# Adapting a Vector Space Model for Feature Location in Source Code

*(Course Project for CISC 879 – Text Analysis in Software Engineering)*

William Killian
Department of Computer and Information Science
University of Delaware
103 Smith Hall
Newark, Delaware 19716
killian@udel.edu
http://www.eecis.udel.edu/~wkillian

***Abstract*—Maintenance tasks in software systems are not easy since bugs and features can exist in large code bases. A semi-automated approach is possible by using a Vector Space Model (VSM) for feature analysis. We chose to implement the VSM FLT in C++ for performance benefit compared to other FLTs (feature location techniques). We performed a case study with jEdit (an open-source Java IDE). We defined two levels of effectiveness for our evaluation and found that over 75% of the methods verified fell within the top 20% of our effectiveness. Possible improvements for our VSM could be combining with a dynamic execution system to better determine related documents to maintenance tasks, encorporating additional FLT such as Latent semantic indexing. Implementation improvements, specifically structure pools, can also be made to have the system run more efficiently.**

## I. Introduction

Maintenance tasks on software systems are difficult. It is hard to know where bugs and features exist in large codes or when developers are unfamiliar with the system. In order for a maintenance task to be completed, a developer must find where the system needs to be changed; this includes analyzing source code, specifically files, classes within those files, and methods within those classes. Instead of the developer manually looking through the source code, an semi-automatic approach is possible with a Vector Space Model (VSM) as a Feature Location Technique (FLT). The goal of using the VSM as an FLT is to aide the developers in finding where code needs to be changed based on the text of the maintenance task (usually a bug report or feature request). The VSM matches textual queries against terms scraped from the source code to identify methods found in the software system. The ideal workings of the system is to show that if a query matches a method found in the software system within a certain threshold, the likelihood of that method being a part of the maintenance task is higher and should be considered as a candidate.

The paper is organized into a few sections. Section II goes over the proposed solution. Subsections II-A, II-A, II-B, and II-C go over the Vector Space Model, implementation of the VSM, and optimization of the VSM, respectively. Section III covers the overall evaluation of the system and implementation with subsections III-A, III-B, III, and III-D describing the benchmarking of the implementation, analysis, evaluation, discussion, and verification. Section IV covers the related work, expanding on where VSM has been used before, other FLTs, and a brief listing of other work done in the field of text analysis of source code. Finally, Section V will wrap up the discussion of this work and expand with future work.

## II. Proposed Solution

A feature location technique must be developed in order to aide developers with maintenance tasks. We implemented a Vector Space Model to perform the feature location on a set of documents (methods) and queries. Below is a break-out on each component of our proposed solution.

### A. Vector Space Model

Vector Space Model (VSM) is a model for representing text documents as vectors of identifiers. Given a document, $d$, we can say that there exists a list of identifiers $i$ of size $n$. All identifiers are inserted into a vector for each document, e.g. $v_d = (x_1,\ x_2\ \cdots\ x_{n-1},\ x_n)$.

A term-document matrix, $f$, is constructed from a list of documents, each with their own list of identifiers. Any entry in this matrix $f_{i,j}$ corresponds to the count of the number of terms $j$ found in the document $i$. The term document matrix, $f$, is then transformed into a weighted matrix, $w$, by considering the maximum frequency of any term in each document $i$. This maximum frequency is defined by the equation $freq = max\,(f_i)$. The term frequency is then calculated as $tf_{i,j} = \frac{f_{i,j}}{max(f_i)}$. This term frequency is then combined with a inverse document frequency weight. This weight, $idf_j$, is defined as $ln\left(\frac{N}{df_j}\right)$ where $df_j$ is the number of documents containing the term $j$ and $N$ is the total number of documents. The resulting weighted matrix, $w$, is defined as the product of $tf_{i,j}$ and $idf_j$ for each row-column pair $(i, j)$.

Once the weighted matrix is created, the queries need to be compared with the set of documents found in the weighted matrix. The queries are each normalized similar to how the list of documents were: represented as a vector of frequencies. In order to compare a query with a document, the following equation for cosine similarity is used:

$$
\begin{aligned}
d_i &= (w_{i,1},\ w_{i,1},\ \cdots\ w_{i,M}) \\
q_j &= (w_{j,1},\ w_{j,1},\ \cdots\ w_{j,M}) \\
sim\,(d_i,\ d_j) &= \frac{d_i \cdot d_j}{\|d_i\|\|d_j\|}
\end{aligned}
\tag{1}
$$

As a final note regarding the Vector Space Model and cosine similary, the higher the cosine similarity, the more "similar" the two vectors are to each other. The range of cosine similarity goes from $[-1,\ 1]$ with values usually in the range $[0,\ 1]$ when applied to feature location since negative counts are not possible with this implementation.

