Using Paragraph Vectors to improve our existing code review assisting tool-CRUSO

Conference Paper · December 2020

CITATIONS
0

4 authors, including:

Ritu Kapur
Indian Institute of Technology Ropar
11 PUBLICATIONS **39** CITATIONS
SEE PROFILE

Poojith Rao
Indian Institute of Technology Ropar
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Using Paragraph Vectors to improve our existing code review assisting tool-CRUSO

Ritu Kapur[*]
Poojith U Rao
Shipra Sharma
Balwinder Sodhi
ritu.kapur@iitrpr.ac.in
poojith.19csz0006@iitrpr.ac.in
shipra.sharma@iitrpr.ac.in
sodhi@iitrpr.ac.in
Indian Institute of Technology
Ropar, Punjab, India

ABSTRACT

Code reviews are one of the effective methods to estimate defectiveness in source code. However, the existing methods are dependent on experts or inefficient. In this paper, we improve the performance (in terms of speed and memory usage) of our existing code review assisting tool–CRUSO. The central idea of the approach is to estimate the defectiveness for an input source code by using the defectiveness score of similar code fragments present in various StackOverflow (SO) posts.

The significant contributions of our paper are i) *SOpostsDB*: a dataset containing the PVA vectors and the SO posts information, ii) *CRUSO-P*: a code review assisting system based on PVA models trained on *SOpostsDB*. For a given input source code, CRUSO-P labels it as {`Likely to be defective`, `Unlikely to be defective`, `Unpredictable`}. To develop CRUSO-P, we processed >3 million SO posts and 188200+ GitHub source files. CRUSO-P is designed to work with source code written in the popular programming languages {C, C#, Java, JavaScript, and Python}.

CRUSO-P outperforms CRUSO with an improvement of 97.82% in response time and a storage reduction of 99.15%. CRUSO-P achieves the highest mean accuracy score of 99.6% when tested with the C programming language, thus achieving an improvement of 5.6% over the existing method.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; **Maintaining software**.

KEYWORDS

Automated code review, StackOverflow, Paragraph Vector, Code quality, Software maintenance

ACM Reference Format:
Ritu Kapur, Poojith U Rao, Shipra Sharma, and Balwinder Sodhi. 2020. Using Paragraph Vectors to improve our existing code review assisting tool-CRUSO. In *ISEC '21: ACM SIGSOFT Innovations in Software Engineering Conference, Feb 25–27, 2021, Bhubaneswar, India.*. ACM, New York, NY, USA, 11 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

Code reviews play a significant role in detecting potential defects that remain undiscovered through the software testing process. Some such examples include memory leaks, buffer overflows, and scalability issues. However, the existing methods to perform the code reviews are dependent on subject-matter experts (SMEs) and being significantly time-consuming [17]. Therefore, we worked on **improving the performance of our existing code review assisting tool**– CRUSO [19].

For a given source code c, CRUSO performs the following steps:

(1) Identifies the set of StackOverflow (SO)[1] posts P such that each $p \in P$ contains source code fragment(s), which sufficiently resemble c.
(2) Determines the likelihood of c being defective by considering all $p \in P$. CRUSO uses the Winnowing algorithm [18] to represent source code as fingerprints, whose length is almost the same as the length of input source code. When used for source code matching, the variable-length fingerprints lead to a large number of source code comparisons, resulting in a significant memory usage and execution time.

To improve the performance of CRUSO, we replaced the Winnowing algorithm with the Paragraph vectors algorithm (PVA) [10], which uses a fixed-length vector representation for source code. We develop a reference dataset that stores the vector representations for code fragments present in SO posts generated using PVA, and the cosine similarity[2] of all the code fragments from a reference code fragment is chosen randomly. Thus, detecting relevant SO

[*] Corresponding author

ISEC '21, 25-27 February 2021, Bhubaneswar, India.
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/xx.xxx/xxx_x

[1] https://stackoverflow.com
[2] http://bit.ly/2ODWoEy

posts to given source code under review becomes a database search query for projecting the SO posts with the similarity score above a specific threshold value. We named the newer PVA-based version of CRUSO as CRUSO-P.

1.1 Existing techniques for source code representation

The representation of source code plays a significant role while training ML models. A broad categorization of the existing ML approaches based on their representation is as follows:

(1) *Fingerprint-based approaches*: A typical *code fingerprint* is a compact collection of integers, which summarizes the source code's critical aspects. The fingerprint-based approaches make use of code fingerprints generated by different algorithms to detect the source code similarity. The algorithms used generally comprise of Winnowing algorithm [18] and hash-based methods such as MD5 and SHA-1. Winnowing has been used as a source code similarity detection in software activities such as *plagiarism detection* [24] and *code review* [19]. Similarly, MD5 and SHA have been used to detect different source code clones [1]. We use cosine similarity measure to detect the source code similarity between different code fragments.

(2) *Abstract Syntax Tree (AST)-based approaches*: ASTs capture the syntactical details of programming constructs' used in source code. An AST of source code represents a hierarchical structure (tree) comprising of the programming constructs used in the source code in the order of their usage. However, the usage of AST differs in various research works. For instance, authors in [26] use a linear collection of programming constructs present in the source code's AST for training a Deep Belief Network to perform defectiveness estimation. In contrast, the authors in [4] use the AST fingerprints to detect similar source code existences using exact matches' clustering.

(3) *Software metrics-based approaches*: Software metrics such as Chidamber and Kemerer's (CK's) OO metrics [7] and McCabe's cyclomatic complexity [12] have been used to extract source code specific information from various Open Source Software (OSS). Such software-specific information is used to develop large datasets (such as PROMISE repository [13]), which are used to train ML models to perform defectiveness estimation. Programming Construct (PROCON) metrics proposed by authors in [9] capture the usage patterns of programming constructs occurring in source code. The specific programming constructs are fetched by using the AST generated by parsing the source code. Authors [9] show that the defectiveness estimation performed using PROCON metrics and the state-of-the-art ML technique produce effective results and outperform the existing methods [26], [7], and [13].

Need and opportunity for newer methods: The fingerprint approaches have the limitations of high processing cost and storage requirement. In contrast, the AST-based techniques generally train the ML models using binary classifiers, such as SVM. The binary classifiers, however, focus on classifying the input source files as {defective, unpredictable} [9], but do not provide any information for the source files *unlikely to be defective*. In our previous work [19], we provide a method for estimating the source files *unlikely to be defective*, but the method used is slower and inefficient. Thus, a faster and efficient method is desirable. Further, to the best of our knowledge, there do not exist efficient algorithms for assisting code reviews. On the other hand, in our current work, we compute fixed-size vector representations of source code. Computing the similarity of two vectors is more efficient and speedy in comparison to that in the case of fingerprints.

