BloatLibD: Detecting Bloat Libraries in Java Applications

Agrim Dewan[1], Poojith U. Rao[2], Balwinder Sodhi[2] and Ritu Kapur[2][a]

[1]*Department of Computer Science and Engineering, Punjab Engineering College, Chandigarh, India*

[2]*Department of Computer Science and Engineering, Indian Institute of Technology Ropar, Punjab, India*

Keywords: Third-party Library Detection, Code Similarity, Paragraph Vectors, Software Bloat, Obfuscation.

Abstract: Third-party libraries (TPLs) provide ready-made implementations of various software functionalities and are frequently used in software development. However, as software development progresses through various iterations, there often remains an unused set of TPLs referenced in the application's distributable. These unused TPLs become a prominent source of software bloating and are responsible for excessive consumption of resources, such as CPU cycles, memory, and mobile devices' battery-usage. Thus, the identification of such bloat-TPLs is essential. We present a rapid, storage-efficient, obfuscation-resilient method to detect the bloat-TPLs. Our approach's novel aspects are *i)* Computing a vector representation of a .class file using a model that we call Jar2Vec. The Jar2Vec model is trained using the Paragraph Vector Algorithm. *ii)* Before using it for training the Jar2Vec models, a .class file is converted to a *normalized* form via semantics-preserving transformations. *iii)* A *Bloat*ed *Lib*rary *D*etector (BloatLibD) developed and tested with 27 different Jar2Vec models. These models were trained using different parameters and >30000 .class files taken from >100 different Java libraries available at MavenCentral.com. BloatLibD achieves an accuracy of 99% with an F1 score of 0.968 and outperforms the existing tools, viz., LibScout, LiteRadar, and LibD with an accuracy improvement of 74.5%, 30.33%, and 14.1%, respectively. Compared with LibD, BloatLibD achieves a response time improvement of 61.37% and a storage reduction of 87.93%. Our program artifacts are available at https://bit.ly/2WFALXf.

1 INTRODUCTION

The development of a non-trivial software application invariably involves the use of Third Party Libraries (TPLs), which provide ready-made implementations of various functionalities, such as image manipulation, data access and transformation, and advertisement. According to (Wang et al., 2015), 60% of an application's code is contributed by TPLs. However, the unused TPLs present in an application waste almost half of the CPU cycles and memory. Such unused TPLs have become a prominent source of software bloat. We refer to such unused TPLs as bloat-TPLs. Resource wastage is a critical problem for mobile devices that possess limited computing resources and significantly impact the performance by affecting the execution time, throughput, and scalability of various applications (Mitchell and Sevitsky, 2007; Xu et al., 2010). Therefore, the **identification and removal of the bloat-TPLs** present in an application are desirable.

Definition 1 (Bloat-TPL)**.** *We define a **bloat-TPL** as the one bundled in the distributable binary of a software application A but not relevant to it. The examples of such TPLs would be the Mockito[a] or JUnit[b] Java ARchives (JARs) that get packaged in the deployable release archive of a Java application.*
The relevance of a TPL is application-specific. For instance, relevant vs. irrelevant, reliable vs. unreliable, anomalous vs. non-anomalous, etc. The idea is to compare with a reference list of relevant libraries or white-listed libraries[c] in an automated manner.

[a]https://site.mockito.org/
[b]https://junit.org/
[c]Black libraries matter

[a] https://orcid.org/0000-0001-7112-0630

Dewan, A., Rao, P., Sodhi, B. and Kapur, R.
BloatLibD: Detecting Bloat Libraries in Java Applications.
DOI: 10.5220/0010459401260137
In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2021)*, pages 126-137
ISBN: 978-989-758-508-1

1.1 Motivation

Our primary objective is to develop a technique for detecting *bloat-TPLs* present in a software application's distributable binary. An essential task for achieving this objective is to *develop a robust technique for TPL-detection*. The existing TPL-detection methods (Backes et al., 2016; Ma et al., 2016) generally depend, directly or indirectly, on the package names or the library's structural details and code. Thus, they are potentially affected by various obfuscations, such as package-name obfuscation and API obfuscation. Also, most of the works are restricted to Android applications (Zhang et al., 2018; Ma et al., 2016; Li et al., 2017; Backes et al., 2016).

We propose an obfuscation-resilient tool that detects the TPLs present in a given application by identifying the similarities between the source code of the "available TPLs" and the TPLs used in the given application. To obtain this set of "available TPLs," we leverage the TPLs present at MavenCentral repository[2] that hosts TPLs for various functionalities, which we assume to be a trustworthy source. Another important goal of our work is to evaluate the Paragraph Vector algorithm (PVA) (Le and Mikolov, 2014) in computing a reliable and accurate representation of binary and textual code.

