Categorization In A Hierarchically Structured Text Database

Ferhat Kutlu and H. Altay Güvenir
Computer Engineering Department
Bilkent University Ankara, Turkey
{fkutlu,guvenir}@cs.bilkent.edu.tr

ABSTRACT

This paper describes a new categorization learning algorithm, called *Categorization In A Hierarchically Structured Text Database* (CHSD), for exploiting the built-in hierarchy of a text database. CHSD has two phases called *learning phase* and *categorization phase*. At the former, it exclusively learns the hierarchy and constructs an index tree representing the hierarchy with a node for each different level. At the latter, using this index-tree, it categorizes a given news text into its relevant categories. Index-tree is constructed by an agglomerative bottom-up hierarchical approach and categorization is done by a supervised overlapping approach. Speed is the main criterion in CHSD keeping the cost of accuracy in a tolerable level. It tries to ignore most of the redundant parts of the data in learning and similarity calculations in categorization by the help of hierarchical structure. The k Nearest Neighbor categorization algorithm is used to assess the loss in accuracy, since it has a very high categorization accuracy. A Usenet news database is compiled and used to test CHSD since it has such a built-in hierarchy with many text based documents appropriate for the requirements of CHSD.
Keywords: learning, categorization, hierarchy, Usenet, index-tree.

I Introduction

Amount of on-line information is growing at an ever-increasing rate and the needs for concepts to help manage this huge size of information are rising each day. One of these concepts is the *categorization* of every kind of data which makes data parts with similar contents to be in the same category. To date there have been many categorization algorithms implemented. In this paper, we investigate the benefits of built-in hierarchical structures in text databases. We used a sample database downloaded from Usenet newsgroups system because it has such a built-in hierarchical structure in itself which makes us able to run our algorithm without a pre-arrangement of the input data, and postings are grouped together by category, so initiating a supervised learning is very easy. In addition, Usenet is a source of large number of documents and there is always new data available for training and testing [10]. Most of the news are text-based and there is no need to worry about removing HTML commands or interpreting image files, and there is a large variety of subjects covered, so it is possible to study a particular area or more general topics. The algorithm presented here is called *Categorization in a Hierarchically Structured Text Database* (CHSD in short). CHSD takes as input a collection of Usenet messages. Then it constructs an index tree which is as high as the longest hierarchy in the database such that each node of the tree contains different number of features inherited from its children. Finally a new document/a set of documents travel(s) down the tree by the guidance of a similarity measure and a *threshold* value of this similarity measure to find its/their related category/categories. In Section II, we present the pseudocodes of CHSD, explain them in details. In Section III, we do complexity analysis

and empirical evaluation of our algorithm, present our test results, and compare results of our algorithm with the results of k Nearest Neighbor categorization algorithm (KNN) [5, 9] which we ran over the same sample database. At the end of Section III we give the results of a scalability test that we did with as many documents as we could collect. Finally we give our conclusions and determine the future work.

II CHSD Algorithm

CHSD algorithm is similar to the other *categorization learning* algorithms [10, 4, 8]. First of all, it executes a learning phase which takes more time than the others, but after such a heavy work it achieves the capability of doing faster categorization. Because an index tree is constructed to the cost of many data replications in the learning phase but this makes it easier and faster to categorize new documents.
CHSD operates on a sample space of m categories in which each category consists of n documents. Each category is represented by a three-row vector in which rows contain *words*, *frequencies* and *norm-scaling values* respectively. Beginning with m vectors an **agglomerative, bottom-up hierarchical** approach is applied that ends up with a single node at root and an index tree with nodes that have different numbers of children.
At learning phase CHSD begins with raw data and takes in a database which consists of multiple groups such that each group has multiple documents and a group name which is a concatenation of multiple names implying the hierarchy of database. First of all InitTree function which is given in Figure 1 takes database as input and passes it to ProcessData(Figure 2).

```
InitTree (DB) /* DB: database of newsgroups */
[1]  GroupNo ← ProcessData(DB)
[2]  Create RootNode[GroupNo]
[3]  for each group_i ∈ DB do
[4]      Create NewNode /* a node with a name and a words array */
[5]      NewNode.name ← group_i.name
[6]      for each word_k ∈ group_i do
[7]          NewNode.words[k].name ← word_k.name
[8]          NewNode.words[k].frequency ← word_k.frequency
[9]          NewNode.words[k].scale ← word_k.scale
[10]     RootNode[i] ← NewNode
[11]BuildTree(RootNode, 1)
```

