Chapter 14

HIERARCHICAL TEXT CLASSIFICATION METHODS AND THEIR SPECIFICATION

Aixin Sun   Ee-Peng Lim   Wee-Keong Ng

School of Computer Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 639798

Abstract Hierarchical text classification refers to assigning text documents to the categories in a given category tree based on their content. With large number of categories organized as a tree, hierarchical text classification helps users to find information more quickly and accurately. Nevertheless, hierarchical text classification methods in the past have often been constructed in a proprietary manner. The construction steps often involve human efforts and are not completely automated. In this chapter, we therefore propose a specification language known as HCL (Hierarchical Classification Language). HCL is designed to describe a hierarchical classification method including the definition of a category tree and training of classifiers associated with the categories. Using HCL, a hierarchical classification method can be materialized easily with the help of a method generator system.

Keywords: Hierarchical Text Classification, Specification Language

1. Introduction

1.1 Motivation

Text classification is a research area that develops methods for assigning text documents to a pre-defined set of categories [6, 8]. When the given categories are defined independently of one another, this is known as *flat classification*. Most of the studies in text classification have focused on flat classification. In most flat classification methods, either a *binary classifier* is assigned to each category to determine if a given document belongs to the category, or a *m-ary classifier* is assigned to a group of categories to determine if a given document

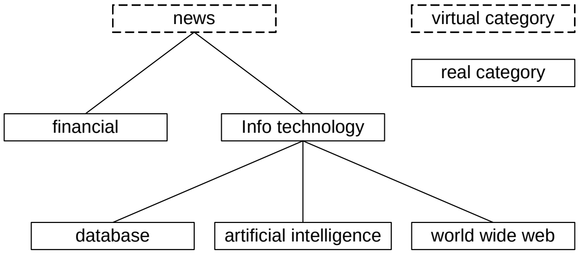

Figure 1.1. A sample category tree

belongs to zero or more categories in the group. After many years of research, flat classification has become a well-established research area and many good classifiers have been developed. A good survey of the text classification approaches is given in [6].

More recently, increasing attention has been given to *hierarchical classification* where the pre-defined categories are organized in a tree-like structure. A category tree example is shown in Figure 1. In a category tree, there are parent-child relationships between categories. These parent-child relationships may suggest *strong* or *weak subsumption constraint* between categories. A parent and child category pair with strong subsumption constraint suggests all documents belonging to the child also belong to the parent. Weak subsumption, on the other hand, allows a child category to have documents not belonging to its parent category. By organizing a large number of categories in a tree, hierarchical classification allows us to address a large classification problem using a divide-and-conquer approach, also known as the top-down approach [7]. At the root level, a text document can be first classified into one or more child categories. The document can then be further classified at each child category to determine if it belongs to categories at the next lower level. The classification step can be repeated until the document cannot be further classified into any lower-level categories. While in flat classification a given document is assigned to a category based on the outcome of one or one set of classifiers, the assignment of document to the category can be the outcome of multiple sets of classifiers in hierarchical classification. These classifiers are associated to different levels of the category tree to filter away documents that do not belong to the lower level categories.

While the applications of text classification (flat and hierarchical) are wide and diverse, there has not been a specification language for defining classification methods. Most of the existing text classification systems have been developed either manually by information retrieval (IR) experts who constructed each classifier by providing appropriate training data and input parameters, or by specialized scripts that are customized for the particular classification systems. The manual approach clearly incurs much overhead in time. The latter script approach takes less time in the actual construction of the classification systems but the scripts themselves still require time for development. Since such scripts are usually proprietary and not necessarily easy to use, they are not really suitable to be used directly the application developers and end users.

1.2 Objectives

To promote wider adoption and acceptance of text classification methods, we believe that a declarative specification language for defining text classification methods will be important and useful. Such a specification language, once standardized across all commercial text classification products, will facilitate users to directly create their text classification systems, to build layers of applications that use the specification language to define their text classification modules, and to exchange classification methods and results. In particular, the specification language can hide differences among the heterogeneous classifier packages used to implement a classification method. This greatly reduces the efforts of developing and maintaining the classification method, and this in turn translates to savings in the development and maintenance costs of the application using the classification method. From the performance experiments perspective, a specification language will also allow users to compose variations of classification methods for performance evaluation and tuning.

