CISC 879 Text Analysis for Software Engineering

Fall 2013

Project #1

Assigned date: 09/17/2013 (Start Early)
Due date: Monday, 10/7/2013, 12:59pm

1. Objectives

- Gain first-hand experience implementing the Vector Space Model (VSM) for feature location (FL)
- Gain first-hand experience executing an evaluation study
- Learn to write up a conference-paper style evaluation study report

2. Introduction

Maintenance tasks on software systems are very challenging, especially when the software systems contain thousands or millions of lines of code, or when a developer is unfamiliar with the system. In order to complete a maintenance task, such as fixing a bug in the software or adding a new feature (i.e., functionality) to the system, a developer has to find the correct places (i.e., the correct files, classes, methods, etc.) in the system where they should make the changes. This tedious process is mostly done manually and it might involve the developer searching for key terms or browsing source code files. A semi-automatic approach to this problem that could save developers substantial effort and also make the developers more productive involves using Vector Space Model (VSM) as a Feature Location Technique (i.e., the process of finding artifacts such as methods, classes or files that are related to a particular maintenance task). The high level description of this process is presented in Figure 1. The developer uses the textual description of the maintenance task (i.e., summary of the causes or of the manifestation of the bug, or description of the new functionality that needs to be implemented in the system) as a query for the VSM Feature Location Technique. This technique matches the textual query against the terms contained in the methods of the software system and returns to the developer a ranked list of methods that contain terms most similar to the maintenance task query. The intuition is that if a query matches words from a method, the higher are the chances that the method is the one that the developer is looking for.

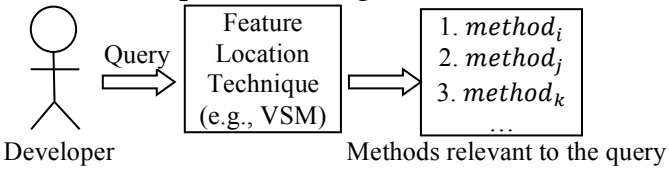

Figure 1 The semi-automated process of locating relevant artifacts in the source code using the VSM FLT

3. Vector Space Model

Vector Space Model (VSM) is a model for representing text documents as vectors of identifiers. More specifically, every text document and user query are represented as m-dimensional vectors, where m is the total number of indexed terms in the corpus (i.e., the set of documents). For example, document d can be represented as the vector $d = (x_1, \dots, x_m)$, where x_i corresponds to the "importance" of term i.

A collection of n documents can be represented in the VSM by a Term-Document matrix (see Figure 2), which has n rows corresponding to the documents and m columns corresponding to the terms. An entry $w_{i,j}$ in the matrix corresponds to the "weight" of the term j in the document i. The

weight 0 means that the term has no significance in the document or it simply does not exist in the document.

	t_1	t_2	$\cdots$	t_m
d_1	$w_{1,1}$	$w_{1,2}$	$\cdots$	$w_{1,m}$
d_2	$w_{2,1}$	$w_{2,2}$	$\cdots$	$w_{2,m}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
d_n	$w_{n,1}$	$w_{n,2}$	$\cdots$	$w_{n,m}$

Figure 2 Term-Document matrix

The Term-Document matrix is then transformed into a weighted matrix which has as values tf-idf[1] (term frequency-inverse document frequency) weights. The intuition behind using this measure is twofold. First, the more frequent a term occurs within a document, the more relevant that term is to the semantics of the document. Second, the less frequent a term occurs within all the documents, the more that term has a discriminative influence. In other words, if a term appears in almost all the documents, it should be assigned a very low importance (e.g., 'the', 'an').

The degree of similarity between a document d and a query q is calculated as the correlation between the vectors that represent them. More specifically, we use the cosine of the angle between these two vectors.

4. Example VSM

This section shows all the steps needed to compute the similarities between an external query and all the documents in a corpus.

A list of all the notations used throughout this example is presented in Figure 3.

