_Compress**" respectively. For example, "**ex_Compress**" searches for a file checksum and "**ex_MHFolder**" opens the directory and looks for mail files. ◇

Finding appropriate features for each class is crucial to the accuracy of a classifier [Jam85]. This has been verified by our experiments. For example, to define the 47 classes of the UNIX file classifier, a total of 206 features were carefully specified. We come therefore back to the issue of feature selection in Section 4.

2.2 Alternative Similarity Metrics

Diverse similarity metrics are proposed in the literature. For example, [vR79] describes Asymmetric, Cosine, Dice, Euclidian, Jaccard, and Overlap distance.

Table 1 summarizes our extensive classifier performance experiments. Experiments involved choosing random subsets from a collection of 26MB of sample

```
PostScript {
        filenames {
                "\.ps$" regexp
        }
        firstpats {
                "^%!" regexp must
        }
        restpats {
                "%EndComments"
                "%Creator:"
        }
}

MHFolder {
        extract
}
```

```
LaTeX {
        filenames {
                "\.tex$" regexp
        }
        restpats {
                "\begin{"
                "\end{"
                "{document}"
        }
}

Compress {
        extract
        filenames {
                "\.[zZ]$" regexp
        }
}
```

Figure 2: Sample classifier schema

data for training and then for performance testing. To find the closest centroid, the distance between document d and centroid c was alternatively measured with the above six common distance metrics (d_j, c_j means the j-th coefficient of the vector d or c respectively).[1]

Best and most reliable accuracy has been achieved using the cosine as similarity measure in our VSC. A promising alternative though is the asymmetric measure. It captures the inclusion relations between vectors, i.e., the more that properties of d are also present in c, the higher the similarity. Dice, Jaccard, and Overlap metrics give lower accuracy for our purposes. Surprisingly very low results have been achieved by Euclidian distance.

The bottom line of this evaluation is that the classifier's accuracy could not have been improved by choosing a different distance measure. In the following section we discuss a way of getting feedback about the classifier's confidence which can, in turn, be used to improve the accuracy of the classifier.

[1]The accuracy of a classifier is measured for a particular class C as [Jon71]

$$\text{recall}(C) = \frac{\text{objects of } C \text{ assigned to } C}{\text{total objects of } C}$$

$$\text{precision}(C) = \frac{\text{objects of } C \text{ assigned to } C}{\text{total objects assigned to } C}\ .$$

To measure a classifier as a whole, we use the arithmetic mean of recall or precision over all classes. Notice that every object is classified into exactly one class (no unclassified or double classified objects). The E-value [vR79]

$$\text{E-value} = \frac{2\ \text{precision recall}}{\text{precision} + \text{recall}}$$

is a single measure of classifier accuracy that combines and equally weights both, recall and precision.

3 The Confidence Measure

Independent of which similarity measure is chosen, closeness to a centroid is not a very useful indicator of the classifier's confidence in its result. Hence, we introduce the following novel measure that gives important feedback on how sure the classifier is about a result.

Definition. The *confidence* of an assignment of document d to class c_i is defined as

$$\text{confidence}(d, c_i) \stackrel{\text{def}}{=} \frac{\text{sim}(d, c_i) - \text{sim}(d, c_j)}{\text{sim}(d, c_i)}$$

with c_i the closest centroid and c_j the second closest centroid.

The confidence is the ratio of the similarity of the closest and second closest centroid over the similarity of the file and the closest centroid.[2] The following example illustrates how the confidence measure works.

Example 2: Consider two centroids c_1 and c_2, having both the same distance from a given document d, i.e. $\text{sim}(d, c_i) = \text{sim}(d, c_j)$. Classification as one or the other class is therefore completely arbitrary.

However, if these centroids are very close to the document, the similarity alone suggests a very good classification result, which is not correct. The true situation is reflected by the confidence, which gives a very low value, namely 0. ◇

The confidence measure can be used to tell whether the classifier probably misclassified a document. The

[2]The confidence measure can be generalized to take into account the n closest centroids. In this paper however, we use the closest and second closest centroids only.

