Empirical Evaluation of The Use of DV and LDA for Feature Location on Java and Python Programs

**Hongxin Chen**

**Matriculation Number: 16250132**

Dissertation 2018

Erasmus Mundus MSc in Dependable Software Systems

Department of Computer Science

Maynooth Univerisy

A dissertation submitted in partial fulfilment

of the requirements for the

Erasmus Mundus MSc Dependable Software Systems

Supervisor: Dr. James Power

19th June 2018

ii

I, Hongxin Chen, declare that this dissertation titled, “Empirical Evaluation of The Use of DV and LDA for Feature Location on Java and Python Programs” and the work presented in it are my own. I confirm that:

* This work was done wholly or mainly while in candidature for a research degree at this University.
* Where any part of this dissertation has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
* Where I have consulted the published work of others, this is always clearly attributed.
* Where I have quoted from the work of others, the source is always given. With the exceptions of such quotations, this dissertation is entirely my own work.
* I have acknowledged all main sources of help.
* Where the dissertation is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

iii

Abstract

Feature location is the activity of locating artifacts in the source code that related to the implementation of specific software functionalities. Typically, the maintenance and development of dependable software systems will include eliminating undesired functionalities and fixing bugs, as well as adding new features and enhancing existing functionalities. Identifying location of where the concreate features are implemented in the source code is one of the most frequent, general and essential activities undertaken by software engineers. This makes feature location techniques (FLTs) significant to maintenance of dependable software, and thus beneficial to practitioners and researchers. In this thesis, a new methodology of change set generation for source code files through mining git commit logs is introduced. Using the generated change set, we evaluate the effectiveness of Document Vector (DV) and Latent Dirichlet Allocation (LDA) techniques for feature location at the level of file documents. DV, an unsupervised framework based on deep learning, appears to be suited for source code as it retains not only the order but also the semantics of words in source code. LDA, a generative probabilistic model, is capable of extracting a set of topics from the code corpora, which could be regarded as representation of features. Results of experiments over 50 Java projects and 50 Python projects collected from github show that LDA outperforms DV in terms of file level feature location.

iv

Acknowledgements

Firstly, I would like to thank my supervisor Dr. James Power of the Department of Computer Science at Maynooth University for his dedication and patient guidance. He is always responsive, patient and never complains about my puzzles even they are sometimes simple and silly ones. His generous and selfish share on the domain knowledge as well as constructive advices helped me a lot specially when I was stuck.

Then I would like to thank my parents who are far away in China. They provide the whole tuition fee and living expenses during my undergraduate and the master, which makes me capable of focusing on my study without being disturbed by financial issues. They always encourage me to chase my dream and cheer me up whenever I feel depressed and frustrated. I would be not able to complete my study and dissertation without their supports.

Finally, I sincerely appreciate all the brilliant professors, lecturers and other staffs in the Maynooth University who make the school such an amazing place for me.

5

Table of Contents

1 Introduction 10

2 Background and Related Work 11

2.1 Feature Location Techniques 11

2.2 Latent Dirichlet Allocation 12

2.3 Document Vector 13

2.4.1 A Dataset from Change History 15

2.4.2 The Document Vector for Method Level Feature Location 17

3 Problem and Solution 19

3.1 Research Questions 19

3.1.1 Automated Goldset Generation 19

3.1.2 Experiments on Wider Range 19

3.1.3 Impact of Model Parameters 19

3.1.4 Our Work versus Corley et al’ Work 20

3.2 Experiment Design 20

3.2.1 Methodology Overview 20

3.2.2 System Design 21

4 Experiment Implementation 23

4.1 Related Tools & Libraries 23

4.1.1 Git & Github & GitPython 23

4.1.2 Gensim 23

4.2 Goldset Generation 24

4.2.1 The Subject Software Systems 24

4.2.2 Glossary of Artifacts 25

4.2.3 Goldset Generation Workflow 25

4.3 Feature Location 26

4.3.1 Corpora Generation 26

4.3.2 Model Generation 26

4.3.3 Rank Generation 27

Reference 28

Appendix 30

Figure 1 Graphical Model Representation of LDA 12

Figure 2 Example of Learning Paragraph Vector 14

Figure 3 Architecture of The System 22

*Table 1 Dataset from Change History* 16

Table 2 Our Work vs Corely et al.'s Work 20

# 1 Introduction

In software systems, a feature stands for a functionality that is described and defined by requirements from system users and stakeholders such as software developers. At present there is no one ever gives a formal definition for feature location, concept location or the difference between them. In general, feature location could be as an activity of locating corresponding implementation of functionalities in the source code of software system (Rajlich & Wilde, 2002). Software maintenance is one of the most general but necessary activities performed by software engineers in building system which has strict requirements on dependability. In general, behaviors during software maintenance normally consist of adding new functionalities, upgrade of existing features, bugs fixing and removing undesired functionalities. However, software developers are not able to start maintenance tasks without first pinpointing the relevant code.

For instance, Bob is a developer who has just taken over a software project and the first task given out by his leader is to fix a bug that has been recently reported. Due to unfamiliarity with the code base, he does not know where and how to begin especially. It is even worse if only few documentations on the system are provided and the help of the predecessors are not available. The only option that is left to Bob seems to be manual searching around the code base, which could be trivial, frustrating and time-consuming even with the help of conventional tools like integrated development environments (IDE). Unfortunately, this scenario is quite common among practitioners in real world. This makes feature location techniques significant to programmers like Bob in the context of developing and maintaining dependable software project.

