Automatically Identifying Performance Issue Reports with Heuristic Linguistic Patterns

Anonymous Author(s)

ABSTRACT

Performance issues compromise the response time and resource consumption of a software system. Modern software systems use issue tracking database to keep track of all kinds of issue reports, including performance issues. However, performance issues are largely under-tagged in practice, since it is voluntary and manual. For example, the performance tag rate in Apache's Jira system is below 1%. This paper contributes a novel, hybrid classification approach, combining linguistic patterns and machine/deep learning techniques, to automatically detect performance issue reports. We manually learn from 980 real-life performance issue reports, and summarize 80 project-agnostic linguistic patterns that recur in the reports. Our approach uses these linguistic patterns to construct the sentence-level and issue-level learning features for training effective machine/deep learning classifiers. We test our approach on two new datasets, each with 980 unclassified issues reports. We compare our approach with 31 baseline methods, including simple keyword matching and the combinations of machine/deep learning models and classic NLP features. Our approach can reach up to 83% precision and up to 59% recall. The only comparable baseline method is BERT, which is still 25% lower in the F_1-score.

KEYWORDS

software performance, performance optimization, software repositories mining

ACM Reference Format:

Anonymous Author(s). 2020. Automatically Identifying Performance Issue Reports with Heuristic Linguistic Patterns. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software performance issues can cause dissatisfaction and drive users to switch to competitor products [1]. Like other types of software bugs, performance problems are reported to developers via issue tracking systems. Modern issue tracking systems support tagging a specific issue report as being performance-related (e.g., via a *"Label"* in Jira), to allow product managers to quickly find and prioritize addressing such problems. However, manual tagging of issue reports is tedious and leads to significant under-tagging in practice. We find that only 5 of the 13 largest open-source Apache projects have over 1% of issue reports tagged as performance issues—when empirical studies [2] find that performance issues should be around 4% to 16%.

Automatic tagging of performance issues is an ideal that presents a challenge for existing analysis techniques. Simple techniques, such as keyword matching, tend to find excessive false positives; while machine/deep learning methods struggle due to the imbalance of issue types for training. For example, our manual investigation, of almost 2000 randomly selected issues from Apache's Jira system, finds only 7% performance issues. With such a significantly unbalanced dataset, machine/deep learning models tend to miss the few positive cases—leading to high precision and low recall.

This paper presents an approach to automatically tag performance issues, based on linguistic analysis of the issue description. Our approach stems from the observation that performance-related issues often contain similar linguistic characteristics at the sentence-level, agnostic of specific software product. By manually analyzing almost 1000 issues, we contribute a set of 80 heuristic linguistic patterns to capture these common linguistic features. Combining these linguistic patterns with state-of-the-art machine/deep learning models, we offer a method for practitioners to automatically identify performance-related issue reports. The advantage of our approach lies in two aspects: 1) the linguistic patterns provide specific learning features to pin-point descriptions of performance issues, while classic NLP features are mostly based on general statistical models; and 2) our sentence-level classifier plays an important role in extracting more accurate learning features for issue-level classification. In fact, incorporating sentence-level analysis improves our recall by 24% on average.

We evaluated our approach on two different datasets, each containing 980 unclassified issue reports, randomly selected from Apache's Jira platform. We compared the sentence-level and issue-level classifiers in our approach, with a total of 31 baseline methods—including keyword matching, and combinations of different machine/deep learning models and classic NLP features. The results showed that our approach can identify performance-related issues with up to 83% precision and 59% recall. In comparison, most state-of-the-art methods produce only 29% recall.

The rest of this paper is organized as following. Section 2 provides fundamental background information. Section 3 details our approach. Section 4 describes our evaluation design and Section 5 explores the evaluation results. Section 6 reviews related prior work, and Section 7 discusses threats to validity and potential future work. Finally, Section 8 concludes.

2 BACKGROUND

This paper is tackling a text classification problem. We will introduce the background information of this problem in this section.

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

a) Linguistic Patterns: *"Linguistic patterns are grammatical rules that allow their users to speak properly in a common language"* [3]. Previous research has been using heuristic linguistic patterns to classify texts [4, 5]. For example, *"As a xx..., I want to..., so that..."* is a heuristic linguistic pattern, which captures how a user story is usually described [6]. It can be used to automatically match texts that describe users stories in a large corpus. Sometimes, the classification is not definite due to the fuzzy nature of the problem. In these cases, researchers combine heuristic linguistic patterns with fuzzy logic [7–9]. For example, in the study of Shi *et al.*, the authors combine linguistic patterns with fuzzy logic to classify texts in issue reports into different information types, including *Intent, Benefit, Drawback, Example, Explanation* and *Trivia*. [10]. The tricky part is that an input text could reasonably relate to multiple information types without being definite. Thus, they assign a confidence value (0 to 1) to each linguistic pattern. The confidence value models the association between this pattern and a information type. The linguistic fuzzy model provides interpretability of the classification process [11, 12].

In this paper, we are dealing with a binary classification: performance related or not, but not in between. Thus, fuzziness is not necessary in our case. The commonality is that we manually learn heuristic linguistic patterns from a large body of developer-tagged performance issue reports, similar to the practice of Shi *et al.* [10].

b) Machine/Deep Learning. Machine learning and deep learning models are commonly used for text classification [13]. They gain knowledge of classification via large amounts of training based on manually tagged datasets.

An effective classifier relies on extracting relevant features from the data for training. A common approach is to transform input texts into a numerical representation in the form of vectors and matrix [14]. For example, *Count Vector* works on the frequency of terms. It is a matrix notation of a corpus, where every row represents a document from the corpus, every column represents a term from the corpus, and every cell represents the frequency count of a particular term in a particular document [15].

However, simply calculating the frequency of terms suffers from a critical problem that all terms are considered equally important when it comes to assessing relevancy on a query. Thus, *TF-IDF (Term Frequency - Inverse Document Frequency)* is proposed to scale down the weights of terms with high collection frequency [16]. *TF-IDF* can be generated at different levels of input tokens [17]: 1) *Word-Level TF-IDF* is a matrix representing scores of every term in different corpus []; 2) *N-gram Level TF-IDF* is the combination of *N* terms together []; 3) *Character Level TF-IDF* is a matrix representing scores of character level N-gram in the corpus.

Unlike sparse matrix such as *Count Vector*, *Word Embedding* is a form of representing words and documents using a dense vector representation [18]. The position of a word within the vector is learned from text and based on the words that surround it. *Word Embedding* can be generated using pre-trained embeddings, such as *Glove* [19], *FastText* [20], and *Word2Vec* [21].

The used model also impacts the accuracy of the classification. Machine learning models have been used for addressing various classification problems. Each model has its unique feature. *Naive Bayes* is a classification technique based on Bayes' Theorem with an assumption of independence among predictors [22–24]. *Logistic regression* measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic/sigmoid function [25]. *Support Vector Machine* is a supervised machine learning algorithm which can be used for both classification or regression challenges [26, 27]. *Decision Tree* is a flowchart-like structure that are commonly used in operations research and management [28]. *Random Forest* models are a type of ensemble models, particularly bagging models [29]. *Extreme Gradient Boosting* is a machine learning ensemble meta-algorithm for primarily reducing bias, and also variance in supervised learning, and a family of machine learning algorithms that convert weak learners to strong ones [30].

