TEXT CATEGORIZATION USING k-NEAREST NEIGHBOR CLASSIFICATION

Gülen Toker, Öznur Kırmemiş

Computer Engineering Department, Middle East Technical University

ABSTRACT

This paper introduces a document organization application of text categorization, using k-Nearest Neighbor (k-NN) classification. The method makes use of training documents, which have known categories, and finds the closest neighbors of the new sample document among all. These neighbors enables to find the new document's category. The solution involves a similarity function in finding the confidence of a document to a previously known category.

KEYWORDS : text categorization, k-NN classification, similarity functions, document organization

1. INTRODUCTION

Text categorization (or text classification) can be briefly described as the automatization of the document organization process to a set of pre-defined categories.

Text classification has many applications in daily life because non-automated organization methods are not satisfactory any more and there is a need to organize a huge amount of data and documents automatically. Some examples of applications in daily life can be listed as follows[1][5][6]:

1. Web page search engines use text categorization in an hierarchical manner. Documents are categorized according to their topics and chosen to be in a category listed hierarchically.
2. Text Categorization is used in text filtering. In this approach, a document is either forwarded or stopped in a flow to a user -also called consumer- according to the consumer's profile. In this application, two categories are necessary.
3. Text Categorization is used in some Natural Language Processing (NLP) applications such as, finding the meaning of a disambiguos word in a given text. Such words have more than one meaning and text categorization will be used in finding out, which meaning of the word is used in that sample text.

As it is mentioned, application areas of text categorization vary widely from document organization and text filtering to NLP applications. The main idea behind is however, automatization. [1]

Among all methods, k-Nearest Neighbor is considered to be the best method to start research on text categorization. k-NN classification requires some initial files, that are generally called as training documents. These documents' categories are known before method is applied. A new document with no category yet, is compared with all these training documents with respect to the terms they share. Finally, the documents, that the new document is closest to, are determined, and the category is assigned accordingly. Text categorization will give better results if machine learning is included in the approach, in such a way that newly categorized documents, could be added to the training documents set, together with their previously determined categories. However, in this paper and the implementation of the method described, k-NN is concentrated on, itself.

This paper is organized as follows: section 2 is devoted to the problem description, section 3 describes the algorithm proposed to solve the problem and clarifies the implementation details, section 4 demonstrates the results of the tests obtained through testing the algorithm via different parameter settings, and section 5 offers a few concluding remarks. Future lines of research are proposed in section 6. References can be seen in the final section.

2. DESCRIPTION

2.1 Problem Description

We designed and implemented a document organization application of text categorization where k-NN method is used. It is the classic problem: We have some documents, which are all in some predefined categories and the category is exactly known. (Categories and documents are in Turkish. Some examples of our pre-defined categories are; 'sinema', 'astroloji' etc.). We require to find the category of a new document taken as input to our program.[1]

In documents, there are words that are frequently used in daily life, which do not have any importance and effect in categorizing documents. Some examples can be Turkish words like : 've', 'ile', 'de', 'bugün', 'yarın' and some other words used in questions like: 'ne', 'kim', 'nasil'…etc. While categorizing documents, these terms should be eliminated for the sake of efficiency. In our problem, it is useful to create terms dynamically reading the training documents, rather than predefining them. So, while additions to training documents are done, terms will be added also dynamically, and this will provide further accuracy to the categorization.

3. STATEMENT OF SOLUTION

We will try to explain the solution method in detail, and give a step-by-step algorithm to guarantee clarification about the problem solution. Algorithm used in solving the problem is given in section 3.1, and implementation details will be described in section 3.2.

3.1 Algorithm

Term space model representation is an important concept in text categorization [3]. This model has been developed to impose a structure on text. Within this model, different representations exist:

1. *Binary representation:*
   The document can be represented as a binary vector where a 1 bit in a slot *i* means the term *i* exists in that document and a 0 bit tells us the opposite. This representation is efficient; programs implemented using such an implementation run fast.

2. *Frequency Representation:*
   A document can also be represented according to the frequencies/occurrences of the terms. It should be mentioned that this representation is more accurate. To give an example; a word computer may occur once in a text about games but hundred times in a paper about computer science. The former representation cannot distinguish the difference. We use this representation in our implementation to provide accuracy.

Frequency representation is symbolized as *tf* and will be used seriously in the algorithm. It is not directly used to find similarities between documents but instead it is used in calculating the weights of every term. Let us give the formulas[2]:

$wtf(t) = tf * idf(t)$ *for term t,*

$idf = \log 2(N/n)$

Here N is the number of all documents and n is the number of documents where that term appears. The wtf calculation has to be done for every document and for every term that appears in it. This means, the wtf calculation is done $N*(\#\ of\ terms)$ times. So is the idf (t) calculation.