# 2 Background and Related Work

In this chapter we introduce the background and previous work with respect to feature location, covering the domain knowledge used in this thesis. We start by introducing the classes of feature location techniques and give a brief introduction to the textual approach, as well as highlightting the significance of FLTs in section 2.1. Section 2.2 gives an overview of the definition and kernel workflow of the LDA. In section 2.3 the principle of Document Vector is introduced. Finally, Previous work that is directly related to this thesis is presented.

## 2.1 Feature Location Techniques

Feature location is an extensive area involving various research fields and is an indispensable part of software maintenance (Dit, Revelle, Gethers, & Poshyvanyk, 2013). In general, the goal of a typical feature location task is generating ranked list of program elements at different level of granularities (e.g., files, methods or classes) given a developer query as input, which could provide sufficient information to developers for tracking location of the implementation of functionalities in source code. Different FLTs vary in input requirements, the specific algorithm of locating features and how they present the results. The dimensions of a FLT concsist of types of analysis (dynamic analysis, static analysis, textural analysis, etc.), user input (natural language query, execution scenario, source code artifacts), data sources (compliable element, non-executable element, execution trace, historical information, etc.), output (file/class, method/function and statements), programming language support and evaluation methods (Dit, Revelle, Gethers, & Poshyvanyk, 2013).

One of the mainstream solutions for feature location is the text-based approaches. Using information such as source code or natural language queries as input, textual methodologies for feature location analyze program elements and their properties reflected in the code corpus. The belief is based on the hypothesis that the domain knowledge could be encoded and the implementation of a feature in a software system may be relevant to the associated requirement. Information Retrieval (IR) techniques are one of the major directions among textual approaches to feature location. Instances of IR include Latent Dirichlet Allocation and other associated statistical learning methods that used to find the corresponding codes by learning and recognizing program elements, such as comments and identifiers, that have similarities to a query offered by a user. IR techniques are significant in all phases of the software life cycle especially the area of Software Development, Maintenance and Evolution (Binkley & Lawrie, 2010a, 2010b). Whatever the classes of textual analysis used, the effectiveness of the feature location technique is heavily relied on the quality of the source code style routine as well as the user-given query (Dit, Revelle, Gethers, & Poshyvanyk, 2013).

The dependability and reliability of a software system is essential for the Software Service Providers especially whose primary product are safety-critical software products (e.g. bank system and e-payment system). As the result of fierce business competition and commercial pressure, maintenance engineers are required to fix bugs quickly and efficiently, leading to the importance of FLTs in building dependable software systems. FLTs can assist software engineers in the challenges such as software comprehension that the practitioners are confronting in software engineering during maintenance and development, those are the troubles that software engineers inevitably need to cope with after the initial release of a software system.

## 2.2 Latent Dirichlet Allocation

LDA (Blei, Ng & Jordan, 2003) is a generative probabilistic model, a typical class of bag-of-words model, as well as an unsupervised learning approach designed for processing discrete data such as text corpora. Motivated by the problem of modelling text corpora, LDA extracts a set of topics associated with ­ the artifacts and then gauges the distribution of finite topics over the corpus. It aims at finding shorter descriptions of the entries of a large collection while representing statistical relation with respect to basic learning tasks (e.g., text classification, similarity and relevance judgements, novelty detection, etc.). This process can be considered as dimensionality reduction on traits with good interpretation for the data in the matter of probabilistic semantics.

The core concept of LDA is that documents can be represented as the composition of random latent topics K and each topic could in turn expressed as distributions over words. This is quite easy to understand, because creating articles always begins with setting up a few topics of the article.

The procedure of training of LDA is separated into two phases. The first phase is finding out appropriate K topics and the second phase is determining the distribution of these topics on each document. The algorithm of LDA is interpreted as followings.

Assuming there are two matrices A and B. A is a m × k matrix where the m means the number of documents and k represents the distribution of topics on single document. The ith document of A means a topic vector that could represent the word. B is a k × v matrix where k and v represent the number of topics and frequency of words related to the topic respectively. The Bi represents a vector on vocabulary vectors (the total number of words is v) The process of determining matrix A and matrix B is equivalent to generation of a LDA model.

Using a three-level hierarchical Bayesian model, LDA models every single entity of a document collection into a combination of various topics where each document is characterized by a distribution of topics over unstructured document. The Figure 1 shows the three levels of LDA structure in a graphical representation. The boxes are plates representing replicates, which are repeated entities. The outer plate represents documents and the inner plate means the repeated choice of words positions in a given document.