Table 1: Alternative similarity metrics

distance metric	$\text{sim}(d, c)$	recall	precision	E-value
1. Cosine	$\frac{d \cdot c}{\|d\|\|c\|} = \cos\alpha$	0.97	0.97	0.97
2. Asymmetric	$\frac{\sum_j \min(d_j, c_j)}{\sum_j d_j}$	0.94	0.95	0.95
3. Dice	$\frac{2\,(d \cdot c)}{\sum_j d_j + \sum_j c_j}$	0.94	0.93	0.94
4. Jaccard	$\frac{d \cdot c}{\sum_j d_j + \sum_j c_j - \sum_j (d_j\ c_j)}$	0.94	0.93	0.94
5. Overlap	$\frac{d \cdot c}{\min(\sum_j d_j, \sum_j c_j)}$	0.93	0.90	0.91
6. Euclidian	$\sqrt{\sum_j (d_j - c_j)^2}$	0.69	0.87	0.77

higher the confidence value, the higher the classifier's certainty and therefore the higher the probability that the file is classified correctly.

Figure 3 shows the distribution of the confidence for a sample classifier. Each dot represents one of the ~2500 test files. The (logarithmic) x-axis shows the classifier's confidence in assigning a test file to a file type. The y-axis is separated into two areas, the lower one for correctly classified files and the upper one for incorrectly classified files. Both areas have one row for each of the 47 file types.

This distribution illustrates the tendency of correctly classified files to have a confidence around 0.7 and the incorrectly classified files around 0.07. One can make use of that to alert a human expert, that is, to apply the following algorithm: choose a *confidence threshold* Θ; classify document d, resulting in a class c with confidence γ; if $\gamma < \Theta$ then ask a human expert to approve the classification of document d as class c.

Figure 4 illustrates how much feedback can be derived from the confidence measure. Assumes a given confidence threshold Θ (vertical line), such that the user has to approve the classification if a file is classified with a smaller confidence.

The dotted curve shows the percentage of test files for which the assumption is true that they are classified correctly if classified with a confidence above threshold Θ and classified incorrectly otherwise. If, for example, the threshold Θ is set to 0.1, then about 94% are classified correctly if their confidence is above 0.1 and incorrectly otherwise (see dotted line hitting threshold).

The solid curve shows the percentage of test files that were classified with a confidence below Θ. With $\Theta = 0.1$, about 10% of the files are presented to the user for checking (see solid curve hitting the threshold). These were shown to a human expert. Note that about 5% of the files had a confidence of 0. These files were equidistant from 2 centroids indicating that the classifier had to make an arbitrary choice between them.

Finally, the dashed curve shows the percentage of test files that were classified correctly even though they have a confidence below threshold Θ. These are the files where the classifier "annoyed" the user for no good reason. With $\Theta = 0.1$, only 30% of the presented files were actually classified correctly (see dashed line hitting the threshold). Thus, using the confidence measure, a user had to touch 10% of all files, of which in fact 70% were classified incorrectly. The classifier's overall recall could therefore be improved by 7% without bothering the user too much.

In this classifier, $\Theta = 0.1$ provides a maximum accuracy (dotted line) while providing a reasonable number of files for the user's consideration while maintaining a modest "annoyance" level.

3.1 Classifier Training Strategies

The confidence measure's primary use is to detect misclassified documents. This not only improves the classifier's performance, but also proved to be useful for other purposes. In this section, we concentrate on using the confidence measure to speed up classifier training. Quick (re)training is an ability that is crucial for any classifier, especially for extensibility, as we will see later.

To train the classifier, a human expert has to provide a reasonable number of documents that are typical of each class. The first questions is: how much training data is required for it to perform well? Preliminary experiments showed that a surprisingly small set of training data produces a sufficiently accurate classifier. In Figure 5, the solid line shows the performance (E-value) of a classifier built with different sizes of training data.