To design a specification language for text classification, we need to identify the essential primitives of the language. We focus on hierarchical classification in this chapter since it is more appropriate in practical applications. The language primitives to be designed must therefore be able to support the definition of the category tree structure, the association of classifiers to the categories, and the training and classification steps. Our proposed specification language, HCL (Hierarchical Classification Language), attempts to provide the above primitives. As flat classification problem can be treated as hierarchical classification problem on a one-level category tree where the root of the tree is virtual, we believe HCL is able to handle most text classification problems. In this chapter, we will overview the existing flat and hierarchical classification methods and illustrate HCL using a few examples. While the design of HCL is still tentative, we hope that it will serve to motivate further work in this area.

1.3 Outline of Paper

The rest of the chapter is organized as follows. We first examined the related work in Section 2. The proposed classification method generation system is described in Section 3. Our proposed classification language HCL is given in Section 4. Section 5 concludes our discussion.

2. Related Work

Recently, hierarchical classification has gained much attention in the IR research community due to its practical usage. Koller and Sahami used multiple Bayesian classifiers to classify the Reuter's collection into some pre-defined categories [3]. The categories were arranged in two-level category trees. Dumais and Chen proposed the use of Support Vector Machine (SVM) classifiers to classify web pages into a category tree using a top-down approach [1]. Nevertheless, their method allows a web page to be assigned to a child category even if the former is not favored by the parent category. The category structure used is a 2-level category tree with a virtual root category obtained from the LookSmart's web directory (http://www.looksmart.com). We have not seen any work on the specification language for text classification. In this section, we therefore describe some script languages specially designed for text classification, and also survey some other existing data mining languages.

The RAINBOW script language is part of the BOW library developed by the CMU Information Retrieval group [4]. Designed to facilitate the definition of classifiers for mainly performance evaluation purposes, the Rainbow language is very cryptic and not very readable. It also does not support the definition of working relationships between classifiers required in the hierarchical classification methods.

Text classification can be considered as a special kind of data mining. In relational databases, data mining often refers to association rule mining, classification of database records, etc.. DMQL is a data mining query language developed by Han et al. to allow a user to specify the mining parameters and the type of knowledge to be mined from a given relational database [2]. Meo et al developed another SQL-like data mining language for specifying how association rules can be discovered from a relational database [5]. Nevertheless, the above two data mining languages are mainly for relational databases where data are stored as structured records.

3. Classification Method Generation System

To adopt the HCL specification language, a classification method generation system must be provided to take a classification method specification in HCL and transform it into a working classification method. Figure 2 depicts the components of our proposed classification method generation system.

Given a HCL classification method specification, a *classification language compiler* compiles it into an *untrained classification method file* which contains the internal representation of the category tree structure and the association of classifiers to the category tree. The classification method is untrained since the classifiers are yet to be constructed.

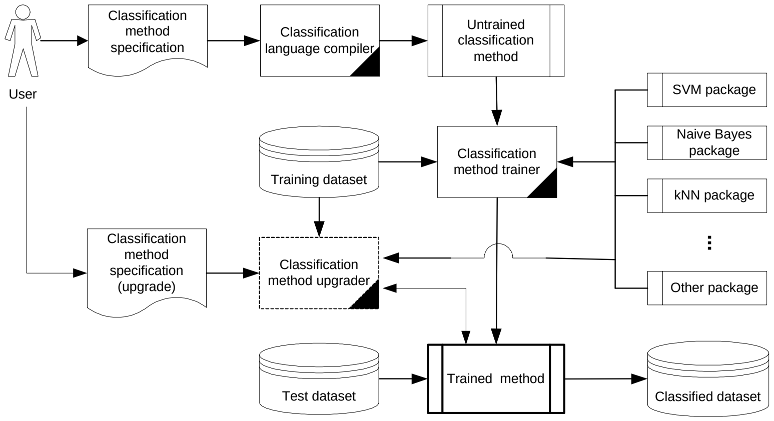
User
Classification method specification
Classification language compiler
Untrained classification method
SVM package
Naive Bayes package
kNN package
Other package
Training dataset
Classification method trainer
Classification method specification (upgrade)
Classification method upgrader
Test dataset
Trained method
Classified dataset

Figure 1.2. Classification method generation system

The *classification method trainer* takes the untrained classification method and starts feeding training data to the classifier packages. Here, each classifier package is required to provide a training interface which allows a new executable classifier to be constructed when a training set is given. Note that the training data may include both positive and negative training data for binary classifiers, and training data of different classes for m-ary classifiers. Since the method is designed for hierarchical classification, the trainer must incorporate into the method classification steps in which the invocations of classifiers associated with the higher-level categories come before those associated with the lower level categories. This ensures that documents will be filtered by the classifiers associated with the higher level categories first before they are examined by those associated with the lower level categories. The order of applying classifiers will be further illustrated in Section 4.5.

