# Predict if a git commit is bug fix - BUGZY

Vivek
Computer Science, North Carolina State University
Raleigh, NC
vivek.7266@gmail.com

## ABSTRACT

This paper discusses the possibility of predicting if a commit is a bug fix or not. Keywords based models have been mostly used in the past. Every software project has certain unique characteristics associated with them with respect to change messages. Hence, it is required that an automated machine learning model, which captures local characteristics of the project, be best to classify change messages. This project, BUGZY, proposes to apply topic modeling (Latent Dirichlet Allocation) to understand the keywords associated with a project. Words' probabilities are then used to generate features for Support Vector Machine. The project goes on to compare TF-IDF-vectorized features with SVM, TF-vectorized features with SVM and BUGZY (LDA with SVM). Final results are discussed corresponding to the baseline model of TF-IDF SVM. Comprehensibility of this automated machine learning model, BUGZY, is discussed. It is established that LDA based SVM adds to comprehensibility and stability to a change message classification model.

## KEYWORDS

Latent Dirichlet Allocation, Support Vector Machine, Defect Prediction, Natural Language Processing, BUGZY

**ACM Reference Format:**
Vivek. 2018. Predict if a git commit is bug fix - BUGZY. In *Proceedings of Computer Science, North Carolina State University (FSS 2018).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Defect prediction in software projects is of crucial importance. This helps software developers and project managers keep a tab on what changes are harmful for software development life cycle. Most importantly, defect prediction will be helpful in reduction in time for triangulating a buggy code and fixing it. However, to create a defect prediction model, it is of paramount importance to first understand the features of a buggy code.

Almost every software project uses version control system. Not only does version control system maintain the software, it also helps maintain owners of a particular piece of code. It can be well established with a particular change, who did the change, for what purpose, at what time, what files that change corresponds to and why that change. *Why that change* is very important as this natural language message can give a wealth of information on the nature of the change introduced. We have open source software projects available for mining through popular git version control named 'github'. We can mine change message for particular projects from github.

There are numerous amount of open-source software repositories on github.com that have been managed for years now. It can be learned from their history to understand commit messages. Commit messages give a huge amount of information. Big chunk of information can be obtained from the natural language used and key words that uniquely characterizes a particular change. This unique meaning of certain messages can be characterized as a bug fixing commit or not. If a change message is correctly characterized as bug-fix change, then it will help defect predictor learn features of a bug, by looking at the changes that were introduced to fix that bug. Hence, it becomes very important to identify a change message as corrective change. In particular, predicting if a git commit corresponds to a bug fix.

In this project, we explore automated machine learning methods to predict a git commit as a bug fix. We will see in later sections that in past a dictionary based approach has been taken to identify words in a message that point to a bug fix [8]. We explore a general model that can be applied on a project to find specific "bug fix indicating words". Most of the git projects are managed by only a certain group of individuals, hence, they tend to have a characterizing way of expressing commit messages. However, as all of them are software projects, certain software engineering domain specific words are frequently used. These general keywords are more prominent to identify a bug-fixing commit, but not exhaustive. Due to this localized nature of software management, we need to explore a model that learns these words specific to a project. In a nutshell, we will explore a generalized "local" automated model for discovering bug-fixing messages.

Natural language processing is core to understanding of git commit message. Earlier works have been based on natural language processing of these messages for change classification. Mockus et al. [17] studied and created classification based on word frequency analysis and keyword clustering. It does establish that git commit message expressed in natural language has a wealth of information that can be successfully mined. Conversion of natural language expressed in text to vectors understandable by classifiers requires a method that maps words to vectors efficiently ideally conserving the syntactic and semantic information. However, this is very computationally expensive. More popular and less computation expensive vectorization method are based on word counts across documents. Generally, frequency based features generation are used, namely Count vectorization and TF-IDF vectorization. This forms our first research question:

- **RQ1:** Is the occurrences of terms in a particular document in classifying bug-fix message or the discriminatory nature


*FSS 2018, December 2018, Raleigh, NC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of term frequency weighted by logarithm of document frequency better?

Mauckza et al. [14] proposed a variation of keyword based approach to automated classification change messages. This work has been based on Mockus et al. [17] natural language "modification request" words. Keyword based models have been a benchmark for change message classification. In this project, we aim to question if a generic keyword dictionary is suitable for all projects. Or if there exists some localized natural language reference of equal importance with respect to global software engineering specific keywords. This leads to our second and third research questions:

- **RQ2:** Do we specify generic keywords for all software projects for change message classification? Can we create a general unsupervised keywords identification algorithm local to a software project?
- **RQ3:** Does a supervised classification algorithm based on auto-detected keywords work better than baseline classifiers? Does it add to comprehensibility of the model?

This project is aimed at an exploratory analysis of above research questions and lay framework for future research work.

## 2 BACKGROUND AND RELATED WORK

Change messages do capture the life story of a software project. Mining of software repositories has been done for a long while now. This helps us analyze the data for software development life cycle. The research field of change classification has been evolving over many decades. The first classification system of software changes was put forward by Swanson in 1976 [19]. Swanson defined that a change can be classified in one of the following 3 categories:

- Corrective: Changes that are fix for prior failures.
- Adaptive: Changes in environment.
- Perfective: Changes to improve efficiency.

