NOVA IMS Information Management School

MDSAA

USING SELF-ORGANIZING MAPS TO TRIAGE SOFTWARE BUG REPORTS

Studying the Effect of Using Different Text Vectorization Methods

Farina Garrido Pontejos

Dissertation

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

NOVA Information Management School

Instituto Superior de Estatística e Gestão de Informação

Universidade Nova de Lisboa

USING SELF-ORGANIZING MAPS TO TRIAGE SOFTWARE BUG REPORTS

by

Farina Garrido Pontejos

Dissertation presented as partial requirement for obtaining the Master's degree in Advanced Analytics, with a Specialization in Business Analytics

Supervisor: Professor Fernando José Ferreira Lucas Bação

July 2023

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledge the Rules of Conduct and Code of Honor from the NOVA Information Management School.

Farina Pontejos

Lisbon, Portugal
14 July 2023

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Professor Fernando Bação, for providing guidance and advice throughout the process of writing this thesis. I have learned a lot from this experience.

I would also like to thank my family and friends, whose endless encouragement proved invaluable to the success of this endeavor.

Abstract

Bug Report Triage involves the assignment of software developers to fix bugs for which their experience is a good match. Given the numbers of bugs being reported to software projects, several methods have been proposed to address this task in an automated way. Approaches based on Information Retrieval make use of the textual components of the Bug Reports, however most previous studies have been concerned only with evaluating the performance of the models being used, with very few also considering the question of how best to represent the textual features. Self-Organizing Maps have long been used in Information Retrieval, however very few studies have considered it for use in the field of Bug Report Triage. This thesis aims to bridge these gaps by using Self-Organizing Maps for triaging Bug Reports, and evaluating the effect of using different text vectorization methods to represent the text. Using ten Bug Report corpora and applying six different preprocessing treatments to each, we tested five different vectorization methods: TFIDF, Word2Vec, Doc2Vec, BERT, and SBERT. We found that there was a statistically significant difference in the performance of the five vectorization methods, with SBERT obtaining better $Accuracy@1$ scores than the rest. We also tested other initialization parameters for training the SOM and found that the results were largely consistent across the different conditions. In addition, we developed SOMBR, an interactive dashboard based on the SOM, which allows users to query the Bug Report corpus to find the query's Best Matching Unit, enabling the user to inspect the developers that have fixed bugs that are semantically similar. Based on these experiments, we believe that the SOM is a promising tool for semi-automated Bug Report triage, however further studies need to be conducted to determine how the evaluation scores can be improved further.

Keywords

Self-Organizing Maps; Information Retrieval; Bug Report Triage; NLP; Text Mining; Visualization

Contents

List of Figures xiii

List of Tables xiv

List of Acronyms xvii

1. Introduction 1

2. Literature Review 5
- 2.1. State of the Practice in Automated Bug Triage . . . 5
- 2.2. State of the Art in Automated Bug Triage . . . 5
- 2.3. Text Representation . . . 8
- 2.4. Use of SOM in Information Retrieval . . . 11
- 2.5. Summary and Research Gap . . . 13

3. Methodology 15
- 3.1. Proposed Approach and Experimental Setup . . . 15
- 3.2. Description of the Data . . . 15
- 3.3. Characteristics of the Corpora . . . 20
- 3.4. Preprocessing . . . 21
- 3.5. Text Vectorization . . . 22
  - 3.5.1. TF-IDF . . . 22
  - 3.5.2. BERT and SBERT . . . 22
  - 3.5.3. Word2Vec and Doc2Vec . . . 24
- 3.6. Training the SOM . . . 25
  - 3.6.1. Map Size . . . 25
  - 3.6.2. Decay . . . 26
- 3.7. Evaluation . . . 27
  - 3.7.1. Precision . . . 27
  - 3.7.2. Recall . . . 27
  - 3.7.3. Accuracy . . . 27
  - 3.7.4. Quantization Error . . . 28
  - 3.7.5. Topographic Error . . . 28
- 3.8. Visualization . . . 28

4. Results and Discussion 31
- 4.1. Effect of Text Vectorization . . . 31
  - 4.1.1. Statistical Evaluation . . . 32
  - 4.1.2. Accuracy@1 . . . 33

4.1.3. Accuracy@BMU 33
4.1.4. Precision 34
4.1.5. Summary of BR Triage Evaluation 34
4.1.6. Quantization Error and Topographic Error 37
4.2. Effect of Different Map Initialization Conditions 40
4.3. SOMBR: A Visual Approach to BR Triage Using SOM 43
4.3.1. Interaction Elements 45
4.3.2. Map Grid Interactions 48

5. Conclusions **51**

6. Limitations and Future Work **53**

Bibliography **55**

Appendix A **61**

Appendix B **67**

Annex A **83**

List of Figures

1.1. Issue Triage Flow (Kubernetes, 2022) . . . 1

3.1. Overview of Methodology . . . 16
3.2. Database schema of SEOSS Dataset (Rath & Mäder, 2019a) . . . 17
3.3. Screenshot of Bug Report . . . 19
3.4. Data Filtering . . . 19
3.5. Characteristics of the selected corpora: Number of assignees, mean BR per assignee, and number of documents in the corpus . . . 20
3.6. Characteristics of the selected corpora: mean and median of number of BRs assigned per user . . . 20
3.7. Experimental Setup . . . 21
3.8. Dimensions of BERT Tensors . . . 24
3.9. Dimensions of SBERT Vectors . . . 24
3.10. Default Learning Rate and decay . . . 26

4.1. Count of Ranks: Accuracy@1, Accuracy@BMU, and Precision . . . 31
4.2. Characteristics of the selected corpora: Number of words in the corpus, baseline processing vs camelcase split . . . 34
4.3. Parallel Coordinates Plots, Accuracy and Precision . . . 36
4.4. Count of Ranks: Topographic and Quantization Error . . . 38
4.5. Parallel Coordinates Plots, Quantization and Topographic Error . . . 39
4.6. Rank Values Using Different SOM Initializations . . . 41
4.7. Significance of Pairwise Differences . . . 42
4.8. Screenshot of dashboard . . . 43
4.9. SOM Visualization . . . 44
4.10. Interactive Inputs . . . 45
4.11. Query Results . . . 46
4.12. User Selection . . . 46
4.13. Querying Results for “hive”, zoomed in . . . 47
4.14. Dashboard interactions: detail view . . . 48
4.15. Detail view tabs . . . 48
4.16. Dashboard interactions: graph view . . . 49

A1. Number of Bug Reports Created per Month . . . 65
A2. Boxplots of Number of Bug Reports Created per Month . . . 65

B1. Violin and Scatter Plots of Results: Accuracy@1 . . . 67
B2. Violin and Scatter Plots of Results: Accuracy@BMU . . . 68
B3. Violin and Scatter Plots of Results: Precision . . . 68

B4. Violin and Scatter Plots of Results: Topographic Error 69
B5. Violin and Scatter Plots of Results: Quantization Error 69
B6. Pair-wise p-values of SOM Initialization Experiments: Varying Random Seeds 80
B7. Pair-wise p-values of SOM Initialization Experiments: Varying Map Parameters 81

List of Tables

3.1. Projects included in experiments . . . 18
3.2. Naming conventions and subsequent preprocessing treatment . . . 22
3.3. Examples of Preprocessing Results . . . 23
3.4. Doc2Vec and Word2Vec training parameters . . . 25
3.5. MiniSom training parameters . . . 25

4.1. Results of Friedman Rank Sum Test: Accuracy and Precision . . . 32
4.2. Mean Values of Ranks . . . 32
4.3. Mean Values of Scores . . . 33
4.4. p-values of the pairwise comparisons using the Nemenyi-Wilcoxon-Wilcox all-pairs test for a two-way balanced complete block design . . . 35
4.5. Results of Friedman Rank Sum Test: TE and QE . . . 37
4.6. p-values of the pairwise comparisons using the Nemenyi-Wilcoxon-Wilcox all-pairs test for a two-way balanced complete block design . . . 37
4.7. Mean Values of Ranks: TE and QE . . . 38
4.8. Mean Values of Scores: TE and QE . . . 38
4.9. SOM Initialization . . . 40
4.10. SOM Training Epochs . . . 40

A1. Description of attributes: `issue` table . . . 61
A2. Description of attributes: `issue_link` table . . . 61
A3. Entry for issue SPARK-19748 . . . 62
A4. Corpus sizes before and after data filtering . . . 66

B1. Scores for Accuracy @ 1 . . . 70
B2. Scores for Accuracy @ BMU . . . 72
B3. Scores for Precision . . . 74
B4. Scores for Quantization Error . . . 76
B5. Scores for Topographic Error . . . 78

List of Acronyms

BERT	Bidirectional Encoder Representations from Transformers
BMU	Best Matching Unit
BOW	Bag of Words
BR	Bug Report
CBOW	Continuous Bag of Words
CNN	Convolutional Neural Network
DIANA	Divisive Analysis clustering
ELMo	Embeddings from Language Models
GloVe	Global Vectors
IR	Information Retrieval
LR	Learning Rate
LSI	Latent Semantic Indexing
LSTM	Long short-term memory
MAP	Mean Average Precision
MiniL12	`MiniLM-L12-H384-uncased`
MiniL6	`MiniLM-L6-H384-uncased`
ML	Machine Learning
NLP	Natural Language Processing
OOV	out of vocabulary
OSS	Open Source Software
QE	Quantization Error
SBERT	Sentence-BERT
SOM	Self-Organizing Map
SVM	Support Vector Machine
TE	Topographic Error
TF-IDF	Term Frequency - Inverse Document Frequency
VSM	Vector Space Model

1. Introduction

Bug Reports (BRs), according to Davies and Roper (2014) are "the primary means by which users of a system are able to communicate a problem to the developers". In the same paper, the authors describe the typical contents of BRs, which often consists of a combination of natural language such as descriptions of observed and expected behavior, as well as constructed language such as stack traces and code examples.

Before they can be addressed, BRs first need to be triaged by the appropriate team member responsible. The developers of the Kubernetes project (Kubernetes, 2022) organize their triaging process into the following five steps summarized in Figure 1.1.

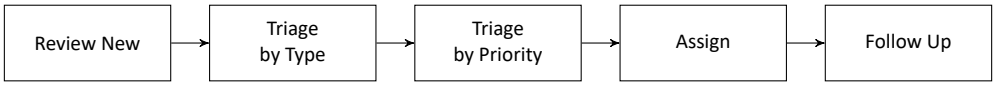


Figure 1.1.: Issue Triage Flow (Kubernetes, 2022)

The process begins by first looking at issues that have not been previously triaged. The triager then has to determine whether the report should be considered a bug, or whether it is better addressed as a support request. Once determined to be a bug, the triager needs to determine the level of priority, and finally assign it, either to themselves, or to the appropriate team to fix. Assigning a bug to a specific developer within the team requires knowledge of developers' expertise and suitability for the task, as well as their workload and availability. Finally, the triager may also follow up with the assignees if no further activity has been made within 30 days.

Triaging BRs can be a significant burden in software development, negatively impacting the efficiency and costs related to the process, as well as affecting end-users' satisfaction. A study by Zaman et al. (2011) found that security-related bugs in the Firefox project took on average 956 hours to be triaged, defined as the amount of time it took between a bug being reported and being assigned to a developer. Rajbhandari et al. (2022) studied how long it took for bugs to move from being reported to being triaged in the Chromium project and found that on average, it took 1 414 hours for security-related bugs. Performance-related bugs took even longer in Firefox (3 483) and Chromium on average took 1 227 hours. Zaman et al. (2011) notes that ideally bug triage times are kept small; security bugs have risk implications, and performance bugs impact end-users' experience. It is clear that improvements in the BR triage process will produce significant gains in cost and time reduction in software development projects.

Bettenburg et al. (2008) conducted a survey among both bug reporters and the developers involved in fixing bugs across several open-source projects. They wanted to determine what kinds of information developers considered to be good qualities to have in a BR. In a subsequent paper (Just et al., 2008), they performed a qualitative analysis on the comments provided by the respondents to further determine "how bug reporting systems could be improved". One of the recommendations identified in this paper was to "[p]rovide a powerful, yet simple and easy-to-use feature to search bug reports".

Different approaches to automatic bug triage have been proposed. Nagwani and Suri (2023) conducted a literature review analyzing automated approaches to software bug triage. They identified Information

Retrieval (IR)-based and Machine Learning (ML)-based techniques as being the largest categories, comprising 29% and 28% respectively of the identified approaches. Several of these approaches will be discussed in more detail in the next chapter.

IR involves obtaining information from large collections of text data (Manning et al., 2008). Sanderson and Croft (2012) detail the history of techniques used in IR, from querying manually catalogued libraries, to the use of language models to match user queries to the relevant documents. A central question in this field is how to vectorize documents, or how to represent this unstructured text data in a way that computer algorithms can process (Yan, 2009). Nagwani and Suri (2023) also analyzed how bug information is typically represented in automated bug triage applications, and found that traditionally, the Vector Space Model (VSM) has been the most used representation, created with the frequency vectors of the terms in the BRs. Word embeddings such as Word2Vec (Mikolov et al., 2013), Global Vectors (GloVe) (Pennington et al., 2014) and Embeddings from Language Models (ELMo) (Peters et al., 2018) have also been used in recent, but fewer works, categorized under Deep Learning approaches.

Naseem et al. (2021) reviewed different word representation models available, and some of these will be discussed further in the following chapter.

Self-Organizing Maps (SOM) have been used extensively in the field of IR. One of the earliest applications of SOM in this field includes WEBSOM (Kaski et al., 1998), in which the words of the corpus are organized into a word category map using SOM, from which the document vectors can then be encoded. SOM have also been previously used on BR corpora (Ahmed & Ghazali, 2016; do Rego et al., 2008), however neither work has compared the results of using different text vectorization methods.

Zaidi et al. (2020) compared the results when using different vector representations of BRs, however they used Convolutional Neural Networks (CNNs), not SOM, to assign bug fixers. These, as well as other relevant works will also be discussed further.

The main objective of this research is to investigate whether a SOM trained on a corpus of BRs can be used as an IR system for BR triage. To that end, these are the research questions that this thesis aims to answer:

RQ 1: Is a SOM trained on a corpus of BRs able to identify the correct bug fixer responsible?

RQ 2: Does vectorizing text using sentence embedding methods improve the performance of the SOM compared to using term frequency-based representations or word embedding?

RQ 3: Do the initialization parameters of the SOM, such as its learning rate and training length, affect its performance in triaging BR?

Additionally, this work aims to develop a dashboard visualization tool that allows users to interact with and explore a large corpus of documents such as a database of BRs. The purpose of this dashboard is to provide a more intuitive way for users to query a corpus and retrieve semantically similar documents. BR triagers might find it useful to assign new BR to users that have fixed similar issues before, and BR assignees might find it helpful to see how previous similar bugs were addressed.

The remainder of this work will be organized as follows: Chapter 2 will provide a review of the related literature; Chapter 3 will explain the methodology used in the study; Chapters 4, 5, and 6 will discuss the results, conclusions, and limitations, respectively, of this research.

2. Literature Review

2.1. State of the Practice in Automated Bug Triage

In January 2023 alone, Microsoft VSCode, a popular Open Source Software (OSS) project with a repository hosted on GitHub, received 1766 new issues on their issue tracker [1]. Of these, 289 were identified as bugs. 194 of these have been subsequently closed, and 95 were still open as of the time of writing[2].

To help streamline their triaging process, this project makes use of GitHub Actions (GitHub, Inc., 2023a), which are automated tools that project developers can use to help manage their activities on GitHub. These Actions can be configured to automatically run each time a particular event occurs, for instance when an issue is reported to a project. VSCode's Issue Triage GitHub Action (Microsoft Corporation, 2022b) runs automatically whenever a new issue is submitted to its GitHub repository. It tries to automatically assign an issue to the correct feature area and assignee (Microsoft Corporation, 2022a) using a pre-trained Bidirectional Encoder Representations from Transformers (BERT) model finetuned on the VSCode issues database. The issues that the bot is unable to label within a threshold confidence level is then manually triaged by a team member. With 116 stars[3] at the time of writing, this tool is one of the most popular results when searching for projects related to "issue triage" on the GitHub platform.

For the same period, the AWS Cloud Development Kit GitHub repository[4] received 168 new issues. Of these, 80 were determined to be bugs, of which 37 remain open, and 43 have been closed. This project uses the AWS Issue Triage Manager (Amazon Web Services, Inc, 2022), another GitHub Action that assigns a feature area and responsible user to newly reported issues. For each new issue reported, it compares the contents of the issue's `title` and `body` fields to a list of keywords configured to each feature area. Each occurrence of an area's keyword increases that corresponding area's score, and the highest-scoring area is determined to be the winner for that issue. The list of assignees to be assigned have been configured manually for each area.

2.2. State of the Art in Automated Bug Triage

Lin et al. (2016) considered three textual correlation relationships in ranking the similarity of a bug report to previously submitted bug reports: cluster-based correlation, document-wide similarity correlation, and word-based semantic correlation.

The word-based semantic correlation is calculated by constructing a word vector model by training Word2Vec on the corpus of all bug reports filed before the bug report being considered for classification. Each word in the new bug report is transformed into a word vector, and the cosine similarity is calculated between it and the candidate documents. Documents are preprocessed by performing stemming and stopword removal.

[1] https://github.com/microsoft/vscode/issues
[2] March 23, 2023
[3] GitHub "stars" are a way for users to bookmark project repositories and can be used as an indicator for a repository's popularity. (GitHub, Inc., 2023b)
[4] https://github.com/aws/aws-cdk/

To calculate the cluster-based correlation, the Cluster-Feature-Centroid weighing scheme introduced by Guan et al. (2009) is used. It is similar to Term Frequency - Inverse Document Frequency (TF-IDF), with the additional consideration of intra-cluster and inter-cluster correlation. The weight of a term is calculated by multiplying the TF-IDF weight with the CFC weight, which corresponds to the importance of the term on a given cluster. Document-based similarity is calculated using BM25 to calculate the term weights. BM25 ranks a document's similarity based on the appearance of the query term within it Amati (2009).

The mentioned correlation features are calculated, on which a Support Vector Machine (SVM) classifier is trained. Datasets of bug reports from three OSS projects were used in the study, and experiments were performed using all the correlation features, different combinations, and each set alone, and compared with the baseline TF-IDF weighing scheme. The top recall metric was achieved by using all three correlation features.

Yang et al. (2016) proposed calculating three similarity scores to determine bug similarity. The first two scores are derived from the Title and Description fields. After combining their contents, stopwords and non-alphanumeric characters are removed, and a stemming algorithm is used. The first score is derived from the TF-IDF vectors, and given by the cosine similarity calculated between the new bug and the rest of the corpus. The second score is derived by using the Word2Vec Skip-gram model to derive the word embeddings. The document embedding is calculated as the averaged word embeddings for each document. Finally, the third score is determined by calculating the similarity scores considering only the Product and Component fields. These three scores are aggregated into a final score, giving the similarity between a new BR and each of the documents in the corpus.

Jonsson et al. (2016) studied using a stacked ensemble machine learning algorithm in assigning developers to bugs. They used both textual and non-textual features in their experiments. To avoid high-dimensionality, they limited the text, derived from the combination of the Title and Description fields, to only the top 100 terms as identified by their TF-IDF values. The non-textual features consist of nominal features describing the submitter, the product, and the priority. The authors performed several experiments, the first of which was to test which base classifier performed the best. They found that the sequential minimal optimization algorithm for training a support vector classifier (Platt, 1998) was able to achieve the highest accuracy score among the algorithms tested.

Having obtained the performance of several classifiers, they created three groups: the best-performing, the worst-performing, and a group composed of the the best-performing classifiers from each of the different algorithm families they tested. These three groups correspond to the three different ensembles tested. The authors found that the "best" and the "selected" groups performed similarly to each other, and scored better than the "worst" group. The "best" group achieved a higher accuracy than the best base classifier on four out of the five datasets tested, while the "selected" group out-performed across all five. The authors further experimented on how much training data to use considering the age of the historical bug reports and found that prediction accuracy became worse with more older data included. Additionally, they found that at least 2000 bug reports should be used when training a stacked ensemble classifier.

Zhang et al. (2016) had two objectives in their study: to be able to predict the severity of a new BR, as well as recommend a developer to fix it. Their approach was to first determine the top K-nearest

neighbors, and then analyze these to make their recommendation. Preprocessing involved stopword removal and stemming. The authors also split variable names into their component words, giving the example of "fillColor" being represented as "fill Color". They used Latent Dirichlet Allocation to model the topics related to each document, in which each document is encoded by the probabilities of each topic being related to it. To compare a new BR to historical ones, the authors modified the REP similarity measure proposed by Tian et al. (2012) by adding the topic feature.

Once the top K neighbors have been identified, the authors predict the new BR severity considering the probability that it shares the same severity as each neighbor, weighted by their similarity. To predict the bug fixer to assign, they construct a social network based on users' commenting activities on the similar bug reports. They also consider the users' general experience in fixing bugs. Using this methodology on five OSS software bug repositories, they were able to achieve improved scores compared to existing approaches.

Hu et al. (2018) calculated the aggregate of four similarity scores to recommend the most similar bugs to a given query BR. They calculate the similarity scores between the query and the rest of the corpus given their TF-IDF vectors, the Word2Vec embeddings, and the Doc2Vec embeddings. The sum of these scores is then multiplied by the similarity score given by the `Product` and `Component` fields. The authors found that they were able to improve on the performance of Yang et al. (2016) using this combination of features.

Zaidi et al. (2020) used CNNs to recommend appropriate bug fixers. Their goal however, was not to test whether using a CNN was the best approach to solve the problem, but rather to determine whether using different word embeddings to represent the text in BRs made a difference in the result when used with this algorithm. They tested the performance of three word embeddings: Word2Vec (Mikolov et al., 2013), GloVe (Pennington et al., 2014), and ELMo (Peters et al., 2018), and found that ELMo, a context-sensitive embedding, performed the best out of the three. They also tested whether class imbalance degrade the results and found that increasing the number of available training samples for each class, in this case the assigned bug fixer, resulted in an improvement in performance.

To evaluate the performance of the word embeddings in the model, the authors used the top K accuracy measure given by Equation 2.1. Here, they used values of k from 1 to 10, N is the total number of BRs, rec_i@k is the list of recommended k developers, and dev_i is the true bug fixer assigned to the BR. The function I returns 1 if dev_i is in the list rec_i@k and 0 otherwise.

$$\textit{Top-k accuracy} = \frac{\sum_{i=1}^{N} I(rec_i@k, dev_i)}{|N|} \tag{2.1}$$

Sajedi-Badashian and Stroulia (2020) performed a systematic review of bug assignment research. They wanted to find out how the "best assignee" should be defined and recommended that the "best assignee" can be considered from any of the following users:

1. "`AUTHOR`: The author of a commit referencing a bug number as resolved. [...] (p.10)"
2. "`COAUTHOR`: A developer (other than the original author of the committed code), who actually commits the code and references a bug as resolved. [...] (p.10)"

3. "ADMIN_CLOSER: The developer who closes the bug. [...] (p.11)"

4. "DRAFTED_ASSIGNEE: The developer tagged as 'assignee' when the bug is closed. [...] (p.11)"

An additional objective of the authors was to determine who should make up the pool of possible assignees, which they termed the "developer community". They found that "all the project members" was the best definition, also being the most realistic. In their experiments, they also tested the results of using using only the subset of developers that have fixed a minimum number of bugs. They found that increasing the threshold also increased the Mean Average Precision (MAP) with a -0.98 correlation. The authors also investigated the different evaluation metrics used for this task. They recommend the MAP and gave the criteria they used for determining this.

Tahir et al. (2021) proposed the addition of sentiment or emotion analysis as an additional feature to use in BR triage. Using the BR summary field, they used Calefato et al. (2017) to determine the BR's emotion. Preprocessing included the removal of symbols and numeric characters, stopwords, and lemmatization. They used Word2Vec to vectorize the BR, and Long short-term memory (LSTM) to determine the assignee. The authors found that the addition of emotion did not impact their results.

2.3. Text Representation

In reviewing the literature around automatic bug triage, Nagwani and Suri (2023) observed that "The most commonly used representations in the existing studies are vector representation [...], and matrix representation [...]." (p.7). In both vector and matrix representations, the documents are represented by the frequencies of the occurrences of the terms present in the corpus. One of the limitations of this Bag of Words (BOW) approach however, is that the context within the documents are not considered. The order of the terms in the documents are disregarded, and the presence of polysemy or synonymy are similarly ignored (Yan, 2009).

In Naseem et al. (2021), the authors divide text representation methods into two main groups: those based on the classical model, and those in which the representations are learned by ML algorithms.

According to Manning et al. (2008), "[t]he representation of a set of documents as vectors in a common vector space is known as the *vector space model* and is fundamental to a host of information retrieval operations ranging from scoring documents on a query, document classification and document clustering." Jurafsky and Martin (2023) further explains that in the VSM, "a document is represented as a count vector".

The VSM represents text by considering the frequencies of the words in the corpus. In this model, all the terms in the corpus are put into a "Bag of Words". Each document is represented by a vector in which each element corresponds to a word in the corpus. In the boolean case, a value of 1 means that the word is present in the document, and 0 otherwise.

TF-IDF is a variation on the boolean BOW model that applies a weighting schema to capture the relative importance of the terms in the documents. The term frequencies (TF) of each word in the document are calculated, and offset by the Inverse Document Frequency (IDF) or the frequency of each word in the

whole corpus. Ramos (2003) explains the mathematical framework to calculate the document vectors using TF-IDF:

> "Given a document collection D, a word w, and an individual document $d \in D$, we calculate [Equation 2.2] where $f_{w,d}$ equals the number of times w appears in d, $|D|$ is the size of the corpus, and $f_{w,D}$ equals the number of documents in which w appears in D ..." (p.2)

$$\begin{aligned} TF &= f_{w,d} \\ IDF &= \log\left(\frac{|D|}{f_{w,D}}\right) \\ w_d &= TF \times IDF \end{aligned} \tag{2.2}$$

One of the limitations of the BOW model is its inability to capture the semantics or the meaning of the terms in the documents. For example, words that exhibit homonymy, or multiple meanings for the same word form, will be represented as the same term. On the other hand, synonymous words, or different word forms that refer to the same meaning, will be represented as different terms. The context of the words are also lost, since the order that they appear in are not taken into account in their representation.