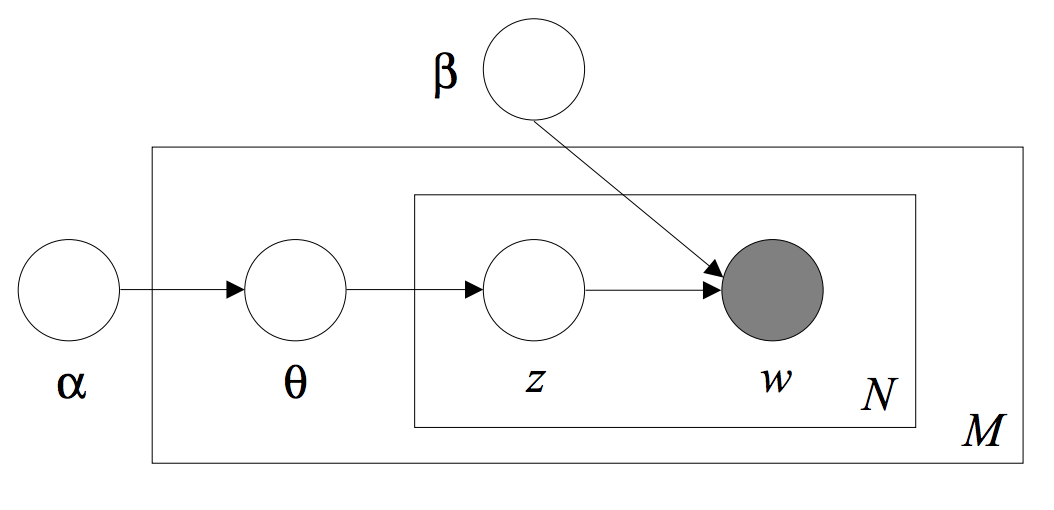


Figure 1 Graphical Model Representation of LDA

The parameters α and β are corpus level parameters, assumed to be sampled once in the process of generating a corpus. The variables θ are document-level variables, sampled once per document. The variables z and w stand for word-level variables and are sampled once for each word in each document. The parameters of LDA that used to generate models during sampling consist of the number of topics, the iterations need for coverage and the hyper parameters α and β. α affects the distribution of topics per document and β has influence on distribution of term per topics. In practical application, appropriate parameters tuning for LDA could lead to precise reflection of the underlying topics representing each document and thus improve the learning accuracy. More specifically, LDA assumes the following generative process for each document w in corpus D:

* 1. Sampling from Dirichlet distribution α and generate Ai  for document i
  2. Sampling from Dirichlet distribution β and generate Bk for topic k
  3. for the jth word of document i in D,
     1. sampling from multinomial distribution Ai  and generate topics for Di,j
     2. sampling from multinomial distribution Bzi,j and finally the wi,jis generated

Although LDA was originally proposed for learning natural language text, the assumption that features in programing language seems to have similar characteristics as those in natural language makes it possible to apply LDA in software artifacts, and could be leveraged to support software maintenance tasks such as feature location. In the context of using LDA for feature location, the collection of documents used for input could be program elements such as source code classes or methods, bug trace information or any other textual software artifact. The Eclipse extension tool called TopicXP has been successfully developed and published in 2010 (Savage, Dit, Gethers & Poshyvanyk, 2010). The plug-in aims at supporting programmers in daily tasks, especially in software maintenance, by extracting the unstructured embedded information from source code comments and identifiers using an advanced information retrieval technique, that is Latent Dirichlet Allocation. The topics generated from the trained models are mapped to the source code, and the underlying relationship among the topics is then determined by censoring the static dependencies of the code components. The developers are able to browse through these topics and access the source code related to the topics with a graphical user interface. The results of an experiment where respondents were required to perform functionality location over two software (jEdit and muCommander) using Eclipse plus TopicXP or using just the pure Eclipse IDE shows that it is a competitive axillary tool compared to Eclipse IDE and even more efficient in some cases regarding feature location.

## 2.3 Document Vector

Document Vector or Paragraph Vector (Le & Mikolov, 2014) is a genre of unsupervised learning algorithm based on deep learning, which enables learning continuous distributed and variable-length sentences or documents. DV takes advantage of not only the context of the document but also the semantics of the document and uses them as features in a prediction task. The algorithm of DV is inspired by and developed on top of the approach of learning distributed vector representation of words using neural networks (Bengio et al, 2010) whose sprit is that the surrounding words in the context contribute to predicting a word in the sentence.

As shown in Figure 2, the fourth word “on” could be predicted via capturing the context of the term whose preceding three words are “the”, “cat” and “sat”. During the training phase, a sliding window that has fixed length is used for sampling. Each single document in the corpus is mapped to a unique vector, represented by a column in matrix D and every word is also mapped to a unique vector, represented by a column in matrix W. Afterward the document vector and the word vector are concatenated or averaged to infer the next word. The paragraph token could be considered as another word. The paragraph token is like a supplement for the missing information of the current context of the paragraph and could be used as memory of the paragraph topic, which is why the model is called the Distributed Memory Model of Paragraph Vectors. The paragraph vector (a column of D) is unique among other paragraph vectors (the other columns of D) while the word matrix W is shared across documents (the word itself has the same meaning even in different paragraphs). Being similar to the process of learning word vectors, both paragraph vectors and word vectors are trained using stochastic gradient descent and backpropagation (Rumelhart, David, Hinton, Geoffrey, Williams & Ronald, 1986). The parameters of the model (a matrix of weights) are updated in every iteration, which makes the model gradually approximate the underlying target function. During the prediction phase, the new paragraph that needs to be inferred is vectorised and inputted into the trained model whose output is also a vector representing that document, which could be fed into any of the conventional classifiers such as logistic regression.