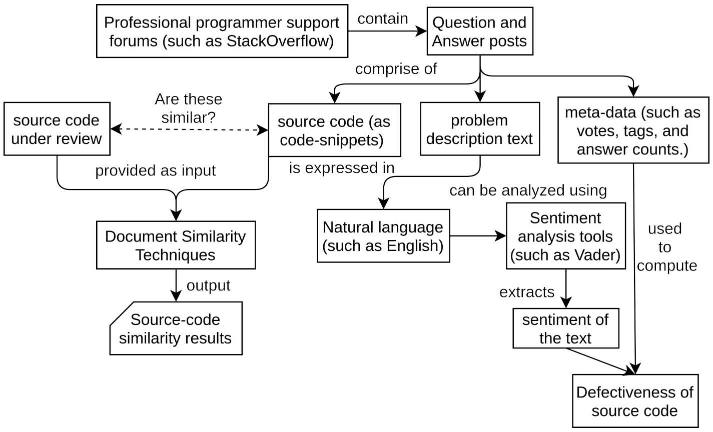

Professional programmer support forums (such as StackOverflow)
contain
Question and Answer posts
comprise of
source code under review
Are these similar?
source code (as code-snippets)
problem description text
meta-data (such as votes, tags, and answer counts.)
provided as input
is expressed in
can be analyzed using
Document Similarity Techniques
Natural language (such as English)
Sentiment analysis tools (such as Vader)
used to compute
output
extracts
Source-code similarity results
sentiment of the text
Defectiveness of source code

Figure 1: Basic idea of our approach

1.2 Basic tenets behind our system

The basic idea of our approach is stated as follows:

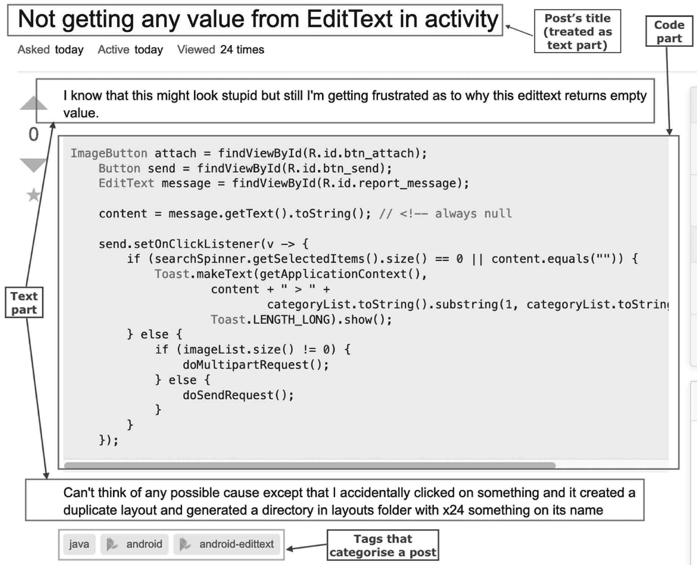
(1) Professional programmer support forums such as StackOverflow, contain useful information about programmers' problems when developing software. The information includes code fragments, the associated problem, and solution discussions in a natural language.

(2) If a *significant* part of given source code under review c' is found *sufficiently similar* to the code fragments c_p present in a SO post p, we can infer the *defectiveness* of the c' by analyzing the information available in p. We estimate a post's defectiveness by considering its natural language text, various metadata such as up-votes, and the post type.

1.3 Leveraging crowd-knowledge to identify problems in source code

Professional programmer support venues such as SO, provide a platform for programmers to discuss various problems related to software development. Figure 2 shows an example SO post. In this example, the programmer has posted a fragment of source code in which he faces some problem. The description of the problem is available in the post's narrative, which is written in English. Further, the posts are categorized with tags fields, which provide information about the technologies, or the platforms related to the post's code.

SO offers a rich and large corpus of such natural language discussions and the source code fragments discussed in software



Not getting any value from EditText in activity

Asked today Active today Viewed 24 times

0

I know that this might look stupid but still I'm getting frustrated as to why this edittext returns empty value.

```
ImageButton attach = findViewById(R.id.btn_attach);
    Button send = findViewById(R.id.btn_send);
    EditText message = findViewById(R.id.report_message);

    content = message.getText().toString(); // <!-- always null

    send.setOnClickListener(v -> {
        if (searchSpinner.getSelectedItems().size() == 0 || content.equals("")) {
            Toast.makeText(getApplicationContext(),
                    content + " > " +
                            categoryList.toString().substring(1, categoryList.toStrin
                    Toast.LENGTH_LONG).show();
        } else {
            if (imageList.size() != 0) {
                doMultipartRequest();
            } else {
                doSendRequest();
            }
        }
    });
```

Can't think of any possible cause except that I accidentally clicked on something and it created a duplicate layout and generated a directory in layouts folder with x24 something on its name

java android android-edittext

Post's title (treated as text part)
Code part
Text part
Tags that categorise a post

Figure 2: Example of a post on StackOverflow

development-related issues. Works such as [15, 16, 19] have shown that it is possible to exploit the crowd-knowledge available at SO for developing tools that address various software development tasks. Thus, our approach makes use of the rich volume of SO content to identify potential problems in a given source file, which is under review.

Association between software development and crowdsourced knowledge has been studied and confirmed by authors in [25], where they studied data from GitHub (an accessible repository of OSS) and StackOverflow. The type of questions that are asked and get answered or remain unanswered on StackOverflow has been explored by [23, 27]. There are mainly two types of posts on StackOverflow:

(1) Questions posted by programmers soliciting help and solutions for a programming or design problem they face with code or API. We refer to such posts as the *question posts.*
(2) Replies posted by other experts for the above type of posts. We refer to such posts as the *answer posts.*

The analysis of SO data shows[3] that the count of Type-2 posts is more than 1.5 times[4] the count of Type-1 posts. Further, we find[5] that more than 16% of Type-1 posts contain source code fragment(s). In contrast, more than 12% of Type-II posts[6] contain source code fragment(s). These SO posts typically describes some problems involving the source code fragment(s) included in the post.

Given the above, it can be argued that i) A code fragment accompanying a SO question is quite likely to be involved in a defect [23, 27], and ii) The code accompanying accepted or high scoring SO answers to such a question is quite likely to be free from the associated question post.

Challenges and opportunities: Though the SO provides a trove of information about the issues faced by professional programmers during software development, exploiting that information to build code review assistant tools poses several challenges. Major ones include:

(1) Accurate identification of the SO posts that contain source code fragments matching the input source file.
(2) Efficient retrieval of the matched or relevant SO posts.
(3) Accurately determining the defectiveness of SO posts.