In addition, the BOW model also suffers from the "curse of dimensionality", which motivated Bengio et al. (2003) to propose a neural architecture that learns the distributed representation of words. The input layer of this model consists of sequences of words in the training vocabulary. The projection layer maps these input vectors into a real vector and the hidden layer applies a nonlinear activation function. A softmax function calculates the probability distribution of the words in the vocabulary to predict the next word in the sequence.

Mikolov et al. (2013) proposed two architectures to use for text representation: the Continuous Bag of Words (CBOW) model uses the surrounding words to predict the current word. It is similar to the architecture proposed by Bengio et al. (2003), however the CBOW model does not make use of the non-linear hidden layer. Additionally, the CBOW model also makes use of words from the future, in addition to the words preceding the term being predicted.

The second architecture they proposed is the Skip-gram model, which uses the current word as input to to predict the surrounding words. According to the authors, the current word is used as "input to a log-linear classifier with continuous projection layer, and predict words within a certain range before and after the current word".

Pennington et al. (2014) proposed Global Vectors (GloVe), a word representation model that uses the statistics of word occurrences in a corpus. Similar to CBOW and the Skip-gram models, GloVe also calculates the probabilities of word co-occurrence. Unlike the Word2Vec models however, GloVe calculates the ratios of the co-occurrence probabilities of words. Further, it looks at the global context of the corpus to determine co-occurrence, not just inside the context window. Instead of using a neural network, it uses a least squares model to learn the embeddings.

Peters et al. (2018) observed that while the previous pre-trained word embedding approaches were able to capture the semantic information of words, each word was represented by the same vector regardless of its context. To address this limitation, they proposed ELMo, a "deep contextualized word representation". It uses a bi-directional LSTM to learn the language model, where in addition to training the model to predict a word given the previous words in a sequence, it is also given the task of predicting the previous word given the words in the future.

Devlin et al. (2018) proposed BERT, a language model that uses a Transformer architecture (Vaswani et al., 2017). It masks words in a sequence randomly, and the model is given the task of predicting these words taking both the left and the right sequences of words in consideration. Unlike the previous vectorization methods which tokenize words by splitting strings at whitespace or other separator characters, BERT uses the WordPiece tokenizer, introduced in Wu et al., 2016. According to the authors, WordPiece splits strings into a "limited set of common sub-word units", making it more robust to handling unknown words.

Using the BOW model we are able to represent our documents as fixed-length vectors the size of the corpus vocabulary. Word embeddings, however, encode each *word* as a fixed-length vector. In the context of IR, we are interested in comparing the similarity at the document level, which are usually of variable word lengths, posing a challenge for algorithms that typically expect inputs of the same shape.

According to Le and Mikolov (2014), one simple approach that has been used to represent documents using word embeddings involves taking the weighted average of the word vectors in the document.

Du et al. (2021) and He et al. (2020) both used Word2Vec, specifically Skip-gram and CBOW respectively, to encode BRs, while Zaidi et al. (2020) used Word2Vec, GloVe and ELMo. They all represented BRs as matrices on which a CNN is trained. They addressed the unequal word lengths (and therefore matrix dimensions) by padding BRs shorter than a threshold BR length with a zero vector. Du et al. (2021) use the maximum BR length as the threshold, while He et al. (2020) and Zaidi et al. (2020) used 300.

Le and Mikolov (2014) proposed the Paragraph Vector (Doc2Vec) framework in order to address this limitation. A vector representing the paragraph is concatenated to the word vectors in the paragraph. Similar to the Word2Vec model, a context window is used to predict the next word in the sequence, with the paragraph vector added to this window.

Reimers and Gurevych (2019) also observed that "The most commonly used approach is to average the BERT output layer (known as BERT embeddings) or by using the output of the first token" when using BERT to represent sentences, which they found resulted in poor performance. To address this, they proposed Sentence-BERT (SBERT), a model based on BERT that uses Siamese and Triplet networks.

The SBERT model is trained on a dataset containing sentence pairs with the labels "*contradiction*", "*entailment*" and "*neutral*". Given a pair of sentences as input, SBERT uses a Siamese network structure which generates embeddings with the same parameters, allowing the two sentences to be represented with fixed-size vectors whose similarity can be obtained. To train the model, this Siamese network is optimized using the Triplet loss function. It takes an anchor sentence, and jointly aims to minimize its distance to a positive sentence, while maximizing its distance to a negative sentence.

2.4. Use of SOM in Information Retrieval

The Self-Organizing Map (SOM) (Kohonen, 1990) is an unsupervised ML algorithm that uses a neural network consisting of a grid of nodes. The SOM algorithm maps the high-dimensional input space into a two-dimensional grid of nodes by iteratively adjusting the weight vectors of the nodes.

During training, the input data is presented to the grid of nodes. The algorithm calculates the Best Matching Unit (BMU) for each point in the data, which is the node with the smallest distance to it. The weights of the nodes are then updated to more closely resemble the data points closest to it, taking into account both the Learning Rate (LR) and the neighborhood radius. The LR and neighborhood radius monotonically decrease after each iteration.

WEBSOM (Kaski et al., 1998) is one of the earliest applications of SOM in IR. This system uses a random mapping of the original VSM into a lower-dimensional vector with components composed of random variables. These vectors are then used to train a SOM. Training a SOM on these vectors results in each node representing a word category, containing words close to each other in meaning. Documents are encoded by generating a histogram representing the number of hits each node receives from the document, and a SOM is further trained on the histograms of hits.

Merkl (1998) used a variant of the SOM to organize a corpus composed of the manual pages of a software library. Each document is represented using a BOW approach, and a SOM is trained on these vectors. After the first map layer is trained, each resulting node is further trained into a two-dimensional SOM, using only the vector components that are different across the inputs in this node. This results in a hierarchical feature map, grouping documents into clusters arranged in an increasing level of granularity across SOM levels.

Instead of using the traditional VSM, Pullwitt (2002) and Pullwitt and Der (2001) first split the documents into sentences on which a SOM is trained. The resulting map contains nodes representing sentence categories, preserving the context otherwise lost in a VSM approach. A SOM is trained on the documents encoded using the histogram of sentence categories, enabling the visualization of the corpus' topology.

The VSM usually results in document vectors that are sparse and have very high dimensionality. In order to avoid this issue, Ampazis and Perantonis (2004) used Latent Semantic Indexing (LSI) on the document vectors to reduce dimensionality. A SOM is trained on the LSI representation of the document vectors, and the resulting map is used to generate documents' word category histograms on which a second SOM is trained. Use of the LSI enables not only the reduction in dimensionality, but also capturing the semantics.

Liu et al. (2008) aimed to preserve the semantic meaning of the words. In the authors' work, they do this by using a conceptual model in addition to using word frequencies to represent documents. They used HowNet (Dong & Dong, 2003), a knowledgebase that maps words to concepts to achieve this.

Amine et al. (2008) similarly used a concept-term database to capture the semantic meaning of words by using WordNet (Miller et al., 1990). The authors used SOM to cluster the documents. By using a corpus[5] with known cluster labels, the authors were able to calculate the precision, accuracy, and F-measure relative to the cluster labels generated by using SOM. The authors compared the results of encoding

[5] "Reuters-21578 Text Categorization Collection" (1997)

documents using concepts, with using the frequencies of occurrences of n-grams. They found that encoding documents using WordNet performed better than using n-grams.

Correa and Ludermir (2008) aimed to address the issue of high dimensionality in the input space. The VSM of the corpus is constructed by first removing stopwords and stemming. Documents are then represented by the frequencies of each of the terms present, weighted by TF-IDF. Using two clustering algorithms, K-means and the modified leader algorithm, the derived cluster centroids are then considered as document prototypes used as input to training the SOM. By using the labeled Reuters-21578 and the 20 Newsgroups[6] corpora, the authors were able to calculate the accuracy and macro- and micro-averaged F1 score. The highest scores were achieved by using the SOM directly on the VSM, however this setup also took the longest time to run. This is followed by the setup consisting of the combination of K-means and SOM, which had a reduced processing time.

Dey (2010) compared the results of using only unigrams to train a SOM with using bigrams in addition. After stopword removal, bigrams are constructed by combining two consecutive terms and considering them as a single unit. Low-frequency bigrams are removed, based on the assumption that high-frequency bigrams are meaningful and capture a semantic relationship. A VSM is constructed with term frequencies weighted with TF-IDF. This is then used to train a variant of SOM, the Emergent SOM, in which regions of the resulting map are separated not by hard borders denoted by each node, but rather by gradients. Using bigrams in addition to unigrams resulted in sharper region separation when compared to only using unigrams.

Stefanovic and Kurasova (2014) performed several experiments to determine the effects of changing the inputs to the SOM. Considering stopword removal, not performing this step results in a large Quantization Error (QE). Using a standard reference list improves this metric, and the best results are obtained when using a curated stopword list taking into account the context of the underlying corpus. Using stemming in combination with each of the previous experiments resulted in even worse values for the QE.

Till et al. (2014) preserved the context inherent in the corpus by creating context vectors generated by taking into account the frequency of the offset bigrams, or how often a word follows another word at offset positions. This produces vectors of high dimensionality, which is then projected down to a 400-dimension space. A word category map is constructed using these context vectors by training a SOM. A document map is similarly created using the context vectors.

Instead of using the report Title or Description fields, do Rego et al. (2008) decided to use the stack trace provided to identify duplicate bug reports in a dataset. While not written in natural language, the authors argue that this removes some issues that come with manually written text such as spelling mistakes and synonymy. Stopwords were removed, as well as terms that occurred too frequently or too infrequently. Documents were then represented using their TF-IDF vectors before being used as input to a SOM. They evaluated the results using the recall rate, the calculation modified such that it refers to “the percentage of duplicates in which the master BR is found”. An additional metric they used was the list size, referring to the number of documents in the BMU of the BR. Their experiments were able to identify 69% of the identified duplicates. In presenting only the contents of the BMU and the neighboring nodes, they were able to narrow down the number of BR to be inspected in searching for duplicates.

[6] Lang (1995)

Ahmed and Ghazali (2016) studied using SOM to cluster a dataset composed of bug reports. Preprocessing consisted of removing stopwords, and documents were vectorized using a binary BOW representation. The authors proposed using a new similarity metric, the Jaccard index. In their proposal, the Euclidean distance will still be used to find the BMU for each data point, and to update the BMU. However the neighbors of the BMU will be updated using the Jaccard index calculated. They compared the results of this setup with the clustering results obtained using both K-means and the original SOM, by measuring the cluster homogeneity and separation. They also used the Jaccard index and the Fowlkes-Mallows index to evaluate cluster quality. Their proposed algorithm obtained better separation than both K-means and the original SOM. It out-performed the original SOM on homogeneity, and had similar scores to K-means.

Yoshioka and Dozono (2019) used a SOM to classify documents labeled with their categories. They used Word2Vec to represent each unique word in the corpus, and used these word vectors to train the first layer of a spherical SOM. The document vectors are created by counting the number of times each word in each SOM node appears in the document, like a "Bag of SOM Nodes" instead of a frequency-based BOW. A second layer of SOM is then trained using these document vectors.

In de Miranda et al. (2020), the SOM was used on the 20 Newsgroups dataset with the objective of determining the topics present in the corpus. Word2Vec was used to generate the vector representation of each word in the corpus, which were then used to train a SOM. Once the training is completed, the K-Means algorithm was used on the resulting map, using the number of categories as the value for k. Looking at the words that appeared in each cluster, the authors found that this approach was able to cluster together words related to each other, and related to the ground truth category label.

Onan and Tocoglu (2021) performed experiments to determine which combination of various text representation techniques, weighting functions, and document clustering algorithms performed the best for topic modeling. They tested using GloVe, Word2Vec, fastText, and Doc2Vec for text representation, combined with various weighing schemes. Each of these was used as input to these algorithms: K-Means, K-Means++, SOM, and Divisive Analysis clustering (DIANA). They found that DIANA achieved the best scores among all the clustering algorithms using Doc2Vec with TF-IDF weighing. SOM was also able to achieve its best performance with this combination.

2.5. Summary and Research Gap

Considering the growth of the number of BR being filed for software projects, there exists great interest and need in developing automated approaches for deciding who should fix each bug. Most of the past literature that made use of the textual components of BRs used a BOW model to represent the textual features. More recently, word embedding techniques have also been proposed in combination with Deep Learning techniques.

SOM have long been used in IR tasks, and being able to capture the semantics of the text has been of interest to researchers. Its application to triaging BRs is not as established, and neither is the use of word embeddings and sentence embeddings in text representation.

This thesis aims to bridge this gap by studying the effect of using different text representation techniques as input to the SOM applied to the task of BR triage.

3. Methodology

All of the code used to develop this thesis, including the application demo, can be found at this GitHub repository: github.com/fpontejos/sombr-thesis.

3.1. Proposed Approach and Experimental Setup

We propose to use SOM as a semi-supervised algorithm for BR triage. We consider the contents of the issue tracker databases of several open source projects as our corpora, treating each qualifying BR as a document. Each dataset is preprocessed in the same way. Several vectorization techniques are used to represent the documents in the corpus, and we use the `assigned_user` as the target variable.

For each dataset, we first remove BRs not considered valid, detailed further in the next section. The remaining BRs are then sorted by `resolved_date`, then divided into a `Train` set and a `Test` set. From the `Test` set we only keep the BR whose `assignee_user` appears in the `Train` set.

After vectorizing the `summary` fields of each of the BRs in these subsets, we use the `Train` set as input to train a SOM. Each BR in the `Test` set is labeled with the `assigned_user` that has fixed the most bugs in the BMU it is associated with, and we evaluate these results using several metrics, discussed in Section 3.7.

Figure 3.7 shows an overview of the setup of the experiments performed in this study. Each document in the corpus undergoes a baseline preprocessing step, followed by one of the variations noted in the figure. The preprocessing steps are discussed in more detail in Section 3.4. After preprocessing, the text is then vectorized using one of the methods, before being used as input to train a SOM. Several SOM initialization parameters are also tested.

We then develop an interactive dashboard that allows users to explore the BR corpus visually. The user is presented with the SOM UMatrix as the main visual feature, with the Graph Layout view and Detail view as additional visuals. A more detailed description of the dashboard as well as its features will be discussed further in Section 3.8. Figure 3.1 presents an overview of the methodology used in this research.

3.2. Description of the Data

Rath and Mäder (2019a) provide “The SEOSS Dataset - Requirements, Bug Reports, Code History, and Trace Links for Entire Projects”, a dataset derived from 33 open-source software projects, each with the given schema in Figure 3.2.

The `issue` table contains the details of each report filed, and the `issue_link` table contains the relationships linking issues to each other. The attributes selected for both tables are summarized in Tables A1 and A2 respectively. Figure 3.3 shows the contents of an example BR with usernames and identifiable information concealed.

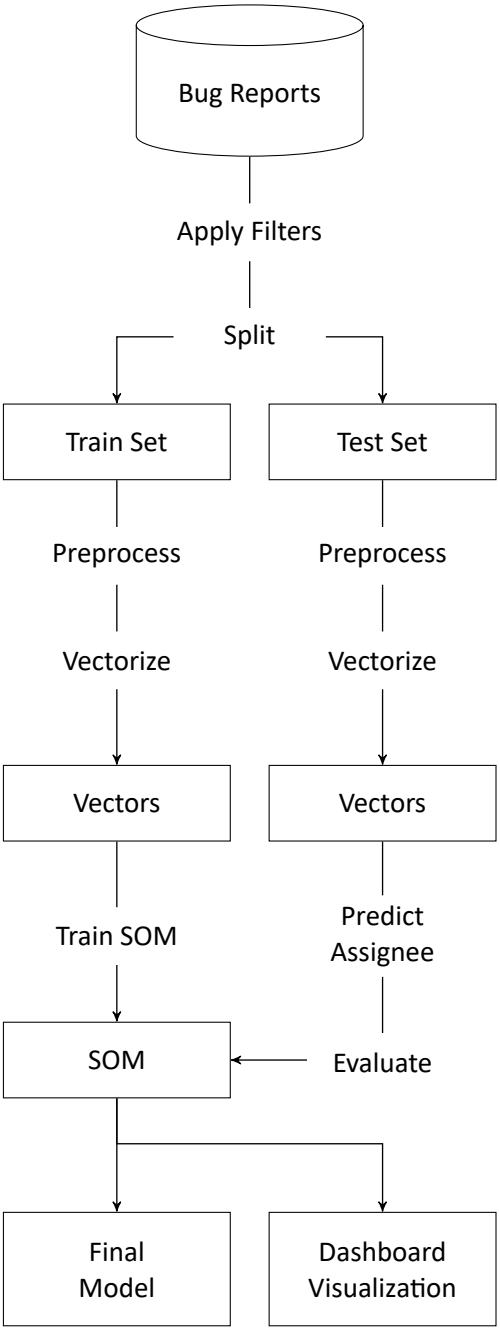


Figure 3.1.: Overview of Methodology

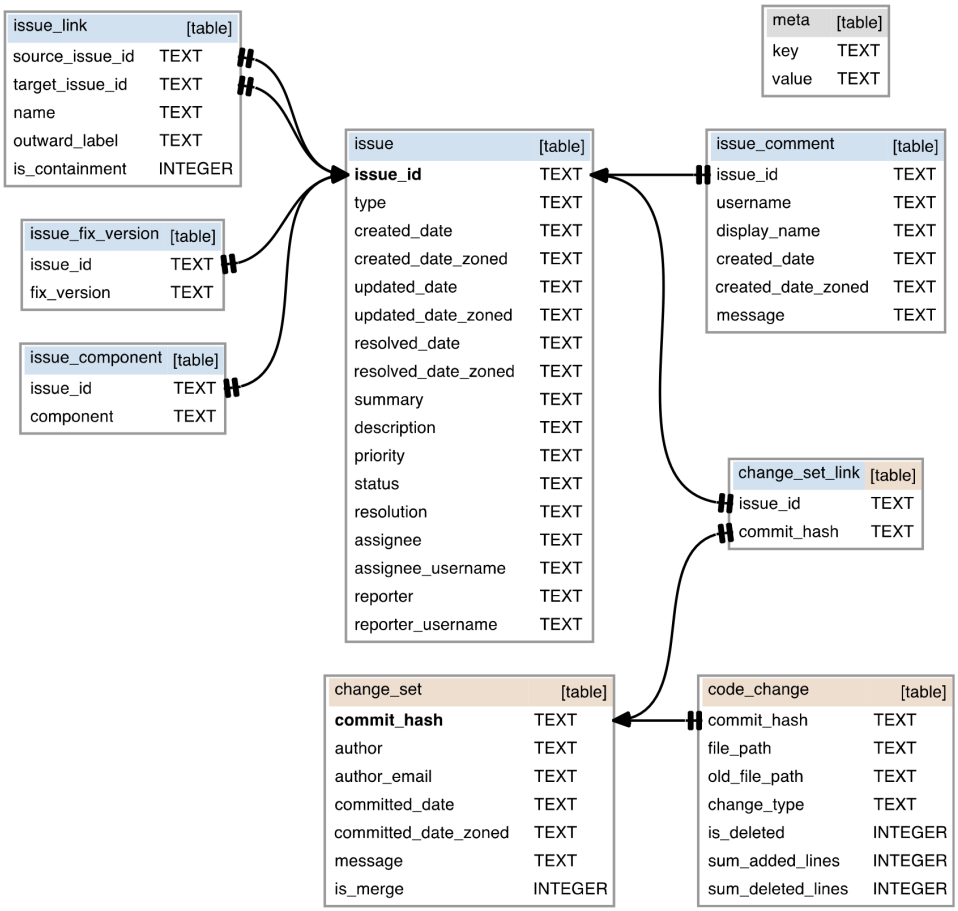


Figure 3.2.: Database schema of SEOSS Dataset (Rath & Mäder, 2019a)

The issues from the `issue` table were filtered according to the following rules, and Figure 3.4 shows a diagram of this schema.

1. `type` must be equal to "`Bug`". This removes from consideration other issue types such as new feature requests, which do not describe software defects.

2. `status` must be one of "`Resolved`" or "`Closed`". This filters out issues that are not in a state of finality, since those issues may either not have a user assigned, or the assigned user may still change.

3. `resolution` must be one of "`Done`", "`Fixed`", or "`Resolved`", for the same reasons as the previous.

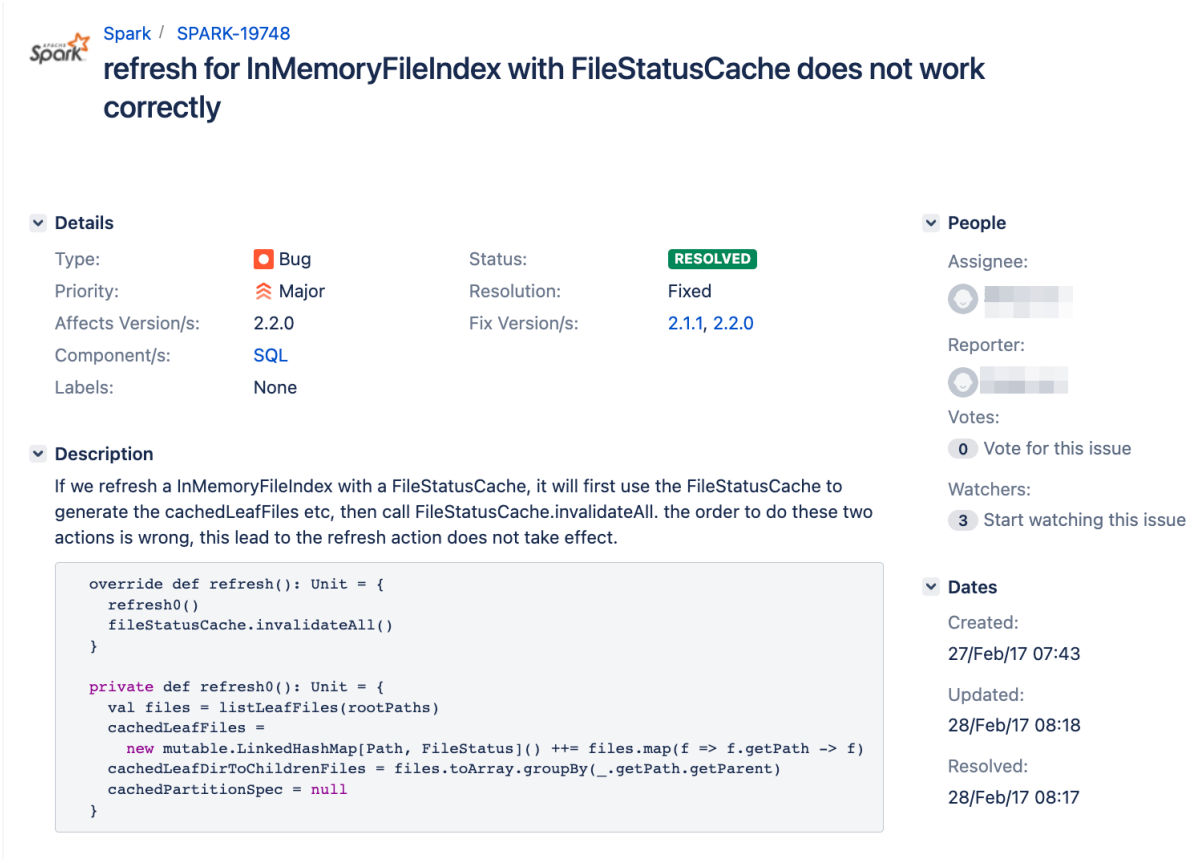
4. `assigned_user` must have a value. This removes issues that do not have a value for the target label.

5. Issues that are duplicated are removed. This filter is in addition to excluding issues of `type` "Duplicate". These issues are identified by identifying entries with exactly the same values for the `summary` and `description` fields.

We select only the databases that have a minimum of 2 500 BRs left after this filtering step, leaving us with the following corpora in Table 3.1. The complete list of projects and their corpus sizes are given in Table A4.

