SOURCE CODE SEARCH FOR AUTOMATIC BUG LOCALIZATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Shayan A. Akbar

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Avinash Kak, Chair

School of Electrical and Computer Engineering

Dr. Arif Ghafoor

School of Electrical and Computer Engineering

Dr. Charles Bouman

School of Electrical and Computer Engineering

Dr. Samuel Midkiff

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

Head of the School of Electrical and Computer Engineering

To my parents Inam and Tahira, and my wife Narmeen

ACKNOWLEDGMENTS

I would like to begin by thanking the Almighty Allah for the blessings and favors He has bestowed upon me, and for the endless opportunities He has granted me to pursue this PhD.

This dissertation would not have been possible without the help and support of many people who touched different aspects of my life.

I would like to express my gratitute to Professor Avinash Kak. His wealth of experience in advising research projects and his utmost support during my stay at Purdue has helped me the most in producing this dissertation. I consider myself extremely fortunate to have such an amazing advisor, mentor, and friend. I always found his advice incredibly useful in developing my technical and communication skills. I am thankful to him for helping me develop a researcher's mindset, and for teaching me how to explore new ideas for research. It's true what they say: better than a thousand days of diligent study is one day with a great teacher.

This dissertation builds upon the works of two wonderful researchers and alumni of RVL: Bunyamin Sisman, and Shivani Rao. I would like to thank them for their early contributions to the field and for their guidance in my research.

Over the years I had the opportunity of working with many amazing researchers: Tommy Chang, Noha Elfiky, Nadir Alawadi, Fangda Li, Ankit Maneriker, and Constantine Roros. I want to thank them for their valuable comments and for sharing their experiences with me.

I would also like to thank my parents Inamuddin Akbar and Tahira Darakhshan for their support in my education and for their continuous prayers for my success. I am grateful to my father for teaching me the importance of always being honest, and to my mother for her care and warmth. I am also thankful to my sisters and brother for their support and encouragement.

This PhD would not have been possible without the love and support of my wife Narmeen. I want to thank her for her understanding and encouragement throughout this long journey. I would also like to thank her parents, Sohail Ahmed and Talat Rukhsana, for their prayers and their support.

I am truly grateful to have my son, Rayan, in my life, and I hope to see him grow into a wonderful human being.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Akbar, Shayan A. Ph.D., Purdue University, December 2020. Source code search for automatic bug localization. Major Professor: Avinash C. Kak.

This dissertation advances the state-of-the-art in information retrieval (IR) based automatic bug localization for large software systems. We present techniques from three generations of IR based bug localization and compare their performances on our large and diverse bug localization dataset — the Bugzbook dataset. The three generations span over fifteen years of research in mining software repositories for bug localization and include: (1) the generation of simple bag-of-words (BoW) based techniques, (2) the generation in which software-centric information such as bug and code change histories as well as structured information embedded in bug reports and code files are exploited to improve retrieval, and (3) the third and most recent generation in which order and semantic relationships between terms are modeled to improve the performance of bug localization systems. The dissertation also presents a novel technique called SCOR (Source Code Retrieval with Semantics and Order) which combines Markov Random Fields (MRF) based term-term ordering dependencies with semantic word vectors obtained from neural network based word embedding algorithms, such as word2vec, to better localize bugs in code files. The results presented in this dissertation show that while term-term ordering and semantic relationships significantly improve the performance when they are modeled separately in retrieval systems, the best precisions in retrieval are obtained when they are modeled together in a single retrieval system. We also show that the semantic representations of software terms learned by training the word embedding algorithm on a corpus of software repositories can be used to perform search in new software code repositories not present in the training corpus of the word embedding algorithm.

1. INTRODUCTION

Over 40 million software developers contribute to the free and open-source software repositories hosted on the GitHub software development platform [1]. Various other such platforms exist, like GitLab [2], BitBucket [3], and GitBox [4], that have their own communities of software developers. Additionally, many large software-centric companies have internal tools and platforms for the development and maintenance of their proprietary software repositories.

The millions of software developers, working in companies or contributing publicly through open-source development, are distributed across the globe, and collaborate with each other using various software development platforms over the internet.

The number of software repositories publicly available on only one platform — GitHub — is in several hundred million. Many of these software repositories contain millions of source code files. And the total number of source code files hosted on GitHub is in billions.

All of the above discussion on the scale of software development activity in the world calls for the development of methods and tools for the organization and maintenance of software repositories. Tools that are capable of locating relevant source code files when provided with software search queries — especially, queries concerning localization of bugs in the software — are incredibly useful for the organization, maintenance, and evolution of large software systems.

In this dissertation, we investigate various software search tools that can localize a bug in a software repository when provided with a bug report query[1]. In order to localize a bug in the codebase, the search system locates the source code files that

[1]A bug report is a piece of document that is submitted usually by a software developer when they encounter a bug in the software system. In future discussion, we will show examples of bug reports filed for various open-source software projects.

are affected by the bug by comparing the textual content of the bug report with the textual contents of the code files.

Over the last decade or so, several bug localization techniques have been developed [5–16] that are inspired from the methods invented for information retrieval (IR), and the field has evolved to such a large scale that there is an entire IEEE/ACM conference — called the International Conference on Mining Software Repositories (MSR) — dedicated mainly to localizing artifacts in software repositories.

The traditional approach to solving the bug localization problem involves modeling software code repositories with the Bag-of-Words (BoW) assumption. The early works [5, 7–13] in the field used simple BoW models that measure the frequencies of individual bug report terms appearing in code files to rank the files according to their relevancy to the bug reports. However, since BoW based models only consider frequencies of individual query terms in source code files, they are unable to exploit any ordering or semantic relationships between the terms in the query and the terms in the code files.

Consider an example search query ‘‘`initialize model parameters`’’ for which a BoW based model, while ignoring the positions, order, and meanings of terms `initialize`, `model`, and `parameters`, retrieves files only based on the frequencies of individual terms `initialize`, `model`, and `parameters`. Whereas, for the same search query if we incorporate ordering relationships between terms in the modeling framework, the files in which the pairs of query terms — such as, `initialize` and `model`, `initialize` and `parameters`, and `model` and `parameters` — occur together in close proximity while maintaining the same order in which they appear in the query would be ranked higher than the files in which individual query terms `initialize`, `model`, and `parameters` occur. And when semantic relationships between terms are built into the search engine, the files containing terms — such as, `start`, `mod`, and `pars` — that are semantically related to the original query terms —`initialize`, `model`, and `parameters` — would be considered potentially relevant to the query in addition to the files containing the original query terms. Developing such a software

search system that models order and semantics would be incredibly useful for bug localization.

In addition to effectively modeling ordering and semantic relationships between terms, searching in a large codebase comes with the challenge of modeling software-centric information embedded in the code files and bug report. Note that the bug report queries are not regular natural language queries, rather strings that often contain source code identifiers and other structured information such as stack traces and code patches. Also, the documents present in the corpus are not regular natural language documents, rather specially structured source code files containing comment blocks and import statements among other things.

A software search system that goes beyond simple BoW modeling, and incorporates inter-term ordering and semantic relationships along with the software-centric information derived from the source code files and bug reports in the modeling procedure, is needed to enhance the search precision of bug localization. **Therefore, the first objective of this dissertation is to develop a code searching framework for automatic bug localization that incorporates semantic and ordering relationships between terms while modeling the software-centric information extracted from bug reports and code files to achieve better precision.**

In recent years, the researchers in the bug localization community have developed techniques to effectively solve the bug localization problem by incorporating software-centric information and term-term dependency relationships in the modeling procedure. However, their studies were conducted on datasets of relatively small sizes, and also only considered Java-based software repositories in the experiments.

A comprehensive large-scale study of state-of-the-art bug localization algorithms on a large and diverse dataset containing bug reports from multiple programming languages is currently absent from the existing bug localization literature. An extensive and thorough study is critical because it is not uncommon for the performance numbers produced by testing with a large dataset to be different from those obtained with smaller datasets.

Another important reason why a large-scale study is needed is because the BoW models continue to be used widely in the industry despite the availability of more advanced tools with greater retrieval accuracies. We believe that the software industry has exhibited reluctance in adopting more advanced retrieval methods largely due to the credibility gap created by the absence of a large-scale comparative evaluation.

Therefore, the second important objective of this dissertation is to develop a large and diverse dataset using bug reports filed for software repositories that are written in multiple programming languages, and subsequently perform a comprehensive comparative evaluation of state-of-the-art bug localization algorithms. We believe that the dataset will serve as a benchmark for the research community to test their bug localization algorithms. And the large-scale study is expected to encourage the adoption of advanced bug localization methods for industrial usage.

1.1 Primary Contributions

We propose the following important contributions towards addressing the objectives of this dissertation:

1. We developed SCOR (Source Code Retrieval with Semantics and Order) — a novel code search tool for automatic bug localization that jointly models semantics and order in a single retrieval framework for improved precision. SCOR combines Markov Random Fields (MRF) based term-term ordering dependencies with semantic word vectors obtained after training a word embedding algorithm, such as word2vec [17], FastText [18], and GloVe [19], on a large corpus of software repositories to model both order and semantics together.

   At the heart of the SCOR retrieval framework are two "layers" that we call the "Match Layer (ML1)" and the "Match Layer 2 (ML2)". In the form of a 2D numeric array, ML1 is simply a record of the similarities between the terms in a query and the terms in a file, with the similarities being computed

by applying the cosine distance measure to the numeric vectors produced by the word embedding algorithm, such as word2vec. Subsequently, in the spirit of convolutional neural networks, we convolve the ML1 layer with a 2×2 kernel — whose elements must possess certain pre-specified properties — to yield another 2D numeric array that is ML2. As we argue in this dissertation, the numbers in the ML2 layer become high only for those sequences of terms in the query and a file, which is being evaluated for retrieval vis-a-vis the query, when there is significant semantic similarity between the two both respect to the terms and with respect to the ordering constraints on the terms. We show that convolving the 2D array of numbers in ML1 with a 2×2 operator produces the same effect as what would be achieved with MRF based logic as presented in [6, 20].

SCOR provides improvements in the range of $7-21\%$ in terms of Mean Average Precision (MAP) when compared with the existing techniques. While there exist several bug localization algorithms that incorporate order or semantics in the retrieval framework, SCOR is the only method that combines both order and semantics into a single retrieval framework.

2. In order to train the word embedding algorithms, like word2vec, FastText, and GloVe, we created a large dataset of 35,000 Java based software repositories downloaded from GitHub. These repositories contain 35 million source code files and around 1 billion software term tokens. After training the semantic word embedding algorithms on this dataset, we obtain word vectors for half a million software terms. We also perform analysis pertaining to the quality of word embeddings produced by the algorithms, and present a comparative study of various word embedding algorithm in identifying semantically similar software terms.

3. For the purpose of performing a large-scale comparative evaluation of IR tools for bug localization we created a novel, large, and diverse Bugzbook dataset. Bugzbook contains over 20,000 bug reports belonging to 30 different software

projects written in Java, C/C++, and Python programming languages. The existing bug localization datasets are significantly smaller in size, and contain bug reports belonging only to Java-based software repositories. Bugzbook is the largest dataset for bug localization in terms of the number of bug reports, and is double the size of the second largest dataset publicly available for experimentation. Bugzbook is also the first dataset of its kind that includes significant number of bug reports belonging to software repositories written in programming languages other than Java. We believe this will encourage the researchers to extend the testing of their bug localization algorithms from Java based projects to non-Java projects.

4. We perform a large-scale study of eight retrieval algorithms on Bugzbook, and report results on their comparative performances. The results drawn from this study reveal: (1) The state-of-the-art tools for bug localization significantly outperform the traditional BoW models, (2) SCOR outperforms the previous algorithms for bug localization, (3) The word embeddings obtained after training the model on the corpus of 35000 Java software repositories are quite generic, and therefore, can be used to effectively perform search in novel software repositories not present in the training dataset even if the software repositories are written in programming languages other than Java.

5. There exist several IR models that incorporate ordering relationships between terms to enhance bug localization. These models include Markov Random Fields (MRF) [6,20], Proximity based Divergence from Randomness (PDFR) [21], and Positional Language model [22]. Unlike BoW models that model frequencies of individual bug report terms in code files to produce a relevance score, these term-term dependency models consider frequencies of sequences of bug report terms in code files to improve retrieval precision. We perform a comparative evaluation of these models in this dissertation.

1.2 Organization of the Dissertation

Chapter 2 reviews a timeline of fifteen years of research in IR based automatic bug localization. We also discuss techniques belonging to three generations of research in the field.

In Chapter 3, we discuss the features and construction of our novel, large, and diverse Bugzbook dataset. We also present statistics about the bug reports and source code files present in the Bugzbook dataset.

Traditional BoW based source code retrieval models are discussed in Chapter 4. In Chapter 5, we compare the performances of three term-term dependency models that exploit proximity and ordering relationships to enhance bug localization.

In Chapter 6, we discuss how inter-term semantic relationships can be modelled using semantic word embedding algorithms, such as word2vec, FastText, and GloVe. We also perform analysis of the quality of word vectors produced by these algorithms.

Chapter 7 introduces SCOR (Source code retrieval with semantics and order), a novel code search tool that significantly improves the performance of bug localization. SCOR combines MRF based ordering with semantic word embeddings to better localize bugs in code files.

In Chapter 8, we perform a large scale study of bug localization using our Bugzbook dataset that contains over 20,000 bug reports belonging to approximately 30 software projects. This is the largest study performed so far on bug localization. We compare the performances of eight retrieval algorithms on Bugzbook and present results in terms of retrieval precisions.

2. A TIMELINE OF PREVIOUS RESEARCH IN BUG LOCALIZATION

A timeline of important publications on the subject of automatic bug localization is presented in Figure 2.1. The figure shows around 30 papers published between the years 2004 and 2019. These publications that appeared in roughly 15 highly-respected venues — conferences and journals — belong to the three generations of software bug localization. The names of these conferences and journals are also shown in the figure.

From 2004 to 2011 — that's when the first-generation tools came into existence — one could say that research in automatic bug localization was in its infancy. The algorithms presented in [5, 23–25] laid the foundations for such tools and these were based purely on the Bag-of-Words (BoW) based assumption. Marcus et al. [23] led the way through their demonstration that Latent Semantic Indexing (LSI) could be used for concept location. Kuhn et al. [24] extended the work of Marcus et al. and presented results in software comprehension. Next came the Latent Dirichlet Allocation (LDA) based bug localization algorithm proposed by Lukins et al. [25]. To round off this series of algorithms, Rao and Kak [5] compared several early BoW based IR techniques for bug localization, and showed that simpler BoW based approaches, such as Vector Space Model (VSM) and Unigram Model (UM) outperformed the more sophisticated ones, such as those using LDA.

The second-generation bug localization tools, developed between the years 2010 and 2016 [8–12, 26–32], exploit structural information embedded in the source code files and in the bug reports as well as the software-evolution related information derived from bug and version histories to enhance the performance of BoW based systems. These studies suggest that the information derived from the evolution of a software project such as historical bug reports [8,26,27,30] and code change [10,11,31,32] history plays an important role in localizing buggy files given a bug report. These

studies also suggest that exploiting structural information embedded in the source code files [10, 12, 29, 31], such as method names and class names, and in the bug reports [9, 28, 29, 31], such as execution stack traces and source code patches, enhances the performance of a bug localization system. BugLocator [8], DHbPd (Defect History based Prior with decay) [11], BLUiR (Bug Localization Using information Retrieval) [12], BRTracer (Bug Report Tracer) [29], LOBSTER (Locating Bugs using Stack Traces and text Retrieval) [9], Amalgam (Automated Localization of Bug using Various Information), [10], BLIA (Bug Localization using Integrated Analysis) [31], and LOCUS (LOcating bugs from software Change hUnkS) [32] are some of the prominent bug localization tools developed during the second-generation.

The third and the most recent generation of bug localization tools date back to roughly 2016 when term-term order and semantics began to be considered for improving the retrieval performance of such tools [6, 15, 16, 33, 34]. For exploiting the term-term order, as for example reported in [6], these tools utilized the Markov modeling ideas first advanced in the text retrieval community [20]. And for incorporating contextual semantics, as in [15, 16, 33, 34], the tools used word embeddings based on the word2vec modelling [17] of textual data.

For the sake of completeness, it is important to point out that the organization of our evaluation study resulted in our having to leave out the contributions in two additional and relevant threads of research: (1) the query reformulation based methods for bug localization, such as those reported in [7,35] and (2) the machine-learning and deep-learning based methods [14, 15, 30, 36–39] in which a ranking model is trained to produce relevance scores for source code files vis-a-vis historical bug reports, and afterwards, the learned model is used to test the relevance of a new bug report to a source code file. We leave the evaluation of such bug localization methods for a future study.

With regards to a large-scale comparative evaluation, we are aware of only one other recent study [40] that evaluates six different IR based bug localization tools on a dataset called Bench4BL that involves 46 different Java projects that come with

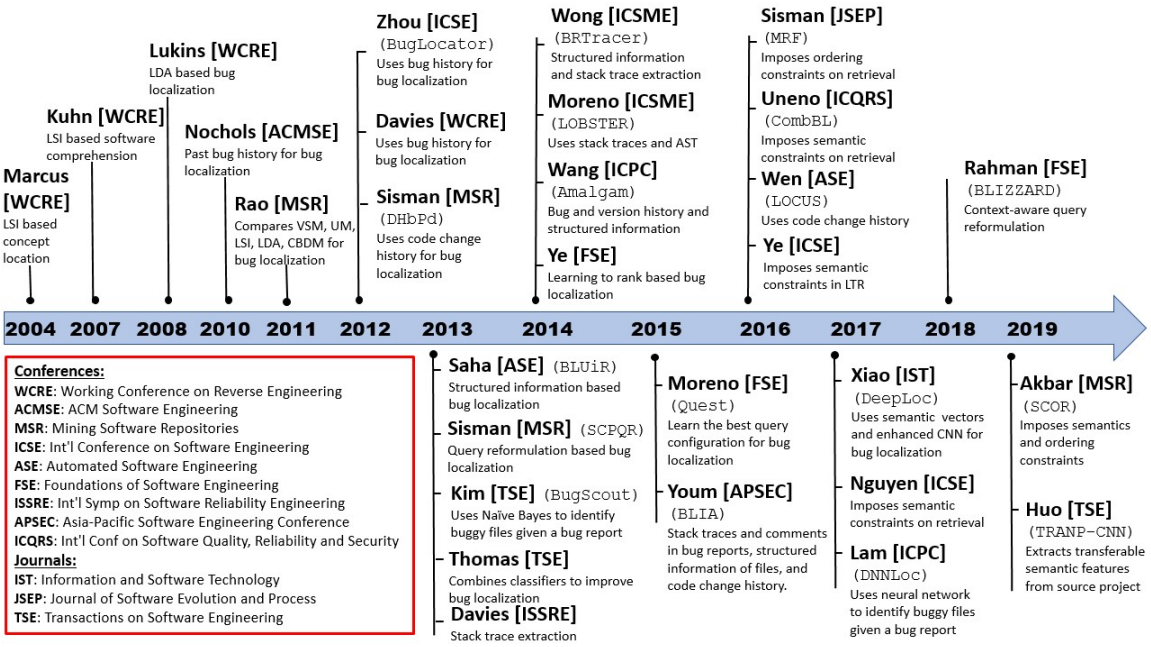


Fig. 2.1.: **A 15-year timeline of the cited publications in the field of IR-based automatic bug localization.** We represent a publication by the last name of the first author of the publication along with the venue in which the publication appeared. The abbreviation inside square brackets, for example "WCRE" in "[WCRE]", refers to the conference or the journal in which the publication appeared, while the abbreviation inside round brackets, for example "`BLUiR`" in "(`BLUiR`)" indicates the name of the tool presented in the publication. The list of publications mentioned in the timeline is, obviously, not complete and is only a representative subset of the hundreds of publications on IR-based bug localization. Notice that we have only included those publications in this timeline in which bug localization experiments were performed (with the exception of a few very early publications — that appeared before the year 2010, such as [23] — which performed experiments for concept location).

61,431 files and 9,459 bug reports. As the authors say, their work was motivated by the current "lack of *comprehensive evaluations* for state-of-the-art approaches which offer insights into the actual performance of the techniques." However, this study only covers bug localization methods from the second-generation of the tools, and therefore, does not include the important developments in bug localization made possible by the third-generation tools. That is, this study has left out the tools that incorporate term-term order and contextual semantics to enhance bug localization performance as in [6, 15, 16, 33, 34].

Additionally, note that the study carried out by Lee et al. [40] considers only Java-based software projects. On the other hand, our evaluation in this dissertation uses the Bugzbook dataset and involves eight different IR tools from all the three generations of software bug localization systems, and is based on a diverse collection of Java, C/C++, and Python based software projects that come with 4.5 million files and over 20,000 bug reports.

For yet another reason as to why we developed our own dataset and did not use the existing Bench4BL toolchain, that toolchain was designed to work only with the Jira issue tracking platform [41]. Because of our interest in cross-language effects on retrieval platforms, we also wanted to download and process the bug reports from GitHub [1]. Our Bugzbook dataset construction tool has the capability of extracting and processing bug reports and code files from both Jira as well as GitHub open-source issue tracking and software development platforms.

We should also mention the past studies by Ye et al. [42] and Thomas et al. [37] in which number of queries analysed are around 20,000 and 8,000, respectively. However, these studies are also focused mainly toward Java-based projects, and also do not consider the tools from the most recent generation of tools that include term-term order and semantics. Whereas, in this dissertation we consider C/C++ and Python projects in addition to Java projects, and also, examine the retrieval performances of novel state-of-the-art tools that consider order and semantic relationships to enhance retrieval.

3. BUGZBOOK — A VERY LARGE DATASET FOR AUTOMATIC BUG LOCALIZATION

The problem of automatic bug localization has been investigated thoroughly in the past decade or so, and various datasets have been developed and published in this regard. These datasets contain bug reports belonging to software projects publicly available on the internet. A thorough examination of these datasets reveals that (1) Their sizes are relatively small; and (2) They consisted mostly of Java-based projects.

To elaborate on the dataset sizes, at the low end, the researchers conducted experiments using datasets with just a few hundred bug reports, and, at the high end, the reported results were based on datasets with just a few thousand bug reports. The studies presented in [40], [42], and [37] are the only ones that include more than a few thousand queries to evaluate the performance of their bug localization algorithms.

Regarding the above-mentioned studies that are based on large datasets, Ye et al. [42] evaluated their bug localization algorithm on around 20,000 bug reports drawn from six Java projects. The study presented in [37] was performed on 8000 bug reports belonging to three Java and C/C++ based projects. The most recent large-scale comparative study carried out by Lee et al. [40] used around 9000 bug reports, all belonging to Java-based projects. Therefore, a large-scale bug localization dataset that contains code libraries in multiple languages is needed for a thorough evaluation of bug localization algorithms.

The goal of this chapter is to present the features and construction of a novel, large, and diverse Bugzbook dataset to overcome the shortcomings of prior datasets. A large-scale evaluation using a dataset such as Bugzbook is important because it is not uncommon for the performance numbers produced by testing with a large dataset to be different from those obtained with smaller datasets.

Table 3.1.: Comparing Bugzbook with other bug localization datasets

Dataset	**#projects**	**# bugs reports**
moreBugs	2	∼400
BUGLinks	2	∼4000
iBUGS	3	∼400
Bench4BL	46	∼10000
Bugzbook	29	∼20000

Another important reason why a comprehensive large-scale evaluation such as the one presented in this dissertation is needed is that the first-generation search tools — those based on the Bag-of-Words (BoW) assumption — continue to be used widely in industry despite the availability of the more advanced tools with greater search precision. We attribute this state of affairs to the credibility gap created by the absence of performance comparisons of the sort reported by us in this dissertation. We believe that the availability of a large-scale study will encourage adoption of state-of-the-art bug localization tools for industrial usage.

An important issue related to any large-scale evaluation is the quality of the evaluation dataset — in our case, that would be the quality of the bug reports — to make sure that the dataset does *not* include duplicate bug reports and other textual artifacts that are not legitimate bug reports. Our Section 3.2.2 describes how the raw data was filtered in order to retain only the legitimate and non-duplicate bug reports. Also, in Section 3.2.5, we present how we performed manual verification of the steps involved in the dataset construction process.

We believe Bugzbook will encourage researchers to carry out large-scale evaluations of their software retrieval algorithms. Given the size of the dataset, it may also encourage further research in deep-learning based approaches for software search. Table 3.1 compares Bugzbook with other bug localization datasets.

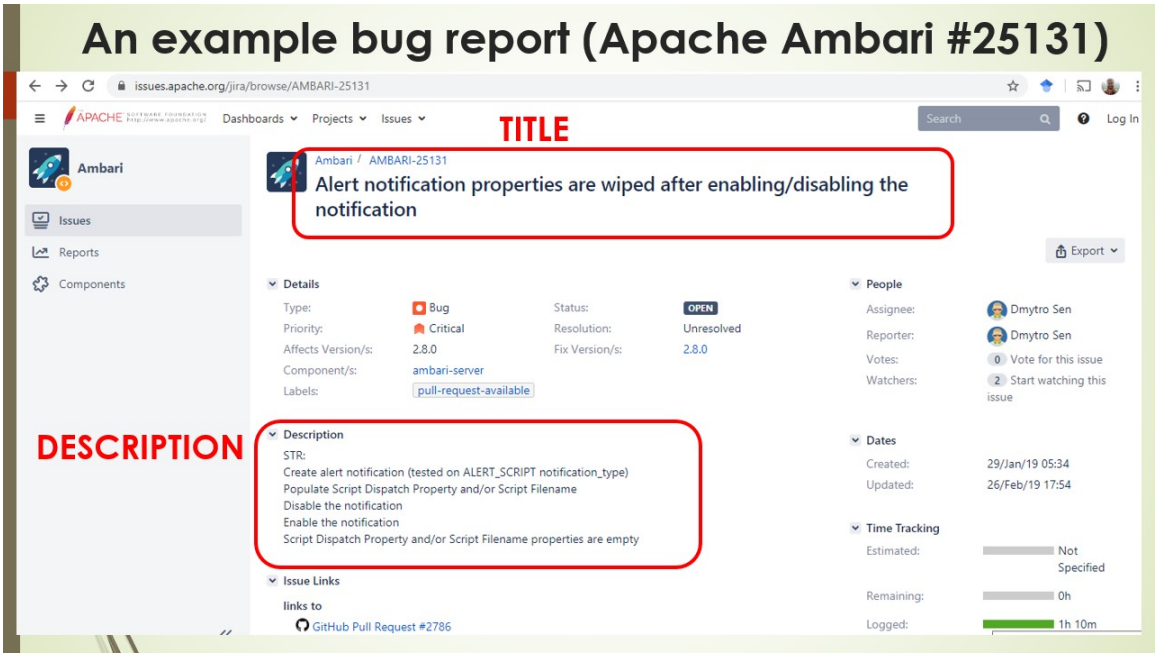


Fig. 3.1.: An example bug report taken from Apache Ambari project. Notice the two most important fields “Title” and “Description” are highlighted in the figure.

In the section that follows, we highlight some unique features of Bugzbook. In Section 3.2, we then explain the process that was used to construct this dataset. Analysis of Bugzbook is presented in Section 3.3.

3.1 Features of Bugzbook Dataset

As shown in Table 3.2, the Bugzbook dataset includes a large collection of Java, C/C++, and Python projects, 29 to be exact. The reader should note that two of the Java based projects listed in the table, AspectJ and Eclipse, were used previously in two datasets, iBugs [43] and BUGLinks [44], that have frequently been used for testing new algorithms for automatic bug localization.

Bugzbook includes several Apache projects. The reason for selecting projects from Apache is because its software developer community is believed to be the largest in

the open-source software world with regards to Java programming language. From Apache we only selected those projects for which we could find the bug reports online in the well managed Jira [41] issue tracking platform.

In addition to the Apache projects, Bugzbook also contains bug reports from other large-scale open-source projects, such as Tensorflow, OpenCV, Chrome, and Pandas. The bug reports for these projects are maintained on the GitHub platform[1].

As shown in Table 3.2, the total number of bug reports in Bugzbook is 21,253. An example bug report is shown in Figure 3.1. The total number of source-code files in all of the projects together adds up to 4,253,610. Note that the last column of the table shows the number of versions for each project. We maintain the association between the bug reports and the project versions they belong to. Additionally, we record the files that were fixed in response to the bug reports. The association between bug reports and fixed code files is important because the files that were fixed in response to a bug report serve as the ground truth relevance files in the evaluation of the bug localization algorithms. The data format used in Bugzbook for storing the bug reports is the same XML schema as used previously for BUGLinks.

3.2 How the Bugzbook Dataset was Constructed

The Bugzbook dataset was constructed from the open-source software repository archives and their associated issue tracking platforms. The Apache project archive and the associated Jira issue tracking platform would be prime examples of that.

In the material that follows in this subsection, we will address the following steps used to create Bugzbook: (1) Gathering the raw data for bug reports and source code files; (2) Filtering the raw bug reports to eliminate any duplicates and other textual artifacts; (3) Linking the bug reports with their respective source code files after the files were fixed; (4) Matching each bug report with the respective project version; and,

[1]Chrome bug reports are obtained from BUGLinks website.

Table 3.2.: Stats related to Bugzbook dataset.

Project	Description	# files	# bugs	# vers
Java projects				
Ambari	Hadoop cluster mgr	85113	2253	29
Aspectj	Java extension	6636	291	1
Bigtop	Big data manager	1291	5	5
Camel	Integration library	1229503	2308	101
Cassandra	Database mgmt tool	187150	514	133
Cxf	Services framework	768444	1795	138
Drill	Hadoop query	42360	800	17
Eclipse	IDE	12825	4035	1
HBase	Database mgmt tool	265491	2476	95
Hive	Data warehouse	114993	2221	32
JCR	Content Repository	472680	457	104
Karaf	Server-side app	63420	390	34
Mahout	Machine learning	27263	162	10
Math	Mathematics tool	16735	17	3
OpenNLP	NLP library	10250	84	11
PDFBox	PDF processor	38943	1163	35
Pig	Database manager	25462	47	11
Solr	Search server	404944	471	54
Spark	Database manager	18737	185	29
Sqoop	Database manager	7415	201	7
Tez	Graph processor	14795	177	14
Tika	Docs processor	16983	183	16
Wicket	Web app	317975	567	63
WW	Web app	72838	87	23
Zookeeper	Distr comp tool	9911	20	9
C/C++ and Python projects				
Chrome	Browser	7232	147	1
OpenCV	Computer vision tool	2865	8	1
Pandas	Data analysis tool	523	179	1
Tensorflow	Deep learning tool	10833	10	1
	Total	**4253610**	**21253**	**976**

finally, (5) Carrying out a manual verification of the dataset on randomly chosen bug reports and the corresponding source code files.

3.2.1 Gathering Raw Bug Reports and Source Code Files

Jira, the issue tracking platform for Apache, provides bug reports in XML format with multiple fields. We wrote a script that automatically downloaded all the bug reports that were marked as "FIXED" by the issue tracker and stored them in a disk file. The reason we downloaded only the fixed bug reports is because we could obtain the relevant source code files that were fixed in response to those bugs. With regard to downloading bug reports from GitHub, we used a publicly available Python script [45]. We modified the script so that it downloaded only those reports from GitHub that were explicitly marked as "closed bugs" by the report filer. This overall approach to the creation of an evaluation dataset has also been used in the past for creating some well-known datasets [6, 8, 40].

That brings us to the downloading of the source-code files. For downloading these files for the Apache projects, we wrote another script that automatically downloaded all the versions of the software projects we use in this study from the Apache archives website [46]. These software repositories were downloaded in the form of compressed ZIP or TGZ archives. The compressed files belonging to the different versions of the projects were then extracted from the archives and stored in the disk.

In addition to downloading the archives for the software projects, we also cloned the most recent snapshot of the projects from the relevant version control platforms (GitHub, GitBox, etc.) in order to obtain the most recent commit logs for the software repositories. As explained later in this section, the commit logs are used to establish associations between the bug reports and the files.

As for the Eclipse, Chrome, and AspectJ projects, we downloaded their bug reports from the BUGLinks and the iBUGS datasets that are available on the internet.

Since these bug reports relate to a single version of the project, we downloaded just those versions from the Eclipse, Chrome, and AspectJ archived project repositories.