Deep learning modesl has gained popularity in text classification, since they are inspired by how human brain works. There are two main deep learning models: *Convolutional Neural Networks (CNN)* and *Recurrent Neural Networks (RNN)*. In addition, the *BERT* model is the landmark of multi-head attention deep learning models. It has shown to outperform the previous methods such as CNNs, RNNs, and among others for a wide variety of natural language processing (NLP) tasks [31, 32]. Instead of selecting features, *Bidirectional Encoder Representations from Transformers (BERT)* benefits from transfer learning. In transfer learning, the model first is trained on a large text set to solve some general-purpose by training the model like language modeling and auto-encoding [33]. This step, regarded as pretraining, prepares the deep learning model to rapidly learn new downstream tasks. Thus, BERT does not use a classic method of using features as proxies of contextual information but the model deep learning weights encode a lot of information about the language. We simply used token embedding as features for the BERT model.

c) Hybrid Approaches: Hybrid approaches combine a base classifier, i.e. the machine/deep learning model, with a rule based system, to improve the classification [34]. These hybrid systems can be easily fine-tuned by adding specific rules for conflicting tags that haven't been correctly modeled by the base classifier.

Our approach resides in this category: we train the classic machine/deep learning models with the heuristic linguistic patterns derived learning features. To prove the advantage of our approach, we will compare with a comprehensive set of baseline methods, by combining different NLP features and machine/deep learning models discussed in the previous section. We envision the advantage of our approach lies in that the heuristic linguistic patterns extracted from known performance issues provide more accurate learning features, compared to classic NLP features, which are based on statistical modeling.

3 APPROACH

Our approach is inspired by the observation that performance-related issues often include sentences that share similar linguistic characteristics. We capture these common characteristics with heuristic linguistic patterns, which we empirically derive from analyzing existing issue reports. Leveraging these linguistic patterns and machine learning techniques, we can classify new issues as performance-related or not. Therefore, our approach consists of

two main phases: a preparation/learning phase to build the linguistics patterns and an issue classification phase. We first elaborate on linguistic patterns before detailing these two phases below.

3.1 Heuristic Linguistic Patterns

Heuristic linguistic patterns allow us to approximate whether some text relates to the topic of performance. An issue report that describes the system as *"running slow"* or *"inefficient"*, for example, would indicate that it likely concerns the system performance. These patterns offer a project-agnostic method for us to identify issue-report texts that describe the symptoms, root causes, solutions, and run-time measurements of performance issues. Our patterns operate at the sentence-level by analyzing the words and grammatical structure of a sentence. Following the approach of Shi et al. [10], we define four types of linguistic patterns: *lexical*, *profiling*, *structural*, and *semantic*.

Lexical linguistic patterns extend the basic keyword match concept by additionally considering synonyms, negated antonyms, and lemmatizations of that keyword/phrase. For example, an *efficiency* lexical pattern would include the such terms as *efficient*, *efficiency*, *inefficient*, and *inefficiency*. Another example, which we call the *infinite_loop* pattern, looks for *infinite*, *forever*, or *endless*; followed by *loop* or *iteration*.

When a performance issue is identified, developers often use a profiler to record performance characteristics of the system. We define *profiling linguistic patterns* to capture this information, as embedded in an issue report. Usually, the matching issue text contains a time or memory usage unit (e.g., milliseconds, megabytes), or the extent of performance change—measured in percentage terms, comparing the run-time parameters of before and after a code revision.

Structural linguistic patterns operate on a higher grammatical level of the sentence structure. Here, we are looking for phrase structures that imply a performance issue. For example, we observe that "when … run/execute/perform … for [NUMBER] seconds/minutes" is a common way in different projects to describe performance issues that happen under a special input.

Semantic linguistic patterns extend lexical patterns by also incorporating sentiment analysis [35], to capture linguistic expressions that imply performance issues. For example, our *negative_necessary* pattern searches for a common way of describing the root cause or solution to performance issues that happen under unnecessary conditions. It searches for the word *necessary*, *required*, or *essential* in a sentence that has a negative sentiment.

In the following subsection, we detail how we derive our linguistic patterns for classifying performance-related issues.

3.2 Preparation/Learning Phase

The goal is to extract a comprehensive *heuristic linguistic pattern set* from known performance issue reports, which can be used to automatically tag more performance issues in a new dataset. The raw *input* is the developer-tagged performance issue reports on Apache's Jira system. The *output* is a comprehensive *heuristic linguistic pattern set*. This part contains five iterative steps.

a) Rank & Retrieve. We rank all Apache Software Foundation projects in descending order based on the number of developer-tagged performance issues. On Apache's Jira system, developers

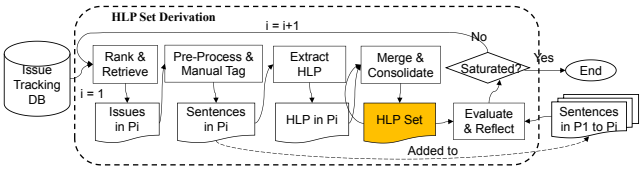


Figure 1: Iterative HLP Set Derivation

defined a special label, named "performance" to tag issues. In each iteration, i (starting from 1), we retrieve all the developer-tagged performance issue reports from the ith project, namely Pi. In each iteration, issues from Pi serve as the source for us to learn how performance issues are described in practice. We start from projects with the largest number of tagged performance issues to accelerate the building process.

b) Pre-Process & Manual Tag. First, we pre-process the issue reports from project P_i. We use the *Stanford Core-NLP*[1] to break each issue report into sentences. For each sentence, we applied lemmatization, Part-Of-Speech (POS) tagging , Named-Entity-Recognition (NER) tagging, dependency parsing, and sentimental analysis using the annotators of *Stanford Core-NLP*. We apply standard data cleaning to remove stop words, such as "a", "an", "the".

Next, we manually tag each sentence in each issue as either performance related or not performance related. The reason is that a performance issue usually also contains many sentences that do not carry performance related information. For example, developers may describe the general background information of an issue or include social notes. Thus, a sentence must contain description relevant to performance problems, such as the symptoms, the causes, the optimization solutions, and performance profiling data, to be considered as performance related. The goal is to identify sentences that contain reusable information that can help identify similar issues in a different context.

A team of five people worked together on the manual tagging, including a senior researcher, a Ph.D candidate, a master student (with 6 years of previous working experience as product manager), and two senior undergraduate students in the Software Engineering major. To best avoid bias in the tagging process, we divided the team into two groups: 1) the Ph.D candidate and one senior undergraduate; and 2) the master student and the other senior undergraduate. The senior researcher worked as the mediator in case of conflicts. The two members in each group tagged the same set of sentences, and cross-validate their results. Each tagger not only tags the sentences as performance related or not, but also provides comments that explain why a sentence should be tagged as performance related. This helps them to be more transparent and definite about their decision. This also provides reference in the case of disagreement between two taggers. In the case of disagreements, the senior researcher examines the case, and makes a final decision based on the grounds of whether general, reusable information for tagging other performance issues exists in the sentence.

c) Heuristic Linguistic Pattern Extraction. Based on the tagged performance-related sentences and the respective comments

[1] https://stanfordnlp.github.io/CoreNLP/

provided by the taggers, we manually extract and summarize *heuristic linguistic patterns* that can recur in different contexts for describing performance problems. This is a combination of automated and manual processes.

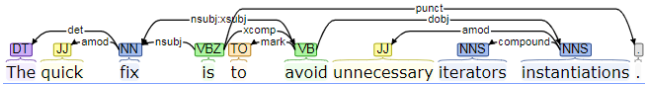
For each sentence that is manually tagged as performance related in P_i, we identify the linguistic properties using the *Stanford Core-NLP*. First, we use the *Part-Of-Speech Tagger (POS Tagger)* to assign parts of speech tags to each word, such as noun, verb, adjective, etc. This helps us to extract lexical, structural and semantic patterns.

For example, *"load_nn"* is a lexical heuristic linguistic pattern, indicating that a sentence must contain keyword "load" or "loads" in the form of a noun, used in describing computation load(s). Similarly, *"nn_by_nn"* (structural heuristic linguistic pattern) captures issues like "pixel by pixel", "byte by byte", which is often used to describe a tedious computation process.