Corrective software changes have been studied before for the purpose of software maintenance [17] and defect prediction[12]. Mockus and Votta [17] established that to understand the motive of introducing a change in the software can be estimated from the associated message. Natural language processing comes into play when understanding these change messages. Initial models are in majority built on keyword basis [17] [8] [7] and Naive Bayes classifiers [3]. Antonio et al [3] proposed a method based on decision trees, naive bayes and logistic regression models that does change message classification. However, Support Vector Machine (SVM) remains a popular choice for this classification [10] [12]. Hindle et al. [9] employed the Weka Machine learning version of SVM.

Hindle et al. [10] discusses Latent Dirichlet Allocation (LDA). The proposal says discovering change message classification topics over a window of time. LDA based method is used to analyze unstructured data. LDA based text mining proves to be useful in increasing comprehension of the model as opposed to TF and TFIDF models [2]. However, LDA does suffer from order bias that can be overcome in a certain way [1]. So, the usage of such Natural Language Processing (NLP) techniques can help capture hidden information from the texts. Another tool that has been studied and shown to have success is pLSA (probabilistic Latent Sematic Analysis) [11]. However, as we are not trying to find semantic information, rather looking for binary classification of change message, we choose LDA for our study. Studies have shown the success of LDA models for defect prediction [20] [6] [2]. Success of these works have motivated us to explore topic modeling, specifically LDA for our studies.

Kim et al [12] mentions that a *bug-introducing* change can be identified through the analysis of a bug-fix. These bug fix changes are in turn identified by mining change messages. He proposes that it can be done through two ways:

- Keyword search model aimed at searching for well identified keywords such as "Fixed" or "Bug". This is a reasonable assumption and is also seen from Hindle et al. to be successful [9]. This sort of injection of domain knowledge into the classification of change message does prove helpful. Therefore, we use a similar model as our baseline.
- Searching for references for bug reports such as their IDs. It is very difficult to track as Issue tracking system is often used for Feature tracking too. We do believe that injecting more domain knowledge may help us identify bug-fixing commits more precisely. However, using issue tracking system IDs does not resonate well with our main idea.

These methods of identifying bug-fixing commits are costly. It is not easy to get messages labeled for each project individually. A generic keyword based model does well for identifying the bug-fix commits, but it is not local to the project. Tracking issue IDs is local to a project but is not a scalable model. We learn from this limitation and try to come up with an algorithms that is both scalable, and can identify both general and local keywords corresponding to a bug-fix commit message.

## 3 RESEARCH METHODOLOGY

### 3.1 Data

The data under study here is collected through mining of open source software projects from github.com. Four projects were selected for this study. These are popular github projects in the field of computation. They have a large number of commits, as will be explained later. The commit messages were labelled during a mechanical turks study. The mechanical turks were graduate students from the Computer Science department at North Carolina State University, Raleigh, NC, USA. The four projects are as follows:

(1) **abinit**
(2) **libmesh**
(3) **mdanalysis**
(4) **lammps**

Each of the commit messages has been labelled as 0 or 1. Here, label 1 corresponds to a message being tagged as "buggy". By saying, "buggy" we imply that the commit message is corresponding a bug-fix. This does not mean that commit is a bug-introducing change. The value counts of each project is as shown in Figure-1. We see that each project has a large number of non-buggy commits compared to bug-fix commits. This is expected. Otherwise, a software project is poorly managed if there are a lot of bug-fix commits. Also, we can see from Table-1, that on an average, we have buggy commits to be at least 10 percent of the total commits. However, in case of

**Table 1: Class labels actual counts**

| Project | non-buggy(0) | buggy(1) | % buggy |
|---|---|---|---|
| abinit | 4024 | 572 | 12.44 |
| libmesh | 6462 | 651 | 9.15 |
| lammps | 7437 | 186 | 2.44 |
| mdanalysis | 2802 | 379 | 11.91 |

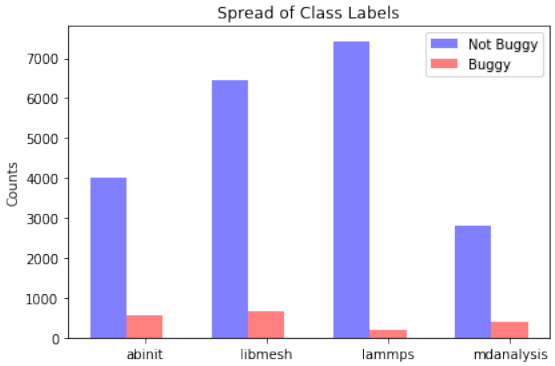


**Figure 1: Spread of class labels.**

*lammps* project we have only 2.5 percent of commits as bug fixing commits. We will later raise a question here if this is enough to train the model we propose.