Table 3.1.: Projects included in experiments

Project	Filtered	Total
cassandra	4 664	13 965
groovy	3 513	8 137
hadoop	11 490	39 086
hbase	5 899	19 247
hibernate	2 749	11 971
hive	5 836	18 025
infinispan	2 910	8 422
jbpm	2 615	10 397
lucene	3 823	17 329
spark	4 889	22 205



Spark / SPARK-19748

refresh for InMemoryFileIndex with FileStatusCache does not work correctly

Details

Type:	Bug	Status:	RESOLVED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	2.2.0	Fix Version/s:	2.1.1, 2.2.0
Component/s:	SQL		
Labels:	None		

Description

If we refresh a InMemoryFileIndex with a FileStatusCache, it will first use the FileStatusCache to generate the cachedLeafFiles etc, then call FileStatusCache.invalidateAll. the order to do these two actions is wrong, this lead to the refresh action does not take effect.

```
override def refresh(): Unit = {
  refresh0()
  fileStatusCache.invalidateAll()
}

private def refresh0(): Unit = {
  val files = listLeafFiles(rootPaths)
  cachedLeafFiles =
    new mutable.LinkedHashMap[Path, FileStatus]() ++= files.map(f => f.getPath -> f)
  cachedLeafDirToChildrenFiles = files.toArray.groupBy(_.getPath.getParent)
  cachedPartitionSpec = null
}
```

People

Assignee:

Reporter:

Votes:

0 Vote for this issue

Watchers:

3 Start watching this issue

Dates

Created:
27/Feb/17 07:43

Updated:
28/Feb/17 08:18

Resolved:
28/Feb/17 08:17

Figure 3.3.: Screenshot of Bug Report

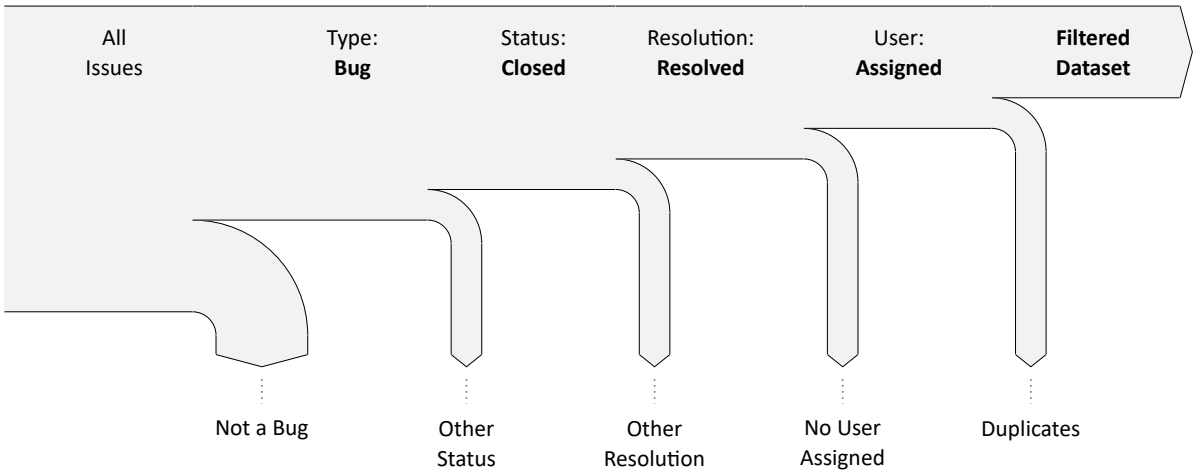


Figure 3.4.: Data Filtering

3.3. Characteristics of the Corpora

Exploring the corpora we can observe that there are a large number of values for `assignee_user`, which is in this case the target label we are aiming to predict. The values range from 56 (`groovy`) up to 709 (`spark`). Additionally, Figure 3.5 also shows the mean number of BR per assignee, as well as how many BR each corpus has. We can observe that we have a wide range of values for all three.

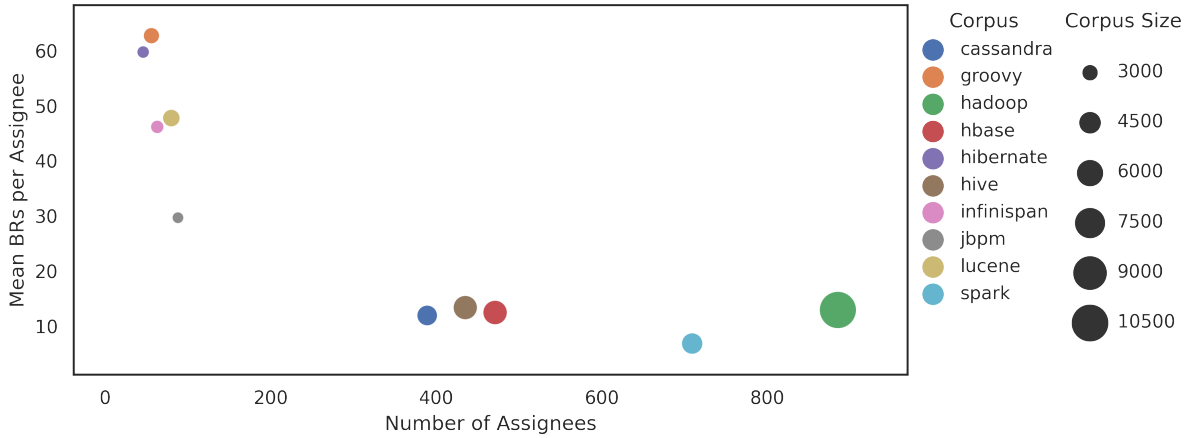


Figure 3.5.: Characteristics of the selected corpora: Number of assignees, mean BR per assignee, and number of documents in the corpus

Figure 3.6 shows the mean and median values for the number of BR assigned to the users, and we can see that the median values are much lower than the mean values.

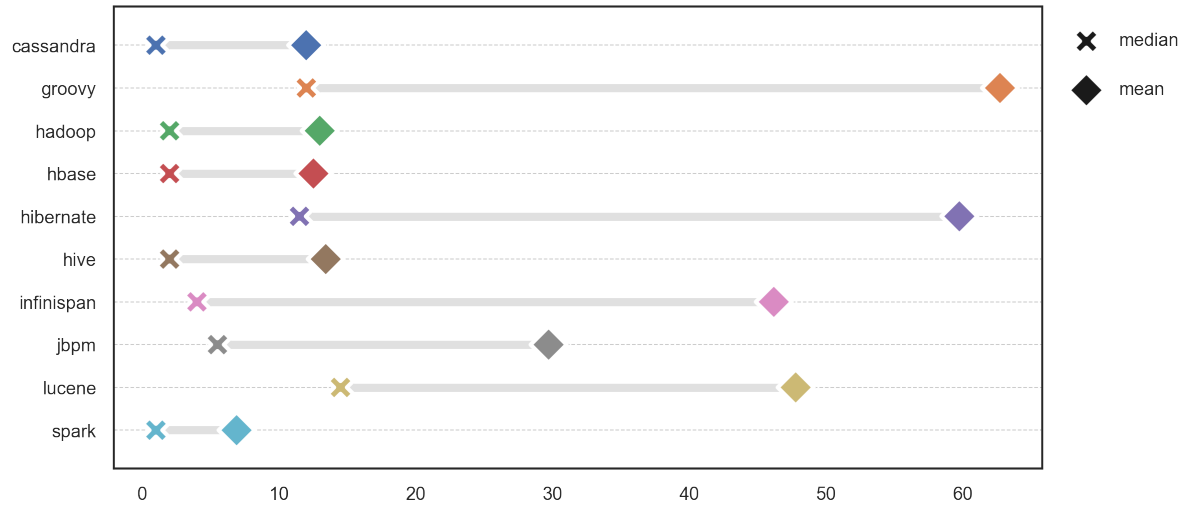


Figure 3.6.: Characteristics of the selected corpora: mean and median of number of BRs assigned per user

3.4. Preprocessing

We tested the performance of the framework using different preprocessing steps to determine if these have any effect on the quality of the vectorization produced. The baseline condition includes basic preprocessing, denoted in Figure 3.7 as "Baseline". We used the tokenizer function of the `nltk` Python package to accomplish this step, which splits text at whitespace characters and punctuation symbols. At this stage we also remove non-alphabetic characters including digits and all other symbols, and replace them with a whitespace character. Excessive whitespaces are combined to a single whitespace instance. All further preprocessing steps use the "Baseline" preprocessing as an initial step.

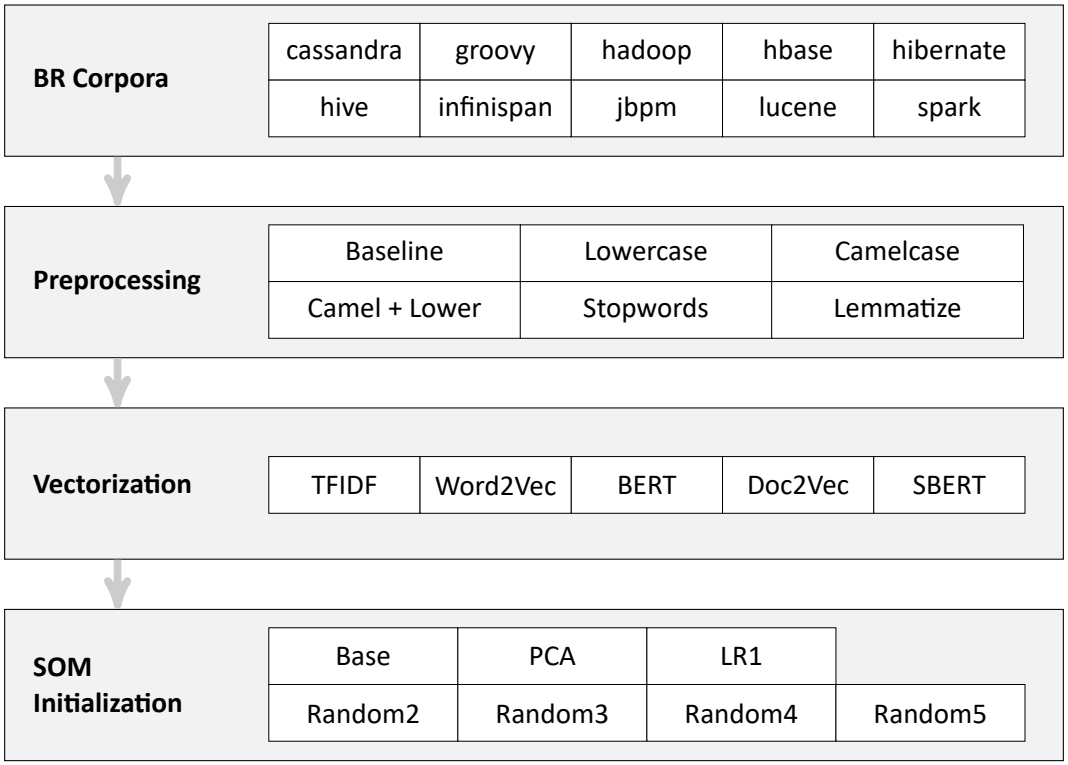


Figure 3.7.: Experimental Setup

In the style guide for Python (van Rossum et al., 2001), the authors describe some common naming styles being used for variables. These are listed and annotated with the respective treatments performed in Table 3.2. The naming styles making use of underscores are addressed in the initial baseline preprocessing step when symbols are removed, and `mixedCase` and `CapitalizedWords` are addressed in the "Camelcase" preprocessing variation.

"Lowercase" preprocessing involves changing the case of the characters in the documents to all lowercase. The "Camelcase" variation involves changing strings from camelcase to sentence case, and "Camel Lower" uses a lowercasing step in addition to "Camelcase". "Sentence case" here means that the first letter in the sentence is in uppercase, with the rest in lowercase.

We test the removal of stopwords, another common preprocessing step in Natural Language Processing (NLP). Stopwords are the most commonly used words and do not add semantic value. Finally, we also test the performance when lemmatization is performed on the documents. Table 3.3 presents the effects of the various preprocessing steps to some BR examples.

Table 3.2.: Naming conventions and subsequent preprocessing treatment

Naming Style	**Treatment**
`CapitalizedWords`	split string at capitalized letter (camelcase)
`mixedCase`	split string at capitalized letter (camelcase)
`lower_case_with_underscores`	replace underscore with whitespace (baseline)
`Capitalized_Words_With_Underscores`	replace underscore with whitespace (baseline)

3.5. Text Vectorization

One of the objectives of this research is to determine whether using sentence embedding techniques improve the performance of a SOM. We selected several vectorization approaches for comparison.

3.5.1. TF-IDF

We used a BOW model with terms weighted using TF-IDF as the baseline vectorization method. We did not set a threshold minimum frequency, which is a requirement that a term must occur in at least this percentage of documents in the corpus to be considered.

3.5.2. BERT and SBERT

We used a BERT model pre-trained on the `MiniLM-L6-H384-uncased (MiniL6)` dataset to generate BERT word embeddings, which in turn is based on `MiniLM-L12-H384-uncased (MiniL12)` (Wang et al., 2020), trained using text from the English Wikipedia corpus and BookCorpus.

`MiniL12` is an uncased model, meaning the input text is transformed to lowercase before being tokenized. It uses a 12-layer Transformer architecture with 384 hidden units. `MiniL6` keeps only every second Transformer layer of `MiniL12`.

This BERT model produced tensors with dimensions given in Equation 3.1 and illustrated in Figure 3.8.

$$|\text{BR}| \times |\text{BR Tokens}| \times |\text{Hidden Units}| \tag{3.1}$$

Obtaining the BR embeddings using the BERT model involves two steps: tokenizing each BR, and encoding the result. In the tokenizing step, we decided to pad the BRs with zeroes up to the length of the BR with the most number of words as described by Du et al., 2021. Once the tokens are encoded, we retrieve the vector corresponding to the first token for each BR, denoted in Figure 3.8 by the shaded cells. According to Devlin et al. (2018), this corresponds to the `[CLS]` token, which is used to denote the beginning of a sentence. The authors further state that "[t]he final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks". This results in each BR vector having dimensionality of 384, dimensions similar in magnitude to those produced by the SBERT model described in Equation 3.2 and Figure 3.9.

Table 3.3.: Examples of Preprocessing Results

Description	Value
issue_id	CASSANDRA-294
Original	missing lib/license/google-collect-1.0-rc1.jar.LICENSE
Baseline	missing lib license google collect rc jar LICENSE
Lowercase	missing lib license google collect rc jar license
Camelcase	missing lib license google collect rc jar L I C E N S E
Camel Lower	missing lib license google collect rc jar l i c e n s e
Stopwords	missing lib license google collect rc jar l c e n e
Lemmatize	miss lib license google collect rc jar l c e n e
issue_id	CASSANDRA-935
Original	“login() request via Thrift/PHP fails with ”“Unexpected authentication problem”“ in cassandra log / ”“Internal error processing login”“ in Thrift”
Baseline	login request via Thrift PHP fails with Unexpected authentication problem in cassandra log Internal error processing login in Thrift
Lowercase	login request via thrift php fails with unexpected authentication problem in cassandra log internal error processing login in thrift
Camelcase	login request via Thrift P H P fails with Unexpected authentication problem in cassandra log Internal error processing login in Thrift
Camel Lower	login request via thrift p h p fails with unexpected authentication problem in cassandra log internal error processing login in thrift
Stopwords	login request via thrift p h p fails unexpected authentication problem cassandra log internal error processing login thrift
Lemmatize	login request via thrift p h p fail unexpected authentication problem cassandra log internal error process login thrift
issue_id	CASSANDRA-966
Original	StorageService.getPartitioner() and QueryFilter.getColumnComparator() should be statically accessed
Baseline	StorageService getPartitioner and QueryFilter getColumnComparator should be statically accessed
Lowercase	storageservice getpartitioner and queryfilter getcolumncomparator should be statically accessed
Camelcase	Storage Service get Partitioner and Query Filter get Column Comparator should be statically accessed
Camel Lower	storage service get partitioner and query filter get column comparator should be statically accessed
Stopwords	storage service get partitioner query filter get column comparator statically accessed
Lemmatize	storage service get partitioner query filter get column comparator statically access

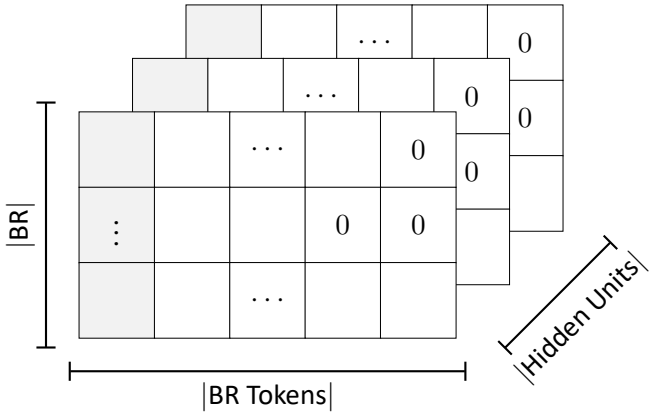


Figure 3.8.: Dimensions of BERT Tensors

To obtain the SBERT embeddings, we used an SBERT model that was pre-trained on the same base `MiniL6` model. The model authors (Reimers, 2022) fine-tuned `MiniL6` with additional sentence pairs from multiple datasets and designed it for use as a general purpose model. This results in vectors with dimensions given by Equation 3.2 and illustrated by Figure 3.9.

$$|\text{BR}| \times |\text{Hidden Units}| \tag{3.2}$$

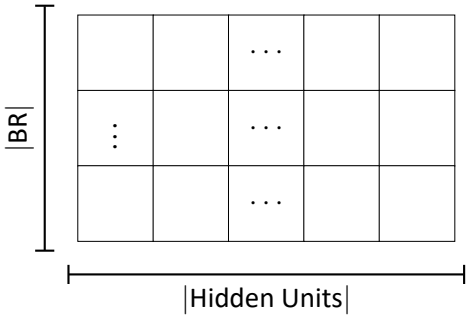


Figure 3.9.: Dimensions of SBERT Vectors

3.5.3. Word2Vec and Doc2Vec

We trained Doc2Vec and Word2Vec models using the English Wikipedia corpus available from the Gensim Data repository (RaRe-Technologies, 2017). The Gensim implementation of the Doc2Vec algorithm provides a way to simultaneously train a Word2Vec model in the process. We used a vector size of 300 and a window size of 8, with the complete list of descriptions and values of the training parameters in Table 3.4. To obtain the embeddings of the whole BR using Word2Vec, we take the element-wise average of the each word vector generated by the model. Words that are out of vocabulary (OOV) are replaced by a zero vector.

Table 3.4.: Doc2Vec and Word2Vec training parameters
Descriptions from Gensim implementation documentation (Řehůřek, 2022)

Parameter	Value	Description
dm	0	Train in DBOW mode (Analogous to Word2Vec CBOW)
dbow_words	1	Train Word2Vec word vectors simultaneously
vector_size	300	Dimensionality of resulting vectors
window	8	Window size as described in Le and Mikolov (2014)
epochs	10	Number of training iterations
workers	10	Number of threads
max_final_vocab	1 000 000	Limit of vocabulary size

3.6. Training the SOM

We used MiniSom, a Python implementation of SOM in our experiments (Vettigli, 2018). According to its documentation, it is "a minimalistic and Numpy based implementation of the [SOM]". The complete list of available input parameters are given in Listing A1 and the values we used for training are listed in Table 3.5.

Table 3.5.: MiniSom training parameters

Parameter	Value
x	15
y	15
input_len	size of word embedding vector
learning_rate	0.7
random_seed	0
num_iteration	10000
topology	hexagonal

3.6.1. Map Size

Kaski et al. (1998) chose a map size such that on average each map unit would contain ten documents to facilitate easy browsing. Following this approach, we used a map size of 15 x 15, to accommodate the different corpus training sizes in our data, which ranged in values between 2 545 and 10 777.

3.6.2. Decay

Kohonen (1990) recommended that the value of LR be close to 1 initially, and monotonically decrease at each training step. We use an initial LR of 0.7, keeping the default value set by MiniSom. The default decay function is used as well, expressed mathematically in Equation 3.3 and illustrated in Figure 3.10. With this function, the LR drops to half its initial value at the halfway point of training, and ends with a third of its initial value.

$$LR_t = \frac{LR_0}{1 + \frac{2 \times t}{T}} \tag{3.3}$$

$where$

$T =$ max number of training steps

$t =$ current training step

$LR_0 =$ initial LR

$LR_t =$ current LR

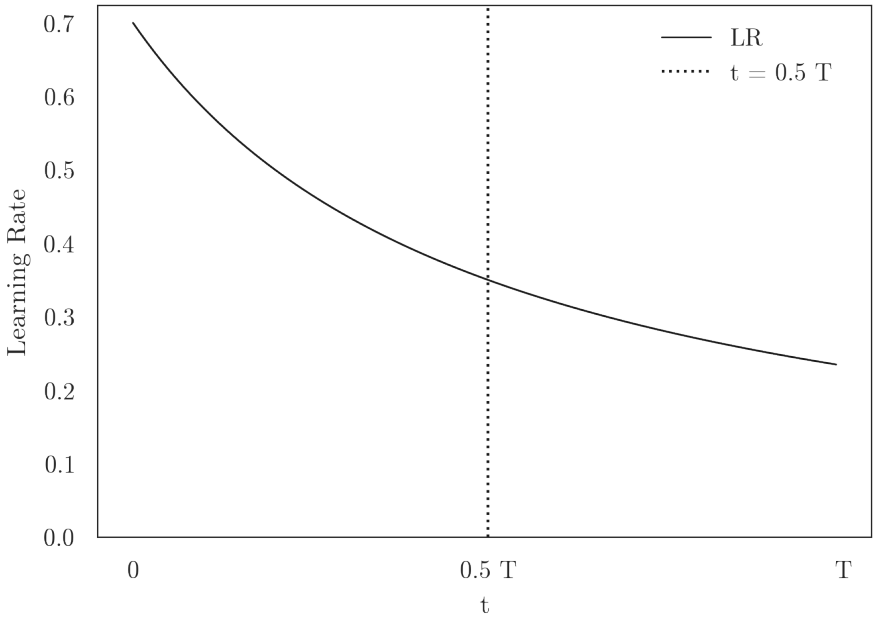


Figure 3.10.: Default Learning Rate and decay

3.7. Evaluation

3.7.1. Precision

Precision is a common metric used in IR tasks, which according to Jurafsky and Martin (2023), is calculated as the fraction of relevant documents among the retrieved documents (Equation 3.4). In the context of BR triage, we consider a document to be relevant if its assigned fixer is the same as the query BR's actual fixer, shown in Equation 3.5.

$$
\begin{aligned}
Precision &= \frac{|R|}{|T|} \\
Recall &= \frac{|R|}{|U|} \\
where\ R &= \text{number of relevant documents} \\
T &= \text{number of documents retrieved} \\
U &= \text{number of all relevant documents}
\end{aligned}
\tag{3.4}
$$

given a query with known assigned user A,

$$
Precision = \frac{\text{number of BRs assigned to } A \text{ in BMU}}{\text{number of BRs in BMU}} \tag{3.5}
$$

$$
Accuracy = \frac{\text{number of BRs correctly assigned}}{\text{number of BRs in corpus}} \tag{3.6}
$$

3.7.2. Recall

Recall is another common metric used to evaluate IR systems, and is calculated as the proportion of relevant documents that are present in the documents retrieved. However, following our above definition of "relevant documents", we considered it not to be a good measure for the task of BR triage. In practice, developers may fix bugs in semantically different areas of the software project, for example relating to memory leaks as well as documentation issues, and these would both be counted towards "relevant documents".

3.7.3. Accuracy

According to Nagwani and Suri (2023), accuracy is another common metric used in BR research. This is calculated by taking the proportion of correctly assigned BR out of all the BRs in the corpus (Equation 3.6).

We consider two ways of reporting a "correct" assignment:

1. $Accuracy@1$, which reports "correctly assigned" when the majority class in a query BR BMU matches the query's actual assigned user. If there are several values for the mode, it is considered "correctly assigned" if the actual `assigned_user` matches one of them.

2. $Accuracy@BMU$, which reports "correctly assigned" when the query BR's assigned user is present in its BMU.

Pölzlbauer (2004) described and compared several evaluation measures that can be used to quantify the quality of a SOM. They divided the measures into those that measure the quality of vector quantization, and those that aim to evaluate the topology preservation.

3.7.4. Quantization Error

According to Pölzlbauer (2004), the QE "is computed by determining the average distance of the sample vectors to the cluster centroids by which they are represented." Mathematically, this is given by Equation 3.7, where N is the number of input vectors assigned to a node in the SOM map, x_i are the input vectors, and BMU_i is the representative vector or the BMU of the node.

$$QE = \left(\frac{1}{N}\right) \sum_{i=1}^{N} \|x_i - BMU_i\| \tag{3.7}$$

3.7.5. Topographic Error

The Topographic Error (TE) measures the topology preservation of the SOM model and is calculated as "the proportion of all data vectors for which first and second BMUs are not adjacent units." (Vesanto, 2003). Kiviluoto (1996) formulates TE as Equation 3.8.

$$TE = \frac{1}{N} \sum_{k=1}^{N} u(x_k) \tag{3.8}$$

$$\text{where } u(x_k) = \begin{cases} 1 & \text{best- and second-best-matching units non-adjacent} \\ 0 & \text{otherwise} \end{cases}$$

3.8. Visualization

We used Bokeh[1], an open source visualization library written in Python to develop the interactive dashboard. We used SBERT to vectorize the corpus, and used MiniSom to train a SOM using the map parameters in Table 3.5.

[1] https://bokeh.org/

The dashboard framework consists of four main parts. The "Main" component orchestrates the flows between the other components: Vectorization, Organization, and Visualization.

The Main component first parses the command-line arguments provided when the dashboard is initially run. Given default settings, it will load a pre-trained SOM model based on a previously vectorized dataset, and use these as input to the Visualization component. Given the argument to force retraining a model, it will read the data from a `sqlite` database file into a Dataframe. The data will be filtered, keeping only the valid rows as discussed in Section 3.4, and only a limited number of columns: the `summary` field, containing the text of the BR, the `assignee_user` column, which is the target label, and other columns containing metadata relevant to the document, in this case the `issue_id`. These will then be passed as input to the Vectorization component.

The Vectorization component takes the text column containing the documents and uses SBERT to calculate the document vectors corresponding to each row. The usernames in the `assignee_user` column are encoded as number labels to anonymize the names, and the rest of the metadata columns are kept as-is. A copy of the original text is also preserved.

The document vectors are used in the Organization component and used to train the SOM. This component returns the trained SOM model with the following properties which serve as input to the Visualization component:

1. the Euclidean coordinates of the map lattice, which allow us to place each map unit hexagon on the 2-dimensional visualization plane;

2. the UMatrix, containing the average distance of each unit to its neighbors. It is visualized as a full-radius hexagon, whose color value is encoded using the Viridis color palette (Figure 4.9b).

3. the hits matrix, representing the number of documents each BMU has won. It is visualized as a white hexagon, whose radius encodes the relative size of the number of hits (4.9a).

4. the quantization errors, calculated as in Equation 3.7. It is visualized as a full-radius hexagon, whose color value is encoded using the Magma color palette (Figure 4.9d).

