Thesis Draft

Rubei Riccardo

April 24, 2018

Contents

1 Introduction

1.1 Overview

A general introduction of Crossminer, my work and how it contributes.

2 The Similarity Problem

2.1 Overview

How to solve the problem of similarity between documents, subsection about SVD, LSI and other techniques.

2.2 Mathematical Background

2.2.1 Term-Document Matrix

In Natural Language Processing [3], a term-document matrix (TDM) is used to represent the relationships between words and documents [17]. In a TDM, each row corresponds to a document and each column corresponds to a term. A cell in the TDM represents the weight of a term in a document. The most common weighting scheme used in document retrieval is the *term frequency-inverse document frequency (tf-idf)* function [15]. If we consider a set of n documents $D = (d_1, d_2, .., d_n)$ and a set of terms $t = (t_1, t_2, .., t_r)$ then the representation of a document $d \in D$ is vector $\vec{\delta} = (w_1^d, w_2^d, .., w_r^d)$, where the weight w_k^d of term k in document d is computed using the *tf-idf* function [14]:

$$w_k^d = tf \cdot idf(k, d, D) = f_k^d \cdot log \frac{n}{|\{d \in D : t_k \in d\}|} \qquad (1)$$

where f_k^d is the frequency of term t_k in document d.

Another common weighting scheme uses only the frequency of terms in documents for cells in TDM, i.e. the number of occurrence of a term in a document, instead of *tf-idf*. As an example, we consider a set of three simple documents $D = (d_1, d_2, d_3)$ as follows:

+ d_1: *She is nice.*

+ d_2: *Today is nice.*

+ d_3: *Nice is a nice city.*

The set of terms t consists of 6 elements, i.e. $t = (she, is, today, a, nice, city)$ and the corresponding term-document matrix for D is depicted in Figure 1.

TDM has been exploited to characterize software systems and finally to compute similarities between them [4], [8], [12]. In a TDM for software systems, each row represents a package, an API call or a function and each column represents a software system. A cell in the matrix is the number of occurrence of a package/an API/function in each

$$\begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \end{array} \begin{array}{c} \begin{array}{cccccc} she & is & today & a & nice & city \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 2 & 1 \end{pmatrix} \end{array} \tag{2}$$

Figure 1: An example of a term-document matrix

corresponding software system. A TDM for software systems has a similar form to the matrix shown in Figure 1 where documents are replaced by software systems and terms are replaced by API calls.

2.2.2 Cosine Similarity

Cosine similarity is a metric used to compute similarity between two objects using their feature vectors [18]. An object is characterized as a vector, and for a pair of vectors $\vec{\alpha} = (\alpha_1, \alpha_2, .., \alpha_n)$ and $\vec{\beta} = (\beta_1, \beta_2, .., \beta_n)$ there is an angle between them. Intuitively, the cosine similarity metric measures the similarity as the cosine of the corresponding angle between the two vectors and it is computed using the inner product as follows.

$$CosineSim(\vec{\alpha}, \vec{\beta}) = \frac{\sum_{i=1}^{n} \alpha_i \cdot \beta_i}{\sqrt{\sum_{i=1}^{n} (\alpha_i)^2} \cdot \sqrt{\sum_{i=1}^{n} (\beta_i)^2}} \tag{3}$$

Figure 2 illustrates the cosine similarity between two vectors $\vec{\alpha}$ and $\vec{\beta}$ in a three-dimension space. This can be thought as the similarity between two documents with three terms $t = (t_1, t_2, t_3)$.