## 3.2 Latent Dirichlet Allocation

Specific to text based corpus, we can say that Latent Dirichlet Allocation (LDA) is an unsupervised clustering algorithm aimed at clustering the words in the corpus to specific *topics*. Since, it is probabilistic in nature, it means that certain words can reside in different topics. According to Blei et al. [4] the joint distribution of LDA can be described as

$$P(w, z|\alpha, \beta) = P(w|z, \beta) * P(z|\alpha)$$

Here, $z$ is the topic and $w$ is the word. And, $\alpha$ and $\beta$ are model parameters. P(z|$\alpha$) is the probability of topic $z$ appearing in the document $d$. P(w|z, $\beta$) describes the probability of word $w$ appearing in a particular topic $z$. We use python scikit-learn implementation of LDA. There are certain hyper parameters to that model. *Number of topics* is the first and foremost hyper parameter. We need to decide how many topics prior to the training process. In our work, we classify change messages in two distinct values 0 or 1 indicating non-buggy or buggy commit message. Hence, our natural option for selecting *'k'* i.e. number of topics is 2. This ensures that the topic could correspond to these labels. Another hyper parameter is the Count (TF) vectorization of the words before sending to LDA for training. This is called *maximum number of features*, i.e. the number of dimensions. This sets the maximum number of words to be fed to LDA. We see that lower number of maximum features is much better [13]. For this experiment we default to the value of 25 for this purpose. Finally, another hyper parameter that we change in our experiment is *number of iterations* for LDA. Essentially, LDA is a generative probabilistic model. LDA has two matrices. First, describes the probability of selecting a particular word when sampling a topic. Second, describes the probability of selecting a topic when sampling a particular document. Now to build word versus topic matrix, a sample is drawn from dirichlet distribution for each topic using $\beta$ as input. For second a composite is defined as a document and a sample is drawn from dirichlet distribution using $\alpha$ as input. Then the actual composites are defined basis these two matrix for each document. LDA does expectation maximization in each iteration. We set *maximum number of features* as 10 for our study. The LDA iteration method is Gibbs sampling. The other hyper parameters namely $\beta$, and that we choose are taken from literature study [5] or set to default.

LDA is a common technique in text mining used for dimensionality reduction [1]. TF and TF-IDF are common feature generation methods applied to text mining. The order of number of features is 1000s or more. However, with LDA we get very small number of dimensions. As we keep only 2 topics, this is a reduced dimension from 25 as used in vectorization. The output of LDA is very comprehensible [1] [2]. We agree with their work and validate LDA and comprehensibility it offers in this paper.

## 3.3 Text Pre-processing

Even before we feed data to LDA, we need to process the natural language expressed in change messages. Some examples of change messages are as follows:

- Quickfix: paral_kgb=1 shoulkd be ignored for DFPT (was OK in previous versions)
- Fix bug that was preventing the kxc to be stored.
- multibinit: add initialization for the fit and add restartxf -3,0.0.
- Fixup receive_packed_range call.

These messages are expressed in a natural language, not necessarily following the English grammar well. Most of the the times it contains only few words. What we want is this message to be expressed in a form a vector. First of all we extract important words from this message. Using the python NLTK module, the words are first tokenized using WordPunct Tokenizer. Then all the tokens which are either numerical values of symbols or punctuations are remove. To remove redundant tokens, all words that are English stopwords like "is", "are", "has", etc are removed. One caution is taken that even if there remains any token of length 2 character or less it is removed. In addition, all tokens are stripped of white spaces and symbols like "-" and "_". Finally, the tokens are lemmatized. Lemmatization brings the natural language English word to its root form. For instance, "fixes" is reduced to "fix".

Above processing of words to a list of tokens gives us the corpus that we want the vectorizer to fit on. We use CountVectorizer (TF vectorzation) provide in scikit-learn for our study. One thing to note is that in our experiments we will also use TfidfVectorizer. This vectorizer is different from TF vectorizer as it penalizes those words that occur frequently across documents. Maximum number of features is defined here as the length of our feature space for our experiments. We define it as 25.

### 3.4 Support Vector Machine

Support Vector Machine (SVM) is a well established text based classifer [12]. SVM primarily is built for binary classification. It is a discriminative model that finds the boundary between the two class labels. It finds a maximum margin hyperplane. This maximum margin corresponds to a linear decision boundary with maximum width between the two labels. The positive and negative hyperplanes are the edges of this boundary for the classes. The formula for binary classification is

$$(x_i, y_i), ..., (x_n, y_n), x \in \mathbb{R}^m, y \in \{+1, -1\}$$

where, $(x_i, y_i), ..., (x_n, y_n)$ are training samples, n is the total number of training samples and m is the total number of features indication the dimension of the data, and y denotes the separation. In case of linear separation, the new point is classified basis this formula

$$(w.x_i) + b > 0 \text{ for } y_i = +1, (w.x_i) + b < 0 \text{ for } y_i = -1$$

The vectors that lie on these hyper-planes are called support vectors. SVM facilitates that a particular problem can be trained in higher dimensions, and yet perform computations in lower dimensions. This is done through 'kernel trick'. We choose linear kernel for our work. The hyper parameter here is penalty function that restricts how hard the margin can be. Even after doing kernel trick the computation remains inexpensive (involves only dot product in lower dimension) and this is good for a large number of feature set such as text corpus.