5. the weights of the map units, used to calculate the distance between each unit to each of its neighbors.

We used the `neato` program of the graphviz library, which uses the stress majorization algorithm (Gansner et al., 2005) to calculate the graph layout of the nodes. The library accepts the edge distances as input (Figure 4.9c) to perform this calculation. The visual properties of the edges of the graph layout further encode the edge distances using color: using the Viridis palette, the yellow region denotes larger distances and the darker purple color denotes smaller distances. The opacity of the edges also reflect the distance, with larger distances appearing more transparent.

Using the interactive features of the dashboard allows the triager to obtain the BR that are the most semantically similar to the new BR being considered for assignment. Additionally, developers may also use this as a tool to explore the corpora of previous BR to retrieve those most similar to a BR assigned to them to see how they were resolved.

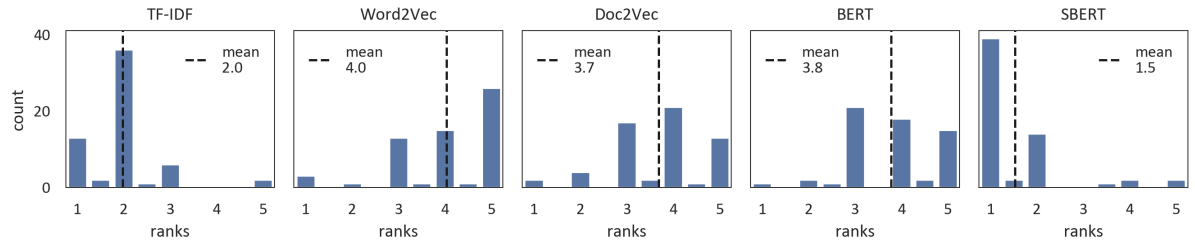
4. Results and Discussion

In this chapter we present the results of the experiments we performed. First we discuss the effects of using different text vectorization techniques on the BR corpora. Using statistical analysis we show that the differences in the resulting metrics are significant. We then examine the results of using different initialization parameters when training the SOM. Finally we present SOMBR, the interactive dashboard that was developed.

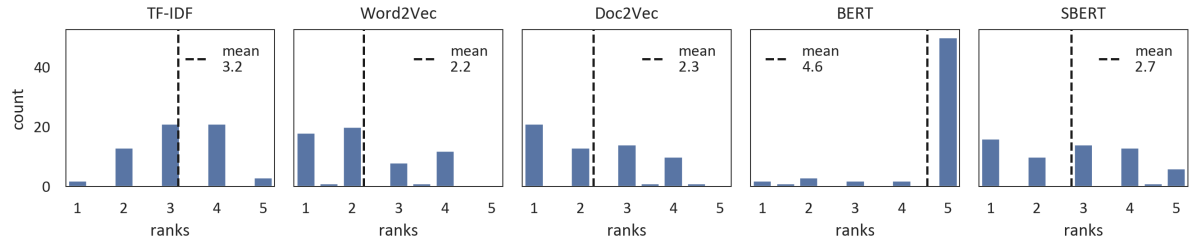
4.1. Effect of Text Vectorization

As described in the previous chapter, different preprocessing methods were performed on each of the BR corpora. After preprocessing, each BR dataset was vectorized using each of the five methods in consideration. Since we are interested in determining whether the different vectorization methods result in a statistically significant difference in the metrics we are measuring, we calculated the ranks attained by each vectorization method for each corpus and preprocessing method.

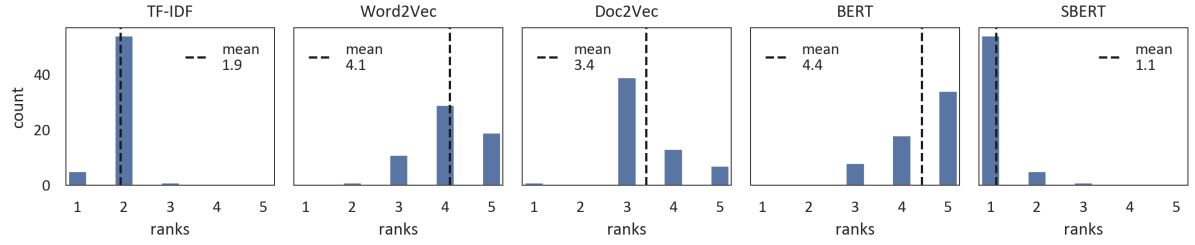
The summary of the rank values can be visualized in Figure 4.1, which shows the frequency at which each vectorization method attained each rank value for each of the observations.



(a) Accuracy@1



(b) Accuracy@BMU



(c) Precision

Figure 4.1.: Count of Ranks: Accuracy@1, Accuracy@BMU, and Precision
Values in between whole numbers (e.g. 1.5) correspond to ties.

4.1.1. Statistical Evaluation

Demšar (2006) recommended the use of the Friedman test in evaluating the performance of different classifiers when used on multiple datasets, as is the case in this research. According to the author, it compares the average ranks of algorithms, instead of the values of the metrics measured, based on the null hypothesis that if the models are equal, their average ranks are also equal. More formally, we state the null hypothesis as follows:

> H_0: The scores obtained when using the SOM to predict the `assigned_user` is the same when different methods are used to vectorize BR.

To perform the Friedman rank sum test, we calculate the rank of each of the vectorization method on each preprocessed corpus. For $Accuracy@1$, $Accuracy@BMU$ and $Precision$, the highest scores are better and correspond to a rank of 1. For TE and QE however, the opposite is true; the lowest score is the best and correspond to a rank of 1. The complete results of the values and ranks obtained can be found in Tables B1, B2, and B3.

The Friedman rank sum test was performed considering the 60 observations (10 corpora$\times 6$ preprocessing) as the population of interest and each of the five vectorization methods as different treatments. The test was performed for each of the metrics we measured, and the results for Accuracy and Precision are summarized in Table 4.1.

Table 4.1.: Results of Friedman Rank Sum Test: Accuracy and Precision

Metric	**Friedman chi-squared**	**p-value**
Accuracy@1	129.710	<0.001
Accuracy@BMU	86.593	<0.001
Precision	194.947	<0.001

We can observe that each test statistic for each evaluation metric measured correspond to a p-value below the 0.001 significance level. Based on this we can reject the null hypothesis that the different vectorization methods have the same effect. Tables 4.3 and 4.2 summarize the mean values of the scores and ranks of each, respectively.

Table 4.2.: Mean Values of Ranks

Metric	**TFIDF**	**W2V**	**D2V**	**BERT**	**SBERT**
Accuracy @ 1	1.975	4.033	3.692	3.775	1.525
Accuracy @ BMU	3.167	2.250	2.283	4.558	2.742
Precision	1.933	4.100	3.417	4.433	1.117

Table 4.3.: Mean Values of Scores

Metric	TFIDF	W2V	D2V	BERT	SBERT
Accuracy @ 1	0.159	0.131	0.136	0.135	0.174
Accuracy @ BMU	0.410	0.427	0.426	0.356	0.417
Precision	0.087	0.066	0.069	0.062	0.097

Following this result, we performed the Nemenyi post-hoc test for each of the metrics measured. This test compares the performance of classifiers with each other, and the difference in the average ranks of each pair being compared is found to be significant if they differ by at least the Critical Distance given by Equation 4.1 (Demšar, 2006).

$$
\begin{aligned}
CD &= q_\alpha \sqrt{\frac{k(k+1)}{6N}} \\
where\ N &= \text{number of datasets} \\
k &= \text{number of algorithms being tested} \\
q_\alpha &= \text{critical values}
\end{aligned}
\tag{4.1}
$$

4.1.2. Accuracy@1

Performing a pairwise comparison of the resulting $Accuracy@1$ scores from each vectorization method results in this p-value matrix in Table 4.4a. We can observe that there are some pairs that have a statistically significant difference between them: the $Accuracy@1$ scores of both TF-IDF and SBERT are larger and highly statistically significant compared to Word2Vec, Doc2Vec, and BERT, thus we can conclude that using TF-IDF or SBERT is better than using any of the other methods considered. However, despite SBERT having a better $Accuracy@1$ score and rank on average than TF-IDF, the difference between them is not statistically significant.

4.1.3. Accuracy@BMU

Comparing the values of $Accuracy@BMU$, we can observe that BERT is highly statistically different from all the other vectorization methods, and at $\alpha = 0.05$ TF-IDF has a statistically significant difference to the other methods except SBERT. Since BERT and TF-IDF have the lowest and second-lowest ranks on average, we can conclude that these two performed the worst on average; however among the other three methods there is no difference in their performance.

4.1.4. Precision

The $Precision$ metric mirrors the results of the $Accuracy@1$ pairwise comparison in that TF-IDF and SBERT are both highly statistically significant compared to the other methods, but the difference between each other is not. In addition to this, BERT and Doc2Vec are significantly different, with Doc2Vec performing better on average.

4.1.5. Summary of BR Triage Evaluation

Based on the statistical tests we performed, we can conclude that there is a statistically significant difference in the performance on the BR triage task when different vectorization methods are used to represent documents. SBERT, a sentence embedding method, and TF-IDF, a frequency-based vectorization technique, both performed better than the other methods considering the metrics of $Accuracy@1$ and $Precision$.

Using the less strict metric of $Accuracy@BMU$, all vectorization methods observed an increase of between 140% up to 225% compared to their mean $Accuracy@1$ scores.

Figures B1, B2, and B3 show the different scores obtained by each vectorization, identified by each corpus and the preprocessing method applied.

Plotting the mean word count for each corpus after the baseline preprocessing and comparing this with the results after camelcase splitting, we can see from Figure 4.2 that there is a large difference for all corpora. However looking at Figure 4.3 there does not seem to be a consistent corresponding increase in performance.

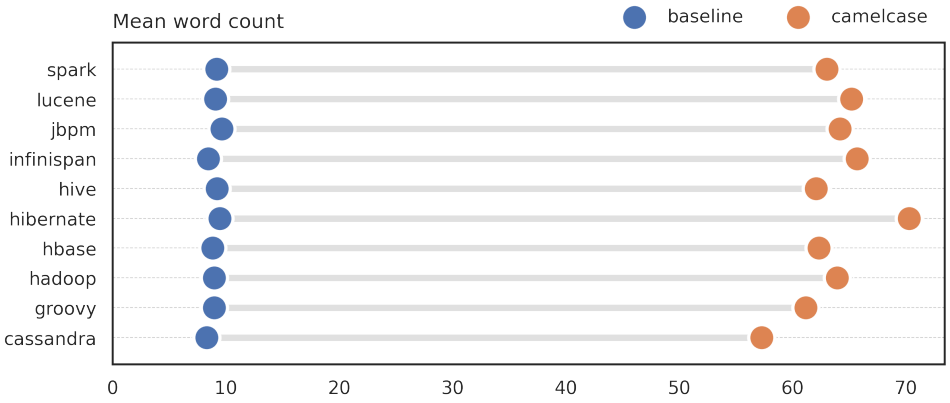


Figure 4.2.: Characteristics of the selected corpora: Number of words in the corpus, baseline processing vs camelcase split

Table 4.4.: p-values of the pairwise comparisons using the Nemenyi-Wilcoxon-Wilcox all-pairs test for a two-way balanced complete block design

	TFIDF	**Word2Vec**	**Doc2Vec**	**BERT**
Word2Vec	<0.001			
Doc2Vec	<0.001	0.761		
BERT	<0.001	0.899	0.998	
SBERT	0.524	<0.001	<0.001	<0.001

(a) Accuracy@1

	TFIDF	**Word2Vec**	**Doc2Vec**	**BERT**
Word2Vec	0.013			
Doc2Vec	0.019	1.000		
BERT	<0.001	<0.001	<0.001	
SBERT	0.581	0.432	0.505	<0.001

(b) Accuracy@BMU

	TFIDF	**Word2Vec**	**Doc2Vec**	**BERT**
Word2Vec	<0.001			
Doc2Vec	<0.001	0.124		
BERT	<0.001	0.777	0.004	
SBERT	0.038	<0.001	<0.001	<0.001

(c) Precision

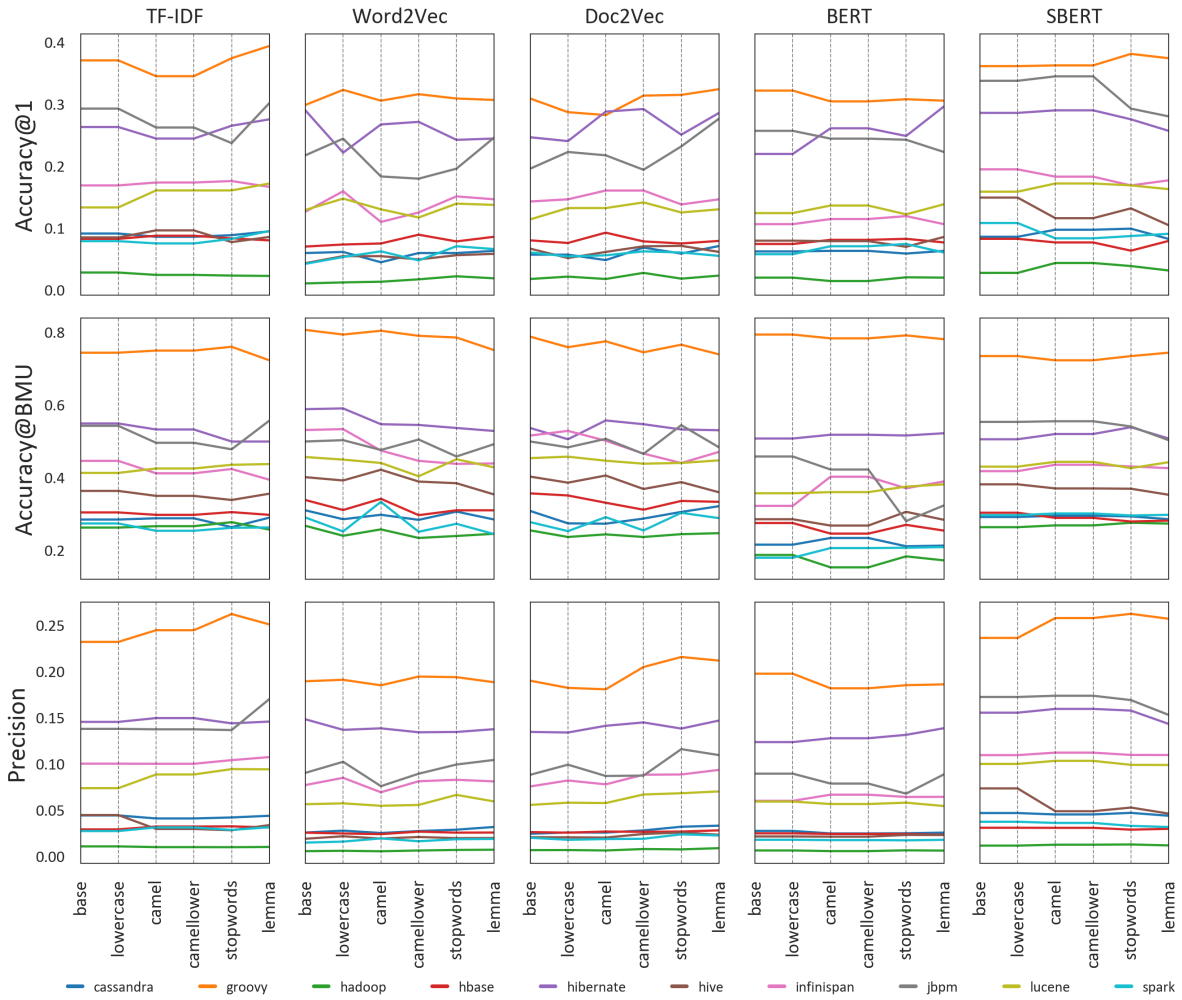


Figure 4.3.: Parallel Coordinates Plots, Accuracy and Precision

4.1.6. Quantization Error and Topographic Error

The results of the Friedman test in Table 4.5 show that the differences in ranks between the different vectorization methods are statistically significant.

Table 4.5.: Results of Friedman Rank Sum Test: TE and QE

Metric	Friedman chi-squared	p-value
Topographic Error	195.880	<0.001
Quantization Error	209.920	<0.001

From Table 4.6a we can observe that the difference between SBERT and Word2Vec is statistically significant at $\alpha = 0.05$, and the difference between Word2Vec and TF-IDF is not significant. Meanwhile from Table 4.6b the differences between Doc2Vec and BERT, as well as between SBERT and BERT, are not statistically significant.

Table 4.6.: p-values of the pairwise comparisons using the Nemenyi-Wilcoxon-Wilcox all-pairs test for a two-way balanced complete block design

	TFIDF	Word2Vec	Doc2Vec	BERT
Word2Vec	0.637			
Doc2Vec	<0.001	<0.001		
BERT	<0.001	<0.001	0.005	
SBERT	<0.001	0.044	<0.001	<0.001

(a) Quantization Error

	TFIDF	Word2Vec	Doc2Vec	BERT
Word2Vec	<0.001			
Doc2Vec	<0.001	<0.001		
BERT	<0.001	<0.001	0.562	
SBERT	<0.001	<0.001	0.006	0.314

(b) Topographic Error

Observing Tables 4.7 and 4.8, we can see that SOM trained with vectors generated from BERT and Doc2Vec produce maps with the best and second-best values for QE. Looking at the TE, TF-IDF produces the vectors that generate the worst values, while Word2Vec produces the best. Figure 4.4 shows the ranks of TE and QE values obtained by each vectorization method.

From Figures 4.5 and B5 we can see that TF-IDF, BERT and SBERT have values of QE that appear consistent

Table 4.7.: Mean Values of Ranks: TE and QE

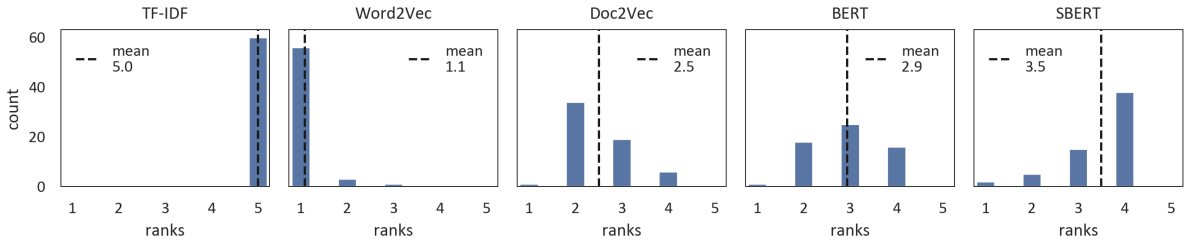
Metric	TFIDF	W2V	D2V	BERT	SBERT
Quantization Error	4.533	4.133	2.000	1.000	3.333
Topographic Error	5.000	1.083	2.500	2.933	3.483

Table 4.8.: Mean Values of Scores: TE and QE

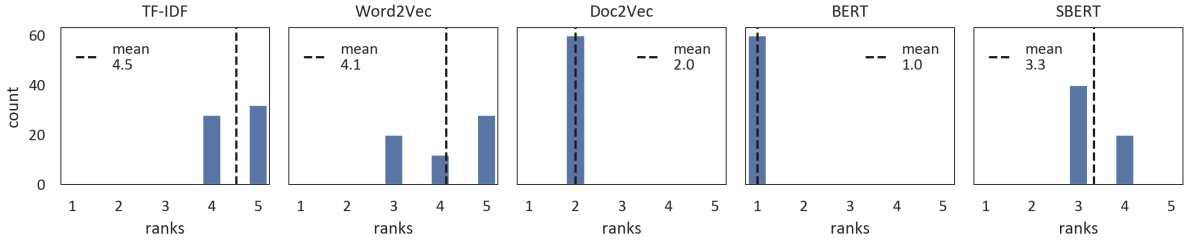
Metric	TFIDF	W2V	D2V	BERT	SBERT
Quantization Error	0.914	0.937	0.482	0.289	0.761
Topographic Error	0.644	0.369	0.429	0.445	0.462

across different preprocessing approaches. Further, we can observe that Word2Vec seems to display the largest variance in the resulting QE relative to the preprocessing applied to the datasets. From Figure B4, TF-IDF appears to show the largest values and variability for TE.

Neither TE nor QE consider the correctness of the `assigned_user` labels when being calculated; these two metrics depend only on the vectors representing the documents being organized. Based on the results of the TE, we can see that the SOM is best able to preserve the topology of the document vectors produced by Word2Vec. Meanwhile, SOM is able to produce prototype vectors that are most representative of their input data when using document vectors produced by BERT and Doc2Vec.



(a) Topographic Error



(b) Quantization Error

Figure 4.4.: Count of Ranks: Topographic and Quantization Error
Values in between whole numbers (e.g. 1.5) correspond to ties.

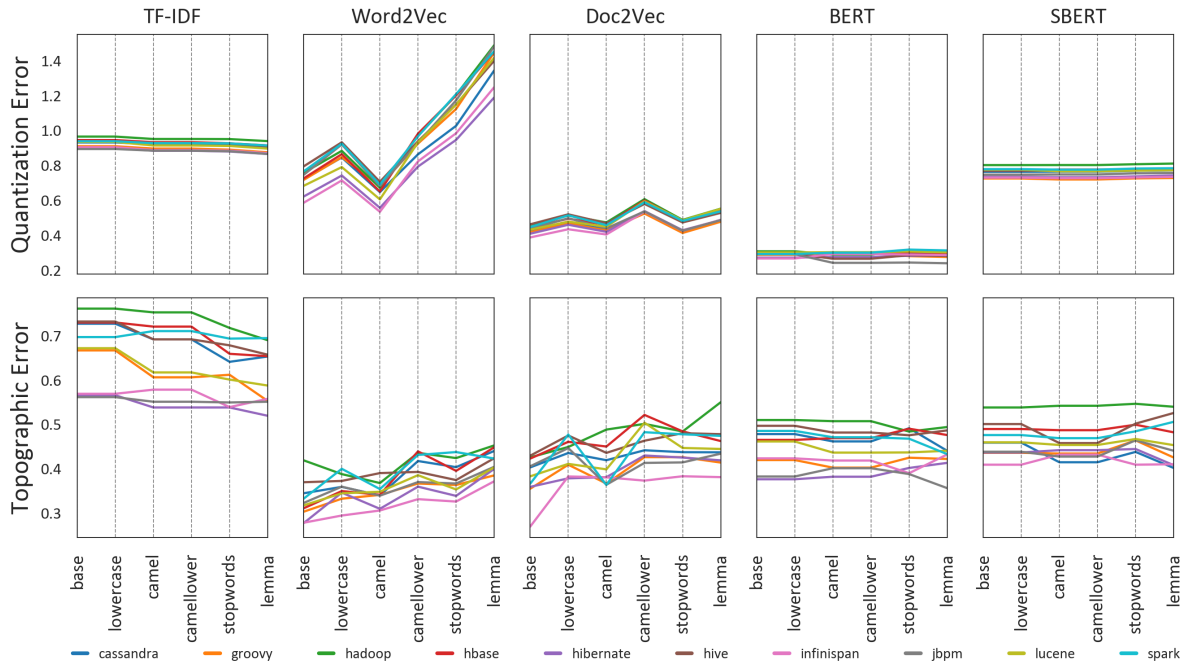


Figure 4.5.: Parallel Coordinates Plots, Quantization and Topographic Error

4.2. Effect of Different Map Initialization Conditions

The tests reported previously were performed on a SOM trained with the parameters in Table 3.5, selected based on default settings and rules of thumb mentioned previously. We also performed experiments using different initialization conditions, summarized in Table 4.9. The map size of 15×15 was maintained across all experiments. Descriptions of the parameters are given in Listing A1.

Table 4.9.: SOM Initialization

Label	Epochs	Learning Rate	Random Seed	Initialization
Base	10 000	0.7	0	random
Random2	10 000	0.7	2733	random
Random3	10 000	0.7	9846	random
Random4	10 000	0.7	3265	random
Random5	10 000	0.7	4860	random
PCA	10 000	0.7	0	pca
LR1	10 000	1	0	random

Table 4.10.: SOM Training Epochs

Label	Epochs	Learning Rate	Random Seed	Initialization
Base	10 000	0.7	0	random
150k	150 000	0.7	0	random
50k	50 000	0.7	0	random

In addition to varying these map initialization parameters, we also tested running the Base configuration with 50 000 and 150 000 epochs (Table 4.10).

Figure 4.6 shows the average ranks that each vectorization method attained under various initialization conditions. While the average values are not the same across initialization conditions, they maintain their relative positions. Figures B6 and B7 shows all the p-values of all the pair-wise comparisons across the different initializations, and Figure 4.7 shows the summary of the significance of the pairwise differences. We can observe that the pairs that show highly significant differences seem to persist across initializations with few exceptions:

For $Accuracy@1$, the difference between SBERT and TF-IDF is not significant in 6 cases, and significant only at $\alpha = 0.05$ at 150 000 epochs, as well as in two differing random seeds. TF-IDF is highly significantly different from BERT 7 times, changing to a different significance level with a different LR and one of the different random seeds.

For $Accuracy@BMU$, SBERT and Doc2Vec are not significantly different in 8 cases, becoming significant

at $\alpha = 0.01$ only at a different LR. The difference between SBERT and Word2Vec is significant only when the PCA initialization is used.

The differences compared to TF-IDF show the greatest variability in significance. Comparing with Doc2Vec, a different LR and a different random seed each account for significance at $\alpha = 0.01$, and two other random seeds, including the base case of 0, account for significance at $\alpha = 0.05$, while the remaining 4 cases show no significant difference.

Comparing with Word2Vec, 4 cases are significant at $\alpha = 0.01$, and a further 3 at $\alpha = 0.05$, including the base case and two random seeds. Using 50 000 epochs and a different LR each account for a case showing no significant difference.