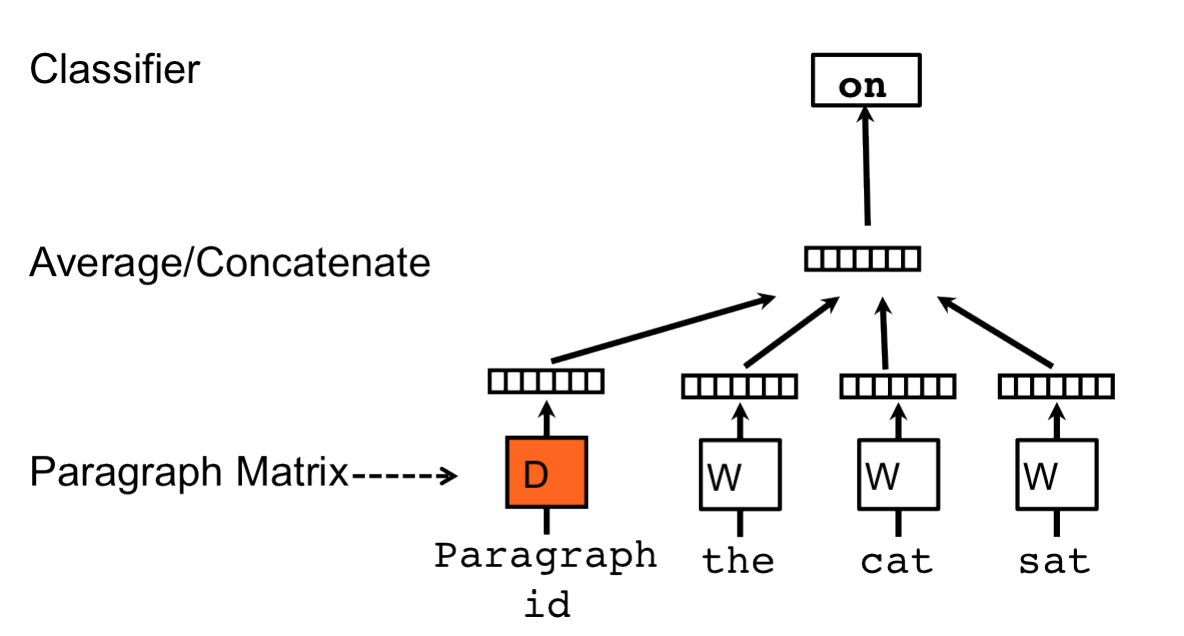


Figure 2 Example of Learning Paragraph Vector

Compared to other conventional methods like bag of words and n-gram for text learning tasks (e.g. text classification and sentiment analysis), one of the merits of DV is that it is capable of retaining adequate semantics of the corpus during training. Results on a couple of text classification tasks show that DV is competitive to state-of-art approaches with respect to learning representation for sequential data (Le & Mikolov, 2014).

2.4 Related Work

### 2.4.1 A Dataset from Change History

A dataset from six Java applications was created and published in 2013, in order to support various software maintenance activities like feature location and impact analysis (Dit, Poshyvanyk, Holtzhauer & Kagdi, 2013). It contains a set of benchmarks that could be used for evaluation and comparison of effectiveness of different approaches in software maintenance. These benchmarks contain textual description of change request, the specific locations of their implementations in the code base and execution traces.

A vocabulary of artifacts is defined and described in Dit et al.’s paper. The portion of which relates to this thesis is stated as below:

* 1. Dataset/Benchmark: a collection originated from the source code repositories and ITS (issue tracking system, e.g. Trac, Zendesk).
  2. Issue: is a universal terminology associated to change requests, which could be feature requests, bug reports, and any other types of information submitted to ITS.
  3. IssueID: is the unique identification of an issue, which is automatically generated and assigned by ITS in general.
  4. Goldset: is the set of unique method names where each item is associated with the method that were modified when an issue is implemented in the system. More precisely, they are the names of functions of the corresponding changed methods, which are fully qualified by the way of recoding package name, class name, method name and signature for each item. Every goldset is identified by a unique IssueID.
  5. Trace: is the execution trace for an issue. It represents the information of execution trace that was collected through reproducing the same scenario mentioned in the description of the issue. A trace is described by a sequence of methods that were executed when a stakeholder tries replicating the behaviours led to the bug or test a feature. Each trace is recognized by a unique a unique IssueID.
  6. Query: represents the textual description of the issue, and is composed of the title and detailed description. Each query is distinguished by a unique IssueID.
  7. Corpus: is a collection of textual documents which could be the contents of the source code files, classes, methods.

A methodology with respect to generating datasets is proposed, whose workflow is demonstrated as followings.

* 1. Choosing the software systems. The first step is selecting a Java software projects with following traits 1) uses SVN as software version control system and code repository, 2) has an ITS that keeps recording the change requests, 3) a subset of log messages mentioning issueIDs and optionally 4) the software allows collecting execution traces.
  2. Choosing the SVN release versions. The second step is firstly picking out two major releases and then analyzing the commit log messages between two releases. The earlier release is referred as previous version while the older one is declared as current version.
  3. Retrieving the issues. The log message of each commit is resolved so as to find out which of them mention IssueIDs. These commits are assumed to be associated with the corresponding issue. For instance, the commit #123 with log message “implementation of request #45678” is mapped to issue #45678. It is worth noticing that Dit el al. manually verified each mapping to make sure the correctness of the data and abandoned the commits not containing issueIDs. In addition, they also handled the cases of where an IssueID was mapped to multiple commits.
  4. Generating the goldsets. For each commit associated with an issue, the current version of each modified file is parsed then compared to the previous version of the file such that methods that have been changed can be recognized during the commit. These methods are part of the goldsets with the IssueID the SVN commit maps to.
  5. Generating the corpus. The corpus is produced via parsing all the Java files in the current version code base, then collecting the context associated with a method, which includes name, signature, function body, comments and modifiers.
  6. Generating the execution trace. The execution traces on the current release is generated through reproducing the same scenario mentioned in the description of the issue.
  7. Cleaning up the data. Some of the goldsets generated in previous steps were eliminated from the ultimate dataset. An example given by the authors is only the goldsets with at least one method in the corpus and at least one method in trace execution were kept.