### 3.5 Naive Bayes Classifier

Bayesian classifier are simple and intuitive. It classifies a $d$-dimensional observation $x_i$ by determining most probable class that can be computed.

$$C = \max(P(c_k|d_1, d_2, ...d_d))$$

Naive Bayes classifier assumes conditional independence among all the attributes. This is known as "Naive Bayes assumption". Since, the parameters of the features are independently computed, because of the assumption, NB has low computational complexity in case of large number of features. For textual classification, we have large number of features that corresponds to number of words that we deem is representative of a document on average. There are thousands of words in the vocabulary for text based classification. Naive Bayes has been successfully applied to many document classification researches [3].

We assume that the documents are generated using a mixture model $\theta$. Mixture components $c_j \in C = \{c_1, ..., c_{|C|}\}$ are contained in the model. Each component is parameterized by a disjoint subset of $\theta$. Thus, a document i is created by following equation:

$$P(d_i|\theta) = \sum_{j=1}^{|C|} P(c_j|\theta)P(d_i|c_j;\theta)$$

The following Bayes rule is applied to classify a document by calculating a probability given a test document

$$P(c_j|d_i;\hat{\theta}) = \frac{P(c_j|\hat{\theta})P(d_i|c_j;\hat{\theta}_j)}{P(d_i|\hat{\theta})}$$

Since, the frequency of words can be transformed as conditional probabilities, it is also common to use Naive Bayes (NB) classification model for text based classification. Also, it has been shown that Multinomial model works better on large number of features than Multi-variate Bernoulli model [15]. Two documents are said to be correlated if they belong to the same category specified by the conditional probabilities based on the frequency of words. New document is classified basis Bayes estimates. Hence, we choose Multinomial NB for our work.

## 4 EVALUATION STRATEGY

The main aim of this project is to predict a bug fix commit by looking at the message. This prediction directly affects the developers as it will be an input to defect predictors. Therefore, we will evaluate the performance of the model by precision and recall. This model is a binary classifier. Hence, we can define metrics based on precision and recall by the use of confusion matrix. The definitions of these metrics are below:

- Precision: Defined as number of true labels as a fraction of total truly predicted labels, it says, number of true positives of all the positive predicted labels. Basically it indicates, of all positives that was predicted how many of them were actually positive. For precision score, the higher the better.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive + False Positive}}$$

- Recall: Defined as number of predicted true labels as a fraction of total true labels, it says, number of true positives of all the positive labels. Basically it means, of all the true labels how many did the model correctly recall. For recall score, the higher the better.

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive + False Negative}}$$

- F1-measure: F1 measure is the harmonic mean of precision and recall. For both precision and recall to be high, we need to have high F1 score.

$$\text{F1} = \frac{2 * \text{Precision} * \text{Precision}}{\text{Precision + Recall}}$$

## 5 EXPERIMENT SETUP

### 5.1 Baseline 1: Keyword based model

For the first set of baselines, we look at the keywords based model discussed by Hindle et al. [9]. The keywords are as explained in Table-2. We deal with only the corrective category provided in the table. A simple keyword search using only the corrective category gives us the result. Table-3 shows the evaluation metrics corresponding to the baseline. Our emphasis in evaluation metrics is F1-measure as we want to maximize both precision and recall. There is a lot of variation in F1-score among the projects. It varies from 18.37% to 64.1%. We go on to see that even a keyword dictionary based model does not perform well on the *lammps* dataset compared to the other projects. Apart from nearly 2% class labels, that cast doubt on this data set being a valid set for experiment, very low F1-score and precision raises another doubt. We remove *lammps* data set from further comparison.

**Table 2: Words and categories for change message classification**

| Category | Associated words | Explanation |
|---|---|---|
| Corrective | bug, fix, wrong, error, fail, problem, patch | Processing failure |
| Feature addition | new, add, requirement, initial, create | Implementing a new requirement |
| Merge | merge | Merging new commits |
| Non Functional | doc | Requirements not dealing with implementations |
| Perfective | clean, better | Improving performance |
| Preventive | test, junit, coverage, asset | Testing for defects |

**Table 3: Keywords search evaluation metrics**

| Project | Precision | F1-score |
|---|---|---|
| abinit | 57.8 | 64.1 |
| libmesh | 38.9 | 51.8 |
| lammps | 10.6 | 18.4 |
| mdanalysis | 38.2 | 42.3 |

## 5.2 Baseline 2: TF Naive Bayes

Naive Bayes offers a simplistic model that can offer simple explanation in terms of probabilities. We explore Multinomial Naive Bayes classifier on term frequencies of the tokens in corpus. We use scikit-learn's MultinomialNB model and CountVectorizer for getting term frequencies. Results are reported in Table-4. We see that in general Multinomial NB is comparable at recall, but very poor in precision. False positives are very high in this case.

**Table 4: Multinomial Naive Bayes evaluation metrics**

| Project | Precision | F1-score |
|---|---|---|
| abinit | 50.9 | 57.1 |
| libmesh | 12.6 | 20.1 |
| lammps | 24.2 | 26.3 |
| mdanalysis | 8.9 | 14.3 |

## 5.3 Baseline 3: TF-IDF SVM

The third model we explore for baseline is term frequency - inverse document frequency weighted vectors fed into Support Vector Machine with linear kernel. TF-IDF is comprised of two distinct terms: tf and idf.