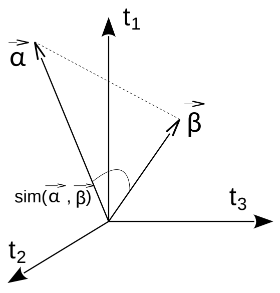


Figure 2: Cosine similarity between two feature vectors $\vec{\alpha}$ and $\vec{\beta}$

Cosine similarity has been popularly adopted in many applications that are related to similarity measurement in various domains [5], [6], [9], [11], [13]. Among the similarity metrics being recalled in this deliverable, the prevalence of Cosine Similarity is obvious as it is utilized in almost all of them as follows: *MUDABlue* [4], *CLAN* [12], *CLANdroid* [8], *LibRec* [16], *SimApp* [2], *WuKong* [19], *TagSim* [10], and *RepoPal* [20]

2.2.3 Latent Semantic Analysis

The problem with the term-document matrix is that the intrinsic relationships among different terms of a document cannot fully be captured. Furthermore, same words can be used to explain different requirements or the other way around, the same requirements can be described using different words [4]. Latent Semantic Analysis (LSA), also known as Latent Semantic Indexing (LSI), has been proposed to overcome these problems [7]. The technique exploits a mathematical model that can infer latent semantic relationships to compute similarity. LSA represents the contextual usage meaning of words by statistical computations applied to a large corpus of text. It then generates a representation that captures the similarity of words and text passages. To perform LSA on a text, a term-document matrix is created to characterize the text. Afterwards, Singular Value Decomposition (SVD) - a matrix decomposition technique - is used in combination with LSA to reduce matrix dimensionality [1]. SVD takes a highly variable set of data entries as input and transforms to a lower dimensional space but reveals the substructure of the original data. Essentially, it decomposes a rectangular matrix into the product of three other matrices as given below [1]:

$$A_{mn} = U_{mm} S_{mn} V_{mn}^T \tag{4}$$

in which

- U_{mm}: Orthogonal matrix.
- S_{mn}: Diagonal matrix.
- V_{mn}^T: The transpose of an orthogonal matrix.
- X: Low Rank matrix.

U_{mm} describes the original row entities as vectors of derived orthogonal factor values. S_{mn} represents the original column entities in the same way, and V_{mn} is a diagonal matrix containing scaling values. With the application of LSA it is possible to find the most relevant features and remove the least important ones by means of the reduced matrix U_{mm}. As a result, an equivalence of A_{mm} can be constructed using the most relevant features. LSA helps reveal the latent relationship among words as well as among passages which cannot be guaranteed by a simple term-document matrix. The similarity measurement by LSA reflects adequately human perception of similarity and association among texts. Using LSA, similarities among documents are measured as the cosine of the angle between their row vectors (see Sec. **??**). LSA has been applied in [4], [8], [12] to compute similarities of software systems. The main disadvantage of LSA is that it is computational expensive when a large amount of information is analyzed. Another very relevant issue related to LSA is the low rank approximation applied by the SVD procedure. If the singular values in S_{mn} are ordered by size, the first k largest may be kept and the remaining smaller ones set to zero. The product of the resulting matrices is a matrix X which is only approximately equal to A_{mm} , and is of rank k . It can be shown that the new matrix X is the matrix of rank k which is closest in the least squares sense to A_{mm} . The amount of dimension reduction, i.e., the choice of k , is critical to our work. Ideally, we want a value of k that is large enough to fit all the real structure in the data, but small enough so that we do not also fit the sampling error or unimportant details. The proper way to make such choices is an open issue in the factor analytic literature. In practice, we currently use an operational criterion - a value of k which yields good retrieval performance. In our we decided a k value = numer of reposotories/2 [TO BE COMPLETED]