The final dataset they successfully generated is extracted from six public open source software system. As shown in Table 1, the dataset includes 633 queries for method level goldsets and was automatically collected from changsets that relate to quires on issue tracking system. These open-source systems are ArgoUML, jEdit, JabRef and muCommander.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data | Description | Issues | Version Interval | Files | Methods |
| ArgoUML 0.22 | UML diagram tool | 91 | 0.20-0.22 | 1439 | 11000 |
| ArgoUML 0.24 | UML diagram tool | 52 | 0.22-0.24 | 1480 | 11464 |
| ArgoUML 0.26.2 | UML diagram tool | 209 | 0.24-0.26.2 | 1752 | 14597 |
| JabRef 2.6 | BibTeX bibliography management tool3 | 39 | 2.0-2.6 | 579 | 4607 |
| JEdit 4.3 | text editor | 150 | 4.2-4.3 | 503 | 6413 |
| muCommander 0.8.5 | cross-platform file manager | 92 | 0.8.0-0.8.5 | 1069 | 8187 |

*Table 1 Dataset from Change History*

A few Limitations of the dataset is presented by Dit and the co-authors. Due to the approach for generating the dataset, the quality of available sources of information and the refactoring process between two continuous version releases, members of the methods from the goldsets do not appear in the corpus. Furthermore, the SVN commits with the log message not includes description of issue are discarded.

### 2.4.2 The Document Vector for Method Level Feature Location

Previous researches show that the DV-based feature location techniques have better performance than an analogous FLT based on LDA in terms of feature location at the level of method granularity (Corley, Damevski & Kraft, 2015). The experimental results reveal that DV model has lower computational cost regarding model training while maintaining accuracy on par with LDA, which makes it a potential solution of implementing an intelligent programmer search tool in IDE.

The paper of Corley et al. introduces a series of methodologies for corpus preprocessing, model comparison and evaluation with respect to method level feature location based on the dataset generated by Dit et al. (see section 2.4.1). The experiment workflow starts with corpus preprocessing, whose purpose is transforming the original corpus into the sensible input that could be directly fed into the learning algorithms. Before training, series of preprocessing steps are employed on the corpora and are executed in following order:

* Splitting: split items of corpora into tokens according to the conventional coding style rules (e.g., the use of camel case or underscores) and on the appearance of non-letters (e.g., punctuation or digits).
* Normalizing: replace any upper case letters with corresponding lower case letters.
* Filtering: remove stop words, short words, programming language keywords, and standard libraries entity names.

After extracting, tokenizing and preprocessing the documents of both the corpus of source code corpus and the corpus of query, the performance of LDA and DV on training time and accuracy respectively in terms of method level feature location are evaluated by applying the learning methods on systems then measuring the outcomes with the goldsets. In the training phase, the approach is quite straight forward for both DV and LDA, that is training a model using all source code as input data. As for prediction, the vector related to the query is firstly obtained from the generated model and a rank, sorted by similarity to the query is calculated and inferred using that output. Finally, the rank can be evaluated with the metric.

Moreover, they found out a method based on word vector are efficient for locating feature through firstly gaining the word vector for each word in the query and sum them then taking the summed vectors and perform pair-wise similarity to each document vector in the trained model. The impact of one of the model parameters, that is number of topics (the analogous parameter in DV is vector size), was also investigated in their research. It turns out that a few trends could be observed from the results. The DV inference has the worst performance on all versions of ArgoUML, JabRef and jEdit except for muCommander where it beats the other two methods, while LDA’s performance is in general a bit better comparing to that of DV inference. The DV word vector summation method performs the best on most projects. The DV inference seems much faster than LDA and DV word vector summation in respect with training time.

# 3 Problem and Solution

## 3.1 Research Questions

### 3.1.1 Automated Goldset Generation

Dit et al.’ methodology for generating the dataset is considerably limited and trivial, for example, only the softwares supplying an issue tracking system could be used as data source for generating goldsets (step 1) and only the software systems with different release versions or tags could be used as materials (step 2). Moreover, the workflow of their approach involves the manual process, which indicates its low scalability to other software projects (step 3).

Therefore, the first question we investigate is that if it is possible to using github rather than SVN as the code repository for software systems concerning to generating the benchmarks for evaluation of approaches for software development and maintenance activities especially for feature location. Furthermore, we also research on that whether commits instead of issues could be used as the basic unit of goldsets during dataset generation.

### 3.1.2 Experiments on Wider Range

Despite the fact that the work of Corley et al. seems to have proved that the FLT based on deep learning outperforms the FLT based on Latent Dirichlet Allocation with regard to method level feature location, the conclusions are less persuasive and restricted due to the limitation of the datasets, which is the lack of statistical significance on the number of test subjects. The dataset generated in work was derived from only 6 software projects and three of them are just different release versions of the same project (ArgoUML), not to mention all of them are written in Java.

The second question that is probed in our study is that if the experiments and the relevant results could be scaled up and extended regarding the amount of the subject software systems and programming language that used to develop the systems with the emphasis on the utility of LDA and DV for feature location at the file rather than method level.

Another direction of exploiting the use of LDA and DV for feature location is stretching the types of subject software systems in the matter of different programming language, for instance, Python. Since the context and semantics embedded in Java source code could be captured by machine learning approaches and be used for feature location, being similar to Java as an Object-oriented programming language, Python is very likely to have the homologous characteristics on which the machine learning can be utilized in connection with feature location.

### 3.1.3 Impact of Model Parameters

At the end of Corley and the co-authors’ paper, they proposed one of the directions for future work is to explore the impact of parameters tuning on DV. In their experiments, the only parameter that has been surveyed is the number of topics for LDA (the corresponding one in DV is vector size of the input). In order to reduce the possibility that the experimental results and the conclusions drawn from the experiments are confounded by confounders like arbitrary configuration of model parameters, the last question that is worth considering is that how much the effectiveness of the FLTs based on machine learning could be effected by attempting different number of iteration during model training.