The trainer assembles the constructed classifiers and incorporates them into a top-down hierarchical classification algorithm before generating the *trained classification method* as an executable program.

A trained classification method can now be directly applied on a given document collection. During the classification phase, the method will call the different classifiers to classify and assign appropriate category labels to the documents. Since the classification method may not be perfect and there may be changes to the category tree or documents to be classified, one would expect further upgrading to be done on the classification method when new training data or new category tree are available. As opposed to a full-fledged retraining of the classification method, the *classification method upgrader* is able to take care of such incremental changes.

Table 1.1. Extended BNF symbols

Symbol	Meaning
|	Separators for alternative symbols
[ ]	Symbol(s) enclosed is optional
{ }	Symbol(s) enclosed is used zero or more times
()	Enclose groups of alternative symbols

In this chapter, we will only focus on classification specification language constructs that are mainly used in the classification language compiler and classification method trainer. The language constructs for supporting method upgrading will be part of our future work since there has not been much research work done in upgrading hierarchical classification methods.

4. Classification Specification Language

Our proposed classification language consists five components, namely, *document modeling*, *category tree modeling*, *classifier construction*, *classifier training*, and *document assignment*. Throughout this section, we express the syntax of the classification specification language using the *Backus Naur Form* (BNF) notation. To improve readability, terminal and non-terminal symbols are shown in capital and `Courier` respectively. The single character terminals are enclosed by quotation marks, e.g. ";". The other extended symbols used in our language syntax are listed in Table 1.

4.1 Document Modeling

In document classification, we deal with documents. There are two kinds of documents, i.e., training documents and documents to be classified. Since HCL is about the construction of classification methods, we only focus on modeling training documents. Regardless of its kind, HCL supports the notion of document variable which is of *Document* class. As shown in Table 2, a document class consists of a unique `id`, `date`, `length`, a labeled category set (`labeledCat`), and an assigned category set (`assignedCat`).

We assume that every document must belong to some document pool. The document `id`, an integer value, is unique within a document pool. Other than `id`, each document has a creation `date`, and `length`. For a training document, the set of labeled categories is represented by the `labeledCat` attribute. In the training processes, a training document can be assigned a set of categories by the classification method and is represented by the `assignedCat` attribute. Furthermore, each document may be assigned a score value by a classifier with respect to a set of categories. This is represented by the `score (classifier, category)` attribute. In HCL, we may use a document variable, say `docVar`,

Table 1.2. Attributes of document class

Attribute	Description
ID	document id
date	time when the document is created
length	length of the document
labeledCat	set of categories the document belongs to
assignedCat	set of categories assigned by the classification method
score(classifier , category)	score assigned by classifier with respect to category where category must be in the domain of classifier (see Section 4.3)

Table 1.3. Attributes of category class

Attribute	Description
description	description of the category
type	real if the category can hold documents, virtual otherwise
parent	the parent of the given category and null for root category
children	returns a set of categories containing all the child categories of the given category and null for leaf category
cover	returns a set of categories within the subtree rooted at the given category

to represent a document in a document pool. We represent the attribute, say attrib, of the document variable by docVar.attrib.

HCL also allows a training document pool to be imported using the define train pool statement. These document pool statement specifies where the classification method should obtain their training documents and the documents to be classified. The syntax of the define train pool statement is given below.

```
define train Pool ::= DEFINE TRAIN POOL docPoolName
docPoolPath";"
```

In the following example statements, we define myTrgPool as a training pool. The data file train.dat is formatted to store the appropriate document attributes and references to the document files.

```
DEFINE TRAIN POOL myTrgPool /htc/train.dat;
```

4.2 Category Structure Definitions

Similar to document, HCL provides a category class to model categories. The category attributes include description, type, parent, children, and cover. Given a category, the description attribute provides a user readable description about the category. The type of a category can be real or virtual.

A category is real when documents can be assigned to it, and virtual otherwise. The `parent` attribute refers to the parent category of the category. The `children` are a set of categories that are directly under the category. The `cover` attribute refers to the set of categories within the subtree rooted at the category. These attributes are depicted in Table 3.