For example, a classifier trained with only one document per class has average E-value of 0.89. The same

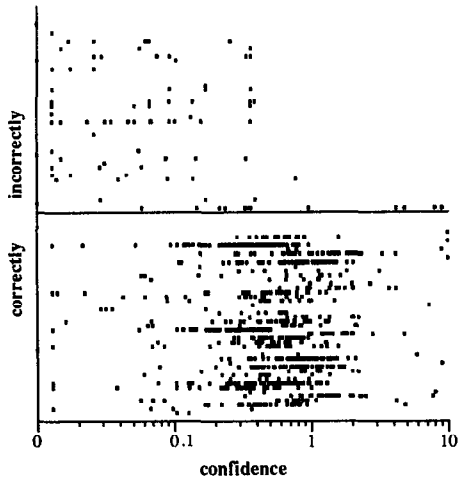

Figure 3: Distribution of the confidence measure

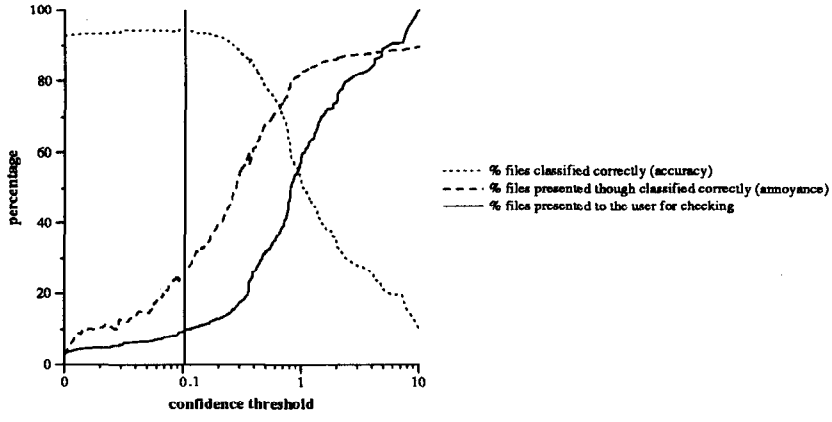

Figure 4: Feedback from the confidence measure

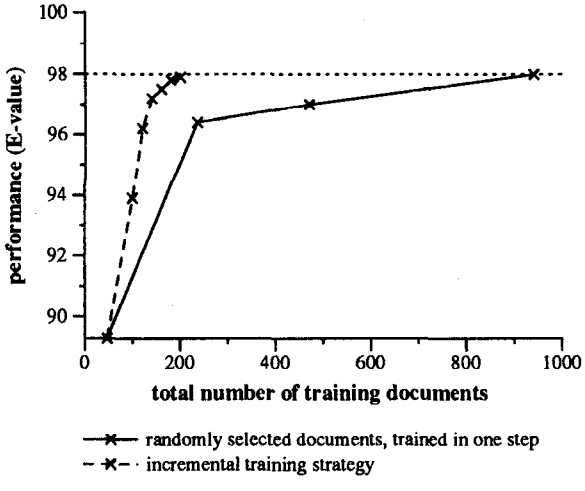

Figure 5: Number of training documents vs. classifier performance

classifier with 5 training documents has average E-value of about 0.96, with 10 documents about 0.97, and with 20 documents (~1000 total) nearly 0.98.

In the experiments discussed in the remainder of this paper, we use (unless stated otherwise) training data sets with an average of ~17 documents per class (total ~800 documents = ~26 MBytes). These data sets have randomly been selected as subsets of a large collection of training documents. On an IBM RISC System/6000 Model 530H, training the experimental file classifier takes about 50 seconds, with this amount of training data. For evaluating the classifier's performance randomly selected data is used that is always disjoint from the training data.

Though it shows that only little training data is required, the second question is: what are good training documents and how can they be found. One common way is to use an *incremental training strategy*, where the classifier is initially trained with few (one or two) documents of training data for each class. Then the classifier is run on unclassified test documents. A human expert manually classifies some of them and adds them to the training data. After about 20 documents have been added to the training data, the classifier is retrained with the extended training set.

The crucial parameter of this strategy is whether the correctly or the incorrectly classified documents should be added to the training data set. We actually used a third approach and added those documents to the training data for which the classifier was least confident about the classification, i.e., the confidence measure was below a given threshold. The final incremental training algorithm is illustrated in the following:

```
step 1:
   train an initial classifier with N0
            documents per class;
step 2:
   while the classifier's performance is
            insufficient
   and a user is willing to classify
      documents do
      classify document using current
         classifier;
      if confidence was below a certain threshold
         then
         classify document by user and
         add it to training data set;
      if N1 training documents have
         been added then
         retrain the classifier with new
         training set;
   end
```

Incremental training is very efficient when adding the least confident documents to the training data set. Consider again Figure 5: the dashed line shows the classifier's performance using the incremental training strategy, as opposed to training the classifier with randomly selected data, all at once (solid line). An initial classifiers was built with $N_0 = 2$ training documents per class (~100 documents in total), which resulted in an E-value of about 0.94. In five iterations, $N_1 = 20$ documents per iteration were incrementally added to the training data.