Notation	Explanation
$f_{i,j}$	frequency of term j in document i
$\max\{f_i\}$	maximum frequency of any term in document i
$tf_{i,j} = \frac{f_{i,j}}{\max\{f_i\}}$	term frequency of term j in document i (normalized term frequency)
df_j	document frequency of term j (i.e., number of documents containing term j)
$idf_j = ln\left(\frac{N}{df_j}\right)$	inverse document frequency of term j ($where\ N$ is the total number of documents)
$w_{i,j} = tf_{i,j} * idf_j = tf_{i,j} * ln\left(\frac{N}{df_j}\right)$	tf-idf (term frequency – inverse document frequency) of term j in document i
$d_i = (w_{i,1}, w_{i,2}, \dots, w_{i,M})$ $d_j = (w_{j,1}, w_{j,2}, \dots, w_{j,M})$ $sim(d_i, d_j) = \frac{d_i \cdot d_j}{\|d_i\|\|d_j\|} =$ $= \frac{\sum_{t=1}^{M} w_{i,t} * w_{j,t}}{\sqrt{\sum_{t=1}^{M}(w_{i,t})^2} * \sqrt{\sum_{t=1}^{M}(w_{j,t})^2}}$	cosine similarity between two documents, d_i and d_j (M is the total number of terms)

Figure 3 VSM notations and their significance

Input Data:

As an input, we consider:

[1] http://en.wikipedia.org/wiki/Tf-idf

[2] http://www.jedit.org/

- the corpus presented in Figure 4, which is composed of 4 documents $(d_1, \dots, d_4)$ and 8 unique terms $(t_1, \dots, t_8)$.
- the external textual query is "$t_4\ t_7\ t_7$"

Documents	Terms
d_1	$t_2, t_2, t_3, t_3, t_3, t_4, t_4, t_4, t_4, t_6, t_6, t_6, t_6, t_8, t_8, t_8, t_8, t_8$
d_2	$t_2, t_2, t_2, t_3, t_3, t_5, t_5, t_5, t_5, t_6, t_7, t_7, t_7$
d_3	$t_1, t_1, t_1, t_2, t_3, t_3, t_3, t_3, t_3, t_4, t_5, t_6, t_6, t_6, t_7, t_7, t_7, t_7, t_7$
d_4	$t_1, t_1, t_2, t_2, t_2, t_3, t_3, t_6, t_6, t_6, t_6, t_6, t_7, t_7, t_7, t_8, t_8, t_8, t_8, t_8, t_8, t_8, t_8, t_8$

Figure 4 Example corpus

Step 1: Generate Term-Document matrix

The Term-Document matrix (see Figure 5) that models the corpus has 4 rows (each row corresponding to one document) and 8 columns (each column corresponding to one term). The elements of the matrix $f_{i,j}$ represent the frequency of term j in document i.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
d_1	0	2	3	4	0	4	0	5
d_2	0	3	2	0	4	1	3	0
d_3	3	1	5	1	1	3	5	0
d_4	2	3	2	0	0	5	3	9

Figure 5 Term-Document matrix $(f_{i,j})$

Step 2: Normalize the Term-Document matrix

The Term-Document matrix has to be normalized (see Figure 6), and each element is of the form $tf_{i,j}$ (i.e., the **term-frequency** of term j in document i). Each frequency $f_{i,j}$ is divided by the maximum frequency of any term inside a document (i.e., $\max\{f_i\}$).

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
d_1	0	0.4000	0.6000	0.8000	0	0.8000	0	1.0000
d_2	0	0.7500	0.5000	0	1.0000	0.2500	0.7500	0
d_3	0.6000	0.2000	1.0000	0.2000	0.2000	0.6000	1.0000	0
d_4	0.2222	0.3333	0.2222	0	0	0.5556	0.3333	1.0000

Figure 6 Normalized matrix $(tf_{i,j})$

Step 3: Compute document frequencies

Next we compute the document frequencies df_j of each term (see Figure 7). In other words, for each term, we count the number of documents that contain that term.

j	1	2	3	4	5	6	7	8
df_j	2	4	4	2	2	4	3	2

Figure 7 Document Frequency of term j (df_j)

Step 4: Compute inverse document frequencies

The next step is to compute the **inverse document frequency** idf_j of each term (see Figure 8) using the formula $idf_j = ln\left(\frac{N}{df_j}\right)$, where N is the total number of documents and ln is the natural logarithm.

j	1	2	3	4	5	6	7	8
idf_j	0.6931	0	0	0.6931	0.6931	0	0.2877	0.6931

Figure 8 Inverse Document Frequency of term j (idf_j)