Addressing the above challenges: While we note the above challenges, there exist techniques that can be exploited to address them. For an input source code to be reviewed (c'), we address the existing challenges:

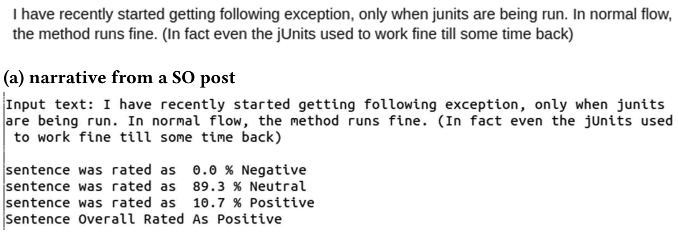
(1) *Identifying SO posts containing similar source code to c'*: The Paragraph Vector algorithm (PVA) [10] has delivered state-of-the-art results [5] in many Natural Language Processing (NLP) tasks that require a vector representation of text. One of this work's goals is to evaluate the effectiveness of the well-known PVA in computing an accurate representation of source code. Having such a representation of source code is useful for performing efficient and accurate source code comparisons.
(2) *Efficient retrieval of the matched SO posts (P'):* Most of the existing source code comparison methods, such as [19, 28], have high processing time and storage requirements. Thus, there is a need to provide a code review solution that accelerates the process and has a lower storage requirement.
(3) *Accurately determining the defectiveness of SO posts:* To determine the defectiveness of a SO post, one can analyze the post's narrative's sentiment. Tools such as CoreNLP [11] and Valence Aware Dictionary and sEntiment Reasoner (VADER) [8] can be used to infer the "sentiment" of a given input text, but may not prove to be effective when used in the context of some domain-specific narrative. For instance, consider the SO post narrative[7] shown in Figure 3a. The sentiment analysis tools, such as VADER, classify the post as positive with the sentiment score of 10.4%, which is thus a misclassification. The results obtained are shown in Figure 3b. Therefore, it is inadequate to rely on a SO post's narrative text to compute the code's defectiveness embedded in it solely.

2 PROPOSED APPROACH

Our system's primary goal can be stated as follows: *Given a source file f written in a programming language λ, determine if f is likely to have semantic issues.* The central idea behind our approach to addressing the above goal is to *look for any existing source code, which is $\underline{\text{sufficiently similar}}$ to the source code present in f, and is known to $\underline{\text{have a semantic issue}}$.*

Thus, two tasks become crucial for our approach a) determining the similarity of two source code samples, and b) establishing that the given source code is Likely-to-be-defective. SO is a widely used

[3] Our query is available at https://bit.ly/2JSSMez
[4] On StackOverflow (SO), there are more than 19m questions and 29m answers as of April 2020. See https://data.stackexchange.com/.
[5] Our query for finding this number is available here: https://bit.ly/3c4P79y.
[6] Our query for finding this percentage is available at: https://data.stackexchange.com/meta.stackoverflow/query/edit/1223740
[7] SO post considered as an example: https://bit.ly/2UvZXyg



I have recently started getting following exception, only when junits are being run. In normal flow, the method runs fine. (In fact even the jUnits used to work fine till some time back)

(a) narrative from a SO post

```
Input text: I have recently started getting following exception, only when junits
are being run. In normal flow, the method runs fine. (In fact even the jUnits used
 to work fine till some time back)

sentence was rated as  0.0 % Negative
sentence was rated as  89.3 % Neutral
sentence was rated as  10.7 % Positive
Sentence Overall Rated As Positive
```

(b) Results from VADER

Figure 3: An example of misclassification by the existing Sentiment Analysis tools

channel for professional programmer support. It offers a rich corpus of question and answers reply with relevant source code fragments.

Table 1: Table of Notation

L	$\triangleq$	The set of programming languages {C, C#, Java, JavaScript, Python}. We consider the source files written in any one of these.
G	$\triangleq$	Set of considered GitHub repositories, containing source files written in L.
S	$\triangleq$	Set of source files in G that are written in L.
M	$\triangleq$	The set of PVA models trained using S.
T	$\triangleq$	Test-bed used for testing the performance of M.
D	$\triangleq$	Database containing code, text, metadata and other computed items for SO posts.
R	$\triangleq$	The set of reference vectors chosen for programming languages $\lambda \in L$.
I	$\triangleq$	Set of metadata items of the SO posts.
P	$\triangleq$	Set of PVA parameter variation scenarios.
p_λ	$\triangleq$	An SO post containing k code fragments written in a language $\lambda \in L$. Here, $k > 0$.
c	$\triangleq$	A code fragment present in p_λ.
v	$\triangleq$	PVA computed vector representation of c.
α	$\triangleq$	Cosine similarity between two PVA vectors v and v'.
$\hat{\alpha}$	$\triangleq$	The threshold of cosine similarity between two PVA vectors to categorize them as similar.
μ	$\triangleq$	The threshold for score metadata field value of various SO posts.
ψ	$\triangleq$	No. of training samples used for training a PVA model.
γ	$\triangleq$	PVA vector size.
β	$\triangleq$	No. of training iterations or epochs used for training a PVA model.
χ	$\triangleq$	Sentiment score of different sentiment values provided as output by VADER.

Table-1 shows the notation used for various terms in this paper.

2.1 Steps in our approach

Figure 4 shows the architecture of the proposed system that implements our approach, and the critical steps in our approach are listed as follows. Along with each step, we highlight the relevant design decisions that were addressed when implementing those steps.

(1) **Preparing SO posts and code vectors**
  (a) Download a data dump of SO posts.
  (b) Extract the code, text, and metadata parts from each SO post p_λ and store in a database D.
  Design decision: How to decide whether a SO post and its content are relevant and useful?
  (c) Download source files from GitHub repositories, such that they are written in a programming language $\lambda \in L$.
  Design decision: Why only GitHub? How do we select a source file? Why only these programming languages?
  (d) For each language $\lambda \in L$, train PVA models using the samples from GitHub source files.
  Design decision: Why use GitHub source files for training? How to decide which files to choose from them? Why use PVA and how to choose the values of its tuning parameters?
  (e) For each SO posts' code fragment c_i available in D, compute its vector representation v_i using a suitable language-specific PVA model trained above. The vector is stored along with the corresponding code fragment in D itself.
  Design decision: Why use vector representations of source code?

(2) **Determining the defectiveness of source code under review (c')**
  (a) Compute the vector representation v' for the input source code c' after suitably preprocessing it.
  (b) Find all vectors $v \in D$ such that the similarity (α) between v and v' is above a similarity threshold ($\hat{\alpha}$).
  Design decision: What should be the value of $\hat{\alpha}$? On what factors does it depend?
  (c) For each $v \in D$, compute the defectiveness value, δ, using the narrative and metadata of the SO post p_λ of v.
  Design decision: How to compute δ?
  (d) The defectiveness value δ' for the input source code c' is computed by considering all δ of $\forall v \in D$.
  Design decision: How to compute δ'?

We discuss the crucial design decisions faced in our approach in the next subsection.

2.2 Design considerations in our approach

In the following subsections, we describe the details of the steps involved in developing our software artifacts and the rationale for design decisions addressed at each step.

2.2.1 Selection of the SO posts and the programming languages. The SO posts have mainly three types of content: i) questions, ii) answers, and iii) comments and metadata of the post. Further, two SO questions (or answers) may not have the same level of detail. For instance, a question post may have very little or no source code present in it. Alternatively, a post may have multiple code fragments written in different programming languages. Similar issues are present for other types of SO posts. Thus, it becomes essential to decide *if a SO post and its content are relevant and should be considered or not.*

Selecting the suitable SO posts: To address the above question, we adopt the following criteria for selecting SO posts for our use:

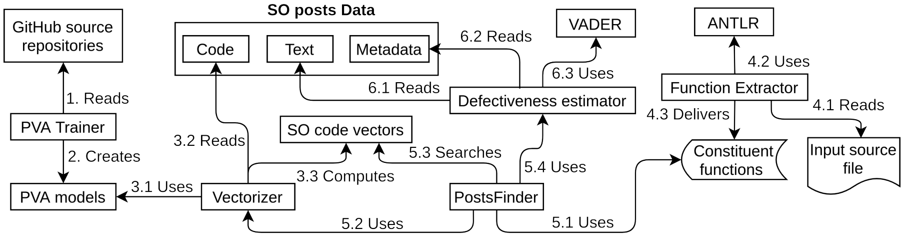


Figure 4: Architecture of the proposed system

(1) *Size constraint:* The size of the code fragment(s) present in the post should be greater than a certain threshold (> 100 characters without white spaces). We assume that any source code performing a logical function is of size above this threshold.
(2) *Tag constraint:* The post should be tagged (i.e. categorised) with at least one of the programming languages that we consider.

