Email classification for spam detection using string detection

Meenakshi kumari

Department of Computer Science & Engineering, University of Invertis, Bareilly

ABSTRACT

Nowadays, spammers change one or more characters of offensive words in their spam in order to foil content based filters. But the important thing to observe is that the spammers change the words in such a way that a human being can understand the meaning of the words without any difficulty. That’s why one might wonder why his spam filter does not detect some emails as spam when he can clearly see that those are spam! Spammers do not make any drastic change in the words so that it can be easily recognized by humans. Based on the above mentioned observations, we developed a rule based hashable technique that can match string those both look alike and sound alike, cannot be detected by conventional spam filters. E-mail content classification for spam control is proposed. It used the string matching or hash table Technique for improving the efficiency of the content based spam filter. The proposed system extract the base or stem of a misspelled or modified string, to detect spam emails. In previous also provides an Email archiving solution which classifies the E-mail relating to a person, family, corporation, association, community, or nation & showed some algorithm results to corroborate our claim.

Keywords
Spam, Filters, Bayesian, content based spam filter, Word Stemming, Email, Email archiving.

1. INTRODUCTION

Generally a content based spam filter works on words and phrases of email text and if it finds offensive content it gives that email a numerical value (depending on the content). After crossing a certain threshold, that email may be considered as SPAM. This technique works well only if the offensive words are lexically correct. That means the words must be valid words with correct spelling. Otherwise most content based spam filters will be unable to detect offensive words. In this paper, we showed that if we use some sort of string matching or data structure technique that can extract the base or stem of a misspelled or modified string, the efficiency of any content based spam filter can be significantly improved. Here we presented a simple datastructuresalgorithm specifically designed for spam detection.

2. LITERATURE SURVEY

This section gives a brief literature review of the work have done on spam classification. The Naive Bayes classifier is a simple statistical algorithm with a long chronicle of providing amazingly better results. It has been used in many spam classification studies [1, 2, 3, 4], and has become somewhat of a benchmark. It is based on Bayes' rule of conditional probability that’s by its name become naïve bayes classifier, combined with the “naive” assumption that all conditional probabilities are independent [5]. Another method for spam classification is artificial neural network (ANN) [6], normally called neural network (NN), and is a computational model or mathematical model that is inspired by the functional and structure aspects of biological neural networks. A neural network comprises of an interconnected group of artificial neurons, and it processes information using a connectionist approach for computation. In many cases an ANN is an adaptive system that modifies its structure based on internal or external information that flows through the network during the learning phase. Modern neural networks are non-linear statistical data modelling tools. To distinguish spam patterns, the neural network must first be “trained”. This training involves a computational analysis of email content using more number of samples for both spam and non-spam email messages. Essentially the network will “learn” to descry what the humans mean by “spam” and “non-spam”. To assist in this process, authors first need to have a clear and concise definition of "spam".

The KNN (k-Nearest Neighbour) classifier is an example-based classifier; it means that the classifier needs training files for comparison rather than an precise category representation, like the category profiles used by existing other classifiers. As such, there is no real training phase. When a new file needs to be categorized, the k most similar files (neighbours) are found and if a large adequate proportion of them have been assigned to a certain category, the new file is also assigned to this category, otherwise not. Additionally, finding the nearest neighbours can be accelerated using traditional indexing methods. To decide whether a email is legitimate or not, we look at the class of the emails that are closest to it. The comparison between the vectors is a real time process. This is the basic idea of the k nearest neighbour algorithm.
A GANN (Generalized Additive Neural Network) is less susceptible being perceived as a black box, because partial residual plots are generated providing graphical results that aid the spam researcher in obtaining insight about the constructed models [7]. Pattern recognition is another benefit of a GANN and can help in the classification of spam messages.