1.2 Basic Tenets Behind Our System

In this paper, we present a novel TPL-detection technique by creating a *library embedding* using PVA – we named it Jar2Vec. The central idea underlying our approach is illustrated in Figure 1 and stated as follows:

1. Each of the TPLs consists of a collection of binary .class files.
2. Semantics-preserving transformations (such as decompilation and disassembly) are applied to the binary .class files to obtain their *normalized* textual form(s), viz., the textual forms of source code and bytecode instructions.
3. With a large corpus of such text, we train Jar2Vec models, using which a vector representation of any .class file can be computed.
4. Further, the vector representations of a TPL can be computed as a suitable function of the vector representations of all the .class files contained in that TPL.
5. If the vector representations of a TPL J in an application, have a considerable cosine similarity[3] with the vector representations of the set of "available TPLs," we label J as *likely-to-be-non-bloat-TPL* or else *likely-to-be-bloat-TPL*.

[2]https://mvnrepository.com/repos/central

[3]http://bit.ly/2ODWoEy

1.3 Handling Obfuscated Libraries

One of the significant issues faced in TPL-detection is the obfuscation of the library code. The TPL-detection techniques that rely on the obfuscation-sensitive features of a library would fail to detect a library under obfuscation. The key idea underlying our approach towards handling obfuscated libraries is to produce a "normalized" textual representation of the library's binary .class files before using it to train the Jar2Vec models and when checking an input .class using a Jar2Vec model. We perform the decompilation and disassembly of .class files to obtain their "normalized" textual forms, viz., source code and bytecode instructions as text. These operations on a .class file are obfuscation-invariant. For example, we transform a .class file using a decompiler (Strobel, 2019) (with suitable configuration), which produces almost identical output for both obfuscated and unobfuscated versions. The decompiler can emit either bytecode or the Java source code for the .class files.

2 RELATED WORK

Most of the existing works that target TPL-detection assume that "*the libraries can be identified, either through developer input or inspection of the application's code.*" The existing approaches for TPLs detection can be categorized as follows:

1. *Based on a "Reference List" of TPLs:* The techniques presented in (Chen et al., 2014; Liu et al., 2015; Grace et al., 2012; Book et al., 2013) are significant works in this category. A "reference list" comprises libraries known to be obtained from a trustworthy source and useful for software development. The basic idea behind the approach is first to construct a "reference list" of libraries and then test the application under consideration using the list. In this process, it is evaluated that the application's constituent libraries are present in the "reference list" or not. All the constituent libraries, which are not present in the list, are deemed to be bloat-TPLs. In practice, this approach requires keeping the "reference list" up-to-date. Since these methods require manually comparing the libraries with the "reference list" and a periodic update of this list, they tend to be slower, costly, and storage-inefficient.

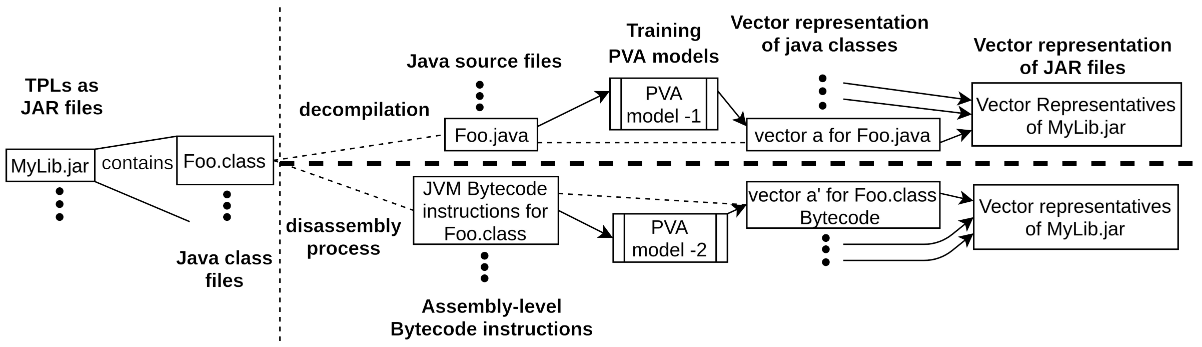

TPLs as JAR files
MyLib.jar
contains
Foo.class
Java class files
decompilation
disassembly process
Java source files
Foo.java
JVM Bytecode instructions for Foo.class
Assembly-level Bytecode instructions
Training PVA models
PVA model -1
PVA model -2
Vector representation of java classes
vector a for Foo.java
vector a' for Foo.class Bytecode
Vector representation of JAR files
Vector Representatives of MyLib.jar
Vector representatives of MyLib.jar

Figure 1: Basic idea of our method.