### 3.1.4 Our Work versus Corley et al’ Work

The overall comparison between our work and Corley et al.’s work is clearly demonstrated in table 2. In order to overcome the weakness of Corley et al.’s work including the potential bias of the results caused by the lack of test subjects and the types of programming language, our experiments are based on 50 Java projects and 50 Python projects respectively. In addition, our work concentrates on the exploration of the effectiveness of DV and LDA for feature location at the level of file rather than method.

|  |  |  |
| --- | --- | --- |
| Difference | Ours | Corley et al.’s |
| Data Source | github | SVN |
| Feature Location Granularity | file | method |
| The Identification of Goldsets | commit | issue |
| The Number of The Subject Software Systems | 50 + 50 | 6 |
| Programming Language of The Subject Software Systems | Java and Python | Java |

Table 2 Our Work vs Corely et al.'s Work

## Experiment Design

### 3.2.1 Methodology Overview

The methodology that we probed into the research questions demonstrated in section 3.1 could split into three independent phases:

1. Goldset generation. The first stage is the generation of a qualified datasets that could be used as the benchmarks regarding the effectiveness assessment of the LDA-based FLT and DV-based FLT for feature location.
2. Model Training. The second step is generating the predictive models using all the source code files from the corpus of each subject software system respectively.
3. Empirical Evaluation. The last phase is, for each subject software system, calculating the similarity between a given query and all the source code files for each query from the goldsets, and output a many-to-one rank list according to their semantic distance to the query. Afterwards, the models and the associated FLTs are evaluated regarding their performance on feature location with the rank lists.

### 3.2.2 System Design

The implementation of the methodology is supposed to result in the derivation of a solid software system equipping with a suit of handy tools, which should be able to empower other researchers to reproduce our experiments, or apply the same approach in any other Java or Python git projects for model evaluation concerning file level feature location, or easily extend it into a practical application for feature location on Python or Java files, for example, a smarter search tool in IDE. Based on the combination of the points stated above, the system ought to be low coupling, well scalable and friendly to developers, thus object-oriented-based design seems to be a reasonable solution.

As the general picture of the system design indicated in Figure 3, the software consists of three modules, which are Goldset Generator, Model Trainer, Common Utilities and Corpora Generator. The modules of the system are described in the following respectively:

* Goldset Generator: This module is responsible for accessing and manipulating the subject software systems and goldset generation. It generates the goldsets by visiting the records of the files among different versions of the system in the git object and then analyzing and extracting the necessary changeset information (e.g. package name, file name, method name).
* Corpora Generator: This module is responsible for generating the corpora of the queries or the corpora for the code base using all the source code of a subject software system or the queries from the goldsets as raw input. It also supplies preprocessing like splitting and tokenization before yielding each document of the corpora.
* Model Trainer: This module is in charge of model training using the structured corpora generated from the Corpora Generator as input, as well as the prediction task (is also the process of feature location) for the final rank list generation given each single entity of the queries from the goldsets.
* Common: This module is composed of the collaborative functionalities that supports the Goldset Generator, Model Trainer and the Corpora Generator module in the aspects of project configuration, various utilities, common variables, etc., which makes the system more scalable, flexible and configurable.

The complete system implementation is demonstrated in detail in the next chapter.

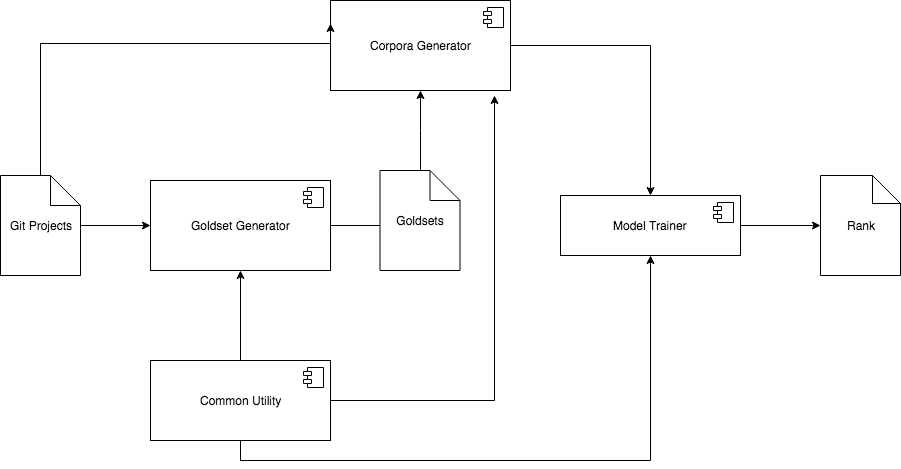


Figure 3 Architecture of The System

# Experiment Implementation

## Related Tools & Libraries

### 4.1.1 Git & Github & GitPython

Git is a free and open source distributed version control system designed to handle the software projects ranging from small to very large in terms of the tasks during software development and maintenance like source code management by tracking historical changes on computer files especially source code and coordinating work on those files among many programmers, with friendly API and features like local branching and multiple workflows. Git was firstly created in 2005 by Linus Torvalds for the purpose of development of the Linux Kernel. As with most other existing mainstream distributed control systems, every git directory on every machine is an independent repository with full history and version tracking, leading to the invulnerability when network access is not available.

Github (https://github.com) is the worlds’ leading software development platform and the largest community for software developers or, any programmers to share and build better software, which is established as an online hosting web service for software version control system using Git. In other words, github is a centralized code repository of the software projects that uses git for version management.

The specific programming interface that we utilized in this thesis for interacting and operating git repositories is GitPython (https://gitpython.readthedocs.io), which is written in Python and provides abstraction of git objects for light and efficient access of repository data with either pure Python implementation, or the faster, git command implementation. One principle advantage of GitPython is that the optimized object database implementation enable dealing with large amount of objects and large body datasets using streaming and low-level structure. We took advantage of the GitPython to access and manipulate the source code files, visit their historical changes and extract the related information of the goldsets or prepare the corpus for training.