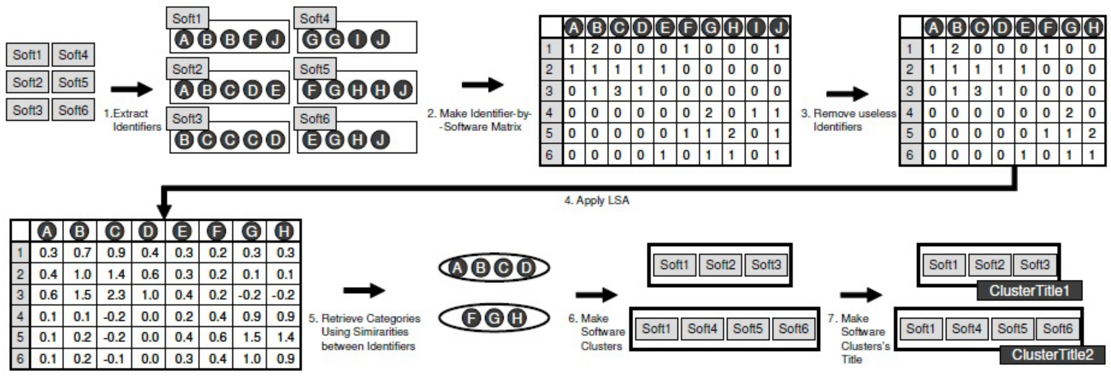
3 MUDABlue

3.1 Overview

The first procedure analysed was MUDABlue, unfortunately none implentation was available on the web, so i reimplemented it from scratch. The MUDABlue method is an automatc categorizaton method or a large collecton of software systems. MUDABlue method does not only categorize sooware systemsd but also determines categories rom the sooware systems collecton automatcally. MUDABlue has three major aspects: 1) it relies on no other information than the source code, 2) it determines category sets automatically, and 3) it allows a software system to be a member of multiple categories. Since we were interested only in the evaluation of the similarity we discarded the phases related to clusterization and categorization.

3.2 The Approach

The MUDABlue approach can be briefly summarized in 7 steps, as the following image depicts:



Soft1 Soft4
Soft2 Soft5
Soft3 Soft6
1.Extract Identifiers
Soft1 A B B F J
Soft2 A B C D E
Soft3 B C C C D
Soft4 G G I J
Soft5 F G H H J
Soft6 E G H J
2. Make Identifier-by--Software Matrix
3. Remove useless Identifiers
4. Apply LSA
5. Retrieve Categories Using Simirarities between Identifiers
A B C D
F G H
6. Make Software Clusters
Soft1 Soft2 Soft3
Soft1 Soft4 Soft5 Soft6
7. Make Software Clusters's Title
Soft1 Soft2 Soft3 ClusterTitle1
Soft1 Soft4 Soft5 Soft6 ClusterTitle2

	A	B	C	D	E	F	G	H	I	J
1	1	2	0	0	0	1	0	0	0	1
2	1	1	1	1	1	0	0	0	0	0
3	0	1	3	1	0	0	0	0	0	0
4	0	0	0	0	0	0	2	0	1	1
5	0	0	0	0	0	1	1	2	0	1
6	0	0	0	0	1	0	1	1	0	1

	A	B	C	D	E	F	G	H
1	1	2	0	0	0	1	0	0
2	1	1	1	1	1	0	0	0
3	0	1	3	1	0	0	0	0
4	0	0	0	0	0	0	2	0
5	0	0	0	0	0	1	1	2
6	0	0	0	0	1	0	1	1

	A	B	C	D	E	F	G	H
1	0.3	0.7	0.9	0.4	0.3	0.2	0.3	0.3
2	0.4	1.0	1.4	0.6	0.3	0.2	0.1	0.1
3	0.6	1.5	2.3	1.0	0.4	0.2	-0.2	-0.2
4	0.1	0.1	-0.2	0.0	0.2	0.4	0.9	0.9
5	0.1	0.2	-0.2	0.0	0.4	0.6	1.5	1.4
6	0.1	0.2	-0.1	0.0	0.3	0.4	1.0	0.9

Figure 3: MUDABlue phases.

3.2.1 Exctract Identifiers

With identifier we are talking about relevant strings that can allow to characterize a document. In this phase each repository is scanned in order to find the target files, and for each of them the identifiers are exctracted, avoiding adding useless items such as comments. The dataset was a 41C projects gathered from SourceForge.