To achieve a classifier of E-value 0.96, one iteration was necessary. Notice, that at this point of time, only a total of 120 training documents were used, compared to 250 needed documents if training with random data in one step. After five iterations we already achieved 0.98 and used only 200 training documents, compared to 1,000 if trained with random data in one step (cf. Figure 5).

The incremental training algorithm is similar to the uncertainty strategy proposed by [LG94]. However, the number of files needed by their strategy is significantly larger than ours (up to 100,000 documents), because they are doing semantic full textual analysis of all the words in the documents. In contrast, we look for a few syntactic patterns and can get enough randomness in 10 files.

4 Feature Selection

Finding good features is crucial for a classifier's performance. However, it is a difficult task that can not be automated.

On one hand, features must identify one specific class and should apply as little as possible to other classes. This is easy for classes that can be identified by examining files for matching string literals, like e.g., FrameMaker documents, or GIF pictures. But it is difficult for classes that are very similar, like different kinds of electronic mail formats, e.g. RFC822 mail, Usenet messages, MBox folders. It may also be a problem for textual files containing mainly natural language and having only few commonalities.

On the other hand, there must be enough features to identify all kinds of files of a particular class. This causes a problem, if classes can only be described by very general patterns or can take alternative forms, like for instance word processors having different file saving formats. In these cases, it is advisable to either define completely different classes or to combine features together.

In this section, we present techniques to analyze and improve the schema of a classifier. These techniques help a human expert choose good features. To reveal

the results in advance, we managed to improve a classifier's performance from an average E-value of 0.86 to 0.94, just by optimizing the schema.

4.1 Distinguishing Power

The most important property of features is how precise they identify one particular class. Thus, good features can be separated from bad features in how distinguishing they are, i.e., the number of classes they match.

We use the variation of feature coefficients over all centroids to measure how distinguishing features are. Consider vector $f_i = (a_{i1}, \ldots, a_{in})$, where a_{ij} is the coefficient of feature f_i in centroid c_j ($1 \leq i \leq m, 1 \leq j \leq n$). This vector represents the feature's distribution over classes. Assuming normal distribution, we define:

Definition. The *distinguishing power* of feature f_i is defined as

$$\text{dist-power}(f_i) \stackrel{\text{def}}{=} \frac{s^2}{\overline{a}}$$

where $s^2 = \frac{1}{n-1}\sum_{j=1}^{n}(a_{ij} - \overline{a})^2$ is the variance and $\overline{a} = \frac{1}{n}\sum_{j=1}^{n} a_{ij}$ is the mean.

This definition of distinguishing power values both, low variance and low mean. It ranges from 0 to 1. For an optimal feature that has all coefficients $a_{ij} = 0$ except for one that is 1, the variance s^2 is equal to its mean $\overline{a}$. Hence, the distinguishing power of a perfect feature is 1. For a worst-case feature that has a uniform distribution over all classes (and $\overline{a} \neq 0$), the variance, and therefore the distinguishing power, is 0. The higher dist-power(f_i) is, the more distinguishing is feature f_i.

Example 3: Figure 6 illustrates distinguishing power for two sample features. Feature "`Shellscript__set_`" is defined for class SHELLSCRIPT and searches for string literal " `set` ". This feature matches many different classes to a low degree, which is reflected in a very low distinguishing power (0.1978).

Feature "`RFC822_^From:`" is defined for class RFC822 (an e-mail format) and searches for lines beginning with "`From:`". This feature has a much better distinguishing power (0.7742). It selects fewer classes, most of them to a high degree. Notice that this feature now identifies a group of four classes that are similar (e-mail like).