Step 5: Generate tf-idf weighted matrix

The weighted matrix (see Figure 9) is generated by combining the **term frequencies** (see Step 2) and the **inverse document frequencies** (see Step 4). The tf-idf $w_{i,j}$ of term j in document i is computed using the formula $w_{i,j} = tf_{i,j} * idf_j = tf_{i,j} * ln\left(\frac{N}{df_j}\right)$.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
d_1	0	0	0	0.5545	0	0	0	0.6931
d_2	0	0	0	0	0.6931	0	0.2158	0
d_3	0.4159	0	0	0.1386	0.1386	0	0.2877	0
d_4	0.1540	0	0	0	0	0	0.0959	0.6931

Figure 9 Weighted matrix with tf-idf $(w_{i,j} = tf_{i,j} * idf_j)$

Step 6: Compute similarities between a query and all documents

To compute the similarity between an external query (i.e., "$t_4\ t_7\ t_7$") and all the documents in the corpus, we follow these steps:

Step 6 (a): Represent a query as a vector

First, we represent the textual query (i.e., "$t_4\ t_7\ t_7$") as a vector (see Figure 10) with the weights equal to the number of times the word appears in the query. For example, t_7 has weight 2 because it appears twice in the query, t_4 has weight 1 because it appears once in the query and the other terms have weight 0 because they are missing from the query. Note that the query vector has the same dimension as all the other documents from the weighted matrix. That dimension is the number of unique terms found in the corpus.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
Query	0	0	0	1	0	0	2	0

Figure 10 Representing query "$t_4\ t_7\ t_7$" as a vector

Step 6 (b): Compute cosine similarity

The next step is to compute the cosine similarity between the query vector and each other document which is already represented as a vector in the weighted matrix.

$$sim(d_1, Query) = 0.2794$$
$$sim(d_2, Query) = 0.2658$$
$$sim(d_3, Query) = 0.5887$$
$$sim(d_4, Query) = 0.1197$$

Step 6 (c): Generate ranked list

The final step is to sort the documents into descending order based on their textual similarities to the query (see Figure 11). This is the ranked list of documents most similar to the external query. Note that d_3 is ranked the highest because it contains both terms t_4 and t_7, whereas the other documents contain only one of these terms.

Position	Document	Similarity
1	d_3	0.5887
2	d_1	0.2794
3	d_2	0.2658
4	d_4	0.1197

Figure 11 Ranked list of documents for query "$t_4\ t_7\ t_7$"

5. Using VSM as an FLT

In order to apply the VSM as an FLT, we must construct a corpus where each document is a method of the system (for a coarser approximation, a file or a class could be used as a document).

After that, the VSM FLT will use the corpus of methods and the user query to rank the methods based on their relevance to the user query.

Generating the corpus

Figure 12 presents an overview of the process of constructing the corpus.

First, a parser extracts all the methods of a software system. The information extracted from a method includes its comments, return type, name, signature and body. Each method is associated with an identifier, called **methodID**, which consists of the full method name (i.e., package, class, method name and signature).

The second step is to preprocess the corpus extracted from the source code, using the following techniques:

- **Remove non-literals**: all the special characters are eliminated. Only the letters and the underscore character ('_') are kept (see Figure 12, next to last corpus)
- **Split identifiers**: all the compound terms are split into their basic terms using some simple heuristics, such as the camel case notation (e.g., "addNumbers") and the underscore notation (e.g., "add_numbers"). For example, these words are going to be split into the words "add" and "numbers" (see Figure 12, last corpus)
- **Lower case**: all the words are transformed into lower case letters
- **Stemming**: all the words are transformed into their root form (i.e., their suffixes are removed using some heuristics). For example, the words "documents" or "documenting" will be reduced to the form "document"

Using the corpus and the user query to generate relevant results

The corpus obtained after preprocessing will be used to construct the Term-Document matrix, which in turn is used to construct the weighted tf-idf matrix. After that, a query generated by a developer is transformed into a vector, and the cosine similarity between that query vector and all the other documents (i.e., methods) is computed. Finally, the list of methods is sorted into descending order based on their cosine similarities, and this ranked list is presented to the developer. In other words, the methods in the ranked list are the most similar methods to the query and theoretically, the top ranked methods should be the ones of most interest to the developer.