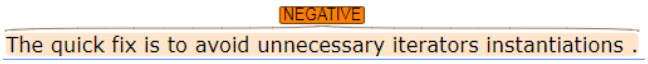
Meanwhile, we use *Named Entity Recogonizer Tagger (NER Tagger)* to capture specific terms for describing time or memory consumption of an issue. This helps to extract profiling heuristic linguistic patterns such as *"Percentage"* (NER Tagger contains "PERCENT") and *"Duration"* (NER Tagger contains "DURATION") that describe the profiling measurements.

Furthermore, *Stanford Core-NLP* categorizes a sentence in sentiment such as positive, neutral, and negative. This helps to extract semantic heuristic linguistic patterns. Figure 2 shows an example of a sentence in issue report *KAFKA-5512* matches *"negative_necessary"* (semantic heuristic linguistic pattern), since it has keyword *"unnecessary"* and its sentiment is categorized as negative.

By the end of this step, the output is a set of heuristic linguistic patterns from the project, P_i.



(a) Dependencies Analysis



(b) Sentimental Analysis

Figure 2: An Example of Extracting a Semantic Heuristic Linguistic Pattern

d) Merge & Consolidate. We merge the heuristic linguistic pattern set from project P_i with the set built from previous iterations. The heuristic linguistic patterns from P_i may overlap/duplicate with the existing heuristic linguistic pattern set. Thus, we merge overlapping, duplicating, or similar patterns together. For example, *"infinite_loop"* and *"loop_forever"* can be merged to one heuristic linguistic pattern, i.e. *"loop_infinite/forever*, which means that an infinite loop occurs. While merging, we also consolidate the merged heuristic linguistic pattern through divergent thinking. That is, we add possible variations of identified rules to be inclusive and predictive. For example, the example rule, *"loop_infinite/forever*, above, can be extended to a more predicable rule, *"loop/iteration_infinite/forever.*

e) Evaluate & Reflect. The last step of each iteration is to evaluate and reflect the updated heuristic linguistic pattern set. First, we check whether the heuristic linguistic pattern set has grown compared to previous iterations. If there is no (or only a few) new patterns added in the current iteration, it indicates that the heuristic linguistic pattern set is (or close to being) saturated. Next, we evaluate the precision/recall of the heuristic linguistic pattern set using the data from the processed projects. We use a naive matching approach: as long as a sentence matches one of the heuristic linguistic patterns from the set, we consider it as performance-related. We calculate the precision/recall of this naive tagging by comparing with our manual tagging. If the precision is low, it means that the heuristic linguistic patterns can also frequently appear in non-performance related sentences, implying that the heuristic linguistic pattern set is irrelevant. If the recall is low, it means that the heuristic linguistic pattern set is not comprehensive to capture all different ways that performance issues are presented. As the heuristic linguistic pattern set grows with iterations, the recall should increase gradually and become stable when no more new rules can be found. We call this status as heuristic linguistic pattern set saturation, which means the heuristic linguistic pattern building is complete and no more iterations are needed.

3.3 Issue Classification Phase

In the second part, we develop an automatic issue tagging approach by leveraging the heuristic linguistic pattern set from part 1. The goal is that given an input issue report, our approach automatically outputs Yes or No, indicating whether this issue is related to performance problems. Our approach works at two progressive levels: 1) sentence tagging and 2) issue tagging (which depends on the sentence tagging).

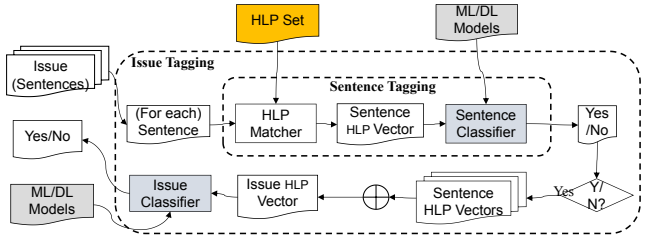


Figure 3: Performance Issue Classification

a) Sentence Tagging. First, given an input sentence, our approach automatically tags it as "Yes" or "No" in terms of performance-related. For each sentence in an issue, we use a "*HLP* Matcher" to calculate a *Sentence HLP Vector* representing which heuristic linguistic patterns are matched in this sentence. The *Sentence HLP Vector* is a binary vector with n dimensions, where n is the total number of heuristic linguistic patterns identified in part 1. The ith value in the vector is either 0 or 1, indicating whether the sentence matches the ith heuristic linguistic pattern. We use the *Sentence HLP Vector* as the learning feature to train machine learning and deep learning algorithms. Given any sentence as input, we can use the trained model to determine whether it is related to performance problems.

b) Issue Tagging. Next, we perform the issue-level tagging, built upon the sentence tagging. The input is an issue report, the output is also "Yes" or "No" indicating whether this issue report is relevant to performance problems. Towards this, we collect the *Sentence HLP*

Vectors of all the sentences that are tagged as "Yes" above. We add these vectors to calculate a weighted *Issue HLP Vector*, which also has n dimensions. The ith value in this vector indicates the number of sentences, which are tagged as "Yes" from the previous step and also matches the ith rule. Similarly, using the *Issue HLP Vector* as the learning feature, we train classic machine learning and deep learning models to automatically tag the issue.

c) Approach Variations. The unique contribution of this work is the heuristic linguistic pattern features, i.e. *Sentence HLP Vector* and *Issue HLP Vector*. They can be combined with different machine/deep learning models to provide the variations of our approach. In this study, we combine the heuristic linguistic pattern features with 6 classic machine learning models and 2 deep learning models. Thus we have 8 variations in our approach as shown in Table 1. Note that each variation can be applied either at the sentence level, by using *Sentence HLP Vector*, or the issue level, by using *Issue HLP Vector*.

Table 1: Sentence/Issue Tagging Approach Variations

Type	Abbre.	Model Name	Feature
HLP+ML	HLP+NB	Naive Bayes	Sentence HLP Vector OR Issue HLP Vector
	HLP+LR	Logistic Regression	
	HLP+SVM	Support Vector Machine	
	HLP+DT	Decision Tree	
	HLP+RF	Random Forest	
	HLP+XGB	Extreme Gradient Boosting	
HLP+DL	HLP+CNN	Convolutional Neural Network	
	HLP+RNN	Long Short Term Memory RNN	

4 EVALUATION DESIGN

We design our evaluation to answer to following research questions.

RQ1: When can the heuristic linguistic pattern set become saturated? This RQ investigates the number of projects we need to review until no more new heuristic linguistic pattern can be found, which indicates that the heuristic linguistic pattern set has became saturated.

RQ2: How accurate is our sentence tagging approach? We will compare the sentence tagging precision, recall, and F_1-score of our approach with that of a comprehensive set of baseline approaches.

RQ3: How accurate is our issue tagging approach? We will compare the issue tagging precision, recall, and F_1-score of our approach with that of a comprehensive set of baseline approaches.

RQ4: How much does sentence tagging impact the accuracy of issue tagging? That is, if we directly add the *Sentence HLP Vector* of all the sentences in an issue report without tagging the sentences first to calculate the *Issue HLP Vector*, how much accuracy will be compromised?

4.1 Experiment Setup

Next, we talk about the datasets used in this study, our experiment setting, and the comprehensive set of baseline approaches that we compare our approach against.

a) Datasets. We extract three datasets containing real-life issues from Apache's Jira system, for building heuristic linguistic patterns, as well as for training and testing our approach. Table 2 shows basic information of the three datasets.