### B. Implementation

The implementation of the VSM FLT is separated into four main components: input file, algorithm implementation, data representation in code, and output data. The input data for our FLT were the following files:

- Corpus method names, one per line
- Corpus methods identifiers (split and stemmed), with each line referring to a single method found in the method names file
- Corpus feature identifiers, one per line. Feature identifiers are commonly issue numbers, feature request identifiers, or link to bugs filed in a bug tracker.
- Corpus queries, one per line. Queries consist of a list of split and stemmed identifiers originating from the feature request plaintext
- GoldSet directory with a file representing the list of methods related to the feature request. This is required for proper evaluation of the FLT system

The algorithm implemented consists of the following segments:

- File processing – build initial weighted matrix and query list
- Compute cosine similarity for each query
- Construct ranked list from each cosine similarity

The Vector Space Model Feature Location Technique was implemented in C++. Because of the sparse representation of the data, unordered collections (specifically `unordered_map` and `unordered_set` ) were used to hold the matrix information. The query vectors were stored in another unordered collection to simplify the computation of cosine similarity with the documents.

The output data for the VSM is a list of documents found in each query sorted by rank. We reference this output data in section III to evaluate our system's implementation.

### C. Optimization

Besides implementing the FLT in a native-targeting language like C++, additional optimizations could be made to improve the performance of the system. First, move semantics were used; this enables large rhs expressions, specifically temporary data structures, to be moved into a permanent structure with no cost. This is done by essentially swapping internal pointers and primatives of the assigned instance with the temporary instance. Second, the queries could all be executed in parallel, so `thread` was used. Third, non-modified objects were always passed by constant reference to enable vectorization of any or all code. Finally, only strings and integers were used as keys in the unordered structures. When combined with targeting the native architecture, the restricted key types enabled hardware-accelerated hash functions to be employed. Unfortunately, these features all require the latest standard of C++, C++11.

## III. Evaluation

The implementation of the feature location using VSM is evaluated by looking at the overall ranks of the query similarities. Below, we will address the system targeting in our case study, discuss how we analyze the generated data, analyze the results gathered, a brief discussion of the results, and verification of our system.

### A. Systems and Benchmarks

We used the JEdit [1] IDE source code for our evaluation benchmark. As part of our input data, the source code had already been tokenized with identifiers split and stemmed.

### B. Data Analysis

Provided with the sorted list of documents by similarity rank for each query, we elected to analyze the data by a measure of effectiveness. We were able to compare the sorted list to those documents that were found to be in the gold (actual) set for a given query. Each one of these documents are then assigned an index, corresponding to where they are found in a given sorted list. If they are not found, a value of $-1$ is assigned. To account for potential duplicate weights for two or more documents, three measures of rank assignment were given:

1) **default** – index of document in ranked list
2) **ceiling** – last index of the document's weight in list
3) **floor** – first index of the document's weight in list

These measures of effectiveness are known as the effectiveness for all methods. Provided with this detailed effectiveness, we can reduce our data size further by only looking at the best rank found in each method. This reduced effectiveness list is known as the best method effectiveness. To clarify again, here are the two measures of effectiveness used in our system:

- **All Methods** — for each query each document is assigned an index corresponding to a function of where it exists in a ranked list
- **Best Methods** — for each query, it is a subset of **All Methods**; returns the best effectiveness value for each query

Effectiveness all methods will show a fine-grained analysis of our system while effectiveness best methods will show a more generic analysis of our system to see how well it identifies components part of a maintenance request.

| | Floor | Default | Ceiling |
|---|---|---|---|
| *Min* | 1 | 1 | 1 |
| *Q1* | 43 | 43 | 43 |
| *Median* | 133 | 133 | 133 |
| *Q3* | 554 | 554 | 554 |
| *Max* | 4971 | 6401 | 6412 |
| *Average* | 577 | 680 | 767 |
| *StdDev* | 968 | 1294 | 1571 |

Fig. 1: Effectiveness All Method of jEdit

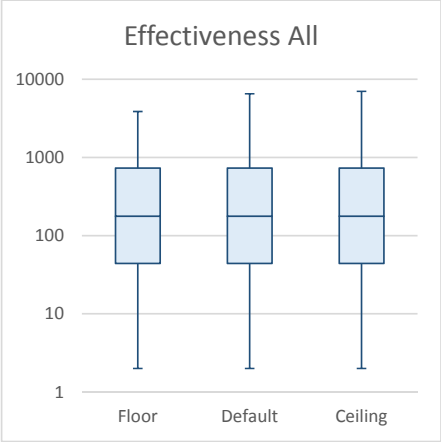


Fig. 2: Plot of Effectiveness All Method of jEdit

| | Floor | Default | Ceiling |
|---|---|---|---|
| *Min* | 1 | 1 | 1 |
| *Q1* | 8 | 8 | 8 |
| *Median* | 50 | 50 | 50 |
| *Q3* | 139 | 139 | 139 |
| *Max* | 3275 | 5926 | 6412 |
| *Average* | 193 | 230 | 248 |
| *StdDev* | 451 | 685 | 813 |