To define a category tree consisting of a set of categories, we use the `define category tree` statement with the following syntax.

```
define category tree ::= DEFINE CATEGORY TREE
categoryTreeName WHERE (VIRTUAL | REAL) rootCatName
"(" description ")" IS ROOT "," define category  ";"

define category ::= (VIRTUAL | REAL) categoryName "("
description ")" IS CHILD OF categoryName
```

Every category must be defined within a category tree and for any category tree a root category must be defined. For example, to define the category tree in Figure 1, the following `define category tree` statement is used. The statement defines `news` as the category name of the root category and it is virtual. The other categories, `fin`, `it`, `db`, `ai`, and `web` are real and their parent-child relationships are expressed by `IS CHILD OF`.

```
DEFINE CATEGORY TREE myTree WHERE VIRTUAL news ("news") IS
ROOT, REAL fin ("financial") IS CHILD OF news, REAL it
("information technology") IS CHILD OF news, REAL db
("databases") IS CHILD OF it, REAL ai ("artificial
intelligence") IS CHILD OF it, REAL web ("world wide web")
IS CHILD OF IT;
```

Given a category c, we can access its category attribute, attrib, using the "." (dot) notation, i.e., `c.attrib`. For example, the cover of `it` category is written as `it.cover` and according to the above category tree, `it.cover` is a 5 set of categories, {`it, db, ai, web`}. HCL can also model a set of categories by supporting the container class, `categorySet`. Let c be a category and $C1$ and $C2$ be two `categorySets`. The following are the set operations and logical expressions that can be used in HCL.

- $C1$ union $C2$ returns the union of $C1$ and $C2$.
- $C1$ intersect $C2$ returns the intersection of $C1$ and $C2$.
- $C1$ minus $C2$ returns $C1$ - $C2$.
- c in $C1$ is true if and only if c exists in $C1$.
- $C1$ is subset of $C2$ is true if all the $C1$ categories are found in $C2$.

When multiple category trees are defined, HCL allows a category name to be prefixed by its category tree name to distinguish it from other categories in other category trees, e.g. `myTree.fin`. By supporting multiple category trees, HCL allows different sets of criteria to be used in classifying a set of documents and each set of criteria results in a different category tree. Note that the category tree definition can also be used for flat classification by having all categories defined as the children of a dummy virtual root category.

4.3 Construction of Classifiers

As shown in Figure 2, the construction of a classification method requires the creation of *classifier instances* using the given classifier packages. Examples of such classifier packages are *SVM* (Support Vector Machine), *Naïve Bayes*,*k-NN*, and *Rocchio* (see [6] for more details). Each classifier instance may require different set of parameter values. HCL is designed to allow classifier instances of different types to be integrated together to support hierarchical classification.

HCL considers both binary and m-ary classifiers. A binary classifier is usually associated with a category and it generates only one score value given a document. An m-ary classifier, on the other hand, is associated with a set of categories, and is able to generate multiple score values for a given document, one for each category. Here, we assume that each classifier is able to generate a score value as a real number. This assumption generally holds for most of the classifier packages. HCL represents the classifier package information, and the associated category as the *engine* and *domain* attributes of the classifier instance respectively. The latter refers to the category(ies) to which the classifier will assign score(s) to. For a binary classifier, the domain is a single category. For an m-ary classifier, the domain is a set of categories.

The `construct_classifier` statement is provided to define classifier instances. Note that the *setting* of a classifier is optional. If not specified, the default parameter values are used for the classifier.

```
construct classifier::= CONSTRUCT CLASSIFIER classifierName
WHERE ENGINE = engineName "," TYPE = classifierType
"," DOMAIN =( category | "" categorySet "" [ "," SETTING =
parameterSetting]";"

categorySet ::= category "," category | categorySet
```

For the category tree shown in Figure 1, three classifiers can be defined as follows.

```
CONSTRUCT CLASSIFIER svmLocal WHERE ENGINE = SVM,
TYPE = binary, DOMAIN = fin, SETTING = -j 2.0;
```

```
CONSTRUCT CLASSIFIER svmSubtree WHERE ENGINE= SVM,
TYPE= binary, DOMAIN= it;

CONSTRUCT CLASSIFIER nbLocal WHERE ENGINE= NaïveBayes, TYPE
=m-ary, DOMAIN= it,db,ai,web;
```

In the above statements, `svmLocal` is a binary SVM classifier associated with the `fin` category. The `svmSubtree` classifier is associated with the it category. During classification, a score value will be assigned to each document by the `svmLocal` classifier for the `fin` category, but not the other categories. The string "`-j 2.0`" serves as the parameter value for invoking the SVM classifier package. Given a document d, the score value assigned is represented by `d.score(svmLocal,fin)`. The same applies to the `svmSubtree` classifier for the `it` category.