Figure 1: InitTree Function

ProcessData deals with each word of each document in each group in lines 5-17. In lines 10-14 frequency table is filled up and in lines 15-17 denominator of the norm-scaling formula (equation 1) is calculated for each word. Hash function mentioned in line 8 could be any hash function which generates a definitely distinct number for each different word in the database.
After all documents are processed in the current group, norm-scaling values are calculated and a vector file is written out for each group which contains names, frequencies and norm-

scaling values of words (line 21). The *if check* in line 19 prevents zero values to be written out thereby CHSD deals with only non-zero values. After all groups are processed ProcessData returns the number of groups in the database.
We scale the word frequencies by norm-scaling method [2, 6] which is given by equation:

$$d_i = \frac{TF_i}{\sqrt{\sum_j TF_j^2}} \tag{1}$$

where d_i stands for the relative frequency of word i, TF_i stands for the total frequency of word i, and TF_j stands for the frequency of word i in particular document j. By this scaling unimportant words for similarity calculation gain lower values while important words gain higher values.
In line 2 of InitTree a root node is created which has enough number of pointers for leaf nodes. For each newsgroup in the database a new node is created such that each node keeps the name of the group and a word vector to include leaf vector of the current group. Initialization is done after all nodes get filled up and joined to the root.

```
ProcessData (DB)
[1]  StopList /* list of words to be ignored */
[2]  SubDictionary /* list of words in current group under DB */
[3]  Frequency /* keeps frequency of each word in SubDictionary */
[4]  Scale /* keeps normal scale value of each word in SubDictionary */
[5]  for each group_i ∈ DB do
[6]      for each document_j ∈ group_i do
[7]          for each word_k ∈ document_j do
[8]              hashValue ← Hash(word_k)
[9]              /* Hash function returns a bucket number for current word */
[10]             if word_k ∉ StopList and word_k ∉ SubDictionary then
[11]               SubDictionary[hashValue] ← word_k
[12]               Frequency[hashValue] ← Frequency[hashValue]+1
[13]             if word_k ∉ StopList and word_k ∈ SubDictionary then
[14]               Frequency[hashValue] ← Frequency[hashValue]+1
[15]         for each word_k ∈ SubDictionary do
[16]             if Frequency[k] ≠ 0 then
[17]               Scale[k] ← Scale[k] + Frequency[k] * Frequency[k]
[18]     for each word_k ∈ SubDictionary do
[19]         if Frequency[k] ≠ 0 then
[20]           Scale[k] ← Frequency[k] / √Scale[k]
[21]           WriteToFile(SubDictionary[k], Frequency[k], Scale[k])
[22]           SubDictionary[k] ← NULL /* reset arrays */
[23]           Frequency[k] ← 0
[24]           Scale[k] ← 0
[25] return i
```

Figure 2: ProcessData Function

At the next step, InitTree calls BuildTree (Figure 3) by passing the root and number 1 to it. Number 1 stands for the first parts of the hierarchical names mentioned above. Thus, BuildTree begins to construct the index tree by merging the leaf nodes with similar first names at its first recursion.

```
BuildTree (Node, key)
[1]  for each Group_i in the children of Node
[2]  such that first key many parts of their names are similar do
[3]      Create NewNode /* a node with a name, a words array, child pointers */
[4]      NewNode.name ← (concatenation of key many similar parts detected)
[5]      wordCounter ← 0
[6]      childCounter ← 0
[7]      for each node_j ∈ Group_i do
[8]          for each word_k ∈ node_j do
[9]              if node_j.words[k].scale ≥ √2 then /* eliminate ignorable words */
[10]                 NewNode.words[wordCounter].name ← node_j.words[k].name
[11]                 NewNode.words[wordCounter].frequency ← node_j.words[k].frequency
[12]                 NewNode.words[wordCounter].scale ← node_j.words[k].scale
[13]                 wordCounter ← wordCounter + 1
[14]         NewNode[childCounter] ← node_j
[15]         childCounter ← childCounter + 1
[16]Node[i] ← NewNode
[17]BuildTree(NewNode, key+1) /* go on recursively */
```

Figure 3: BuildTree Function

BuildTree takes in a node and a key value as input. It detects the groups which have similar first key many name parts. In other words siblings are found first and a new node is created for each sibling group. Each new node takes concatenation of those key many names as its name (line 4). Then BuildTree fills in the words vector of new node with the words of its children as shown in lines 7 through 14 of Figure 3.