### Gensim

The specific implementation of the algorithms of LDA and DV that was used in our research is provided by Gensim (https://radimrehurek.com/gensim/). Gensim is a vigorous open-source topic modeling and vector space modeling toolkit written in Python. It is based on SciPy (an open-source Python library used for scientific computing), Numpy (a library that supports a large collection of mathematical functions to operate multi-dimensional matrices and arrays), which is designed and developed to handle large text collections using data streaming and efficient incremental algorithms. Gensim provides implementations of various machine learning algorithms such as latent semantic analysis, LDA, word vectors, document vectors, tf-idf and so on including distributed parallel version.

The ad-hoc LDA implementation in Gensim allows both model estimation from a training corpus and prediction of topic distribution on unseen, new document. The specific class for LDA in Gensim is called *LdaModel,* which supports parameters tuning with respect to number of topics, the number of iterations, decay, configuration of the dictionary that is used for mapping word id and words, etc. As for the document vectors or the paragraph vectors with deep learning, the specific implementation in Gensim is called Doc2Vec, which allows tuning the parameters like alpha, beta, dimensionality of feature vectors, the size of window that used for sampling, number of iterations (epochs) over corpus, the number of working threads during training, etc.

## Goldset Generation

Inspriend by Dit et al.’s work, we introduce a new methodology in this section, which is capable of automatically generating a set of benchmarks from the software projects that use git for version control, which can support the effectiveness evaluation for the approaches in software maintenance like feature location. We then applied this methodology on 50 Python projects and 50 Java projects, which successfully generated 100 datasets in total, preparing for the evaluation of the FLTs.

### The Subject Software Systems

The 50 subject software systems written in Python are drawn from the Qualitas Corpus (Orru, Tempero, Marchesi & Tonelli, 2015). In the paper, Orru et al. presented a dataset of metrics relevant to a curated collection of Python systems, which aims at offering a benchmark for empirical studies of Python systems that permit reproducibility of results and lowers the overhead of experiments. The samples in the corpus cover Python systems varying in different domain, different Python versions, number of source code files and number of program elements (classes, methods, lines of code, etc.). The comprehensive descriptions of the 50 Python projects are available in the appendix. All the systems are downloaded before the experiments via git clone.

As for the 50 Java subject systems, they are provided by the GitHub Archive and filtered individual projects that were forked at least once (http://groups.inf.ed.ac.uk/cup/javaGithub/). For each of these projects the URL, the project's name and its language were collected and filtered by forks aims to create a qualified corpus. The projects in the corpus are also ordered according to popularity, defined as the number of forks plus the number of watchers (<http://groups.inf.ed.ac.uk/cup/javaGithub/top_projects.tar.gz)> . We sampled the top 60 projects of the corpus in terms of the number of folkers rather than the top 50, because some of the top 50 projects have been depreciated or not available anymore. The detailed descriptions of these Java projects are available in appendix.

### Glossary of Artifacts

Before introducing the procedure of producing the dataset, a bunch of artifacts in terms of the need to be defined in advance:

* Dataset/Benchmarks: A collection of documents generated from the source code files and git commit logs of the subject software systems.
* Commit: is a frequent and conventional operation in git that represents the behavior of storing the current contents of the index in a new commit along with a log message from the user describing the changes of the files in the code base.
* CommitID: is the unique identification of a commit, which is automatically generated and assigned by git. More precisely, it is a string that is formatted in forty hexadecimal characters that specify a 160-bit SHA-1 hash, which uniquely represents the new, post-commit state of the repository.
* Goldset: is the set of source code file names where each entity is associated with the source code files that were changed when a commit is submitted to the git. More specifically, they are the names of the corresponding modified source code files, which contains the package name and the file name for each item. Every goldset is identified by a unique CommitID. One CommitID is mapped to one or more goldsets.
* Query: represents the textual description of a commit, which is delivered by software developers and normally describes what changes have been made to the code base or the reports regarding bug fix, the implementation of a features and branch merging, etc. Each query is distinguished by a unique CommitID. One CommitID is associated to one query.
* Corpus: is a collection of textual documents representing the all the contents of source code files or queries.

### Goldset Generation Workflow

The workflow of the goldset generation is quite analogous to Dit et al’s, which consists of the following steps:

1. Choose the subject software system. The workflow starts with selecting the Java or Python projects with the following characteristics 1) use git for version control and 2) have sufficient historical information regarding file changes. As mentioned in section 4.2.1, in our case the test objects are 50 Java systems and 50 Python systems.
2. Choosing the reference commit. The second step is picking out one of the commit records which is used as the starting point for goldset generation, for instance, a latest commit or the commit associated with the latest release. In our experiments, we chose the latest commit as the reference commit.
3. Generating the goldsets. For each system, this step begins with setting up the total number of CommitIDs wanted (50 in our survey). Starting at the reference commit (reference commit inclusive), the commits recorded in the git object are visited backwards one by one, for example, the reference CommitID is #123, then the next commit will be visited is #122, until the number of goldsets is reached. For each commit, we use git diff command to check the changed source code files and only mark down the needed information (package name, file name, the commit message aka query, the commitID) using the commitID as ID.

## Feature Location

Based on the way of how Corley et al. researched on the use of the FLT that is based on DV and compared it to the LDA-based FLT for method level feature location, we explore the FLTs that utilized LDA or DV as learning strategy in respect with their effectiveness for feature location in file granularity.

### Corpora Generation

The approach to generate the corpora is quite straight forward. The corpora could be categorized into query corpus and the code corpus. The code corpus is generated from the source code files of the subject software systems using the whole code base as input and is considered as the training set for the machine learning models.