The classifier, `nbLocal`, is a m-ary Naïve Bayes classifier associated with the categories `it`, `db, ai,` and `web`. In other words, `nbLocal` can assign a score to a given document, say d, for each of the above four categories. The scores are represented by `d.score(nbLocal,it)`, `d.score(nbLocal,db)`, `d.score(nbLocal,ai)`, and `d.score (nbLocal,web)`.

Although the above `construct classifier` statements define the required classifier instances, they do not provide further instructions on how these classifier instances will be trained. Neither do they specify how the score values assigned by classifier instances are used in the actual assignment of categories to documents. These two requirements will be discussed in the following two subsections.

4.4 Training of Classifiers

In HCL, the train statement (`train stmt`) specifies how the classifier instances can be trained. The focus of the train statement is the selection of the training documents rather than how the training documents are handled by the classifiers. In other words, HCL does not determine what document features will be extracted and selected by the different classifier packages for the purpose of training. Below is the syntax of the train statement.

```
train stmt ::= TRAIN classifierName USING docVar AS
POSITIVE FROM docPoolSet ["," docVar AS NEGATIVE FROM
docPoolSet][WHERE logical exp] ";"

train stmt ::= TRAIN classifierName USING
docVar FROM docPoolSet FOR CATEGORY category {"," docVar
FROM docPoolSet FOR CATEGORY category}[WHERE logical exp] ";"
```

```
docPoolSet ::= docPoolName "+" docPoolName
```

The first train statement is designed for binary classifier instances while the second one is for m-ary classifier instances. For binary classifier instances, HCL considers both positive training documents and negative training documents. In some binary classifier packages such as SVM, the training step requires both kinds of training documents. Some other packages may only require positive training documents. For m-ary classifier instances, we need to identify a set of training documents for each category in the classifier domain. The `FOR CATEGORY` clause is designed for this purpose. The above train statement uses a SQL-like statement to choose the training documents for classifiers. Similar to a SQL query statement where a tuple variable can be used to step through tuples from one or more relational table, the train statement provides document variables (`docVar`) to represent some document from one or more document pool. Note that several document pools can be aggregated together to form a larger document pool using the "+" operator. The `logical_exp` clause specifies the conditions to be imposed on the document variables representing the different sets of training documents used in training.

The train statement does not make any assumption about the number of training documents qualifying their `where` conditions. It is therefore the user's responsibility to ensure that adequate documents are available for training purposes.

The training statements of the three classifier instances constructed in Section 4.3 are given below.

```
TRAIN svmLocal USING posDoc AS POSITIVE FROM myTrgPool,
negDoc AS NEGATIVE FROM myTrgPool WHERE fin IN
posDoc.labeledCat AND fin NOT IN negDoc.labeledCat;

TRAIN svmSubtree USING posDoc AS POSITIVE FROM myTrgPool,
negDoc AS NEGATIVE FROM myTrgPool WHERE it.cover INTERSECT
posDoc.labeledCat != NULL AND it.cover INTERSECT
negDoc.labeledCat = NULL;

TRAIN nbLocal USING itDoc FROM myTrgPool FOR CATEGORY it,
dbDoc FROM myTrgPool FOR CATEGORY db, aiDoc FROM myTrgPool
FOR CATEGORY ai, webDoc FROM myTrgPool FOR CATEGORY web,
WHERE it IN itDoc.labeledCat AND db IN dbDoc.labeledCat
AND ai IN aiDoc.labeledCat AND web IN webDoc.labeledCat;
```

Since both `svmLocal` and `svmSubtree` are binary classifiers, we need to specify both the positive and negative training documents. Each training documents in the training pool `myTrgPool` may be labeled with multiple categories.

For svmLocal, as long as a document is labeled with the fin category, i.e., fin IN posDoc.labeledCat, we want the document to be selected as a positive training document. Other documents can be used as negative training documents.

For the binary classifier svmSubtree, we want the positive training documents to be from any of the categories in the subtree rooted in category it. Therefore, a document is used as positive training example if and only if its labeled categories overlap with fit, db, ai, web, that is it.cover. As mentioned in Section 4.3, the svmSubtree classifier will only assign a score value to a given document for the it category. As we train the classifier with training documents belonging to the subtree rooted at it, this score value therefore suggests how strong the classified document belongs to the subtree instead of the it category.