An example of a perfect feature (distinguishing power 1.0) is feature "`CHeader_.h$`" (not shown in Figure 6), a regular expression looking for file names ending with ".`h`". Its coefficients are 1 for class CHEADER and 0 for all others. ◇

In general, feature analysis can be performed in two different ways. These approaches are complementary:

- the analyzer scans human generated features and identifies those with poor distinguishing power;
- the analyzer scans all training documents and proposes features with high distinguishing power.

A human expert is necessary in both cases. Ultimately, the expert must decide whether to include a proposed feature into a schema, change an existing feature's definition in order to make it more specific, delete a feature, or keep it as it is. It is difficult to automate this task. Some features must be included although they are not very distinguishing, for instance, those that are the only feature of a top-level class in the hierarchy (TEXT, BINARY, DIRECTORY, SYMLINK). On the other hand, regular expression patterns, for example, may contain an error that cannot be detected and corrected automatically.

To illustrate feature analysis, the experimental file VSC was built using a non-optimized schema with about 200 features, created by a user with moderate experience in using the classifier. This classifier had an average E-value of 0.86.

Subsequent feature analysis showed that only about 15% of these features identified exactly one type (dist-power = 1), 10% did not match any type at all, and more than 50% had dist-power < 0.5. Based on this feature analysis, the schema was optimized. Patterns were changed to make features more specific and syntax errors that caused features to fail to identify any class were corrected. A new classifier was built with this improved schema. The average E-value increased to 0.94, just from using the optimized schema.

4.2 Combining Features

Some file types have the property that documents of these classes match a highly varying number of features (e.g. SCRIPT, TROFFME, YACC, CSOURCE). Some documents match 20 to 30 features, whereas others only 1 or 2. Even if the 1 or 2 features are a subset of the 20 to 30 features, the classifier performs poorly for these classes, because it can only be trained to properly recognize one of the two styles of documents.

One approach would be to define two different classes to cover the two styles. However, it proved to be extremely difficult to define the schemas for the two separated classes and to separate the training data.

A better solution is combining several features $f_1, \ldots, f_m$ into one feature. The new feature is built as a regular expression $f = f_1|\ldots|f_n$, connecting the

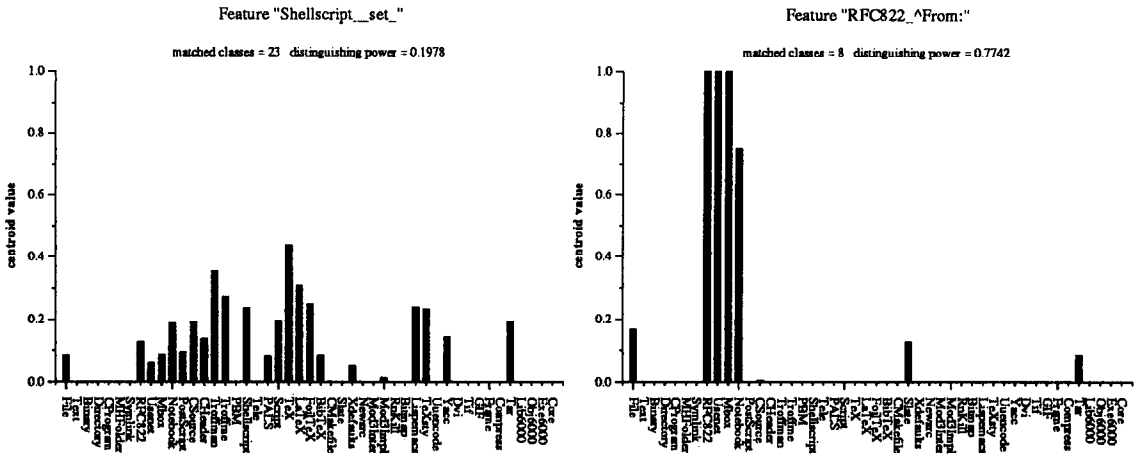

Figure 6: Distinguishing power of features

original features via "or" patterns. There are two ways to combine the features of a given class together:

- *Combining "disjoint" features.* The first way is to combine "disjoint" features that never (less than a given amount of the time) appear together. Consider as an example file types with two different initialization commands where only one of which appears at the beginning of the file.