Table 2: Datasets

ID	Purpose	#Issues (P%)	#Sentences (P%)
1	HLP Set Building	980 (100%)	5754 (48%)
2	Homologous Evaluation	980 (8%)	4790 (11%)
3	Heterologous Evaluation	980 (6%)	5371 (6%)

- **Dataset 1** is iteratively collected for building the heuristic linguistic pattern set following Section 3.2. As we will discuss in more details later, there are 5754 sentences from 980 issues in 13 projects.
- **Dataset 2** is for evaluating how the heuristic linguistic pattern set can identify more untagged performance issues from the same (i.e. homologous) projects. It contains 980 randomly selected non-tagged issues from the same projects of the heuristic linguistic pattern building. Therefore, only 8% of issues and 11% of sentences are verified as performance related.
- **Dataset 3** is for evaluating whether the heuristic linguistic patterns are general for tagging issues from heterologous projects, independent from the heuristic linguistic pattern building. It contains 980 issues and 5371 sentences from projects other than the projects of heuristic linguistic pattern building. 6% issues and sentences are verified as performance related.

We followed the process in Section 3.2 to manually tag each sentence in the three datasets. For dataset 2 and dataset 3, we also manually tag each issue report, following similar practice. The tagger provides comments for each issue to justify their decision of whether to manually tag this issue as a performance issue or not. The entire tagging process of three datasets took a total of approximately 762 human hours for the entire group of 4 taggers and 1 mediator.

Of a particular note, we noticed that containing a performance-related sentence in an issue report does not warrant a performance issue. The presence of performance related sentences in an issue could just serve as problem context or background information, instead of being the main focus/purpose of the issue. Issue *IGNITE-7849* below is such as an example:

> **IGNITE-7849**: *"Currently when we want to use the same predicate for the continuous query and initial query, it's easy enough to write something like this. Assuming we are fine with the performance of ScanQuery: <CODE SNIPPET>. However, this becomes more inconvenient when we want to use SqlQuery in the initial query to take advantage of indexing. This is obviously not ideal because we have to specify the predicate in two different ways. A quick Google revealed that there are products out there that more seamlessly support this use case of continuous querying. I understand that Ignite isn't built on top of SQL, unlike the commercial RDBMSes I found, so maybe this is an out-of-scope feature."*

The performance-related sentence is underlined above. However, it is just an assumption related to the performance of *ScanQuery*.

The main focus of this issue is a functional improvement related to "*continuous querying*". Thus this issue is tag as a "NO".

b) Experiment Setting. The iterative process of heuristic linguistic pattern building leads to the dataset 1. In answering RQ1, we will show how the heuristic linguistic pattern set became saturated with the growth of dataset 1.

In answering RQ2 (sentence tagging) and RQ3 (issue tagging), we use both the dataset 2 for the homologous evaluation and the dataset 3 for the heterologous evaluation respectively. For each dataset, we run the experiment for 20 times. Each time, we randomly divide the dataset into 70% for training and 30% for testing. The precision, recall, and F_1-score are calculated as the average of the 20 times.

In answering RQ4 (impact of sentence tagging on the accuracy of issue tagging), we implement a variation of our issue tagging approach. That is, instead of adding up the *Sentence HLP Vectors* for sentences that are tagged as performance-related, we directly sum the *Sentence HLP Vectors* of all sentences in an issue. Then, we compare the precision/recall and F1-score of this variation with our original issue tagging approach.

c) Comparison Baselines. To answer RQ2 and RQ3, we compare our approach with a total of 31 baseline methods, grouped into three types of baselines listed in Table 3. Note that, each baseline approach can also be applied at the sentence level or issue level. For fair comparison, some baseline methods only compare with a subset of our approach variations. For example, it is not fair to compare a deep learning model with a machine learning model. Thus, we will focus on three comparisons:

Table 3: Comparison Baselines

Baseline 1: ML Models + NLP Features		
Abbreviation	ML Model	NLP Feature
NB+CV|WD|NG|CH	Naive Bayes	Count Vectors (CV) Word Level TF-IDF (WD) N-gram Level TF-IDF (NG) Character Level TF-IDF (CH)
LR+CV|WD|NG|CH	Logistic Regression	
SVM+CV|WD|NG|CH	Support Vector Machine	
DT+CV|WD|NG|CH	Decision Tree	
RF+CV|WD|NG|CH	Random Forest	
XGB+CV|WD|NG|CH	Extreme Gradient Boosting	
Baseline 2: DL Models + NLP Features		
Abbreviation	DL Models	NLP Feature
BERT	BERT Classifier	Token Embedding
CNN	Convolutional Neural Network	Word Embedding
RNN-LSTM	Long Short Term Memory RNN	Word Embedding
RNN-GRU	Gated Recurrent Units RNN	Word Embedding
Bi-RNN	Bidirectional RNN	Word Embedding
Baseline 3: Keyword-based Matching		
Abbreviation	Method	
Kw LR	Matching Keywords from Literature	
Kw HLP	Matching Keywords in HLP Set	

(1) **(ML+HLP) VS. Baseline 1.** We compare 6 of our approach variations with machine learning models (ML) with baseline 1 methods. The baseline 1 contains 24 different methods. As shown in Table 3 (from row 3 to row 8), the baseline 1 is the combination of 6 classic machine learning models and 4 classic NLP features. For each machine learning model, we will compare the accuracy of using the heuristic linguistic pattern features vs. using the NLP features.

(2) **(DL+HLP) VS. Baseline 2.** We compare 2 of our approach variations with deep learning models (DL) with baseline 2 methods. As shown in Table 3 (row 11 to row 15), the baseline 2 contains 5 different approaches using 5 deep learning models and 2 NLP features.

(3) **(ML/DL+HLP) VS. Baseline 3.** As shown in Table 3 on the bottom, the baseline 3 includes: matching common keywords from literature and matching keywords derived from the lexical heuristic linguistic patterns (since this type can be used as keywords). We compare all of our approach variations with baseline 3 for comprehensiveness.

As a particular note, for issue tagging, our approach build the *Issue HLP Vector* by filtering out sentences that are not tagged as performance related. For a fair comparison, in each baseline method for issue tagging, we also extract the NLP features based only on sentences that are tagged (by this method) as performance related. For instance, when using BERT for issue tagging, the feature "Token Embedding" is extracted from sentences that are tagged by BERT as performance related.

5 RESULTS

5.1 Heuristic Linguistic Pattern Saturation (RQ1)

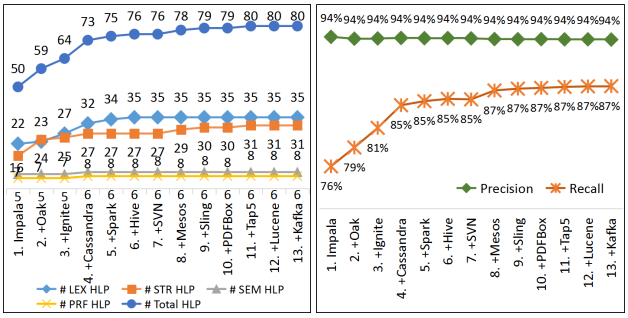
RQ1 : When can the heuristic linguistic pattern set become saturated? We went through 13 iterations to make sure that the fuzzy rule set was saturated. Table 4 shows the information of the 13 iterations, in terms of the project being studied, the domain of the project, the tag rate of performance issues, the number of developer-tagged performance issues, and the number (and percentage) of manually tagged performance sentences in each project. We collected 5754 sentences that describe performance related information from totally 980 issues. Overall, the performance issue tag rate is only 0.9%, and 48% of the sentences are manually verified as performance related.