3.2.2 Filtering the Raw Bug Reports

On the Jira online platform [47], the individual filing a report has the option to label it as belonging to one of the following categories: "task", "subtask", "story", "epic", or "bug". We made sure that we downloaded only those reports that were labeled "bug" for inclusion into our Bugzbook dataset.

On GitHub as well, the individual filing a report has the option to assign labels to the report based on pre-defined categories[2]. We select only those reports for Bugzbook that had been marked explicitly as "bug" or "Bug" by whomsoever filed the reports.

Finally, in order to avoid including duplicate bug reports in the Bugzbook dataset, we only selected those bug reports that were *not* marked as a "duplicate" of another bug report by the report filer.

3.2.3 Linking Bug Reports with Source Code Files

The most difficult part of what it takes to create a dataset like Bugzbook is the linking of the bug reports with the source code files which were fixed in response to the bug reports. This step is critical because it provides the ground truth data with which a bug localization technique can be evaluated.

The commit messages that are filed when the developers modify or fix the files play an important role in linking the bug reports with the relevant files. If a commit is about a bug having been resolved, the developer who fixed the bug includes in the commit message the ID of the bug that was fixed as a specially formatted string. For most of the projects we examined, this string is in the following format: "PROJECT-###", where "PROJECT" is the name of the software project, such as "AMBARI", and "###" is the ID of the bug report that was resolved. An example of a commit

[2]These categories are defined by the project administrators

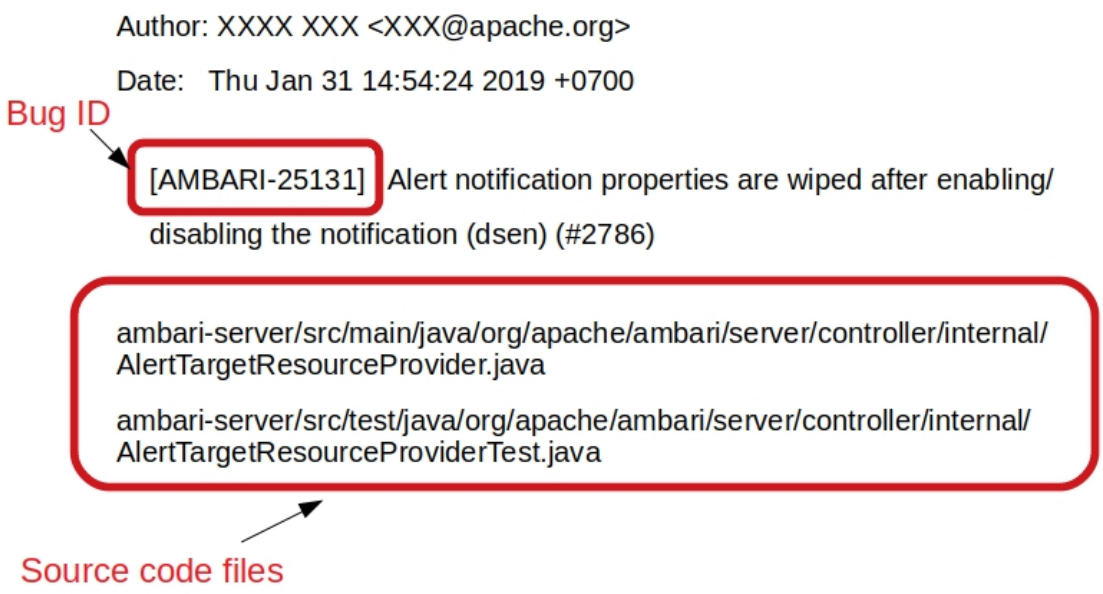


Fig. 3.2.: A commit message with bug ID and source code files highlighted in the text. The commit message is taken from the Apache Ambari project. It includes the specially structured string "[AMBARI-25131]" in the commit message. Notice that this is the commit message filed in response to the bug report shown in Figure 3.1. This means that the developer responsible for this commit notified that this commit message resolves the bug with the ID 25131. Also, the version control system records the files that were modified in response to this commit message. Therefore, the two files highlighted are the files that were fixed in order to resolve the bug 25131.

message with the bug ID and the names of the source code files fixed is shown in the Figure 3.2.

A GIT based version control system that manages a software project also attaches the names of the files that were modified in response to a bug report with the commit messages. The associations thus created between the file names and the bug reports can be used directly to link the bug reports with the relevant source code files.

Although there are advanced techniques available in the software engineering literature [48] that automatically link bug reports with source code files on the basis of textual information contained in the bug reports and the commit messages, we use the explicit method described above to establish the links between the bug reports and the files. To elaborate further, by explicit we mean that if a commit message mentions a file name along with the bug ID, then we can match up the two and form a link. Otherwise, we discard the commit message. The reason to use this explicit method for linking the bug reports with the source code files is because we want to avoid false positives in the linking process at the possible cost of incurring false negatives.

3.2.4 Versioned Associations between the Bug Reports and the Files

In much research in the past on automatic bug localization, the practice was to use only the latest version of the software library for the source code and for file identification. Bugzbook, on the other hand, maintains all of the different versions of a software project and the files relevant to a bug belong to a specific version of the project.

The bug reports often come with either the version number of the software that is presumably the source of the bug, or the version number in which the bug is fixed. If the affected version of the project that resulted in a bug is present in the bug report description, we link the bug report with the version mentioned in the report. On the other hand, if the bug report mentions the fixed version of the software, we use the version that was released prior to the fixed version as the linked version for the bug report. This obviously is based on the assumption that the version that was released prior to the fixed version contained the bug mentioned in the bug report.

3.2.5 Manual Verification of the Bugzbook Dataset

For verification of the steps described previously in this section to control the quality of the dataset, we manually check a randomly chosen small portion of the dataset by comparing the bug report entry in the Bugzbook dataset with the bug report entry in the online bug tracking platform like Jira and GitHub. In particular, we randomly selected two bug reports from each software project present in Bugzbook and manually verified its entry in the online platform. We check if the bug ID associated with a bug report in Bugzbook indeed belongs to the correct bug report in the online tracking system. We also verify all the attributes, such as title and description entries, of the bug reports. In addition to verifying the bug report entry in the online tracking system, we also verify if the bug ID associated with the bug report has a commit message associated with it in the GIT commit log, and that the fixed files mentioned in the commit log match the repaired files stored in the Bugzbook entry of the bug report.

3.3 Analysis of Bugzbook Dataset

We present our analysis of the Bugzbook dataset in this section. In particular, we show statistics related to bug reports and code files, and also present a novel measure of the level-of-difficulty of performing retrieval in different software projects.

The analysis report is present in Table 3.3. The rows of the table correspond to the software projects while the columns present the statistics about that project.

3.3.1 Bug Reports Statistics

We show the average number of bug reports present in different versions of the project, the average length of the title of the bug reports, and the average length of the description of the bug reports, in the third, fourth, and fifth columns of the table, respectively.

Table 3.3.: A thorough analysis of the Bugzbook dataset.

Project	**LoD**	**avg #bugs$_{vers}$**	**avg title**	**avg desc**	**avg #files$_{vers}$**	**avg files**
Java projects						
Ambari	1.98	75	11	40	2837	373
Aspectj	1.07	291	7	99	6630	147
Bigtop	0.47	0	11	36	143	125
Camel	1.9	20	11	133	10880	139
Cassandra	1.69	2	9	151	1127	379
CXF	1.46	11	11	158	5122	221
Drill	1.58	42	11	232	2229	329
Eclipse	1.3	4035	9	35	12825	357
Hbase	1.74	19	10	125	2074	698
Hive	1.77	69	10	175	3593	536
JCR	1.88	2	9	109	2685	272
Karaf	1.8	5	10	128	834	156
Mahout	1.27	6	9	152	1514	195
Math	0.81	1	6	114	1045	236
Opennlp	1.06	7	9	46	854	139
Pdfbox	1.57	29	8	116	973	233
Pig	0.85	2	10	96	1591	372
Solr	1.2	6	10	131	5399	324
Spark	1.77	5	9	122	520	268
Sqoop	1.4	14	11	131	529	192
Tez	1.48	11	10	63	924	458
Tika	1.27	7	9	110	679	219
Wicket	1.99	4	10	107	2717	143
WW	1.34	1	9	147	1693	189
Zookeeper	0.99	1	9	108	550	228
C/C++ and Python projects						
Chrome	0.58	147	11	138	7232	294
OpenCV	0.16	8	9	34	2399	456
Pandas	0.64	149	9	56	587	2570
Tensorflow	0.23	10	8	338	10883	238

We notice that the average length of the title remains in the range $7 - 11$ words across all projects, while the average length of the description changes significantly across all projects ranging from $34 - 338$ words.

The average number of bug reports across all versions is also not uniform for different projects. Some projects have only a few (as low as only one) bug report per version, whereas, the Eclipse project has the most number of bug reports (4035) in a single version.

3.3.2 Source Code Files Statistics

For each project we also examine the source code files present in different versions. The stats about the average number of files present in different versions of a specific project, and the average length of files are reported in the last two columns of Table 3.3.

We notice that the Bigtop project has on average only 147 code files in different versions, while Camel, Eclipse, and Tensorflow projects have over 10,000 code files on average in their different versions.

We also notice that usually the average length of code files range from 100 to 500 words. However, for the Pandas project, the average number of words in a code file is around 2500.

3.3.3 Level of Difficulty (LoD) of Different Projects in Bugzbook

When a bug localization dataset involves multiple projects, it is unlikely that all the projects would present the same level of difficulty (LoD) to a retrieval engine. So, ideally, one should weight the performance numbers for the different projects with some measure of LoD for the individual projects. We have experimented with the information-theoretic idea of Mutual Information (MI) for the source-code library and the bug reports as a measure of retrieval LoD for the library. We characterize each project by two random variables X and Y, where X represents the vocabulary in the

source code and Y represents the vocabulary in all the bug reports for that project. Now we can measure MI for any given project by

$$MI(X, Y) = H(X) + H(Y) - H(X, Y) \tag{3.1}$$

where $H(X)$ and $H(Y)$ are the marginal entropies and $H(X, Y)$ the joint entropy. Note that $MI(X, Y)$ quantifies the amount of information that the two random variables X and Y share. So the higher the value of MI for a project, the more the bug reports can tell us about the project vocabulary and vice versa.

The second column of Table 8.2 shows the calculated MI values for the software projects in Bugzbook. We observe that OpenCV project has the least MI value, while the project with largest MI value is Wicket.

In Chapter 8, we will perform an analysis on the relationship between MI values and the performances of retrieval algorithms on different software projects.

4. TRADITIONAL BAG-OF-WORDS BASED SOURCE CODE RETRIEVAL MODELS FOR BUG LOCALIZATION

Now that we have discussed the components of a bug localization dataset and how one can go about creating one, in this chapter, we will start our discussion of retrieval techniques used for bug localization with the traditional approach of BoW based modeling [5, 7–13].

In BoW based modeling the relevance score for a file to a query is computed based on the frequencies of individual query terms appearing in the code files. Afterwards, the files are ranked according to their relevance scores and presented as output to the user.

There are several methods for BoW modeling of software code repositories for bug localization. In this dissertation we will focus on the Dirichlet Language Model (DLM) and Term Frequency Inverse Document Frequency Model (TFIDF). These two serve as the baseline models for evaluating the performances of more advanced models we will discuss later in this dissertation.

In what follows, we will review DLM and TFIDF. We will derive the formulas for their relevance scores and provide rationale behind the heuristics used in developing these models.

4.1 Dirichlet Language Model (DLM)

The task of retrieving code files given search queries, such as bug reports, can be interpreted as a language modeling problem. In that, a code file is considered a good match to a bug report query if the model of the file is likely to generate the query, which in turn will happen if the query terms appear frequently in the code file.

Given a query $Q = q_1, q_2, ..., q_n$, and a code file $f = f_1, f_2, ..., f_m$, the goal is to measure the conditional probability $P(f|Q)$, i.e., the probability that the file f generates the observed query Q.

Using the Bayes' rule:

$$P(f|Q) = \frac{P(Q|f)P(f)}{P(Q)} \tag{4.1}$$

The denominator $P(Q)$ is dropped because it's a constant, and therefore, does not contribute to the ranking of files, and the term $P(f)$ is dropped because it is assumed to be uniform.

Therefore, the retrieval model reduces to the computation of $P(Q|f)$, which is the multinomial model:

$$P(Q|f) = \prod_i P(q_i|f) \tag{4.2}$$

Clearly, we assumed independence between query terms q_i in the above unigram model. This independence assumption is at the heart of Bag-of-Words modeling. Taking the logarithm on both sides to overcome the problem of numerical underflow:

$$\log P(Q|f) = \sum_i \log P(q_i|f) \tag{4.3}$$

Given a term q_i from the vocabulary of the collection C, the maximum likelihood estimate is given by:

$$P(q_i|f) = \frac{tf(q_i, f)}{\sum_{w \in f} tf(w, f)} \tag{4.4}$$

where $tf(w, f)$ is the frequency of term w in file f, and $\sum_{w \in f} tf(w, f)$ is simply the total number of terms in the file f.

To resolve the issue of zero frequencies when a query term is not present in the file, we need to perform smoothing of the probability distribution. Using Bayesian Smoothing with Dirichlet Priors [49], the file likelihood of a term is given by:

$$P(q_i|f) = \frac{tf(q_i, f) + \mu P(q_i|C)}{\sum_{w \in f} tf(w, f) + \mu} \tag{4.5}$$

where μ is the smoothing parameter. The above equation is the equation of the Dirichlet Language Model (DLM) for code retrieval.

4.2 Term Frequency Inverse Document Frequency (TFIDF)

In [50], Robertson and Jones presented a BoW model for text retrieval called the Term Frequency Inverse Document Frequency Model (TFIDF). In this section we show how TFIDF can be used for bug localization.

The TFIDF model is based on two important measurements: (1) the number of times a query term q_i appears in the file f, and (2) the document frequency of the term q_i. Here, the document frequency of the term q_i refers to the number of files that contain the term q_i.

The rationale for having the document frequency in the equation for score calculation is that the terms present in only a few documents are often more valuable than the ones that occur in many.

We define the inverse document frequency weight for a term q_i as:

$$IDF(q_i) = \log N_f - \log df(q_i) \tag{4.6}$$

where, N_f is the number of code files in the repository, and $df(q_i)$ is the document frequency of the term q_i.

The score function for TFIDF is given as follows:

$$TFIDF(Q, f) = \sum_{q_i \in Q} \frac{IDF(q_i) * tf(q_i, f) * (K + 1)}{K * ((1 - b) + (b * |f|)) + tf(q_i, f)} \tag{4.7}$$

where, K and b are parameters that require tuning, and $|f|$ denotes the total number of terms present in the code file.

5. MODELING TERM-TERM ORDERING RELATIONSHIPS FOR SOURCE CODE RETRIEVAL

In the previous chapter, we have discussed how simple BoW models that are based on measuring frequencies of individual query terms in code files, can be used for automatic bug localization. Because of their simplicity and reasonable effectiveness, the usage of BoW models is quite prevalent in the industry. The academic researchers in the field have also found them useful for quite some time. Several BoW based source code retrieval methods exist in the literature of mining software repositories [5, 7–12, 29].

With regard to the BoW based methods in the literature, Sisman and Kak [7] introduced a BoW model called SCP-QR which is a Query Reformulation method based on the Spatial Code Proximity of non-query terms with the query terms inside the source code files. Zhou et al. [8] proposed BugLocator which takes into account past bug history to identify similar bugs. The relevance of a file to a bug report is determined using simple BoW based retrieval combined with an analysis of the source code files that were fixed for past similar bug reports. Moreno et al. [9] and Wong et al. [29] exploited the stack trace information present in the bug reports to better localize the bug. Sisman and Kak [11] incorporated version history in information retrieval based bug localization for source code search. BLUiR (Bug Localization Using information Retrieval) was introduced by Saha et al. [12] in which different components of a source code file — classes, methods, variables, and comments — are assigned different weights based on their importance in the retrieval process. Rao and Kak [5] showed that for the purpose of retrieval from software libraries for bug localization the simplest BoW models, such as those based on Vector Space Model (VSM), Simple Unigram Model (SUM), etc., are much more effective than the more complicated ones like Latent Dirichlet Allocation (LDA).

Notice that since all of the above mentioned models are based on the BoW assumption and, therefore, only consider the frequencies of query terms in source code files, all positional, ordering and semantic relationships between the terms are lost. In this chapter, we will focus only on the positional and ordering relationships between terms for improved retrieval. As for the inter-term semantic relationships, we will discuss them in later chapters.

Several researchers in the IR community have developed retrieval techniques [20–22, 51] that, in addition to considering the frequencies of the terms, also take into account the term-term positional and ordering relationships. The earliest of these *term-term dependency models*[1], by Metzler and Croft [20], uses the notion of Markov Random Fields (MRF) to generate a second-order probabilistic model for a corpus, which is in contrast to the first-order probabilistic models that are generated by the BoW assumption.

The Weighted Sequential Dependence (WSD) model by Bendersky's et al. [51] is an extension of MRF based modeling in which weights are assigned to query terms and their combinations based on their importance to the retrieval engine.

Another extension of the basic MRF modeling is by Peng et al. [21]. Their algorithm, known as Proximity Based Divergence From Randomness (PDFR), forgoes Dirichlet smoothing for the corpus terms that are absent in a query and instead uses DFR based weighting for such terms as advocated in [52].

Finally, we have the Positional Language Model (PLM) by Lv and Zhai [22] that is very different from MRF. In PLM, the significance of a term in the model is defined by the proximity-distance-weighted counts associated with all other terms in the corpus.

The fact that these *term-term dependency models* greatly improve the retrieval precision is now well established in the literature [20–22, 51, 53, 54]. In this chapter we compare these term-term dependency models with the simple BoW based models for solving the problem of automatic bug localization. We also carry out a com-

[1]We will refer to the retrieval techniques that incorporate term-term ordering relationships in the modeling procedure as the term-term dependency models in this chapter.

parative study of the different term-term dependency models — MRF, PDFR, and PLM — using the open-source search engine Terrier[2]. We have extended Terrier by incorporating in it the implementations for the WSD [51] and PLM [22] models.

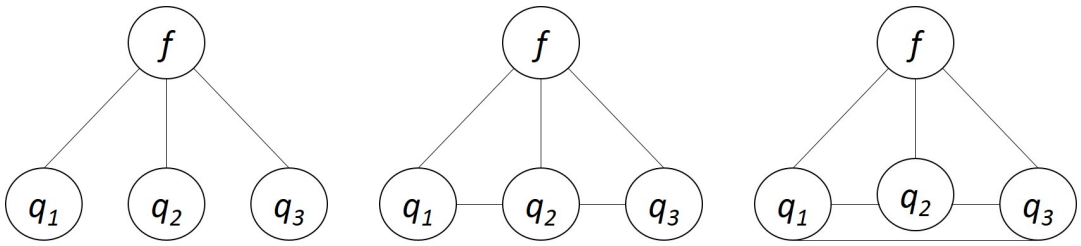
In this comparison, we use the same software-centric query conditioning (QC) for all models. The query conditioning step can easily be turned off and on in our extension of Terrier. We report comparative results with QC on and off. For our comparative investigation, we use approximately 4000 bug reports of Eclipse software library obtained from the BUGLinks dataset and approximately 300 bug reports of AspectJ obtained from the widely popular iBUGS dataset. Notice that BUGLinks and iBUGS datasets are also a part of our large-scale novel Bugzbook dataset discussed in Chapter 3.

We report comparative results based on retrievals with (1) just the titles of the bug reports as queries; and (2) the entire description of the bug reports as queries. With both software libraries, we show that all three term-term dependency models significantly outperform the BoW based retrievals by as much as 5.2% to 26.4%.

When comparing the term-term dependency models with the more advanced BoW based techniques — BugLocator, BLUiR, and SCP-QR — we see that the term-term dependency models outperform BugLocator and SCP-QR, but fails to outperform the BLUiR model. Notice that BugLocator requires a history of all bug reports that are filed for the software. Whereas no such history is used in our methods. Also, the BLUiR model analyses the structured information embedded in code files for better retrieval, whereas no such analyses is performed in the term-term dependency models. Additionally, SCP-QR is a query reformulation based method, whereas term-term dependency models are executed without the reformulation of query.

Our experiments also demonstrate that not all three term-term dependency models are equally effective in improving retrieval precision. We show that MRF consistently performs well in all our experiments, and MRF and PDFR are comparable in their superiority over BoW, with both outperforming PLM in all but one experiment

[2]http://terrier.org



(a) Full Independence (b) Sequential Dependence (c) Full Dependence

Fig. 5.1.: *Using a three-term query as an example, illustrated here are the three term-term dependency assumptions for MRF modeling of a software library.*

by around 10%. PLM, however, did perform better than PDFR in one experiment but not better than MRF. Therefore, our comparative study also concludes that out of the three term-term dependency models MRF is the most reliable model to use. Therefore, in the later chapters of this dissertation, we will build semantics-based retrieval engine on top of the variants of MRF model.

The organization of this chapter is as follows. In the next section we review the three term-term dependency models, these being MRF, PDFR, and PLM. Subsequently, in Section 5.2, we discuss how we use software-centric Query Conditioning before using the queries for retrieval. Notice that the same preprocessing of bug reports as described in this chapter is used throughout this dissertation for more advanced methods. Experimental results are reported in Section 5.3.

5.1 Three Term-Term Dependency Models

In this section, we briefly review the three term-term dependency models for IR based retrieval.

5.1.1 Markov Random Field

The genesis of Markov Random Field (and the closely related Bayesian Belief Networks) lies in attempts at making probabilistic inferences from a network of nodes, with each node representing a variable that depends on some or all of the other nodes. The dependency between the variables at the different nodes may be expressed in the form of conditional or joint probabilities. Kollar and Friedman [55] present a thorough overview of all such graphical models for probabilistic inference.

Metzler and Croft [20] were the first to recommend that MRF be used for retrieving information from text corporas. Subsequently, it was shown by Sisman et al. [6] that when MRF modeling for incorporating term-term dependencies is combined with query conditioning specific to software retrieval, one obtains a powerful retrieval engine for automatic bug localization.

In the context of IR based bug localization, a Markov Random Field is an undirected graph G in which one of the nodes represents a source-code file f that is being evaluated for its relevance to a given query Q and all other nodes represent the individual terms $Q = \{q_1, q_2, ..., q_{|Q|}\}$ in the query. The arcs between the nodes represent probabilistic dependencies between the nodes. We measure the relevance of file f to query Q by computing the conditional probability $P(f|Q)$. Since

$$P(f|Q) = \frac{P(Q, f)}{P(Q)} \tag{5.1}$$

and since, the denominator shown above, being independent of f, would remain the same for all the files in a repository, the relevance of a file f to the query Q can also be measured by estimating the joint probability $P(Q, f)$. That is, we can say $P(f|Q) \stackrel{rank}{=} P(Q, f)$.

A most interesting property of a network of probabilistic nodes in a graph G is regarding the necessary and sufficient condition that must be satisfied by any joint probability distribution over the nodes: Any normalized product of nonnegative

values assigned to the cliques in G is a legitimate value for the joint distribution.[3] A set of nonnegative values for a clique can be thought of as defining a potential function for that clique. Using this property, the joint distribution $P(Q, f)$ in the graph G formed by f and the terms in Q can be expressed as

$$P(Q, f) = \frac{1}{Z} \prod_{k=1}^{K} \phi(C_k) \stackrel{rank}{=} \sum_{k=1}^{K} \log(\phi(C_k)) \tag{5.2}$$

where $C_k = \{C_1, C_2, ..., C_K\}$ represents the set of cliques in graph G, and $\phi(C_k)$ a nonnegative potential function associated with clique C_k. The normalization factor Z is usually ignored since the goal is to rank the files according to their relevance to the bug report terms. Note that we express the potential functions in logarithmic form for computational ease.

That leads to the question of how to actually determine the potential functions for the different cliques in G. The answer depends on what sort of assumptions we want to make regarding the probabilistic dependence of the query terms on one another and on the file node f. With regard to such assumptions, in the following subsections we will discuss four different ways of using the MRF model for bug localization: the full independence assumption (FI), the sequential dependence assumption (SD), the full dependence assumption (FD), and the weighted sequential dependence assumption (WSD).

For the purpose of explaining the basic ideas of what goes into calculating $P(Q, f)$ in Equation (2), we will assume that the query Q has just three terms (q_1, q_2, q_3). In what follows, our elaboration of the above mentioned dependency assumptions are based on the three subfigures in Figure 1.

[3]This legitimacy holds in the same as: Any nonnegative assignment of numbers to the different outcomes of a random variable X is a legitimate probability distribution for X provided, of course, the numbers add up to 1. By the way, the normalization constraint applies also to the nonnegative numbers assigned to the cliques in a graph G, as should be evident by the normalization in Equation (5.2).

Full Independence (FI, BoW)

When using MRF modeling, the simplest assumption to make is that of full independence (FI), which amounts to the old BoW assumption that has already been investigated extensively for bug localization [5, 7–12, 29]. In particular, the FI BoW model is equivalent to the Dirichlet Language Model (DLM) discussed in Chapter 4.

Examining MRF with FI serves two important purposes: (1) It will help us clarify how the potential functions needed in Equation (2) are calculated from the source code files for the simple case of FI. Subsequently, it will be easier for the reader to see the generalization of those explanations to the other term-term dependency assumptions. And, (2) It will provide us with a baseline for comparing the BoW retrieval performance with those from the other term-term dependency assumptions.

Figure 5.1a illustrates the FI assumption for the simple case of three-term query. Notice the absence of any arcs between the nodes that stand for the query terms. Naturally, FI implies that the graph G will consist of just 2-nodes cliques, with each clique focusing on just one query term and its importance to the file f. The most straightforward way to set the potential for each clique is to make it proportional to the frequency of the query term in the file.

The ploy of setting the potential of each clique to the frequency of the corresponding term in file f only works well for those query terms that are actually present in the file. So what about the query terms that are absent in f? In MRF modeling, the cliques formed by those query terms are taken care of through Dirichlet smoothing [49].

As discussed in Chapter 4, Dirichlet smoothing means that, for a query term q that is not present in the file whose relevance to the query is being tested, we make the potential for the corresponding clique proportional to the *collection frequency* for q. If we represent the entire software library as the collection C, we can now write the following equation for the clique potentials for the FI case:

$$\log(\phi_{fi}) = \log \frac{tf(q_i, f) + \frac{\mu_{fi} tf(q_i, C)}{|C|}}{(|f| + \mu_{fi})} \tag{5.3}$$

where $tf(q_i, f)$, and $tf(q_i, C)$ are, respectively, the frequency of the term q_i in the source code file f and in the collection C. The notations $|f|$ and $|C|$ stand for the size of file and the collection, respectively. Finally, μ_{fi} is the smoothing parameter. Therefore, the joint distribution $P_{fi}(Q, f)$ to compute the relevance score for a file f for the case of FI is given by:

$$P_{fi}(Q, f) = \sum_{i=1}^{|Q|} \log \frac{tf(q_i, f) + \frac{\mu_{fi} tf(q_i, C)}{|C|}}{(|f| + \mu_{fi})} \tag{5.4}$$

Sequential Dependence (MRF SD)

Figure 5.1b illustrates the sequential dependence assumption for the simple case of a three-term query Q. Notice the arcs that connect the query term nodes sequentially. The SD model is a result of the dependency assumption that given a file f and the full query Q, how an individual query term q_i depends on those two can be expressed as: $P(q_i|f, q_1, q_2, ..., q_{|Q|}) = P(q_i|f, q_{i-1}, q_{i+1})$. For the three-term example shown in Figure 5.1b, this implies that we place an arc between the nodes q_1 and q_2, and between the nodes q_2 and q_3. Those two arcs would be in addition to the arcs between the file node f and the term nodes q_1, q_2, and q_3. Consequently, we now have 3-node cliques in addition to the 2-nodes cliques of the FI model.

While we can continue to use Equation (3) for 2-node clique potentials in Figure 5.1b, the 3-node clique potentials can be made proportional to the number of times consecutively occurring pairs of query terms occur together in a file. Given a pair of query terms q_i and q_{i+1} in a graph such as the one shown in Figure 5.1b, all that the SD assumption requires is that, when we count the number of times the pair (q_i, q_{i+1}) occurs in a file, we ensure that the term q_{i+1} comes *after* the term q_i in the file without requiring that the two terms be adjacent. For practical reasons, this requires using a

Table 5.1.: Features used by WSD variant of MRF

Feature	Description
c_1	constant value for terms (1.0)
$tf(q_i, C)$	collection frequency of term q_i
$df(q_i)$	document frequency of term q_i
c_2	constant value for pair of terms (1.0)
$tf(q_i q_{i+1}, C)$	collection frequency of pair of terms $q_i q_{i+1}$
$df(q_i q_{i+1})$	document frequency of pair of terms $q_i q_{i+1}$

window of size w in which to look for q_{i+1} after we have encountered q_i while scanning through the file f. The window size w is a tunable parameter for the SD assumption.

Denoting pair of terms $q_i q_{i+1}$ by ρ the clique potentials for the three-node cliques in the SD assumption are given by:

$$\log(\phi_{sd}) = \log \frac{tf_w(\rho, f) + \frac{\mu_{sd} tf_w(\rho, C)}{|C|}}{(|f| + \mu_{sd})} \tag{5.5}$$

where the expressions $tf_w(\rho, f)$, and $tf_w(\rho, C)$, respectively, are the frequencies of the pairs of terms $\rho = q_i q_{i+1}$ in file f and in the collection C. The notation μ_{sd} stands for the smoothing parameter.

Combining these with the potentials for the 2-node cliques, we can write down the following formula for the joint distribution $P(Q, f)$ for the case of SD:

$$P_{mrf,sd}(Q, f) = \lambda_{fi} \sum_{i=1}^{|Q|} \log(\phi_{fi}) + \lambda_{sd} \sum_{i=1}^{|Q|-1} \log(\phi_{sd}) \tag{5.6}$$

Note that in Equation (5.6) the weight parameters λ_{fi} and λ_{sd} control the relative importance of the 3-node cliques vis-à-vis the 2-node cliques. We set the values of these parameters such that $\lambda_{fi} + \lambda_{sd} = 1$.

Full Dependence (MRF FD)

The fully connected graph in Figure 5.1c illustrates the FD assumption. In general, this graph implies that a file f is to be considered relevant to a query when all possible permutations of the query terms occur with frequencies that are proportional to their frequencies in the query itself. However, since the number of permutations of the query terms grows exponentially with the size of the query, any practical implementation of FD must place a limit on the length of the permutations considered. For example, it is common to only consider 2- and 3-node cliques for the FD assumption.[4] And when limiting oneself to 3-node cliques, one usually also places a constraint on the order in which the terms appear in a query vis-a-vis their order in the file.

Denote again a pair of terms $q_i q_j$ by ρ, the clique potentials for FD model can be calculated as follows:

$$\log(\phi_{fd}) = \log \frac{tf_w(\rho, f) + \frac{\mu_{fd} tf_w(\rho, C)}{|C|}}{(|f| + \mu_{fd})} \tag{5.7}$$

where, as for the SD case, $tf_w(\rho, f)$, and $tf_w(\rho, C)$, are, respectively, the frequencies of the pair of terms $\rho = q_i q_j$ in file f and in collection C. The notation μ_{fd} is the smoothing parameter.

Now the joint distribution for FD case, $P(Q, f)$ can be expressed in the following form:

$$P_{mrf,fd}(Q, f) = \lambda_{fi} \sum_{i=1}^{|Q|} \log(\phi_{fi}) + \lambda_{fd} \sum_{i=1}^{|Q|} \sum_{\substack{j=1 \\ j \neq i}}^{|Q|} \log(\phi_{fd}) \tag{5.8}$$

As with the SD assumption, we set the values of the parameters λ_{fi} and λ_{fd} such that they sum to unity.

[4]The 3-node cliques used in FD are not to be confused with 3-node cliques in SD. For example, in Figure 5.1b, the nodes $\{f, q_1, q_3\}$ do NOT form a 3-node clique. However, in Figure 5.1b, the same three nodes do.

Weighted Sequential Dependence (MRF WSD)

In the Weighted Sequential Dependence (WSD) assumption proposed by Bendersky et al. [51], a weight is assigned to individual and pairs of query terms based on their importance to the retrieval process. This weight depends on several features that can be computed using internal and/or external data sources. Since the goal of Bendersky et al. [51] was retrieval from natural language corpus, the authors used Wikipedia and Google n-grams collection as external data sources. In our study, for the purpose of an unbiased comparative evaluation, we only consider internal data sources —the source code files— to compute feature values. Specifically, we find the collection and document frequency of each term and pair of terms which serves as their feature values. The set of six features used to compute the weight of each term and pair of terms are listed in Table 5.1.