Selecting suitable programming languages: Following factors were considered when selecting the programming languages:

(1) We should cover multiple programming paradigms, such as object-oriented, procedural, and scripting.
(2) The languages should have a significant active deployment in the field.

After surveying the existing literature [22] and studying the online trends of developer-usage[8], we arrived at the set L = {C, C#, Java, JavaScript, and Python} of actively used programming languages.

2.2.2 *The rationale for choosing PVA.*

A crucial design decision that requires explanation is the use of PVA in our approach. The following are the main reasons for our choice of PVA: i) It allows us to compute vectors of the same length that accurately represents the source code samples. Keeping the length of such vectors the same for every source code sample is critical for implementing an efficient and fast system. ii) Our experiments show that the PVA works equally well for all the programming languages that we considered. iii) Recent works such as [2], a close variant of PVA, have proven that it is possible to compute accurate vector representations of source code and that such vectors can be very useful in computing semantic similarity between two source code samples. Thus, we chose the PVA to compute vector representations of source code samples in our approach.

2.2.3 *The rationale for using source code samples from GitHub and SO posts data.*

For training the PVA models, we used source code samples taken from various GitHub repositories. We used source code from GitHub repositories due to the following reason: To train the PVA models with realistic source code samples that we expect to encounter in real-world usage, we use source code samples from GitHub repositories. The source code samples taken from GitHub are syntactically complete units (e.g., a complete Java class instead of just a method definition).

A related question is "which source files to choose from GitHub?". We randomly selected repositories which met the following criteria:

Table 2: Details of the dataset used for training and testing of PVA models

	Training corpus (lines of code measured by cloc)				Testing corpus		
Language	Files	Blank	Comment	SLOC	File pairs	Corpus size	Models tested
C	32099	2908784	2490163	14908295	5000	30036	21
C#	8112	303416	198693	2342959	5000	7076	21
Java	142266	437851	659172	2157881	5000	127568	21
JavaScript	15737	177587	226724	1259902	5000	12485	21
Python	6012	300109	412452	1248494	5000	5378	20

(1) The repository had the source files written in a programming languages $\lambda \in L$ (see Table-1).
(2) The repository had earned 100 or more stars.
(3) The repository had more than 1000 source files.

Based on the above criteria, we selected about 105 different repositories[9] on GitHub from which the source files were taken for training and testing of the PVA models. Table 2 presents the detail of GitHub source files selected corresponding to various programming languages to train the PVA models. We use the *cloc* tool [6] to compute the count of comments, blank lines, and source lines of code (SLOC) present in the source code.

We use the SO posts data to create a reference dataset to perform the source code matching during the code review process. Using the PVA models, we obtain the vector representations for code fragments present in various SO posts and store them in a relational database to perform the vector comparisons. We choose the SO posts data for the following reason: The code present in SO posts is a mix of syntactically partial code fragments and full ones. For example, some posts contain the complete Java class definitions, while others may contain only a method definition or a small code block. On the other hand, the input code that we want to check for defectiveness will almost always be a syntactically complete unit of source code, such as a Java class or a Python module.

2.2.4 *Choosing PVA parameters and code similarity threshold, $\hat{\alpha}$.*

Performance, in terms of accuracy, storage efficiency, and response time, is determined by its input parameters such as β, γ, and ψ (see Table-1). Therefore, one of the key challenges in using PVA in our system is determining the minimum threshold value of α, indicating a significant similarity between two source code samples. Further, we would also like to select the optimal values of β, γ, and ψ that can result in such a value of $\hat{\alpha}$ (see Table-1). The details of the experiments performed to determine the optimally tuned values

[8] Sources of stats: https://githut.info/

[9] Details can be found in our dataset script available at http://bit.ly/2KJVWCh

of β, γ, ψ and, $\hat{\alpha}$ for obtaining the best performing PVA models are provided at https://bit.ly/2Ig3crd.

2.2.5 *Computing defectiveness, δ.*

The essential parameters considered while computing the defectiveness are:

(1) *The sentiment of a post's narrative:* This describes the view or opinion of the problem and is computed using the VADER sentiment analysis tool. A post's sentiment can be *positive*, *negative*, or *neutral*, depending on the problem's narration. A SO post consisting of a "negative" narration is most likely to describe a problem, thus comprising a defective code. Similarly, a SO post with a "positive" narrative is most likely to propose a solution code for a programming problem.

(2) *The score value of the post:* This is an integer value, available with every SO post, describing the approval or disapproval of the post by various viewers. An approval increments the score value, while the disapproval decrements it. A post with a high score value reflects a considerable confidence value in the post's content. For instance, the source code present in a high score answer post is likely to be free from defects.

(3) *The type of SO post:* A SO post can be classified as a question post or an answer post. A *question* post generally projects a programming problem containing a *source code defect*. On the contrary, an *answer* post mostly provides a solution source code, which is *unlikely-to-be-defective*.

To compute the defectiveness of a SO posts' code snippet c_p, we combine the results obtained based on p's metadata (viz., score (μ_p) and its post-type) and the sentiment information of the narrative in p. The complete procedure for computing the defectiveness (δ) of a code snippet c present in a SO post p is listed in Algorithm 1. To obtain the thresholds of SO post's score values, we compute the statistical measures, viz., maximum (*max*), minimum (*min*), average (*avg*), and standard deviation (*stddev*) of the respective score (μ) values of different types of SO posts. To find the source code's programming language present in a SO post, we use the *tag* metadata field. We choose the $avg(\mu)$ values under each of the language category and the post types as the respective thresholds. We represent the threshold values for question and answer posts as $avg(\mu_q)$ and $avg(\mu_a)$. By observing the threshold values, we select the $\langle avg(\mu_q), avg(\mu_a)\rangle$ as $\langle 1, 1.9\rangle$.

The defectiveness score computed based on the narrative sentiment of a SO post p is represented as $\delta_p^{narrative}$. The $\delta_p^{narrative}$ values assigned for the sentiment outcomes {negative, positive, neutral} are {-1,1,300}, respectively. We label a code snippet c_p as *unpredictable* if it has a "neutral" narrative sentiment of post p, and μ_p is below the respective post-type thresholds (μ_q or μ_a). Also, the δ values corresponding to various defectiveness labels {Likely-to-be-defective, Unlikely-to-be-defective, Unpredictable} are {-1,1,300}, respectively. To avoid the false negatives, we compute the *min* defectiveness score on Step 11 of Algorithm 1. We define the false negative as a case when a defective source code gets labeled as non-defective. A high value of $\delta_p^{narrative}$ for *neutral* sentiment value is chosen to consider the defectiveness inferred from the score metadata field of p.