For $Precision$, BERT and Doc2Vec are significantly different at $\alpha = 0.05$ most of the time. Two cases are SD at $\alpha = 0.01$: the base case, and a different random seed. Three cases show no significant difference: two of which are from different random seeds, and one from a different LR. The differences between SBERT and TF-IDF are either not significant (4 cases), or significant at $\alpha = 0.05$.

The QE metric shows the least amount of variability in the differences in significance. SBERT and Word2Vec are significantly different at $\alpha = 0.05$, with the two exceptions showing no significance resulting from using 50 000 and 150 000 iterations.

For TE, BERT and Doc2Vec are significantly different at $\alpha = 0.05$ in only one case: using 50 000 epochs, and the difference is not significant otherwise. The difference between SBERT and Doc2Vec is significant in all cases, however they vary in the level of significance.

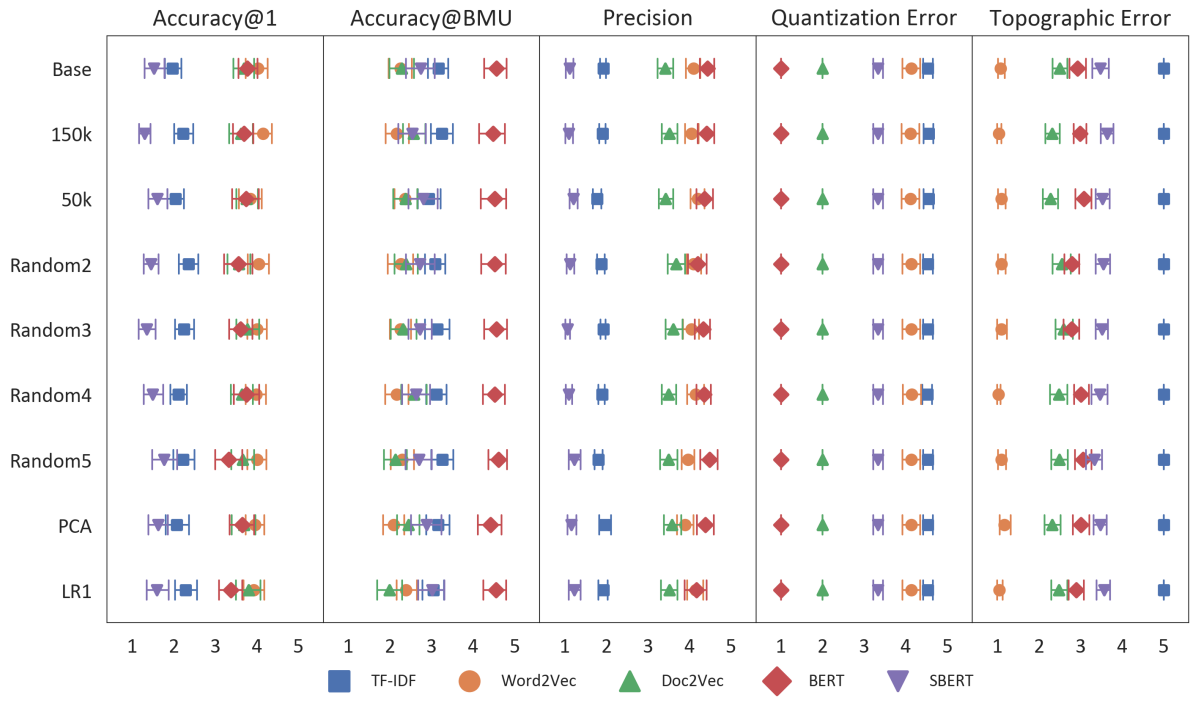


Figure 4.6.: Rank Values Using Different SOM Initializations

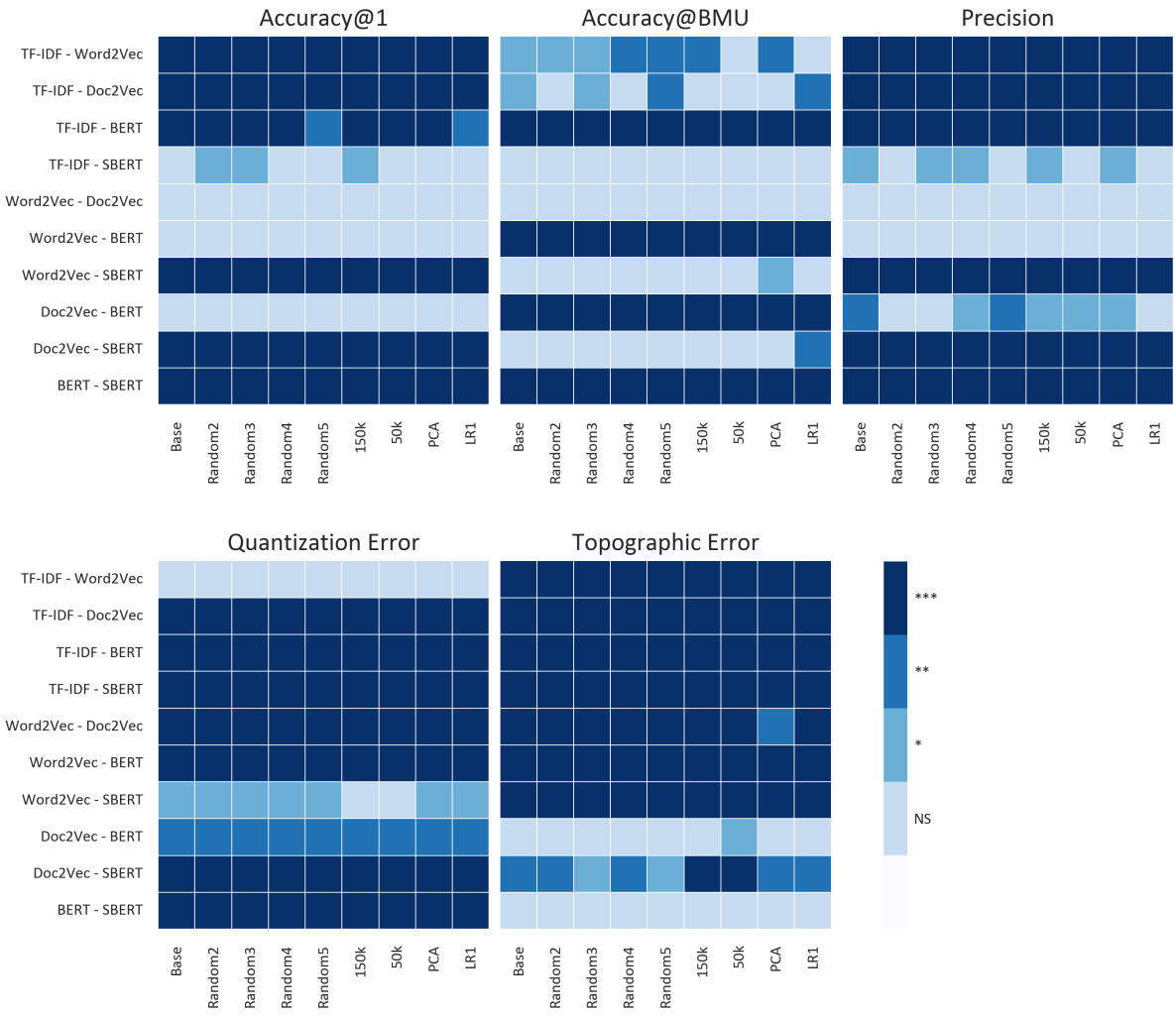


Figure 4.7.: Significance of Pairwise Differences

4.3. SOMBR: A Visual Approach to BR Triage Using SOM

In this section we present SOMBR: Self-Organizing Maps for Bug Report Triage, the interactive dashboard that we developed to demonstrate how SOM can be used for the task of BR triage.

The examples provided in this section are all based on the spark BR dataset. The corpus was filtered according to the rules discussed in Section 3.4, and baseline preprocessing was applied to it. The "Base" parameters in Table 3.5 were used to train the SOM using all valid BR. A link to the deployed dashboard is available at github.com/fpontejos/sombr-thesis.

The main visual feature of the dashboard (Figure 4.8) is the SOM grid map, highlighted in Figure 4.9a. The SOM was trained using a hexagonal lattice structure, and each SOM node is accordingly represented with a hexagonal marker. The default view shows the UMatrix representation of the SOM, with the fill color of the markers encoding the node's average distance to its neighbors. We used the Viridis color palette to calculate the colors because it is a perceptually uniform color scheme in addition to being grayscale- and colorblind-friendly (Garnier et al., 2023).

An additional hexagon marker with a white fill is superimposed on top of the main hexagon markers. The radii of these markers (denoted as the "hits hexagons") encode the number of documents located in the corresponding SOM node, scaled relative to the largest value.

The QE hexagons are markers with fill colors encoding the QE for that node (Figure 4.9d). We use the Magma color palette (Garnier et al., 2023) to calculate the values of the colors to use. This palette was chosen as it shares the properties of the Viridis palette of being perceptually uniform, grayscale- and colorblind-friendly.

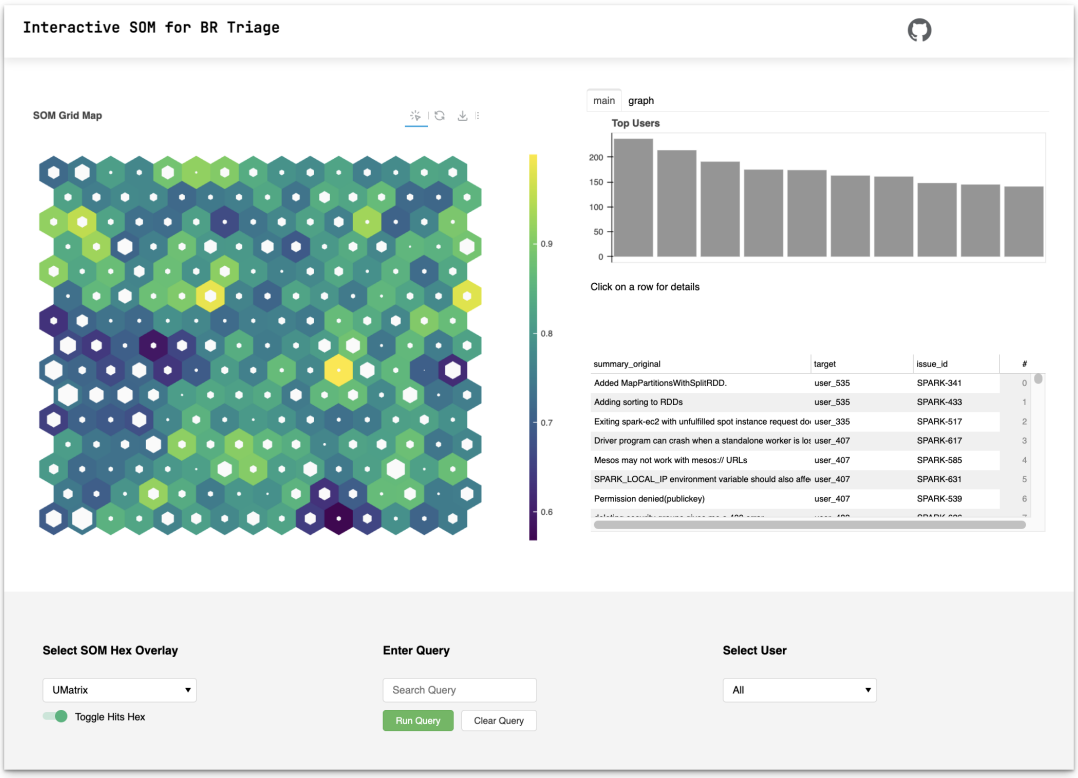
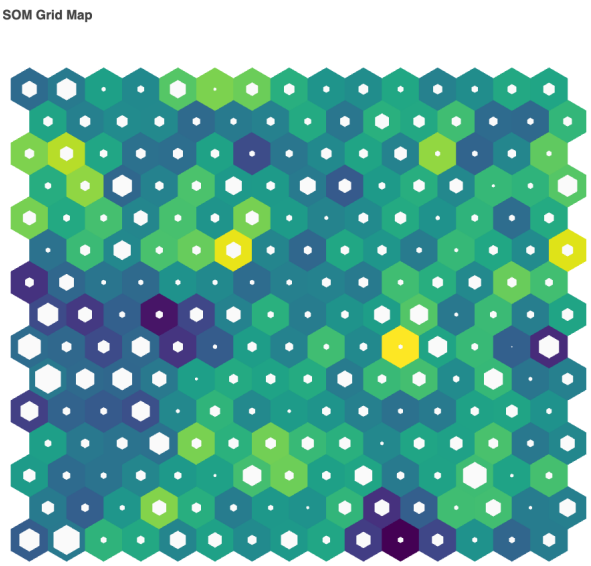
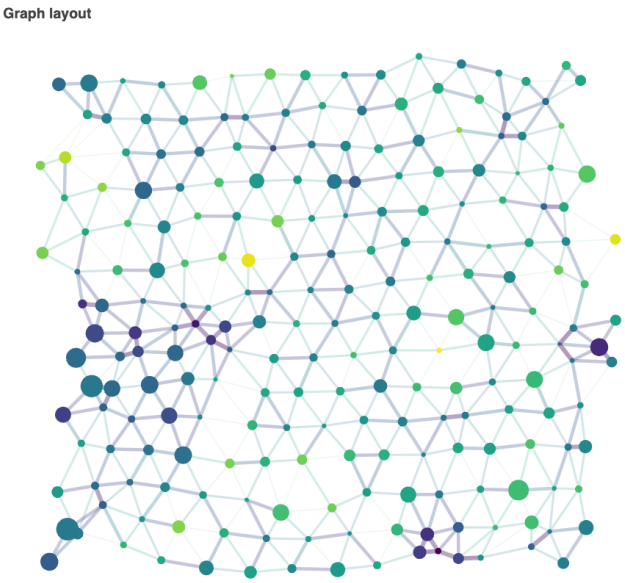


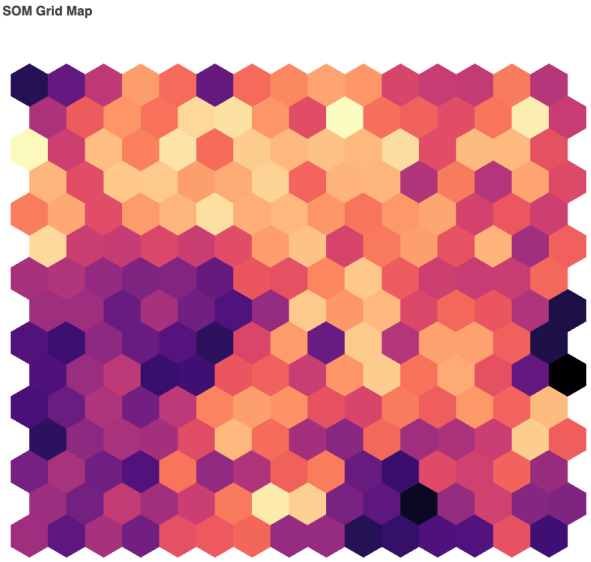
Figure 4.8.: Screenshot of dashboard



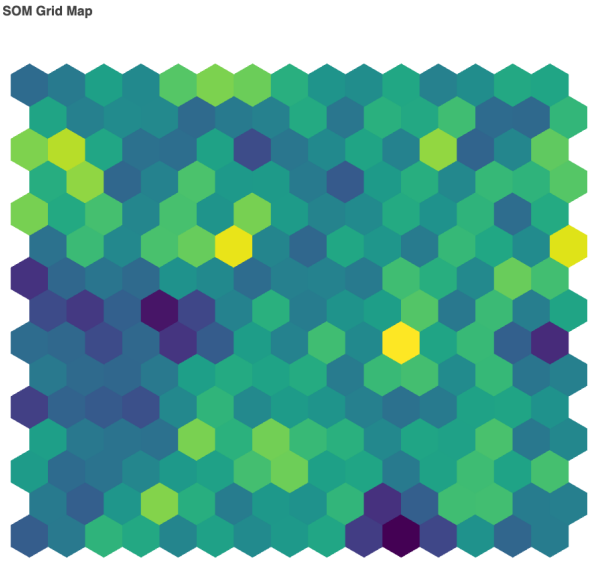
(a) Distance Matrix



(b) Graph Layout



(c) Quantization Error



(d) Hits toggled off

Figure 4.9.: SOM Visualization

4.3.1. Interaction Elements

The Hits hexagons can be toggled on or off, and the UMatrix hexagons can be replaced with the QE hexagons. These can be controlled using the toggle button and selection input elements respectively, located in the left-most panel of the dashboard footer shown in Figure 4.10.

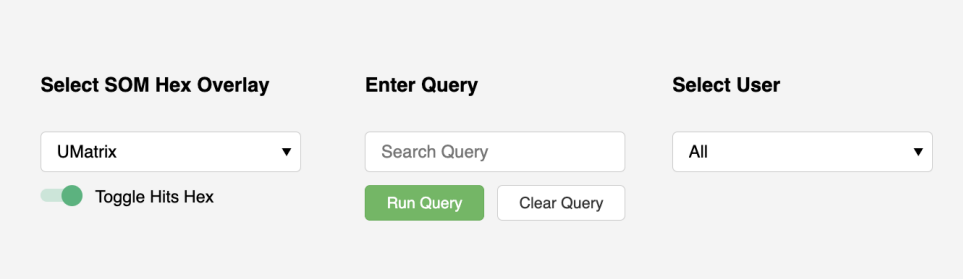


Figure 4.10.: Interactive Inputs

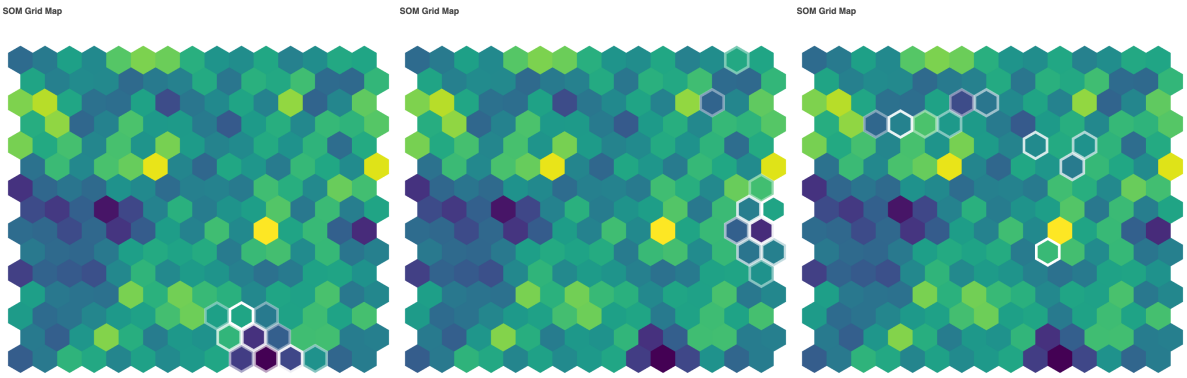
Users can type in a text query into the input element located in the middle panel of the dashboard footer in Figure 4.10. Toggling "Hits Hex" off, typing "hive" into the input box, and clicking on the "Run Query" button results in Figure 4.11a. The top 10 BMU are highlighted with a white border, whose value for opacity decreases the further away from being the top 1 BMU it is. Running this query results in neighboring hexagons being highlighted, indicating that the BR belonging to the units in this region may be related to Hive. Indeed, inspecting the BRs of the top 1 and top 2 units we can observe that they both contain Hive-related issues (Figure 4.13). These two units also have some overlap in the users with the most bugs fixed: `user_403` is the top 1 bug-fixer in these two units, and users 191, 172, 519, 579, 266, and 371 have fixed bugs on both. Clicking on "Clear Query" removes the highlighting of the results.

The query "parquet" also results in a tight grouping of top-10 BMU, located in a different region to "hive" (Figure 4.11b). This can be expected because these two topics have little in common: "parquet" refers to Apache Parquet[1], a data format, while "hive" in this context refers to Apache Hive[2], a data warehouse system.

We can contrast these results with running the query "bug", which is a more generic term in the context of BRs. This results in the top-10 BMUs being more spread out across the map (Figure 4.11c).

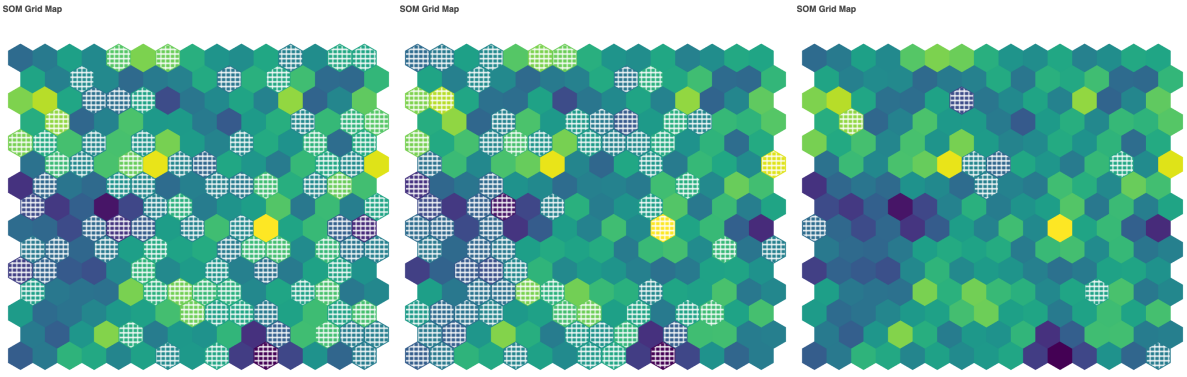
Selecting a username from the right-most panel (Figure 4.10) highlights the units where the particular user has fixed bugs. We can see that while both `user_651` and `user_646` are prolific bug-fixers, their units seem to occupy different regions of the map with few overlaps. The units where `user_55` has fixed bugs are fewer, and have some overlap with the first two.

[1] https://parquet.apache.org/
[2] https://hive.apache.org/



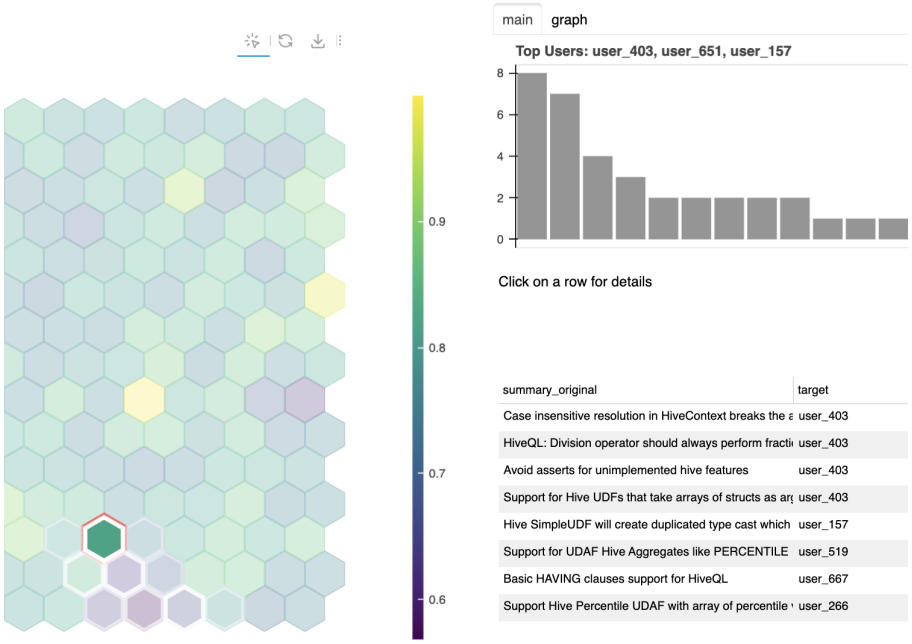
(a) Query: “hive” (b) Query: “parquet” (c) Query: “bug”

Figure 4.11.: Query Results

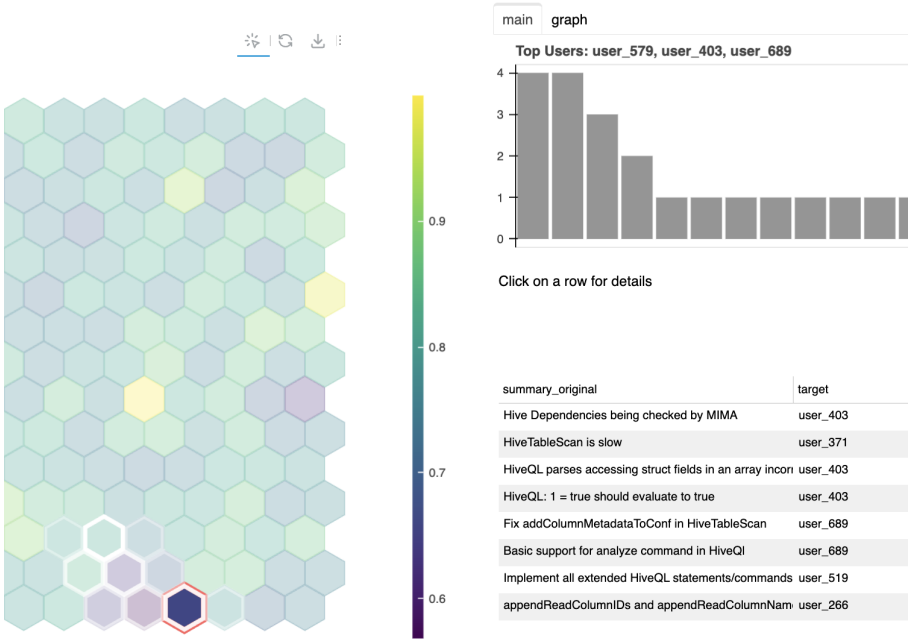


(a) Highlight user_651 (b) Highlight user_646 (c) Highlight user_55

Figure 4.12.: User Selection



(a) Query: “hive” Top 1 BMU



(b) Query: “hive” Top 2 BMU

Figure 4.13.: Querying Results for “hive”, zoomed in

4.3.2. Map Grid Interactions

In addition to the input elements discussed above, users can also interact with the hexagon markers by clicking or hovering over them. The interactions annotated in Figure 4.14 are described below:

1. Clicking on a hexagon filters the table to show only the BRs in that node
2. Hovering over a hexagon shows how many "hits" or the number of BR in that node, and the node's average distance to its neighbors
3. Clicking on a table row shows the detailed information about the selected BR
4. Selecting a hexagon shows the `assignee_user` labels of the BR in that node and their corresponding frequencies

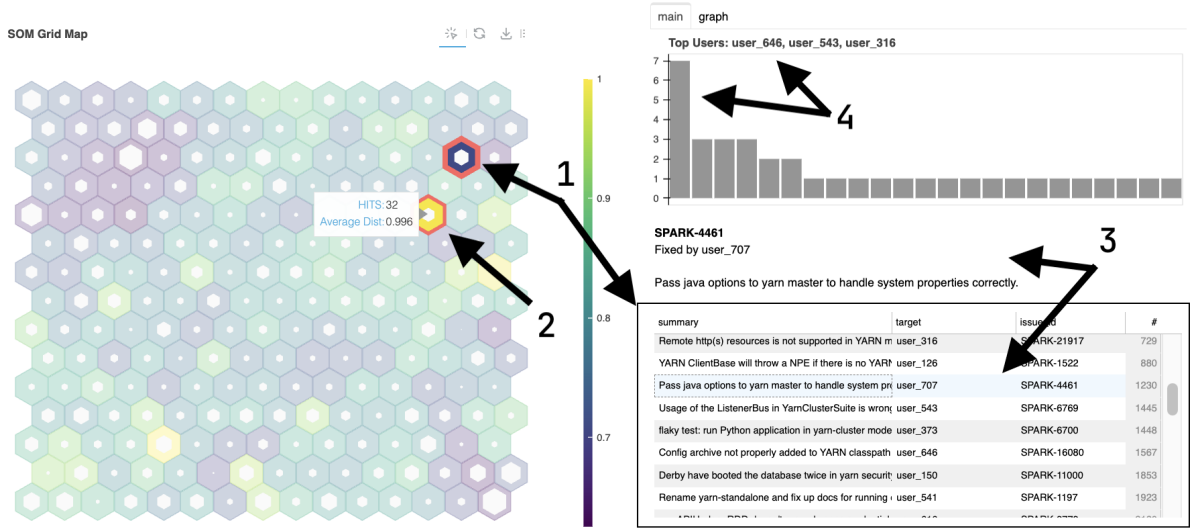


Figure 4.14.: Dashboard interactions: detail view

Selecting the "graph" tab, highlighted in Figure 4.15, allows users to inspect the graph layout view of the SOM network.