The following scoring function is used to calculate the relevance of a bug report Q to source code file f:

$$P_{mrf,wsd}(Q,f) = \sum_{i=1}^{k_t} \lambda_{i,t} \sum_{j=1}^{|Q|} \beta_t(q_j, f, i) + \sum_{i=1}^{k_p} \lambda_{i,p} \sum_{j=1}^{|Q|-1} \beta_p(q_j q_{j+1}, f, i) \tag{5.9}$$

where k_t and k_p respectively are the numbers of single-term features and pair-of-terms features. Note that we have three single-term features and three pair-of-terms feature as listed in Table 5.1. Hence, $k_t = k_p = 3$. In Equation (5.9), $\lambda_{i,t}$ and $\lambda_{i,p}$ respectively are the weights assigned to ith single-term feature and ith pair-of-terms features. The functions $\beta(.)$ for single-term q_j and pair-of-terms $\rho = q_j q_{j+1}$ respectively are given below:

$$\beta_t(q_j, f, i) = \begin{cases} 0 & , g_i(q_j) = 0 \\ \log\left(g_i(q_j) \frac{tf(q_j,f) + \frac{\mu_{wsd} tf(q_j,C)}{|C|}}{(|f| + \mu_{wsd})} \right) & , \text{otherwise} \end{cases}$$

$$\beta_p(\rho, f, i) = \begin{cases} 0 & , h_i(\rho) = 0 \\ \log\left(h_i(\rho)\frac{tf_w(\rho,f)+\frac{\mu_{wsd}tf_w(\rho,C)}{|C|}}{(|f|+\mu_{wsd})}\right) & , \text{otherwise} \end{cases}$$

where $g_i(q_j)$ and $h_i(\rho)$ respectively are functions which return the value of ith feature for jth single term and ith feature for $\rho = q_j q_{j+1}$ pair of terms.

Note that WSD model reduces to SD model if we only consider the constant features (c_1 and c_2) mentioned in Table 5.1. Also note that if a feature value is zero for any term or pair of terms, the function $\beta(.)$ is set to zero since that term or pair of term has no significance in determining the relevance of file to the bug report.

5.1.2 Proximity based Divergence from Randomness

The basic idea of Divergence From Randomness (DFR) for retrieval, originally proposed by [52], has previously been incorporated in BoW based bug localization studies [11, 13]. Using this notion in term-term dependency modeling owes its origins to Peng et al. [21] who incorporated the sequential-dependency (SD) and the full-dependency (FD) assumptions as described in the previous section in a DFR based retrieval framework.

The main idea in DFR is that the importance of a term to a file should be measured by the extent to which the probability associated with the occurrence of the term in the file exceeds the probability associated with a pure chance based occurrence. Obviously, the more sparingly a term appears in a file, vis-a-vis its appearance in other files, should dictate the importance of that term for making discriminations between the files. Measuring this effect requires both a measure of the frequency with which the term occurs in the file and a measure of how unlikely it is to occur again in the same file, as we explain below.

DFR modeling says that the score of a file f with respect to either a single query term q_i, or a pair of terms q_i and q_j when we need to take term-term dependencies

into account, is a decreasing function of two probabilities, $Prob_1$ and $Prob_2$, in the form of:

$$P_{dfr}(\rho, f) = (-\log Prob_1)(1 - Prob_2) \tag{5.10}$$

where, for the case of a single term q_i, $\rho = q_i$, and, for the case of a pair of terms q_i and q_j, we set $\rho = q_i q_j$. $Prob_1$ is the probability that ρ will occur in file f purely by chance according to some chosen model of randomness. We must *discount* the importance lent by ρ to f by the complement of the probability that the same ρ will occur again in that file. As to why, the ability of a given ρ to discriminate between the files depends obviously on the extent to which ρ occurs sparingly in the files. The discounting provided by $Prob_2$ serves that purpose.

The Binomial model of randomness that we have used for our comparative study yields the following formula for calculating $Prob_1$:

$$Prob_1 = \binom{tf(\rho, C)}{tf(\rho, f)} p^{tf(\rho,f)} (1-p)^{tf(\rho,C)-tf(\rho,f)} \tag{5.11}$$

where, for the case of a single term q_i, $\rho = q_i$, and, for the case of a pair of terms when we want to incorporate term-term dependency, we set $\rho = q_i q_{i+1}$.

We can use the following Poisson approximation to the formula in Equation (5.11) for the case when the query terms are considered individually:

$$Prob_1 = \frac{e^{\xi} \xi^{tf(q_i,f)}}{tf(q_i, f)!} \tag{5.12}$$

where $\xi = tf(q_i, C)/n_f = p \cdot tf(q_i, C)$ is the expected frequency of term q_i in collection C.

As was mentioned previously, $Prob_1$ all by itself cannot be trusted to tell us how important a file f is to query term q_i (or to a pair of terms $q_i q_j$ taken together). What $Prob_1$ says about a file needs to be discounted by the complement of $Prob_2$, which gives us the probability that the same ρ will not occur again in the file. We compute $Prob_2$ as follows:

$$Prob_2 = \frac{tf(q_i, f)}{1 + tf(q_i, f)} \tag{5.13}$$

This formula can be extended to pair of terms ρ in which case q_i is replaced by ρ in the above equation. ρ can either be $q_i q_{i+1}$ or $q_i q_j$ depending on whether we are using the SD or the FD assumption.

When translating $Prob_1$ and $Prob_2$ into file scoring functions, it is necessary to factor in a normalization with respect to file sizes, as has been argued by Singhal et al. [56]. We refer to this normalization by *Normalization 2* in the context of DFR based retrieval. Under *Normalization 2*, the normalized term frequency is given by:

$$tf_n(q_i, f) = tf(q_i, f) \log(1 + \mu \frac{|f_{avg}|}{|f|}) \tag{5.14}$$

where μ is a tunable parameter, and $|f_{avg}|$ is the average length of a file in the collection. The normalization formula for the case of a pair of terms is obtained from Equation (5.14) by replacing q_i with $\rho = q_i q_{i+1}$ for the SD assumption and $\rho = q_i q_j$ for the FD assumption.

Based on the discussion so far, we have investigated two separate retrieval models under DFR, one for SD and the other for FD, with both using the Binomial model of randomness, denoted *BiL2*, for pairs of query terms and both using the Poisson approximation, denoted *PL2*, for single query terms. Here is the file scoring function for a single term q_i :

$$P_{PL2}(q_i, f) = \frac{1}{tf_n(q_i, f) + 1} \Big\{ tf_n(q_i, f) \log \frac{tf_n(q_i, f)}{\xi} + (\xi + \frac{1}{12 tf_n(q_i, f)} - tf_n(q_i, f))) \log e + 0.5 \log(2\pi tf_n(q_i, f) \Big\}$$

and here is the file scoring function for a single pair of terms ρ:

$$P_{BiL2}(\rho, f) = \frac{1}{tf_n(\rho, f) + 1}\{ - \log(|f| - 1)! + \log(tf_n(\rho, f)) + \log(|f| - 1 - tf_n(\rho, f))! - tf_n(\rho, f)\log(p_p) - (|f| - 1 - tf_n(\rho, f))\log(p'_p)\}$$

where $p_p = \frac{1}{|f|-1}$, $p'_p = 1 - p_p$.

Next we combine the single-term and pair-of-terms file scoring functions into functions that can be applied to the whole query. The final formula after combination for the case of the SD assumption is given by:

$$P_{dfr,sd}(Q, f) = \lambda_1 \sum_{i=1}^{|Q|} P_{PL2}(q_i, f) + \lambda_2 \sum_{i=1}^{|Q|} P_{BiL2}(\rho, f)$$

where $\rho = q_i q_{i+1}$, and the weights λ_1 and λ_2 are set such that they sum to unity. And, for the case of the FD assumption, the scoring function is given by:

$$P_{dfr,fd}(Q, f) = \lambda_1 \sum_{i=1}^{|Q|} P_{PL2}(q_i, f) + \lambda_2 \sum_{i=1}^{|Q|} \sum_{j=i+1}^{|Q|} P_{BiL2}(\rho, f)$$

where $\rho = q_i q_j$ and as for the case of SD. Again, the weights λ_1 and λ_2 are set such that $\lambda_1 + \lambda_2 = 1$.

5.1.3 Positional Language Model

We will now review the final term-term dependency model for our comparative study: the Positional Language Model (PLM) [22].

PLM modeling is based on the assumption that all query terms are dependent on one another in some probabilistic manner. One could argue that the MRF model with the FD assumption does the same thing. Note, however, that, in principle, the FD assumption implies an exponential number of dependency term groups. On the

Table 5.2.: Summary of all the term-term dependency models investigated along with the parameter settings that yielded the best results for each. For the sake of completeness, we have included the FI case (which is the same thing as BoW) in this table. Note that the subscript for the parameter μ is dropped in this table.

Method Name, Modeling scheme	Eclipse (title)	Eclipse (title+desc)	AspectJ (title)	AspectJ (title+desc)
MRF SD, MRF with Dirichlet smoothing	$w = 8$ $\lambda_{sd} = 0.2$ $\mu = 4000$	$w = 20$ $\lambda_{sd} = 0.2$ $\mu = 4000$	$w = 2$ $\lambda_{sd} = 0.2$ $\mu = 4000$	$w = 16$ $\lambda_{sd} = 0.2$ $\mu = 4000$
MRF FD, MRF with Dirichlet smoothing	$w = 8$ $\lambda_{fd} = 0.2$ $\mu = 4000$	$w = 8$ $\lambda_{fd} = 0.2$ $\mu = 4000$	$w = 8$ $\lambda_{fd} = 0.2$ $\mu = 4000$	$w = 16$ $\lambda_{fd} = 0.2$ $\mu = 4000$
MRF WSD, MRF with Dirichlet smoothing	$w = 2$ $\lambda_1 = 0.8$ $\lambda_{2,3} = 0.05$ $\lambda_4 = 0.2$ $\lambda_{5,6} = 0.05$ $\mu = 35000$	$w = 2$ $\lambda_1 = 0.95$ $\lambda_{2,3} = 0.03$ $\lambda_4 = 0.25$ $\lambda_{5,6} = 0.03$ $\mu = 500$	$w = 2$ $\lambda_1 = 0.8$ $\lambda_{2,3} = 0.1$ $\lambda_4 = 0.2$ $\lambda_{5,6} = 0.1$ $\mu = 4000$	$w = 2$ $\lambda_1 = 0.85$ $\lambda_{2,3} = 0.05$ $\lambda_4 = 0.4$ $\lambda_{5,6} = 0.05$ $\mu = 4000$
DFR SD, MRF with PL2&BiL2	$w = 2$ $\lambda_2 = 0.8$ $\mu = 10$	$w = 30$ $\lambda_2 = 0.8$ $\mu = 10$	$w = 8$ $\lambda_2 = 0.5$ $\mu = 150$	$w = 8$ $\lambda_2 = 0.8$ $\mu = 50$
DFR FD, MRF with PL2&BiL2	$w = 26$ $\lambda_2 = 0.6$ $\mu = 5$	$w = 16$ $\lambda_2 = 0.8$ $\mu = 5$	$w = 8$ $\lambda_2 = 0.5$ $\mu = 10$	$w = 2$ $\lambda_2 = 0.7$ $\mu = 20$
PLM BS, Propagation (Gaussian)	$\sigma = 65$ $\mu = 250$	$\sigma = 200$ $\mu = 2500$	$\sigma = 100$ $\mu = 500$	$\sigma = 90$ $\mu = 500$
PLM MS, Propagation (Gaussian)	$\sigma = 65$ $\mu = 250$ $\gamma = 0.5$	$\sigma = 250$ $\mu = 2500$ $\gamma = 0.7$	$\sigma = 100$ $\mu = 500$ $\gamma = 0.7$	$\sigma = 100$ $\mu = 500$ $\gamma = 0.6$
FI BoW, MRF with Dirichlet smoothing	$\mu = 4000$	$\mu = 4000$	$\mu = 4000$	$\mu = 4000$

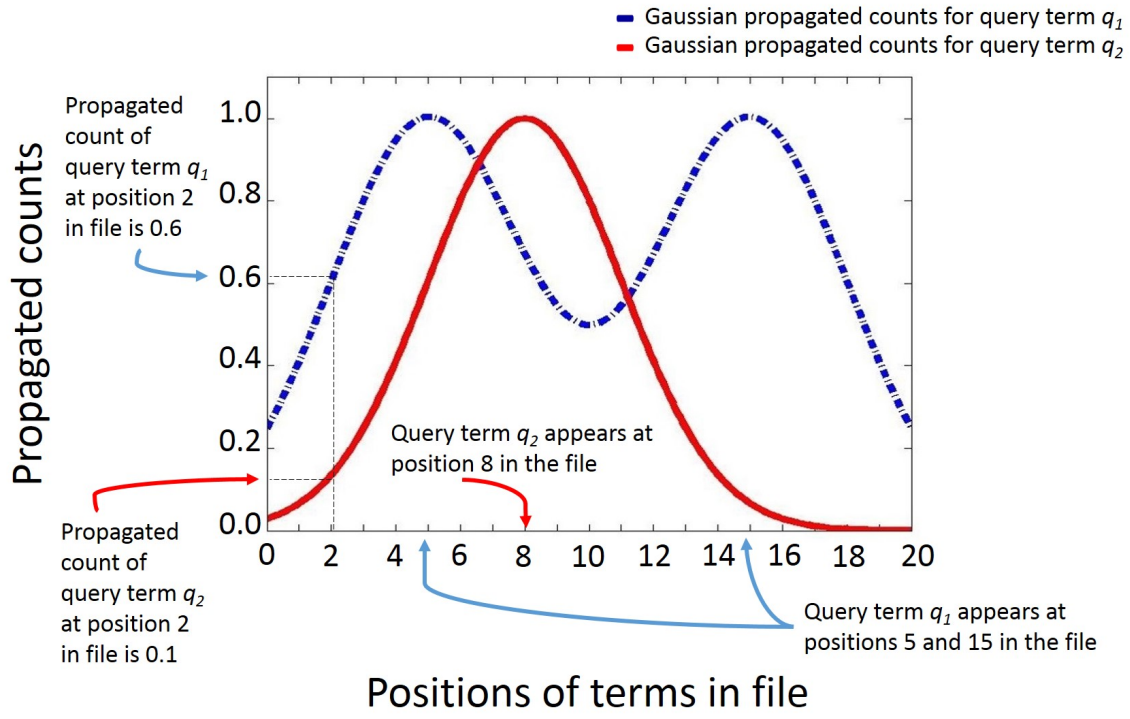


Fig. 5.2.: Illustration of term propagation using Gaussian density function. Notice that term q_1 appears at two positions 5 and 15, while term q_2 appears only at one position 8 in the file. The propagated count of q_1 at position * is approximately 0.6 while that of q_2 is approximately 0.1.

other hand, PLM creates just a single group of dependent terms — the group of all query terms.

In PLM, we represent each position i of a source code file f by a probabilistic distribution that, implicitly, depends on all other positions through a proximity-based density function as given in Figure 5.2. Consequently, we obtain what is referred to as *propagated counts* at position i from all other positions in the file. As to the choice of density function for this propagation effect, following [22], we have used the Gaussian density function in our bug localization experiments. After experimenting with several density functions for this purpose, we observed that Gaussian function worked the best in terms of retrieval accuracy. A smoothed version of this density function at each position i of source code file is given by:

$$P(w|f,i) = \frac{c^{'}(w,i) + \mu \frac{tf(w,C)}{|C|}}{(Z_i + \mu)\frac{tf(w,C)}{|C|}} \tag{5.15}$$

with the following notation:

- w is a term in source code file f
- $c^{'}(w,i)$ is the total propagated count of term w at position i from the occurrences of w in all the positions and is given by:

$$c^{'}(w,i) = \sum_{j=1}^{|f|} c(w,j)k(i,j) \tag{5.16}$$

  where $c(w,j)$ is 1 if w is at i and 0 otherwise, and $k(i,j)$ is the propagated count from a term at j to a term at i determined using Gaussian function $\exp(\frac{-(i-j)^2}{2\sigma^2})$. Here σ is a parameter.

- $tf(w,C)$ is the collection frequency of term w
- $Z_i = \sum_{w \in V} c^{'}(w,i)$ and is calculated using cumulative normal distribution [22].
- μ is a smoothing parameter.

As the reader can see, each file f is represented by a collection of the probability densities of the sort shown above. In the explanation that follows, we will refer to each density as a single PLM.

In order to determine the importance of a file f, as represented by the PLMs that apply to it, to a given query Q, we use KL-divergence as shown below to measure the weight that should be given to each of those PLMs:[5]

$$P_{KL}(Q, f, i) = -\sum_{w \in V} P(w|Q) \log \frac{P(w|Q)}{P(w|f, i)} \tag{5.17}$$

where $P(w|Q)$ is estimated using a maximum likelihood estimate. Note that the KL-divergence score involves summing over only those terms that both have a non-zero probability according to $P(w|Q)$ and occur in file f.

With PLMs we estimate a position specific score $P_{KL}(Q, f, i)$ at i using KL-divergence between a query, which is also represented by a collection of PLMs, and the PLMs for f. In what follows, we describe two variants of this model to estimate the overall score for the file f vis-a-vis a given query Q.

Best Position Strategy (PLM BS)

In this variant we simply score the file f based on the best matching position. The best matching position is the one where KL-divergence score will be maximum. Formally,

$$P_{plm,bs} = \max_{i \in [1,|f|]} \{P_{KL}(Q, f, i) \tag{5.18}$$

[5]Because calculating the PLM at each position i of file f is computationally expensive, we resort to computing the PLMs at only those positions of f at which the query terms appear. Intuitively, that makes sense because, as we discuss in Sections 5.1.3 and 5.1.3 we eventually require the score as computed from the best matching positions between f and Q. The matching values yielded by KL-divergence will be the highest at such positions. This approximation is in accordance with the implementation of PLM made available by its original authors. See http://sifaka.cs.uiuc.edu/ ylv2/pub/plm/plm.htm

Table 5.3.: Stats related to the two different software libraries used in our comparative study: Eclipse and AspectJ.

	Eclipse	AspectJ
Description	IDE	Java extension
Programming Language	Java	Java
number of bug reports	4035	291
average number of relevant files / report	2.76	3.09
number of bug reports with stack traces	519	89
number of bug reports with patches	8	4

Multi-σ Strategy (PLM MS)

We compute several best position scores for different σ values, and then combine these scores together to estimate the final relevance score for a file f with respect to query Q. As recommended by Lv and Zhai [22] we find scores using two values of sigma: ∞ and σ_0. Note that with $\sigma = \infty$, the PLM model degenerates into a traditional BoW model. The relevance score for multi-σ variant of PLM is given by:

$$P_{plm,ms} = \lambda_1 P_{\sigma_\infty}(Q, f) + \lambda_2 P_{\sigma_0}(Q, f) \tag{5.19}$$

where $P_{\sigma_\infty}(Q, f)$ can be computed using any of the BoW models, for example, we have used FI variant of MRF, and $P_{\sigma_0}(Q, f)$ can be calculated using BS variant of PLM described in the earlier section. λ_1 and λ_2 are the weights we assign to the FI and PLM models with the condition that they must sum to unity.

5.2 Query Conditioning

In the context of retrieval from software repositories, it is important to investigate the role of software-centric query conditioning (QC) procedures that give special

importance to what are usually referred to as "structured artifacts" — in particular, stack traces, patches, and the terms that stand for identifiers in code — that can be found in some bug reports [6, 9, 29]. The comparative results in this chapter are with and without query conditioning. These results help us understand the contribution that QC can make to each term-term dependency model.

When a stack trace or a patch is present in a bug report, it is first extracted from the bug report before further processing. For such bug reports, only the terms present in the structured artifacts are considered for further processing. Otherwise, the entire bug report is considered as it is. For structured-artifact-bearing bug reports, we next extract any camel-cased terms contained therein; these would normally be the identifiers used in the source code. The number of camel-cased terms must exceed a user-specified threshold n_{cc} in order to trigger the special treatment for such bug reports.

In the rest of this section, we briefly discuss the extraction of stack traces, patches, and the camel-cased terms when they are present in the bug reports.

For extracting a stack trace from a bug report, we take note of the fact that the most recent method call appears first in the call sequence of a stack trace[6]. Therefore, we extract only the topmost n_{st} methods since the methods which appear further down in the trace have a low probability of containing relevant terms, and they may introduce noise into the retrieval process. In our experiments, we have set n_{st} to 3 since it was found to provide the best accuracy. Note that only the methods belonging to the software library from which we want to retrieve files are extracted from a stack trace. In particular, the methods belonging to the Java platform itself are ignored while extracting stack traces.

Regarding source code patches, they are incorporated in bug reports using the Unified Format which calls for showing the differences between the original and the modified versions of a file. In this format, a patch contains the lines that need to

[6]For languages in which call sequence appears in the reverse order, obviously, the logic of identifying the methods needs to be reversed.

be removed or added apart from the contextual lines that are supposed to remain unchanged after the application of the patch. Obviously, the terms present in the lines that would be added after the application of the patch are not extracted since these lines are not yet present in the source code and, therefore, will not help in improving the retrieval process.

Regarding camel-cased terms, in addition to existing on their own in the bug reports, they are most likely to occur inside stack traces and code patches. When our regular expression based detectors identify a bug report as containing either a stack trace or a code patch or both, extraction of camel-cased terms is confined to the stack trace and/or patch.

5.3 Comparative Evaluation

We now present our comparative evaluation of the different term-term dependency models for automatic bug localization.

We report results using two different software libraries: Eclipse and AspectJ. The results for the Eclipse [7] and AspectJ [8] libraries are for two different types of retrievals: using just the titles of the bug reports as queries, and using the entire bug reports as queries.

The bug reports for Eclipse and AspectJ software libraries were obtained from the publicly available BUGLinks[9] and iBUGS[10] datasets, respectively. Table 5.3 presents the relevant stats related to the two software libraries. Notice that Eclipse is a Java based software with a large number of bug reports (4035) in the BUGLinks dataset. The iBUGS dataset, on the other hand, contains relatively smaller number of bug reports (291) for AspectJ repository [11].

[7] www.eclipse.org

[8] https://eclipse.org/aspectj

[9] https://engineering.purdue.edu/RVL/Database/BUGLinks/

[10] https://www.st.cs.uni-saarland.de/ibugs/

[11] Although BugLinks dataset contains 4650 bug reports for Eclipse and iBUGS dataset 350 bug reports for AspectJ software libraries, we have chosen 4035 and 291 bug reports from the dataset

In the rest of this section, we will first discuss the data flow in our bug localization framework. Then, we will describe the metrics we use in our study for comparative evaluation. That will be followed by the experimental results of the comparative study.

5.3.1 Overall Framework

Figure 7.2 illustrates the different steps in the bug localization framework used in this chapter.

The source code files in a library must go through stemming with the Porter stemming algorithm and stop word removal. Subsequently, each file is indexed, an inverted index constructed from the main index, and the two stored in hash tables. Also, note that while indexing the source code files we extract the positions of each term in the file and store them as part of the index. These positions are then loaded from the index and used in the retrieval process. Specifically, frequencies of occurrence of a pair of terms $q_i q_j$ (within a window of size w) in a file f can be computed using the positions of the query terms in file.

On the other hand, the bug reports are first subject to regular-expression based testing for the detection of stack traces or code patches. If these are absent, further regular-expression based testing is carried out for the detection of camel-cased source code identifiers. In the absence of such identifiers, the bug reports are subject to the same preprocessing steps as the source code files.

5.3.2 Evaluation Metrics

The retrieval accuracies of different term-term dependency models are measured using precision based metrics [57]. Specifically, precision at rank r ($P@r$), and mean average precision (MAP) metrics are used to evaluate the retrievals for different term-

for our analysis , respectively. We ignored those bug reports for which we could not find any of the accompanied source code file(s) in the software library.

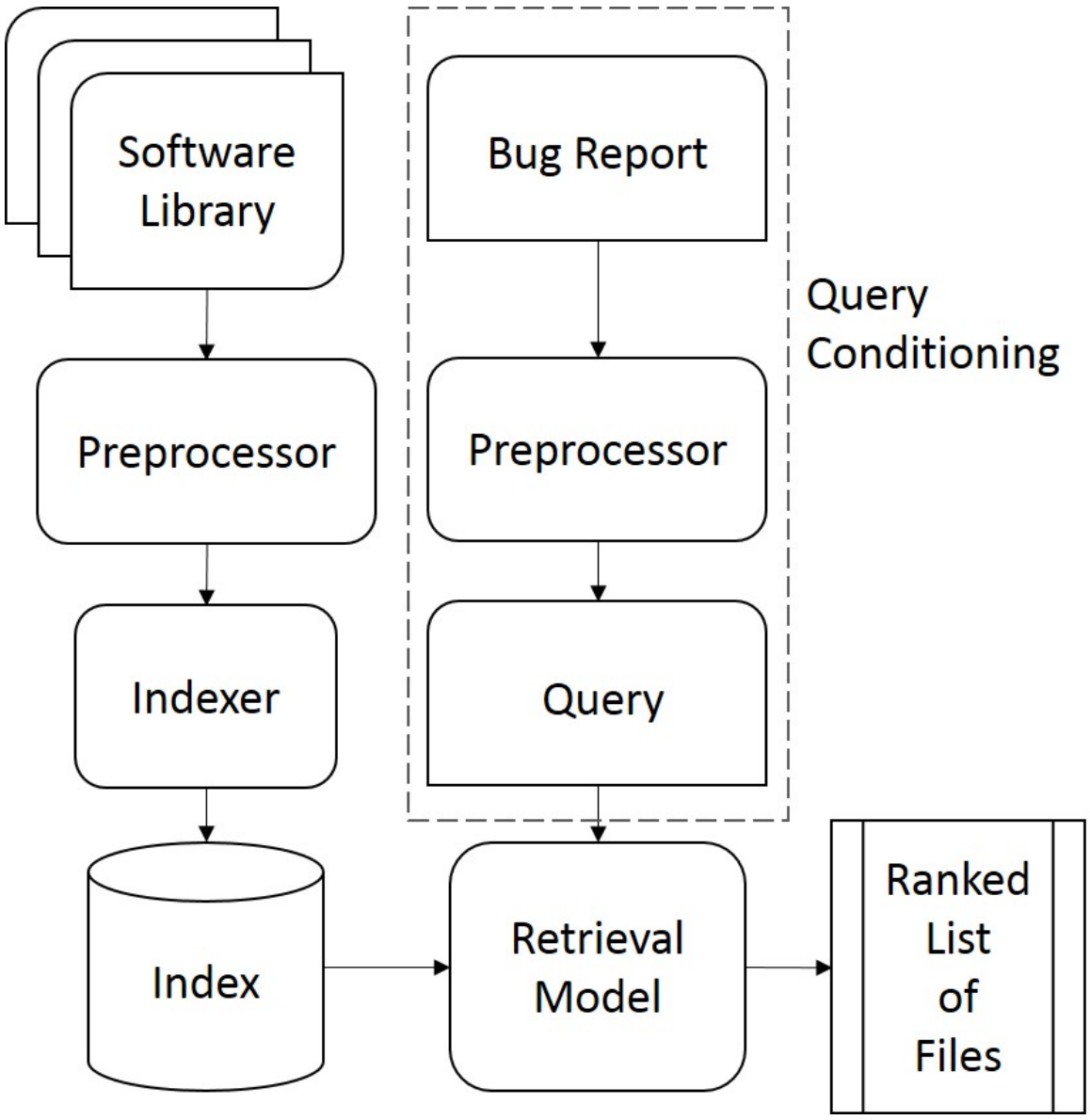


Fig. 5.3.: Block diagram for the retrieval framework.

term dependency models. $P@r$ measures the accuracy with which retrieval is performed upto rank r. The average precision (AP) for a query Q is given by:

$$AP(Q) = \frac{\sum_{r=1}^{RF} P@r \cdot I(r)}{rel_Q}$$

where rel_Q is the total number of relevant files for Q, RF the number of top-ranked files which are analyzed for the calculation of AP, and $I(r)$ the indicator function which returns 1 when the file at rank r is a relevant file, and 0 otherwise. MAP and $P@1$ are the most important metrics which determine the power of retrieval engine.

Significance testing based on student's t-test is performed to determine whether improvements observed in the retrieval results obtained using term-term dependency models vis-à-vis the BoW model are significantly different. However, since significance testing can only tell whether the results are significantly different or not without providing any measure of the amount of difference, we also compute the *effect size* to determine that measure. In this chapter, we report one such popular effect size metric called Cohen's d which is calculated as the standardized difference between two sample means.

5.3.3 Bug Localization Experiments

As summarized in Table 5.2, we compared the following eight retrieval models: (1) MRF FD, (2) MRF SD, (3) MRF WSD, (4) DFR SD, (5) DFR FD, (6) PLM BS, (7) PLM MS, and (8) FI BoW. That table also displays the tunable-parameter values used for each case. These parameters were set to yield the best performance from each model. For obvious reasons, the larger the number of parameters in a model, the greater the effort required for tuning them. The number of parameters ranges from eight for WSD to just one for FI.

Table 5.4.: Retrieval accuracy for the "title-only" queries.

Method	Eclipse			
	MAP (%) (d)	P@1	P@5	R@10
MRF FD	0.2527 (23.5%) (0.150)	0.2169	0.1094	0.4007
MRF SD	0.2466 (20.5%) (0.135)	0.2134	0.1055	0.3895
MRF WSD	**0.2585 (26.4%) (0.167)**	**0.2226**	**0.1114**	**0.4123**
DFR SD	0.2496 (22.0%) (0.141)	0.2102	0.1087	0.4014
DFR FD	0.2551 (24.7%) (0.158)	0.2208	0.1107	0.4066
PLM BS	0.2173 (6.20%) (0.041)	0.1772	0.0957	0.3651
PLM MS	0.2332 (14.0%) (0.091)	0.1963	0.0998	0.3841
FI BoW	0.2045	0.1695	0.0883	0.3400
	AspectJ			
MRF FD	0.1462 (16.5%) (0.091)	0.1478	0.0845	0.2483
MRF SD	0.1411 (12.5%) (0.068)	0.1615	0.0742	0.2312
MRF WSD	**0.1506 (20.0%) (0.110)**	**0.1615**	**0.0811**	0.2566
DFR SD	0.1502 (19.7%) (0.109)	0.1478	0.0825	**0.2601**
DFR FD	0.1495 (19.2%) (0.105)	0.1409	0.0832	0.2600
PLM BS	0.1336 (6.50%) (0.037)	0.1340	0.0784	0.2362
PLM MS	0.1373 (9.40%) (0.053)	0.1478	0.0784	0.2370
FI BoW	0.1254	0.1306	0.0708	0.2128

Results for title-only queries

The experimental results for the case of title-only queries are shown in Table 5.4. From the results it can be seen that WSD variant of MRF outperforms all the other algorithms in terms of MAP values for Eclipse as well as for AspectJ software libraries. MRF WSD is an extension of SD variant of MRF in which instead of considering all terms and pairs of terms being equal, weights are assigned to them based on their importance.

It is also worth noting that every variant of every term-term dependency model is more powerful than the FI BoW model. The values in brackets in the second column refer to the percent improvement over BoW model. All the dependency-model based improvements reported for Eclipse in Table 5.4 are significant at $\alpha = 0.05$ level when evaluated using student's t-test. Also the effect sizes d for each dependency model when compared against the BoW model are noted in the second column. It is interesting to note that the top 3 methods that perform the best in terms of MAP values for Eclipse are: (1) MRF WSD, (2) DFR FD, and (3) MRF FD. While, for AspectJ the top 3 models are: (1) MRF WSD, (2) DFR SD, and (4) DFR FD.

All MRF based term-term dependency models perform better than PLM. The improvements observed over the best of PLM variant (i.e. PLM MS) using the best of MRF variant (i.e. MRF WSD) and the best of PDFR variant (i.e. DFR FD) on Eclipse title-only queries are 10.85% and 9.39%, respectively. Notice that on AspectJ title-only queries, the best performing PDFR variant is DFR SD. And the improvements over PLM MS using MRF WSD and DFR SD are 9.6% and 9.3%, respectively. All the dependency models except PLM MS and PLM BS are statistically significant at $\alpha = 0.05$ level when evaluated using student's t-test.