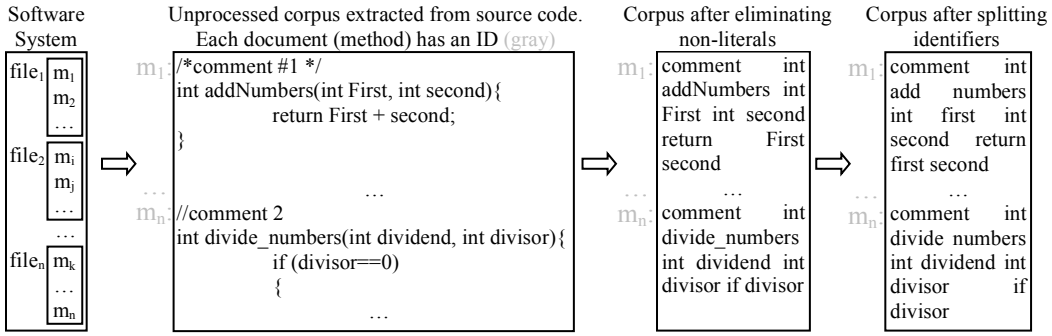

Figure 12 Overview of building the corpus

6. Evaluation

To demonstrate the usefulness of the VSM FLT, we must evaluate it. The evaluation process (see Figure 13) is only possible by having the following historical data about a software system:

- a set of maintenance tasks, such as fixing a bug or adding a new feature; in this evaluation, we call these tasks "features", and each of these features has a unique identifier, called **featureID** (see in Figure 13 the gray text f_1 (i.e., featureID #1) and f_q (i.e., featureID #q))
- textual description of the feature; in this evaluation, we call it "textual query" or simply "**query**"

- a list of methods that were modified or used in order to address the feature (i.e., either fix the bug or implement the new functionality); in this evaluation, we call this set of methods the "**gold set**"
- note that each **featureID** has associated with it a **query** and a **gold set**

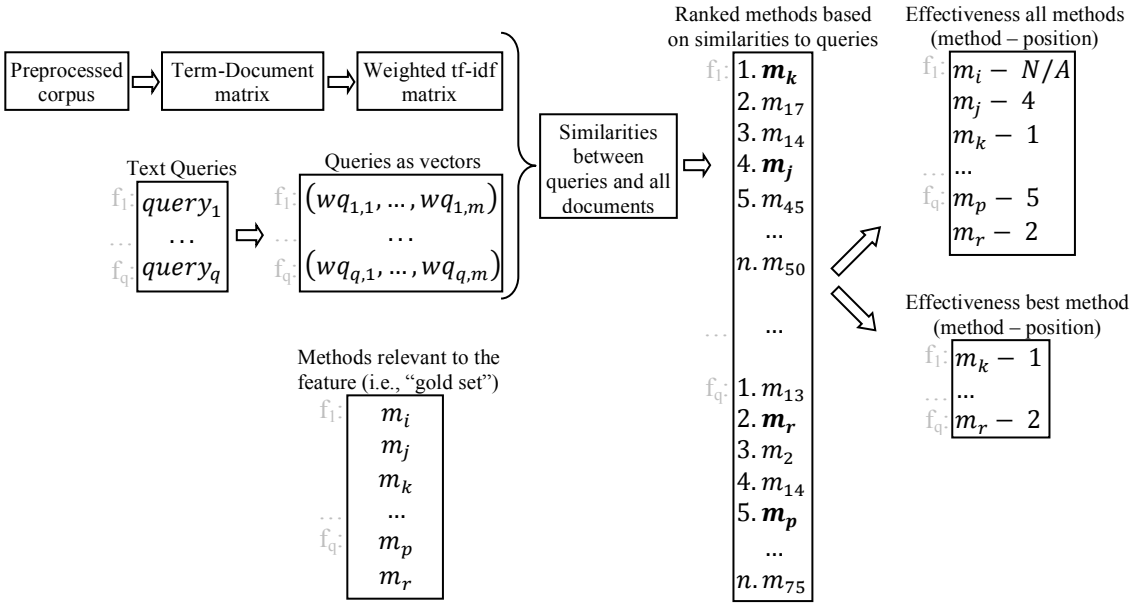

Figure 13 Overview of the evaluation process of the VSM FLT

The steps to evaluate the VSM FLT are the following:

1. From the preprocessed corpus (i.e., the corpus after removing non-literals, splitting identifiers and stemming), build a Term-Document matrix and then a weighted tf-idf matrix (see Figure 13, the three top-left boxes).
2. Transform the textual queries into their vector forms. Note that there are m unique terms in the corpus, which means that each query will have m weights (see Figure 13, the "Text Queries" and "Queries as vectors" boxes). Any other terms of the query that do not appear in the m unique terms of the corpus will be discarded.
3. For each query represented as a vector, compute the cosine similarity between that query and all the documents in the corpus. This will result in a list of document identifiers (i.e., methodIDs) and a cosine similarity which is in the $[-1, 1]$ interval. All these similarities are associated with that query, which means that they are associated with the featureID the query is associated with (see Figure 13, the "Similarities between queries and all documents" box).
4. For each feature featureID, their corresponding methods are sorted in descending order based on their similarities. This will produce a ranked list of methods that are most similar to the query associated with that feature. For example, in Figure 13, sorting the methods for feature f_1 produces the list m_k (on position 1), m_{17} (on position 2), …, m_{50} (on position n). Note that there are n documents in the corpus.
5. For each feature featureID, using the ranked list of methods most similar to a query (produced at Step 4) and the methods from the gold set which are associated with that feature (see Figure 13, the bottom-left box), we can identify the **position** of the methods from the gold set in the ranked list of all the methods. This position is called the **effectiveness** measure. The effectiveness measure can be computed in two ways:
   - **effectiveness all methods**, which are the positions of all the methods from the gold set. For example, in Figure 13, the methods from the gold set of feature f_1, which are m_i, m_j and m_k, are ranked on the following positions: m_j on position 4, m_k on position 1 and m_i does not have

a position (N/A). Note that there might be cases where a method from the gold set cannot be found in the ranked list, because the methods from the gold set and from the corpus are generated from two different sources (often independent) and some inaccuracies may occur.

- **effectiveness best method**, which is the best position (i.e., best rank) of all the methods from the gold set. For example, in Figure 13, the best position among all the methods from the gold set for feature f_1 is position 1 (for method m_k), and the best position among all the methods from the gold set for feature f_2 is position 2 (for method m_r).

It is intuitive that a method from the gold set found on a lower position (i.e., higher rank) is better than a method found on a higher position. This is because a developer who usually starts investigating the methods from the top of the list will find a relevant method in less time.

7. Project Requirements

For this project, you are required to:

1. Implement the VSM FLT
2. Evaluate the effectiveness of the VSM FLT on the jEdit[2] system, which is a popular textual editor written in Java
3. Write a report describing the VSM FLT, your evaluation, and the results.

Note that along with this project description, you will receive an archive called *Homework1AdditionalFiles.zip*, which contains the necessary files to complete this project. The list of files and their explanation is discussed in Sections 7.2 and 7.3.

1. Implement the VSM FLT

You are required to write a suite of programs that implement the functionality of using the Vector Space Model as a Feature Location Technique. You are free to use any programming language, such as Java, C, C++, C#, Python, Perl, Linux Scripts, etc.

You are required to use the SVN version control system on the mlb.acad.ece.udel.edu (see https://www.eecis.udel.edu/wiki/ececis-docs/index.php/FAQ/Subversion#toc6). Make available all your source code required to implement this assignment, and give to the instructor access to your SVN repository.

Your application should take as input data from the jEdit version 4.3 system (i.e., the data that will be provided to you). The output of your application should be in the format explained in the next section.

2. Evaluate the effectiveness of the VSM FLT

Your VSM FLT implementation should have the following input and output data.

Input Data:

The *jEdit4.3.zip* archive (from *Homework1AdditionalFiles.zip*) contains the following files:

- The file *CorpusMethods-jEdit4.3-AfterSplitStopStem.txt* contains the preprocessed corpus (i.e., after removing non-literals, after splitting identifiers and stemming the words) of the jEdit system. Each line of this file represents a document (i.e., a method).
- The file *CorpusMethods-jEdit4.3.mapping* contains the methodIDs (i.e., the identifier of the method consisting of the package name, class name, method name and signature) from the corpus. The methodID from line i corresponds to the method on line i from the file *CorpusMethods-jEdit4.3-AfterSplitStopStem.txt*.

[2] http://www.jedit.org/

- The file *jEdit4.3ListOfFeatureIDs.txt* contains the featureIDs of 150 features (i.e., maintenance tasks, such as bug fixes or features) that will be used in the evaluation. The featureIDs are represented by a unique number (e.g., "950961", "1193683", etc.).
- The file *CorpusQueries-jEdit4.3-AfterSplitStopStem.txt* contains the queries (i.e., set of words describing the maintenance tasks). Each query is associated with a unique feature. The query on line *i* is associated with the feature that has the featureID on line *i* in file *jEdit4.3ListOfFeatureIDs.txt*.
- The folder *jEdit4.3GoldSets* contains 150 files with the name *GoldSet[featureID].txt* (e.g., "*GoldSet*950961.*txt*"). The featureID is one of the featureIDs found in the file *jEdit4.3ListOfFeatureIDs.txt.* Each of the 150 files contains a list of methodIDs (same as the ones found in the file *CorpusMethods-jEdit4.3.mapping*) which are related to the feature featureID.