- **TF**: TF stands for term frequency. This means how many times a particular word appears in a document. For multiple documents with the same word it is defined as follows where $i$ is the $i^{th}$ word corresponding to $j^{th}$ document and $k$ is the number of documents containing the word.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

- **IDF**: IDF is defined as log of number of documents containing the word in entire data set. This accounts for how rare a word is across the data set. Here, $N$ represents number of documents in data set and $df$ is number of documents containing the word.

$$idf_i = \log(\frac{N}{df_i})$$

- **TF-IDF**: TF-IDF is a simple multiplication of TF and IDF. This indicates that those words are important that appear a lot in a very few documents.

$$tfidf_i = \frac{n_{i,j}}{\sum_k n_{i,j}} * \log(\frac{N}{df_i})$$

First of all the corpus is split into 80% training and 20% testing. Using the TfidfVectorizer of python's scikit-learn, the training set of each project is vectorized and top 25 features are sent to SVM for training. The evaluations metrics of this experiment is shown in Table-5.

**Table 5: TF-IDF SVM evaluation metrics**

| Project | Precision | F1-score |
|---|---|---|
| abinit | 62.7 | 64.4 |
| libmesh | 37.1 | 44.2 |
| mdanalysis | 35.6 | 43.0 |

Comparing TF-IDF SVM and Keyword based model we see tha TF-IDF performs equivalently well. As TF-IDF is an automated machine learning model. We select this to be our Baseline for further analysis. We see that TF-IDF SVM neither uses keywords nor emphasises what words are more related to a bug fix commits, we can comment that our baseline has very low comprehensibility.

## 5.4 Proposed Model 1: TF SVM

First model that we propose, tries to understand RQ1 i.e. is TF-IDF an ideal way to do bug-fix classification? Our intuition is that irrespective of any word to be present in as number of documents it should not matter to "bugginess" of the commit message. In fact, presence of higher number of buggy-words in a document, irrespective of the inverse document frequency, should indicate higher "bugginess". This experiment pertains to this notion. We feed in vector of text corpus using CountVectorizer (used for TF vectorization) from scikit-learn. Table-6 summarizes the evaluation metrics for this model.

**Table 6: TF SVM evaluation metrics**

| Project | Precision | F1-score |
|---|---|---|
| abinit | 63.0 | 64.1 |
| libmesh | 64.2 | 59.2 |
| mdanalysis | 47.5 | 49.8 |

This model is used to make comments on TF versus TF-IDF vectorization for our use case. However, we see that comprehensibility is still the same as the model has only slight variation from the baseline.

## 5.5 Proposed Model 2: LDA-SVM (BUGZY)

This model is based on topic modeling to increase comprehensibility of the automated machine learning classifier we attempt to make. Latent Dirichlet Allocation is used to select features (words) that matter the most in this classification. These features are then fed into SVM. We call this model *BUGZY*.

First of all the training corpus of the data set is used for initializing the CountVectorizer. The TF-vectorized data is then fed into scikit-learn's LatentDirichletAllocation with topic numbers as 2 and iterations as 10. This model is used to transform the feature vectors as a probability of the document belonging to these topics. Let's call these probabilities as *Topic_0* and *Topic_1*. Table-7 describes the top 10 words found in each topic according to the probabilities of occurrence in the topic. It is quite evident that *Topic_1* corresponds to a topic that describes bug fix words. Again, we take a look at *lammps* data set. The words that come up in buggy topic, is generally present in the non-buggy topic of other data sets. This is one other reason for us to exclude *lammps* from our final study.

After LDA transformation, we go through the following algorithm:

(1) Compute the probability of each word in the topic and store in a `word_topic_map`
(2) Go through each document.
    - Iterate over all the initial tokens present.
    - If a token matches the featured words, add the probability (from `word_topic_map`) of it being in a topic.
    - If the word is in both topics, sum the probabilities.
    - Along with these 25 columns, either 0 or otherwise computed in previous step, append the topic probability for the document as last two columns.
(3) Append the transformed 27 columns as new features to original data

Above 27 columns become our reduced feature vectors. These feature vectors are fed into a linear SVM (C=50, found through grid search). Finally the evaluation metrics are collected and stored as shown in Table-8.

**Table 8: BUGZY evaluation metrics**

| Project | Precision | F1-score |
|---|---|---|
| abinit | 57.8 | 64.8 |
| libmesh | 66.2 | 59.6 |
| mdanalysis | 51.7 | 52.6 |

# 6 RESEARCH QUESTIONS - CONCLUSIONS

**RQ1:***Is the occurrences of terms in a particular document in classifying bug-fix message or the discriminatory nature of term frequency weighted by logarithm of document frequency better?*

This research question essentially deals with the comparison of the importance to frequency of certain words versus mere presence of discriminative words. So, the comparison lies between term frequency (TF) and term frequency - inverse document frequency (TF-IDF). For this purpose we compare similar automated machine learning model with these two different types of vector i.e. TF+SVM and TF-IDF+SVM.