Results for title+desc queries with QC

In this section, we discuss the results for the second experiment which uses title+desc queries with QC. Shown in Table 5.5 are the results for this experiment.

Table 5.5.: Retrieval accuracy for the "title+desc" queries.

Method	Eclipse			
	MAP (%) (d)	P@1	P@5	R@10
MRF FD	0.3066 (15.6%) (0.113)	0.2756	0.1292	0.4640
MRF SD	0.3022 (13.9%) (0.105)	0.2701	0.1281	0.4553
MRF WSD	0.3089 (16.5%) (0.123)	0.2793	0.1290	0.4602
DFR SD	0.2976 (12.2%) (0.092)	0.2691	0.1261	0.4481
DFR FD	**0.3098 (16.8%) (0.126)**	**0.2796**	**0.1303**	0.4687
PLM BS	0.2697 (1.74%) (0.013)	0.2305	0.1174	0.4298
PLM MS	0.2791 (5.28%) (0.040)	0.2401	0.1190	0.4358
FI BoW	0.2651	0.2283	0.1144	0..4114
	0AspectJ			
MRF FD	0.2371 (10.1%) (0.073)	0.2818	0.1175	0.3468
MRF SD	0.2311 (7.30%) (0.053)	0.2680	0.1148	0.3530
MRF WSD	**0.2458 (14.1%) (0.102)**	0.2852	**0.1265**	**0.3768**
DFR SD	0.2276 (5.71%) (0.041)	0.2474	0.1162	0.3534
DFR FD	0.2343 (8.80%) (0.063)	0.2543	0.1155	0.3582
PLM BS	0.2340 (8.60%) (0.063)	0.2749	0.1203	0.3533
PLM MS	0.2456 (14.0%) (0.100)	**0.2887**	0.1203	0.3599
FI BoW	0.2153	0.2440	0.1072	0.3179

As can be seen in Table 5.5, DFR FD beats all the other term-term dependency models in terms of MAP values for Eclipse dataset. The top three models that perform the best in terms of MAP values are: (1) DFR FD, (2) MRF WSD, and (3) MRF FD. While for AspectJ dataset, MRF WSD outperforms all other models. And the top three models for AspectJ are (1) MRF WSD (2) PLM MS, and (3) MRF FD.

As with the results for the title-only queries, note that all term-term dependency models significantly improve the retrieval accuracy when compared with the BoW model. Again, we perform significance testing to evaluate whether the improvements over BoW model are significant or not. We note that all the results except for PLM BS are statistically significant at $\alpha = 0.05$ level for Eclipse dataset. While for AspectJ dataset, only the top three models — MRF WSD, PLM MS, and MRF FD — are statistically significant. The effect sizes d are noted in the second column of the table.

Additionally, PLM based term-term dependency models do not perform as well as the MRF based term-term dependency models for Eclipse title+desc queries. But the results of PLM on AspectJ dataset are surprisingly good.

The improvements observed over the best of PLM variant (i.e. PLM MS) using the best of MRF variant (i.e. MRF WSD) and the best of PDFR variant (i.e. DFR FD) for Eclipse title+desc with QC experiment are 10.68% and 11.00%, respectively. It is interesting to note that PDFR which performed better than PLM in all previous experiments — Eclipse title-only, AspectJ title-only, and Eclipse title+desc — performs worse than PLM in AspectJ title+desc experiment.

Effect of using QC

When structured components like stack traces and patches are present in a bug report we found that the retrieval precision greatly improves if they are utilized appropriately. Figure 5.4 shows the MAP values for different term-term dependency models with and without using QC. It can be noted that MAP values are significantly

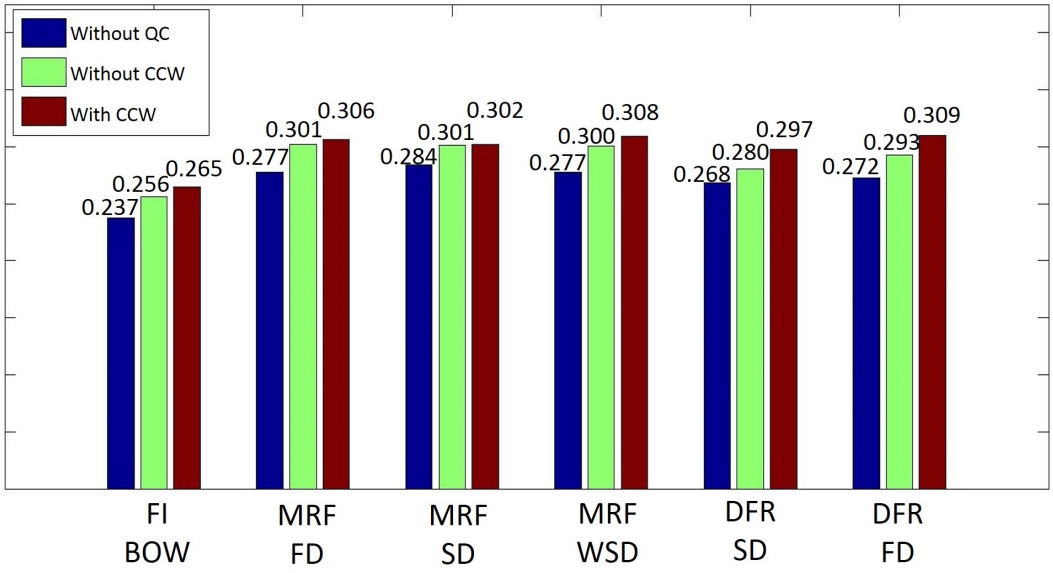


Fig. 5.4.: Effect of using QC for bug localization.

higher when QC is used than when it is not used for all the term-term dependency models.

In addition to these structured components, when the bug report narrative contains camel-cased terms, it implies that the individual who has filed the bug report is familiar with the software library. These camel-cased terms, when present in abundance in the report, are very significant because they usually refer to the source code identifiers used in the software library. Therefore they can directly point to the source code files which contain the bug. Exploiting this important information as described in Section 5.2, we perform experiments using title+desc queries. The results with and without using these camel-cased words (CCW) for different models on Eclipse dataset are shown in Figure 5.4. It can be seen from the results that the camel-cased words are very informative and their presence must be exploited to fully utilize the power of term-term dependency models.

Comparison with state-of-the-art BoW models

In order to evaluate the power of term-term dependency models vis-à-vis the state-of-the-art techniques in bug localization, we perform a comparative study between the term-term dependency models and three popular bug localization techniques using Eclipse dataset. These three techniques are: BugLocator [12] [8], BLUiR [13] [12], and SCP-QR [14] [7]. It is important to note that at the heart of these three techniques are three separate ideas; BugLocator exploits bug history, BLUiR uses the notion of field-based retrieval, and SCP-QR uses query reformulation.

In BugLocator when a new bug report is received it is treated as an initial query and revised Vector Space Model (rVSM) is used to retrieve files from the repository. For this new bug report, previously resolved similar bugs are also collected. The final ranks of files are determined by combining the ranks obtained from the initial query on the source code files as well as from the analysis of past similar bugs.

BLUiR splits the bug report and source code files into separate fields. Specifically, a bug report is broken into summary and description. While each source code file is split into four components: class, method, variable, comments. Afterwards, a separate search is performed for each of the eight (bug report field, file field) combinations using Okapi BM25 retrieval model. The final score for a certain file is obtained by summing the scores across all eight searches.

The comparative results of term-term dependency models vis-á-vis BugLocator and BLUiR for Eclipse title+desc queries are shown in Figure 5.5. Notice that DFR FD and MRF WSD beat all state-of-the-art bug localization methods except BLUiR. Also, BugLocator comes very close to these dependency models. However, it is worth noting that term-term dependency models outperform BugLocator without having to keep track of the history of bug reports.

[12]https://code.google.com/archive/p/bugcenter/wikis/BugLocator.wiki
[13]http://www.riponsaha.com/BLUiR.html
[14]The software for SCP-QR is obtained by contacting the authors.

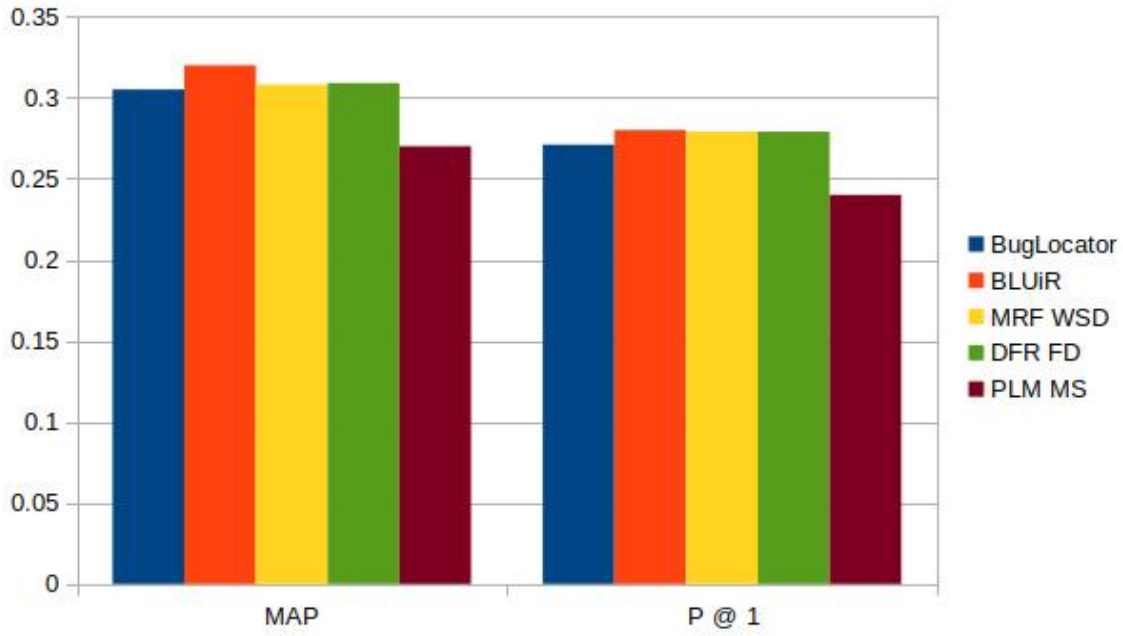


Fig. 5.5.: Comparison with state-of-the-art bug localization.

Using SCP-QR, the MAP value obtained for Eclipse title-only queries is 0.2296. In comparison, MRF WSD, DFR FD, and PLM MS outperform SCP-QR with MAP values 0.2585, 0.2551, and 0.2332, respectively. Notice that the results are compared using title-only queries because, as argued in [7], the query reformulation method is only effective for title-only queries.

Even though term-term dependency models greatly improve retrieval performance when compared against simple BoW models, their performance against advanced BoW models —BugLocator and BLUiR— is not very impressive. Therefore, in the subsequent chapters of this dissertation, we will explore additional ways to improve the performance of bug localization systems.

6. CONSTRUCTING WORD EMBEDDINGS FOR SOFTWARE-CENTRIC TERMS

In the earlier chapters, we have discussed retrieval models that compute the relevance score for a file given a query based on the frequencies of individual query terms appearing in the code files. These models are called BoW models and are considered the baseline models for evaluating the performances of more advanced methods. In particular, we have discussed the DLM and TFIDF BoW retrieval models in Chapter 4.

We have also discussed advancements in retrieval frameworks that extend simple BoW models by incorporating term-term ordering dependencies. For example, in the MRF based retrieval model presented in Chapter 5, the frequencies of pairs of query terms are measured in code files to enhance retrieval precision.

These advances in the development of software search tools can now be further augmented with semantics by drawing on the work that shows how the words in a corpus can be represented by numeric vectors — referred to as their *semantic word embeddings* — such that the cosine distance between the vectors for two different words is small when the words are semantically related. This entirely new way of looking at textual data was popularized by the word2vec contribution by Mikolov et al. [17] and remains highly popular for incorporating semantics in text based processing of information.

The semantic word embedding algorithms are trained on a large corpus of text documents in a self-supervised fashion. In this chapter, we study various word embedding algorithms for modeling software repositories along with their relative performances on the identification of semantically similar software-centric words.

The most popular algorithm for constructing word embeddings — word2vec — is based on a shallow single-layer neural network and works by scanning the corpus

with a window of pre-defined size while predicting the context words appearing in a window around target words. One important limiting factor of word2vec is that it cannot generate vectors for what are called out-of-vocabulary (OOV) words, which are words that were not seen by the word2vec neural network during the time of training. However, if word2vec is trained to understand the morphological structure with which words are formed in a given language, then the vector for an OOV word can be constructed at the testing time using the vector representation of morphologies of words that word2vec had already seen in the training dataset.

This is exactly what FastText proposes [18]. It extends word2vec by enriching the word vectors with subword level information. More specifically, FastText proposes to learn representations for character n-grams, as opposed to words, and to represent words using the summation of the vectors of their respective component n-grams.

Both word2vec and FastText are called prediction-based methods because they both work by training a neural network to discriminate words that appear in the context of target words from all the other words that are present in the vocabulary. Both word2vec and FastText, therefore, ignore the statistics of pairs of terms occurring together in the corpus. GloVe [19], which is another semantic word embedding algorithm, is considered a count-based method because it considers the statistics of pairs of terms occurring together, and works by factorizing the term-term co-occurrence count matrix to construct word embeddings[1].

Before we start discussing the three semantic word embeddings algorithms mentioned above in greater detail, it is important to note that the word2vec algorithm builds on top of the works proposed in [58–60]. Bengio et al. [58] introduced for the first time a neural network model that could be used for generating semantic embeddings. Subsequently, refinements were made to their algorithm in order to make it more efficient by several researchers [59, 60].

[1]While word2vec embeddings have previously been utilized for building a source code search engine in the SCOR [16] retrieval framework, the other techniques — FastText and GloVe — are used for the first time in this study.

Table 6.1.: Stats related to the large Java corpus that we used to learn the semantic word vectors from the word2vec model.

	Statistics
Number of repositories	34264
Programming Language	Java
Size of raw dataset	368 GB
Number of source code files	3444730
Number of word tokens	940053404
Number of words in vocabulary	415554

In the rest of this chapter, we first discuss the dataset that we have used to train the word embedding algorithms. We will then review the three word embedding models — word2vec, FastText, and GloVe — to construct semantic word vectors for software-centric terms. Finally, we will analyse the quality of semantic word vectors produced by these word embedding models.

6.1 Dataset for Training Word Embedding Models

The word2vec, FastText, and GloVe word embedding models were originally developed to learn semantic word vectors for regular English language words. Therefore, the authors of these models used thousands of Wikipedia and news articles for the purpose of training their models.

Our task, however, involves searching in software codebases which contain terms that are quite distinct from regular English words. Additionally, when writing code for a software, the developers invent their own abbreviations for commonly occurring software terms. For example, the term ‘‘`model`’’ is often abbreviated as ‘‘`mod`’’, ‘‘`delete`’’ as ‘‘`del`’’, ‘‘`socket`’’ as ‘‘`skt`’’, and ‘‘`iteration`’’ as ‘‘`iter`’’.

Therefore, in order to train word embedding models to generate vectors for software-centric terms, we developed our own dataset of software repositories called the SCOR[2] word embeddings dataset[3] [16]. This dataset contains 1 billion software term tokens present in 35 million Java source code files belonging to 35000 repositories. The resulting semantic word vector space contains vectors for half a million unique software terms.

Table 6.1 shows the stats related to the Java source code dataset that we have used for training the word embedding algorithms. In light of the diversity of the software projects used for generating the embeddings, it is safe to assume that these word vectors are sufficiently generic and should be useful for software search in the wild even when the target library was not present in the training dataset of the word embedding model.

6.2 Word Embedding Models

In this section, we discuss the word embedding models that we have used for incorporating term-term semantic relationships in the retrieval framework. These models include: word2vec Skipgram, word2vec CBoW, FastText, and GloVe.

6.2.1 W2V: Word2vec Model

The word2vec model [17] is based on a shallow single-layer neural network which gives us a way to construct a vector space that holds contextually semantic relationships between the words in a vocabulary. Contextual semantics means that we consider two terms similar if the words appearing in their contextual neighborhoods are similar. For example, for software-centric word embeddings, one would expect to

[2]SCOR, which stands for source code retrieval with semantics and order, is a retrieval framework that combines MRF [6] with word2vec based word embeddings to model both order and semantics into a single retrieval framework. The word embeddings obtained from FastText and GloVe can directly be plugged into the SCOR retrieval framework to replace word2vec. In Chapter 7, we provide a detailed treatment of the SCOR retrieval framework.

[3]https://engineering.purdue.edu/RVL/SCOR_WordEmbeddings

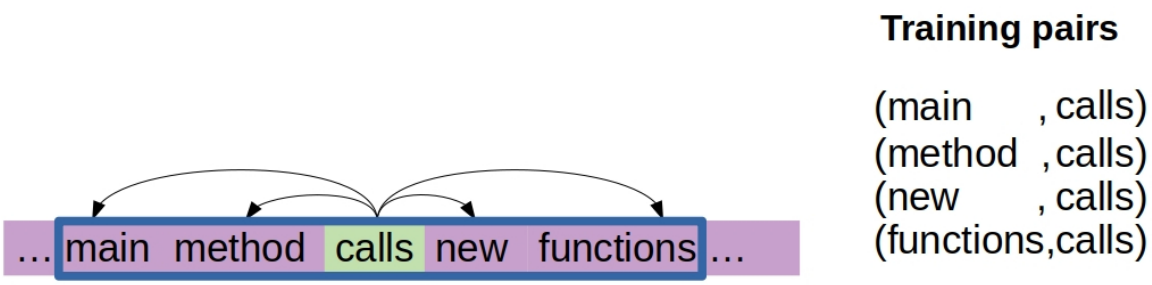


Fig. 6.1.: An illustration of a context window of size 5 around the target term "calls". Notice that the four terms "main", "method", "new", and "functions" are the context terms of the target term "calls" and will be used to train the model.

see vectors for terms like "delete", "del", "remove", "cancel", and "update" similar from the standpoint of the cosine distance measure.

For generating these word embeddings, the training samples are automatically generated by scanning the source code files using a window of a pre-specified size to define the contextual terms in the vicinity of each target term. That is, a training sample consists of a term in a repository along with a list of terms that appear inside the window. The width of the window is a user defined parameter. The term around which a window is placed is called the "target" term, while the terms appearing inside the window other than the target term are called the "context" terms with respect to that target term. Figure 6.1 illustrates the concepts of target and context terms using an example. Notice that there are four training pair samples that can be extracted from the window shown in the figure. These training pairs are ("main", "calls"), ("method", "calls"), ("new", "calls"), and ("functions", "calls").

The word2vec neural network is trained using the training pairs consisting of target and context terms. The prediction task can either be formulated as being from target terms to context terms as carried out in the word2vec's Skipgram model, or from context terms to target terms as in word2vec's Continuous Bag of Words (CBoW) model. The important thing to note is that after word2vec has scanned through all

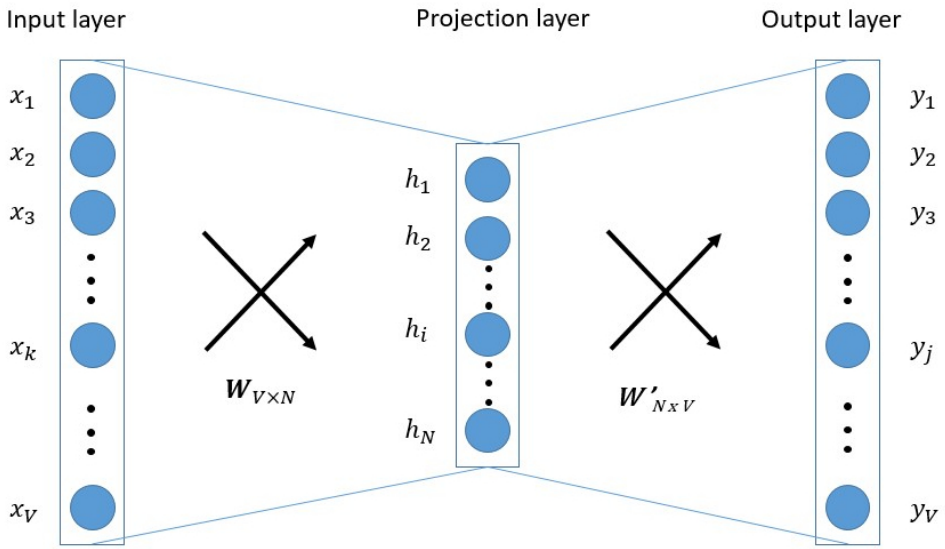


Fig. 6.2.: The Skip-gram neural network predicts the softmax probabilities of context terms from one-hot encoding of given target terms.

the source code files and has finished training, the weights of the neural network can be used to construct meaningful semantic word vector representations for software terms in the vocabulary set.

In our study we use Skipgram as well as CBoW word embeddings for source code retrieval. The Skipgram neural network shown in Figure 6.2 is designed to predict target terms from the context terms. Throughout the corpus, the context terms are believed to appear with a high likelihood in the neighborhood of the target term. The neural network consists of three layers: input, projection, and output. Notice that the nodes in the adjacent layers are fully connected.

The sizes of input and output layers in the network shown in the figure are equal to the number of words V in the vocabulary set, while the size of the projection layer present in between input and output layers is N — a tunable parameter. The size of the projection layer N is set to be always less than V, and is used to control the dimensionality of the vectors produced by the neural network.

The vocabulary set is sorted in alphabetical order, and each term in the vocabulary is assigned an index corresponding to its position in the vocabulary. The V dimensional one-hot encoding vector of the target term, which is a vector of all zeros except for a 1 at the target term index, is provided as input to the neural network. The network is trained to predict V dimensional output in which each node represents the softmax probability of prediction for each term in the vocabulary set.

The input, projection, and output layers are represented by the vectors $\mathbf{x}$, $\mathbf{h}$ and $\mathbf{y}$, while the weights between the input and projection layers are denoted by a $V \times N$ matrix $\mathbf{W}$, and between projection and output layers by another $N \times V$ matrix $\mathbf{W}'$. The rows of $\mathbf{W}$ are referred to as the N-dimensional *input vectors* $\mathbf{v}_w$ for the terms w in the vocabulary, while the columns of $\mathbf{W}'$ are referred to as the *output vectors* $\mathbf{v}'_w$ for the terms w in the vocabulary.

If $\mathbf{v}_{w_I}$ is the input vector for a target term, the projection layer output would be given by $\mathbf{h} = \mathbf{W}^T\mathbf{x} = \mathbf{v}_{w_I}$, which is the I^{th} row of the matrix W.

Applying the weights in the matrix $\mathbf{W}'$ to the outputs of the projection layer we get the score $\mathbf{v}'^T_{w_j}\mathbf{h} = \mathbf{v}'^T_{w_j}\mathbf{v}_{w_I}$, where $\mathbf{v}'^T_{w_j}$ denotes the j-th column of $\mathbf{W}'$. To convert the results into probability estimates $p(w_j|w_I)$ we apply the softmax nonlinearity function:

$$p(w_j|w_I) = y_j = \frac{\exp(\mathbf{v}'^T_{w_j}\mathbf{h})}{\sum_{j'=1}^{V}\exp(\mathbf{v}'^T_{w_{j'}}\mathbf{h})} \tag{6.1}$$

The goal of this neural network is to maximize the above conditional probability that the term w_j is the context term given the target term w_I at input. Maximizing this probability amounts to minimizing the loss function:

$$E = -\log p(w_j|w_I) = -\log\frac{\exp(\mathbf{v}'^T_{w_{j*}}\mathbf{h})}{\sum_{j'=1}^{V}\exp(\mathbf{v}'^T_{w_{j'}}\mathbf{h})} \tag{6.2}$$

$$= -\mathbf{v}'^T_{w_{j*}}\mathbf{h} + \log\sum_{j'=1}^{V}\exp(\mathbf{v}'^T_{w_{j'}}\mathbf{h}) \tag{6.3}$$

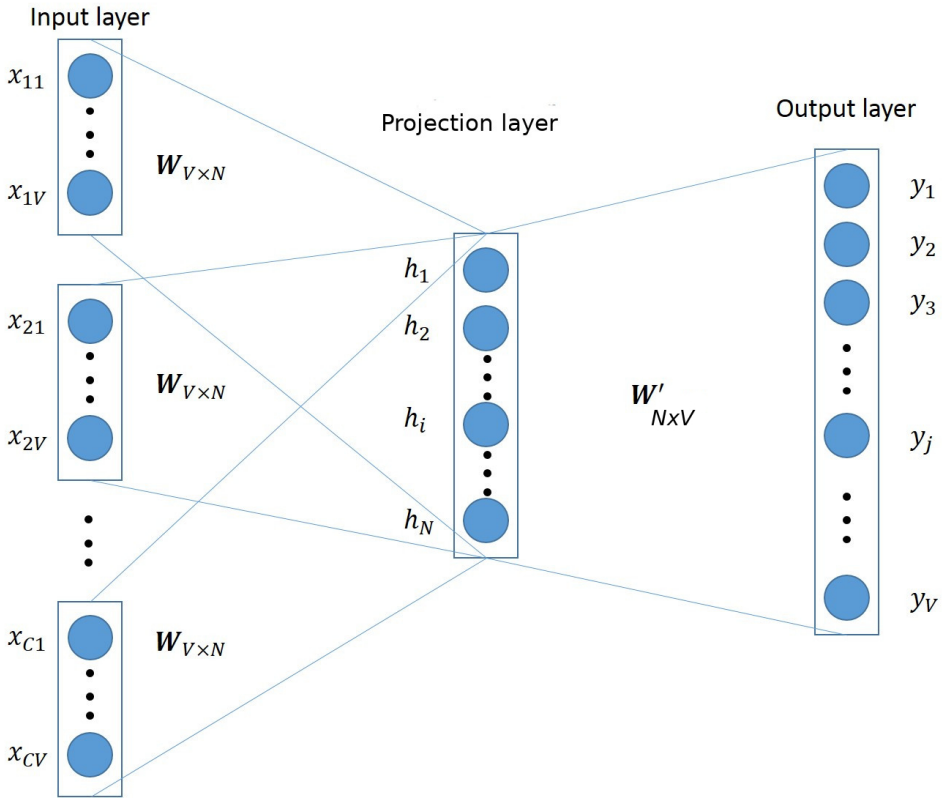


Fig. 6.3.: The CBoW neural network predicts the softmax probabilities of target terms from one-hot encoding of given context terms.

In the above equation, $j*$ corresponds to the index of the actual context term observed in the context window of target term w_I.

The training of Continuous Bag-of-Words (CBoW) model is very similar to the Skipgram model as shown in Figure 6.3. Notice that in Skipgram we predict context terms from target terms, while in CBoW we predict target terms from context terms.

The difference between CBoW and Skipgram is in the computation of input-to-projection layer. The input layer in CBoW has a size of C times V, where C is the number of context terms appearing inside the window of the target term.

While in Skipgram $\mathbf{h}$ is simply the input vector of the target term $\mathbf{v}_{w_I}$, in CBoW $\mathbf{h}$ is computed by averaging the input vectors $\mathbf{v}_{w_1}, \mathbf{v}_{w_2}, ..., \mathbf{v}_{w_C}$ of all the context terms of the target term:

$$\mathbf{h} = \frac{1}{C}(\mathbf{v}_{w_1} + \mathbf{v}_{w_2} + ... + \mathbf{v}_{w_C}) \tag{6.4}$$

The formulas for conditional probability and loss function for CBoW take the same forms as they do for the Skipgram model, with the **h** vector being computed as presented in the equation above and w_j representing the predicted target term instead of predicted context term at the output layer.

A detailed treatment of the training procedure that can learn the weights of the Skipgram and CBoW neural networks is provided in Appendix A and in [61]. After the learning process converges, the resulting vectors $\mathbf{v}_w$ and $\mathbf{v'}_w$ correspond to the terms w in the vocabulary. In our retrieval experiments, we only use the input vectors $\mathbf{v}_w$ to represent terms in the vocabulary. Two terms can be compared on the basis of their vector representations using an appropriate similarity metric to determine the semantic relatedness of one vis-a-vis the other.

6.2.2 FastText: Enriching Vectors With Subword Information

FastText extends word2vec by introducing subword level information into the vector representation of words. The main motivation behind FastText is to solve the problem of out-of-vocabulary (OOV) words appearing at the testing stage. The OOV words are the words that did not appear in the training corpus.

FastText solves this problem by learning vector representations of character n-grams, instead of words themselves, and by representing each word as the sum of the n-gram vectors.

Before constructing character n-grams from a word, special boundary symbols < and > are added at the start and end of each word, respectively. For example, the word "method" becomes "<method>". The role of these special symbols is to distinguish prefixes and suffixes of a word from the other character sequences. Considering n=3, the word "<method>" becomes a bag of following character n-grams:

<me, met, eth, tho, hod, od>

In addition to the above character n-grams a special sequence "<method>", i.e. the word itself along with the boundary symbols, is also included in the bag of n-grams.

To train the FastText model, the first step is to construct a dictionary G of all the n-grams that can be formed from the words present in the input corpus. The set of n-grams for a specific word w is denoted by G_w which would be a very small subset of G.

Using $\mathbf{z}_g$ to represent the vector representation of n-gram g, the vector representation of a word w can be computed by summing the vectors $\mathbf{z}_g$ of all the n-grams g in G_w:

$$\mathbf{v}_w = \sum_{g \in G_w} \mathbf{z}_g \tag{6.5}$$

FastText uses the same neural network as the word2vec model. The only difference is that the words are represented by the summation of the vectors belonging to their character n-grams in FastText. In doing so, when a new OOV word appears in the testing stage, as long as the character n-grams of the word are present in the dictionary G of the FastText model, a vector representation of the OOV word can be constructed by summing the vectors of its constituent n-grams[4].

The loss function for FastText is the same as the one given in Equation (A.4), except that the vectors $\mathbf{v}_w$ for words w are represented as summation of the character n-grams formed by w. The loss function is given by:

$$E = -\sum_{g \in G_w} \mathbf{z}_g^T \mathbf{v}_{w_j} + \log \sum_{j'=1}^{V} \exp(\sum_{g \in G_w} \mathbf{z}_g^T \mathbf{v}_{w'_j}) \tag{6.6}$$

[4]Obviously, the vector representation for the special sequence, i.e. OOV word itself, would not be present in the model, and will therefore be ignored in the summation. FastText paper shows that ignoring it does not affect the vector representation of the word.

6.2.3 GloVe: Global Vectors for Word Representation

Unlike word2vec model which works by predicting context terms from target terms using a neural network, GloVe [19] uses frequencies of pairs of context and target terms occurring together in the corpus to construct vector representations for terms. GloVe has been shown to improve results over word2vec for certain NLP tasks.

GloVe works in the following manner. The term-term co-occurrence statistics $X_{w_i w_j}$ is gathered by scanning the corpus with a small window of predefined size, and counting the number of times the terms w_i and w_j appear together in the corpus.

To train GloVe, two randomly initialized vectors $\mathbf{v}_{w_i}$ and $\mathbf{v}'_{w_i}$ are associated with each word i in the vocabulary and the following loss function is minimized:

$$E = \sum_{i,j} f(X_{w_i w_j})(\mathbf{v}_{w_i}^T \mathbf{v}'_{w_j} - \log(X_{w_i w_j}))^2 \tag{6.7}$$

In the above equation equation $f(X)$ is a weighting function whose main purpose is to downweight the rare term-term cooccurrences using the following heuristic:

$$f(X_{w_i w_j}) = \begin{cases} \left(\frac{X_{w_i w_j}}{X_{max}}\right)^{\alpha} & \text{if } X_{w_i w_j} \leq X_{max} \\ 1 & \text{otherwise} \end{cases} \tag{6.8}$$

where X_{max} and α are parameters that are chosen to be 100, and 0.75, respectively.

6.3 Implementation Details

The source code files present in the 35000 Java software repositories that are used to construct word embeddings are first preprocessed using tokenization and stop-word removal. The software terms that survive the preprocessing step are stemmed to their roots using the Porter stemming algorithm. The preprocessed source code files are stored in a large text file to provide as input to the word embedding algorithms — word2vec, FastText, and GloVe.

Table 6.2.: Some pairs of words and their abbreviations sampled from the SoftwarePairs-400 benchmark.