Note that there are 6,413 documents (methods) in the corpus, which means that there are 6,413 lines in the files *CorpusMethods-jEdit4.3-AfterSplitStopStem.txt* and *CorpusMethods-jEdit4.3.mapping*.

Note that the files *GoldSet[featureID].txt* contain some methodIDs that are not found in the *CorpusMethods-jEdit4.3.mapping* file. These discrepancies are due to inaccuracies introduced during the data collection process, which also used different sources of information. The solution to the problem of having methodIDs from the files *GoldSet[featureID].txt* that do not appear in the *CorpusMethods-jEdit4.3.mapping* file is to discard them.

Also, there are 150 features, which means there are 150 queries (in the file *CorpusQueries-jEdit4.3-AfterSplitStopStem.txt*) and 150 sets of methods associated with these features (found in the folder *jEdit4.3GoldSets*).

Output data:

Your implementation of the VSM FLT should produce a Comma-Separated Values (CSV) file that uses a tab character (\t) to separate between its values. The *.csv file will contain 5 columns, and their headers are:

- **featureID** - represents the number from the *jEdit4.3ListOfFeatureIDs.txt*file (e.g., "950961")
- **GoldSetMethodID Position** - represents the position (i.e., the line number) of the gold set method in the file *CorpusMethods-jEdit4.3.mapping*. The value -1 indicates that the gold set method does not appear in the file *CorpusMethods-jEdit4.3.mapping*
- **GoldSetMethodID** - represents the methodID of the gold set method (i.e., the name of the package, class, method and signature). This column represents the methodIDs from the *GoldSet[featureID].txt* files, and it should list these methods in the same order as they appear in the *GoldSet[featureID].txt* file
- **VSM GoldSetMethodID Rank - All Ranks** - represents the rank (i.e., position) of the gold set method in the list of methods returned by VSM, when ranking the corpus methods based on their similarities to the featureID query
- **VSM GoldSetMethodID Rank - Best Rank** - represents the best rank among all the methods from the gold set for feature featureID, when using VSM to rank the corpus methods based on their similarities to the featureID query

Figure 14 illustrates an excerpt from the output *.csv file. For example, featureID 1533473 (see column 1) has two methods in its gold set, namely org.gjt.sp.jedit.EditPane.saveCaretInfo() and org.gjt.sp.jedit.EditPane.loadCaretInfo() (see column 3). These two methods appear in the file *CorpusMethods-jEdit4.3.mapping* on positions 1,555 and 1,556 respectively (see column 2). Based on the similarities between the query 1533473 and all the other methods from the file *CorpusMethods-jEdit4.3.mapping*, these two methods are ranked on positions 376 and 181, respectively (see column 4). The best rank among these two methods from the gold set of feature 1533473 is 181 (see column 5).

Note that some values in the *.csv file are intentionally left empty (see light gray cells in columns 1 and 5 in Figure 14) in cases where a feature has more than one method in the gold set. In addition, some values in the *.csv file are intentionally left blank (see dark gray cells in column 4) if the method from the gold set is not found in the file *CorpusMethods-jEdit4.3.mapping*, in which case its corresponding value in column 2 will be set to -1. For example, for feature 1538051, method org.gjt.sp.jedit.gui.StatusBar.MemoryStatus.getToolTipText() does not appear in the file *CorpusMethods-jEdit4.3.mapping*, thus its corresponding value for column 2 will be set to -1.

You can use the values from Figure 14 to verify if your implementation produces the correct results (the lines with content "[...] " denote intentionally omitted values from the table with results).