- *Combining "duplicate" features.* The second way is to combine "duplicate" features, that is, features that always (more than a given amount of the time) appear together, but do not appear often (in more than a given amount of the files). For example, patterns "`argc`" and "`argv`" in CSOURCE. The second limitation allows the classifier to keep the really good features like "`^Received`" and "`^From`" which appear in all RFC822 files, but it will combine "`argc`" and "`argv`" which only sometimes occur in CSOURCE files.

The algorithm for combining features looks as follows (choosing 80% as the threshold to combine features and 60% for the number of files duplicate features should not appear in has given the best results):

foreach class c_i $(i = 1 \ldots n)$ of the schema **do**
step 1:
  m = number of features of class c_i;
  F = features $\{f_1, \ldots, f_m\}$ of class c_i;
  $P(F)$ = the powerset of F,
    without the empty set;
step 2:
  train the classifier:
  scan all training documents of class c_i
    for feature occurrences;
  foreach $s \in P(F)$ **do**
    $occ(s)$ = percentage of training
      documents of class c_i
      where features s occur together;
  end
step 3:
  find features to be combined:
  while $m > 1$ **do**
    foreach $s \in P(F)$ with m features **do**
      if $(\text{avg}_{f \in s} occ(s)/occ(f) < 0.20)$
      or $((\text{avg}_{f \in s} occ(s)/occ(f) > 0.80)$
      and $occ(s) < 0.60)$
      then
        combine features in s;
        remove all sets from $P(F)$
        containing any of the
        features in s;
    end
    $m - -$;
  end
end

The algorithm works class by class and combines only features that are defined within the same class, that is, features from different classes are never combined together.[3]

In step 1, the algorithm computes $P(F)$ as the set of all possible subsets of features $\{f_1, \ldots, f_m\}$ for the current class c_i. In step 2, the classifier is trained by classifying a large number of documents (~40 - 50 per class). While scanning training documents, the algorithm remembers for each of the feature combinations $s \in P(F)$ the percentage of documents in which this combination occurred. In step 3, the algorithm searches features to be combined. It tries to combine as many features as possible and starts therefore with the largest feature combination having all m features. If the combination fulfills one of the "disjoint" or "du-

[3] In the current experimental classifier, feature combination runs on `restpats` features only.

plicate" occurrence conditions, all features of the set s are removed from the schema and replaced by one new feature that combines them as described above. All sets are removed from $P(F)$ containing any of the features from s, because maximum combination was already achieved for these features. If all feature combinations of this size m have been processed, m is decremented and the algorithm tries to combine the remaining feature combinations of the smaller size.

Running feature combination on the `restpats` of the already optimized schema of the previous section, combined 37 disjoint features into 14 new features and 58 duplicate features into 20 new ones. Just by automatic feature combination, the classifier's performance has been improved from an overall E-value of 0.94 to now 0.96. In detail, the recall of file type TROFFME has been increased by 9.1% combining 7 into 2 features, YACC by 7.1% combining 2 into 1 feature and SCRIPT by 6.2% combining 38 into 13 features.

5 Comparison of other Classifier Technologies

There are diverse technologies for building classifiers. *Decision tables* are very simple classifiers that determine according to table entries what class to assign to an object. The UNIX "file" command is an example of a decision table based file classifier. It scans the `/etc/magic` file, the decision table, for matching values and returns a corresponding string that describes the file type. *Decision tree classifiers* construct a tree from training data, where leaves indicate classes and nodes indicate some tests to be carried out. CART [BFOS84] or C4.5 [Qui93] are well known examples of generic decision tree classifiers. *Rule based classifiers* create rules from training data. R-MINI [Hon94] is an example of a rule based classifier that generates disjunctive normal form (DNF) rules. Both, decision tree and rule classifiers, usually apply pruning heuristics to keep these trees/rules in some minimal form. *Discriminant analysis* (linear or quadratic) is a well known basic statistical classification approach [Jam85].

To rate our experimental vector space classifier, we built alternative file classifiers using quadratic discriminant analysis, decision tables, the decision tree system C4.5, and the rule generation approach R-MINI. Table 2 summarizes the conducted experiments. The intent of this table is to give a rough overview of how the different techniques compare on the file classification problem and not to present detailed performance numbers ("+" means an advantage and "-" means a disadvantage of a particular classifier technology).