One of the most crucial points in BuildTree function is to sieve the words of leaf nodes. In our experiments we determined $\sqrt{2}$ as the **threshold of norm-scaling value** for a word to be copied up to the nodes over the leaf level. That is, the words with norm-scaling values less than $\sqrt{2}$ will be present only in leaf nodes. The *if check* in line 9 of BuildTree function does this work. Purpose of such an elimination is to get rid of ignorable words and to lessen the number of words in the vectors of nodes that are higher than the leaf level. So that categorization phase becomes faster and the negative effect of noisy data is prevented to some extent.

Recursion of BuildTree stops when there are no siblings to be merged by a new parent node. That is, all hierarchies of the database are constructed and the index tree is ready for categorization phase.

FindCategories takes a text document and a **threshold value of similarity** as input (Figure 4). First of all it creates a new node for this document and fills in the words vector of this new node from the document as shown in the lines 3-8. At this step it is a must to process data with the same functions used in ProcessData (Figure 2) for consistency. Otherwise

accuracy decreases too much. After the new node gets filled up it is passed to HierarchicalSearch (Figure 5) in line 9 to make it travel down the tree and to get its FoundCategories array filled up with the names of the leaf nodes visited.
HierarchicalSearch is a recursive function which takes in a node, a threshold value and an empty array as input as shown in Figure 5. Beginning by the root's children it calculates the similarity value between the new node and the nodes of index tree by our Similarity function (Figure 6) and it gives way to recursion through the nodes which has similarity to new node higher than the threshold value of similarity. Actually the threshold value of similarity is an option of user which floats between 0 - 1. It acts as a measure which determines the sensitivity of the algorithm. If we use **0** as a threshold value then the new node will visit all other nodes in the index tree and we will get the names of all leaf nodes as a result of our categorization request and because of this the query will take much time. If we use **1** as a threshold then we will most probably get no result of our categorization request and the query will take very short time.
Similarity function given in Figure 6 takes in two word vectors - containing words and their frequencies - as input and calculates similarity between them according to the equation 2 :

$$Sim(v_1, v_2) = \frac{\sum_t w_{t,v_1}.w_{t,v_2}}{\sqrt{\sum_t w_{t,v_1}^2}.\sqrt{\sum_t w_{t,v_2}^2}} \tag{2}$$

where $w_{t,x}$ stands for the frequency of word t in vector x, and the result is the similarity between vectors v_1 *and* v_2 such that 1 for identical vectors and 0 for the vectors with no common terms. We only deal with the common words of both vectors in this formula.
Finally in line 6, HierarchicalSearch fills in the FoundCategories array with the names of **visited leaf nodes** by the help of Insert function which is given in Figure 7. Insert function takes in an array of category names, a new category name and a similarity value belonging to that new category name as input as shown in Figure 7. It inserts the new name according to its similarity value in the array, so that the input array is kept sorted by similarity values in non-increasing order. So the first category name in FoundCategories array is the best match for the new document.
After all recursions are popped up in HierarchicalSearch, control returns to FindCategories and FoundCategories array becomes filled up with the category names determined by CHSD

```
FindCategories (document, threshold)
[1]  StopList /* list of words to be ignored */
[2]  FoundCategories /* keeps names of categories found */
[3]  Create NewNode /* a node with a words array only */
[4]  for each word_k ∈ document do
[5]      if word_k ∉ StopList and word_k ∉ NewNode.words then
[6]          NewNode.words[k] ← word_k
[7]      if word_k ∉ StopList and word_k ∈ NewNode.words then
[8]          NewNode.words[k].frequency ← NewNode.words[k].frequency+1
[9]  HierarchicalSearch(NewNode, RootNode, threshold, FoundCategories)
[10] return FoundCategories
```