Table 4: Heuristic Linguistic Pattern Building Iterations

Iter.	Project	Domain	T.Rate	#P.Issues	# P.Sentences (%)
1	Impala	Query Engine	4%	265	1339 (53%)
2	Oak	Repository Tool	2%	151	832 (50%)
3	Ignite	Distributed Database	2%	146	767 (44%)
4	Cassandra	Database Management	0.9%	124	696 (56%)
5	Spark	Analytic Engine	0.2%	52	479 (35%)
6	Hive	Database Tool	0.3%	39	194 (43%)
7	SVN	Revision Control	1%	38	357 (36%)
8	Mesos	Computer Cluster	0.5%	36	161 (53%)
9	Sling	Web Framework	0.4%	30	175 (60%)
10	PDFBox	PDF Library	0.7%	28	186 (53%)
11	Tapstry-5	Web Framework	1%	27	163 (48%)
12	Lucene	Retrieval Library	0.3%	22	210 (49%)
13	Kafka	Distributed Streaming	0.4%	22	204 (39%)
Total			0.9%	980	5754 (48%)

Figure 4 shows how the overall heuristic linguistic pattern set reaches saturation. In Figure 4a, the line on the top shows that the entire heuristic linguistic pattern set grows fast in the first 5 iterations, and it slows down after iteration 6, and becomes stable in iteration 11. We extracted a total of 80 heuristic linguistic patterns, with 46% lexical patterns, 38% structural patterns, 9% semantic patterns, and 8% profiling patterns. The accumulation process of the four types are detailed in the four lines on the bottom of Figure 4a. Figure 4b shows the precision and recall of using the updated fuzzy

rules by the end of each iteration to match sentences from all the reviewed data. If a sentence can match any heuristic linguistic pattern, we consider it as performance related. The precision/recall is calculated by comparing to our manual tagging. The precision remains constantly at 94%, indicating that the fuzzy rule set is precise. The recall grows from 76% in the first iteration and remains at 87% from iteration 9 to 13, indicating the the fuzzy rule set becomes comprehensive in iteration 9.



(a) # HLPs with Iterations (b) Precision and Recall

Figure 4: Heuristic Linguistic Pattern Saturation with Iterations

> **Answer to RQ1:** The fuzzy rule set becomes saturated after 11 iterations, containing 80 heuristic linguistic patterns, in the type of lexical (44%), structural (39%), semantic (10%), and profiling (8%). The heuristic linguistic pattern set can match performance related sentences in the rule building dataset with precision of 94% and recall of 87%. This indicates that the heuristic linguistic pattern set is saturated—and can precisely and comprehensively capture the features of how performance issues are described in the heuristic linguistic pattern building dataset.

5.2 Sentence Tagging Accuracy (RQ2)

RQ2: How accurate is our sentence tagging approach? We evaluated the 8 variations of our approach (see Table 1) on both dataset 2 and dataset 3 for homologous and heterologous evaluation. The precision, recall, and F_1-score of the 8 approach variations is shown in Figure 5. We can make two key observations:

- The performance of the 8 approach variations on dataset 2 and dataset 3 is highly consistent, indicating that the heuristic linguistic pattern set is generally applicable to different datasets, including those that are independent from the heuristic linguistic pattern building.
- DT+HLP, RF+HLP, CNN+HLP, and RNN+HLP can quite precisely (76% to 81% precision) and comprehensively (61% to 79% recall)) capture performance related sentences.

Next, we compare our approach variations with the three baseline approaches as listed in Table 3. We tested on both dataset 2 and dataset 3, yielding to the same conclusions. Due to space limit, we present the details based on dataset 2 in the paper, details based on dataset 3 can be found here [2].

[2]https://sites.google.com/view/fse-2020-hlp

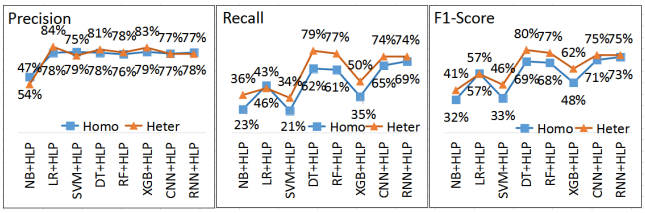
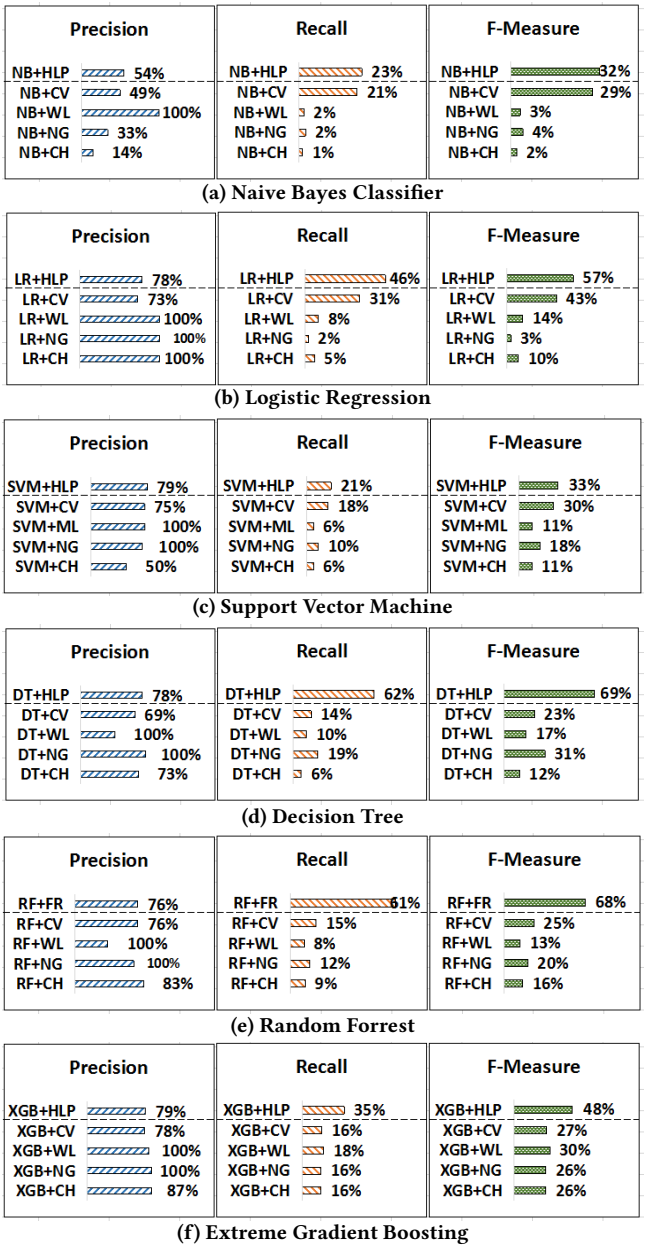


Figure 5: Sentence Tagging Precision, Recall, and F_1-Score



(a) Naive Bayes Classifier
(b) Logistic Regression
(c) Support Vector Machine
(d) Decision Tree
(e) Random Forrest
(f) Extreme Gradient Boosting

Figure 6: (ML+HLP) vs. Baseline 1 on Dataset 2

(ML+HLP) vs. Baseline 1. In Figure 6, each sub-figure shows the comparison of the precision, recall, and F_1-score of a particular machine learning model, when 1) using the *Sentence HLP Vector* as

learning feature (above the horizontal dash line) vs. 2) using the four classic NLP features (below the horizontal dash line). We can make the following observations: Our sentence tagging is 1.2 to 19 times better than most baseline 1 methods based on the F_1-score. For example, in Figure 6b, the F_1-score in LR+HLP is 57%, comparing with the 3% F_1-score in LR+NG. The exception is NB+CV and SVM+CV, compared to which the F_1-score of our approach only outperforms by 3%. Of a particular note, our approach always has the highest recall. It is challenging for machine learning approaches to achieve high recall when using unbalanced data like the performance issues with low positive cases. For example, the recall of most baseline 1 methods is very low (between 2% and 31%). In comparison, our DT+HLP and RF+HLP can have 62% and 61% recall, respectively, which is significantly higher.