The weighted term frequencies (wtf) are used in the similarity function. Similarity function takes one training document and the new document as parameter. It returns a value that corresponds to the amount of similarity between these documents. The similarity function is given below[4]:

$$Sim(X,Dj) = [\ \sum_{ti \in (X \cap Dj)} xi*dij\ ]\ /\ [\ ||X|| * ||Dj||\ ]$$

Here X is a vector that keeps all terms in the new document (according to the term space model). Briefly, it represents the new document. *Dj* is the vector with same properties that represents the training document *j*. *ti* is a term that is found in both vectors. Both documents have this term. *xi* and *dij* are the weighted term frequencies for this term *i* and the two documents. All multiplications of weighted term frequencies are added for all shared terms in these two documents. If documents have a great number of terms, then they will have more common terms. But this should not increase the similarity rapidly. So norms divide the summation in the formula. However these norms also do not take directly the number of different terms (in other words, the length of the vectors) but they are calculated from the weighted term frequencies also[4]:

$$||X|| = squareroot(x1^2 + x2^2 + x3^2 + ...)$$

where all *x*' s are weight term frequencies of all terms in *X*.

The similarity function calculation is also repeated several times. The similarity of the new document to all training documents will be used later. So the calculation is repeated as many times as there are training documents. In practice, finding similarities means defining the neighborhood for the documents. Now we will find the category of the document *X* using our knowledge about the neighborhood. We will make use of a self chosen *k* in this operation to limit number of neighbors that will be used in determining category. In our algorithm; we chose *k* as the number of training documents.[4]

$$Conf(c,d) = [\ \sum_{ki \in K |\ (Class(ki') = c)} Sim(ki',d)\ ] / [\ \sum_{ki \in K} Sim(k,d)]$$

(Conf is confidence in long terms.)

Here c is any category, d is the new document whose category is desired to be found (*X* in the formula above). *K* is the neighborhood of size *k* for the document *X*. All similarities between the new document and the documents that belong to class c are added. Then they are divided to all similarities between the new document and training documents belonging to the neighborhood of *X* in size *k*.

Similarities may be reused because they were calculated before. Division is repeated for categories. Finally the confidences are compared and the category, for which the greatest confidence is calculated, is chosen as the category for the new document *d* (or *X*).

3.2 Implementation

3.2.1 Data Structures

We use arrays to represent vectors together with two-dimensional arrays (matrices) as the data structures in our implementation. The most significant structures that are used in the implementation are listed below:

terms[NO_OF_TERMS]
ele[NO_OF_ELE]
termspace[NO_OF_TRAININGDOCS][NO_OF_TERMS]
sample[NO_OF_TERMS]
cats[NO_OF_CATEGORIES]
noofdocsincats[NO_OF_CATEGORIES]
termcount[NO_OF_TERMS]
wf[NO_OF_TRAININGDOCS][NO_OF_TERMS]
double wfsample[NO_OF_TERMS]
double norms[NO_OF_TRAININGDOCS]
similarity[NO_OF_CATEGORIES]

The explanations of these structures are important in demonstrating the solution, therefore let us describe them.

As it is obvious from the names,

NO_OF_TERMS is defined as the number of terms. Since arrays cannot grow dynamically; we put an upper limit for the number of terms (6000). The convention is trivial.

NO_OF_ELE holds the number of unimportant terms in the vector *ele*. It is mentioned before what kind of terms these are (Turkish words like‘ve’, ‘ile’, ‘bugün’….etc). Its value is currently 148 but can be modified as new terms are required to be eliminated.

NO_OF_TRAININGDOCS holds the number of training documents (and set to 30 currently) whereas *NO_OF_CATEGORIES* holds the total number of categories (10).

If we want to change any of the values above, we have to do it by opening the source and changing the #DEFINE lines manually.

terms is an array of strings and it holds terms gathered together dynamically during the reading process of all training documents.

ele is also an array of strings and holds the terms that will be eliminated from the training documents and from the tested document. Terms that appear in *ele* will be deleted from *terms.*

termspace is a matrix that holds the number of occurrences of each term that appears in each document. It holds integer values.