Figure 4: FindCategories Function

```
HierarchicalSearch (NewNode, Node, threshold, FoundCategories)
[1]  i ← 0
[2]  while Node.child_i ≠ NULL do
[3]      sim ← Similarity(NewNode.words, Node.child_i.words)
[4]      if sim ≥ threshold then
[5]        if Node.child_i.child = NULL then /* if NewNode met a leaf node */
[6]          Insert(FoundCategories, Node.child_i.name, sim)
[7]        else
[8]          HierarchicalSearch (NewNode, Node.child_i, threshold, FoundCategories)
[9]      i ← i+1
```

Figure 5: HierarchicalSearch Function

and then it is returned as the result of the query (line 10).

III Evaluation

i Time Complexity Analysis

Notations given in Table 1 are used to explain complexity analysis. Actually these notations are abbreviations retrieved by concatenating initial letters of the input data features such as numbers of words, documents and newsgroups.
Time complexities of each function of CHSD are given in Figure 8. As explained in Section II ProcessData, InitTree and BuildTree are implemented sequentially in the learning phase. Thus the time complexity of learning phase is $O(tgodb \cdot tdog \cdot twog)$.
FindCategories is the main function of the categorization phase. It calls HierarchicalSearch and HierarchicalSearch calls auxiliary functions Similarity and Insert. Since HierarchicalSearch and Similarity functions are the most time consuming functions, time complexity of categorization phase goes to $O(h \cdot \log tcon \cdot twon^2)$.

```
Similarity (guest, host)
[1]  number ← 0 /* upper part of formula */
[2]  divisor1 ← 0 /* first part of divisor */
[3]  divisor2 ← 0 /* second part of divisor */
[4]  for each word_k ∈ guest do
[5]      for each word_m ∈ host do
[6]          if word_k = word_m then
[7]              number ← number + word_k.frequency * word_m.frequency
[8]              divisor1 ← divisor1 + word_m.frequency * word_m.frequency
[9]      divisor2 ← divisor2 + word_k.frequency * word_k.frequency
[10] return (number / (√divisor1 * √divisor2))
```

Figure 6: Similarity Function

NOTATION	MEANING	NOTATION	MEANING
twon	total words of node	twod	total words of a document
tcon	total children of a node	tgodb	total groups of database
tdog	total documents of a group	twog	total words of a group
tcf	total categories found	h	height of the index tree

Table 1: Notations Used in Complexity Analysis Formulations

As for the overall time complexity of CHSD, learning phase is so dominant that time complexity of CHSD is $O(tgodb \cdot tdog \cdot twog)$. Because main goal of CHSD is to learn the built-in hierarchy in the database and to make the categorization phase easier and faster. Briefly, the time complexity of CHSD grows by the number of groups in the database, the number of documents in the groups and the number of words in the documents.

ii Space Complexity Analysis

An algorithm which requires only constant memory space such that the memory required is independent on input size is called as *in-place algorithm*. CHSD is not an in-place algorithm because its space complexity is $O(now \cdot non)$ in the worst case where now is the *number of words* in the global dictionary of the database and non is the *number of nodes* in the index tree. That is each word occurs in all documents and copied to all nodes in the index tree. However it is not possible for this worst case to be realized since CHSD eliminates the words according to their norm-scaling values as explained in Section 2.

iii Performance Measures

While a number of different accuracy measures have been used in evaluating text categorization in the past, almost all have been based on the same model of decision making by the

```
Insert (FoundCategories, category, sim)
[1]  i ← 0
[2]  while FoundCategories[i].sim ≥ sim do
[3]       i ← i+1 /* find the right place in sorted order */
[3]  temp ← FoundCategories[i]
[4]  FoundCategories[i] ← category
[5]  i ← i+1
[6]  j ← i
[7]  while FoundCategories[j] do
[8]       j ← j+1 /* go to the end of FoundCategories */
[9]  j ← j+1
[10] while j ≥ i do /* one right shift until i'th element */
[11]      FoundCategories[j] ← FoundCategories[j - 1]
[12]      j ← j - 1
[13] FoundCategories[j] ← temp
```