Fig. 3: Effectiveness Best Method of jEdit

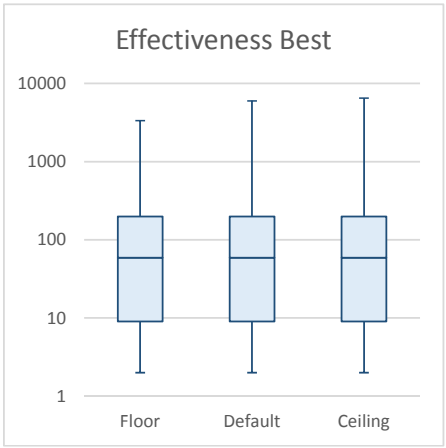


Fig. 4: Plot of Effectiveness Best Method of jEdit

### C. Results and Discussion

Figures 1 and 2 show the Effectiveness All Method of our case study evaluation with jEdit. Regardless of which indexing method used (floor, default, or ceiling) we observe that the minimum, 25% quartile, median, and 75% quartile are all the same. Consequently, the maximum, average, and standard deviation all increase as we change our indexing method. This is due to outliers either being shifted closer to the rest of the points (floor), outliers remaining where they defaulted (default), or outliers being pushed to the global maximum (ceiling).

Figures 3 and 4 show the Effectiveness Best Method of our case study evaluation with jEdit. The data trend matches that of the Effectiveness All Method. However, comparing Best Method to All Method tells us quite a bit about our sample collection. There were a total of 681 method identifiers that had a rank for the All Method effectiveness. There were 149 queries (of a possible 150) that had a best rank assigned. The data points for best method is up to 5x better than the data points collected for all method effectiveness. By pruning the higher values for each feature, the best method effectiveness better shows how effective the VSM is for each feature. The all method effectiveness shows how effective the VSM is for identifing each method of a feature. All method can be viewed as a fine-grained analysis while best bethod can be viewed as a coarse-grained analysis.

### D. Verification

Each component of the VSM FLT has been verified with other tools that have formally been verified to work. The count and distribution of the corpus information was verified using `sed`, `awk`, `wc`, and other `coreutil` applications. The query and feature vectors were verified in a similar manner.

The algorithms for term frequency, document frequency, inverse document frequency, and cosine similarity were all directly mapped to expressions. Generated output closely matched sample output provided from a third party, and any discrepancies fit within the bounds established by the different indexing functions.

The correctness and performance of the program was also analyzed using valgrind and it's built-in tools (callgrind, cachegrind, and memcheck). Cosine similarity took 64.8% of the total serial execution time. File loading and initial construction of the VSM took less than 5% of the total serial execution time. Cache analysis indicated highly optimized code for the architecture targeted. The total execution time (wall clock) was under 6 second on average while the process time never exceeded 36 seconds. The total number of instruction cache misses was less than 0.001% of total instructions executed. L3 cache hit rate was over 96% and the L1 cache miss rate was less than 0.2%. GCC 4.8.1 was used to compile the application with optimizations flags set to `-O3 -march=native`. The specification of the machine was a quad-core Intel Core i7 950 running at 3.06GHz. The RAM speed and amount is

1333MHz DDR3-SDRAM with 24GB of total memory. For file processing, a SATAIII solid-state drive (SSD) was used to provide additional bandwidth for reading into memory.

## IV. Related Work

The related work for this study is divided into two main subcategories: Vector Space Model and Feature Location. Vector space models have been used in two different recommender systems; Musto [2] used enhanced VSMs for content-based systems while Chang et. al. [3] derived web page recommendations based on Wikipedia's content. Vector space models have also been used to analyze software libraries for bug localization [4]. Conversely, VSMs with a lightweight feedback method has been used to reconstruct documents by Takano et. al. [5]. Radovanović et. al.čiteRadovanovic:2010 proposes alternative measures to improve the similarity metric between two documents. VSMs have also been adapted to domain-specific information retrieval [6]. Finally, Vector Space Models have also been used in analyzing healthcare data to make decisions [7].