Sample is what we called the new document whose category is going to be tested. So *sample* is an integer array that holds the number of occurrences of all terms in the sample document.

cats array holds all the category names and *noofdocsincats* is an integer array that holds the number of documents that belongs to each corresponding category.

termcount is an integer array and is used in wtf calculation. It is used as *n* in the idf formula. The array holds the number of documents, a term appears in, for each term. That’s why its length is *NO_OF_TERMS.*

wf is again a matrix that holds real numbers. These real numbers correspond to the weighted term frequency for each term in each document. This matrix is of the same size with the *termspace* matrix.

wfsample is similar to sample except that it holds not term occurrences but wtfs of the sample document. *norms* holds real number norm calculations for each training document. Finally, *similarity* array holds double values and keeps the total similarity of each category found by calculateSIM().

3.2.2 Implementation Details

Let us shortly describe the functions used in the implementation before explaining the main code.

1. int calculateSIM()

calculateSIM() function is the most significant function in the code since it calculates the similarity of each training document to the sample document and categorizes it accordingly. It works as follows:

For each category, the wtf ’s of each training document is taken from the *wf* array - if not equal to zero – and put into the similarity function formula that is given in section 3.1.

When the calculation of the similarity of each training document is found, similarities of these documents belonging to the same category are added and put into the *similarity* array of length *NO_OF_CATEGORIES*. We have used a variable called *totalsim* that holds the current total similarities. If similarity[i] holds the greatest value among all, then the document belongs to category i. As you see, this function also performs the confidence calculation. k value used in k-NN method is chosen as the number of training documents, which simplifies this operation to dividing all similarity calculations to *totalsim.*

2. void calculateWF( )

The function fills the matrix *wf* and the array *wfsample*. It calculates weighted term frequencies using the formula given in our algorithm. While calculating log2(N/n), it adds the tested sample document to *n* if the term also exists in this new document but skips this operation if not.

3. void calculateNORMS( )

The function fills the *norms* array and sets the *normsample* variable**,** which will hold the norm of the sample document. It adds up the squares of all wtf's in the *wf* matrix; then takes the square root. Same operation is done on wfsample**.**

4. int isPunc(char)

This is a small helper function. It detects punctuation characters like '.', ',', '!' …etc. It returns either a 1 or a 0, if the parameter is one of these characters or not.

5. double log2 (double)

Trivially, it returns the logarithm of the parameter in base 2.

6. int testString(string ,int)

This function is used for efficiency purposes. Our program reads a whole document and takes every term excluding the eliminated ones as a new term. So there may occur both terms like 'bilgisayar' and 'bilgisayarcılık'. Instead of keeping both terms, we want just the shortest term to remain and the others to disappear since it will cause unnecessary calculation. The function works as follows;

For every term *i*, we take the length of the term and compare it with the length of the input string. If the length of the term is smaller than the length of the input string, we try to find out, if the input string contains the term in itself. If it contains the term, we mark the index. If the term is longer, we will check if it contains the input string and mark the index, in case it does. The function allows us to replace stems with roots and will prevent us to repeat one term or different forms (stems) of the same term, which will in turn helps the program to be more efficient and accurate.

Using the descriptions given above, let us explain the main program. After a definition of necessary parameters, the main program initializes the data structures. Then, it reads the file, which holds all the terms that should be eliminated from the training documents, while the termspace is being created.

Term space is created through using a file, namely input.txt, which keeps 'the name of the category', 'the number of training documents in that category' and 'the names of these documents'. The format should be like below:

category no_of_docs
document(1)
document(2)
.
.
document(no_of_docs)

This block is repeated for each category. The values are assigned to several parameters and help reading the training documents and operate on them.

Values read from the input file are filtered while being read. The code uses tolower(char) function to convert uppercase characters to lowercase letters and the self-defined isPunc() function to eliminate punctuation symbols. It finally eliminates unimportant terms and digits using isdigit(). While creating the term space, we have also overwritten longer terms (Roots overwrite stems.)

After the termspace is built, the interface of the program is created, which is demonstrated below;

---------- MAIN MENU ----------
DISPLAY CATEGORY NAMES: type 1, press enter
FIND CATEGORY OF A FILE: type 2, name of the file to be tested, press enter
FIND CATEGORIES OF FILES: type 3, name of the input file, press enter
EXIT: type 4, press enter

The user has four options, together with quitting the program.

If the user chooses 1, the *cats* array is displayed which holds all the category names.

Choice of 2 invokes the main operation, which takes the sample document file whose category is desired to be found, and reads it. It eliminates the punctuation characters and unimportant terms as trivial, converts lowercase letters and then invokes calculateWF(), calculateNORMS() and calculateSIM() functions in turn. After the category of the

document is found, it is displayed to the user.

Operation 3 is mainly created to categorize a number of documents in one step. It takes a file which contains the names of all sample documents whose categories are to be determined. It displays the categories of all files, as a result. This is especially useful in collecting statistics about the performance of the program.