Abbr.	**Word**	**Score**
del	delete	5
tmp	temporary	5
rght	right	2
min	minimum	5
num	number	5

Abbr.	**Word**	**Score**
med	median	3
col	column	5
tot	total	4
acc	accept	1
alloc	allocate	5

We used our own implementation of word2vec[5] for training on large software corpus. The FastText model is trained using the popular Gensim[6] implementation, while the GloVe[7] model is trained using the official implementation made available by the authors.

These models are trained on a multi-core machine with 20 processes for 20 epochs. The FastText and word2vec models take more time to train than the GloVe model. FastText and word2vec work by updating the weight vectors on each occurrence of a pair of terms in the corpus, while GloVe first collects the counts of all the pairwise occurrences of a pair of term in the corpus and then updates the weight vectors only once for that pair of terms. The word2vec and FastText models take 2-5 days to train depending on the size of vectors, while GloVe takes only several hours to a single day to train.

For more details on their implementation, please refer to the Appendix A.

Table 6.3.: Some words with their top 3 most (cosine) similar words as learned from the word2vec Skip-gram model.

rank	**alexnet**	**delete**	**rotation**	**add**	**parameter**
1	resnet	remove	angle	list	param
2	lenet	update	rot	set	method
3	imagenet	copy	lhornang	create	argument

6.4 How Good are the Software-centric Word Vectors?

To answer the question posed above we need to evaluate the quality of the word embeddings generated by the word2vec model for terms that occur in the software context. We, therefore, need a semantic similarity benchmark with which we can measure the quality of the word2vec generated word embeddings.

In this section we present an evaluation benchmark for word embeddings obtained for software-centric words. We also evaluate the software-centric word vectors obtained from word2vec model using our novel evaluation benchmark.

Semantic Similarity Benchmark

There exist in the natural language processing (NLP) research literature [62] many semantic similarity evaluation benchmarks for natural language words that are found in the articles of general interest, e.g. Wikipedia articles. There also exist domain-specific evaluation benchmarks for semantic word embeddings of a wide range of concepts in medicine, such as disease names, and medical procedures [63]. However, to the best of our knowledge, no such evaluation benchmark exists for software-centric

[5]https://github.com/sakbarpu/bme_project/word2vec.py
[6]https://radimrehurek.com/gensim/
[7]https://nlp.stanford.edu/projects/glove/

word embeddings. Therefore, we present in this section for the first time a semantic similarity evaluation benchmark exclusively for software-centric word vectors.

In NLP and also in the literature related to medicine, what these human-created benchmarks contain are pairs of words that are similar in meaning, and therefore, should have similar word vectors. For example, a pair could contain the words "female" and "woman", or "tumor" and "cancer", because they are semantically similar. These benchmarks also contain a human supplied semantic similarity score on a scale of 1 through 5 for each pair of such words. Therefore, to evaluate the word embeddings obtained from a model like word2vec, the word vectors of the semantically similar words in the list of pairs in the benchmark are compared against each other using a similarity measure, say the cosine similarity. Subsequently, these calculated similarity scores can be compared with the human-supplied scores to measure the quality of the embeddings.

With inspiration drawn from prior research in semantic word embeddings in NLP and medicine, we have created a benchmark called SoftwarePairs-400 for terms occurring specifically in the software programs. In our novel benchmark, we carefully compiled a list of pairs of 400 words along with their commonly used abbreviations in programming languages. A sample from this list of pairs along with their respective human-assigned scores is given in Table 6.2.

Evaluation

We now evaluate the word embeddings produced by word2vec for software-centric words using two evaluation metrics — correct @r ($C@r$), and Pearson correlation score. The correct at r ($C@r$) metric is defined as the number of pairs in the list of 400 pairs, in which the abbreviation appears in the top r ranked positions when the vector for the term corresponding to the abbreviation is compared with all the vectors in the database using cosine similarity. The higher the value for $C@r$ the better is the word embedding model. On the other hand, the Pearson correlation

Table 6.4.: Evaluation results on semantic similarity benchmark SoftwarePairs-400 for Skip-gram and CBoW models while changing N, which is the dimension of the word vectors.

Model (N)	**C@1** (%)	**C@5** (%)	**C@10** (%)	**Correlation**
SG (1500)	105 (26%)	140 (35%)	161 (40%)	0.221
SG (1000)	112 (28%)	153 (38%)	172 (43%)	0.224
SG (500)	23 (5%)	52 (13%)	62 (15%)	0.108
SG (200)	19 (4%)	42 (10%)	53 (13%)	0.128
CBoW (1500)	38 (9%)	61 (15%)	69 (17%)	0.010
CBoW (1000)	45 (11%)	63 (15%)	67 (16%)	0.020
CBoW (500)	15 (3%)	37 (9%)	57 (14%)	0.053
CBoW (200)	19 (4%)	45 (11%)	53 (13%)	0.141

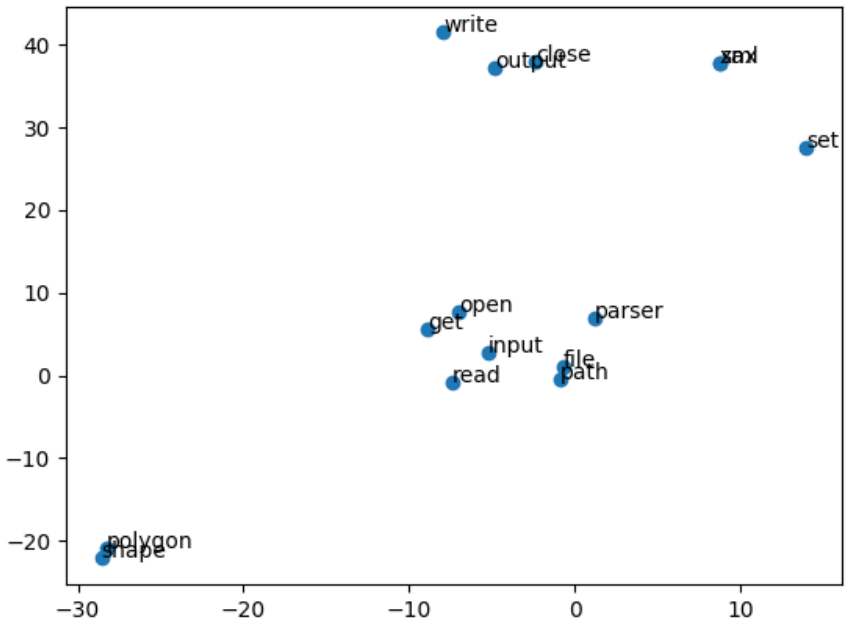


Fig. 6.4.: Zoomed-in view of the word vectors in the first two dimensions. Note the clusters: (“write”, “output”, “close”), (“open”, ”read”, “input”, “file”), (“polygon”, “shape”), (“xml”, “sax”)

score is calculated by comparing the human-assigned scores to the pairs of words, with the cosine similarity values obtained when the vectors of words present in the pair are compared against each other. Higher correlation values imply better word embedding models.

In Table 6.4 we provide evaluation results on SoftwarePairs-400 for two different models: (1) Skip-gram (SG), and (2) Continuous Bag of Words (CBOW). Both the models are run with four different values of N — 200, 500, 1000, or 1500. Our results show that the Skip-gram model with $N = 1000$ produces the best values for $C@r$. It can also be observed from the experimental results that CBoW performs worse than Skip-gram in terms of $C@R$. Also, the SG model with $N = 1500$ performs worse than SG model with $N = 1000$. Therefore, our results show that increasing the number of dimensions of word vectors beyond 1000 actually reduces the performance of the model in identifying similar meaning terms present in SoftwarePairs-400.

The reason for evaluating the quality of word vectors is simply to ensure that the terms with similar meanings have their vectors close to each other in the semantic vector space. As we will show in Chapter 8, the performance of a semantics based retrieval engine is not significantly affected by which word embedding algorithm is used to construct the semantic vector space.

Visualization

In addition to quantitatively evaluating the word embeddings as described in the previous subsection, we can visualize them after their dimensionality is reduced with an algorithm like PCA (Principal Component Analysis). After such a step, usually one retains only a very small number of dimensions, typically 2 or 3.

A zoomed-in view for a few software-centric words when considering only the top two dimensions of PCA is shown in Figure 6.4. It can be readily observed from the figure that the words with similar meanings form distinguishable clusters in the 2-dimensional PCA space.

In addition to visualization, in Table 6.3 we show five interesting software terms in the first row along with their three most cosine similar terms in the following three rows.

Now that we have established that meaningful vector representations of software terms can be constructed using semantic word embedding algorithms, in the next chapter, we will exploit these representations to build a powerful code search tool and evaluate its performance on bug localization task.

7. SCOR: SOURCE CODE RETRIEVAL WITH SEMANTICS AND ORDER

As discussed in the previous chapter, representing textual words with their dense numeric vectors has emerged as a powerful approach for associating context-based meanings with the words and for comparing words for their similarity on the basis of such meanings. What is perhaps the most commonly used algorithm today for creating such embeddings is the word2vec algorithm proposed by Mikolov et al. [17].

Given the semantically rich representation for the words created by, say, word2vec, one is led to wonder if a retrieval framework based on such representations can be made even more powerful if it is subject to term-term ordering constraints modeled by, say, Markov Random Fields (MRF).

As we demonstrate in this chapter, the answer to the question posed above is a categorical yes.

The SCOR retrieval framework that we present in this chapter for establishing the above claim invokes MRF based ordering constraints on the query terms and the file terms that are matched on the basis of contextual semantics using the word2vec generated numeric vectors for the terms. This is facilitated by two "layers" that we refer as the "Match Layer (ML1)" and the "Match Layer 2 (ML2)". In the form of a 2D numeric array, ML1 is simply a record of the similarities between the terms in a query and the terms in a file, with the similarities being computed by applying the cosine distance measure to the numeric vectors produced by word2vec. Subsequently, in the spirit of convolutional neural networks, we convolve the ML1 layer with a 2×2 kernel — whose elements must possess certain pre-specified properties — to yield another 2D numeric array that is ML2. As we argue in this chapter, the numbers in the ML2 layer become high only for those sequences of terms in the query and a file, which is being evaluated for retrieval vis-a-vis the query, when there is significant

semantic similarity between the two both respect to the terms and with respect to the ordering constraints on the terms. As we show later, convolving the 2D array of numbers in ML1 with a 2×2 operator produces the same effect as what would be achieved with MRF based logic as presented in [6, 20].

In this chapter, we have established the superiority of the "semantics plus order" approach for source-code retrieval over the more traditional methods by comparing the following retrieval frameworks in the context of automatic bug localization: (1) an approach based on the BoW (Bag-of-Words) assumption; (2) an approach based on MRF modeling using term and term-term frequencies; (3) a retrieval framework that uses contextual semantics through the word embeddings produced by the word2vec algorithm; and, finally, (4) a framework that uses MRF modeling on top of the word embeddings produced by the word2vec algorithm.

While the first three retrieval methods in the comparison mentioned above refer to frameworks that are already well known, the last — which combines MRF with the word embeddings produced by word2vec — is something that has not been attempted before. Combining MRF with word embeddings allows us to jointly model the semantic and the ordering relationships in a single source code retrieval framework.

For our experiments, we use approximately 4000 bug reports of the Eclipse software library obtained from the BUGLinks dataset and approximately 300 bug reports of AspectJ obtained from the popular iBUGS dataset. We report results based on retrievals with (1) just the titles of the bug reports as queries; and (2) the entire description of the bug reports as queries. With both software libraries, we show that the retrieval precision can be improved between 6% and 30% over the best results that can be obtained with the more traditional applications of MRF to the representations based on term and term-term frequencies. In the next chapter we will use all the 29 software libraries present in the Bugzbook dataset considering title+description queries for evaluating the performances of eight retrieval algorithms.

Comparing our SCOR model with the best of what the literature has to offer, it significantly outperforms the purely MRF based framework presented in Sisman et al.

[6]. SCOR also outperforms the more advanced BoW based techniques — BugLocator [8], BLUiR [12], and SCP-QR [7]. We also compare our retrieval model with semantic embeddings based bug localization algorithms — LSA (Latent Semantic Analysis) [5], and Ye et al.'s semantic retrieval algorithm [15]. Our results show that our retrieval algorithm can significantly outperform the LSA algorithm presented in [5], while the performance of our retrieval algorithm with respect to Ye et al.'s [15] semantic retrieval algorithm is comparable in terms of retrieval precision.

We use the word embeddings obtained as described in the previous chapter after training the semantic word embedding algorithm on the large corpus of 35000 Java software repositories present in the SCOR word embeddings dataset. The reason for generating word vectors for such a large software vocabulary is to ensure that our semantic word embeddings are sufficiently generic so that they can be applied to new software repositories that were not used for generating the embeddings. Our results demonstrate that to be the case.

With this introduction the rest of the chapter is organized as follows. In the next section "Related Works", we present past works in the field of semantics based retrieval. In section 7.2 we explain our novel retrieval model. Finally, We present our experimental results in section 8.2.

7.1 Related Works

The notion of generating corpus-based semantic embeddings to model semantic relationships in a retrieval framework dates back to 1990's when Latent Semantic Analysis (LSA) was first published. Since then, LSA has successfully been applied to solve the problem of software search [5, 23].

The use of neural networks to learn semantic word embeddings was first proposed by Bengio et al. [58], with several modifications made to their neural network architecture by the contributions in [17, 59, 60]. From amongst these contributions,

the word2vec implementation of [17] is arguably the most successful and the most commonly used today.

Several authors have investigated using word2vec in different application domains [64–68] that include web search, questing-answering systems, and paraphrase identification. Authors have also reported using word2vec for software search [15, 33, 34, 69]. These contributions, however, do not include ordering constraints. Finally, note that researchers have also proposed using deep-learning based frameworks for solving the IR problem [70–75].

In contrast with the word2vec-based software search investigations as reported in [15, 33, 34, 68, 69], our SCOR framework [16] combines MRF based modeling with word2vec based semantic word embeddings to lead to a new class of retrieval algorithms that account for both order and semantic relationships between terms. Our SCOR model improves retrieval precision over pure MRF based approaches presented in [6], as well as over modern BoW models [8, 12].

7.2 Modeling Ordering and Semantic Relationships for Software Retrieval

In this section we present how we jointly model the two seemingly disparate aspects of our framework for source code retrieval: the term-term ordering constraints and the word2vec generated semantic relationships between the terms. The term-term ordering constraints are imposed using the MRF framework described by Sisman et al. [6], and described in Chapter 5, whereas the semantic relationships between the terms are modeled by comparing the semantic word embeddings [17] of the query and the file terms using cosine similarity measure. The computation of semantic vectors for software terms is presented in Chapter 6.

7.2.1 Modeling Ordering Relationships Between Terms

In the context of IR based bug localization, a Markov Random Field is an undirected graph G in which one of the nodes represents a source-code file f that is being

evaluated for its relevance to a given query Q and all other nodes represent the individual terms $Q = \{q_1, q_2, ..., q_{|Q|}\}$ in the query. The arcs between the nodes represent probabilistic dependencies between the nodes [6].

The MRF framework gives us the liberty to choose different kinds of probabilistic dependencies we want to encode in the retrieval model. In Chapter 5 we showed three possible dependency assumptions that we can make about the construction of the MRF graph G. We will focus on the FI (Full Independence), and SD (Sequential Dependence) assumptions in this chapter. In FI assumption all the query terms are considered independent of one another. Therefore, the arcs between nodes that stand for the query terms q_1, q_2, and q_3 are absent. In SD assumption the nodes for the consecutive query terms are connected to each other via arcs. We use the SD model to incorporate term-term ordering relationships between the query terms that may be present in the file f.

As shown in Chapter 5, the Dirichlet smoothed BoW based Full Independence (FI) relevance score is calculated as:

$$score_{fi}(Q, f) = \sum_{i=1}^{|Q|} \log \frac{tf(q_i, f) + \frac{\mu_{fi} tf(q_i, C)}{|C|}}{(|f| + \mu_{fi})} \tag{7.1}$$

where $tf(q_i, f)$, and $tf(q_i, C)$ are, respectively, the frequencies of the term q_i in the source code file f and in the collection C. The notations $|f|$ and $|C|$ stand for the size of the file and the collection, respectively. Finally, μ_{fi} is the smoothing parameter.

Denoting a pair of sequential terms $q_i q_{i+1}$ by ρ, the MRF SD relevance score is calculated as:

$$score_{sd}(Q, f) = \sum_{i=1}^{|Q|-1} \log \frac{tf_w(\rho, f) + \frac{\mu_{sd} tf_w(\rho, C)}{|C|}}{(|f| + \mu_{sd})} \tag{7.2}$$

where $tf_w(\rho, f)$ and $tf_w(\rho, C)$ are, respectively, the frequencies of the pair of terms $\rho = q_i q_{i+1}$ in a file f and in the collection C. The notation μ_{sd} is for the smoothing parameter.

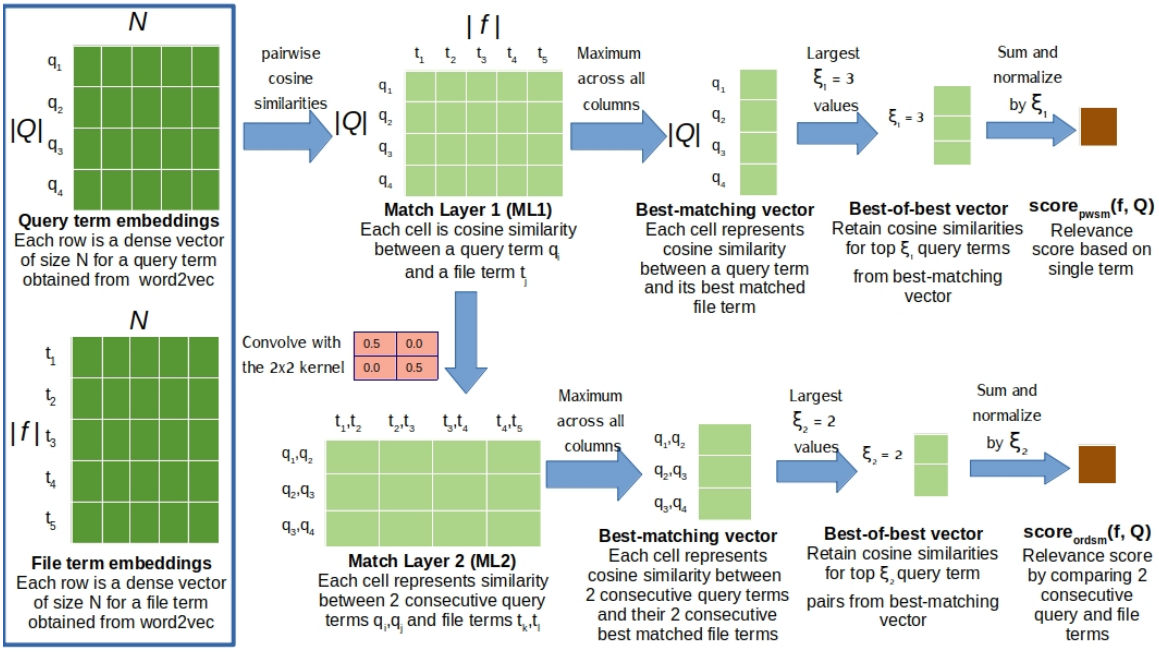


Fig. 7.1.: **An illustration of a semantic retrieval framework with ordering constraints:** The N-dimensional numeric vectors for the terms in a query Q and a file f are provided as inputs as shown at left. The query terms and file terms are compared pairwise using cosine similarity to produce ML1. As shown in the top row, we perform further processing on ML1 to produce relevance score based on matching terms individually $score_{pwsm}(f, Q)$. As shown in the bottom row we convolve ML1 with a pre-defined 2×2 kernel to produce ML2, which is subsequently processed to produce the relevance score $score_{ordsm}(f, Q)$.

7.2.2 Modeling Semantic Relationships Between Terms

With the help of Figure 7.1, we now illustrate using an example our semantic retrieval framework that includes ordering constraints. The example used in the figure assumes that the query consists of just four terms. That is, $Q = \{q_1, q_2, q_3, q_4\}$. And that a file being evaluated for its relevance to the query consists of just five terms. That is, $f = \{t_1, t_2, t_3, t_4, t_5\}$. We also assume that the dimensionality of the numeric term vectors provided by word2vec is N. As we explained earlier in Chapter 6, we obtain these vectors by training a word2vec Skip-gram model on a large corpus of Java source code repositories.

As shown in the figure at its left, the numeric vectors for the query terms and the file terms serve as inputs to the processing chain in Figure 7.1. The query terms q_i and the file terms t_j are compared pairwise using the cosine similarity measure shown below to produce the "Match Layer 1" (ML1):

$$\sigma_1(\mathbf{v}_{q_i}, \mathbf{v}_{t_j}) = \frac{\mathbf{v}_{q_i} \cdot \mathbf{v}_{t_j}}{||\mathbf{v}_{q_i}||||\mathbf{v}_{t_j}||} \tag{7.3}$$

Each row of ML1 corresponds to the cosine similarity between a given query term q_i and all the terms in the file f. The layer ML1 is then treated as described below to produce a *semantic* relevance score.

As shown in the top row of Figure 7.1, we take the maximum of ML1 across all the file terms to produce the "Best-matching vector". In this vector, we retain only the largest cosine similarity value for each query term that corresponds to its best matched file term. The cosine similarity values for all other file terms are discarded and, therefore, do not influence the final relevance score.

Afterwards, only the largest ξ_1 values from the "Best-matching vector" are retained. This second maximum selection operation discards the cosine similarity values for all those query terms that are deemed to not match the file sufficiently. In this manner, the terms that may negatively influence the final relevance score are dropped.

We refer to the vector obtained by the second maximum operation as the "Best-of-best vector" since it is obtained as a result of two successive maximum operations on array ML1 of cosine similarity values: (1) The first maximum is taken across all file terms for each query term to produce the best cosine similarity value for each query term that corresponds to its best-matched file term, (2) The second maximum is taken across all the query terms to produce a vector of cosine similarity values for only those ξ_1 query terms that best match the file. Notice that the parameter ξ_1 is tunable.

The retained ξ_1 cosine similarity values are summed and normalized to obtain the relevance score $score_{pwsm}(f, Q)$ based on our Per-Word Semantic Model (PWSM):

$$score_{pwsm}(f, Q) = \frac{1}{|Q'|} \sum_{q_i \in Q'} \underset{\substack{Q' \subseteq Q \\ |Q'| \leq \xi_1}}{\mathbf{argmax}} \sum_{q_i \in Q'} \underset{t_j \in f}{\mathbf{max}}\, \sigma_1(\mathbf{v}_{q_i}, \mathbf{v}_{t_j}) \tag{7.4}$$

where Q' is a subset of query Q composed of only those ξ_1 query terms for which we get the largest cosine similarities.

The great thing about the array of numbers in ML1 is that it lends itself to further processing that allows the incorporation of the MRF based term-term ordering constraints in how a query is matched with a file.

As shown in the bottom row of Figure 7.1, we now convolve the array of numbers in ML1 with a pre-defined 2×2 kernel K to produce another layer "Match Layer 2" (ML2). Each element in the array of numbers in ML2 represents the similarity measure between two consecutive query terms $q_i q_{i+1}$ and two consecutive file terms $t_j t_{j+1}$. This obviously implies incorporating ordering relationships between the query terms that semantically match the file terms.

The 2×2 kernel K mentioned above is designed such that it has non-zero values on its diagonal, while its off-diagonal elements are either zero or very close to zero. This is a key condition on K that allows the ordering constraints to be satisfied as explained below.

During the convolution of K with the array of numbers in ML1, the output at a specific location (i, j) results in a high value if the cosine similarity values between the terms q_i and t_j, and terms q_{i+1} and t_{j+1} are both high. The convolution operation, which results in a weighted sum over 2×2 neighborhoods in ML1, is similar to what is used in modern deep convolutional neural networks. The similarity values σ_2 in the ML2 are computed from the similarity values σ_1 of the ML1 as follows:

$$\begin{aligned}\sigma_2(q_i q_{i+1}, t_j t_{j+1}) = K_{11}\sigma_1(\mathbf{v}_{q_i}, \mathbf{v}_{t_j}) + K_{22}\sigma_1(\mathbf{v}_{q_{i+1}}, \mathbf{v}_{t_{j+1}}) \\ + K_{12}\sigma_1(\mathbf{v}_{q_i}, \mathbf{v}_{t_{j+1}}) + K_{21}\sigma_1(\mathbf{v}_{q_{i+1}}, \mathbf{v}_{t_j})\end{aligned}$$

The array of numbers in ML2 are treated in the same manner as those in ML1 to produce the ordered-semantic (ORDSM) relevance score $score_{ordsm}(f, Q)$ based on comparing two consecutive query with two consecutive file terms. Notice that we use a new parameter ξ_2 that is different from ξ_1 to obtain the "Best-of-best vector" from the "Match Layer 2" (ML2). The ordered-semantic (ORDSM) score is computed as:

$$score_{ordsm}(f, Q) = \frac{1}{|P'|} \sum_{\rho_i \in P'} \mathbf{argmax}_{\substack{P' \subseteq P \\ |P'| \leq \xi_2}} \sum_{\rho_i \in P'} \max_{e_j \in E} \sigma_2(\rho_i, e_j) \tag{7.5}$$

where $\rho_i \in P$ represents the pair of query terms $q_i q_{i+1}$, $e_j \in E$ represents the pair of file terms $t_j t_{j+1}$, and P and E are the sets of pairs of query terms and file terms, respectively.

The computation of $score_{pwsm}$ and $score_{ordsm}$ in the manner described above is one of the key contributions of this dissertation.

7.2.3 Computing a Composite Score for a Repository File

The previous two subsections presented different formulas for measuring the relevancy of a file to a query. The formulas in Section 7.2.1 showed how a file could be ranked vis-a-vis a query using just the BoW modeling of the relationship between the two and also using the MRF based ordering constraints. And the formulas in

Section 7.2.2 showed how a file could be ranked purely on the basis of the similarities of the term-contextual relationships and on the basis when ordering constraints are superimposed on top of the term-contextual relationships.

We now combine all those measures of relevancy of a file to query to create a composite file relevancy score. As shown below, this composite formula uses a weighted aggregation of the scores given by the Equations (7.1), (7.2), (7.4), and (7.5):

$$score_{scor}(f, Q) = \alpha \cdot score_{fi}(f, Q) + \beta \cdot score_{sd}(f, Q) + \\ \gamma \cdot score_{pwsm}(f, Q) + \eta \cdot score_{ordsm}(f, Q)$$

where α, β, γ, and η are tunable parameters.

To review quickly, $score_{fi}(f, Q)$ represents a simple BOW score in which frequencies of the query terms and the file terms are compared directly; $score_{sd}(f, Q)$ is the MRF based sequential dependence score in which the frequencies of pairs of consecutive query terms and the corresponding consecutive file terms are compared to impose ordering constraints for "exact" term matches; $score_{pwsm}(f, Q)$ is the relevance score computed using cosine similarity measures on the term numeric vectors for the word embeddings of the individual query terms and the file terms to incorporate semantic relationships for "inexact" term matches; and, finally, $score_{ordsm}(f, Q)$ represents the ordered semantic relevance score which is based on "inexact" matching between pairs of consecutive query terms and the corresponding consecutive file terms.

7.3 Experimental Results

With the framework we have presented in the preceding sections, we now present our experimental results on source-code retrieval for solving the problem of automatic bug localization. We report results using two different software libraries: Eclipse and AspectJ. The results for the Eclipse [76] and AspectJ [77] libraries are for two different types of retrievals: using just the titles of the bug reports as queries ("title-only"), and using the entire bug reports as queries ("title+desc").

The bug reports for Eclipse and AspectJ software libraries were obtained from the publicly available BUGLinks [44] and iBUGS [43] datasets, respectively. We presented the statistics related to these two datasets in Chapter 5. Notice that our large and diverse Bugzbook dataset that we discussed in Chapter 3 contains Eclipse and AspectJ bug reports taken from BUGLinks and iBUGS datasets.

Eclipse is a Java based software with a large number of bug reports (4035) in the BUGLinks dataset. The iBUGS dataset, on the other hand, contains relatively smaller number of bug reports (291) for AspectJ repository.

In what follows, we first present the overall processing pipeline of our bug localization framework. Then we describe the metrics used for evaluating the source code retrieval results. That is followed by a motivating example that illustrates the power of semantic modeling for retrieval. Lastly, we present our experimental results.

7.3.1 Overall Framework

Figure 7.2 shows the steps involved in our SCOR bug localization framework. The word2vec neural network shown at left takes for its input a very large corpus of Java source code repositories and generates semantic word embeddings for the software-centric words present in those repositories. Approximately 35000 open-source Java repositories were downloaded from GitHub [1] were used in this study. Except for a few that are now defunct, we downloaded the same repositories as those listed in [78]. In Chapter 6, Table 6.1 we showed the statistics of our Java source code dataset.

We used the popular Gensim library [79] to learn the word embeddings from the Java source-code dataset. The two important parameters that we can tune for training the Skip-gram model are: (1) vector size (N), and (2) window size (w). We set $N = 1000$, and $w = 8$ for all our retrieval experiments. The word vectors learned from the Skip-gram model are stored in a disk file.

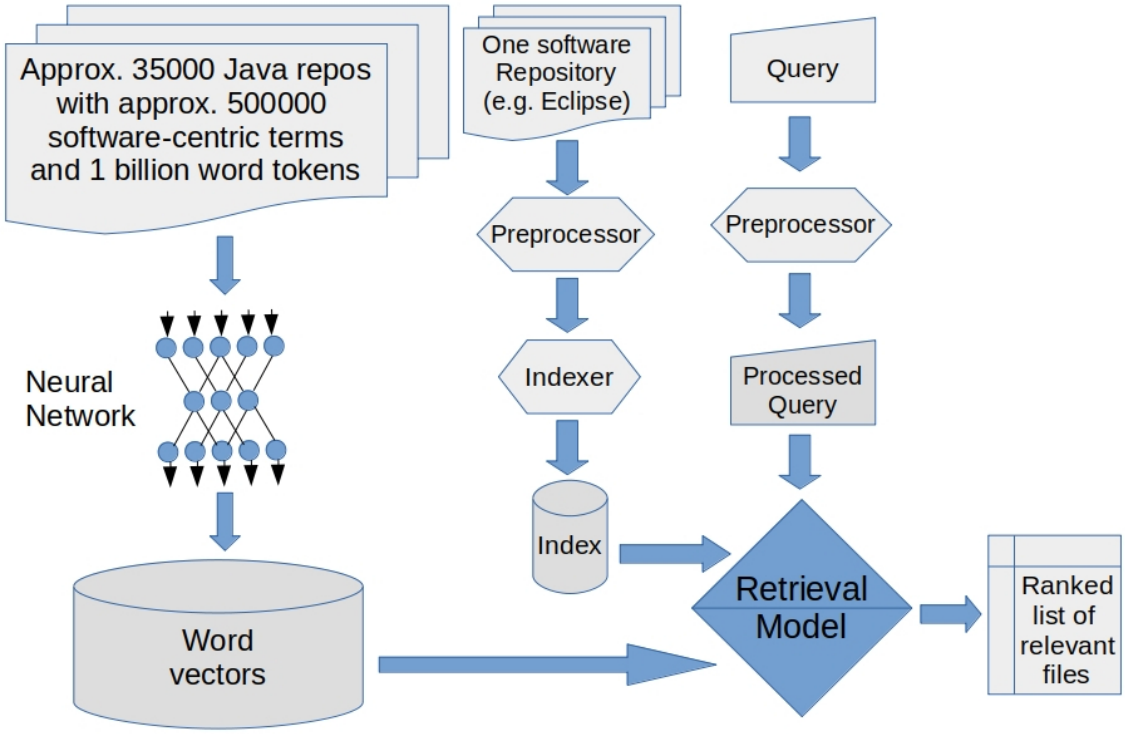


Fig. 7.2.: Block diagram for the retrieval framework.

The word embeddings along with the source-code files for the library (such as Eclipse) that we are interested, and also the bug reports, are fed into the retrieval engine.

The source-code files go through Porter stemming and stop word removal. Subsequently, each source-code file is indexed, an inverted index constructed from the main index, and the two stored in hash tables. On the other hand, the bug reports are first subject to regular-expression based testing for the detection of stack traces or code patches. If these are absent, further testing is carried out for the detection of camel-cased source code identifiers. In the absence of such identifiers, the bug reports are subject to the same preprocessing steps as the source code files.