Algorithm 1 Steps for computing δ of a code snippet c present in a SO post p

```
Require: $I$ = Set of metadata items associated with the StackOverflow post $p$.
    $t$ = The narrative text present in $p$.
    $\mu_q$, $\mu_a$ = The threshold score values for question and answer posts of SO respectively.
Ensure: $\delta_p$ = The defectiveness score for code snippet $c$ present in $p$.
 1: $\delta_p$ = 0
 2: if I(p.PostType) = question and I(p.Score) > $\mu_q$ then
 3:     $\delta_p$ = -1
 4: else if I(p.PostType) = answer then
 5:     if I(p.Score) > $\mu_a$ then
 6:         $\delta_p$ = 1
 7:     else
 8:         $\delta_p$ = -1
 9:     end if
10:     $\delta_p^{narrative}$ = computeNarrativeSentiment($t$){Using VADER}
11:     $\delta_p$ = min($\delta_p$, $\delta_p^{narrative}$)
12:     saveToDatabase($p.id$, $\delta_p$)
13: end if
```

2.3 Implementation details

We developed our software artifacts using the programming languages - Java and Python. These languages were selected because of the available expertise.

2.3.1 *Developing the SO posts database – SOpostsDB.*

SOpostsDB forms the knowledge base of our code review assistant system. The significant steps involved in developing the database are as follows:

(1) *Data Collection*: We downloaded a dump of SO posts [21] between July 2008 and March 2018 containing more than 5 million posts[10]. We extracted various sub-components (viz., code, text, and metadata) of these SO posts and store them in the database as the *SOpostsData* table. We also downloaded 105 OSS repositories from GitHub, containing different source files written in various programming languages. For developing the SOpostsDB, we considered the SO posts containing source code written only in five programming languages, viz., C, C#, Java, JavaScript, and Python. We collected about 188200 source files in total from GitHub written in these programming languages. We used the data collected from GitHub repositories to train our PVA models, while the source code extracted from SO posts to perform the source code matching with given source code. The link to access the training data and the SO posts details is provided at https://bit.ly/39KiA7l.

Figure 5 shows the relational schema of our database. A SO post p may consist of multiple code fragments c_i surrounded by different narratives t_i. We consider each of such c_i and the t_i preceding p as a single fragment and represent all such fragments with distinct `fragId`(s). *In other words, we consider the code fragments the basic unit of source code comparison and*

[10] https://archive.org/download/stackexchange

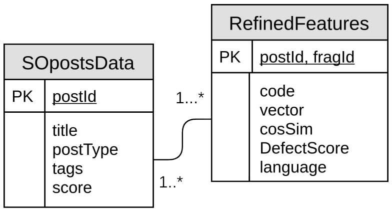


Figure 5: Relational Schema of SOpostsDB

thus obtain the PVA vectors corresponding to them. We consider the SOpostsData and CodeVectors as two separate tables because *a)* the SOpostsData comprises of the initial preliminary information required to obtain the information present in the CodeVectors table, and *b)* The SOPostsData majorly comprises of the metadata fields of a post which are common to all the code fragments present in a post.

(2) *Storing the vector representations of the code fragments present in SO posts*: The code fragments extracted from SO posts are preprocessed and stored in the database. To expedite the process of source code comparison, we perform the following:
   (a) Using language-specific PVA models, we obtain the vector representations of the extracted code fragments and store them. We use the python implementation of the `gensim` library [29] to implement PVA, preprocess the source code, and obtain the vector representations corresponding to them. The vector representations obtained for the code fragments present in various SO posts are stored in the database' *CodeVectors* table.
   (b) For each of the considered programming languages $\lambda \in L$, we select a PVA vector and refer to it as a *reference vector*. To select the reference vectors, we order the vector representations v present in the database using the `postId` and the `fragId` of the records and select the first entry for each of the languages as the reference vectors v_r.
   (c) We store the cosine similarity[11] (`cosSim`) of the PVA vectors (obtained in step 2a) with the respective reference vectors v_r. We use the `sklearn` library's python implementation [14] to compute the cosine similarity measure values between two vector representations.

   The idea behind storing the cosine similarity with the reference vectors: Consider the following scenario:
   (a) If a PVA vector v of a code fragment c present in the database, has a high cosine similarity score α with the reference vector v_r, and
   (b) The PVA vector v' of an input source code c' also represents a high cosine similarity score with v_r, then
   (c) The code fragments c and c' are likely to have a high degree of code similarity between them.

   The queries to the *CodeVectors* table for finding vectors that are "cosine similar" to a given vector are expensive if done on the vectors. To make such queries efficient, we pre-compute each SO code vector's cosine similarity with a language-specific reference vector and store it in the database. The cosine similarity value is a scalar quantity, and we can create an index on this relational attribute to speed up the database queries.

   A smaller version of our database, comprising the *CodeVectors* table and a subset of metadata fields (the SO post's title field) in *SOPostsData*, used by our tool, is shared https://bit.ly/2xs8CtV.

2.3.2 Developing the Code review system using SO and PVA – CRUSO-P.

CRUSO-P is an automated solution for assisting the code review activity. For providing this assistance, CRUSO-P utilizes the knowledge accumulated in SOpostsDB and code-similarity models trained using PVA. For a given input source code c', CRUSO-P labels c' as {Likely-to-be-defective, UnLikely-to-be-defective, Unpredictable}. The major sub-modules driving CRUSO-P and their implementation details are described next.

(1) *PVA Trainer*: This module is responsible for developing the PVA models. The best performing model corresponding to each of the programming languages $\lambda \in L$ is found by running several experiments. To find the optimal values of the input parameters β, γ, and ψ, the PVA trainer learns different models built using various parameter combinations. The `doc2vec` function of the `gensim` library is used to learn various PVA models. Several experiments were performed to obtain the optimal values of β, γ, and ψ, and to determine the similarity threshold values $\hat{\alpha}$ for source code written in different programming languages. We deem two source files as highly similar or identical when the PVA similarity score (α) is higher than a threshold value, $\hat{\alpha}$. For each of the considered programming languages $\lambda \in L$, the best performing models are found using various evaluation metrics. The details of the experiments performed to obtain the best performing PVA models are discussed at https://bit.ly/2Ig3crd.

(2) *Vectorizer:* This module is used to obtain the vector representations of code fragments present in the SO posts. The fixed-length vector representation of source code improves storage utilization and makes the search and retrieval process efficient. The best performing PVA models corresponding to each of the programming languages $\lambda \in L$ are used to obtain the vector representations. The `infer_vector` function of the `gensim` library is used to obtain the vector representations.