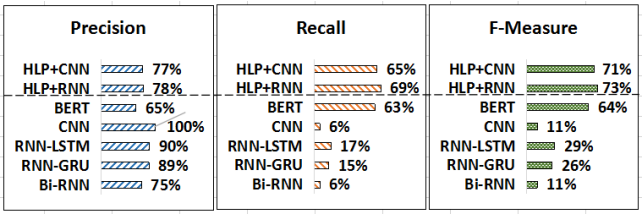


Figure 7: (DL+HLP) VS. Baseline 2 on Dataset 2

(DL+HLP) vs. Baseline 2. Figure 7 shows the comparison between 1) two deep learning models based on the *sentence HLP Vector* (above the horizontal dash line) vs. 2) five deep learning models based on classic NLP features (below the horizontal dash line). We observe that 1) our approach, RNN+HLP, has very high precision (78%), recall (69%), and F_1-score (73%) compared to the five baseline 2 methods. In particular, the classic DL models using the NLP features also suffer from low recall due to the unbalanced nature of the performance issue dataset. 2) BERT is comparable to our approach, but still with 13%, 6%, and 9% lower precision, recall, and F_1-score respectively.

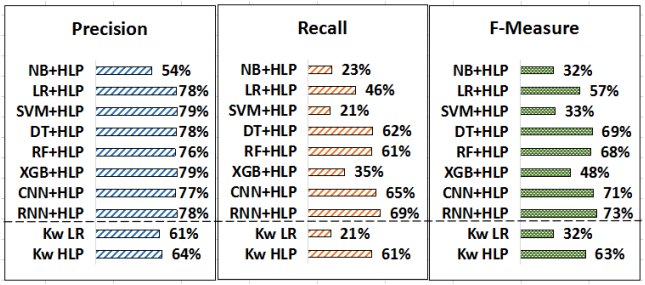


Figure 8: (ML/DL+HLP) VS. Baseline 3 on Dataset 2

(ML/DL+HLP) vs. Baseline 3. Figure 8 compares the 8 variations of our approach (above the horizontal dash line) with two keyword matching methods (below the horizontal dash line). We observe that: 1) the Kw matching based on keywords from previous literature yields to very low recall (21%), thus it is not a good method to tag performance sentences; 2) the Kw HLP method based on the lexical heuristic linguistic patterns has pretty good precision (64%) and recall (61%), indicating that the lexical heuristic linguistic patterns play a significant role in sentence tagging; and 3) our approach, leveraging all 8 types of heuristic linguistic patterns and combining with ML/DL models, can optimize the precision and recall by about 15% and 7%, compared to using the lexical heuristic linguistic pattern in a simple keyword matching manner.

Answer to RQ2: Four of our approach variations, namely DT+HLP, RF+HLP, CNN+HLP, and RNN+HLP, can precisely (76% to 81% precision) and comprehensively (61% to 79% recall) capture performance related sentences in different datasets, including those that are independent from heuristic linguistic pattern building. Our approach is significantly (1.2 to 19 times) better than most baseline methods. Only two baseline methods, BERT and KW HLP (matching keywords from the lexical rules), have comparable performance. However, our approach is still 13% to 15% better in precision and 6% to 7% better in recall.

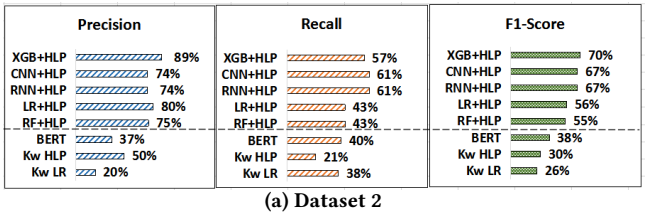
5.3 Issue Tagging Accuracy (RQ3)

RQ3: How accurate is our issue tagging approach? The issue tagging is built upon the sentence level tagging. As discussed earlier, there are a total of 39 methods for issue tagging: 8 variations of our approach, 2 keyword matching methods (baseline 3), 5 deep learning methods (baseline 2), and 24 (6*4) machine learning methods (baseline 1). In our approach, we build *Issue HLP Vectors* as the learning feature (see Section 3.3). In the baseline methods (used at the issue-level), we extract the NLP features from sentences that are tagged by the same method as performance related. For keyword matching, we tag an issue as performance related as long as there is one match.

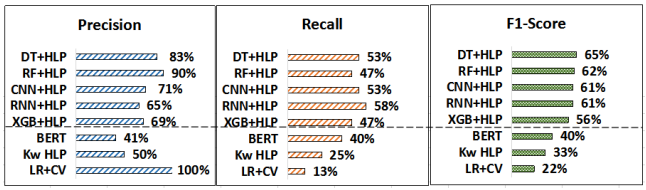
We repeat our experiment on three datasets: dataset 2, dataset 3, and their combination, to avoid the interference of particular datasets and generalize our findings. Figure 9a to Figure 9c show the precision, recall, and F_1-score of the issue tagging on dataset 2, dataset 3, and the combined dataset respectively. We listed the top five methods (among all 39 methods) ranked by the F_1-score, as well as BERT, Kw HLP, and the next best baseline method in each experiment. We observe that:

- The top five methods based on the three datasets are always from our approach variations, with up to 90% precision and up to 64% recall. As shown in Figure 9d, the winners in different datasets are highly consistent. CNN+HLP, XGB+HLP, RF+HLP, and RNN+HLP consistently rank in the top five for the three datasets. On average, these four methods have 67% to 83% precision, 51% to 59% recall. This indicates that these three methods can precisely identify the majority of performance issues from different datasets.
- The only comparable baseline method for issue tagging is BERT, whose F_1-score is still averagely 22% to 25% lower than CNN+HLP, XGB+HLP, RF+HLP, and RNN+HLP.

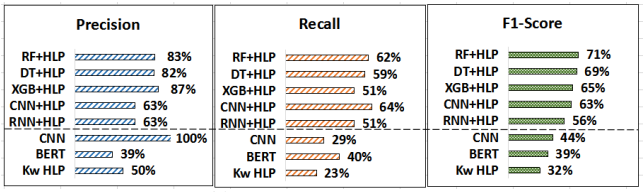
Answer to RQ3: Four of our approach variations, CNN+HLP, XGB+HLP, RF+HLP, and RNN+HLP, constantly outperform the other methods in the three datasets, reaching 67% to 83% precision and 51% to 57% recall. The only comparable baseline approach is BERT, whose F_1-score is still 22% to 25% lower.



(a) Dataset 2



(b) Dataset 3



(c) Dataset 2 + Dataset 3

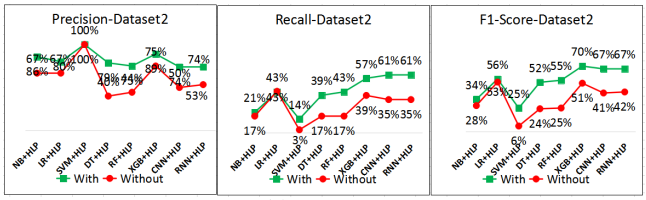
Rank	Dataset 2	Dataset 3	Combined	Winner	Precision	Recall	F1-Score
1	XGB+HLP	DT+HLP	RF+HLP	CNN+HLP	69%	59%	64%
2	CNN+HLP	RF+HLP	DT+HLP	XGB+HLP	82%	52%	63%
3	RNN+HLP	CNN+HLP	XGB+HLP	RF+HLP	83%	51%	62%
4	LR+HLP	RNN+HLP	CNN+HLP	RNN+HLP	67%	57%	61%
5	RF+HLP	XGB+HLP	RNN+HLP				

(d) Cross Dataset Winners for Issue Tagging

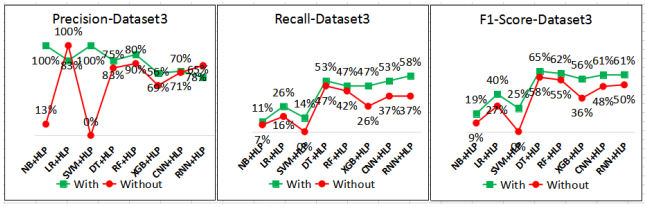
Figure 9: Issue Tagging

5.4 Impact of Sentence Tagging (RQ4)

RQ4: How much does the sentence tagging impact the accuracy of the issue tagging? This RQ examines how the precision/recall and F_1-score of our approach will be impacted by not performing sentence tagging prior to issue tagging. That is, we calculate the *Issue HLP Vector* by adding the *Sentence HLP Vector* of all the sentences in an issue, without filtering out sentences that are tagged as not related to performance.