7.3.2 Evaluation Metrics

We use precision based metrics — specifically MAP (Mean Average Precision), $P@r$ for precision at rank r, and $R@r$ for recall at rank r — for a quantitative characterization of the performance of the different retrieval models. MAP and $P@1$ would generally be considered to be the most important metrics for evaluating the power of retrieval algorithms of the type under investigation here [57].

In order to determine whether the improvement observed in the retrieval results obtained using one model vis-a-vis another is significant, we carried out significance testing based on the Student's Paired t-Test [80].

7.3.3 A Motivating Example

In this section we use a simple retrieval exercise to show the power of semantic modeling in a source code retrieval system. We use the title of a bug report with ID 106140 filed for the Eclipse software library as a query to the two retrieval models — FI-BOW and PWSM — and compare their results. For simplification of our analysis we ignore the description of the bug that is a part of the bug report.

The title of bug ID 106140 reads "[compiler] Eclipse3.1.0: unrecognized class invisibility", which after preprocessing produces the term tokens ["compiler", "eclipse", "unrecognized", "class", "inivisibility"]. The Eclipse source code files that were fixed in response to this bug are:

1. org.eclipse.jdt.core/compiler/../lookup/Scope.java

2. org.eclipse.jdt.core.tests.compiler/../LookupTest.java

Examining the top-ranked 100 retrievals, while the average precision using the FI-BoW model for this bug report is just 0.0, when semantic relationships between terms are modeled using Per-Word Semantic Model (PWSM) the AP increases slightly to 0.03. What that means is that while FI BoW could not retrieve any of the two relevant files, the PWSM model could retrieve the "Scope.java" file.

Table 7.1.: Summary of retrieval models along with the parameter settings that yielded the best results for each.

Method Name, Modeling scheme	Parameters Eclipse (title)	Parameters Eclipse (title+desc)	Parameters AspectJ (title)	Parameters AspectJ (title+desc)
SCOR, FI BoW + MRF SD + PWSM + ORDSM	$w = 8$ $\mu_{fi} = 1k$ $\mu_{sd} = 8k$ $\xi_1 = 10$ $\xi_2 = 3$ $\alpha = 0.32$ $\beta = 0.12$ $\gamma = 3.2$ $\eta = 25$	$w = 8$ $\mu_{fi} = 1k$ $\mu_{sd} = 4k$ $\xi_1 = 10$ $\xi_2 = 3$ $\alpha = 0.3$ $\beta = 0.12$ $\gamma = 2.5$ $\eta = 30$	$w = 8$ $\mu_{fi} = 3.5k$ $\mu_{sd} = 4k$ $\xi_1 = 10$ $\xi_2 = 5$ $\alpha = 0.3$ $\beta = 0.15$ $\gamma = 2.8$ $\eta = 25$	$w = 8$ $\mu_{fi} = 3.5k$ $\mu_{sd} = 4k$ $\xi_1 = 10$ $\xi_2 = 5$ $\alpha = 0.7$ $\beta = 0.04$ $\gamma = 2.8$ $\eta = 27$
MRF SD, FI + SD	$w = 8$ $\lambda_{sd} = 0.2$ $\mu = 4k$	$w = 20$ $\lambda_{sd} = 0.2$ $\mu = 4k$	$w = 2$ $\lambda_{sd} = 0.2$ $\mu = 4k$	$w = 16$ $\lambda_{sd} = 0.2$ $\mu = 4k$
PWSM, FI BoW + PWSM	$\mu_{fi} = 2.5k$ $\xi_1 = 10$ $\gamma = 0.95$	$\mu_{fi} = 4k$ $\xi_1 = 10$ $\gamma = 0.9$	$\mu_{fi} = 3.5k$ $\xi_1 = 10$ $\gamma = 0.95$	$\mu_{fi} = 3.5k$ $\xi_1 = 10$ $\gamma = 0.8$
FI BOW, Dirichlet smoothing	$\mu = 4000$	$\mu = 4000$	$\mu = 4000$	$\mu = 4000$

Investigating further, we found that while the non-discriminatory query terms "compiler", "eclipse", and "class" did appear in the "Scope.java" source code, the important discriminatory terms "unrecognized", and "invisibility", which could rank "Scope.java" higher than the other files containing the non-discriminatory terms, did not. Examining the "Scope.java" more closely, we found that even though it did not contain the exact terms "unrecognized" and "invisibility", it did contain their semantically related terms "missed", and "visible". In fact, the term "visible" appeared approximately 100 times in the file. Therefore, when "Scope.java" is scored using PWSM, the semantic matching between terms "visible" and "invisibility" makes the overall score for the file higher than the other files which only contained the non-discriminatory query terms.

7.3.4 Retrieval Experiments

We provide experimental results for comparing the following four retrieval models: (1) FI BOW, (2) MRF SD, (3) PWSM, (4) SCOR. Table 7.1 displays the tunable parameter values used for each model. These parameters were set to yield the best performance from each model.

Through our experiments we attempt to answer the following important research questions (RQs):

RQ1: Does the IR model that only incorporates ordering relationships improve the result over BoW IR models?

RQ2: Does the IR model that only incorporates semantic relationships improve the result over BoW IR models?

RQ3: How does our novel SCOR retrieval model perform against various BoW based source code retrieval models?

RQ4: How good is SCOR against pure MRF based retrieval techniques that only model ordering relationships?

Table 7.2.: Retrieval accuracy for the "title-only" queries.

Method	Eclipse			
	MAP (%) (p-value)	P@1	P@5	R@10
SCOR	**0.2709 (32.8%)**$^{\mp *}$ (3e-20)	**0.238**	**0.115**	**0.332**
FI + MRF SD	0.2493 (22.2%)* (1e-10)	0.211	0.108	0.315
FI + PWSM	0.2336 (14.5%)* (2e-5)	0.192	0.101	0.297
FI BoW	0.2039	0.169	0.088	0.258
	AspectJ			
SCOR	**0.1802 (44.2%)**$^{\mp *}$ (8e-3)	**0.206**	0.092	0.193
FI + MRF SD	0.1348 (7.9%)* (6e-2)	0.134	0.079	0.167
FI + PWSM	0.1641 (31.3%)* (4e-2)	0.164	**0.094**	**0.197**
FI BoW	0.1249	0.137	0.070	0.149

$*$ significantly different from FI BOW. $\mp$ significantly different from MRF SD.

RQ5: How does our SCOR retrieval model perform against other semantic embeddings based retrieval models?

RQ6: Are word2vec based word vectors generic enough to be used for searching in a new software library?

To answer the questions that follow we will frequently refer to Tables 7.2 and 7.3. Table 7.2 shows the evaluation results of the four different retrieval algorithms on two different datasets — Eclipse and AspectJ — under "title-only" setting, while Table 7.3 shows the same for "title+desc" queries. In the tables we show the percentage improvements and the p-values for each of the retrieval models vis-a-vis the FI BOW model along with the MAP values.

RQ1: Does the IR model that only incorporates ordering relationships improve the result over simple BoW IR model?

The IR model that only incorporates ordering relationships, i.e. MRF SD model, significantly beats the traditional FI BoW model in all four experiments — Eclipse title-only, AspectJ title-only, Eclipse title+desc, and AspectJ title+desc.

The improvement observed using MRF SD over FI BoW model ranges from 6% to 22% in different experiments with respect to MAP values. The $P@r$ and $R@r$ values are also significantly higher for MRF SD vis-a-vis FI BOW. We also observe that MRF SD model shows more improvement when run against Eclipse dataset as opposed to AspectJ dataset.

RQ2: Does the IR model that only incorporates semantic relationships improve the result over simple BoW IR models?

To answer the question posed above we again refer to Tables 7.2 and 7.3. As we can observe from the tables the PWSM model that only incorporates term-term semantic relationships significantly outperforms the traditional FI BoW model with improvements ranging from 9% to 31% in terms of MAP values. The values for $P@r$ and $R@r$ are also significantly higher for PWSM in relation to BoW model.

The biggest improvement is observed in the retrieval experiment performed for the Aspectj software library using title-only queries. It can also be observed that the improvements for title+desc experiments are lower than the improvements for title-only experiments.

RQ3: How does our novel SCOR retrieval model perform against various BoW IR models?

This question requires comparing our SCOR retrieval model against various BoW based source code retrieval models — FI BOW, BugLocator [8], BLUiR [12], and SCP-QR [7].

Note that SCOR and FI BoW results are provided in Tables 7.2 and 7.3. When comparing SCOR against the simple FI BOW model we observe that the improvements observed in term of MAP values, by SCOR over FI BOW range from 17% for AspectJ title+desc queries to even 44% for AspectJ title-only queries.

BugLocator takes into account past bug history to identify similar bug reports. The relevance of a source code file to a bug report is then determined using simple BOW based retrieval combined with an analysis of the source code files that were fixed in response to the past similar bug reports. BLUiR (Bug Localization Using information Retrieval) divides the source code file into different components — classes, methods, variables, and comments — and assign different weights to each component based on their importance in the retrieval process. Sisman and Kak [7] introduced SCP-QR which is a Query Reformulation method based on the Spatial Code Proximity of non-query terms with the query terms inside the source code files.

For comparing SCOR with more advanced retrieval engines — BugLocator [8] and BLUiR [12] — we refer to the Figure 7.3. We observe from the chart that for Eclipse title+desc dataset our SCOR retrieval algorithm with MAP value of 0.320 beats BugLocator, which has a MAP value of 0.310, while performs comparably to BLUiR, which has a MAP value of 0.320. Also, for AspectJ title+desc dataset, SCOR with MAP value of 0.255 outperforms BugLocator, which have MAP values 0.220 and 0.250, respectively. However, SCOR and BLUiR retrieval accuracies on AspectJ dataset are quite comparable.

Finally, we compare SCOR with SCP-QR [7]. The MAP value obtained using SCP-QR for 4035 Eclipse title-only queries is 0.2296. In comparison, SCOR with MAP value of 0.2709 outperforms SCP-QR by 18%.

RQ4: How good is SCOR against pure MRF based retrieval techniques that only model ordering relationships?

The answer to this question requires comparing SCOR with the MRF based Sequential Dependence (SD) and Full Dependence (FD) source code retrieval models presented in [6], and described in Chapter 5. Notice that the same MRF SD model used in [6] is an important component inside our SCOR retrieval framework.

As shown in the retrieval results provided in Figure 7.4 our SCOR retrieval model significantly outperforms both MRF SD and FD models on Eclipse and AspectJ title-only as well as title+desc queries. The improvements observed using SCOR over

Table 7.3.: Retrieval accuracy for the "title+desc" queries.

Method	Eclipse			
	MAP (%) (p-value)	P@1	P@5	R@10
SCOR	**0.3204 (29.1%)**$^{\mp *}$ (6e-20)	**0.289**	**0.134**	**0.394**
FI + MRF SD	0.3034 (22.2%)* (5e-31)	0.272	0.127	0.374
FI + PWSM	0.2713 (9.3%)* (1e-3)	0.233	0.116	0.341
FI BoW	0.2481	0.210	0.106	0.317
	AspectJ			
SCOR	**0.2506 (17.6%)**$^{\mp *}$ (1e-3)	**0.299**	**0.127**	**0.299**
FI + MRF SD	0.2263 (6.24%)* (2e-2)	0.264	0.114	0.265
FI + PWSM	0.2334 (9.5%)* (1e-2)	0.244	0.125	0.283
FI BoW	0.2130	0.244	0.105	0.243

$*$ significantly different from FI BOW. $\mp$ significantly different from MRF SD.

MRF SD range from 5.6% for Eclipse title+desc queries to 33% for AspectJ title-only queries, while the improvements observed using SCOR over MRF FD range from 5.6% for AspectJ title+desc to 27% for AspectJ title+desc queries. With regard to significance testing with the Student's Paired t-test [80], the p-values obtained when comparing SCOR with MRF SD model for different experiments are as follows: 3e-2 for Eclipse title+desc, 4e-3 for Eclipse title-only, 3e-2 for AspectJ title+desc, and 3e-2 for AspectJ title-only.

RQ5: How does our SCOR retrieval model perform against other semantic embeddings based retrieval models?

To answer this question we compare our retrieval algorithm with two semantic embeddings based retrieval algorithms — the neural embeddings based retrieval algorithm proposed by Ye et al. [15], and Latent Semantic Analysis (LSA) based source code retrieval model proposed by Rao and Kak [5].

Notice that while the model presented in [15] is composed of many relevance scores that linearly combine to produce a single composite score, we limit our focus only to the semantic portion of the model. Therefore, for the purpose of a comparative study, we only implemented the semantic embedding based scoring mechanism presented in [15] and linearly combined it with our baseline Dirichlet smoothed FI BoW model as provided in Equation (7.1). We compare this model against our PWSM retrieval model instead of the more powerful SCOR retrieval engine. We believe that such a comparison is fair in nature. Notice that we use the same word vectors we learn by applying the Skip-gram to 35000 repositories for both the algorithms. We notice that the MAP value obtained for PWSM for Eclipse title+desc is 0.2713, while the MAP value for Ye et al.'s algorithm is 0.2687.

With regards to comparing with LSA model proposed in [5], we notice that Rao and Kak used the same iBUGS AspectJ queries in their study as we have used in this chapter. Therefore, a direct comparison is possible between PWSM and the models presented in [5]. The best MAP value reported by Rao and Kak for LSA on iBUGS using 291 queries is 0.0700, while the MAP value for PWSM on the same dataset with the same 291 queries is 0.2334.

RQ6: Are word2vec based word vectors generic enough to be used for searching in a new software library?

The answer to this question is yes, because the AspectJ library on which we perform retrievals using iBUGS queries was not present in the training set when we generated the word embedding using the Skip-gram model. Yet, the improvements observed in retrieval precision when semantic embeddings based models are used for searching in the AspectJ library are very impressive as shown in Tables 7.2 and 7.3.

In the next chapter on large-scale evaluation of retrieval algorithms for bug localization, we will see more examples of software libraries that were not present in the training corpus of word embedding algorithms. Still, the performance of semantic based retrieval algorithms improve when generic word embeddings are used in the process.

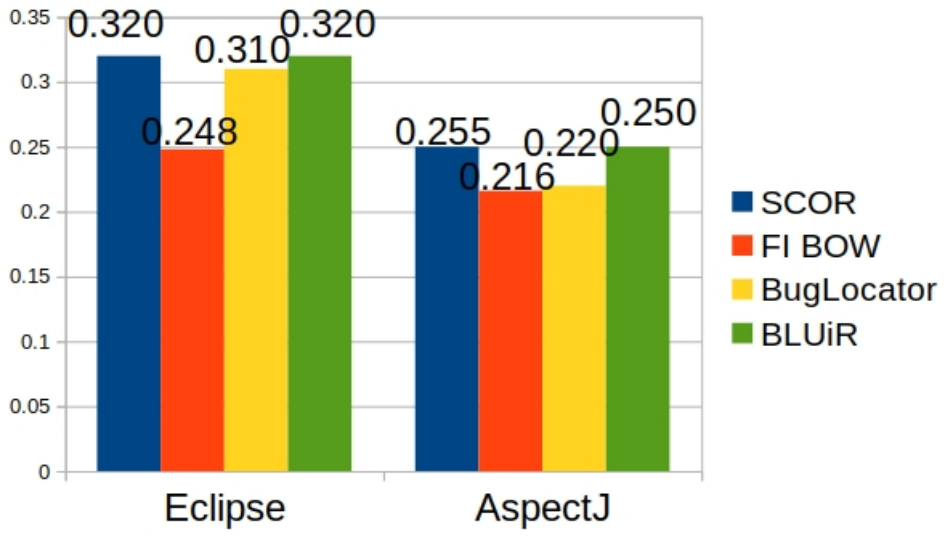


Fig. 7.3.: Comparison of SCOR with various BoW models on Eclipse and AspectJ "title+desc" queries using MAP values.

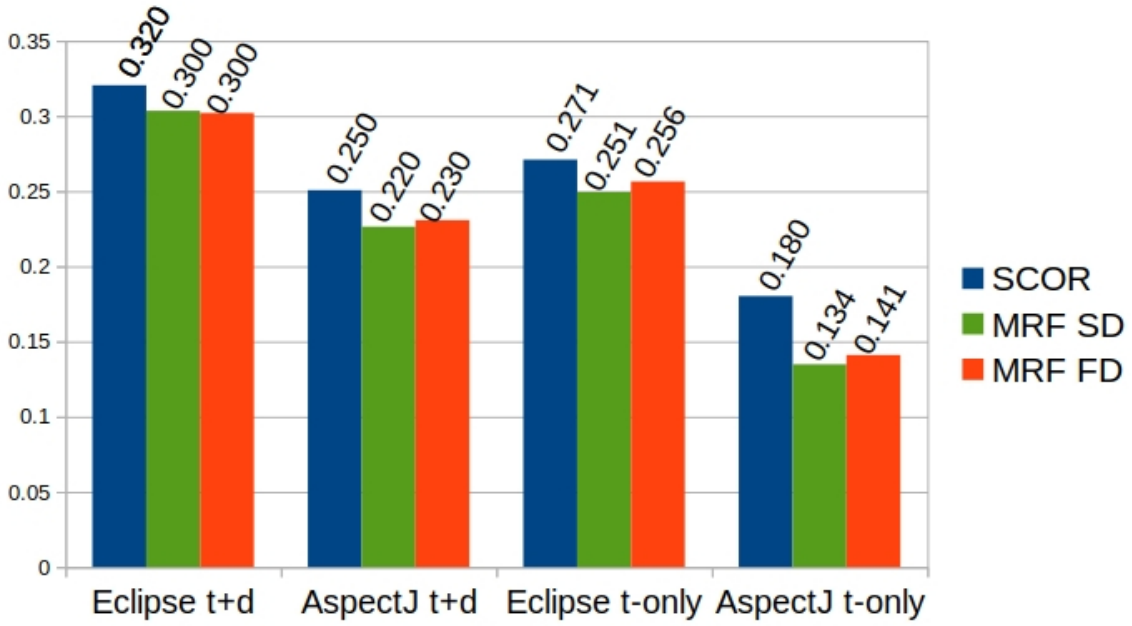


Fig. 7.4.: Comparison of SCOR with MRF SD and FD models on Eclipse and AspectJ queries using MAP values.

8. EXPERIMENTAL EVALUATION: A LARGE SCALE STUDY OF IR TOOLS FOR BUG LOCALIZATION

Retrieving relevant source code files from software libraries in response to a bug report query plays an important role in the maintenance of a software project. We showed a timeline of past research in bug localization in Chapter 2, and noticed that the past fifteen years have witnessed the publication of several algorithms for an IR based approach to solving this problem. An examination of these prior contributions reveals that (1) They mostly used datasets of relatively small sizes for the evaluation of the proposed algorithms; and (2) The datasets used consisted mostly of Java-based projects.

Except for the evaluations presented in [40], [42], and [37], all other publications performed their experiments using datasets that range in sizes from a few hundred to a few thousand bug reports.

Regarding the three studies mentioned above that performed relatively large-scale evaluations, Ye et al. [42] performed their bug localization experiments on around 20,000 bug reports taken from six Java projects. The study presented in [37] was performed on 8000 bug reports belonging to three Java and C/C++ based projects. The most recent large-scale comparative study carried out by Lee et al. [40] used around 9000 bug reports, all belonging to Java-based projects. These three studies, however, evaluate bug localization methods belonging only to the first and the second generations of tools, and are mostly focused toward Java based projects. Therefore, a large-scale bug localization study that involves code libraries in multiple languages and that includes all three generation of tools has yet to be carried out. The goal of this chapter is to remedy this shortcoming.

We present a comprehensive large-scale evaluation of a representative set of IR-based bug localization tools with the set spanning all three generations. The eval-

uation dataset we use, named Bugzbook, consists of over 20,000 bug reports drawn from a diverse collection of Java, C/C++, and Python software projects at GitHub. The features, construction and statistics of Bugzbook are presented in Chapter 3. A large-scale evaluation such as the one we report here is important because it is not uncommon for the performance numbers produced by testing with a large dataset to be different from those obtained with smaller datasets.

For the large-scale evaluation reported here, we chose eight search tools, one from each generation of the now 15-year history of the development of such tools. As mentioned previously, the earliest of the tools — the first-generation tools — are based solely on BoW modelling in which the relevance of a file to a bug report is evaluated by comparing the frequencies of the terms appearing in the file with the frequencies of the terms appearing in the bug report. In general, a BoW approach may either be deterministic or probabilistic. For the deterministic versions of such tools, we chose the TFIDF (Term Frequency Inverse Document Frequency) approach presented in [50]. And, for probabilistic BoW, we chose what is known as the FI (Full Independence) version of the framework based on Markov Random Fields (MRF) [6, 20]. The probabilistic version is also referred to as the Dirichlet Language Model (DLM) [49]. We have discussed DLM and TFIDF in greater detail in Chapter 4.

For representing the second generation tools, we chose BugLocator [8] and BLUiR (Bug Localization Using information Retrieval) [12]. In addition to the term frequencies, these tools also exploit the structural information (when available) and information related to the revision history of a software library.

That brings us to the third generation tools that, in addition to the usual term frequencies, also take advantage of term-term order and contextual semantics in the source-code files, on the one hand, and in the bug reports, on the other. We have used the algorithms described in [6] and [16] to represent this generation of tools.

In addition to generating the usual performance numbers for the algorithms chosen, our large-scale evaluation also provides answers to the six research questions that are listed in Section 8.2.3 of this chapter. Most of these questions deal with the

Table 8.1.: Comparison of the different bug localization tools based on the logic components used for ranking files.

	TI	**DLM**	**BL**	**BR**	**MRF**		**SCOR**	
					SD	FD	PWSM	SCOR
BoW	✓	✓	✓	✓	✓	✓	✓	✓
Order					✓	✓		✓
Semantic							✓	✓
Trace					✓	✓	✓	✓
Structure				✓				
Past Bugs			✓					

TI - TFIDF, DLM - Dirichlet LM, BL - BugLocator, BR - BLUiR

relative importance of the different components of the algorithms that belong to the second and the third generation of the tools.

At this point, the reader may ask: What was learned from our large-scale multi-generational evaluation that was not known before? To respond, here is a list of the new insights we have gained through our study:

1. Contrary to what was reported earlier, the retrieval effectiveness of two different ways of capturing the term-term dependencies [6] in software libraries — these are referred to as MRF-SD and MRF-FD — is the same.

2. The performance of second generation tools BugLocator and BLUiR are not equivalent in terms of retrieval precision, contradicting the finding presented in [40].

3. Including software libraries in different languages (Java, C++, and Python) in our study has led to a very important new insight: for the contextual semantics needed for the third-generation tools, it is possible to use the word embeddings

generated for one language for developing a bug localization tool in another language. We refer to this as the "cross-utilization of word embeddings."

Note that these are just the high-level conclusions that can be made from the answers to the six questions listed in Section 8.2.3.

8.1 Catalog of the Bug Localization Tools in Our Evaluation

The comparative evaluation we report in this chapter involves the following bug localization tools:

1. TFIDF: TFIDF (Term Frequency Inverse Document Frequency) [50] works by combining the frequencies of query terms in the file (TF) and the inverse document frequencies of the query terms in the corpus (IDF) to determine the relevance of a file to a query.

2. DLM: DLM (Dirichlet Language Model) [6, 49] or FI (Full Independence) BoW is a probabilistic model that estimates a smoothed first order probability distribution of the query terms in the file to produce the relevance score for the file given the query.

3. BugLocator: BugLocator [8] takes into account the history of the past bug reports and leverages similar bug reports that have been previously fixed to improve bug localization performance.

4. BLUiR: BLUiR (Bug Localization Using information Retrieval) [12] extracts code entities such as classes, methods, and variable names from source code files to help in localizing a buggy file.

5. MRF SD: MRF (Markov Random Field) based SD (Sequential Dependence) model [6] measures the probability distribution of the frequencies of the pairs of *consecutively* occurring query terms appearing in the file to compute the relevance score for the file given a query.

6. MRF FD: MRF based FD (Full Dependence) [6] is a term-term dependency model that considers frequencies of *all* pairs of query terms appearing in the file to determine the relevance of the file to the query.

7. PWSM: PWSM (Per-Word Semantic Model) [16] uses word embeddings derived from the word2vec algorithm to model term-term contextual semantic relationships in retrieval algorithm.

8. SCOR: SCOR (Source code retrieval with semantics and order) [16] combines the MRF based term-term dependency modeling, as described in [6], with semantic word embeddings as made possible by word2vec [17] to improve bug localization performance.

In Table 8.1, we compare the bug localization tools in our evaluation based on the logic components they use to produce the relevance score for a file vis-a-vis a given bug report.

8.2 Experimental Results

This section presents the experimental results of our large-scale study on the effectiveness of modeling source code repositories using first, second, and third generations of bug localization tools. Also presented in this section is the evaluation metric used.

8.2.1 Implementation Details

For the first generation tools — DLM (FI BoW) and TFIDF — we used the implementations provided by the popular open-source search engine tool Terrier [81].

For the second generation tools we used the implementations of BugLocator [82], and BLUiR [83] that have been made available online by the authors of the tools. For these tools, we used the parameter settings as suggested by the same authors.

For the third generation tools, for MRF-SD and MRF-FD we used the implementations that are built into Terrier engine. And for PWSM and SCOR, we used

the implementation provided by the authors of [16]. This implementation uses the DLM-FI BoW model as the baseline model upon which enhancements are made to introduce the semantic and ordering relationships between the terms. For the word embeddings needed by PWSM and SCOR, we make use of the semantic word vectors obtained after training word embedding algorithms on the SCOR dataset as described in Chapter 6. These word vectors can be downloaded from the website where we have posted the embeddings for half a million software terms [84].

We use the parameters recommended by the authors of the respective tools to evaluate their performances on bug localization dataset.

8.2.2 Evaluation Metrics

We use the Mean Average Precision (MAP) values to evaluate the performance of retrieval algorithms. This metric is the mean of the Average Precisions (AP) calculated for each of the bug report queries. The MAP values are subject to statistical significance testing using the Student's Paired t-Test. Significance testing tells us whether the measured difference in the results obtained with two different retrieval models is statistically significant. Student's t-Test has been used in previous studies [6, 8] to establish the performance gain of one algorithm over another.

Table 8.2.: MAP values for the retrieval algorithms evaluated on Bugzbook dataset.

Project	MI	TFIDF	DLM	BL	BR	SD	FD	PWSM	SCOR
Ambari	1.98	0.268	0.227	0.257	0.242	0.268	0.278	0.253	0.295
Aspectj	1.07	0.211	0.216	0.220	0.250	0.226	0.230	0.233	0.250
Bigtop	0.47	0.456	0.079	0.080	0.567	0.304	0.300	0.110	0.560
Camel	1.90	0.390	0.369	0.345	0.345	0.405	0.382	0.395	0.407
Cassndr	1.69	0.364	0.394	0.361	0.394	0.367	0.310	0.356	0.411
Cxf	1.46	0.332	0.287	0.319	0.303	0.348	0.342	0.329	0.363
Drill	1.58	0.196	0.210	0.170	0.169	0.218	0.189	0.223	0.240
Eclipse	1.30	0.284	0.248	0.310	0.320	0.303	0.305	0.271	0.320
HBase	1.74	0.387	0.362	0.370	0.333	0.429	0.424	0.408	0.453
Hive	1.77	0.332	0.278	0.219	0.224	0.335	0.346	0.269	0.345
JCR	1.88	0.432	0.396	0.417	0.394	0.453	0.454	0.437	0.450
Karaf	1.80	0.372	0.386	0.348	0.382	0.374	0.332	0.399	0.427
Mahout	1.27	0.320	0.295	0.481	0.367	0.315	0.267	0.320	0.338
Math	0.81	0.482	0.495	0.601	0.557	0.454	0.481	0.458	0.512
Opennlp	1.06	0.435	0.456	0.500	0.261	0.433	0.347	0.437	0.498
PDFBox	1.57	0.351	0.319	0.430	0.370	0.368	0.357	0.358	0.380
PIG	0.85	0.285	0.228	0.360	0.315	0.295	0.312	0.311	0.335
SOLR	1.20	0.343	0.305	0.323	0.331	0.371	0.370	0.344	0.394
Spark	1.77	0.339	0.369	0.398	0.348	0.377	0.332	0.362	0.418
Sqoop	1.40	0.358	0.385	0.379	0.406	0.367	0.307	0.322	0.417
Tez	1.48	0.373	0.373	0.376	0.277	0.424	0.428	0.401	0.431
Tika	1.27	0.341	0.270	0.375	0.411	0.290	0.316	0.333	0.326
Wicket	1.99	0.439	0.420	0.489	0.411	0.450	0.389	0.399	0.440
WW	1.34	0.397	0.354	0.288	0.226	0.414	0.379	0.376	0.430
Zkeepr	0.99	0.468	0.494	0.502	0.456	0.565	0.527	0.532	0.529
MAP_{avg} (Java)		**0.358^{dr}_{dr}**	**0.329**	**0.357^{drf}_{dr}**	**0.346^{d}**	**0.366^{tdlrfp}_{tdlrp}**	**0.348^{dr}_{dr}**	**0.345^{d}_{d}**	**$0.399^{tdlrsfp}_{tdlrsfp}$**
Chrome	0.58	0.113	0.118	0.039	-	0.119	0.101	0.122	0.137
OpenCV	0.16	0.481	0.802	0.195	-	0.845	0.680	0.818	0.819
Pandas	0.64	0.266	0.265	0.266	-	0.375	0.405	0.388	0.435
Tnsrflw	0.23	0.208	0.166	0.111	-	0.246	0.163	0.189	0.182
MAP_{avg} (Others)		**0.267**	**0.338**	**0.153**	-	**0.396**	**0.339**	**0.379**	**0.393**
MAP_{avg} (Overall)		**0.346^{d}_{d}**	**0.330**	**0.328**	-	**0.370^{tdlfp}_{tdlp}**	**0.347^{d}_{d}**	**0.350^{df}_{df}**	**0.398^{tdlsfp}_{tdlsfp}**
MI-wtd MAP		**0.447**	**0.424**	**0.447**	-	**0.467**	**0.442**	**0.446**	**0.500**

t: TFIDF d: DLM (FI) l: BugLocator r: BLUiR s: MRF SD f: MRF FD p: PWSM

8.2.3 Retrieval Experiments

We provide bug localization results for comparing the following eight retrieval algorithms: (1) TFIDF, (2), FI BoW, (3) BugLocator, (4) BLUiR, (5) MRF SD, (6) MRF FD, (7) PWSM, and (8) SCOR. Through our retrieval experiments we attempt to answer the following 6 important research questions:

RQ1: In terms of retrieval precision, how do the first, second, and third generation tools compare against each other?

RQ2: Does the performance of the retrieval algorithms depend on the programming language used in the software?

RQ3: Are the word embeddings provided by SCOR really generic?

RQ4: How to best create a composite retrieval performance metric for large-scale evaluations?

RQ5: Does changing the semantic word embeddings affect the performance of the semantics-based retrieval algorithms?

RQ6: Does replacing DLM with TFIDF in MRF based frameworks enhance the performance of bug localization systems?

The questions RQ1 and RQ2 are important because they represent the primary motivation for our research. As for RQ3, RQ5, and RQ6, they are included because of the current focus of research in software mining and text retrieval, which is exploiting semantics and term-term ordering for retrieval. Finally, RQ4 reflects moving from small-scale evaluations to large-scale evaluations.

The MAP performance numbers for the eight retrieval algorithms evaluated on 29 Java, C/C++, and Python projects present in Bugzbook are shown in Table 8.2. Notice that the last two rows contain the MAP values for the eight retrieval algorithms averaged across all projects, and the Mutual Information weighted (MI-wtd) MAP values for the same, respectively. Also notice that the second column contains the MI values for each project. We also show in the second to the last row for just the C/C++/Python projects and, in the 8th row from the bottom for just

the Java-based projects, the results of significance testing. The superscript denotes the significant difference when considering p-value less than 0.05, while the subscript denotes significance difference when considering p-value less than 0.01. For example, in the second to the last row of MRF SD column, we have 0.370^{tdlfp}_{tdlp} which specifies that MRF SD is significantly better than TFIDF, DLM, BugLocator, MRF FD, and PWSM when considering p-value less than 0.05, while it is significantly better than only TFIDF, DLM, BugLocator, and PWSM when considering p-value less than 0.01.