Figure 7: Insert Function

FUNCTION	TIME COMPLEXITY
ProcessData	$O(tgodb \cdot tdog \cdot twog)$
InitTree	$O(tgodb \cdot tdog \cdot twog)$
BuildTree	$O(h \cdot \log tgodb \cdot twog)$
Insert	$O(tcf)$
Similarity	$O(twon^2)$ ($O(twon \cdot \log twon)$ when Binary Search is used)
HierarchicalSearch	$O(h \cdot \log tcon \cdot twon^2)$
FindCategories	$O(h \cdot \log tcon \cdot twon^2)$

Figure 8: Time Complexities of Functions

categorization system [3]. Some of these measures are *recall and precision, accuracy or error, break-even point, micro average, macro average* and *11-point average precision* [11]. For the evaluation of our test results of CHSD and KNN we used interpolated 11-point average precision measure method which is especially designed for category ranking.

iv Data Set

For the evaluation tests of CHSD a sample database of 2000 documents is collected from Usenet top-level groups called **comp** and **bionet**. Each newsgroup under comp contains 100 documents and each newsgroup under bionet contains different number of documents. This is done on purpose to test robustness of CHSD. Because in real life it is not possible to find a balanced database such that each group contains equal number of documents. In fact categorization algorithm has to prevent the effect of different data sizes and irrelevant features on results [1]. This database does not include all newsgroups of comp but we collected all news in all newsgroups of under the top-level bionet.

v Test Results

Three measures taken for the evaluation of test results. *Train time* is the time spent during learning phase but does not include the time spent for processing raw data, *test time* is the time spent during categorization of query documents, and *accuracy* is the value calculated by interpolated 11-point average precision method which takes in the real labels and the labels found by the running algorithm.
10-fold cross validation technique [7] is used in the experiments. Therefore, the accuracy of algorithms on data set is computed as the average of 10 runs in each of which a disjoint set of 1/10 of the data set is used in the querying, and the remaining 9/10 in the training phase. Firstly we ran [1] CHSD over the database of 2000 documents for 16 times with different threshold values floating between 0.5 - 0.875. At each run we increased the threshold value by 0.025 to determine the best threshold value which gives the highest accuracy.
The results are given in Table 2. The best accuracy value that we found is **0.87** with threshold value **0.825** at the 14^{th} run which took **4724** msec. train time and **6173** msec. test time. We accept these values as the representative values of CHSD to compare it with

[1] We used a computer with 64 MB memory and an Intel Celeron 333 Mhz. processor in our experiments.

KNN. We did not run CHSD with further more threshold values since accuracy began to decrease after 0.825.
Since we shuffled data at the beginning of each run of CHSD, the train time changed sporadically. However the test time decreased as the threshold value increased because number of unvisited nodes becomes higher by the increase in threshold value. That is, as threshold value approaches to 1 test time approaches to 0.
Next we ran KNN over the same database with k parameter as 10 and got **0.93** accuracy, in **205492** msec. train time and **27196800** msec. test time. In other words, for each data fold KNN spent nearly 3.5 minutes to train and 454 minutes to test the data.
As compared with KNN, CHSD resulted in 0.06 less accuracy even with the most optimal threshold value as shown in Table 2. Nevertheless this is not an intolerable loss of accuracy as we consider the speed difference between them. According to the results of this experiment CHSD was faster than KNN **43** times in the training time and for **4405** times in the test time.

vi Scalability Test

The database used in previous experiment is expanded to the size of 5000 documents by adding a database of 30 newsgroups with 100 news in each. We took the threshold value of similarity as 0.825 since it was found to be the most optimal value in the previous test. Accuracy is calculated as **0.892** in **21922** msec. train time and **48656** msec. test time. Naturally train and test times increased but not beyond the considerable limits. Fortunately accuracy increased by 0.022 as compared to 0.87 in previous test. The main reason of this increase is the effect of larger data such that the more learning brought out the better results. By this results we conclude that it is definitely scalable.