For the m-ary classifier nbLocal, there are four groups of training documents required, one for each category in the classifier's domain. Such grouping of training documents will allow the nbLocal classifier to later assign a score value to a classified document for each of the four categories.

4.5 Category Assignment

While the construct classifier and train stmt statements define classifier instances and their training data, the actual assignment of categories to documents must be specified by the assign category statement with the following syntax.

```
assign category ::= ASSIGN DOCUMENT docVar TO category
IF logical_exp ";"
```

The assign category statement essentially assigns documents to categories according to the score values given by the classifiers. Recall that the same document may be classified by different classifiers with different score values. The logical expression (logical_exp) allows Boolean conditions to be specified on the score values, and these conditions essentially compare score values with some thresholds. The document variable docVar, representing the documents to be classified, is required mainly for expressing the logical expression.

For example, to assign documents to the categories in our earlier category tree example, the following assign document statements can be used.

```
ASSIGN DOCUMENT d TO fin IF d.score(svmLocal,fin)>0.1;

ASSIGN DOCUMENT d TO it IF d.score(svmSubtree,it)>0.2
AND d.score(nbLocal, it)>0.1;
```

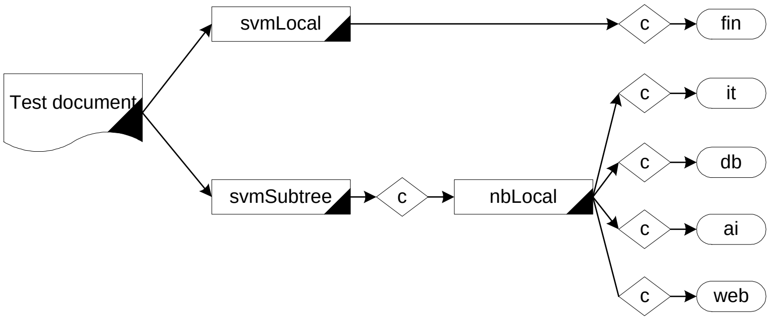

Figure 1.3. Classification order of using svmLocal, svmSubtree and nbLoocal

```
ASSIGN DOCUMENT d TO db IF d.score(svmSubtree,it)>0.2
AND d.score(nbLocal, db)>0.2;

ASSIGN DOCUMENT d TO ai IF d.score(svmSubtree,it)>0.2
AND d.score(nbLocal, ai)>0.2;

ASSIGN DOCUMENT d TO web IF d.score(svmSubtree,it)>0.2
AND d.score(nbLocal, web)>0.2;
```

In the first assign statement, a document is assigned to category `fin` if the score given by `svmLocal` for the category is greater than 0.1. The second assign statement indicates that only if a document is given a score higher than 0.2 by the `svmSubtree` classifier and a score higher than 0.1 by the `nbLocal` classifier, it is assigned to the `it` category. The other statements can be interpreted similarly.

Note that the `assign document` statements not only assign documents to categories, they together suggest the ordering of applying classifiers on a given document to assign categories to the document and the conditions of the assignment. In Figure 3, the diamonds with the character "c" indicate the assignment conditions. The figure shows that the categories `it`, `db`, `ai` and `web` share a common condition (i.e., `score(svmSubtree,it)>0.2`) on the score returned by the `svmSubtree` classifier. The common condition suggests that classification should be performed by the `svmSubtree` classifier before the `nbLocal` classifier. Such ordering information will help to reduce the amount of classification efforts tremendously since the `svmSubtree` classifier can discard documents before they are classified by the `nbLocal` classifier.

4.6 Discussions

So far, we have defined the important HCL language constructs to define a hierarchical classification method. We have illustrated the language using a

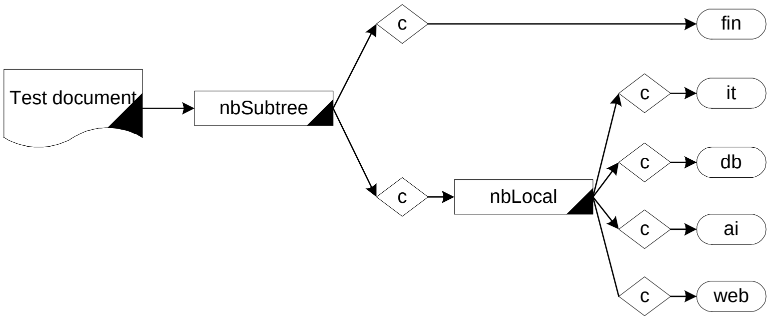

Figure 1.4. Classification order of using Naïve Bayes classifiers only

hierarchical classification example that uses a top-down approach to classify documents. In such an approach, the classifiers associated with the top-level categories will have to accept a document before the document is classified by the classifiers associated with the low-level categories. In this subsection, we will use HCL to specify another two variants of top-down hierarchical classification methods. The first uses Naïve Bayes classifiers only while the second was proposed by Dumais and Chen [1].