From the Figure-2 we can see that SVM trained on TF performs better on all data sets than the one trained on TF-IDF transformed vectors. We can conclude here that frequency of words denoting as buggy matters more than finding discriminative words. If we may ponder over the words that can be discriminative, it happens that most of the bug-fixing commit contains "fix". And, it is also common that "fix" does not appear more than once in a particular message. "Fix" becomes a non-discriminative word because of high IDF. However, presence of this word, even if the frequency is one, is quite potent in classifying a bug as bug-fix [12]. It can be said that TF matters more than TF-IDF. However, in future, we may experiment on whether it would be more prudent to get TF of "bug-fixing" words that may be a synonym of another one than just look at TF of an individual word.

**RQ2:** *Do we specify generic keywords for all software projects for change message classification? Can we create a general unsupervised keywords identification algorithm local to a software project?*

We see in the Keyword search model, a general dictionary is decided for all software projects. The pre-defined dictionary is not localized to any project. This can be verified from top 10 words shown for each topic after LDA transformation in Table-7. We also note that each project has a unique set of keywords that can be described *buggy* that are common to other projects e.g. "fix". However, some words can be different e.g. "ref", "typo", "changelog", etc. Hence, it is more prudent to discover these keywords localized to a software project. A globally defined dictionary may be used as a guidance system to identify which topic corresponds to a bug-fix topic. However, after this identification machine learner should work on discovered keywords rather the dictionary. To answer the question, we should not make an absolute dictionary for generic to all software projects. And we can use LDA as an unsupervised learning model to discover the keywords local or unique to a specific software project. As we see in Table-7, the discovered keywords have some common keywords with the dictionary described in Table-2. At the same time, there are many more keywords on *Topic_1* that may be "buggy" keywords, specific to that project.

**Table 7: Top 10 words probability of occurrence per topic**

| Project | Topic_0 | Topic_1 |
|---|---|---|
| abinit | merge, branch, develop, abinit, trunk, remote, gitlab, tracking, release, org | fix, test, file, update, add, new, ref, change, typo, error |
| libmesh | merge, request, pull, libmesh, mesh, update, test, make, new, example | fix, added, add, use, element, change, file, function, fixed, class |
| lammps | svn, lammps, git, icms, trunk, temple, edu, sync, library, added | fix, merge, request, pull, kokkos, user, pair, update, dpd, add |
| mdanalysis | merge, doc, mdanalysis, develop, pull, request, branch, code, atomgroup, issue | test, added, fixed, fix, issue, updated, file, analysis, changelog, new |

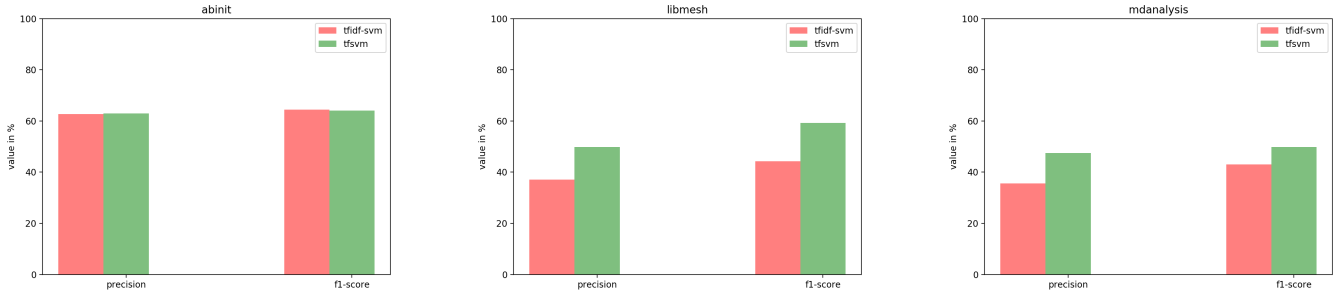


**Figure 2: TF-IDF SVM vs TF SVM - compared across all projects**

**RQ3:** *Does a supervised classification algorithm based on auto-detected keywords work better than baseline classifiers? Does it add to comprehensibility of the model?*

To answer this question we look at Figure-3, that compares the evaluation metrics of all the proposed and baseline models. We find that proposed model number 2 i.e. BUGZY performs equivalent or better in all the projects. There is a slight dip in precision compared to baseline model in case of *abinit* project of nearly 3%. However, looking at F1 measure we can say that recall is certainly the highest in this case. Also, in all other projects BUGZY model is better in both precision and F1-measure. To answer the question, if SVM fed with LDA transformed features work better, we can certainly say, yes it does. Moreover, we see that metrics are fairly reasonable across the project and does not fluctuate as much as *keyword search* model. Hence, we can confidently say that BUGZY is stable.

To answer for comprehensibility part, we need to understand that apart from *Keyword search* model, no other model offers comprehensibility other than BUGZY. In fact, bugzy offers a better comprehensibility than *Keyword search* model. It is such because, BUGZY identifies keywords that are local to the software project. Not only does BUGZY capture localized keywords, but it also captures keywords that are similar to corrective category keywords used in *Keyword search* model. We can conclusively say that, BUGZY offers more comprehensibility than any other model.