In the discussion that follows, we use this table to answer the six important research questions posed above. Regarding the empty entries in the last seven rows of the BLUiR column in Table 8.2, since this tool was designed specifically for Java source code, we do not report on its performance on non-Java projects (these being Chrome, OpenCV, Pandas, and Tensorflow). BLUiR uses a Java-specific parser to extract the method, the class, and the identifier names, and the comment blocks from Java source code files. Therefore, in all our comparison involving BLUiR, we include only the Java based projects in Bugzbook.

RQ1: In terms of retrieval precision, how do the first, second, and third generation tools compare against each other?

TFIDF and DLM are the two first generation tools whose average MAP values across all software projects in Bugzbook are 0.346 and 0.330, respectively, as shown in the table. Our results show that TFIDF outperforms DLM (or FI BoW) model by around 5%. The performance difference between TFIDF and FI is significant even when considering p-value less than 0.01. This implies that when considering pure-BoW based tools one should choose TFIDF over FI (DLM) model.

The second generation tools BugLocator and BLUiR that incorporate software-evolution history and structural information have average MAP values across all Java projects of 0.357 and 0.346, respectively. The MAP value for BugLocator on all the projects present in Bugzbook is 0.328. Both these bug localization tools perform significantly better than the FI BoW (DLM) model when only Java projects are considered in evaluation and when p-value is 0.05. However, when p-value is 0.01, only

BugLocator outperforms DLM. The performance numbers for BugLocator and DLM when all the projects in Bugzbook (including Java, C/C++, and Python projects) are considered are comparable.

We note that the simple TFIDF BoW model significantly outperforms BLUiR by 4% when examined through our large-scale bug localization study of Java projects. In a project-by-project comparison, BLUiR outperforms TFIDF in just 12 out of the 25 Java-based projects in Table 8.2 . Amongst these, the comparative results for AspectJ and Eclipse are along the same lines as those reported previously in the original BLUiR paper. However, with regard to the projects on which BLUiR was not evaluated previously, its performance on several Apache based projects is worse than that of TFIDF.

On the other hand, the performance numbers for TFIDF and BugLocator are comparable. The performance of BugLocator is significantly better than that of BLUiR for the Java only projects. This contradicts the finding presented in [40] and [12].

The third generation order-only MRF SD and MRF FD models with average MAP values across all projects of 0.370 and 0.347, significantly outperform the first generation tool DLM by 12% and 5%, respectively. This confirms the finding in [6]. However, when compared with TFIDF, while MRF SD significantly outperforms TFIDF, the performance of MRF FD is similar to that of TFIDF.

We observe that the two order-only MRF SD and MRF FD retrieval models perform equivalently when evaluated using statistical t-testing and considering p-value less than 0.01. This contradicts the finding in [6] which shows that the performance of MRF SD and MRF FD are similar in terms retrieval accuracy.

We notice that MRF SD outperforms MRF FD on 19 out of 29 projects. The projects on which MRF FD outperforms MRF SD are Ambari, AspectJ, Eclipse, Hive, JCR, Math, Pig, Tez, Tika, and Pandas. Most of these projects have large number of bug reports and contribute in total around 10000 — that is roughly around 50% — of bug reports to the Bugzbook dataset. Since both MRF SD and MRF FD outperform

each other on roughly equal number of bug reports, this is a possible reason for their statistically equivalent performance.

We compare the performance of second generation tools, BugLocator and BLUiR, with the pure-ordering based third generation tools, MRF SD and MRF FD, and observe that both MRF SD and MRF FD outperform both BugLocator and BLUiR.

The performance of MRF SD is significantly better than that of both BugLocator and BLUiR on Java based projects. MRF SD also significantly outperforms BugLocator by 13% on all the projects in Bugzbook. The performance of MRF FD is better than that of BLUiR. The performance of BugLocator is better than that of MRF FD on Java projects with a p-value of 0.05. However, the performance numbers for the two are comparable when p-value of 0.01 is considered. Their performance is also comparable when all projects in Bugzbook are considered. This result contradicts the results reported in [6].

When considering semantics-only based retrieval with the PWSM model we observe a mean MAP value of 0.350 across all the projects in the Bugzbook dataset. We notice that whereas PWSM outperforms DLM significantly by 6%, it does not do so vis-a-vis TFIDF. The performance of PWSM is comparable to that of BLUiR when only the Java projects are considered. Additionally, PWSM does not significantly outperform BugLocator when all projects in Bugzbook are considered. The percentage difference between the all-projects performance numbers for PWSM and BugLocator is around 6%.

The performance of PWSM — which is a pure-semantics based third generation tool — is comparable to the performance of pure-ordering based MRF SD model. However, PWSM significantly outperforms MRF FD model. This comparison is not performed in [16]. The performance of SCOR — which combines MRF based term-term ordering dependencies with word2vec based semantic word embeddings, outperforms the first and second generation tools. We observe that the performance of SCOR is significantly better than the other seven retrieval algorithms when considering retrieval accuracies.

RQ2: Does the performance of the retrieval algorithms depend on the programming language used in the software?

In many past studies, only Java based software projects were used for evaluating the performance of bug localization tools. This question is important as it helps in determining the performance of these bug localization tools on non-Java projects. To answer this question we compare the performance of each retrieval algorithm on projects written in Java and other programming languages.

The average MAP values for all the eight retrieval algorithms on projects that only use Java programming language are shown in the 8th row from the bottom in Table 8.2. The average MAP values for all retrieval algorithms except BLUiR on C/C++ and Python based projects are shown in the 3rd row from the bottom in Table 8.2.

We notice that the performance of all retrieval algorithms except for TFIDF and BugLocator on Java-based libraries is similar to what we get on C/C++ and Python based projects. TFIDF and BugLocator perform significantly poorly on non-Java projects. We also observe that the semantics-based retrieval algorithms perform surprisingly very well on C/C++ and Python projects. What makes the last observation all the more surprising is that the word2vec algorithm was trained on only the Java based projects used by SCOR.

With a minimum MAP value of 0.039 for BugLocator and a maximum MAP value of 0.137 for SCOR, Chrome is the project on which the performance of all the retrieval algorithms is the lowest. The top three algorithms on Chrome are SCOR, PWSM, and MRF SD with MAP values of only 0.137, 0.122, and 0.119, respectively.

The MAP values for all retrieval algorithms except for TFIDF and BugLocator on the 8 bug reports of the OpenCV project are very high. The three bug localization techniques that worked the best on the OpenCV project are MRF SD, SCOR, and PWSM with MAP values of 0.845, 0.819, and 0.818, respectively.

As for Pandas — a pure Python project — SCOR, MRF FD, and PWSM are the three algorithms that perform the best in terms of retrieval precision with MAP values of 0.435, 0.405, and 0.388.

The MAP values of the retrieval algorithms on Tensorflow are not very impressive. The lowest performing algorithm, BugLocator, achieved a MAP value of only 0.111, while the top performing algorithm, MRF SD, works with a MAP value of only 0.246. The top three algorithms for this project are MRF SD, TFIDF, and PWSM with MAP values of 0.246, 0.208, and 0.189.

RQ3: Are the word embeddings provided by SCOR really generic?

In the SCOR paper [16], we claimed that the SCOR word embeddings generated by the word2vec algorithm in that paper would be generic enough so that they could be used for carrying out semantic search in new libraries, that is, the libraries that were not used for generating the embeddings. However, in [16], this claim was supported with the results from just one library, AspectJ.

Our new results, as reported in this chapter, provide further affirmation for that claim. The Java-based dataset that was used for training the word2vec algorithm in [16] did not include the following Apache projects in the Bugzbook dataset: Bigtop, OpenNLP, PDFBox, and Drill. The retrieval results for SCOR on these four projects as shown in Table 8.2 speak for themselves.

Further affirmation of our claim is provided by the C/C++ and Python based projects in Bugzbook. We observe that the performance of SCOR on roughly 150 Chrome bug reports and roughly 180 Pandas bug reports is the best among all the retrieval algorithms. Notice that Chrome is a pure C/C++ based project while Pandas is a pure Python based project. On the 8 OpenCV bug reports and the 10 Tensorflow bug reports, however, the performance of MRF SD is better than that of SCOR.

Therefore, in answer to this question, we can say that the word embeddings generated by the word2vec algorithm in SCOR are generic enough to be used for carrying

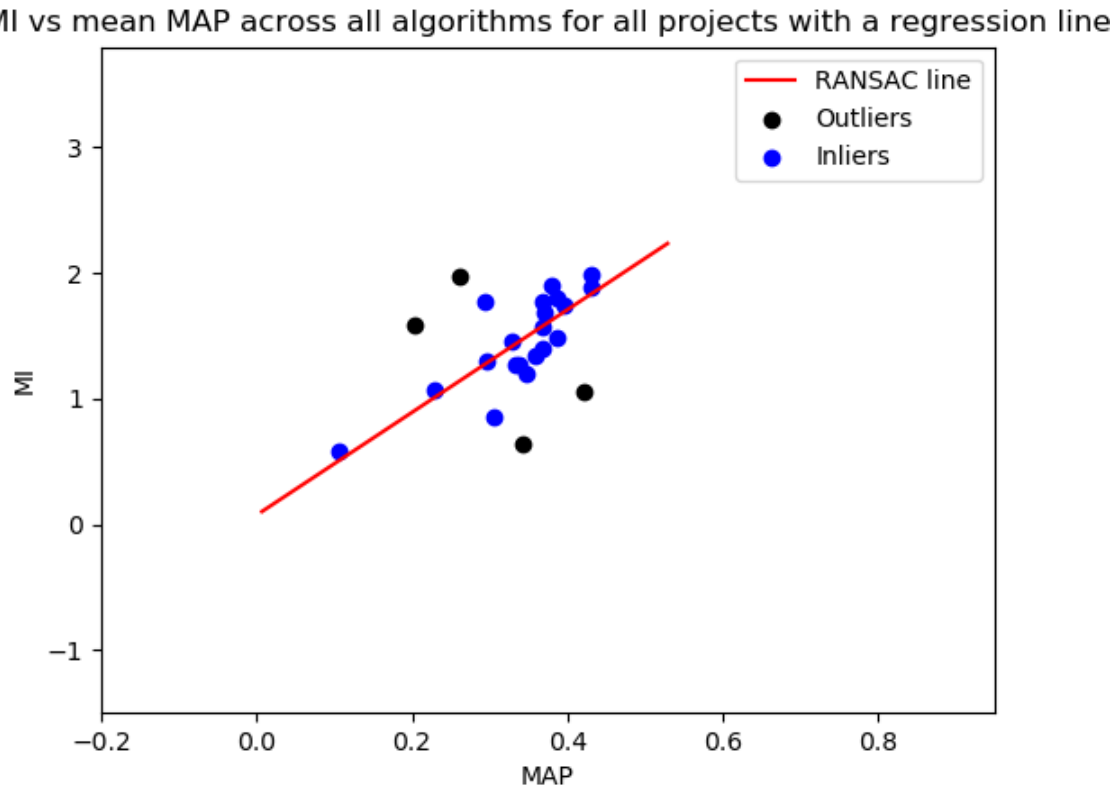


Fig. 8.1.: Scatter plot of average MAP vs MI values. Each data point in the plot is a tuple (MAP, MI) for a software project. The MAP value plotted for a software project is the mean of the 8 MAP values obtained while evaluating the 8 retrieval algorithms on a specific project. Also shown in the figure is the RANSAC fitted line along with inlier and outlier points. A low MI implies a difficult project, which in turn, implies a low mean MAP value for the retrieval algorithms.

out semantic search not only in Java based projects not seen in SCOR but also in C/C++ and Python projects.

RQ4: How to best create a composite retrieval performance metric for large-scale evaluations?

When a bug localization dataset involves multiple projects, it is unlikely that all the projects would present the same level of difficulty (LoD) to a retrieval engine. So, ideally, one should weight the performance numbers for the different projects with some measure of LoD for the individual projects.

In Chapter 3 Section 3.3.3, we show how Mutual Information (MI) between source code files and bug reports can be used as measure of LoD of performing retrieval. The higher the value of MI for a project, the more the bug reports can tell us about the project vocabulary and vice versa, and in turn the easier the retrieval is for the project.

When we plot the MI value for each project against the mean of the MAP performance numbers obtained with the different retrieval algorithms for that project, we obtain the scatter plot shown in Figure 8.1. We also show in the figure a least-squares line fitted to the data points using the RANSAC (Random Sample and Consensus) algorithm along with the inlier and the outlier points. The slope of this line is 4.085 and the intercept 0.073.

The correlation that is present between MI and MAP implies that MI captures, albeit approximately, the level of retrieval difficulty for a given software library along with its bug reports. Note that at any given value of MI, we do not distinguish between the retrieval algorithms in terms of their performance values. Rather, we take a mean MAP value across all the algorithms to represent the performance of all the retrieval algorithms on the project that corresponds to MI value.

The second column of Table 8.2 shows the calculated MI values for the software projects in Bugzbook. Also, the last row of the table shows the MI-weighted MAP values averaged across all the projects for each retrieval algorithm. We observe that when the MI value for a project is high — as for example 1.90 for Camel and 1.99 for

Table 8.3.: Shown are the MAP values obtained while changing the size of the semantic word vectors for the SCOR algorithm, evaluated on the Eclipse software project. Also shown are the results obtained by replacing word2vec with other word embedding generators.

SCOR-V500	**SCOR-V1000**	**SCOR-V1500**
0.3191	0.3204	0.3193
SCOR$_{\text{skipgram}}$	**SCOR**$_{\text{glove}}$	**SCOR**$_{\text{fasttext}}$
0.3204	0.3192	0.3182

Wicket — the MAP values of the retrieval algorithms for that project are also high. The lowest MAP value observed for Camel is 0.345 and for Wicket is 0.389. On the other hand, with a low MI value of 1.57 for Drill project, the highest MAP value in that row is only 0.240 for SCOR algorithm.

RQ5: Does changing the semantic word embeddings affect the performance of the semantics-based retrieval algorithms?

The word embeddings can be changed either by changing the sizes of the vector involved, or by using different embeddings altogether.

To address the question related to the sizes of the vectors, we varied the size of the word2vec representations and generated the retrieval results for the SCOR retrieval model. The results are presented in Table 8.3. We refer to the different versions of SCOR, with each version using vectors of a specific size, as SCOR-V500, SCOR-V1000, and SCOR-V1500. In this notation, SCOR-VN uses word embedding vectors of size N. The first row of Table 8.3 shows the retrieval results on the Eclipse dataset that contains 4000 bug reports with the different versions of SCOR. Based on these results, we conclude that the size used for the word embeddings has no significant impact on the retrieval performance. We chose Eclipse for this test as it has already been used in several previous studies related to bug localization.

We also compare the performance of SCOR when it is used with different types of word embeddings. In addition to word2vec, there are now two other well-known word embeddings: GloVe (Global Vector Representations) [19] and FastText [18]. When we replace the word2vec Skipgram model with GloVe and FastText in SCOR, the difference observed in terms of MAP performance on the 4000 bug reports of Eclipse dataset is negligible as shown in the second row of Table 8.3. In that table, $\text{SCOR}_{\text{skipgram}}$ refers to the original SCOR algorithm, and $\text{SCOR}_{\text{glove}}$ and $\text{SCOR}_{\text{fasttext}}$ refer to the versions of SCOR using GloVe and FastText word embeddings. For all the three word embedding algorithms we use the same input training dataset that is available at our SCOR website. For this study, we used embedding vectors of size 500. We conclude that the retrieval results with SCOR are not affected by either the choice of the embeddings used or the sizes of the vectors involved.

RQ6: Does replacing DLM with TFIDF in MRF based retrieval enhance the performance of bug localization systems?

Since TFIDF performs better than FI in terms of retrieval precision, and is comparable in performance to the more advanced BoW tools like BugLocator and BLUiR as we discussed in the answer to RQ1, we believe tht the question posed above is important. The comparative results presented in Table 8.4 say that the answer to this question is a definite yes. That table shows the performance of MRF SD and SCOR using TFIDF as the BoW model versus the results shown previously in Table 8.2. The notation SD-T and SCOR-T is for these algorithms when use the TFIDF score for the BoW contribution when computing the composite relevance score of a file vis-a-vis a bug report.

Table 8.4.: We compare MAP values of MRF SD with MRF SD-T, and SCOR with SCOR-T retrieval algorithms evaluated on Bugzbook dataset. Notice that SD-T and SCOR-T are the versions of MRF SD and SCOR when using TFIDF scores in computing the composite score for retrieval, respectively.

Project	**SD**	**SD-T**	**SCOR**	**SCOR-T**
Ambari	0.268	0.294	0.295	0.298
Aspectj	0.226	0.232	0.250	0.235
Bigtop	0.304	0.325	0.560	0.572
Camel	0.405	0.420	0.407	0.417
Cassandra	0.369	0.389	0.411	0.451
CXF	0.348	0.358	0.363	0.376
Drill	0.218	0.231	0.240	0.245
Eclipse	0.303	0.313	0.320	0.323
HBase	0.429	0.443	0.453	0.449
Hive	0.335	0.370	0.345	0.339
JCR	0.453	0.454	0.450	0.465
Karaf	0.374	0.393	0.427	0.424
Mahout	0.315	0.334	0.338	0.348
Math	0.545	0.519	0.512	0.481
Opennlp	0.433	0.470	0.498	0.510
PDFBox	0.368	0.381	0.380	0.394
PIG	0.295	0.353	0.335	0.396
SOLR	0.371	0.384	0.394	0.398
Spark	0.377	0.437	0.418	0.441
Sqoop	0.367	0.384	0.417	0.419
Tez	0.424	0.439	0.431	0.468
Tika	0.290	0.328	0.326	0.361
Wicket	0.450	0.458	0.440	0.440
WW	0.414	0.426	0.430	0.448
Zookeeper	0.565	0.507	0.529	0.524
Chrome	0.119	0.125	0.137	0.125
OpenCV	0.845	0.699	0.819	0.804
Pandas	0.375	0.365	0.435	0.437
Tensorflow	0.246	0.201	0.182	0.186
Average	**0.370**	**0.380**	**0.398**	**0.409**
MI-wtd	**0.467**	**0.490**	**0.500**	**0.512**

9. CONCLUSION

We advance the state-of-the-art in information retrieval (IR) based automatic bug localization in this dissertation. We developed a novel source code search tool for bug localization, called SCOR, that jointly models order and semantics for better retrieval. Our SCOR retrieval model outperforms various state-of-the-art tools in bug localization.

SCOR is the first source code search algorithm that combines both order and semantics into a single retrieval framework. We achieve this by modeling term-term ordering dependencies using Markov Random Fields (MRF) and inter-term semantic relationships using cosine similarities between the word vectors of software terms present in the code files and bug reports.

The manner in which we combine order and semantics into a single retrieval framework is quite novel, and is therefore, one of the key contributions of this dissertation. We compute a 2D numeric array of cosine similarity values between pairs of terms in the query and in the file. Afterwards, we subject the 2D array with a convolution operator designed specifically for the purpose of imposing ordering constraints in the semantic-based retrieval. The specially designed kernel used for convolution contains high values in its diagonal, and zeros elsewhere. We argue that convolution with such a kernel produces the same effect as what would be achieved with MRF based logic.

SCOR requires semantic word vectors for software terms present in the vocabulary of the corpus. To obtain these word vectors for the software terms, we experimented with various semantic word embedding algorithms, such as word2vec, FastText, and GloVe. We noticed their performances to be comparable when it comes to achieving effective code retrieval for bug localization.

The original authors of word2vec, FastText, and GloVe trained their algorithms on a corpus containing regular English language documents, such as Wikipedia and news articles. Such datasets are readily available in preprocessed form over the internet.

However, these datasets are not useful to us because our task involves bug localization in source code repositories, and the vocabulary used in the software world is quite distinct from the regular English vocabulary. Therefore, we created our own dataset of 35000 Java software repositories downloaded from GitHub. This dataset contains 35 million code files and around 1 billion software term tokens. We processed these code files using our specialized preprocessing pipeline that involves source-code identifier splitting into components, stopwords removal, and Porter stemming. We call the resulting preprocessed dataset the SCOR word embeddings dataset.

We trained the word2vec, FastText, and GloVe models on our SCOR word embeddings dataset and performed a thorough analysis of the word vectors generated by these models. We noticed that the vectors for software terms that are semantically related to each other are very close in the semantic vector space.

In addition to presenting our SCOR retrieval framework, we performed a thorough analysis of existing approaches for bug localization in the literature. We categorized the past fifteen years of research in the field into three generations starting from 2004 until 2019.

The first generation tools are the ones that are based on simple BoW based models in which frequencies of individual bug report terms are measured in source code files. The second generation tools exploited software-centric information embedded in code files and bug reports for better retrieval. However, the underlying ranking function in second generation tools were still based on simple BoW assumption. The third and the most recent generation tools go beyond the BoW assumption and are based on modeling semantic and ordering relationship between terms for better retrieval.

However, a thorough analysis of past studies revealed a disconcerting trend that the datasets used by the researchers in bug localization were all relatively small in sizes, and also, were all based mostly on Java software repositories.

Therefore, we created our novel, large, and diverse Bugzbook dataset that contains over 20,000 bug reports taken from 30 different software projects written in multiple programming languages. We then selected eight notable techniques from the three generations of bug localization research, and performed experiments on them using our Bugzbook dataset.

Our comprehensive large-scale evaluation of three generation of techniques reveal some important insights: (1) The third generation tools are more effective in performance than the previous generation tools, (2) Our SCOR retrieval framework outperforms various state-of-the-art bug localization algorithms, and (3) The word embeddings generated after training a semantic embedding model on a large corpus of 35000 Java repositories are quite generic, and therefore, can be used to perform retrieval in new software repositories not seen by the model in the training phase, even if the new repositories are written in non-Java programming languages.

We expect that the bug localization research community will benefit from the SCOR retrieval framework along with the semantic word embeddings dataset, as well as the Bugzbook bug reports dataset and the accompanying large-scale comparative study of retrieval algorithms from different generations.

REFERENCES

REFERENCES

[1] "Github website." [Online]. Available: https://github.com/

[2] "Gitlab website." [Online]. Available: https://gitlab.com/

[3] "Bitbucket website." [Online]. Available: https://bitbucket.org/

[4] "Gitbox website." [Online]. Available: https://gitbox.apache.org/

[5] 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*. ACM, 2011, pp. 43–52.

[6] B. Sisman, S. A. Akbar, and A. C. Kak, "Exploiting spatial code proximity and order for improved source code retrieval for bug localization," *Journal of Software: Evolution and Process*, vol. 29, no. 1, 2017.

[7] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 309–318.

[8] 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 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 14–24.

[9] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the use of stack traces to improve text retrieval-based bug localization," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 151–160.

[10] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 53–63.

[11] B. Sisman and A. C. Kak, "Incorporating version histories in information retrieval based bug localization," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 50–59.

[12] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 345–355.

[13] L. Moreno, G. Bavota, S. Haiduc, M. Di Penta, R. Oliveto, B. Russo, and A. Marcus, "Query-based configuration of text retrieval solutions for software engineering tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 567–578.

[14] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," *Information and Software Technology*, vol. 105, pp. 17 – 29, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584918301654

[15] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 404–415.

[16] S. A. Akbar and A. C. Kak, "Scor: Source code retrieval with semantics and order," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 1–12. [Online]. Available: https://doi.org/10.1109/MSR.2019.00012

[17] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119. [Online]. Available: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[18] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *TACL*, vol. 5, pp. 135–146, 2017.

[19] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *In EMNLP*, 2014.

[20] D. Metzler and W. B. Croft, "A markov random field model for term dependencies," in *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2005, pp. 472–479.

[21] J. Peng, C. Macdonald, B. He, V. Plachouras, and I. Ounis, "Incorporating term dependency in the dfr framework," in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2007, pp. 843–844.

[22] Y. Lv and C. Zhai, "Positional language models for information retrieval," in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2009, pp. 299–306.

[23] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *11th Working Conference on Reverse Engineering*, Nov 2004, pp. 214–223.

[24] A. Kuhn, S. Ducasse, and T. Grba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230 – 243, 2007, 12th Working Conference on Reverse Engineering. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584906001820

[25] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, ser. WCRE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 155–164. [Online]. Available: https://doi.org/10.1109/WCRE.2008.33

[26] B. D. Nichols, "Augmented bug localization using past bug information," in *Proceedings of the 48th Annual Southeast Regional Conference*, ser. ACM SE 10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1900008.1900090

[27] S. Davies, M. Roper, and M. Wood, "Using bug report similarity to enhance bug localisation," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 125–134.

[28] S. Davies and M. Roper, "Bug localisation through diverse sources of information," in *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2013, pp. 126–131.

[29] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 181–190.

[30] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 689–699. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635874

[31] K. C. Youm, J. Ahn, J. Kim, and E. Lee, "Bug localization based on code change histories and bug reports," in *2015 Asia-Pacific Software Engineering Conference (APSEC)*, Dec 2015, pp. 190–197.

[32] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 262–273.

[33] Y. Uneno, O. Mizuno, and E. Choi, "Using a distributed representation of words in localizing relevant files for bug reports," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Aug 2016, pp. 183–190.

[34] T. V. Nguyen, A. T. Nguyen, H. D. Phan, T. D. Nguyen, and T. N. Nguyen, "Combining word2vec with revised vector space model for better code retrieval," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017, pp. 183–185.

[35] M. M. Rahman and C. K. Roy, "Improving ir-based bug localization with context-aware query reformulation," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 621–632.

[36] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE transactions on software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[37] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The impact of classifier configuration and classifier combination on bug localization," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1427–1443, 2013.

[38] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 218–229.

[39] X. Huo, F. Thung, M. Li, D. Lo, and S. Shi, "Deep transfer bug localization," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[40] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. L. Traon, "Bench4bl: Reproducibility study of the performance of ir-based bug localization," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, 2018, pp. 1–12.

[41] Atlassian. Jira platform. [Online]. Available: https://www.atlassian.com/software/jira

[42] X. Ye, R. Bunescu, and C. Liu, "Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation," *IEEE Trans. Softw. Eng.*, vol. 42, no. 4, p. 379402, Apr. 2016. [Online]. Available: https://doi.org/10.1109/TSE.2015.2479232

[43] "ibugs dataset." [Online]. Available: https://www.st.cs.uni-saarland.de/ibugs/

[44] "Buglinks dataset." [Online]. Available: https://engineering.purdue.edu/RVL/Database/BUGLinks/

[45] M. Cepl. Github issues export. [Online]. Available: https://github.com/mcepl/github-issues-export

[46] Apache. Apache archives. [Online]. Available: https://archive.apache.org

[47] Atlassian. Jira issue types. [Online]. Available: https://confluence.atlassian.com/adminjiracloud/issue-types-844500742.html

[48] T. B. Le, M. Linares-Vasquez, D. Lo, and D. Poshyvanyk, "Rclinker: Automated linking of issue reports and commits leveraging rich contextual information," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 36–47.

[49] C. Zhai and J. Lafferty, "A study of smoothing methods for language models applied to ad hoc information retrieval," in *ACM SIGIR Forum*, vol. 51, no. 2. ACM, 2017, pp. 268–276.

[50] S. Robertson and K. Spärck Jones, "Simple, proven approaches to text retrieval," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-356, Dec. 1994. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-356.pdf

[51] M. Bendersky, D. Metzler, and W. B. Croft, "Learning concept importance using a weighted dependence model," in *Proceedings of the third ACM international conference on Web search and data mining*. ACM, 2010, pp. 31–40.

[52] G. Amati and C. J. Van Rijsbergen, "Probabilistic models of information retrieval based on measuring the divergence from randomness," *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 4, pp. 357–389, 2002.

[53] S. Huston and W. B. Croft, "A comparison of retrieval models using term dependencies," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. ACM, 2014, pp. 111–120.

[54] T. Tao and C. Zhai, "An exploration of proximity measures in information retrieval," in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2007, pp. 295–302.

[55] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[56] A. Singhal, G. Salton, M. Mitra, and C. Buckley, "Document length normalization," *Information Processing & Management*, vol. 32, no. 5, pp. 619–633, 1996.

[57] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008, vol. 1, no. 1.

[58] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *Journal of Machine Learning Research*, vol. 3, pp. 1137–1155, 2003. [Online]. Available: http://www.jmlr.org/papers/v3/bengio03a.html

[59] F. Morin and Y. Bengio, "Hierarchical probabilistic neural network language model," in *AISTATS05*, 2005, pp. 246–252.

[60] A. Mnih and Y. W. Teh, "A fast and simple algorithm for training neural probabilistic language models," in *In Proceedings of the International Conference on Machine Learning*, 2012.

[61] X. Rong, "word2vec parameter learning explained," *CoRR*, vol. abs/1411.2738, 2014. [Online]. Available: http://arxiv.org/abs/1411.2738

[62] A. Bakarov, "A survey of word embeddings evaluation methods," *CoRR*, vol. abs/1801.09536, 2018. [Online]. Available: http://arxiv.org/abs/1801.09536

[63] Y. Choi, C. Yi-I Chiu, and D. Sontag, "Learning low-dimensional representations of medical concepts," *AMIA Joint Summits on Translational Science proceedings. AMIA Summit on Translational Science*, vol. 2016, pp. 41–50, 07 2016.

[64] B. Mitra, E. T. Nalisnick, N. Craswell, and R. Caruana, "A dual embedding space model for document ranking," *CoRR*, vol. abs/1602.01137, 2016. [Online]. Available: http://arxiv.org/abs/1602.01137

[65] G. Zuccon, B. Koopman, P. Bruza, and L. Azzopardi, "Integrating and evaluating neural word embeddings in information retrieval," in *Proceedings of the 20th Australasian Document Computing Symposium*, ser. ADCS '15. New York, NY, USA: ACM, 2015, pp. 12:1–12:8. [Online]. Available: http://doi.acm.org/10.1145/2838931.2838936

[66] D. Ganguly, D. Roy, M. Mitra, and G. J. Jones, "Word embedding based generalized language model for information retrieval," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '15. New York, NY, USA: ACM, 2015, pp. 795–798. [Online]. Available: http://doi.acm.org/10.1145/2766462.2767780

[67] J. Guo, Y. Fan, Q. Ai, and W. B. Croft, "Semantic matching by non-linear word transportation for information retrieval," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '16. New York, NY, USA: ACM, 2016, pp. 701–710. [Online]. Available: http://doi.acm.org/10.1145/2983323.2983768

[68] T. Kenter and M. de Rijke, "Short text similarity with word embeddings," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '15. New York, NY, USA: ACM, 2015, pp. 1411–1420. [Online]. Available: http://doi.acm.org/10.1145/2806416.2806475

[69] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: A neural code search," in *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018. New York, NY, USA: ACM, 2018, pp. 31–41. [Online]. Available: http://doi.acm.org/10.1145/3211346.3211353

[70] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, "Learning deep structured semantic models for web search using clickthrough data," in *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, ser. CIKM '13. New York, NY, USA: ACM, 2013, pp. 2333–2338. [Online]. Available: http://doi.acm.org/10.1145/2505515.2505665

[71] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil, "A latent semantic model with convolutional-pooling structure for information retrieval," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, ser. CIKM '14. New York, NY, USA: ACM, 2014, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/2661829.2661935

[72] B. Hu, Z. Lu, H. Li, and Q. Chen, "Convolutional neural network architectures for matching natural language sentences," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, pp. 2042–2050. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969033.2969055

[73] Z. Lu and H. Li, "A deep architecture for matching short texts," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'13. USA: Curran Associates Inc., 2013, pp. 1367–1375. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999611.2999764

[74] L. Pang, Y. Lan, J. Guo, J. Xu, S. Wan, and X. Cheng, "Text matching as image recognition," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, pp. 2793–2799. [Online]. Available: http://dl.acm.org/citation.cfm?id=3016100.3016292

[75] B. Mitra, F. Diaz, and N. Craswell, "Learning to match using local and distributed representations of text for web search," in *WWW*, 2017.

[76] "Eclipse software." [Online]. Available: eclipse.org

[77] "Aspectj software." [Online]. Available: https://eclipse.org/aspectj

[78] X. Wang, L. Pollock, and K. Vijay-Shanker, "Developing a model of loop actions by mining loop characteristics from a large code corpus," 2015.