(3) *Posts Finder:* This is the main module that interacts with the front-end tool to provide the defectiveness estimates for an input source code c'. The SOPostsDB used for performing the source code matching comprises the vector representations corresponding to the code fragments present in the SO posts. The code fragments are generally of the form of code blocks or function bodies. For the reviewed input source files, *we consider a function-definition as the basic unit of source code comparison.*

   For the input source files that are reviewed, *we consider a function-definition as the basic unit of source code comparison.*

   Posts Finder uses the *Function extractor* to obtain the constituent functions (W) present in an input source code c'. For each of the obtained constituent functions $\omega \in W$, Posts Finder performs the following:

[11] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html

(a) It uses the *vectorizer* module to obtain the PVA vector corresponding to ω, say v_ω.
(b) It obtains the cosine similarity score α between v_ω and the language-specific reference vectors v_r.
(c) It fetches the top K matching code fragments from the database. The matching code fragments are obtained by fetching the top-K PVA vectors V having α' closest to α. We set K = 5 for our tool.
(d) Uses the *defectiveness estimator* to obtain the matching code fragments' defectiveness estimates and thus of ω.

The final estimate on the defectiveness of f is taken by performing a majority vote on the constituent functions' defectiveness estimates. *We compute the majority vote defectiveness estimate by computing the statistical* mode *of the defectiveness values obtained for the constituent functions W of f*. The complete procedure followed in computing the defectiveness of the input source code c' is listed in Algorithm 2.

(4) The *function extractor:* This module's goal is to extract the constituent function definitions present in an input source code. The *Function extractor* parses the input source code using *ANTLR*[12] and builds a custom *Listener* by modifying the function or method call event definitions. We use the Java programming language to build this module and transform it into a JAR executable using the Apache Maven Shade plugin[13]. This component works as a back-end module in our tool.

(5) *Defectiveness estimator*: This module is used to obtain the defectiveness estimates of the code fragments present in SO posts. For an input code fragment c of a SO post p, the *defectiveness estimator* reads the narration and the metadata fields. It uses the VADER tool to compute the narrative sentiment, which can be {positive (pos), negative (neg), or neutral (neu)}. For an input text t, VADER returns a sentiment score (χ) associated with these sentiment values. The decision function (ξ) used to compute the final sentiment value for t is as follows:

$$\xi[t] = \begin{cases} \text{positive}, & \text{if } \chi[\text{pos}] > \chi[\text{neg}] >= 0.5 \\ \text{negative}, & \text{if } \chi[\text{neg}] > \chi[\text{pos}] >= 0.5 \\ \text{neutral}, & \text{if } \chi[\text{neu}] >= 0.5 \text{ and } \chi[\text{pos}] < 0.5 \text{ and } \chi[\text{neg}] < 0.5 \end{cases}$$

The complete procedure to compute the defectiveness of c is listed in Algorithm 1.

2.3.3 Testing the defectiveness of source code using the CRUSO-P tool.

CRUSO-P provides a file-uploading interface to the end-user to submit the file to be reviewed. On submitting the source-file f to be reviewed, CRUSO-P outputs the defectiveness decision about f and provides the top matching SO posts results. The complete testing procedure used by CRUSO-P to detect the defectiveness of f is listed in Algorithm 2. CRUSO-P uses the PVA models, and the vector database of SO posts provided as input in this testing procedure. The database also contains the necessary metadata information, such as the type of posts, score of various SO posts, and the defectiveness of various SO posts (computed using Algorithm 1).

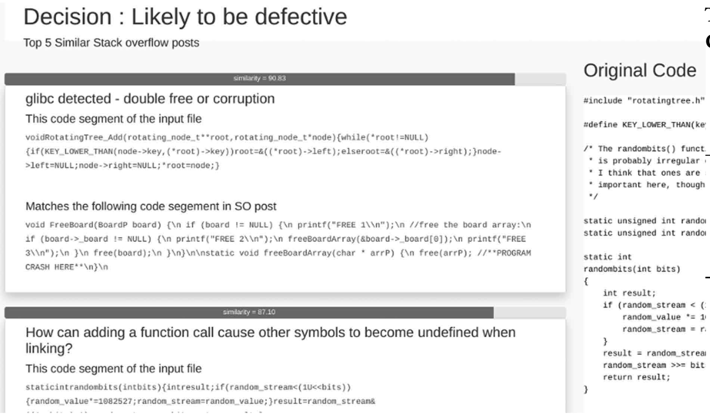
Algorithm 2 Steps for detecting defectiveness δ associated with a source-file f

```
Require: $f$ = Source file to check for defectiveness.
    $\lambda$ = Programming language in which $f$ is written.
    $M$ = Set of PVA models trained for various programming languages ($\forall \lambda \in L$).
    $R$ = Set of reference vectors chosen for various programming languages $L$.
    $SOPostsDB$ = Database containing the vector representations of code
    fragments and metadata information of SO posts.
Ensure: $\delta_f$ = The defectiveness score for source file $f$.
  $\delta_f = 0$
  Z = $\phi$
  $M_\lambda$ = fetchAndLoadModel($M, \lambda$){read from the local file system}
  $R_\lambda$ = fetchRefVector($R, \lambda$){a query into the SOpostsDB}
  W = parseAndObtainFunctions($f$){using Function Extractor}
  for all code fragment $\omega \in W$ do
    $v_\omega$ = obtainVectorRep($\omega, M_\lambda$){using Vectorizer}
    $\alpha_\omega$ = obtainCosSim($v_\omega, R_\lambda$)
    $C$ = fetchTopMatchCodeFrags($\alpha_\omega$, SOPostsDB)
    {Every $c \in C$ has $\langle postId, fragId \rangle$}
    for all code fragment $c \in C$ do
      $\delta_c$ = obtainDefectivenessEstimate(c.postId, SOPostsDB){using Algorithm 1}
      Z = Z $\cup$ $\langle \delta_c \rangle$
      $\delta_f$ = computeMajorityVoteDecision(Z){by computing statistical Mode(Z)}
    end for
  end for
```

The complete testing procedure is listed in Algorithm 2. Figure 6 shows an example of the usage of our tool. Here, we test the tool with an input source file containing the code fetched from GitHub repository cpython[14]. The figure shows the defectiveness results, the matching code fragments C, the associated similarity score, and defectiveness estimates. From the results shown in the figure, 4/5 matching code fragments depicted the defectiveness estimates as *Likely-to-be-defective*, and thus the input post was marked as *Likely-to-be-defective* as per the majority vote criterion.

It can be validated from the associated defect report link[15] that the source file contains defects, and validates our tool's results. Our tool (CRUSO-P) can be accessed at https://bit.ly/2V80NCT. For a given input source file f, CRUSO-P outputs the top matching SO posts' code fragments with their defectiveness estimates and the similarity scores.

[12] https://www.antlr.org/
[13] http://maven.apache.org/plugins/maven-shade-plugin/examples/executable-jar.html
[14] https://bit.ly/2RyxYxe
[15] https://bit.ly/2yf3RnO



Decision : Likely to be defective

Top 5 Similar Stack overflow posts

similarity = 90.83

glibc detected - double free or corruption

This code segment of the input file

```
voidRotatingTree_Add(rotating_node_t**root,rotating_node_t*node){while(*root!=NULL)
{if(KEY_LOWER_THAN(node->key,(*root)->key))root=&((*root)->left);elseroot=&((*root)->right);}node-
>left=NULL;node->right=NULL;*root=node;}
```

Matches the following code segement in SO post

```
void FreeBoard(BoardP board) {\n if (board != NULL) {\n printf("FREE 1\\n");\n //free the board array:\n
if (board->_board != NULL) {\n printf("FREE 2\\n");\n freeBoardArray(&board->_board[0]);\n printf("FREE
3\\n");\n }\n free(board);\n }\n}\n\nstatic void freeBoardArray(char * arrP) {\n free(arrP); //**PROGRAM
CRASH HERE**\n}\n
```

similarity = 87.10

How can adding a function call cause other symbols to become undefined when linking?