3. SUPPORT VECTOR MACHINE

Support vector machine model had been the most successful algorithm in the field of Text classification. It is mainly popular because of its ease of implementation and high accurate results. Originally it was presented to classify data into two fixed classes making it supervised non-probabilistic binary classifier. But with time it has been used by researchers for classifying data into N categories. One of the works in email classification using SVM discussed in this paper is by FagbolaTemitayo, Olabiyisi Stephen and Adigun Abimbola [8]. They have combined the SVM with the string algorithm to enhance the performance of SVM. In its simplest form SVM can be used to represent a document in vector space where each feature (word) represents one dimension. Identical feature denotes same dimension. Two of the parameters namely Term Frequency (TF) and TF-Inverse Document Frequency (TF-IDF) add value to these vectors. TF— the number of times a word occurs in a document. Harris Drucker, Donghui Wu, and Vladimir N. Vapnik proposes a word is a feature only if it occurs in three or more documents which prevents misspelled words and words used rarely. TF-IDF uses the above TF multiplied by the IDF (inverse document frequency). The document frequency (DF) is the number of times that word occurs in all the documents. The inverse document frequency (IDF) is defined as

$$\text{IDF}(w_j) = \log (\frac{||D||}{\text{DF}(w_j)})$$

(1)

4. APPROXIMATE STRING MATCHING

In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike an binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest. For the space-optimized presentation of prefix tree, see compact prefix tree.

In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a tree-shaped deterministic finite automaton. Each finite language is generated by a trie automaton, and each trie can be compressed into a deterministic acyclic finite state automaton.

Though tries are usually keyed by character strings, they need not be. The same algorithms can be adapted to serve similar functions of ordered lists of any construct, e.g. permutations on a list of digits or shapes. In particular, a bitwise trie is keyed on the individual bits making up any fixed-length binary datum, such as an integer or memory address.

A trie for keys "A", "t", "to", "tea", "ted", "ten", "i", "in", and "inn".

5. ALGORITHMS

Lookup and membership are easily described. The listing below implements a recursive trie node as a Haskell data type. It stores an optional value and a list of children tries, indexed by the next character:

```haskell
import Data.Map

data Trie a = Trie { value :: Maybe a, children :: Map Char (Trie a) }

find :: Trie a -> String -> Maybe a
find t s = find' t s (toList t)

find' :: Trie a -> String -> Trie a -> [Char] -> Maybe a
find' (_ { value = Nothing }, _) s _ = Nothing
find' (Trie v cs) s (c:cs) |
    s == c = Just v
find' (Trie v cs) s c' |
    c == c' = find' cs s (c:cs)
find' _ _ _ = Nothing

toList :: Trie a -> [Char]
toList (Trie v cs) = toList cs ++ (toList v)
```

A trie for keys "A", "t", "to", "tea", "ted", "ten", "i", "in", and "inn".
In an imperative style, and assuming an appropriate data type in place, we can describe the same algorithm in Python (here, specifically for testing membership). Note that `children` is a list of a node's children; and we say that a "terminal" node is one which contains a valid word.

```python
def find(node, key):
    for char in key:
        if char not in node.children:
            return node
        else:
            node = node.children[char]
    return node.value
```

Insertion proceeds by walking the trie according to the string to be inserted, then appending new nodes for the suffix of the string that is not contained in the trie. In imperative pseudocode,

```python
Algorithm insert(root : node, s : String, value : any):
    Node = root
    I = 0
    N = length(s)
    While I < N:
        If node.child(s[I]) != Nil:
            Node = node.child(s[I])
            I = I + 1
        Else:
            Break
            (*append new nodes, if necessary*)
    While I < N:
        Node.child(s[I]) = new
        Node
        Node = node.child(s[I])
        I = I + 1
    Node.value = value.
```

6. CONCLUSIONS

Trie and merging the common branches can sometimes yield large performance gains. This works best under the following conditions: The trie is mostly static (key insertions to or deletions from a pre-filled trie are disabled). Only lookups are needed. The trie nodes are not keyed by node-specific data, or the nodes' data are common. The total set of stored keys is very sparse within their representation space. The result of such compression may look similar to trying to transform the trie into a directed acyclic graph (DAG), because the reverse transform from a DAG to a trie is obvious and always possible. However, the shape of the DAG is determined by the form of the key chosen to index the nodes, in turn constraining the compression possible.

Another compression strategy is to "unravel" the data structure into a single byte array. This approach eliminates the need for node pointers, substantially reducing the memory requirements. This in turn permits memory mapping and the use of virtual memory to efficiently load the data from disk.

One more approach is to "pack" the trie. Liang describes a space-efficient implementation of a sparse packed trie applied to automatic hyphenation, in which the descendants of each node may be interleaved in memory.
References