3.2.2 Create identifier-by-software matrix

As stated before, the main item to work with is the term-document matrix, in this case we count how many times each term appears in each file for all the projects. The result is matrix **m x n** with m terms and n projects.

3.2.3 Remove useless identifiers

From the matrix we remove all the useless terms, that is all the terms that apperas in just one repository, considered a specific terms, and all the terms that appears in more than 50% of the repositories, considered as general terms.

3.2.4 Apply the LSA

Once the matrix is ready con be worked, the SVD procedure is applied and then the LSI. As explained before [NOTE] the SVD procedure decompose the original matrix in 3 other matrices. When we multiply back these matrices we use a rank reducted version of the S matrix in order to generete the final one. The authors didn't provide us any details about their final rank value, so we tested many values and eventually selected one.

3.2.5 Apply the Cosine Similarity

By using the cosine similarity method, we compare each repository vector with all the others and eventually getting an **n x n** matrix, in which is expressed the similarity of all the repository couple, with a value [0.0-1.0].

3.2.6 Categorization

The point 6 and 7 are not covered because not related to our work.

4 CLAN

4.1 Overview

CLAN (Closely reLated ApplicatioNs) [12] is an approach for automatically detecting similar Java applications by exploiting the semantic layers corresponding to packages class hierarchies. *CLAN* works based on the document framework for computing similarity, semantic anchors, e.g. those that define the documents' semantic features. Semantic anchors and dependencies help obtain a more precise value for similarity computation between documents. The assumption is that if two applications have API calls implementing requirements described by the same abstraction, then the two applications are more similar than those that do not have common API calls. The approach uses API calls as semantic anchors to compute application similarity since API calls contain precisely defined semantics. The similarity between applications is computed by matching the semantics already expressed in the API calls.

4.2 The Approach

The process consist of 12 steps here graphically reported.

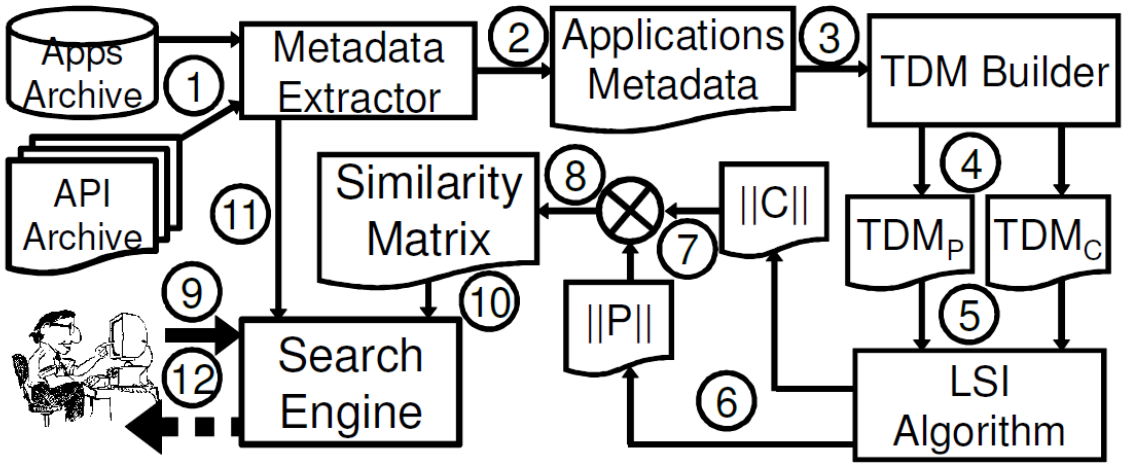


Figure 4: CLAN phases.

4.2.1 1 - 3: Terms Extraction

Steps from 1 to 3 can be merged together since are related to extraction of terms from the repositories. As stated before, an important concept is that terms extracted are only API calls, this means that all other things present in a piece of code are discarded, for example all the variables or the function declaration and invocation. Furthermore these API calls belong only to the JDK, in such a way also the calls to any other external library are discarded. This idea is also applied in the extraction of the import declaration, focus only on the JDK packages import. The result of this process will be an ordered set of data, representing the occurrencies of any Package;Class for all the projects.