Speed. Training and classification using quadratic discriminant analysis is very slow because extensive computations must be performed. All other classifier technologies provide fast training and classification. The vector space classifier simply needs to compute angles between the document vector and all centroids. For example, on an IBM RISC System/6000 model 530H, an individual document is classified by the experimental classifier in about 40 milliseconds, on average. The other classifier technologies (except of discriminant analysis) proved a similar speed.

Accuracy. Quadratic discriminant analysis and decision tables did not achieve our accuracy requirements. They had error rates up to 30%. All other classifier technologies proved much lower error rates. The C4.5 file classifier showed error rates from 2.6 to 5.0% misclassified files. The R-MINI file classifier showed error rates from 2.6 to 4.9%. The vector space classifier had 2.1 to 3.1% error rates. Hence, all three technologies have approximately the same range of errors.

Extensibility. A classifier is usually trained with a basic set of general classes. However, this basic class hierarchy must be extensible. Users want to define and add specific classes according to their personal purposes. *Extensibility* of a classifier is therefore crucial for many applications. In order to add classes to a classifier, a user must provide class descriptions (schema) and training data for the new classes. Training documents of the existing classes must be available too. This "old" training data is necessary because new classes must be trained with data for existing classes as well.

Vector space classifiers are highly suited for extensibility purposes. Consider as an example a classifier with existing classes $c_1, \ldots, c_g$ and existing features $f_1, \ldots, f_j$. Assume this classifier is extended with new classes $c_h, \ldots, c_n$ $(h = g + 1)$ and new features $f_k, \ldots, f_m$ $(k = j + 1)$. After extension, the *feature-centroid matrix* A of the classifier, where each coefficient a_{xy} shows the value of feature y in the centroid of class x, looks as follows:

$$A = \begin{array}{c} \begin{array}{cccccc} \boldsymbol{f_1} & \cdots & \boldsymbol{f_j} & \boldsymbol{f_k} & \cdots & \boldsymbol{f_m} \end{array} \\ \begin{bmatrix} a_{11} & \cdots & a_{1j} & a_{1k} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{g1} & \cdots & a_{gj} & a_{gk} & \cdots & a_{gm} \\ a_{h1} & \cdots & a_{hj} & a_{hk} & \cdots & a_{hm} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nj} & a_{nk} & \cdots & a_{nm} \end{bmatrix} \end{array} \begin{array}{c} \\ \boldsymbol{v_{c_1}} \\ \vdots \\ \boldsymbol{v_{c_g}} \\ \boldsymbol{v_{c_h}} \\ \vdots \\ \boldsymbol{v_{c_n}} \end{array}$$

The upper-left (dashed) sub-matrix of A shows the existing feature centroid matrix. To add new classes, these existing centroid coefficients need not be recomputed. The feature-centroid matrix can incrementally be extended with coefficients for newly added classes.

Table 2: Different Classifier Technologies

	Quad. Discr. Analysis	Decision Tables	Decision Trees (C4.5)	DNF Rules (R-MINI)	Vector Space Model
Speed	-	+	+	+	+
Accuracy	-	-	+	+	+
Extensibility	-	-	-	-	+

The lack of extensibility of discriminant analysis, decision table/tree and rule classifiers is the most dramatic difference. In contrast to vector space classifiers, extending this kind of classifiers with new user-specific classes demands rebuilding the whole system (tables, trees, or rules) from scratch, that is, it requires complete reconstruction of the classifier. Incremental, additive extension is not possible.

6 Conclusion and Outlook

High accuracy, fast classification, and incremental extensibility are the primary criteria for any classifier. The experimental VSC for assigning types to files presented in this paper fulfills all three requirements.

We evaluated different similarity metrics and showed that the cosine measure gives best results. A novel confidence measure was introduced that detects probably misclassified documents. Based on this confidence measure, an incremental training strategy was presented that significantly decreases the number of documents required for training, and therefore, increases speed and flexibility. The notion of distinguishing power of features was formalized and an algorithm for automatic combining disjoint and duplicate features was presented. Both techniques increase the classifier's accuracy again. Finally, we compared the VSC with other classifier technologies. It revealed that using the vector space model gives highly accurate and fast classifiers while it provides at the same time extensibility with user-specific classes.