IV Conclusion

As compared with KNN, CHSD results in less accuracy even with the most optimal threshold value. But this is not an intolerable difference. However a query takes shorter time by CHSD

No.	Threshold	Train(msec.)	Test(msec.)	Accuracy
1	0.5	4707	6721	0.22
2	0.525	4657	6679	0.29
3	0.55	4602	6610	0.33
4	0.575	4658	6551	0.38
5	0.6	4687	6509	0.42
6	0.625	4668	6454	0.46
7	0.65	4671	6406	0.48
8	0.675	4642	6392	0.51
9	0.7	4635	6350	0.55
10	0.725	4606	6305	0.59
11	0.75	4721	6289	0.63
12	0.775	4604	6237	0.72
13	0.8	4664	6191	0.81
14	**0.825**	**4724**	**6173**	**0.87**
15	0.85	4623	6122	0.79
16	0.875	4723	6099	0.65

Table 2: Results of CHSD on the Database Given in Table 4.2

than it does by KNN. For instance in the experiment of 2000 documents and 6495 words CHSD is 2514 times faster than KNN at the average of total train and test time. We conclude that time scalability of an algorithm is so important because amount of data is getting bigger each day thereby CHSD accomplishes its mission.
The advantages of CHSD: Robust against the differences among data sizes of groups to some considerable extent, faster than the most of the traditional categorization algorithms, scalable to larger databases in terms of time and space complexities, easy to implement, use and improve since it is a combination of components which are open to modifications and improvements, height of the index tree is limited by the depth of the hierarchy in database, and it has two useful measures to play with for better results which are called threshold of norm-scaling value and threshold of similarity.
The disadvantages of CHSD: Vulnerable to the spamming in the database, necessarily most of the initial data is replicated in the inner nodes of index tree to construct the hierarchy in the memory, the source database that the initial data is produced from must be maintained periodically to keep index-tree updated in long term, and the optimal threshold value of similarity that controls down-travel of a query vector should be determined beforehand by experimentation since it varies according to the types and sizes of databases.
As for the future work, CHSD might have a dynamic structure such that it continues its learning phase while responding to queries and receiving newly posted news from server and updates its index. Some of the newsgroups no longer have message traffic and CHSD can detect these groups and eradicate them from its index.

REFERENCES

[1] Güvenir H. A. A classification learning algorithm robust to irrelevant features. In *Proceedings of AIMSA'98, Giunchiglia F. (Ed.)*, volume AIMSA'98, pages 281–290, Sozopol, Bulgaria, September 1998.

[2] Boley D. Principal direction divisive partitioning. Technical Report TR-97-06, Department of Computer Science, University of Minnesota, Minneapolis, USA, June 1997.

[3] Lewis D.D. Evaluating text categorization. In *Proceedings of the Speech and Natural Language Workshop*, pages 312–318, Morgan Kaufman, San Mateo, CA, February 1991.

[4] Han E., Boley D., Gomno M., Gross R., Hastings K., Karypis G., Kumar V., Mobasher B., and Moore J. Webace: A web agent for document categorization and exploration. Technical Report TR-97-049, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, USA, 1997.

[5] Demiröz G. and Güvenir H. A. Genetic algorithms to learn feature weights for the nearest neighbor algorithm. In *Proceedings of the 6th Belgian-Dutch Conference on Machine Learning, H. J. van den Herik and T. Weijters (Eds.)*, volume BENELEARN-96, pages 117–126, Universiteit Maastricht, The Netherlands, 1996.

[6] Salton G. and McGill M.J. *Introduction to Modern Information Retrieval.* McGraw Hill, 1983.

[7] H. A. Güvenir and İ. Şirin. Classification by feature partitioning. *Machine Learning*, 23:47–67, 1996.

[8] Yavuz T. and Güvenir H. A. Application of k-nearest neighbor on feature projections classifier to text categorization. In *Proceedings of the 13th International Symposium on Computer and Information Sciences, Güdükbay U., Dayar T., Gürsoy A., Gelenbe E. (Eds.)*, volume ISCIS'98, pages 135–142, Antalya, Turkey, October 1998.

[9] Mitchell T.M. *Machine Learning.* McGraw Hill, 1997.

[10] Scott A. Weiss, Simon Kasif, and Eric Brill. Text classification in usenet newsgroups: A progress report. In *Proceedings of the AAAI Spring Symposium on Machine Learning in Information Access*, pages 11–13, Bulgaria, September 1996.

[11] Yiming Yang. *An Evaluation of Statistical Approaches to Text Categorization.* Kluwer Academic Publishers, 1999.