## 7 LIMITAITONS OF THE STUDY

As it was evident from several strong reasons in the paper, *lammps* is a not good enough data set for this study. It is quite evident that the data set is not large enough for a reasonable study. Two projects had the number of commits also around 10000. This number may seem large enough. But the number of bug fixing commits were only around 10 percent. It may or may not be enough to train/validate BUGZY. Using a bigger data set, more projects and well maintained projects may help capture more insights to bug-fixing commits than this study. All our data sets from computing softwares. This could have led to a sampling bias.

It is quite evident from this study that LDA helps in comprehensibility of the model. However, the complexity of the model increases much more. This raises doubt if model is within reasonable limits. However, we see that the model is fairly stable across the projects. We can, thus, say that making a model little bit more complex, did increase the cost of other resources like time to train the model and computing requirements, which may grow as number of projects or size increase.

The results reported are generated over 50 iterations of sampling with replacement. This is done to reduce the order bias. As data sets distributed in training and testing is completely random, that could have incurred order bias. Agrawal et al. [1] discussed that LDA, inherently, has an order bias. The algorithm automatically discovers words that are characteristic to a topic. These words could differ in different samples of data. However, keeping low number of maximum features for LDA training and running this experiment multiple times helps reduce that bias.

Hyper parameter optimization for LDA, along with max features could lead to better results. We have not reported results from other classifiers in the report. It could be possible that other classifiers such as Random Forest may work well or better than SVM. Even after optimized parameter of C value in SVM through GridSearch,

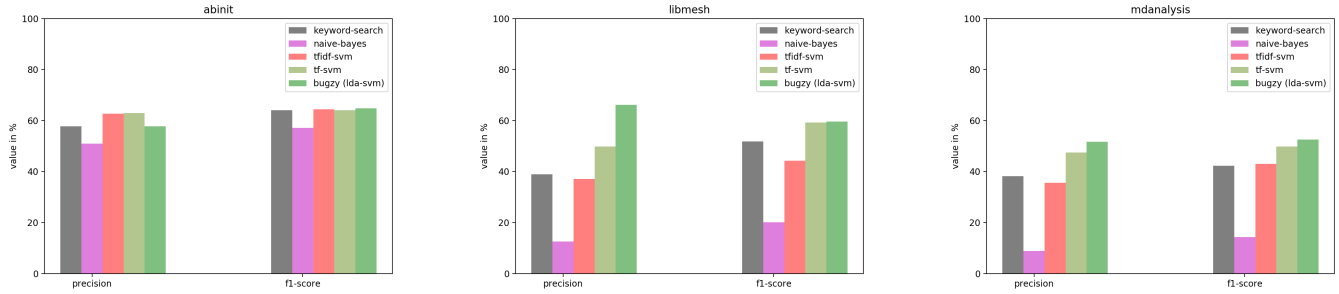


**Figure 3: Proposed models vs baseline models - compared across all projects**

we could still have different parameters or better model when using larger or different data set.

In the algorithm for BUGZY, we first find the 10 words with highest probability per topic. The probability value of these words corresponding a topic is used to generate features when a word in the tokens of a commit message matches it. The match here is exact match. This match could be misleading as a typographical error would not account for it or even a synonym. The match here could have been in terms of natural language, slopped to adjust for spelling variation and synonyms could be used keeping software engineering domain knowledge. If this match is accommodated in the algorithm TF of a buggy word synonym will go up for a "bug-fixing" commit message.

## 8 FUTURE WORK

First and foremost, the same experiment should be run on larger corpus, more projects and better maintained project. The validation should not be based on just the computational software projects but other types of software engineering project. Therefore, the future work involves validating and running this experiment over large number of software projects. In case of validation, we should also explore F-2 instead of F-1. This is important because our use case needs a very high recall.

BUGZY's performance in the studied projects is acceptable. However, the amount of improvement that we notice is a small change 2-3% increase over TF-SVM. This raises the question to look back at the algorithm, even after dimension reduction and using more profound features, we do not get a drastic improvement. A close look into the algorithm tells us that we have introduced many magic parameters such as for LDA input dimensions, number of iterations and number of topics. These parameters are on top of the usual ones in SVM of penalty function and maximum number of features for TF vectorization. These hyper parameters can be optimized through an optimization technique such as Differential Evolution. Agrawal et al [1] mentions finding optimal number of topics for LDA using LDADE, a combination of LDA and Differntial Evolution.

Our current algorithm has a heuristic around incorporation LDA weights into SVM. There could be other ways of incorporating these weights into the classifier. Exploration of other algorithms is one such task. In addition, we could explore ensemble methods. One such method was tried and tested during this project. We called it E-BUGZY. E-BUGZY is an ensemble of two classifiers, BUGZY and TF-SVM. They are trained on different number of features. Because of low comprehensibility of the model the results were not reported in this paper. In future, we can explore models based on ensemble methods.

Injecting Software Engineering domain knowledge into LDA and feature generation may perform better. As described in limitations section, feature generation using matching words can make use of SE domain knowledge.

There are other word embeddings models that can be used. As this is a semantic natural language sentences, it might help to use word2vec or GloVe embedding instead of TF or TF-IDF vectorization. Word2vec is known to capture semantic and syntactic relationships at low computational cost [16]. GloVe is unsupervised learning algorithm to obtain word vector representations [18]. However, this may lead to enhanced way to understand comprehensibility of the model. If these models improve our evaluation metrics, further questions can be raised on the comprehensibility and reasonability of these models. We may explore usage of such vectorization techniques for natural language.