4.2.2 4: TDMs Creation

Once the dataset as been created, is reorganized in TDMs. Here two different matrices are created, one for the Classes and one for the Packages. Class-level and package-level similarities are different since applications are often more similar on the package level than on the class level because there are fewer packages than classes in the JDK. Therefore, there is the higher probability that two applications may have API calls that are located in the same package but not in the same class.

4.2.3 5: LSI Procedure

4.2.4 6: Apply the Cosine Similarity

4.2.5 7: Sum of the matrices

The 2 matrices are summed, but before are multplied by a certain value. Since the values for the entries in the 2 matrices are between 0.0 and 1.0 a simple sum could result in a value over 1.0, by this multiplication these values are reducted in order to be summed togheter but still maintaining the logical meaning. The authors chosen 0.5, also we, since is a good value to equal distribute the weight of the packages and method calls.The sum of this value is 1.0, and can span from 0.1 to 0.9 for each matrix, is clear that more is high on a matrix, more is important the values that we are considering from such matrix.

4.2.6 8: Final similarity matrix

References

[1] Kirk Baker. Singular value decomposition tutorial. 2005.

[2] Ning Chen, Steven C.H. Hoi, Shaohua Li, and Xiaokui Xiao. Simapp: A framework for detecting similar mobile applications by online kernel learning. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, pages 305–314, New York, NY, USA, 2015. ACM.

[3] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, November 2011.

[4] Pankaj K. Garg, Shinji Kawaguchi, Makoto Matsushita, and Katsuro Inoue. Mudablue: An automatic categorization system for open source repositories. *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 184–193, 2004.

[5] Lan Huang, David Milne, Eibe Frank, and Ian H. Witten. Learning a concept-based document similarity measure. *J. Am. Soc. Inf. Sci. Technol.*, 63(8):1593–1608, August 2012.

[6] Aminul Islam and Diana Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Trans. Knowl. Discov. Data*, 2(2):10:1–10:25, July 2008.

[7] Thomas K. Landauer, Peter W. Foltz, and Darrell Laham. An introduction to latent semantic analysis. *Discourse processes*, 25:259–284, 1998.

[8] Mario Linares-Vasquez, Andrew Holtzhauer, and Denys Poshyvanyk. On automatically detecting similar android apps. *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 00:1–10, 2016.

[9] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, January 2003.

[10] David Lo, Lingxiao Jiang, and Ferdian Thung. Detecting similar applications with collaborative tagging. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 600–603, Washington, DC, USA, 2012. IEEE Computer Society.

[11] Ainura Madylova and Sule Gündüz Ögüducü. A taxonomy based semantic similarity of documents using the cosine measure. In *ISCIS*, pages 129–134. IEEE, 2009.

[12] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 364–374, Piscataway, NJ, USA, 2012. IEEE Press.

[13] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 775–780. AAAI Press, 2006.

[14] Juan Ramos. Using tf-idf to determine word relevance in document queries, 1999.

[15] Joel W. Reed, Yu Jiao, Thomas E. Potok, Brian A. Klump, Mark T. Elmore, and Ali R. Hurson. Tf-icf: A new term weighting scheme for clustering dynamic data streams. In *Proceedings of the 5th International Conference on Machine Learning and Applications*, ICMLA '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Ferdian Thung, David Lo, and Julia Lawall. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 182–191, Oct 2013.

[17] Peter D. Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *J. Artif. Int. Res.*, 37(1):141–188, January 2010.

[18] Amos Tversky. Features of similarity. *Psychological Review*, 84(4):327–352, 1977.

[19] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 71–82, New York, NY, USA, 2015. ACM.

[20] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. Detecting similar repositories on github. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 00:13–23, 2017.