Figure 4.15.: Detail view tabs

To calculate the positions of the nodes, we used `neato`, a spring model layout program from graphviz[3]. According to its documentation, "`neato` attempts to minimize a global energy function, which is equivalent to statistical multi-dimensional scaling". We used the parameter `model = 'mds'`, which allowed us to specify a length "`len`" to use "as the ideal distance between its vertices", calculated as the individual distances of each node to each of its neighbors.

[3] https://graphviz.org/

The linked interactivity between the hexagon markers and the graph elements is highlighted in Figure 4.16 and discussed below.

1. Clicking on a hexagon highlights the corresponding graph node and its neighbors
2. Hovering over a hexagon highlights both the hexagon and the corresponding graph node in an orange outline and fill color, respectively

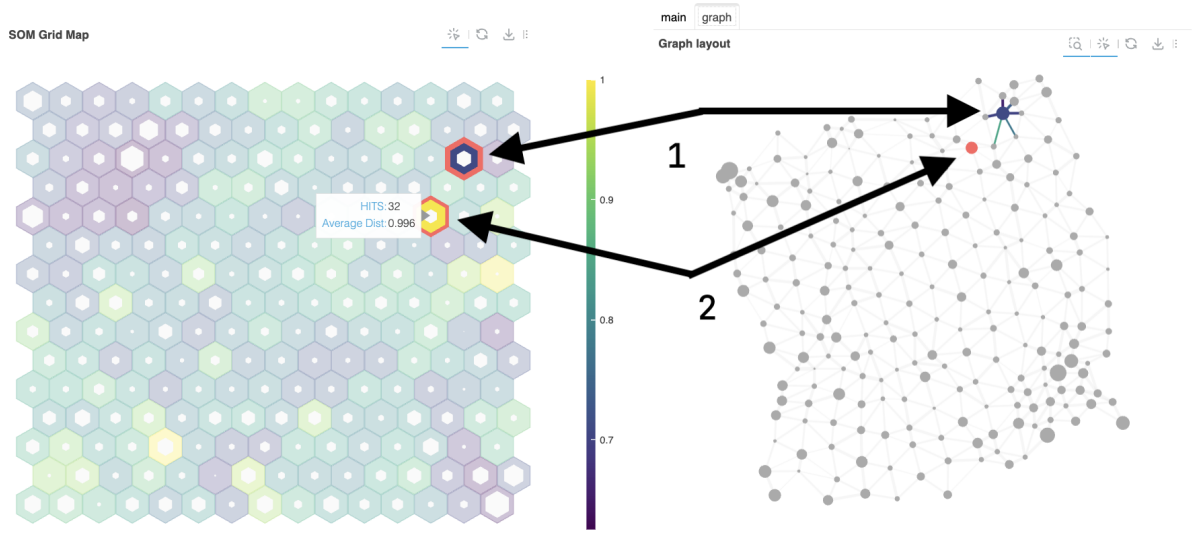


Figure 4.16.: Dashboard interactions: graph view

5. Conclusions

In this thesis we investigated the effects of using different vectorization methods to encode the textual components of BRs for the task of automated BR triage. We used SOM, an unsupervised ML algorithm, to organize the BRs into semantically related units. After determining its BMU, we used the most frequently occurring `assigned_user` value as the predicted label for each document in the Test set.

To answer **RQ 1**: we were able to use SOM to identify the correct bug fixer responsible. Obtaining the BMU for a given BR, we are able to present the user with potential bug fixers that have fixed similar bugs in the past. Measuring $Accuracy@BMU$, the algorithm was able to correctly identify up to 225% `assigned_user` labels more than when measuring $Accuracy@1$. Unlike in multi-class classification tasks, in BR triage there may be more than one developer able to fix a particular bug. In this sense the $Accuracy@BMU$ metric may be the more appropriate measure of success than $Accuracy@1$.

Addressing **RQ 2**, we found that the vectorization methods used had an effect on how well the algorithm was able to predict the correct `assigned_user`: SBERT and TF-IDF performed better than the other vectorization methods tested. Measuring the $Accuracy@BMU$ metric shows that using BERT results in poorer performance.

One of the challenges of using the textual component of BR corpora for automatic BR triage is the nature of the language being used in the BRs. In describing the defects, users use a combination of natural language, as well as programming language-specific identifiers such as function and variable names.

Based on the findings above we hypothesize that a word embedding model such as Word2Vec that has been trained on large amounts of natural language data might consider these codebase-specific naming conventions to be OOV words, and therefore not be able to capture the semantic relationships that may be present. Doc2Vec is a sentence embedding model, and while in theory it should be better at capturing the context and semantics of documents, it will similarly consider the technical terms to be outside of its training vocabulary.

While SBERT is similarly trained using natural language datasets, it may be better able to deal with such terms because of how it tokenizes documents. It breaks words into WordPieces, or character sequences prior to encoding them, making it able to capture the semantics of identifiers composed of concatenated natural language words.

The presence of these programming language constructs may also explain how TF-IDF, one of the oldest ways of representing text data, is surprisingly able to perform better than newer vectorization methods such as Word2Vec. TF-IDF does not rely on a training corpus, and builds a representation of documents based only on the Training set vocabulary, which would include these special terms. However, this is also one downside of using TF-IDF: the codebase of software projects is constantly changing, with new identifiers being added as time passes. Unless the TF-IDF model is retrained frequently, it will also suffer from a large number of words it considers OOV.

Lastly, addressing **RQ 3**: testing several different initialization parameters, we found that while the mean values of the scores and ranks change, the relative positions of the performance of the vectorization methods remained largely stable.

We were also able to develop an interactive dashboard that may help with the task of semi-automated BR triage. This dashboard allows users to input a query, for instance the summary of a new BR, which would result in the BMU for that query being highlighted. The user can then use these suggested units to investigate which users have been previously involved in fixing bugs that are semantically similar and narrow down the list of candidates considerably.

6. Limitations and Future Work

One of the challenges of the BR triage task is the large imbalance present in the data. Across all the datasets we used, we found that most users contributed between 1 to 3 bug fixes, a common scenario in community-driven OSS projects. Future studies might consider comparing the results in the context of proprietary software development, in which developer roles are more clearly defined, and bug fixers are not community volunteers. Given the potential difficulty in obtaining this data however, an alternative approach would be to further filter the same dataset to only include BRs assigned to developers who have fixed bugs greater than a particular threshold.

Related to this issue, we observed that the results of Accuracy and Precision (Figures B1, B2, and B3) seemed to display some relation to the kind of corpus being used. The dataset `groovy` for instance, seems to consistently be related to higher values relative to `hadoop`. It would be interesting to study which characteristics, if any, of the various corpora would make this approach more suitable for use.

Another potential direction for future work is to test other vectorization models, since only five were compared in this study. In addition, testing the results of pre-training or fine-tuning these models using programming language corpora might be interesting. Software documentation and BR datasets from other projects would be good candidates for this purpose. Additionally, given the recent increase in their availability, we can investigate how to leverage Large Language Models for the task of BR triage.

The dashboard presented can also be further developed for use in domains other than BR triage. General IR tasks may be similarly well suited to this visual approach.

Bibliography

Ahmed, A., & Ghazali, R. (2016). An improved self-organizing map for bugs data clustering [IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS), Shah Alam, Malaysia, OCT 22, 2016]. *2016 IEEE International Conference on Automatic Control and Intelligent Systems (i2cacis)*, 135–140.

Amati, G. (2009). Bm25. In L. LIU & M. T. ÖZSU (Eds.), *Encyclopedia of database systems* (pp. 257–260). Springer US. https://doi.org/10.1007/978-0-387-39940-9_921

Amazon Web Services, Inc. (2022). AWS Issue Triage Manager [Accessed: 20.03.2023]. https://github.com/microsoft/vscode-github-triage-actions/tree/

Amine, A., Elberrichi, Z., Bellatreche, L., Simonet, M., & Malki, M. (2008). Concept-based clustering of textual documents using som [6th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA-08), Doha, Qatar, MAR 31-APR 04, 2008]. *2008 IEEE/ACS International Conference on Computer Systems and Applications, Vols 1-3*, 156+. https://doi.org/10.1109/AICCSA.2008.4493530

Ampazis, N., & Perantonis, S. (2004). LSISOM - a latent semantic indexing approach to self-organizing maps of document collections. *Neural Processing Letters*, *19*(2), 157–173. https://doi.org/10.1023/B:NEPL.0000023449.95030.8f

Bengio, Y., Ducharme, R., Vincent, P., & Janvin, C. (2003). A Neural Probabilistic Language Model. *J. Mach. Learn. Res.*, *3*, 1137–1155.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., & Zimmermann, T. (2008). What Makes a Good Bug Report? *Proceedings of the 16th ACM Sigsoft International Symposium on Foundations of Software Engineering*, 308–318. https://doi.org/10.1145/1453101.1453146

Calefato, F., Lanubile, F., Maiorano, F., & Novielli, N. (2017). Sentiment polarity detection for software development. *Empirical Software Engineering*, *23*(3), 1352–1382. https://doi.org/10.1007/s10664-017-9546-9

Correa, R. F., & Ludermir, T. B. (2008). A quickly trainable hybrid som-based document organization system [9th Brazilian Symposium on Neural Networks, Ribeirao Preto, BRAZIL, 2006]. *Neurocomputing*, *71*(16-18, SI), 3353–3359. https://doi.org/10.1016/j.neucom.2008.02.021

Davies, S., & Roper, M. (2014). What's in a Bug Report? *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1145/2652524.2652541

de Miranda, G. R., Pasti, R., & de Castro, L. N. (2020). Detecting topics in documents by clustering word vectors [16th International Symposium on Distributed Computing and Artificial Intelligence (DCAI), Avila, SPAIN, JUN 26-28, 2019]. In F. Herrera, K. Matsui & S. RodriguezGonzalez (Eds.), *Distributed computing and artificial intelligence, 16th international conference* (pp. 235–243). https://doi.org/10.1007/978-3-030-23887-2_27

Demšar, J. (2006). Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research*, *7*(1), 1–30. http://jmlr.org/papers/v7/demsar06a.html

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.48550/ARXIV.1810.04805

Dey, S. (2010). Enhancing text cluster visualization in emergent self organizing maps by incorporating bigram vectors [3rd International Conference on Modeling, Simulation and Optimization, Beijing, Peoples R China, DEC 25-26, 2010]. In P. Yang (Ed.), *Third international conference on modeling,*

simulation and optimization (CMSO 2010) (pp. 21–26). London Science Publishing Ltd. https://www.researchgate.net/publication/302276396_Enhancing_Text_Cluster_Visualization_in_Emergent_Self_Organizing_Maps_by_Incorporating_Bigram_Vectors

Dong, Z., & Dong, Q. (2003). HowNet - a hybrid language and knowledge resource. *International Conference on Natural Language Processing and Knowledge Engineering, 2003. Proceedings. 2003*, 820–824. https://doi.org/10.1109/NLPKE.2003.1276017

do Rego, R. L. M. E., Ribeiro, M., Aleixo, E., & de Souza, R. M. C. R. (2008). Bug reports retrieval using self-organizing map [3rd International Conference on Digital Information Management, Univ E London, London, ENGLAND, NOV 13-16, 2008]. *2008 Third International Conference on Digital Information Management, Vols 1 and 2*, 327–332.

Du, X., Zheng, Z., Xiao, G., Zhou, Z., & Trivedi, K. S. (2021). Deepsim: Deep semantic information-based automatic mandelbug classification. *IEEE Transactions on Reliability*. https://doi.org/10.1109/TR.2021.3110096

Gansner, E. R., Koren, Y., & North, S. (2005). Graph drawing by stress majorization. In J. Pach (Ed.), *Graph drawing* (pp. 239–250). Springer Berlin Heidelberg.

Garnier, Simon, Ross, Noam, Rudis, Robert, Camargo, Pedro, A., Sciaini, Marco, Scherer & Cédric. (2023). *viridis(Lite) - colorblind-friendly color maps for r* [viridis package version 0.6.3]. https://doi.org/10.5281/zenodo.4679423

GitHub, Inc. (2023a). Features • GitHub Actions [Accessed: 20.03.2023]. https://github.com/features/actions

GitHub, Inc. (2023b). Saving repositories with stars - GitHub Docs. https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars

Guan, H., Zhou, J., & Guo, M. (2009). A Class-Feature-Centroid Classifier for Text Categorization. *Proceedings of the 18th International Conference on World Wide Web*, 201–210. https://doi.org/10.1145/1526709.1526737

He, J., Xu, L., Yan, M., Xia, X., & Lei, Y. (2020). Duplicate Bug Report Detection Using Dual-Channel Convolutional Neural Networks. *Proceedings of the 28th International Conference on Program Comprehension*, 117–127. https://doi.org/10.1145/3387904.3389263

Hu, D., Chen, M., Wang, T., Chang, J., Yin, G., Yu, Y., & Zhang, Y. (2018). Recommending Similar Bug Reports: A Novel Approach Using Document Embedding Model. *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, 725–726. https://doi.org/10.1109/APSEC.2018.00108

Jonsson, L., Borg, M., Broman, D., Sandahl, K., Eldh, S., & Runeson, P. (2016). Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering*, *21*(4), 1533–1578. https://doi.org/10.1007/s10664-015-9401-9

Jurafsky, D., & Martin, J. H. (2023). *Speech and Language Processing: An Introduction to Natural Language Processing,Computational Linguistics, and Speech Recognition* (3rd) [DRAFT].

Just, S., Premraj, R., & Zimmermann, T. (2008). Towards the next generation of bug tracking systems. *2008 IEEE Symposium on Visual Languages and Human-centric Computing*, 82–85. https://doi.org/10.1109/VLHCC.2008.4639063

Kaski, S., Honkela, T., Lagus, K., & Kohonen, T. (1998). WEBSOM - self-organizing maps of document collections. *Neurocomputing*, *21*(1-3), 101–117. https://doi.org/10.1016/S0925-2312(98)00039-3

Kiviluoto, K. (1996). Topology preservation in self-organizing maps. *Proceedings of International Conference on Neural Networks (ICNN'96)*, *1*, 294–299 vol.1.

Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE*, *78*(9), 1464–1480. https://doi.org/10.1109/5.58325

Kubernetes. (2022). Issue Triage Guidelines [Accessed: 15.04.2023]. https://www.kubernetes.dev/docs/guide/issue-triage/

Lang, K. (1995). NewsWeeder: Learning to Filter Netnews. In A. Prieditis & S. Russell (Eds.), *Machine learning proceedings 1995* (pp. 331–339). Morgan Kaufmann. https://doi.org/10.1016/B978-1-55860-377-6.50048-7

Le, Q., & Mikolov, T. (2014). Distributed Representations of Sentences and Documents. In E. P. Xing & T. Jebara (Eds.), *Proceedings of the 31st International Conference on Machine Learning* (pp. 1188–1196). Pmlr. https://proceedings.mlr.press/v32/le14.html

Lin, M.-J., Yang, C.-Z., Lee, C.-Y., & Chen, C.-C. (2016). Enhancements for duplication detection in bug reports with manifold correlation features. *Journal of Systems and Software*, *121*, 223–233. https://doi.org/10.1016/j.jss.2016.02.022

Liu, Y., Wang, X., & Wu, C. (2008). Consom: A conceptional self-organizing map model for text clustering. *Neurocomputing*, *71*(4-6), 857–862. https://doi.org/10.1016/j.neucom.2007.03.006

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Merkl, D. (1998). Text classification with self-organizing maps: Some lessons learned. *Neurocomputing*, *21*(1-3), 61–77. https://doi.org/10.1016/S0925-2312(98)00032-0

Microsoft Corporation. (2022a). Issues Triaging [Accessed: 20.03.2023]. https://github.com/microsoft/vscode/wiki/Issues-Triaging/

Microsoft Corporation. (2022b). VSCode GitHub Triage Actions [Accessed: 20.03.2023]. https://github.com/microsoft/vscode-github-triage-actions/tree/

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. https://doi.org/10.48550/ARXIV.1301.3781

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D., & Miller, K. J. (1990). Introduction to WordNet: An on-line lexical database. *International Journal of Lexicography*, *3*(4), 235–244.

Nagwani, N. K., & Suri, J. S. (2023). An artificial intelligence framework on software bug triaging, technological evolution, and future challenges: A review. *International Journal of Information Management Data Insights*, *3*(1), 100153. https://doi.org/10.1016/j.jjimei.2022.100153

Naseem, U., Razzak, I., Khan, S. K., & Prasad, M. (2021). A Comprehensive Survey on Word Representation Models: From Classical to State-of-the-Art Word Representation Language Models. *ACM Trans. Asian Low-resour. Lang. Inf. Process.*, *20*(5). https://doi.org/10.1145/3434237

Onan, A., & Tocoglu, M. A. (2021). Weighted word embeddings and clustering-based identification of question topics in mooc discussion forum posts. *COMPUTER APPLICATIONS IN ENGINEERING EDUCATION*, *29*(4, SI), 675–689. https://doi.org/10.1002/cae.22252

Pennington, J., Socher, R., & Manning, C. (2014). GloVe: Global Vectors for Word Representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543. https://doi.org/10.3115/v1/D14-1162

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations.

Platt, J. (1998). Fast training of support vector machines using sequential minimal optimization. In B. Schoelkopf, C. Burges & A. Smola (Eds.), *Advances in Kernel Methods - Support Vector Learning*. Mit Press. http://research.microsoft.com/%5C~jplatt/smo.html

Pölzlbauer, G. (2004). Survey and Comparison of Quality Measures for Self-Organizing Maps. In J. Paralič, G. Pölzlbauer & A. Rauber (Eds.), *Proceedings of the fifth workshop on data analysis (wda'04)* (pp. 67–82). Elfa Academic Press.

Pullwitt, D. (2002). Integrating contextual information to enhance SOM-based text document clustering. *Neural Networks*, *15*(8-9), 1099–1106. https://doi.org/10.1016/S0893-6080(02)00082-5

Pullwitt, D., & Der, R. (2001). Integrating contextual information into text document clustering with self-organizing maps [3rd Workshop on Self-organising Maps (wsom), Univ Lincolnshire & Humberside, Kingston Hull, England, Jun 13-15, 2001]. In N. Allinson, H. Yin, L. Allinson & J. Slack (Eds.), *Advances in self-organising maps* (pp. 54–60). Springer-verlag London Ltd. https://doi.org/10.1007/978-1-4471-0715-6_8

Rajbhandari, A., Zibran, M. F., & Eishita, F. Z. (2022). Security Versus Performance Bugs: How Bugs are Handled in the Chromium Project. *2022 IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA)*, 70–76. https://doi.org/10.1109/SERA54885.2022.9806745

Ramos, J. (2003). Using tf-idf to determine word relevance in document queries. *Proceedings of the first instructional conference on machine learning*, *242*(1), 29–48.

RaRe-Technologies. (2017). Release wiki-english-20171001 · RaRe-Technologies/gensim-data. https://github.com/RaRe-Technologies/gensim-data/releases/tag/wiki-english-20171001

Rath, M., & Mäder, P. (2019a). The SEOSS 33 dataset — Requirements, bug reports, code history, and trace links for entire projects. *Data in Brief*, *25*, 104005. https://doi.org/10.1016/j.dib.2019.104005

Rath, M., & Mäder, P. (2019b). *The SEOSS Dataset - Requirements, Bug Reports, Code History, and Trace Links for Entire Projects*. https://doi.org/10.7910/DVN/PDDZ4Q

Řehůřek, R. (2022). Doc2vec paragraph embeddings. https://radimrehurek.com/gensim/models/doc2vec.html

Reimers, N. (2022). Pretrained Models — Sentence-Transformers documentation. https://www.sbert.net/docs/pretrained_models.html

Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. https://doi.org/10.48550/ARXIV.1908.10084

Reuters-21578 Text Categorization Collection. (1997).

Sajedi-Badashian, A., & Stroulia, E. (2020). Guidelines for evaluating bug-assignment research. *Journal of Software-evolution and Process*, *32*(9). https://doi.org/10.1002/smr.2250

Sanderson, M., & Croft, W. B. (2012). The History of Information Retrieval Research. *Proceedings of the IEEE*, *100*(Special Centennial Issue), 1444–1451. https://doi.org/10.1109/JPROC.2012.2189916

Stefanovic, P., & Kurasova, O. (2014). Creation of text document matrices and visualization by self-organizing map. *Information Technology and Control*, *43*(1), 36–45. https://doi.org/10.5755/j01.itc.43.1.4299

Tahir, H., Khan, S. U. R., & Ali, S. S. (2021). LCBPA: An enhanced deep neural network-oriented bug prioritization and assignment technique using content-based filtering. *IEEE Access*, *9*, 92798–92814. https://doi.org/10.1109/ACCESS.2021.3093170

Tian, Y., Lo, D., & Sun, C. (2012). Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction. *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, 215–224. https://doi.org/10.1109/WCRE.2012.31

Till, B. C., Longo, J., Dobell, A. R., & Driessen, P. F. (2014). Self-organizing maps for latent semantic analysis of free-form text in support of public policy analysis. *Wiley Interdisciplinary Reviews-data Mining and Knowledge Discovery*, *4*(1), 71–86. https://doi.org/10.1002/widm.1112

van Rossum, G., Warsaw, B., & Coghlan, N. (2001, July). PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. https://doi.org/10.48550/ARXIV.1706.03762

Vesanto, J. (2003). SOM Toolbox. http://www.cis.hut.fi/projects/somtoolbox/package/docs2/somtoolbox.html

Vettigli, G. (2018). MiniSom: Minimalistic and NumPy-based implementation of the Self Organizing Map. https://github.com/JustGlowing/minisom/

Wang, W., Wei, F., Dong, L., Bao, H., Yang, N., & Zhou, M. (2020). MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, Ł., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., … Dean, J. (2016). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation.