(a) Dataset 2



(b) Dataset 3

Figure 10: Impact of Sentence Tagging on Issue Tagging

We conducted two experiments on dataset 2 and dataset 3, and the results are shown in Figure 10a and Figure 10b respectively. For both datasets, the precision and recall of each approach variation are improved non-trivially, on average 22% for the precision and 20% for the recall. Of a particular note, in the experiment with dataset 2, the precision of RF+HLP and DT+HLP increased by 31% and 39% with sentence tagging. This is because the recall in sentence tagging of these two methods have been significantly compromised by 61% and 62% respectively. Therefore, the sentence tagging is an important step before the issue tagging for optimizing the recall in tagging unbalanced data with a small percentage of performance issues.

Answer to RQ4: Sentence tagging as a pre-step can improve the issue tagging in precision (averagely 22%) and recall (averagely 20%), because it ensures that the issue-level features are built from more accurate input.

6 RELATED WORK

6.1 Issue Categorization

Software developers, testers and customers routinely submit issue reports to the issue tracking systems to record the problems they observe or requests they have. Due to the large amount of issue reports, previous research has developed different approaches to automatically categorize issue reports into different types of interests. This helps developers to prioritize different tasks.

Antoniol *et al.* showed that automatic classifiers such as *Decision Trees*, *Naive Bayes* and *Logistic Regression* are able to distinguish bugs from other kinds of issues such as enhancement and refactoring [36]. Pandey *et al.* compared more classifiers for distinguishing bugs from non-bugs, including *Naive Bayes*, *Logistic Regression*, *k-Nearest Neighbors*, *Support Vector Machine*, *Decision Tree*, and *Random Forest*. They found that *Random Forest* performs best in the text mining of bugs [37]. Some prior studies aim at classifying issue reports into more diverse categories. Ohira *et al.* manually reviewed 4000 issues reports in Apache's Jira system and classified the bugs impacted on products into security, performance, and breakage bugs [38, 39]. Limsettho developed an unsupervised framework that can automatically group bug reports together based on their textual similarity [40]. However, their framework labels issue reports using the most frequently appearing words rather than the labels that are already used by labels submitted by users of the issue tracking system. Thung *et al.* [41] and Chawla *et al.* [42] developed automated approaches that can automatically classify issue reports into different types such as functional bugs, security bugs, refactoring improvements, etc. But performance is not considered as a specific type by them.

In addition, some prior studies focused on internal content of issue reports. Shi *et al.* leveraged linguistic fuzzy rules to classify sentences in feature requests issue reports into six categories, namely *Intent, Benefit, Drawback, Example, Explanation* and *Trivia* [10]. Aggarwal *et al.* used machine learning models to detect duplicate issue reports [43].

Futhermore, some prior studies focused on classifying the severity, priority, and complexity of issue reports. Lamkanfi *et al.* proposed a technique to identify bug severity [44]. Tian *et al.* used

machine learning classifiers on issue reports as well as code bases to predict bug priorities [45]. Zhang *et al.* proposed a Markov-based method for estimating bug fixing time [46].

6.2 Performance Issue Analysis

A rich body of prior studies have been conducted to understand and classify performance bugs from different perspectives [1, 2, 47–52]. In past research, however, researchers often employ a simple keyword matching approach to identify performance issues. For example, the most common keywords include *"fast, slow, perform, latency, throughput, optimize, speed, heuristic, waste, efficient, unnecessary, redundant, too many times, lot of time, too much time"* [48, 53, 54]. However, the keyword matching largely compromise the accuracy and completeness of the retrieved data. In addition, there could be many different ways to describe performance issues beyond what is captured in known keywords. Thus, due to the lack of rigor in the keyword matching approach, researchers have to invest a large amount of time on manual verification.

In addition, prior studies usually focused on a specific type of performance bugs based on limited number of issue reports. Nistor *et al.* studied 150 performance bugs caused by inefficient loop [55]. Yu *et al.* studied 106 performance bugs caused by synchronization bottlenecks. Selakovic *et al.* presented an empirical study of 98 performance bugs written in JavaScript language in both [50]. Our study presented an adequate and diversified dataset of more than 1000 real-life performance issues, which can be used as a ground truth dataset for future studies on performance bugs.

7 DISCUSSION

a) Limitations: First, we did not test our approach on datasets from other than the Apache Jira platform, such as *Bugzilla* and *GNATS*. However, since *Apache Jira* is one of the most widely used platforms by real-world software projects, and our study subjects are from diverse domains, we believe that our approach should be general. Second, the heuristic linguistic patterns capture how practitioners describe performance problems in general terms. If issues are described only in project specific terms, understood only by project experts, our approach will be compromised. Lastly, we have not separately evaluated the accuracy of our approach on datasets of different domains. We believe that different heuristic linguistic patterns may provide difference accuracy for different domain data. A possible solution to the last two limitations is to tune the heuristic linguistic patterns with project/domain specific concepts.

b) Validity: We acknowledge that the manual tagging of performance related sentences and issues could pose internal threat to validity. Any manual effort is subjective to bias derived from individual expertise and understanding. We tried to mitigate this risk by allowing multiple taggers to cross-validate results, and by asking for tagging comments to increase transparency. However, this is still an inevitable threat to validity of our work.

c) Future work: We are motivated to optimize our approach on larger scale and more diverse datasets. In particular, we plan to polish our approach leveraging datasets from different issue tracking platforms, such as *Bugzilla* and *GNATS*. We also plan to leverage different types of heuristic linguistic patterns to facilitate the understanding and management of performance issues. For example, profiling linguistic patterns can be used to investigate the dynamic features of different performance issues. We plan to extend the usage of heuristic linguistic patterns to detect other types of issues, such as security issues.

8 CONCLUSION

In this paper, we contribute a hybrid classification approach, combining classic machine/deep learning models and the heuristic linguistic patterns, to automatically detect performance issues. We learned and derived a comprehensive heuristic linguistic pattern set with 80 patterns, from 980 developer-tagged performance issues from the Apache's Jira Platform. We used the heuristic linguistic pattern set to extract learning features that can accurately pin-point performance problem descriptions, and use these features train different machine/deep learning models. We evaluated our approach on two different datasets, each containing 980 unclassified issue reports, randomly selected from Apache's Jira platform. The results showed that our approach can identify performance-related issues with up to 83% precision and 59% recall, which has obvious advantage over 31 baseline methods, including BERT.

REFERENCES

[1] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 199–208. IEEE Press, 2012.

[2] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 237–246. IEEE Press, 2013.

[3] Alberto Rodrigues da Silva. Linguistic patterns and linguistic styles for requirements specification (i) an application case with the rigorous rsl/business-level language. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, pages 1–27, 2017.