As BUGZY is localized to a software project, and is yet a general model. We can explore if generalization of this project is feasible. It is a common practice in software engineering data mining that the model is trained on earlier releases of the software project and tested on more recent version. Can we train a model on older versions of project and predict for current version? However, a big task in change message classification is labeling of messages. Kim et al [12] suggests understanding bug tracking system and use keyword search heuristic to classify. However, recommendation is still made to get these change classifications manually verified. Software engineering domain knowledge is essential, but sample change message classification for each project is costly. Can we omit human in the loop and understand from release notes for validity of this model? Can a fully automated change message classification be built?

## REFERENCES

[1] Amritanshu Agrawal, Wei Fu, and Tim Menzies. 2016. What is Wrong with Topic Modeling? (and How to Fix it Using Search-based SE). *CoRR* abs/1608.08176 (2016). arXiv:1608.08176 http://arxiv.org/abs/1608.08176

[2] Amritanshu Agrawal, Huy Tu, and Tim Menzies. 2018. Can You Explain That, Better? Comprehensible Text Analytics for SE Applications. *CoRR* abs/1804.10657 (2018). arXiv:1804.10657 http://arxiv.org/abs/1804.10657

[3] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is It a Bug or an Enhancement?: A Text-based Approach to Classify Change Requests. In *Proceedings of the 2008 Conference of the Center for*

*Advanced Studies on Collaborative Research: Meeting of Minds (CASCON '08)*. ACM, New York, NY, USA, Article 23, 15 pages. https://doi.org/10.1145/1463788.1463819

[4] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3 (March 2003), 993–1022. http://dl.acm.org/citation.cfm?id=944919.944937

[5] Ying Fu, Meng Yan, Xiaohong Zhang, Ling Xu, Dan Yang, and Jeffrey D. Kymer. 2015. Automated classification of software change messages by semi-supervised Latent Dirichlet Allocation. *Information and Software Technology* 57 (2015), 369–377.

[6] Scott Grant, James R. Cordy, and David B. Skillicorn. 2012. Using Topic Models to Support Software Maintenance. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. IEEE Computer Society, Washington, DC, USA, 403–408. https://doi.org/10.1109/CSMR.2012.51

[7] Ahmed E. Hassan. 2008. Automated Classification of Change Messages in Open Source Projects. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC '08)*. ACM, New York, NY, USA, 837–841. https://doi.org/10.1145/1363686.1363876

[8] Lile P. Hattori and Michele Lanza. 2008. On the Nature of Commits. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. IEEE Press, Piscataway, NJ, USA, III–63–III–71. https://doi.org/10.1109/ASEW.2008.4686322

[9] Abram Hindle, Daniel M. German, and Ric Holt. 2008. What Do Large Commits Tell Us?: A Taxonomical Study of Large Commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR '08)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/1370750.1370773

[10] A. Hindle, M. W. Godfrey, and R. C. Holt. 2009. In *What's hot and what's not: Windowed developer topic analysis*. IEEE, 339–348.

[11] Thomas Hofmann. 2001. Unsupervised Learning by Probabilistic Latent Semantic Analysis. *Mach. Learn.* 42, 1-2 (Jan. 2001), 177–196. https://doi.org/10.1023/A:1007617005950

[12] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Trans. Softw. Eng.* 34, 2 (March 2008), 181–196. https://doi.org/10.1109/TSE.2007.70773

[13] George Mathew, Amritanshu Agrawal, and Tim Menzies. 2016. Trends in Topics at SE Conferences (1993-2013). *CoRR* abs/1608.08100 (2016). arXiv:1608.08100 http://arxiv.org/abs/1608.08100

[14] Andreas Mauczka, Markus Huber, Christian Schanes, Wolfgang Schramm, Mario Bernhart, and Thomas Grechenig. 2012. Tracing Your Maintenance Work — a Cross-project Validation of an Automated Classification Dictionary for Commit Messages. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE'12)*. Springer-Verlag, Berlin, Heidelberg, 301–315. https://doi.org/10.1007/978-3-642-28872-2_21

[15] Andrew McCallum and Kamal Nigam. 1998. A Comparison of Event Models for Naive Bayes Text Classification. In *Learning for Text Categorization: Papers from the 1998 AAAI Workshop*. 41–48. http://www.kamalnigam.com/papers/multinomial-aaaiws98.pdf

[16] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). arXiv:1301.3781 http://arxiv.org/abs/1301.3781

[17] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00) (ICSM '00)*. IEEE Computer Society, Washington, DC, USA, 120–. http://dl.acm.org/citation.cfm?id=850948.853410

[18] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global Vectors for Word Representation.. In *EMNLP*, Vol. 14. 1532–1543.

[19] E. Burton Swanson. 1976. The Dimensions of Maintenance. In *Proceedings of the 2Nd International Conference on Software Engineering (ICSE '76)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 492–497. http://dl.acm.org/citation.cfm?id=800253.807723

[20] Stephen W. Thomas. 2011. Mining Software Repositories Using Topic Models. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 1138–1139. https://doi.org/10.1145/1985793.1986020