Yan, J. (2009). Text Representation. In L. Liu & M. T. Özsu (Eds.), *Encyclopedia of database systems* (pp. 3069–3072). Springer US. https://doi.org/10.1007/978-0-387-39940-9_420

Yang, X., Lo, D., Xia, X., Bao, L., & Sun, J. (2016). Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports. *2016 IEEE 27th International Symposium on Software Reliability Engineering (issre)*, 127–137. https://doi.org/10.1109/ISSRE.2016.33

Yoshioka, K., & Dozono, H. (2019). The classification of the documents based on word embedding and 2-layer spherical self organizing maps [11th International Conference on Machine Learning and Computing (ICMLC), Zhuhai, Peoples R China, FEB 22-24, 2019]. *ICMLC 2019: 2019 11th International Conference on Machine Learning and Computing*, 357–361. https://doi.org/10.1145/3318299.3318378

Zaidi, S. F. A., Awan, F. M., Lee, M., Woo, H., & Lee, C.-G. (2020). Applying convolutional neural networks with different word representation techniques to recommend bug fixers. *IEEE Access*, *8*, 213729–213747. https://doi.org/10.1109/ACCESS.2020.3040065

Zaman, S., Adams, B., & Hassan, A. E. (2011). Security versus performance bugs: A case study on firefox. *Proceedings of the 8th Working Conference on Mining Software Repositories*, 93–102. https://doi.org/10.1145/1985441.1985457

Zhang, T., Chen, J., Yang, G., Lee, B., & Luo, X. (2016). Towards more accurate severity prediction and fixer recommendation of software bugs. *Journal of Systems and Software*, *117*, 166–184. https://doi.org/10.1016/j.jss.2016.02.034

Appendix A

Table A1.: Description of attributes: `issue` table

Attribute	Description
`issue_id`	A unique identifier for the record.
`type`	The kind of issue being reported. Possible values may vary across projects. Some common ones include: `Bug`, `Documentation`, `Improvement`, `New Feature`, `Request`, `Question`
`status`	Possible values include: `Resolved`, `In Progress`, `Open`, `Closed`, `Reopened`
`resolution`	Possible values include: `Won't Fix`, `Fixed`, `Not A Problem`, `Done`, `Duplicate`, `Invalid`, `Resolved`, `Later`, `Cannot Reproduce`, `Incomplete`, `Unresolved`, `Workaround`, `Not A Bug`, `Implemented`, `Auto Closed`, `Information Provided`, `Works for Me`, `Won't Do`,
`summary`	A short description of the issue being reported.
`description`	A more detailed description of the issue being reported
`assignee_username`	The user assigned to resolve the issue.

Table A2.: Description of attributes: `issue_link` table

Attribute	Description
`source_issue_id` `target_issue_id`	The `outward_label` attribute describes how the `source_issue_id` is linked to the `target_issue_id`.
`outward_label`	For example: `SPARK-15685 duplicates SPARK-4783`

Table A3.: Entry for issue SPARK-19748

* Personally identifiable information has been obscured

Attribute	Value
issue_id	SPARK-19748
type	Bug
created_date	2017-02-27T07:43:25Z
created_date_zoned	2017-02-27T07:43:25Z
updated_date	2017-02-28T08:18:23Z
updated_date_zoned	2017-02-28T08:18:23Z
resolved_date	2017-02-28T08:17:55Z
resolved_date_zoned	2017-02-28T08:17:55Z
summary	refresh for InMemoryFileIndex with FileStatusCache does not work correctly
priority	Major
status	Resolved
resolution	Fixed
assignee	*
assignee_username	*
reporter	*
reporter_username	*

```
If we refresh a InMemoryFileIndex with a FileStatusCache,
   ↪ it will first use the FileStatusCache to generate the
   ↪  cachedLeafFiles etc, then call FileStatusCache.
   ↪ invalidateAll. the order to do these two actions is
   ↪ wrong, this lead to the refresh action does not take
   ↪ effect.

{code}
  override def refresh(): Unit = {
    refresh0()
    fileStatusCache.invalidateAll()
  }

  private def refresh0(): Unit = {
    val files = listLeafFiles(rootPaths)
    cachedLeafFiles =
      new mutable.LinkedHashMap[Path, FileStatus]() ++=
         ↪ files.map(f => f.getPath -> f)
    cachedLeafDirToChildrenFiles = files.toArray.groupBy(_.
       ↪ getPath.getParent)
    cachedPartitionSpec = null
  }
{code}
```

Listing A1: Description for issue SPARK-19748

```
SELECT
    issue_id,
    type,
    created_date,
    resolved_date,
    summary,
    description,
    priority,
    status,
    resolution,
    assignee_username
FROM issue
WHERE
    issue.type = 'Bug'
    AND issue.resolution in ('Done', 'Fixed', 'Resolved')
    AND issue.status in ('Resolved', 'Closed')
    AND issue.assignee_username <> ''
```

Listing A2: SQL Query to retrieve issues from `issue` table

```
SELECT
    source_issue_id,
    outward_label,
    target_issue_id
FROM issue_link as issue_link
INNER JOIN (
SELECT
    issue_id,
    type,
    created_date,
    resolved_date,
    summary,
    description,
    priority,
    status,
    resolution,
    assignee_username
FROM issue
WHERE
    issue.type = 'Bug'
    AND issue.resolution in ('Done', 'Fixed', 'Resolved')
    AND issue.status in ('Resolved', 'Closed')
    AND issue.assignee_username <> ''
) as main_query
ON main_query.issue_id = issue_link.target_issue_id
WHERE issue_link.outward_label = 'duplicates'
```

Listing A3: This represents all issues that duplicates a target issue on the main query according to its outward_label field

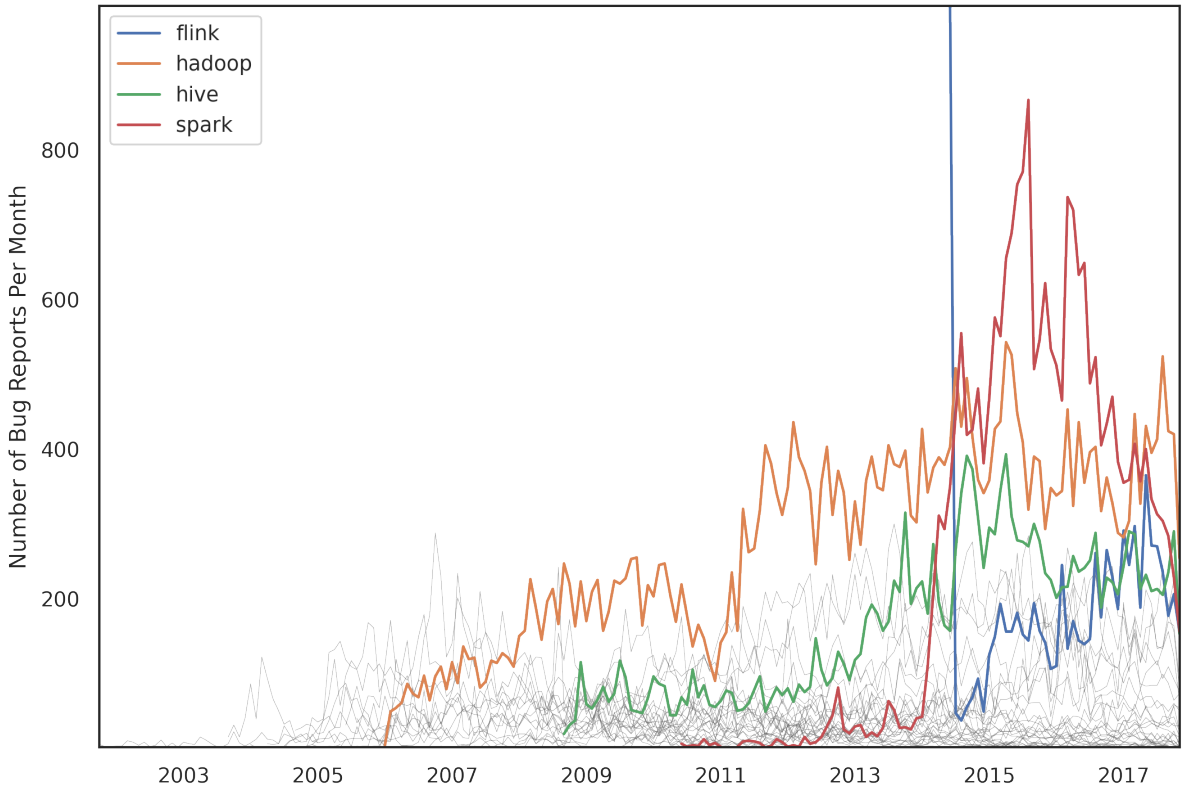


Figure A1.: Number of Bug Reports Created per Month
The highlighted items represent the projects that exceededed 300 BRs in any given month.

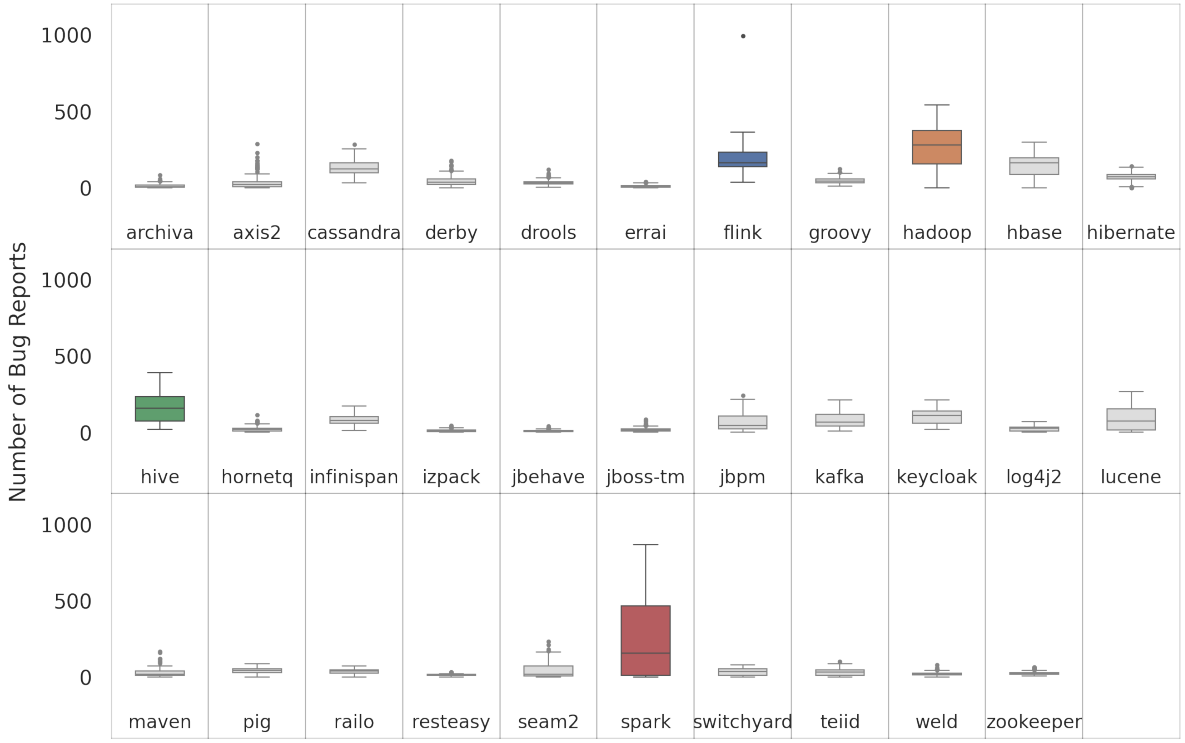


Figure A2.: Boxplots of Number of Bug Reports Created per Month
The highlighted items represent the projects that exceeded 300 BRs in any given month.

Table A4 shows the number of BRs each corpus contained before and after filtering. Highlighted rows show the projects that exceeded the threshold for inclusion. The project names were preserved based on the original `sqlite` filenames from Rath and Mäder (2019b), except for the project `jboss-transaction-manager` which we renamed to `jboss-tm` for improved readability. The project `wildfly` was excluded from analysis due to technical issues with its contents.

Table A4.: Corpus sizes before and after data filtering

Project	Filtered	Total
archiva	702	1 929
axis2	2 006	5 796
cassandra	4 664	13 965
derby	2 489	6 969
drools	1 864	5 103
errai	327	1 060
flink	1 764	8 100
groovy	3 513	8 137
hadoop	11 490	39 086
hbase	5 899	19 247
hibernate	2 749	11 971
hive	5 836	18 025
hornetq	880	3 286
infinispan	2 910	8 422
izpack	542	1 337
jbehave	105	1 243
jboss-tm	891	2 887
jbpm	2 615	10 397
kafka	1 744	6 219
keycloak	1 226	5 523
log4j2	464	2 114
lucene	3 823	17 329
maven	1 519	5 073
pig	1 998	5 234
railo	1 190	3 326
resteasy	643	1 649
seam2	1 405	5 031
spark	4 889	22 205
switchyard	934	3 010
teiid	2 035	4 899
weld	895	2 518
zookeeper	912	2 907

Appendix B

The following violin and scatter plots show the values each vectorization approach attained for each combination of corpus and preprocessing method. The scatter plots have a 'jitter' applied to the x-axis coordinates to reduce overplotting.

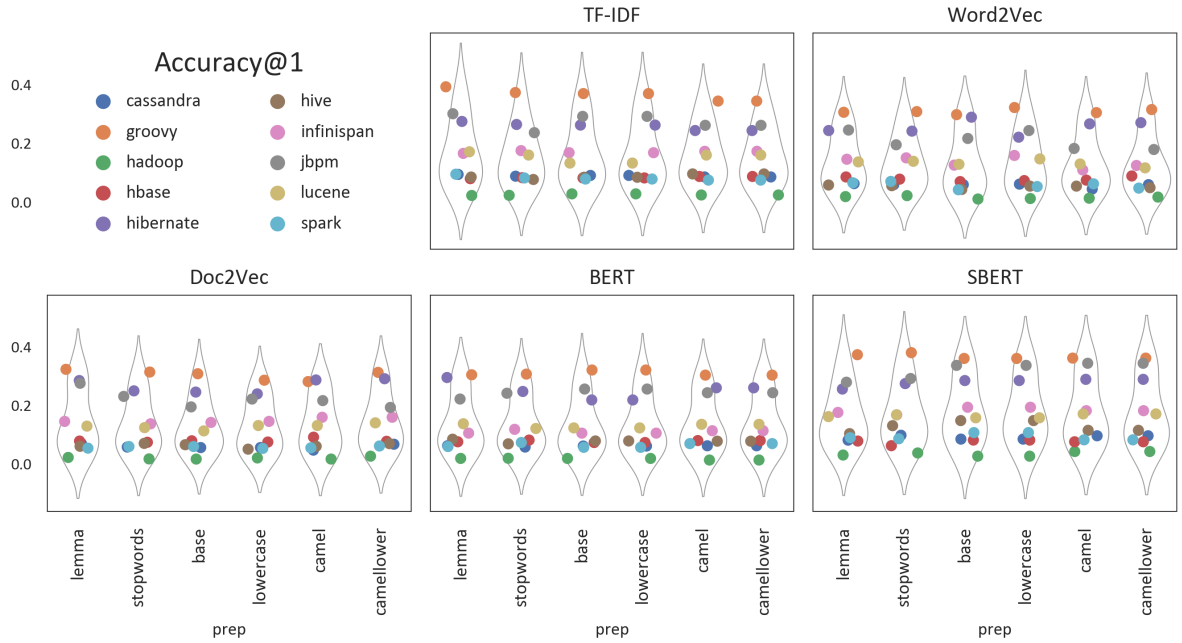


Figure B1.: Violin and Scatter Plots of Results: Accuracy@1

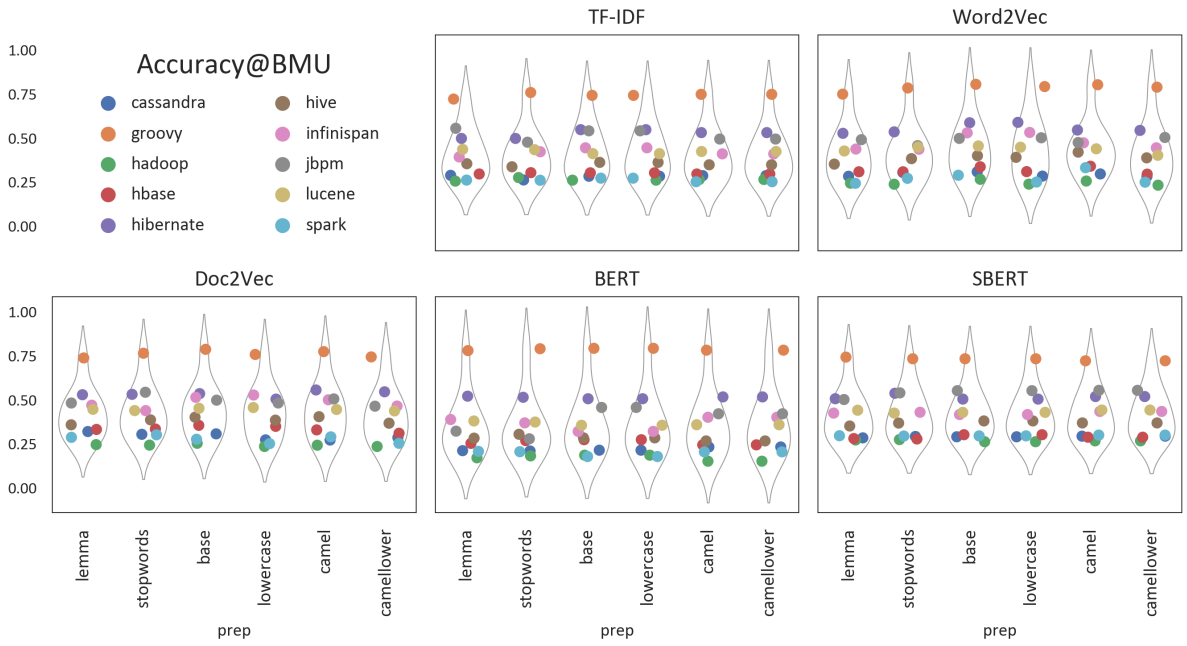


Figure B2.: Violin and Scatter Plots of Results: Accuracy@BMU

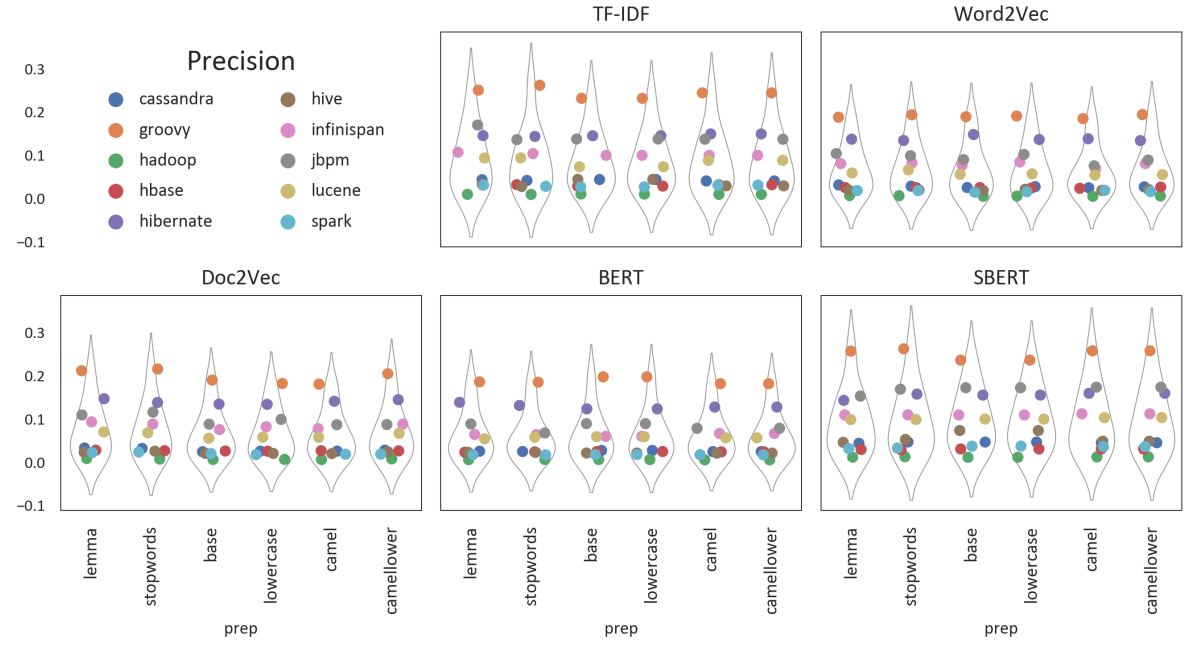


Figure B3.: Violin and Scatter Plots of Results: Precision

Figure B4.: Violin and Scatter Plots of Results: Topographic Error

Figure B5.: Violin and Scatter Plots of Results: Quantization Error

Table B1: Scores for Accuracy @ 1

Corpus	Prep	TFIDF	Word2Vec	Doc2Vec	BERT	SBERT
cassandra	lemma	0.094	0.063	0.071	0.063	0.082
groovy	lemma	0.394	0.307	0.324	0.305	0.374
hadoop	lemma	0.023	0.019	0.023	0.020	0.031
hbase	lemma	0.080	0.086	0.079	0.077	0.079
hibernate	lemma	0.275	0.244	0.286	0.296	0.257
hive	lemma	0.085	0.058	0.061	0.085	0.104
infinispan	lemma	0.166	0.146	0.146	0.106	0.177
jbpm	lemma	0.302	0.246	0.276	0.223	0.280
lucene	lemma	0.172	0.137	0.130	0.138	0.163
spark	lemma	0.095	0.066	0.055	0.060	0.091
cassandra	stopwords	0.088	0.059	0.059	0.059	0.099
groovy	stopwords	0.374	0.309	0.315	0.308	0.381
hadoop	stopwords	0.023	0.022	0.018	0.020	0.038
hbase	stopwords	0.083	0.078	0.075	0.082	0.063
hibernate	stopwords	0.265	0.242	0.251	0.248	0.275
hive	stopwords	0.077	0.056	0.071	0.070	0.132
infinispan	stopwords	0.176	0.151	0.138	0.119	0.169
jbpm	stopwords	0.237	0.196	0.232	0.242	0.293
lucene	stopwords	0.161	0.139	0.125	0.122	0.169
spark	stopwords	0.082	0.070	0.060	0.074	0.087
cassandra	base	0.091	0.059	0.057	0.062	0.086
groovy	base	0.370	0.298	0.309	0.322	0.361
hadoop	base	0.028	0.010	0.018	0.020	0.027
hbase	base	0.082	0.070	0.080	0.074	0.082
hibernate	base	0.263	0.290	0.246	0.219	0.286
hive	base	0.085	0.043	0.067	0.079	0.149
infinispan	base	0.169	0.126	0.143	0.106	0.195
jbpm	base	0.293	0.217	0.196	0.257	0.338
lucene	base	0.133	0.129	0.114	0.124	0.159
spark	base	0.079	0.042	0.060	0.058	0.108
cassandra	lowercase	0.091	0.061	0.057	0.062	0.086
groovy	lowercase	0.370	0.323	0.287	0.322	0.361
hadoop	lowercase	0.028	0.012	0.021	0.020	0.027
hbase	lowercase	0.082	0.073	0.076	0.074	0.082
hibernate	lowercase	0.263	0.222	0.240	0.219	0.286
hive	lowercase	0.085	0.054	0.051	0.079	0.149
infinispan	lowercase	0.169	0.159	0.146	0.106	0.195
jbpm	lowercase	0.293	0.244	0.223	0.257	0.338
lucene	lowercase	0.133	0.147	0.132	0.124	0.159
spark	lowercase	0.079	0.053	0.054	0.058	0.108

Continued on next page

Table B1: Scores for Accuracy @ 1 (Continued)

Corpus	Prep	TFIDF	Word2Vec	Doc2Vec	BERT	SBERT
cassandra	camel	0.086	0.045	0.048	0.063	0.097
groovy	camel	0.345	0.305	0.282	0.304	0.362
hadoop	camel	0.024	0.013	0.018	0.014	0.043
hbase	camel	0.087	0.075	0.092	0.081	0.077
hibernate	camel	0.244	0.267	0.288	0.261	0.290
hive	camel	0.096	0.054	0.061	0.079	0.116
infinispan	camel	0.173	0.110	0.160	0.114	0.183
jbpm	camel	0.262	0.183	0.217	0.244	0.345
lucene	camel	0.161	0.130	0.132	0.136	0.172
spark	camel	0.075	0.062	0.056	0.070	0.083
cassandra	lowcamel	0.086	0.059	0.068	0.063	0.097
groovy	lowcamel	0.345	0.316	0.314	0.304	0.362
hadoop	lowcamel	0.024	0.017	0.027	0.014	0.043
hbase	lowcamel	0.087	0.089	0.078	0.081	0.077
hibernate	lowcamel	0.244	0.271	0.292	0.261	0.290
hive	lowcamel	0.096	0.049	0.070	0.079	0.116
infinispan	lowcamel	0.173	0.125	0.160	0.114	0.183
jbpm	lowcamel	0.262	0.180	0.194	0.244	0.345
lucene	lowcamel	0.161	0.117	0.141	0.136	0.172
spark	lowcamel	0.075	0.048	0.062	0.070	0.083

Table B2: Scores for Accuracy @ BMU

Corpus	Prep	TFIDF	Word2Vec	Doc2Vec	BERT	SBERT
cassandra	lemma	0.290	0.284	0.321	0.213	0.285
groovy	lemma	0.722	0.750	0.739	0.780	0.743
hadoop	lemma	0.257	0.245	0.247	0.172	0.273
hbase	lemma	0.297	0.310	0.333	0.254	0.281
hibernate	lemma	0.499	0.528	0.530	0.522	0.507
hive	lemma	0.356	0.353	0.359	0.284	0.352
infinispan	lemma	0.394	0.439	0.471	0.389	0.426
jbpm	lemma	0.557	0.492	0.483	0.323	0.503
lucene	lemma	0.437	0.428	0.447	0.381	0.442
spark	lemma	0.263	0.244	0.288	0.209	0.297
cassandra	stopwords	0.263	0.306	0.305	0.211	0.293
groovy	stopwords	0.760	0.785	0.765	0.791	0.734
hadoop	stopwords	0.277	0.239	0.244	0.183	0.275
hbase	stopwords	0.305	0.310	0.336	0.270	0.279
hibernate	stopwords	0.499	0.536	0.532	0.516	0.538
hive	stopwords	0.338	0.384	0.387	0.306	0.369
infinispan	stopwords	0.423	0.438	0.440	0.370	0.430
jbpm	stopwords	0.478	0.458	0.544	0.280	0.540
lucene	stopwords	0.435	0.450	0.440	0.375	0.426
spark	stopwords	0.262	0.273	0.303	0.207	0.296
cassandra	base	0.284	0.310	0.308	0.215	0.291
groovy	base	0.743	0.806	0.787	0.793	0.734
hadoop	base	0.262	0.267	0.255	0.187	0.263
hbase	base	0.304	0.338	0.356	0.275	0.303
hibernate	base	0.549	0.588	0.536	0.507	0.505
hive	base	0.363	0.401	0.402	0.285	0.381
infinispan	base	0.446	0.531	0.515	0.322	0.417
jbpm	base	0.542	0.499	0.499	0.458	0.553
lucene	base	0.413	0.456	0.453	0.357	0.430
spark	base	0.274	0.290	0.277	0.179	0.296
cassandra	lowercase	0.284	0.285	0.274	0.215	0.291
groovy	lowercase	0.743	0.793	0.758	0.793	0.734
hadoop	lowercase	0.262	0.240	0.236	0.187	0.263
hbase	lowercase	0.304	0.311	0.351	0.275	0.303
hibernate	lowercase	0.549	0.590	0.505	0.507	0.505
hive	lowercase	0.363	0.392	0.386	0.285	0.381
infinispan	lowercase	0.446	0.533	0.528	0.322	0.417
jbpm	lowercase	0.542	0.503	0.483	0.458	0.553
lucene	lowercase	0.413	0.449	0.457	0.357	0.430
spark	lowercase	0.274	0.252	0.253	0.179	0.296