This code segment of the input file

```
staticintrandombits(intbits){intresult;if(random_stream<(1U<<bits))
{random_value*=1082527;random_stream=random_value;}result=random_stream&
```

Original Code

```
#include "rotatingtree.h"

#define KEY_LOWER_THAN(ke

/* The randombits() funct
 * is probably irregular
 * I think that ones are
 * important here, though
 */

static unsigned int rando
static unsigned int rando

static int
randombits(int bits)
{
    int result;
    if (random_stream < (
        random_value *= 1
        random_stream = r
    }
    result = random_strea
    random_stream >>= bit
    return result;
}
```

Figure 6: Partial view of CRUSO-P code review results

3 PERFORMANCE EVALUATION AND COMPARISON

One of this work's critical goals is to *significantly improve the speed, efficiency, and accuracy of our previous work* [19]. We have achieved this goal by improving the approach for detecting the relevant SO posts for f. Our tool determines the relevant SO posts by comparing the cosine similarity among the vector representations of two source codes. The *vectorizer* module of CRUSO-P is responsible for producing the vector representation for an input source code using the pre-trained PVA models. Thus, the accuracy of this task depends on the performance of PVA models. To obtain the best performing PVA models among all the considered programming languages L, we performed various parameter tuning experiments (details provided at https://bit.ly/2Ig3crd).

CRUSO-P infers the defectiveness of an input source code f by analyzing the defectiveness of the similar code fragments present in SOPostsDB. Thus, the performance of CRUSO-P depends on two essential factors:

(1) Efficacy in detecting the SO posts containing similar code fragments
(2) Precision in computing the defectiveness of SO posts

Therefore, while evaluating the performance of CRUSO-P, we design our experiments around the above two factors. The salient research questions addressed in our experiments are listed below:

(1) What is the highest accuracy achieved by CRUSO-P? Is CRUSO-P inclined to any specific programming language?
(2) How does CRUSO-P perform in comparison to CRUSO?

We used the Python programming language to implement our experiments.

3.1 Evaluation metrics

(1) **Accuracy** is defined as:

$$Accuracy = \frac{Total\ number\ of\ correctly\ detected\ matches}{Total\ number\ of\ tested\ record\ pairs} \quad (1)$$

(2) **F1 Score** is the harmonic mean of precision and recall:

$$F1score = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (2)$$

Table 3: Threshold similarity scores and performance scores of CRUSO-P

	Thresholds		Performance		Number of SO posts	Time taken (in seconds)
Language	Avg(α)	StdDev(α)	Accuracy	F1 score		
C	0.963	0.0704	0.992	0.992	5000	402
C#	0.954	0.0979	0.8559	0.8365	5000	413
Java	0.97	0.0668	0.993	0.993	5000	389
JavaScript	0.967	0.0719	0.8766	0.8612	5000	451
Python	0.9617	0.0764	0.991	0.9909	5000	368

where

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \quad (3)$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \quad (4)$$

We used the implementation provided by the sklearn library to compute these evaluation metrics.

3.2 Evaluating the performance of CRUSO-P

To perform this experiment, we take a subset of code fragments present in our SOposts database and evaluate the performance of our tool in predicting the defectiveness of these code fragments. Our tool can be considered to be effective if it marks a *defective* code fragment as *defective*. and vice versa.

The research question addressed: *What is the highest accuracy achieved by CRUSO-P? Is CRUSO-P inclined to any specific programming language?*

3.2.1 Test-bed setup: For our experiment, we selected 5000 code fragments associated with SO posts in a random manner.

3.2.2 Procedure. The procedure to perform the performance evaluation is as follows:

(1) Test CRUSO-P with the code fragments present in the SO posts, and record the defect estimates provided by CRUSO-P.
(2) Compute the Accuracy score and the F1 score based on the defect estimates provided in the previous step.

3.2.3 Results and observations. Table 3 lists the threshold similarity scores and the evaluation metrics (Accuracy and F1 score) values obtained from the experiment.

- *Observations:* The salient observations from the experiment are:
  - All the $\hat{\alpha}$ values for are the cases are above 95%.
  - The highest accuracy of 99.3% is achieved with source code written in $Java$ programming languages.
  - The accuracy values and F1 score values for all the languages are generally above 86%.
- *Inference:* CRUSO-P performs equally well for all the programming languages.

3.3 Comparison of CRUSO-P with CRUSO

One of this work's key objectives is *to significantly improve our previous work's* [19] *speed and efficiency.* When dealing with code reviews, a significant problem is the amount of time spent performing

Table 4: Performance comparison of CRUSO-P and CRUSO

Tool	Programming Language vs. Response time (in seconds)					Avg. Response time (in seconds)	Storage (in MBs)	Accuracy
	C	C#	Java	JavaScript	Python			
CRUSO-P	1.09	13.15	11.47	4.35	1.35	6.28	121.53	99.3%
CRUSO	284.74	291.09	289.15	281.81	292.8	287.92	14239	94%

them [17, 20]. A code review assisting tool that provides accurate estimates but takes very long to deliver them will be practical of minimal use. Therefore, in this experiment, we evaluate the performance of our CRUSO-P in terms of *response time* and *memory usage*, with its previous version CRUSO.

A short recap of our previous Code Review Assisting tool –CRUSO [19]: For an input source code *c*, CRUSO uses the Winnowing algorithm to identify SO posts containing code fragments similar to *c*, and analyzes the content of these relevant posts to estimate the defectiveness of *c*. CRUSO-P, in comparison to CRUSO, replaces the Winnowing algorithm with PVA. With PVA, the source code representation changes from the variable-length fingerprints to the fixed-length vectors. This experiment intends to investigate how this change in source code representation affects our tool's performance.

The research question addressed: *How does CRUSO-P perform in comparison to CRUSO?*

3.3.1 Test-bed setup. To perform this experiment, we implemented the existing approach [19] for the SO posts containing source code written in the considered set of programming languages, viz., C, C#, Java, Python, and JavaScript. To compare these tools' performance, we selected a random sample of 50 source files for each of the considered programming languages from different GitHub repositories (discussed in §2.2.3). We performed this experiment with the help of a group of programmers involved in developing software projects.

3.3.2 Procedure. The key steps involve the following:

(1) Compute the source code fingerprints for all the code fragments present in the considered SO posts. We use the Winnowing algorithm to perform this step.
(2) The obtained fingerprints are populated as a database table named as *winnow*.
(3) Compare the storage used by the *winnow* table and the *vectors* table of SOpostsDB.
(4) Compare the response time of CRUSO and CRUSO-P on testing with the selected random samples.