There are many different feature location methods currently used. These methods can be divided into three main categories: static FL, dynamic FL, and hybrid FL. Static feature location methods statically analyze related text and documents. Dynamic feature location techniques analyze execution traces, call graphs, interactive behaviors, and other dynamic execution information. Finally, hybrid feature locations techniques employ the use of both static and dynamic feature location techniques to improve accuracy. Sinha et. al. [8] perform a case study where they determine that static FL is an appropriate technique for fault localization. Zhou et. al. [9] use static feature location on bug reports to find where bugs should be fixed in source code. Static feature location techniques have been used in the creation of a tool, FLAT3 which allows developers to find where features exist in codečiteSavage:2010. Zhao et. al. [10] performed a case study on different static feature location techniques to verify the importance of static feature location and show how much static feature location has improved. Many dynamic feature location techniques have also been used. Lament semantic indexing has been used for concept location (more broad than feature location) [11]. Shao and Smith [12] implement a FLT by using a hybrid solution with information retrival and call graph analysis. Koschke and Quante [13] perform a case study on dynamic, static, and hybrid feature location techniques. Finally, Liu et. al. [14] propose a hybrid FLT using IR techniques and a single execution trace.

## V. Conclusions and Future Work

We implemented a Vector Space Model feature location technique. We then used a corpus constructed from information from jEdit to evaluate our system. We found that 75% of all methods were within the top 20% of efficiency ranking. We verified the validity of our implementation to a reduced sample size, optimized the implementation for a target architecture, and introduced an alternative ranking for duplicate indicies. The alternative ranking for duplicate indicies caused the upper 25% of methods to drastically shift ranks, but did not affect the lower 75%. Future work can be improving the model with a different similarity metric, considering a variable effectiveness function that considers the weight in addition to the rank, or even incorporating different FLTs such as Latent semantic analysis or combining VSM with dynamic execution information.

## References


[1] (2013) jedit - programmer's text editor. [Online]. Available: http://www.jedit.org/

[2] C. Musto, "Enhanced vector space models for content-based recommender systems," in *Proceedings of the fourth ACM conference on Recommender systems*, ser. RecSys '10. New York, NY, USA: ACM, 2010, pp. 361–364. [Online]. Available: http://doi.acm.org/10.1145/1864708.1864791

[3] P.-C. Chang and L. M. Quiroga, "Deriving a categorical vector space model for web page recommendations based on wikipedia's content," in *Proceedings of the 73rd ASIS&T Annual Meeting on Navigating Streams in an Information Ecosystem - Volume 47*, ser. ASIS&T '10. Silver Springs, MD, USA: American Society for Information Science, 2010, pp. 174:1–174:2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920331.1920546

[4] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985451

[5] K. Takano and X. Chen, "A light-weight feedback method for reconstructing a document vector space on a feature extraction model," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 1169–1170. [Online]. Available: http://doi.acm.org/10.1145/1363686.1363956

[6] C. Fautsch and J. Savoy, "Adapting the tf idf vector-space model to domain specific information retrieval," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 1708–1712. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774454

[7] D. Mondal, A. Gangopadhyay, and W. Russell, "Medical decision making using vector space model," in *Proceedings of the 1st ACM International Health Informatics Symposium*, ser. IHI '10. New York, NY, USA: ACM, 2010, pp. 386–390. [Online]. Available: http://doi.acm.org/10.1145/1882992.1883048

[8] V. S. Sinha, S. Mani, and D. Mukherjee, "Is text search an effective approach for fault localization: a practitioners perspective," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, ser. SPLASH '12. New York, NY, USA: ACM, 2012, pp. 159–158. [Online]. Available: http://doi.acm.org/10.1145/2384716.2384770

[9] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 14–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337226

[10] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a static noninteractive approach to feature location," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 2, pp. 195–226, Apr. 2006. [Online]. Available: http://doi.acm.org/10.1145/1131421.1131424

[11] D. Liu and S. Xu, "Challenges of using lsi for concept location," in *Proceedings of the 45th annual southeast regional conference*, ser. ACM-SE 45. New York, NY, USA: ACM, 2007, pp. 449–454. [Online]. Available: http://doi.acm.org/10.1145/1233341.1233422

[12] P. Shao and R. K. Smith, "Feature location by ir modules and call graph," in *Proceedings of the 47th Annual Southeast Regional Conference*, ser. ACM-SE 47. New York, NY, USA: ACM, 2009, pp. 70:1–70:4. [Online]. Available: http://doi.acm.org/10.1145/1566445.1566539

[13] R. Koschke and J. Quante, "On dynamic feature location," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 86–95. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101923

[14] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 234–243. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321667