The query corpus is produced from the queries of the goldsets that relates to the subject software system, preparing for the use of querying the trained model later. It is worth noting that unlike Corley et al.’s approach, for the code corpus, an item yielded from the corpus is the representation of a file rather than a method. As for the query corpus, an entity of the corpus indicates a query mapped to a commit.

More specifically, no matter what kind of corpus is generated, the following steps are employed.

1. Preprocessing. For each document (a source code file for code corpus or a query for query corpus), the same preprocessing procedure as Corley et al. are supplied (see section 4.2.2), which are splitting, normalizing and filtering. The only difference is that in filtering phase, the corresponding programming key words will be removed depending on the specific programming language.
2. Yielding documents. After cleaning and tokenization, each document of the corpus is yielded in the form of a pair-wise tuple that maps the meta data of the document to a sequence representing the vectorized tokens of the document. For the code corpus, the meta data is the path of the relevant file and for the query corpus, it is the commitID of the corresponding query.

### Model Generation

Since our investigation focuses on LDA-based FLT and DV-based FLT in terms of their effectiveness for feature location, there are two genres of models need to be trained.

For document vectors, we applied the ad-hoc implementation of document vector in Gensim, that is Doc2Vec, on the corpus and successfully generated a model for each subject software system with the size of vectors as 500 and the number of epochs as 30 for the prediction task.

For LDA, the workflow was quite similar but an extra effort needed to be made, that is each corpus was transformed into a matrix where each element of it represents the frequency of word that appears on the document (bag of words). Like DV, the LDA model of each subject system learned the corpus with the number of topics being 500 topics and the number of iterations being 30.

What’s more, 5 Java projects and 5 Python projects were chosen respectively regarding the number of source code files in the code base, in order to evaluate the impact of different configuration of parameters on the performance for file level feature location. More precisely, for the 5 Java projects and 5 Python projects, we set the number of topics to the range from 100 to 500 at the step of 100 respectively and the times of iterations ranging 10, 30, 50, 80, 100 respectively, leading to generation of 25 models in total for each selected subject system. The details about the projects for the parameter tuning experiment are shown in chapter 5.

### Rank Generation

The final step is outputting rank lists according the queries with the use of the generated models. For each subject software system, we calculate the similarities (cosine distance) between each document from the query corpus and each document from the source code corpus in terms of the, whose process is equivalent to the process of locating the corresponding source code file via user’s queries.

# Reference

Dit, B. , Revelle, M. , Gethers, M. and Poshyvanyk, D. (2013), Feature location in source code: a taxonomy and survey. J. Softw. Evol. and Proc., 25: 53-95.

V. Rajlich and N. Wilde, "The role of concepts in program comprehension," *Proceedings 10th International Workshop on Program Comprehension*, 2002, pp. 271-278.

Blei, D. M., Ng, A. Y., and Jordan, M. I., (2003), "Latent Dirichlet Allocation", Journal of Machine Learning Research, vol. 3, pp. 993-1022.

Roweis, S. T.; Saul, L. K. (2000). "Nonlinear Dimensionality Reduction by Locally Linear Embedding". Science. 290 (5500): 2323–2326.

Q. Le and T. Mikolov, “Distributed Representations of Sentences and Documents,” in Proceedings of the 31st International Conference on Machine Learning, T. Jebara and E. P. Xing, Eds., 2014, pp. 1188–1196.

Rumelhart, David E, Hinton, Geoffrey E, and Williams, Ronald J. Learning representations by back-propagating errors. Nature, 323(6088):533–536, 1986.

Bengio, Yoshua, Schwenk, Holger, Senecal, Jean-Sebastien, Morin, Frederic, and Gauvain, Jean-Luc. ´ Neural probabilistic language models. In Innovations in Machine Learning, pp. 137–186. Springer, 2006.

Binkley, D. and Lawrie, D., (2010a), "Information Retrieval Applications in Software Development", in Encyclopedia of Software Engineering, P. Laplante, Ed.: Taylor & Francis LLC.

Binkley, D. and Lawrie, D., (2010b), "Information Retrieval Applications in Software Maintenance and Evolution", in Encyclopedia of Software Engineering, P. Laplante, Ed.: Taylor & Francis LLC.

Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T., (1987), "The Vocabulary Problem in Human-System Communication", Communications of the ACM, vol. 30, no. 11, pp. 964-971.

C. S. Corley, K. Damevski, and N. A. Kraft, “Exploring the use of deep learning for feature location,” in Int’l Conf. Software Maintenance and Evolution, Sep. 2015, pp. 556–560.

Savage, T., Dit, B., Gethers, M., and Poshyvanyk, D., (2010a), "TopicXP: Exploring Topics in Source Code using Latent Dirichlet Allocation", in Proceedings of 26th IEEE International Conference on Software Maintenance (ICSM'10), Timişoara, Romania, September 12-18, pp. 1-6.

B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. Kagdi, “A dataset from change history to support evaluation of software maintenance tasks,” in Proceedings of the 10th IEEE Working Conference on Mining Software Repositories, 2013, pp. 131–134.

E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, “Qualitas corpus: A curated collection of java code for empirical studies,” in Proc. 2010 Asia Pacific Software Engineering Conference (APSEC’10), 2010.

Matteo Orru, Ewan Tempero, Michele Marchesi, Roberto Tonelli, and ´ Giuseppe Destefanis. A curated benchmark collection of Python systems for empirical studies on software engineering. In Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE ’15, pages 2:1–2:4, New York, NY, USA, 2015. ACM.

Susan Hunston. Corpora in applied linguistics. Cambridge University Press, 2006.

# Appendix