3.3.3 Results and Observations. The salient observations are:

- CRUSO-P has an average response time of 6.28 seconds, while the prior one based on Winnowing has 287.92 seconds.
- The vectors table for CRUSO-P occupies 121.53 MBs, while the CRUSO's Winnowing table occupies of 14239 MBs.

Inference: CRUSO-P achieves a speed improvement of 97.82% and a storage reduction of 99.15% over CRUSO. The highest accuracy achieved by CRUSO-P is 99.3% and 94% in CRUSO [19]. Therefore, CRUSO-P achieves an improvement of 5.6% in terms of accuracy when compared with CRUSO.

3.4 Threats to validity

As observed from the experiments, the PVA models' accuracy depends on the training data's nature. Thus the performance of the tool might vary if trained on a different dataset. The PVA models are trained on the source files written in languages {C, C#, Java, Python, JavaScript}. Therefore, CRUSO-P can detect the defectiveness of the source files written in these programming languages only. However, we can extend this approach to other languages as well.

Further, while designing the function extraction interface based on ANTLR, we could not find the ANTLR grammars of C# and JavaScript. Therefore, C# and JavaScript, the *function extractor* passes the input source code content to CRUSO-P for source code matching.

While performing source code matching to detect the relevant SO posts, we considered the code fragments with the length >= 100 characters (excluding the white spaces). We assume that the source code below this length would not represent any proper functionality, which also helps remove outliers from our dataset and remove the dataset's swamping effect.

Definition 9: *Swamping effect* is defined as the situation where "clean" data is incorrectly labeled as an outlier due to multiple clean sub-groupings within the data [3].

A SO post generally comprises of multiple code fragments surrounding by text descriptions. One of the parameters that we use to infer the source code's defectiveness in SO is the text description present in the post. Thus, there arises a need for mapping the code fragments with the constituent text fragments in various SO posts. While implementing this mapping procedure, we assume that "A text description preceding a code snippet *c* describes the nature of *c*." The SO posts not adhering to this structuring of text and code fragments might result in false positives.

Further, to compute the final defectiveness estimate, we use the *majority vote principle* over the matching code-fragments' defectiveness estimates. However, in the case of safety-critical software, there exists merit in being conservative. In that case, instead of the majority vote principle, it is safer to report the source code is *likely to be defective* if there exists even a single defective code match.

4 CONCLUSION

Code review is an essential software quality assurance activity, intended to find software defects and to estimate the software quality. The existing code review methods are slow and inefficient. We present a novel tool – CRUSO-P, which acts as a code review assistant for a programmer and helps in augmenting code reviews based on the information collected from SO posts. CRUSO-P works by determining the code similarity between the SO code fragments and the source code submitted as input. CRUSO-P leverages the PVA vector representations of source code present in SO posts to perform the code matching, thereby achieving an improvement of 97.82% in response time and a storage reduction of 99.15%, over one of the SOA tools. CRUSO-P achieves the best accuracy of 99.6% in case of models trained on the C programming language. CRUSO-P and the vectors database can be used for building software tools in related application areas such as defectiveness estimation, code review and recommendation. Our results show that CRUSO-P outperforms the existing methods based on Winnowing algorithm and source code fingerprints.

REFERENCES

[1] Junaid Akram, Zhendong Shi, Majid Mumtaz, and Ping Luo. 2018. Droidcc: A scalable clone detection approach for android applications to detect similarity at source code level. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 100–105.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 40:1–40:29.

[3] Jung-Tsung Chiang et al. 2007. The masking and swamping effects using the planted mean-shift outliers models. *Int. J. Contemp. Math. Sciences* 2, 7 (2007), 297–307.

[4] Michel Chilowicz, Etienne Duris, and Gilles Roussel. 2009. Syntax tree fingerprinting for source code similarity detection. In *2009 IEEE 17th International Conference on Program Comprehension.* IEEE, 243–247.

[5] Andrew M. Dai, Christopher Olah, and Quoc V. Le. 2015. Document embedding with paragraph vectors. In *NIPS Deep Learning Workshop*.

[6] Al Danial. 2017. *Count lines of code (cloc).* Retrieved April 15, 2020 from https://github.com/AlDanial/cloc

[7] Martin Hitz and Behzad Montazeri. 1996. Chidamber and Kemerer's metrics suite: a measurement theory perspective. *IEEE Transactions on software Engineering* 22, 4 (1996), 267–271.

[8] Clayton J Hutto and Eric Gilbert. 2014. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth international AAAI conference on weblogs and social media.*

[9] Ritu Kapur and Balwinder Sodhi. 2020. A Defect Estimator for Source Code: Linking Defect Reports with Programming Constructs Usage Metrics. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 1–35.

[10] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. 1188–1196.

[11] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

[12] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[13] Tim Menzies, Bora Caglayan, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. 2012. The promise repository of empirical software engineering data.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[15] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 1295–1298.

[16] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 102–111.

[17] Sasha Rezvina. 2019. Keep Code Review from Wasting Everyone's Time: Code Climate. https://codeclimate.com/blog/time-wasting-code-review/. Retrieved: 05-07-2020.

[18] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data.* 76–85.

[19] Shipra Sharma and Balwinder Sodhi. 2019. Using Stack Overflow content to assist in code review. *Software: Practice and Experience* 49 (2019), 1255–1277.

[20] Smartbear. 2019. The 2019 State of Code Review: Trends and Insights into Collaborative Software Development. https://static1.smartbear.co/smartbearbrand/media/pdf/the-2019-state-of-code-review.pdf. Retrieved: 05-07-2020.

[21] StackExchange. 2019. *Files for stackexchange.* https://archive.org/download/stackexchange

[22] StackOverflow. 2019. *StackOverflow Developer Survey Results 2019: Most Popular Technologies.* Retrieved Mar 24, 2020 from https://insights.stackoverflow.com/survey/2019#technology

[23] C. Treude, O. Barzilay, and M. A. Storey. 2011. How do programmers ask and answer questions on the web?: NIER track. In *33rd International Conference on Software Engineering (ICSE)*. 804–807.

[24] Zoran Đurić and Dragan Gašević. 2013. A source code similarity system for plagiarism detection. *Comput. J.* 56, 1 (2013), 70–86.

[25] B. Vasilescu, V. Filkov, and A. Serebrenik. 2013. StackOverflow and GitHub: Associations between Software Development and Crowdsourced Knowledge. In *2013 International Conference on Social Computing*. 188–195.

[26] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 297–308.

[27] Shaowei Wang, David Lo, and Lingxiao Jiang. 2013. An Empirical Study on Developer Interactions in StackOverflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (Coimbra, Portugal) *(SAC '13)*. ACM, New York, NY, USA, 1019–1024.

[28] Thomas Zimmermann and Nachiappan Nagappan. 2009. Predicting defects with program dependencies. In *2009 3rd international symposium on empirical software engineering and measurement*. IEEE, 435–438.

[29] Radim Řehůřek. 2019. *gensim – topic modelling for humans.* https://radimrehurek.com/gensim/parsing/preprocessing.html#gensim.parsing.preprocessing.preprocess_string