The file classifier can be seen as a component of object. text, and image database management systems. There is recently an increasing interest in merging the functionality of database and file systems. Several proposals have been made, showing how files can benefit from object-oriented technology.

Christophides et al. [CACS94] describe a mapping from SGML documents into an object-oriented database and show how SGML documents can benefit from database support. Their work is restricted to this particular document type. It would be interesting to see how easily it can be extended to a rich diversity of types by using our classifier.

Consens and Milo [CM94] transform files into a database in order to be able to optimize queries on those files. Their work focuses on indexing and optimizing. They assume that files are already typed before reading, for example, by the use of a classifier.

Hardy and Schwartz [HS93] are using a UNIX file classifier in Essence, a resource discovery system based on semantic file indexing. Their classifier determines file types by exploiting naming conventions, data, and common structures in files. However, the Essence classifier is decision table based (similar to the UNIX "file" command) and is therefore much less flexible and tolerant.

The file classifier can also provide useful services in a next-generation operating system environment. Consider for instance a file system backup procedure that uses the classifier to select file-type-specific backup policies or compression/encryption methods.

Experiments have been conducted using the classifier for language and subject classification. Whereas language classification showed encouraging results, this technology has its limitations for subject classification. The reason is that the classifier works mainly by syntactical exploration of the schema, but subject classification must take into account the semantics of a document.

We are currently working on making the classifier extensible even without the requirement of training data for existing classes. We are also investigating the classification of structurally nested documents. A file classifier is being developed that is, for example, able to recognize Postscript pictures in electronic mail or C language source code in natural text documents. Use of this classifier to recognize, and take advantage, of a class hierarchy is an item for future work.

References

[BFOS84] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Wadsworth, Belmont, CA, 1984.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In SIGMOD94 [SIG94b].

to novel query facilities. In SIGMOD94 [SIG94b].

[CM94] M.P. Consens and T. Milo. Optimizing queries on files. In SIGMOD94 [SIG94b].

[GRW84] A. Griffiths, L.A. Robinson, and P. Willett. Hierarchic agglomerative clustering methods for automatic document classification. *Journal of Documentation*, 40(3), September 1984.

[Hoc94] R. Hoch. Using IR techniques for text classification. In SIGIR94 [SIG94a].

[Hon94] S.J. Hong. R-MINI: A heuristic algorithm for generating minimal rules from examples. In *Proc. of PRICAI-94*, August 1994.

[HS93] D.R. Hardy and M.F. Schwartz. Essence: A resource discovery system based on semantic file indexing. In *Proc. USENIX Winter Conf.*, San Diego, CA, January 1993.

[Jam85] M. James. *Classification Algorithms.* John Wiley & Sons, New York, 1985.

[Jon71] K. S. Jones. *Automatic Keyword Classification for Information Retrieval.* Archon Books, London, 1971.

[LG94] D.D. Lewis and W.A. Gale. A sequential algorithm for training text classifiers. In SIGIR94 [SIG94a].

[ODL93] K. Obraczka, P.B. Danzig, and S.-H. Li. Internet resource discovery services. *IEEE Computer*, 26(9), September 1993.

[Qui93] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufman, San Mateo, CA, 1993.

[Sal89] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, 1989.

[Sch93] P. Schäuble. SPIDER: A multiuser information retrieval system for semistructured and dynamic data. In *Proc. 16th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, Pittsburg, PA, June 1993. ACM Press.

[SIG94a] *Proc. 17th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, Dublin, Ireland, July 1994. Springer.

[SIG94b] *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Minneapolis, Minnesota, May 1994. ACM Press.

[SLS+93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The Rufus system: Information organization for semi-structured data. In *Proc. 19th Int'l Conf. on Very Large Data Bases (VLDB)*, Dublin, Irland, August 1993.

[SWY75] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11), November 1975.

[vR79] C.J. van Rijsbergen. *Information Retrieval.* Butterworths, London, 1979.

[YMP89] C.T. Yu, W. Meng, and S. Park. A framework for effective retrieval. *ACM Trans. on Database Systems*, 14(2), June 1989.