4.6.1 Hierarchical Classification using Naïve Bayes Classifiers Only

In this hierarchical classification method, we want to replace the `svmLocal` and `svmSubtree` classifiers in our earlier example by a m-ary classifier, called `nbSubtree`. The `nbSubtree` classifier will have `fin` and `it` as its domain, and assign scores to documents for the two categories. The score for `fin` will indicate if a document belongs to the `fin` category, while the score for `it` will indicate if the document belongs to the subtree rooted at `it`.

The classifier construction, train, and category assignment statements for the `nbSubtree` are shown below.

```
CONSTRUCT CLASSIFIER nbSubtree WHERE ENGINE= NaïveBayes,
TYPE = m-ary, DOMAIN= fin,it;

TRAIN nbSubtree USING finDoc FROM myTrgPool FOR CATEGORY
fin, itTreeDoc FROM myTrgPool FOR CATEGORY it WHERE fin
IN finDoc.labeledCat AND it.cover INTERSECT
itTreeDoc.labeledCat!  = NULL;

ASSIGN DOCUMENT d TO fin IF d.score (nbSubtree, fin)> 0.2;

ASSIGN DOCUMENT d TO it IF d.score(nbSubtree, it)> 0.2
```

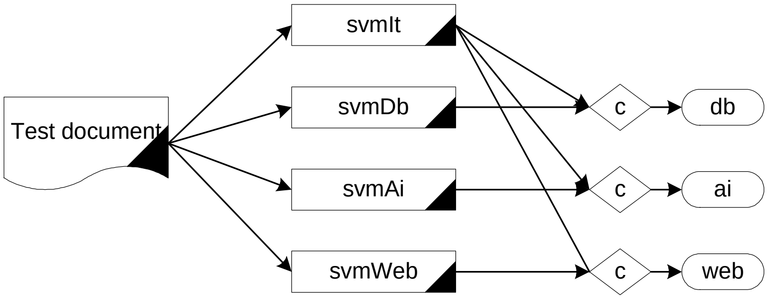

Figure 1.5. Classification order of using multiplicative thresholding and SVM

```
AND d.score(nbLocal, it)>0.2;

ASSIGN DOCUMENT d TO db IF d.score(nbSubtree, it)> 0.2
AND d.score(nbLocal, db)>0.2;

ASSIGN DOCUMENT d TO ai IF d.score(nbSubtree, it)> 0.2
AND d.score(nbLocal, ai)>0.2;

ASSIGN DOCUMENT d TO web IF d.score (nbSubtree, it)> 0.2
AND d.score(nbLocal, web)>0.2;
```

4.6.2 Hierarchical Classification with Multiplicative Thresholding

In this section, we illustrate the use of HCL to define the hierarchical classification method using multiplicative thresholding strategy. This method was first proposed by Dumais and Chen [1]. In their work, a virtual category tree of height 2 was used. A binary SVM classifier is assigned to the root category, and one SVM classifier is assigned to each category at the leaf level. Only if the product of scores returned by the root and leaf classifiers exceeds a pre-defined threshold, a document is then assigned to the leaf category.