The archive *Homework1AdditionalFiles.zip* includes the file *Sample_VSM_Effectiveness.csv*, which contains the output illustrated in Figure 14 in the format required for the assignment.

featureID	GoldSet MethodID Position	GoldSetMethodID	VSM GoldSetMethodID Rank - All Ranks	VSM GoldSetMethodID Rank - Best Rank
[...]				
1533473	1555	org.gjt.sp.jedit.EditPane.saveCaretInfo()	376	181
	1556	org.gjt.sp.jedit.EditPane.loadCaretInfo()	181	
1536064	2054	org.gjt.sp.jedit.GUIUtilities.requestFocus(Window,Component)	370	370
1538051	2776	org.gjt.sp.jedit.jEdit.showMemoryDialog(View)	2	2
	5012	org.gjt.sp.jedit.gui.StatusBar.propertiesChanged()	1805	
	5015	org.gjt.sp.jedit.gui.StatusBar.statusUpdate(WorkThreadPool,int)	978	
	5021	org.gjt.sp.jedit.gui.StatusBar.updateBufferStatus()	5716	
	-1	org.gjt.sp.jedit.gui.StatusBar.MemoryStatus.getToolTipText()		
	-1	org.gjt.sp.jedit.gui.StatusBar.MemoryStatus.paintComponent(Graphics)		
1538702	2042	org.gjt.sp.jedit.GUIUtilities.loadGeometry(Window,Container,String)	871	1
	2043	org.gjt.sp.jedit.GUIUtilities.loadGeometry(Window,String)	991	
	2049	org.gjt.sp.jedit.GUIUtilities.saveGeometry(Window,String)	1170	
	2050	org.gjt.sp.jedit.GUIUtilities.saveGeometry(Window,Container,String)	1019	
	-1	org.gjt.sp.jedit.GUIUtilities.addSizeSaver(Window,String)		
	-1	org.gjt.sp.jedit.GUIUtilities.addSizeSaver(Window,Container,String)		
[...]				

Figure 14 Sample output data and format for the *.csv file

3. Write report

You are required to write a report (minimum 2 pages and maximum 5 pages) describing:

- The motivation for the problem that you are addressing;
- The actual problem that you are addressing;
- The solution that you have implemented;
- Some details about the implementation, including some decisions or assumptions made on the input data or on the implementation
- The evaluation:
    - Design of the case study
        - Systems and benchmarks used
        - Data analysis
    - Results
        - Descriptive statistics for the **effectiveness of all methods** and for the **effectiveness best method**. These statistics include minimum, lower quartile (Q1), median (Q2), upper quartile (Q3), maximum, average and standard deviation. The statistics should be presented in a table format, as well as box-plot chart[3].
    - Discussion about the interpretation of the results

[3] http://en.wikipedia.org/wiki/Box_plot

- Conclusions

The report should use the IEEE template[4] as a format. You can edit these IEEE templates as Microsoft Word or Latex documents. However, your final report should be converted into a PDF document. You will need to submit both PDF documents and the sources (e.g., *.doc or *.tex).

The file *Homework1SampleReport.pdf* (from the archive *Homework1AdditionalFiles.zip*) is a sample paper report that contains some guidelines and some subsections that your final paper should have. You are welcome to add your own sections or subsections, or rename any of the existing sections.

You can use any statistical analysis packages to analyze and present the results (e.g., Microsoft Excel, MATLAB, R, etc.). The file *SampleBoxPlots.xls* (from the archive *Homework1AdditionalFiles.zip*) contains some sample rankings and their associated box-plots.

8. Project submission

You are required to give the instructor access to your implementation that is hosted on an SVN repository as indicated above. Your implementation should take as input the data described in Section 7.1 – Input Data, and produce the *VSM_Effectiveness.csv* file described in Section 7.2 – Output Data.

You are required to create an archive called *HW1-[LASTNAME]-[FIRSTNAME].zip*, which contains the following files:

- *VSM_Effectiveness.csv* (see Section 7.2 – Output Data, for more details and the format)
- *HW1-[LASTNAME]-[FIRSTNAME].pdf*, which is the .pdf report in IEEE format for this homework (see Section 7.3 and the file *Homework1SampleReport.pdf* for more information)
- *HW1-[LASTNAME]-[FIRSTNAME].doc* or *HW1-[LASTNAME]-[FIRSTNAME].tex*, which is the source file required to generate the *HW1-[LASTNAME]-[FIRSTNAME].pdf* report.

You should upload *HW1-[LASTNAME]-[FIRSTNAME].zip onto sakai by the deadline.*

You can access the grading rubric for this project in the project1 directory on mlb.acad.ece.udel.edu with the other files.

[4] http://www.ieee.org/web/publications/pubservices/confpub/AuthorTools/conferenceTemplates.html