[79] "Gensim software." [Online]. Available: https://radimrehurek.com/gensim/

[80] A. C. Kak, "Evaluating information retrieval algorithms with significance testing based on randomization and students paired t-test." [Online]. Available: https://engineering.purdue.edu/kak/Tutorials/SignificanceTesting.pdf

[81] U. of Glasgow. Terrier open-source search engine software. [Online]. Available: http://terrier.org/

[82] D. Kim, Y. Tao, S. Kim, and A. Zeller. Buglocator software. [Online]. Available: https://code.google.com/archive/p/bugcenter/downloads

[83] R. K. Saha. Bluir software. [Online]. Available: http://riponsaha.com/BLUiR.html

[84] "Scor word embeddings." [Online]. Available: https://engineering.purdue.edu/RVL/SCOR_WordEmbeddings

[85] F. Niu, B. Recht, C. Re, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," in *Proceedings of the 24th International Conference on Neural Information Processing Systems*, ser. NIPS'11. USA: Curran Associates Inc., 2011, pp. 693–701. [Online]. Available: http://dl.acm.org/citation.cfm?id=2986459.2986537

APPENDICES

A. SOFTWARE TOOLS FOR CONSTRUCTING WORD EMBEDDINGS

In this Appendix, we discuss the software tools that we have used to generate semantic word embeddings for software terms. Note that these semantic word vectors are essential for modeling semantic relationships between terms in the source code files and the terms in the software search query. Semantic word embeddings are used in Chapter 7 to develop our software search tool called SCOR — Source Code Retrieval with Semantics and Order.

The following software implementations are discussed in this Appendix: (1) Our own implementation of word2vec tool to generate Skipgram word embeddings, (2) Gensim library to generate word embeddings using FastText technique, and (3) GloVe (Global Vectors) software tool to generate word embeddings using GloVe technique.

Note that our dataset consists of a corpus of 35 million source code files drawn from 35000 Java software repositories that are publicly available at GitHub. The stats related to this dataset are presented in Chapter 6.

A.1 A Parallel Multiprocessing Implementation of word2vec

Our implementation for word2vec can be found on the internet[1]. We implemented word2vec from scratch in Python programming language with the ability to process millions of source code files in parallel using Python multiprocessing framework. Notice that we did not use any Python machine learning software tool, such as `scikit-learn` or `PyTorch`, in the process of implementing the word2vec technique in Python. We avoid using these high-level machine learning tools in order

[1]https://github.com/sakbarpu/bme_project

to gain deeper insights about the internal workings of the word2vec algorithm while implementing it from scratch in Python.

We have, however, used Python libraries, such as `regex`, `string`, `numpy`, and `math` to perform low level operations on numbers, arrays, and strings. Our Python implementation of word2vec has the ability to process documents in parallel on multiple processes using Python's `multiprocessing` library.

The Python word2vec tool takes as input the root directory `corpuspath` where all the software repositories are located on the disk, and the output directory `output` where the output semantic word vectors generated by word2vec tool are stored.

The software implementation explained in this section concerns the Skipgram variant of word2vec model that uses negative sampling heuristic for efficient implementation. Notice that in Skipgram model the word2vec neural network is trained to predict context terms appearing in the window from the target term.

The process of generating semantic word vectors using word2vec Skipgram technique can be divided into several steps which we will discuss in this section. These steps are: (1) Preprocessing the raw dataset of source code files, (2) Learning the vocabulary from the source code corpus, and (3) Training word2vec on the software corpus using multiple processes.

In the subsections that follow we provide a detailed explanation of the above mentioned steps. The first two steps simply prepare the data for processing, while the fourth step that involves training word2vec in parallel on multiple processes is the most important part of the word2vec algorithm.

In the final two subsections we discuss the Negative Sampling (NS) approach to efficiently compute the word vectors, and multiprocessing based implementation of word2vec which is inspired from the "Asynchronous Stochastic Gradient Descent (SGD)" or "Hogwild SGD" [85].

A.1.1 Preprocessing Raw Dataset

The preprocessing follows the routine that is common to many retrieval frameworks published in the software engineering literature [5, 11]. Specifically, the source code files are first tokenized and special punctuation characters are removed, which is followed by camel-cased splitting of software terms. The remaining terms are then lower-cased, and subjected to a stopwords removal procedure. The terms survived after stopword removal are stemmed to their roots using Porter stemming procedure.

The preprocessed source code files are stored in a single large text file with textual content of each file stored on a separate line. Each line of the text file, therefore, corresponds to a source code file in the software corpus, and is referred to as a *sentence*.

The whole preprocessing pipeline is implemented in the class called `Preprocessor` in our word2vec implementation. The text file containing contents from all the source code files is fed into the word2vec algorithm for training.

A.1.2 Learning the Software Vocabulary

After preprocessing the source code files and storing them in single large text file, the next step in the word2vec implementation is to learn the vocabulary from the text file containing the content from the entire software corpus. The Python function that implements this procedure is called `learn_vocab()` in our word2vec implementation.

The vocabulary is initialized in the form of a Python dictionary `dict` data structure. The vocabulary contains an entry for each individual unique software term. The value corresponding to a software term key in the dictionary is the total number of times the software term appears in the entire corpus. In other words, the vocabulary provides us with two pieces of information: (1) the software terms present in the corpus, and (2) the frequencies with which each individual software term appears in the entire corpus.

In order to populate the vocabulary, the input text file is scanned while updating the count for each individual software term present in the file.

Notice that the input text file that contains the preprocessed textual content obtained from the entire corpus is very large in size, and therefore, may not be able to load completely into the working memory. Therefore, we implement a mechanism to read the sentences from the file on-the-fly. The class that enables reading text file directly from the disk is called `LineSentences` in our word2vec implementation.

In order to read the sentences directly from the file, without loading the file into RAM, a pointer to the file is first initialized using `open()`, and returned as an object to the calling function that instantiated the object of this class. Afterwards, the calling function can simply iterate through the sentences using a for loop. The iteration is made possible because of a `__iter__` method implemented in the `LineSentences` class.

After the vocabulary is learned it is truncated to remove the software terms that appear extremely rarely in the entire corpus and are considered as noise in the dataset. In our implementation the truncation is controlled by a parameter called `min_count` which is defined as the least number occurrences a software term should have in the corpus to be considered for training using word2vec. The word2vec does not build a semantic word vector for a term that occurs very rarely in the corpus. An example value of `min_count` could be 3.

The truncated vocabulary is stored on RAM to assist in further processing.

A.1.3 Training word2vec

The training of word2vec is divided into several components: (1) initializing model parameters, (2) dividing code files into chunks for distributed processing, (3) forward pass through the network, (4) computing loss function for training, and (5) backpropagating error for weight adjustments.

Model Initialization

We first initialize the model parameters $\mathbf{W}$ and $\mathbf{W}'$ that serve as the 2D weight matrices for input-to-projection and projection-to-output connections, respectively. The dimensionality of $\mathbf{W}$ and $\mathbf{W}'$ is $V \times N$ and $N \times V$, respectively, where V is the size of the vocabulary, and N is the desired size of the word vectors. This model initialization is implemented in the `init_model()` function of our word2vec implementation. Note that $\mathbf{W}$ is initialized as a uniform random distribution, while $\mathbf{W}'$ is initialized to be all zeros.

For fast processing, we utilize Python Numpy's `ctypeslib` module which is an advanced "Foreign Function Interface" package. And we make $\mathbf{W}$ and $\mathbf{W}'$ globally accessible from all the processes using the `multiprocessing` library's `Array()` object. Also, synchronization locks are not placed in these arrays that allows all the processes to simultaneously access these arrays for read/write purposes. The rationale behind making these arrays global is discussed in the last section when we discuss Asynchronous Hogwild SGD.

Distributed Processing

We then divide the training corpus equally into p chunks, where p is the number of worker processes that are executing in parallel on the multicore machine. Each worker process is responsible for processing its dataset chunk of code files and train the word2vec model using that chunk. Obviously, the last worker may get less work depending on how the dataset is distributed. We compute the `start` and `end` sentence indexes in the corpus for each worker and provide the sentences with indexes within the range `start` to `end` to worker for processing.

We make use of the `Pool` class of the `multiprocessing` library for distributed training with multiple processes. The `Pool` object maps a function `callp` across all workers for parallel processing. This `callp` function implements the core functionality of word2vec algorithm.

The Forward Pass

The training pairs of software terms are extracted from each sentence and subjected to word2vec based processing. The terms in each training pair are the input target term w_I and the output context term w_{j*}. Here, I is the index of input target term and $j*$ is the index of output context term in the vocabulary.

The neural network architecture for word2vec Skipgram model is shown in Figure A.1. At the input we provide the target term around which a window is placed, while the goal is to predict the context term appearing in the window around the target term.

In the forward pass, the one-hot encoding $\mathbf{x}$ is provided at the input layer of word2vec. All the values of one-hot vector$\mathbf{x}$ are zero except for the value at index I that corresponds to the input target term, which is 1. The output of projection layer $\mathbf{h}$ is computed as:

$$\mathbf{h} = \mathbf{W}^T\mathbf{x} = \mathbf{v}^T_{w_I} \tag{A.1}$$

Notice that the output $\mathbf{h}$ is actually the Ith row of the matrix $\mathbf{W}$. Therefore, in the actual implementation there is no notion of input one-hot encoding, rather we obtain the index of the target term in the window and directly access the Ith row of the weight matrix W using Python indexing `W[index]` to compute $\mathbf{h}$. Notice also, that the output of projection layer is not passed through a non-linear activation function, such as softmax.

Proceeding further in the forward pass, the output of projection layer $\mathbf{h}$ is subjected to another weight matrix $\mathbf{W}'$ to produce a score u_j for each word in the vocabulary.

$$u_j = \mathbf{v'}^T_{w_j}\mathbf{h} \tag{A.2}$$

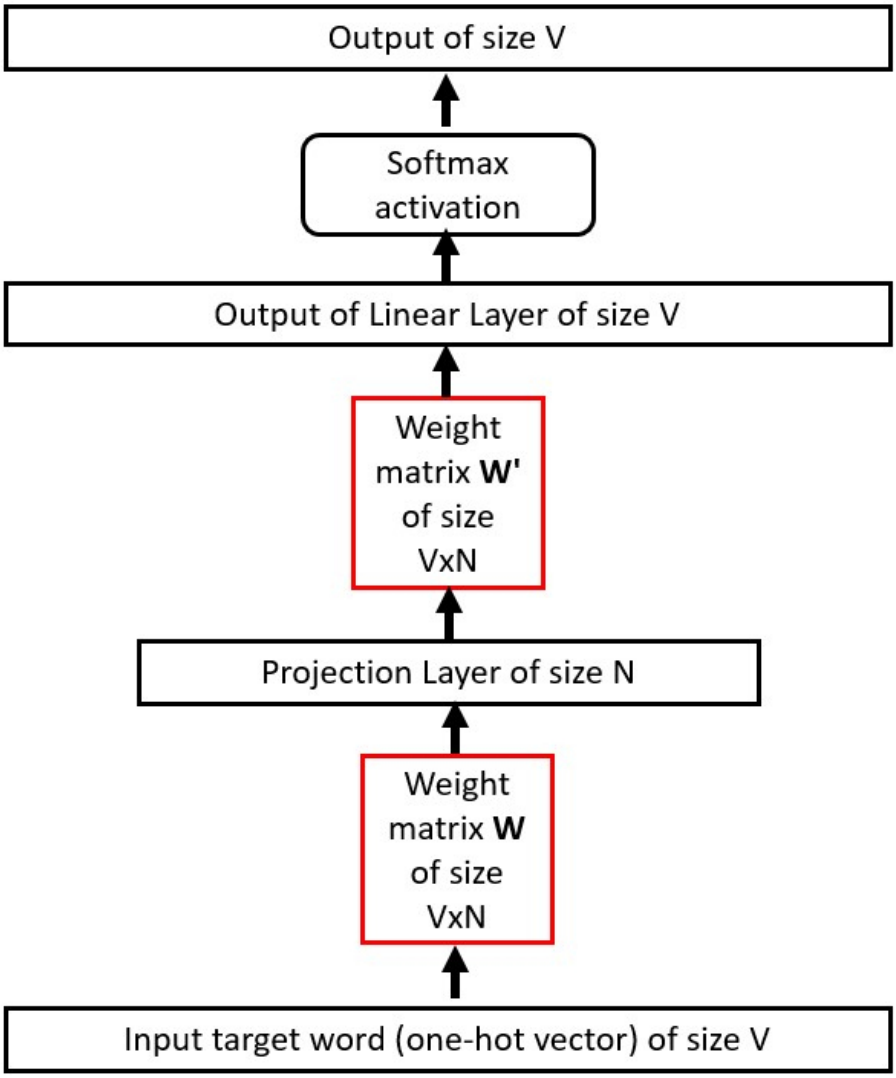


Fig. A.1.: The word2vec neural network architecture

where $\mathbf{v'}_{w_j}$ is the jth column of the matrix $\mathbf{W}'$. Therefore, we can access the vector $\mathbf{v'}_{w_j}$ simply using Python indexing `Z[index]`[2].

At the end of the forward pass, we compute softmax probability estimate for each word j in the vocabulary belonging to the context of the target term I:

$$P(w_j|w_I) = y_j = \frac{\exp(u_j)}{\sum_{j'=1}^{V} \exp(u_{j'})} = \frac{\exp(\mathbf{v'}_{w_j}^T \mathbf{v}_{w_I})}{\sum_{j'=1}^{V} \exp(\mathbf{v'}_{w_{j'}}^T \mathbf{v}_{w_I})} \tag{A.3}$$

where y_j is the output probability value for the jth term in the vocabulary. We refer to $\mathbf{v}_{w_i}$ as the input vector and $\mathbf{v'}_{w_i}$ as the output vector of the term w_i in the vocabulary. The forward pass can computed in word2vec implementation as `numpy.dot(W[index1], Z[index2])`.

Loss Function

We optimize for the conditional probability $P(w_j|w_I)$, meaning that we use the Negative Log Likelihood loss function for training word2vec. Therefore, the optimization problem solved by word2vec is given as:

$$E = -\log P(w_j|w_I) = -\log \frac{\exp(\mathbf{v'}_{w_j}^T \mathbf{v}_{w_I})}{\sum_{j'=1}^{V} \exp(\mathbf{v'}_{w_{j'}}^T \mathbf{v}_{w_I})} \tag{A.4}$$

$$= -\mathbf{v'}_{w_{j*}}^T \mathbf{v}_{w_I} + \log \sum_{j'=1}^{V} \exp(\mathbf{v'}_{w_{j'}}^T \mathbf{v}_{w_I}) \tag{A.5}$$

The energy E is minimized by maximizing the logarithmic conditional probability $\log P(w_j|w_I)$, where w_j is the output context word and w_I is the input target word. We use backpropagation with stochastic gradient descent algorithm to minimize the loss function while adjusting the weight matrices.

[2]In our word2vec implementation `Z` denotes $\mathbf{W'}$

Backpropagation

The goal in backpropagation is to adjust the weights $\mathbf{W}$ and $\mathbf{W}'$ of the word2vec neural network so as to minimize the loss function E. The manner in which we achieve that is by taking derivative of the loss function with respect to the weights in the network using Chain rule for differentiation.

Computation of $\partial E/\partial u_j$

We take the derivative of the negative log likelihood E with respect to the output of the linear layer u_j just before applying softmax activation.

Notice that the output of the network given in Equation A.3 is a vector function, i.e., a vector is given as input and a vector is produced at the output. In order to compute $P(w_j|w_I)$ or y_j for a specific term w_j in the vocabulary, we need u_j for all the terms at the output.

We first compute how loss, i.e. $E_j = -\log(y_j)$ changes with respect to y_j. Since it's a simple scalar function, the derivative calculation is quite straightforward:

$$\frac{dE_j}{dy_j} = \frac{d}{dy_j}(-log(y_j)) = -\frac{1}{y_j} \tag{A.6}$$

Next we compute the derivative of E_i with respect to u_j. Note that u_j is the output before softmax and we are deliberately using two indices, i and j, to measure the effect of changing score u_j for jth term on the loss for ith term. The loss E_i at node i is not affected just by the score u_i, rather by all the scores u_j for all the nodes j because of the software computation in between u_i and y_j.

Therefore, we compute the partial derivative, as opposed to the total derivative, of the loss E_i at output node i with respect to the score u_j at node j.

We can use the Chain rule to compute this partial derivative as follows:

$$\frac{\partial E_i}{\partial u_j} = \frac{\partial E_i}{\partial y_i} * \frac{\partial y_i}{\partial u_j} \tag{A.7}$$

For the first expression of the Chain rule, we can simply use the scalar derivative already computed in Equation A.6.

The derivative in the second part $\partial y_i/\partial u_j$ requires more work since the function y_i that we want to differentiate over is a fraction, i.e., $y_i = \exp(u_i)/\sum_j \exp(u_j)$. Taking the derivative of y_i with respect to u_j:

$$\frac{\partial y_i}{\partial u_j} = \frac{\partial}{\partial u_j} \frac{\exp(u_i)}{\sum_j \exp(u_j)} \tag{A.8}$$

At this point, we have two choices: (1) $i = j$, or (2) $i \neq j$. Therefore, after performing differentiation, we obtain:

$$\frac{\partial y_i}{\partial u_j} = \begin{cases} y_i * (1 - y_i) & , i = j \\ -y_i * y_j & , i \neq j \end{cases}$$

Plugging the appropriate values in Equation A.8:

$$\frac{\partial E_i}{\partial u_j} = y_i - t_{ij} \tag{A.9}$$

where t_{ij} is 1 when $i = j$ and t_{ij} is 0 when $i \neq j$. We know that total loss $E = \sum_i E_i$. Therefore,

$$\frac{\partial E}{\partial u_j} = y_j - t_j = e_j \tag{A.10}$$

where t_j is 1 when j is the actual output term and 0, otherwise.

Note that the above computation implies that for each term in the vocabulary we have to compare its output probability y_j with the expected output t_j which can take on a value of either 0 or 1. In our word2vec implementation, this is achieved by simply taking the difference between `label` and `p`, where `label` refers to the ground truth label t_j and `p` the probability measure y_j.

Obviously, we would never have to adjust the output u_j. Rather, we will use the derivatives $\partial E/\partial u_j$ in backward pass to adjust the weights.

Computation of $\partial E/\partial \mathbf{w'}_{ij}$

We now take the derivative of E with respect to the weights in the matrix **W'**.

$$\frac{\partial E}{\partial \mathbf{w'}_{ij}} = \frac{\partial E}{\partial u_j} * \frac{\partial u_j}{\partial \mathbf{w'}_{ij}} = e_j * h_i = e_j * \mathbf{v}_{w_I i} \tag{A.11}$$

where e_j is the error at output node j, h_i is the output of the ith unit of projection layer, and $\mathbf{v}_{w_{Ii}}$ is the ith unit in the input vector of the target term w_I.

We use the partial derivatives to adjust the weights, and the weight update equation is given as follows:

$$\mathbf{w'}_{ij}^{(new)} = \mathbf{w'}_{ij}^{(old)} - \alpha * e_j * \mathbf{v}_{w_{Ii}} \tag{A.12}$$

In terms of output word vector, the update can be written as:

$$\mathbf{v'}_{w_j}^{(new)} = \mathbf{v'}_{w_j}^{(old)} - \alpha * e_j * \mathbf{v}_{w_I} \tag{A.13}$$

where α is the learning rate. Notice that if y_j and t_j are very close to each other, then there is very little update performed on $\mathbf{v'}_{w_j}$. If $t_j = 0$ but y_j is very high, then we subtract a portion of $\mathbf{v}_{w_I}$ from $\mathbf{v'}_{w_j}$, thereby, pushing $\mathbf{v'}_{w_j}$ away from $\mathbf{v}_{w_I}$. And if $t_j = 1$ but y_j is very small, we add a portion of $\mathbf{v}_{w_I}$ to $\mathbf{v'}_{w_j}$, thereby, making $\mathbf{v'}_{w_j}$ closer to $\mathbf{v}_{w_I}$.

Computation of $\partial E/\partial \mathbf{w}_{ki}$

The goal now is to compute the derivative of E with respect to the input weight matrix **W**. We know from Chain rule that:

$$\frac{\partial E}{\mathbf{w}_{ki}} = \frac{\partial E}{\partial h_i} * \frac{\partial h_i}{\partial \mathbf{w}_{ki}} \tag{A.14}$$

We first backpropagate the error to compute the partial derivative of E with respect to the output of the projection layer h_i or $\mathbf{v}_{w_{Ii}}$. Here, h_i is the output at the ith unit of projection layer and is equivalent to the ith unit of the input vector for the target term $\mathbf{v}_{w_{Ii}}$.

$$\frac{\partial E}{\partial h_i} = \frac{\partial E}{\partial \mathbf{v}_{w_{Ii}}} = \sum_{j=1}^{V} \frac{\partial E}{\partial u_j} * \frac{\partial u_j}{\partial h_i} = \sum_{j=1}^{V} e_j * \mathbf{w'}_{ij} = \nu_i \tag{A.15}$$

where ν is an N-dimensional vector and is the sum of the output vectors $\mathbf{v'}_{w_j}$ of all the terms in the vocabulary weighted by their prediction error e_j.

Next we compute the partial derivative of h_i with respect to $\mathbf{w}_{ki}$. Recall that:

$$h_i = \sum_{k=1}^{V} x_k * \mathbf{w}_{ki} \tag{A.16}$$

Therefore,

$$\frac{\partial h_i}{\partial \mathbf{w}_{ki}} = x_k \tag{A.17}$$

Plugging the appropriate values in Equation A.14,

$$\frac{\partial E}{\partial \mathbf{w}_{ki}} = \nu_i * x_k \tag{A.18}$$

Since, only one component of input x is non-zero, we only need to compute the partial derivative of only one row of $\mathbf{W}$ that corresponds to the input term w_I. This row is also the input vector $\mathbf{v}_{w_I}$ for term w_I. The derivatives for rest of the rows of $\mathbf{W}$ are all zeros.

Therefore, the weight update for the input vector becomes:

$$\mathbf{v}_{w_I}{}^{(new)} = \mathbf{v}_{w_I}{}^{(old)} - \alpha\nu \tag{A.19}$$

We can interpret the above weight update as adding a portion of every output vector in the vocabulary to the input vector of the target term. If $t_j = 0$ and the probability y_j of an output context term w_j is high, then the input vector for w_I will move away from the output vector for term w_j. On the other hand, if $t_j = 1$ and y_j is very small, then the input vector for w_I will move closer to the output vector for w_j. There is very little effect on input vector if the probability measurements y_j are very accurate.

It is also important to note that the prediction error e_j determines how big the update needs to be in order to make the predictions accurate. As we go through the updates, we can imagine the output vector for a word w_j being dragged back-and-forth by the input vector of word w_I that appeared in the same window of w_j. Similarly, an input vector can also be imagined as being dragged back-and-forth by many output vectors.

A.1.4 Negative Sampling

The word2vec processing to generate word vectors as described earlier is very computationally expensive because for each training target-context pair, we have to perform updates to the output vectors of all the terms in the vocabulary. The reason to perform these updates is because we want to distinguish the actual context word from all the other words in the vocabulary. This is a major bottleneck in the computation process.

In negative sampling approach, we only update the output vectors for a small sample terms in the vocabulary. That sample should obviously include the actual output (ground truth) word because it is our positive sample. Additionally, few randomly sampled words are included as negative samples.

To construct a probability distribution for sampling words from the vocabulary, we build a unigram table from the count of words in the vocabulary.

Mikolov et al. [17] provide a heuristic for this purpose that works well for generating word vectors. The function we have used to implement the construction of unigram table using their heuristic is called `build_unigram_table()`. We first compute the normalization factor $\xi = \sum_w U(w)^{0.75}$, where $U(w)$ represents the count of the term w in the corpus.

Afterwards, we compute the unigram distribution as $P(w) = U(w)^{0.75}/\xi$, and store the values in the memory.

The training objective of word2vec with Negative Sampling is different from the one we saw earlier in A.4, and is given as:

$$E = -\log \sigma(\mathbf{v'}_{w_{j*}}{}^{T}\mathbf{h}) - \sum_{w_j \in \Lambda} \log \sigma(-\mathbf{v'}_{w_j}{}^{T}\mathbf{h}) \tag{A.20}$$

where $\sigma(.)$ represents the sigmoid activation function, w_{j*} is the true output word (i.e., positive sample) and $\mathbf{v'}_{w_{j*}}$ its corresponding output vector, $\mathbf{h}$ is the output of projection layer, and Λ is the set of randomly sampled words that form the negative samples for training.

We again denote the score $u_j = \mathbf{v'}_{w_j}\mathbf{v}_{w_I}$ as the output of the linear layer. The derivative of E with respect to u_j is given as:

$$\frac{\partial E}{\partial u_j} = \begin{cases} \sigma(u_j) - 1 & \text{, j is the positive word index} \\ \sigma(u_j) & \text{, j is the negative word index} \end{cases}$$

Therefore,

$$\frac{\partial E}{\partial u_j} = \sigma(u_j) - t_j = y_j - t_j \tag{A.21}$$

Next, we take the derivative of E with respect to output vector $\mathbf{v'}_{w_j}$.

$$\frac{\partial E}{\partial \mathbf{v'}_{w_j}} = \frac{\partial E}{\partial u_j} * \frac{\partial u_j}{\partial \mathbf{v'}_{w_j}} = (\sigma(u_j) - t_j)\,\mathbf{v}_{w_I} = (y_j - t_j)\,\mathbf{v}_{w_I} = e_j \mathbf{v}_{w_I}$$

Note that e_j the output prediction error and is computed as the difference between the probability estimate y_j and the true label t_j. However, obviously j does not iterate over all the terms V in the vocabulary, rather over all the terms in Λ sample.

The output vector of the terms present in the sample are updated as:

$$\mathbf{v'}_{w_j}{}^{(new)} = \mathbf{v'}_{w_j}{}^{(old)} - \alpha\,(\sigma(u_j) - t_j)\,\mathbf{v}_{w_I} \tag{A.22}$$

This is implemented in our word2vec Skipgram implementation with Negative Sampling as:

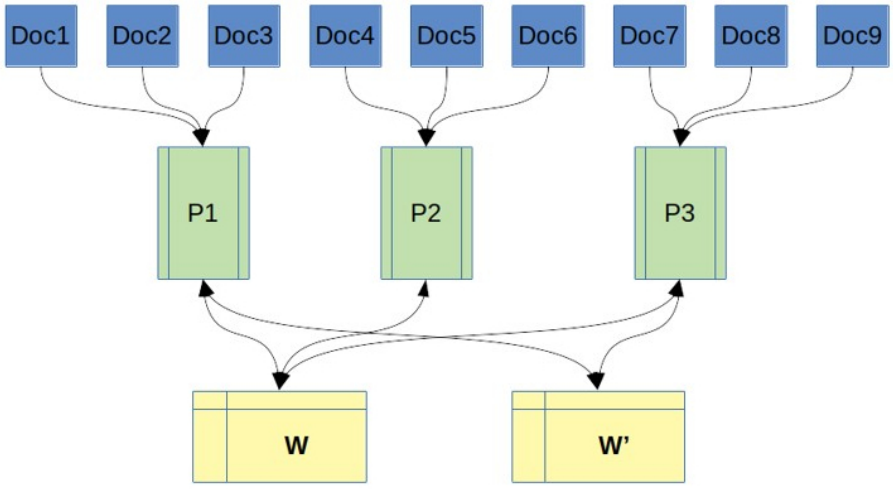


Fig. A.2.: Multiprocessing implementation of word2vec

```
f = numpy.dot(W[target], Z[context]), get net score
p = sigmoid(f), get probability estimate a scalar value
g = local_alpha * (label - p), get derivative/gradient
Z[context] += g * W[target], update output vector
```

We propagate the error backward to compute the derivative of E with respect to the input vector $\mathbf{v}_{w_I}$. But before that we compute the derivative of E with respect to $\mathbf{h}$:

$$\frac{\partial E}{\partial \mathbf{h}} = \sum_{j \in \Lambda \bigcup \{w_{j*}\}} \frac{\partial E}{\partial u_j} * \frac{\partial u_j}{\partial \mathbf{h}} = \sum_{j \in \Lambda \bigcup \{w_{j*}\}} (\sigma(u_j) - t_j)\, \mathbf{v'}_{w_j} = \nu \tag{A.23}$$

We use this value of ν in A.19 to update the input vector. The following implements the weight update portion for input vectors in the implementation.

```
neu += g * Z[context]
W[target] += neu
```

A.1.5 Hogwild SGD Implementation Using `multiprocessing` Library

The word2vec algorithm scans through millions of source code files with a window of predefined size and trains to predict context words from target words using a neural network. The training of neural network requires adjusting weight matrices $\mathbf{W}$ and $\mathbf{W}'$. The weight adjustment procedure is performed using stochastic gradient descent in which weights are adjusted for each training pair of context term and target term, as described earlier.

However, if word2vec algorithm is implemented to perform training on the files in a serial fashion using a single computational process, it would take prohibitively long time to process millions of files. Therefore, the input source code files or sentences are randomly distributed across multiple processes and each process is responsible to perform training using only its respective files. This is illustrated in Figure A.2.

A natural question that arises from the above discussion is that, if we allow the files to be distributed across processes in a multiprocessing framework, and each process is responsible to train on a specific chunk of source code files, then does each process keep a separate copy of weight matrices to train on the files?

The answer to the question posed above is, no. The matrices $\mathbf{W}$ and $\mathbf{W}'$ are, in fact, *global* to all the processes? By global we simply mean that there is only one copy of each of these matrices and that copy is shared across all processes. Obviously, a natural assumption then would be that in parallel multiprocessing implementation of word2vec these global matrices will be synchronized using a certain *locking* mechanism that would restrict multiple processes from accessing — reading from and writing into — these global matrices in order to avoid *race conditions*.

However, if the global weight matrices are implemented using synchronization locks then effectively the implementation would become a serial implementation since at each time step a certain process will lock the global matrices to read from and write into them.

While the global matrices should be locked in a usual setting, in our word2vec implementation these matrices are not synchronized using locks. Instead, inspired from the original implementation of word2vec [3], the processes are allowed to simultaneously read from and make updates to the matrices. This *loose* implementation of SGD is called Asynchronous or Hogwild SGD, and works under the assumption that at each weight adjustment only a few parameters — only those vectors corresponding to the context and target words in the window — out of all the parameters in the weight matrices are updated by a certain computational process in the neural network training. Since the files are randomly distributed across processes and there are millions of files in the dataset, the probability of collision — defined as the probability that two processes will update the same weight vectors in the weight matrices corresponding to the context and target words that the processes have encountered in the window — is very low.

A.2 Gensim Software Library

Gensim[4] is a popular software library that contains implementation of various semantic word embedding algorithms including word2vec, and FastText.

It has the functionality built-in to read the sentences from a large text file without having to load it entirely into the memory using `LineSentences`:

```
sentences = gensim.models.word2vec.LineSentence('path/to/file')
```

where 'path/to/file' is the location of the file containing source code files as sentences. Afterwards the call to execute the model such as FastText is very straightforwards:

```
gensim.models.fasttext.FastText(sentences=sentences, min_count=3,
size=200,sg=1, iter=20, window=8, workers=12)
```

where `min_count` is the minimum number of occurrences of a term in the corpus to be considered in training using the algorithm, `size` is the desired dimensionality

[3]https://code.google.com/archive/p/word2vec/
[4]https://radimrehurek.com/gensim/

of word vectors, `sg` denotes whether we want to train using Skipgram model or not, `window` is the size of the window used to scan and obtain target-context pairs of terms, and `workers` is the number of parallel workers used for training.

A.3 GloVe Software Tool

We used the GloVe[5] implementation that is made publicly available on the internet by the original authors. The input to the system is the same single large file that we have used for our word2vec and Gensim software.

We first run the `vocab_count` tool to construct unigram counts from the corpus, and discard the terms that have very low frequency in the corpus. The output of running this program is a vocabulary file containing unique terms in the corpus along with their respective counts in the corpus.

Next we use the `cooccur` tool to construct the term-term cooccurrence statistics from the corpus. The input to this tools is the corpus file as well as the vocabulary file created in the previous step. The output of this program is a binary file containing cooccurrence statistics.

We then use the `shuffle` program to randomly shuffle the binary file of cooccurrence statistics. And finally, we execute the `glove` program to train the GloVe model.

[5]https://nlp.stanford.edu/projects/glove/

VITA

VITA

Shayan Akbar is a PhD candidate in the ECE department at Purdue University in West Lafayette, IN, USA. His research interest lies in mining software repositories, data mining, machine learning, information retrieval, and computer vision. Shayan received his Bachelors in Electrical Engineering in 2012 from NED University in Karachi, Pakistan, and MS in Computer Engineering in 2015 from Purdue University. Shayan's industrial experience involves working at Intel on performance analysis of multicore architectures.