To specify the classification method using HCL on the 2-level category tree (rooted at `it`) extracted from our earlier example. The three categories in Figure 1, `db`, `ai`, and `web` are real and `it` is virtual. Note that the category tree in this section is different from the one we used in previous sections. In the following, we present the complete classification method specification.

```
DEFINE CATEGORY TREE itTree WHERE VIRTUAL it (‘‘information
technology’’) IS ROOT, REAL db (‘‘databases") IS CHILD OF it,
REAL ai (‘‘artificial intelligence’’) IS CHILD OF it,
REAL web (‘‘world wide web’’) IS CHILD OF it;
```

```
CONSTRUCT CLASSIFIER svmIt WHERE ENGINE = SVM, TYPE = binary,
DOMAIN = it;

CONSTRUCT CLASSIFIER svmDb WHERE ENGINE = SVM, TYPE = binary,
DOMAIN = db;

CONSTRUCT CLASSIFIER svmAi WHERE ENGINE = SVM, TYPE = binary,
DOMAIN = ai;

CONSTRUCT CLASSIFIER svmWeb WHERE ENGINE = SVM, TYPE = binary,
DOMAIN = web;

TRAIN svmIt USING posDoc AS POSITIVE FROM myTrgPool,
negDoc AS NEGATIVE FROM myTrgPool WHERE it.cover INTERSECT
posDoc.labeledCat!= NULL AND it.cover INTERSECT
negDoc.labeledCat = NULL;

TRAIN svmDb USING posDoc AS POSITIVE FROM myTrgPool,
negDoc AS NEGATIVE FROM myTrgPool WHERE db IN
posDoc.labeledCat AND db NOT IN negDoc.labeledCat
AND it.cover INTERSECT negDoc.labeledCat != NULL;

TRAIN svmAi USING posDoc AS POSITIVE FROM myTrgPool,
negDoc AS NEGATIVE FROM myTrgPool WHERE ai IN
posDoc.labeledCat AND ai NOT IN negDoc.labeledCat
AND it.cover INTERSECT negDoc.labeledCat != NULL;

TRAIN svmWeb USING posDoc AS POSITIVE FROM myTrgPool,
negDoc AS NEGATIVE FROM myTrgPool WHERE web IN
posDoc.labeledCat AND web NOT IN negDoc.labeledCat
AND it.cover INTERSECT negDoc.labeledCat != NULL;

ASSIGN DOCUMENT d TO db IF d.score(svmIt, it)
* d.score(svmDb, db)>0.1;

ASSIGN DOCUMENT d TO ai IF d.score(svmIt, it)
* d.score(svmAi, ai)>0.1;

ASSIGN DOCUMENT d TO web IF d.score(svmIt, it)
```

```
* d.score(svmWeb, web)>0.1;
```

In the above train statements for the leaf level classifiers, the selection of positive training documents is quite straightforward, but the selection of negative training documents requires some explanation. Since the `svmIt` classifier is designed to filter away documents not belonging to the category tree, the negative training documents are therefore chosen from those training documents under the category tree but not belonging to the leaf level categories. The category assignment statements specify that a document is assigned to the leaf level category if the product of scores from the root classifier and the leaf level classifier exceeds 0.1. The classification order is shown in Figure 5.

5. Conclusions

In this chapter, we propose a specification language for hierarchical text classification known as HCL. HCL provides four essential language primitives, i.e., `define`, `construct`, `train` and `assign`, to define a hierarchical classification method. We have illustrated the features of HCL using a few examples. With a standard classification language such as HCL, the process of constructing a hierarchical classification method will be simplified making it also easier for performance evaluation and tuning. At present, the design of HCL is still tentative. There are still several important future research issues to be addressed:

- Implementation of HCL: We plan to implement HCL on different classifier packages to study the detailed implementation issues and to promote it to be the standard way for building hierarchical classification methods.
- Design and implementation of the Method Upgrader module: We have not investigated the research issues involved in designing the method upgrader module so far. This is an important component in our classification method generation system. Further research on the method upgrading techniques will be required before they can be incorporated into the overall method generation system.
- Reporting facility: A full-fledged classification method should report its performance when it is used on some given document collection. This reporting facility has not be added to our design yet but should be considered in the future.

References

[1] S. T. Dumais and H. Chen. Hierarchical classification of Web content. In *Proc. of the 23rd ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 256–263, Athens, GR, 2000.

[2] J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. Dmql: A data mining query language for relational databases. In *SIGMOD'96 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'96)*, Montreal, Canada, 1996.

[3] D. Koller and M. Sahami. Hierarchically classifying documents using very few words. In *Proc. of the 14th Int. Conf. on Machine Learning*, pages 170–178, Nashville, US, 1997.

[4] A. K. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. http://www.cs.cmu.edu/∼mccallum/bow, 1996.

[5] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *Proc. of the 22nd Int. Conf. on Very Large Data Bases*, pages 122–133, Mumbai, India, Sep 1996.

[6] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.

[7] A. Sun and E.-P. Lim. Hierarchical text classification and evaluation. In *Proc. of the 1st IEEE Int. Conf. on Data Mining*, pages 521–528, California, USA, Nov 2001.

[8] Y. Yang. An evaluation of statistical approaches to text categorization. *Information Retrieval*, 1(1-2):69–90, 1999.