[4] Hisao Ishibuchi, Tomoharu Nakashima, and Tadahiko Murata. Three-objective genetics-based machine learning for linguistic rule extraction. *Information sciences*, 136(1-4):109–133, 2001.

[5] Prerna Chikersal, Soujanya Poria, Erik Cambria, Alexander Gelbukh, and Chng Eng Siong. Modelling public sentiment in twitter: using linguistic patterns to enhance supervised learning. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 49–65. Springer, 2015.

[6] Laurens Müter, Tejaswini Deoskar, Max Mathijssen, Sjaak Brinkkemper, and Fabiano Dalpiaz. Refinement of user stories into backlog items: Linguistic structure and action verbs. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 109–116. Springer, 2019.

[7] Lotfi A Zadeh. Fuzzy sets. *Information and control*, 8(3):338–353, 1965.

[8] Lotfi A Zadeh. The concept of a linguistic variable and its application to approximate reasoning—ii. *Information sciences*, 8(4):301–357, 1975.

[9] Maria Jose Gacto, Rafael Alcalá, and Francisco Herrera. Interpretability of linguistic fuzzy rule-based systems: An overview of interpretability measures. *Information Sciences*, 181(20):4340–4360, 2011.

[10] Lin Shi, Celia Chen, Qing Wang, Shoubin Li, and Barry Boehm. Understanding feature requests by leveraging fuzzy method and linguistic analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 440–450. IEEE Press, 2017.

[11] Ebrahim H Mamdani. Application of fuzzy algorithms for control of simple dynamic plant. In *Proceedings of the institution of electrical engineers*, volume 121, pages 1585–1588. IET, 1974.

[12] EH Mamdani and S Assilian. An experiment in linguistic synthesis with a fuzzy logic controller. *International journal of human-computer studies*, 51(2):135–147, 1999.

[13] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[14] Justin Martineau, Tim Finin, Anupam Joshi, and Shamit Patel. Improving binary classification on text problems using differential word features. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 2019–2024, 2009.

[15] C Jashubhai Rameshbhai and Joy Paulose. Opinion mining on newspaper headlines using svm and nlp. *International Journal of Electrical and Computer Engineering (IJECE)*, 9(3):2152–2163, 2019.

[16] Ho Chung Wu, Robert Wing Pong Luk, Kam Fai Wong, and Kui Lam Kwok. Interpreting tf-idf term weights as making relevance decisions. *ACM Transactions on Information Systems (TOIS)*, 26(3):1–37, 2008.

[17] Matthias Eck, Stephan Vogel, and Alex Waibel. Low cost portability for statistical machine translation based on n-gram frequency and tf-idf. In *International Workshop on Spoken Language Translation (IWSLT) 2005*, 2005.

[18] Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 302–308, 2014.

[19] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[20] Ben Athiwaratkun, Andrew Gordon Wilson, and Anima Anandkumar. Probabilistic fasttext for multi-sense word embeddings. *arXiv preprint arXiv:1806.02901*, 2018.

[21] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[22] Jingnian Chen, Houkuan Huang, Shengfeng Tian, and Youli Qu. Feature selection for text classification with naïve bayes. *Expert Systems with Applications*, 36(3):5432–5435, 2009.

[23] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998.

[24] Fuchun Peng and Dale Schuurmans. Combining naive bayes and n-gram language models for text classification. In *European Conference on Information Retrieval*, pages 335–350. Springer, 2003.

[25] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[26] Aixin Sun, Ee-Peng Lim, and Ying Liu. On strategies for imbalanced text classification using svm: A comparative study. *Decision Support Systems*, 48(1):191–201, 2009.

[27] Fabrice Colas and Pavel Brazdil. Comparison of svm and some older classification algorithms in text classification tasks. In *IFIP International Conference on Artificial Intelligence in Theory and Practice*, pages 169–178. Springer, 2006.

[28] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[29] Mahesh Pal. Random forest classifier for remote sensing classification. *International Journal of Remote Sensing*, 26(1):217–222, 2005.

[30] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, and Yuan Tang. Xgboost: extreme gradient boosting. *R package version 0.4-2*, pages 1–4, 2015.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[33] Sebastian Ruder, Matthew E Peters, Swabha Swayamdipta, and Thomas Wolf. Transfer learning in natural language processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, pages 15–18, 2019.

[34] Julio Villena-Román, Sonia Collada-Pérez, Sara Lana-Serrano, and José Carlos González-Cristóbal. Hybrid approach combining machine learning and a rule-based expert system for text categorization. In *Twenty-Fourth International FLAIRS Conference*, 2011.

[35] Richard Tong. An operational system for detecting and tracking opinions in on-line discussions. In *Working Notes of the SIGIR Workshop on Operational Text Classification*, pages 1–6, 2001.

[36] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, pages 304–318, 2008.

[37] Nitish Pandey, Debarshi Kumar Sanyal, Abir Hudait, and Amitava Sen. Automated classification of software issue reports using machine learning techniques: an empirical study. *Innovations in Systems and Software Engineering*, 13(4):279–297, 2017.

[38] Yutaro Kashiwa, Hayato Yoshiyuki, Yusuke Kukita, and Masao Ohira. A pilot study of diversity in high impact bugs. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 536–540. IEEE, 2014.

[39] Masao Ohira, Yutaro Kashiwa, Yosuke Yamatani, Hayato Yoshiyuki, Yoshiya Maeda, Nachai Limsettho, Keisuke Fujino, Hideaki Hata, Akinori Ihara, and Kenichi Matsumoto. A dataset of high impact bugs: Manually-classified issue reports. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 518–521. IEEE, 2015.

[40] Nachai Limsettho, Hideaki Hata, Akito Monden, and Kenichi Matsumoto. Automatic unsupervised bug report categorization. In *2014 6th International Workshop on Empirical Software Engineering in Practice*, pages 7–12. IEEE, 2014.

[41] Ferdian Thung, David Lo, and Lingxiao Jiang. Automatic defect categorization. In *2012 19th Working Conference on Reverse Engineering*, pages 205–214. IEEE, 2012.

[42] Indu Chawla and Sandeep K Singh. Automatic bug labeling using semantic information from lsi. In *2014 Seventh International Conference on Contemporary Computing (IC3)*, pages 376–381. IEEE, 2014.

[43] Karan Aggarwal, Finbarr Timbers, Tanner Rutgers, Abram Hindle, Eleni Stroulia, and Russell Greiner. Detecting duplicate bug reports with software engineering domain knowledge. *Journal of Software: Evolution and Process*, 29(3):e1821, 2017.

[44] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 1–10. IEEE, 2010.

[45] Yuan Tian, David Lo, Xin Xia, and Chengnian Sun. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, 20(5):1354–1383, 2015.

[46] Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1042–1051. IEEE, 2013.

[47] Sebastian Baltes, Oliver Moseler, Fabian Beck, and Stephan Diehl. Navigate, understand, communicate: How developers locate performance bugs. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–10. IEEE, 2015.

[48] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[49] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM, 2014.

[50] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, pages 61–72. ACM, 2016.

[51] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *ACM SIGPLAN Notices*, volume 49, pages 561–578. ACM, 2014.

[52] Yutong Zhao, Lu Xiao, Wang Xiao, Bihuan Chen, and Yang Liu. Localized or architectural: an empirical study of performance issues dichotomy. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 316–317. IEEE, 2019.

[53] Michael Pradel, Markus Huggler, and Thomas R Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 13–25. ACM, 2014.

[54] Zhifei Chen, Bihuan Chen, Lu Xiao, Xiao Wang, Lin Chen, Yang Liu, and Baowen Xu. Speedoo: prioritizing performance optimization opportunities. In *Proceedings of the 40th International Conference on Software Engineering*, pages 811–821. ACM, 2018.

[55] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 902–912. IEEE, 2015.