4. RESULTS

For the purpose of performance analysis, the program is tested over a set of documents which are selected form different subjects. The documents have 10 different subjects, some of which are 'eğitim', 'spor', 'sinema' ...etc. We desire to classify the documents into their subjects correctly. We have tested the program through using 50 different documents, and the program correctly categorized 46 of them, whereas it assigned different categories than the desired ones to 4 of them, which can be thought as 92% accuracy rate.

We have studied on the results of the performance tests in order to found out the most important factors that effects the performance of the program. The factors that have been distinguished, can be listed as follows;

1. *The number of training documents:* As the number of the training documents increase, the accuracy of the program increases also. This is due to the fact that, more number of terms related to a category at hand, results in a better classification of sample documents. However, the number of training documents for each category should be equal in order to give all the categories an equal chance.
2. *The terms included in training documents:* The performance of the program increases with the increase in the relatedness of the terms included in the training documents to the category that they have. For instance, we have tested our program with different training documents sets, and the results vary significantly according to the degree to which each training document is really a good sample for its category. In addition, the more a training document contains related terms about its category, the better it helps in classification of a sample document. Therefore, the context of the training documents is very effective for the accuracy rate. So, training documents should be chosen as proper as possible in order to obtain better performance results.

3. *The context of the sample documents:* As for the case of the training documents, the context of a sample document is very important in determining its category correctly. For instance, if a sample document contains terms related to more than one category with nearly the same weight, the document can be classified to a category that is not desired. As mentioned before, 4 sample documents are classified wrongly in our tests, which is mainly due to this effect. For example, a sample document that includes terms related to both the categories 'bilgisayar' and 'eğitim', is classified as an 'eğitim' document, where its subject is actually desired to be 'bilgisayar'.

We believe that our algorithm has good potential in achieving better results when proper sets of training documents, and sample documents that contain much of the terms in its desired category, are used.

5. CONCLUSION

In this paper, a k-Nearest Neighbor algorithm is proposed to solve the text categorization problem statically. The algorithm uses a predefined set of documents with known categories. Their numbers are fixed in run time. With the help of them, the category of an input document is found.

The k-NN algorithm turned out to be an efficient and simple algorithm since it is based on a simple weighting approach. It is easily modifiable and can be developed further, as you will realize in section 6, where we demonstrate our proposed future work.

6. FUTURE WORK

The k-Nearest Neighbor algorithm is a easy modifiable algorithm and is adaptable to different problems. The algorithm allows us to develop our code further to increase efficiency and accuracy.

We assign exactly one category to one article in our implementation for sake of efficiency. But there exists articles containing many terms about many categories. Articles are not always written about one topic, but they may jump from topic to topic in daily life. This means, articles may have multiple categories. This version of our implementation turns out to give less information than it should, at that point. It will not be difficult to show other categories since the k-NN algorithm already calculates the necessary information for assigning multiple categories. Remember that we calculate all similarity totals and confidences of each category and choose the highest value to assign our category. We may calculate a standard deviation among all values and take an interval of values above a value rather than taking the maximum for such type of multi-categorized articles.

Another requirement in a huge k-NN application is dynamism. Our implementation is not used in a great application like Internet search engines but it could be modified accordingly. It

would not be difficult to add the new sample document into the training documents set by adding its name under the category that it is part of, inside the input file mentioned in section 3.2.2. This will create a growing training document set and will give increasingly correct answers as the set enlarges.

The furthest step that we may take in this project will be a modification that makes the code learn new categories. This requires knowledge about artificial intelligence and may be mentioned as a final destination.

REFERENCES

[1] Text Categorization Using k-Nearest Neighbour Classification, Survey Paper, Oznur Kirmemis and Gulen Toker, Middle East Technical University Computer Engineering Department.

[2] A P-tree basd K-Nearest Neighbor Text Classifier Using Intervalization by Imad Rahal, Hassan Najadat and William Perrizo, Computer Science Department, North Dakota State
University.

[3] A Simple k-NN Algorithm For Text Categorization by Pascal Soucy, Guy W. Mineau, Department of Computer Science, Universite Laval, Quebec, Canada

[4] Review of K-Nearest Neighbor Text Categorization Method, http://wwwcsif.cs.ucdavis.edu/~liaoy/research/text_ss02_html/node4.html

[5] Text Categorization: An Optimized P-Tree Based k-NN Approach, www.cs.ndsu.nodak.edu/~rahal/thesis/Thesis_Final.pdf http://www.cs.ndsu.nodak.edu/~rahal/thesis/Thesis_Presentation.pdf

[6] Feature Selection And Representation in Classification, Henry Robinson.