Continued on next page

Table B2: Scores for Accuracy @ BMU (Continued)

Corpus	Prep	TFIDF	Word2Vec	Doc2Vec	BERT	SBERT
cassandra	camel	0.288	0.297	0.273	0.234	0.295
groovy	camel	0.749	0.804	0.775	0.783	0.722
hadoop	camel	0.266	0.257	0.244	0.153	0.268
hbase	camel	0.297	0.341	0.331	0.246	0.289
hibernate	camel	0.532	0.547	0.557	0.518	0.520
hive	camel	0.349	0.421	0.405	0.268	0.370
infinispan	camel	0.412	0.474	0.501	0.402	0.435
jbpm	camel	0.496	0.476	0.506	0.422	0.555
lucene	camel	0.425	0.440	0.446	0.360	0.443
spark	camel	0.253	0.333	0.291	0.206	0.301
cassandra	lowcamel	0.288	0.283	0.287	0.234	0.295
groovy	lowcamel	0.749	0.790	0.744	0.783	0.722
hadoop	lowcamel	0.266	0.234	0.236	0.153	0.268
hbase	lowcamel	0.297	0.296	0.311	0.246	0.289
hibernate	lowcamel	0.532	0.545	0.547	0.518	0.520
hive	lowcamel	0.349	0.389	0.368	0.268	0.370
infinispan	lowcamel	0.412	0.446	0.466	0.402	0.435
jbpm	lowcamel	0.496	0.504	0.465	0.422	0.555
lucene	lowcamel	0.425	0.403	0.438	0.360	0.443
spark	lowcamel	0.253	0.251	0.254	0.206	0.301

Table B3: Scores for Precision

Corpus	Prep	TFIDF	Word2Vec	Doc2Vec	BERT	SBERT
cassandra	lemma	0.044	0.032	0.033	0.026	0.044
groovy	lemma	0.251	0.188	0.212	0.186	0.257
hadoop	lemma	0.010	0.007	0.009	0.006	0.012
hbase	lemma	0.031	0.026	0.028	0.023	0.030
hibernate	lemma	0.146	0.138	0.147	0.139	0.143
hive	lemma	0.034	0.020	0.023	0.023	0.046
infinispan	lemma	0.107	0.081	0.094	0.064	0.110
jbpm	lemma	0.170	0.104	0.110	0.089	0.153
lucene	lemma	0.094	0.059	0.070	0.055	0.099
spark	lemma	0.032	0.019	0.023	0.018	0.032
cassandra	stopwords	0.042	0.029	0.032	0.025	0.047
groovy	stopwords	0.262	0.194	0.216	0.185	0.262
hadoop	stopwords	0.010	0.007	0.008	0.007	0.013
hbase	stopwords	0.033	0.026	0.027	0.024	0.029
hibernate	stopwords	0.144	0.135	0.138	0.132	0.158
hive	stopwords	0.028	0.020	0.026	0.023	0.053
infinispan	stopwords	0.104	0.083	0.089	0.064	0.110
jbpm	stopwords	0.137	0.099	0.116	0.068	0.169
lucene	stopwords	0.094	0.067	0.068	0.058	0.099
spark	stopwords	0.029	0.019	0.024	0.017	0.033
cassandra	base	0.044	0.026	0.025	0.028	0.047
groovy	base	0.232	0.189	0.190	0.198	0.236
hadoop	base	0.011	0.006	0.007	0.006	0.012
hbase	base	0.029	0.026	0.026	0.025	0.031
hibernate	base	0.146	0.148	0.135	0.124	0.155
hive	base	0.045	0.019	0.021	0.022	0.074
infinispan	base	0.100	0.077	0.076	0.060	0.110
jbpm	base	0.138	0.090	0.088	0.090	0.172
lucene	base	0.074	0.056	0.056	0.059	0.100
spark	base	0.027	0.015	0.020	0.018	0.037
cassandra	lowercase	0.044	0.028	0.026	0.028	0.047
groovy	lowercase	0.232	0.191	0.182	0.198	0.236
hadoop	lowercase	0.011	0.006	0.007	0.006	0.012
hbase	lowercase	0.029	0.025	0.026	0.025	0.031
hibernate	lowercase	0.146	0.137	0.134	0.124	0.155
hive	lowercase	0.045	0.022	0.021	0.022	0.074
infinispan	lowercase	0.100	0.085	0.082	0.060	0.110
jbpm	lowercase	0.138	0.102	0.099	0.090	0.172
lucene	lowercase	0.074	0.057	0.058	0.059	0.100
spark	lowercase	0.027	0.016	0.018	0.018	0.037
cassandra	camel	0.041	0.025	0.026	0.025	0.045

Continued on next page

Table B3: Scores for Precision (Continued)

Corpus	Prep	TFIDF	Word2Vec	Doc2Vec	BERT	SBERT
groovy	camel	0.245	0.185	0.181	0.182	0.258
hadoop	camel	0.010	0.006	0.006	0.006	0.013
hbase	camel	0.032	0.024	0.027	0.024	0.031
hibernate	camel	0.150	0.139	0.141	0.128	0.160
hive	camel	0.030	0.019	0.020	0.021	0.049
infinispan	camel	0.100	0.069	0.078	0.067	0.112
jbpm	camel	0.137	0.076	0.087	0.079	0.174
lucene	camel	0.089	0.055	0.058	0.057	0.103
spark	camel	0.031	0.019	0.019	0.018	0.036
cassandra	lowcamel	0.041	0.027	0.028	0.025	0.045
groovy	lowcamel	0.245	0.195	0.205	0.182	0.258
hadoop	lowcamel	0.010	0.006	0.008	0.006	0.013
hbase	lowcamel	0.032	0.027	0.027	0.024	0.031
hibernate	lowcamel	0.150	0.134	0.145	0.128	0.160
hive	lowcamel	0.030	0.021	0.024	0.021	0.049
infinispan	lowcamel	0.100	0.081	0.088	0.067	0.112
jbpm	lowcamel	0.137	0.089	0.087	0.079	0.174
lucene	lowcamel	0.089	0.056	0.067	0.057	0.103
spark	lowcamel	0.031	0.016	0.019	0.018	0.036

Table B4: Scores for Quantization Error

Corpus	Prep	TFIDF	Word2Vec	Doc2Vec	BERT	SBERT
cassandra	lemma	0.905	1.345	0.488	0.292	0.773
groovy	lemma	0.875	1.447	0.478	0.276	0.728
hadoop	lemma	0.940	1.489	0.553	0.312	0.811
hbase	lemma	0.915	1.453	0.540	0.299	0.779
hibernate	lemma	0.866	1.190	0.487	0.284	0.743
hive	lemma	0.911	1.400	0.528	0.281	0.779
infinispan	lemma	0.870	1.249	0.488	0.286	0.735
jbpm	lemma	0.866	1.482	0.487	0.240	0.758
lucene	lemma	0.897	1.417	0.552	0.302	0.773
spark	lemma	0.914	1.455	0.537	0.313	0.784
cassandra	stopwords	0.919	1.027	0.421	0.298	0.770
groovy	stopwords	0.891	1.124	0.413	0.280	0.726
hadoop	stopwords	0.951	1.204	0.488	0.314	0.807
hbase	stopwords	0.927	1.201	0.483	0.304	0.776
hibernate	stopwords	0.880	0.947	0.429	0.283	0.737
hive	stopwords	0.926	1.150	0.474	0.286	0.775
infinispan	stopwords	0.885	0.985	0.425	0.290	0.731
jbpm	stopwords	0.881	1.173	0.426	0.244	0.755
lucene	stopwords	0.912	1.149	0.485	0.307	0.770
spark	stopwords	0.927	1.209	0.482	0.318	0.782
cassandra	base	0.932	0.720	0.426	0.298	0.764
groovy	base	0.910	0.717	0.417	0.288	0.726
hadoop	base	0.966	0.758	0.461	0.308	0.802
hbase	base	0.945	0.726	0.445	0.299	0.772
hibernate	base	0.900	0.622	0.409	0.272	0.734
hive	base	0.938	0.795	0.463	0.294	0.769
infinispan	base	0.904	0.586	0.388	0.267	0.730
jbpm	base	0.894	0.746	0.436	0.292	0.747
lucene	base	0.935	0.683	0.432	0.301	0.776
spark	base	0.939	0.766	0.447	0.293	0.779
cassandra	lowercase	0.932	0.846	0.477	0.298	0.764
groovy	lowercase	0.910	0.848	0.468	0.288	0.726
hadoop	lowercase	0.966	0.884	0.513	0.308	0.802
hbase	lowercase	0.945	0.865	0.495	0.299	0.772
hibernate	lowercase	0.900	0.742	0.460	0.272	0.734
hive	lowercase	0.938	0.933	0.520	0.294	0.769
infinispan	lowercase	0.904	0.714	0.435	0.267	0.730
jbpm	lowercase	0.894	0.920	0.498	0.292	0.747
lucene	lowercase	0.935	0.791	0.478	0.301	0.776
spark	lowercase	0.939	0.925	0.514	0.293	0.779
cassandra	camel	0.922	0.647	0.431	0.294	0.764

Continued on next page

Table B4: Scores for Quantization Error (Continued)

Corpus	Prep	TFIDF	Word2Vec	Doc2Vec	BERT	SBERT
groovy	camel	0.897	0.682	0.427	0.297	0.720
hadoop	camel	0.952	0.673	0.473	0.290	0.802
hbase	camel	0.933	0.651	0.461	0.298	0.777
hibernate	camel	0.884	0.557	0.420	0.276	0.732
hive	camel	0.928	0.706	0.469	0.266	0.773
infinispan	camel	0.888	0.535	0.405	0.293	0.727
jbpm	camel	0.885	0.682	0.440	0.242	0.748
lucene	camel	0.915	0.606	0.449	0.303	0.767
spark	camel	0.928	0.691	0.458	0.300	0.776
cassandra	lowcamel	0.922	0.864	0.529	0.294	0.764
groovy	lowcamel	0.897	0.925	0.525	0.297	0.720
hadoop	lowcamel	0.952	0.970	0.607	0.290	0.802
hbase	lowcamel	0.933	0.982	0.600	0.298	0.777
hibernate	lowcamel	0.884	0.794	0.539	0.276	0.732
hive	lowcamel	0.928	0.939	0.580	0.266	0.773
infinispan	lowcamel	0.888	0.824	0.535	0.293	0.727
jbpm	lowcamel	0.885	0.931	0.537	0.242	0.748
lucene	lowcamel	0.915	0.933	0.595	0.303	0.767
spark	lowcamel	0.928	0.967	0.589	0.300	0.776

Table B5: Scores for Topographic Error

Corpus	Prep	TFIDF	Word2Vec	Doc2Vec	BERT	SBERT
cassandra	lemma	0.653	0.441	0.438	0.441	0.402
groovy	lemma	0.553	0.386	0.414	0.423	0.426
hadoop	lemma	0.690	0.453	0.550	0.494	0.540
hbase	lemma	0.654	0.449	0.463	0.476	0.482
hibernate	lemma	0.520	0.400	0.419	0.414	0.409
hive	lemma	0.657	0.424	0.478	0.487	0.526
infinispan	lemma	0.558	0.372	0.382	0.433	0.411
jbpm	lemma	0.551	0.405	0.434	0.357	0.441
lucene	lemma	0.588	0.404	0.445	0.441	0.454
spark	lemma	0.695	0.423	0.475	0.433	0.506
cassandra	stopwords	0.641	0.404	0.438	0.492	0.438
groovy	stopwords	0.612	0.364	0.427	0.425	0.464
hadoop	stopwords	0.717	0.425	0.484	0.484	0.547
hbase	stopwords	0.659	0.396	0.485	0.491	0.500
hibernate	stopwords	0.538	0.339	0.426	0.402	0.445
hive	stopwords	0.678	0.375	0.481	0.476	0.502
infinispan	stopwords	0.539	0.327	0.384	0.390	0.410
jbpm	stopwords	0.550	0.367	0.415	0.389	0.464
lucene	stopwords	0.601	0.354	0.447	0.437	0.467
spark	stopwords	0.693	0.438	0.478	0.468	0.485
cassandra	base	0.727	0.345	0.403	0.479	0.459
groovy	base	0.667	0.304	0.355	0.420	0.436
hadoop	base	0.761	0.419	0.426	0.510	0.538
hbase	base	0.730	0.311	0.423	0.465	0.490
hibernate	base	0.566	0.278	0.360	0.377	0.437
hive	base	0.732	0.370	0.430	0.497	0.501
infinispan	base	0.569	0.279	0.270	0.424	0.410
jbpm	base	0.562	0.323	0.406	0.383	0.439
lucene	base	0.672	0.318	0.383	0.462	0.460
spark	base	0.697	0.333	0.367	0.486	0.476
cassandra	lowercase	0.727	0.360	0.436	0.479	0.459
groovy	lowercase	0.667	0.333	0.410	0.420	0.436
hadoop	lowercase	0.761	0.388	0.450	0.510	0.538
hbase	lowercase	0.730	0.350	0.461	0.465	0.490
hibernate	lowercase	0.566	0.347	0.379	0.377	0.437
hive	lowercase	0.732	0.373	0.475	0.497	0.501
infinispan	lowercase	0.569	0.295	0.383	0.424	0.410
jbpm	lowercase	0.562	0.360	0.447	0.383	0.439
lucene	lowercase	0.672	0.345	0.411	0.462	0.460
spark	lowercase	0.697	0.400	0.477	0.486	0.476

Continued on next page

Table B5: Scores for Topographic Error (Continued)

Corpus	Prep	TFIDF	Word2Vec	Doc2Vec	BERT	SBERT
cassandra	camel	0.692	0.340	0.420	0.462	0.415
groovy	camel	0.606	0.342	0.367	0.403	0.435
hadoop	camel	0.753	0.368	0.489	0.508	0.542
hbase	camel	0.721	0.342	0.450	0.469	0.487
hibernate	camel	0.538	0.310	0.381	0.383	0.442
hive	camel	0.692	0.391	0.436	0.482	0.458
infinispan	camel	0.579	0.306	0.381	0.419	0.432
jbpm	camel	0.551	0.340	0.364	0.402	0.428
lucene	camel	0.617	0.349	0.399	0.437	0.454
spark	camel	0.710	0.355	0.364	0.471	0.469
cassandra	lowcamel	0.692	0.418	0.442	0.462	0.415
groovy	lowcamel	0.606	0.367	0.427	0.403	0.435
hadoop	lowcamel	0.753	0.436	0.501	0.508	0.542
hbase	lowcamel	0.721	0.440	0.522	0.469	0.487
hibernate	lowcamel	0.538	0.360	0.430	0.383	0.442
hive	lowcamel	0.692	0.393	0.464	0.482	0.458
infinispan	lowcamel	0.579	0.332	0.374	0.419	0.432
jbpm	lowcamel	0.551	0.370	0.414	0.402	0.428
lucene	lowcamel	0.617	0.386	0.504	0.437	0.454
spark	lowcamel	0.710	0.432	0.483	0.471	0.469

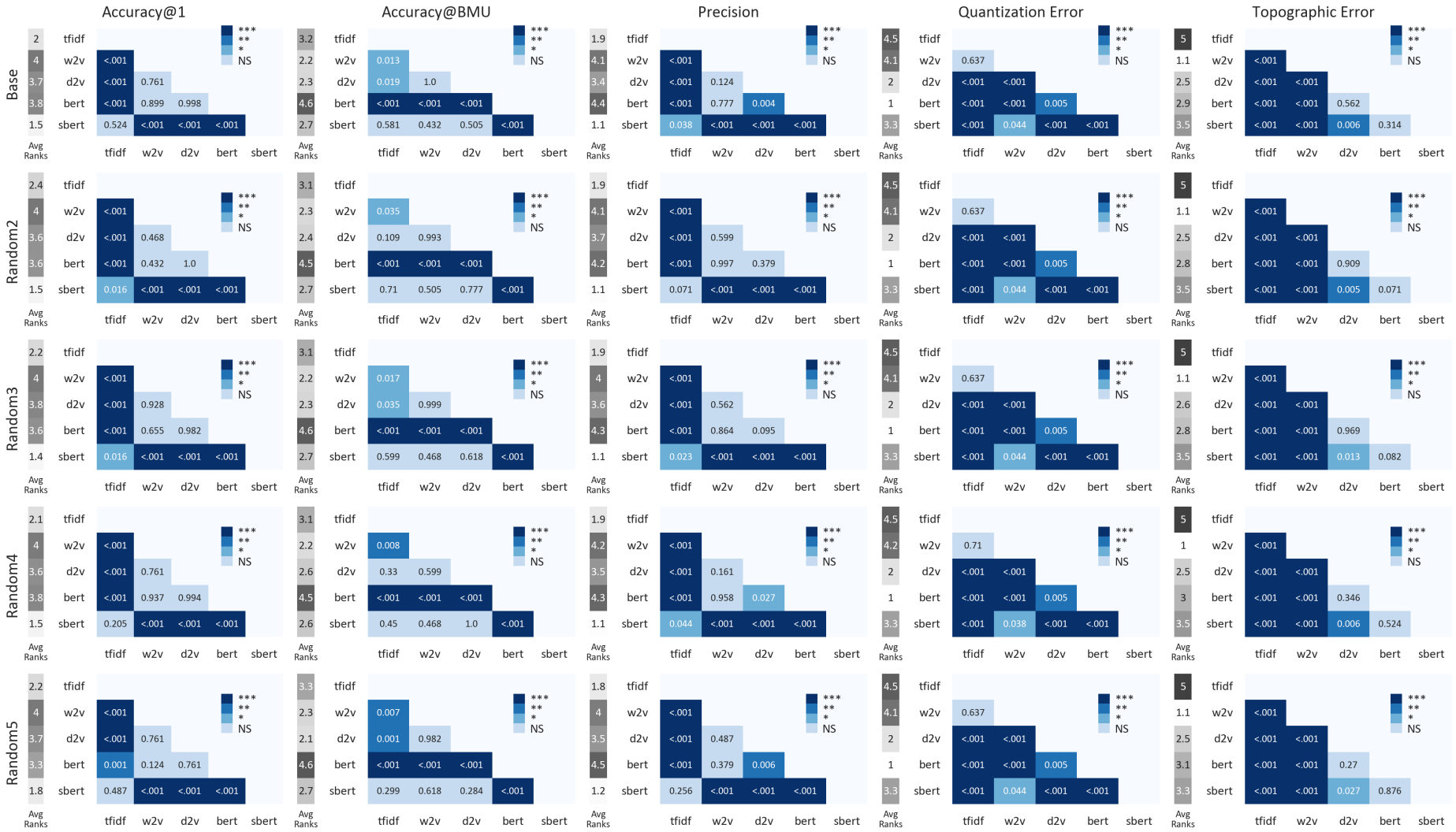


Figure B6.: Pair-wise p-values of SOM Initialization Experiments: Varying Random Seeds
Each row corresponds to one initialization experiment, and each column corresponds to an evaluation metric. The grayscale colorbar on the left shows the average rank values.

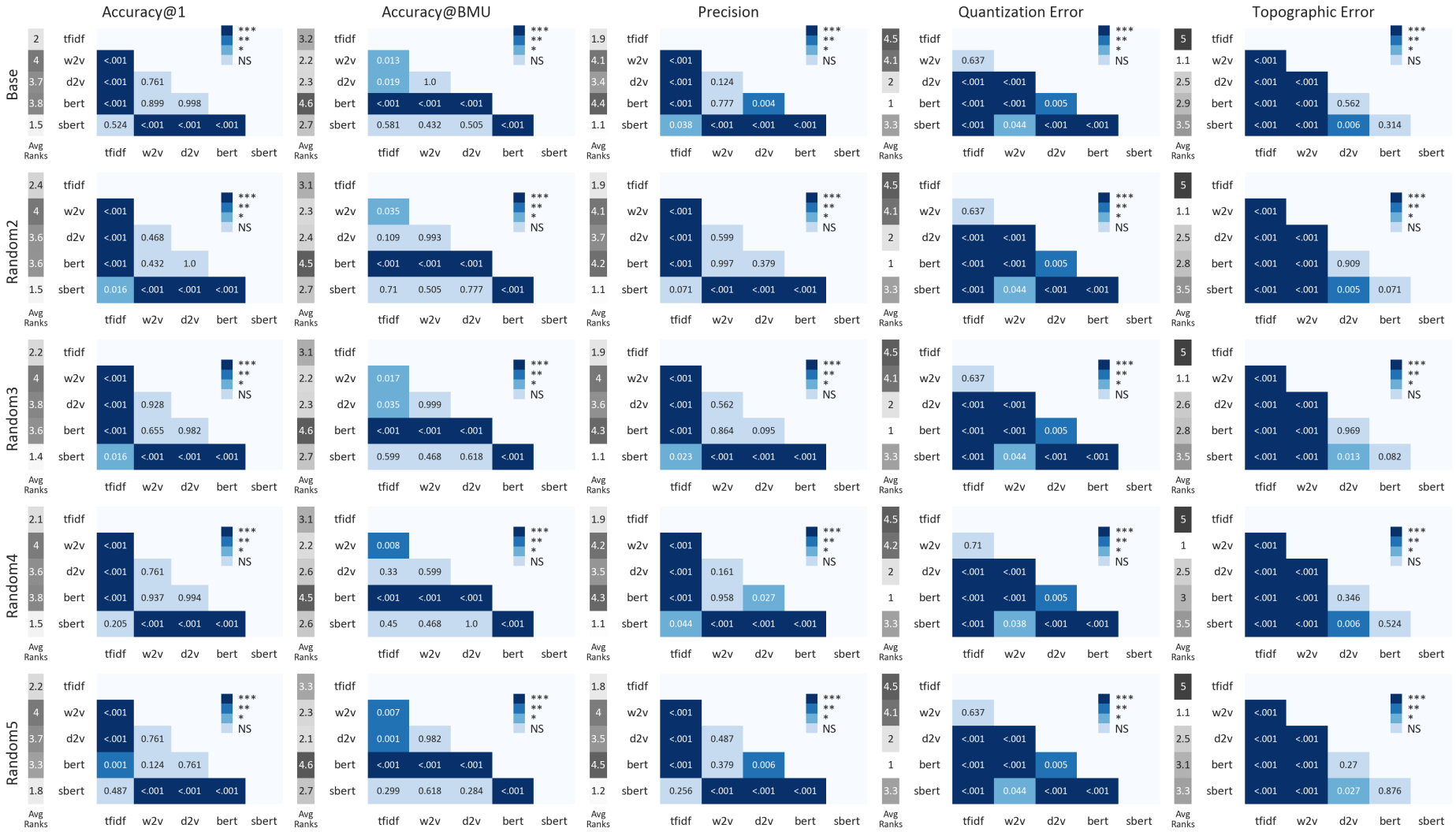


Figure B7.: Pair-wise p-values of SOM Initialization Experiments: Varying Map Parameters
Each row corresponds to one initialization experiment, and each column corresponds to an evaluation metric. The grayscale colorbar on the left shows the average rank values.

Annex A

```
x : int
    x dimension of the SOM.
y : int
    y dimension of the SOM.
input_len : int
    Number of the elements of the vectors in input.
sigma : float, optional (default=1.0)
    Spread of the neighborhood function, needs to be adequate
    to the dimensions of the map.
    (at the iteration t we have sigma(t) = sigma / (1 + t/T)
    where T is #num_iteration/2)
learning_rate : initial learning rate
    (at the iteration t we have
    learning_rate(t) = learning_rate / (1 + t/T)
    where T is #num_iteration/2)
decay_function : function (default=asymptotic_decay)
    Function that reduces learning_rate and sigma at each iteration
    the default function is:
                learning_rate / (1+t/(max_iterarations/2))
    A custom decay function will need to to take in input
    three parameters in the following order:
    1. learning rate
    2. current iteration
    3. maximum number of iterations allowed
    Note that if a lambda function is used to define the decay
    MiniSom will not be pickable anymore.
neighborhood_function : string, optional (default='gaussian')
    Function that weights the neighborhood of a position in the map.
    Possible values: 'gaussian', 'mexican_hat', 'bubble', 'triangle'
topology : string, optional (default='rectangular')
    Topology of the map.
    Possible values: 'rectangular', 'hexagonal'
activation_distance : string, callable optional (default='euclidean')
    Distance used to activate the map.
    Possible values: 'euclidean', 'cosine', 'manhattan', 'chebyshev'
    Example of callable that can be passed:
    def euclidean(x, w):
        return linalg.norm(subtract(x, w), axis=-1)
random_seed : int, optional (default=None)
    Random seed to use.
```

Listing A1: MiniSom training parameters

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação

Universidade Nova de Lisboa