2. *Features-based Approaches:* (Backes et al., 2016; Ma et al., 2016; Li et al., 2017) are some of the approaches that work by extracting individual libraries' features and then use them to identify libraries that are similar to another. The feature-based methods generally depend, directly or indirectly, on the package names or the structural details of the application and source code. A brief description of these works is provided below:
   (a) **LibScout (Backes et al., 2016)** presents a TPL-detection technique based on Class Hierarchical Analysis (CHA) and hashing mechanism performed on the application's package names. Though the method has been proven to be resilient to most code-obfuscation techniques, it fails in certain corner cases. For example, modification in the class hierarchy or package names, or when the boundaries between app and library code become blurred. Another recent work is (Feichtner and Rabensteiner, 2019), which relies on the obfuscation-resilient features extracted from the Abstract Syntax Tree of code to compute a code fingerprint. The fingerprint is then used to calculate the similarity of two libraries.
   (b) **LibRadar (Ma et al., 2016)** is resilient to the package name obfuscation problem of LibScout and presents a solution for large-scale TPL-detection. LibRadar leverages the benefits of hashing-based representation and multi-level clustering and works by computing the similarity in the hashing-based representation of static semantic features of application packages. LibRadar has successfully found the original package names for an obfuscated library, generating the list of API permissions used by an application by leveraging the API-permission maps generated by PScout (Au et al., 2012). Though LibRadar is resilient to package obfuscation, it depends on the package hierarchy's directory structure and requires a library candidate to be a sub-tree in the package hierarchy. Thus, the approach may fail when considering libraries being packaged in their different versions (Li et al., 2017). An alternate version of LibRadar, which uses an online TPL-database, is named as LiteRadar.
   (c) **LibD (Li et al., 2017)** leverages feature hashing to obtain code features, which reduces the dependency on package information and supplementary information for TPL-detection. LibD works by developing library instances based on the package-structure details extracted using Apktool, such as the direct relations among various constituent packages, classes, methods, and homogeny graphs. Authors employ Androguard (Anthony Desnos, 2018) to extract information about central relationships, viz., inclusion, inheritance, and call relations. LibD depends upon the directory structure of applications, which leads to the possibility of LibD's failure due to obfuscation in the directory structure.

Limitations of the Current Works: While the TPL-detection based on "reference list" methods tend to be inefficient and costly, the feature-based methods are potentially affected by various types of obfuscations and are mostly developed for Android applications. Therefore, it is desirable to develop TPL-detection techniques that are resilient against such issues and can be used for applications not-limited to the Android platform.

3 PROPOSED APPROACH

Our system's primary goal can be stated as follows: Given a TPL J, determine if J is likely-to-be-bloat-TPL or a non-bloat-TPL in the given application. Our approach's central idea is to look for source code similarity and bytecode similarity between J and the set

Table 1: Table of Notations.

J	$\triangleq$	A TPL in its JAR file format.
C	$\triangleq$	The collection of JAR files fetched from MavenCentral.
Z	$\triangleq$	The set of PVA tuning-parameter variation scenarios, listed in Table-4.
F_{bc}, F_{sc}	$\triangleq$	The collections of bytecode (bc) and source code (sc) data obtained by disassembling and decompilation of .class files f, respectively, such that $f \in J$, and $J \in C$.
M_{bc}, M_{sc}	$\triangleq$	The collections of Jar2Vec models trained on F_{bc} and F_{sc} respectively, for all the scenarios in Z.
$\hat{M_{bc}}, \hat{M_{sc}}$	$\triangleq$	The best performing Jar2Vec models among M_{bc} and M_{sc}, respectively.
ϕ_{bc}, ϕ_{sc}	$\triangleq$	The PVA vectors corresponding to a .class file's bytecode and source code data, respectively.
$\phi_{sc}^{ref}, \phi_{bc}^{ref}$	$\triangleq$	The reference PVA vectors for source code and bytecode, respectively.
D	$\triangleq$	The database containing the .class files' vectors (ϕ_{sc} and ϕ_{bc}) for C.
β	$\triangleq$	The number of training iterations or epochs used for training a PVA model.
γ	$\triangleq$	The PVA vector size used for training a PVA model.
ψ	$\triangleq$	The number of training samples used for training a PVA model.
α	$\triangleq$	The cosine similarity score between two PVA vectors.
$\hat{\alpha}$	$\triangleq$	The threshold cosine similarity score.

of "available TPLs." However, analyzing the detailed usages of the TPLs in the application is currently out of scope of this work. Our method can be considered as similar to the "reference list" methods, but the similarity here is determined on the basis of source code present in the TPL, and not merely the TPL names or package-hierarchial structure. Table-1 shows the notation used for various terms in this paper.

3.1 Steps of Our Approach

The central ideas behind our approach were presented in §1.2. Here we expand those steps in more detail and highlight the relevant design decisions to be addressed while implementing each step.

1. **Preparing the Dataset of "Available TPLs".**
   (a) Download a set of TPLs C from MavenCentral.
   Design decision: Why use MavenCentral to collect TPLs? How many TPLs should be collected from different software categories?
   (b) For each TPL $J \in C$, obtain the Java source code and bytecode collections (F_{sc}, F_{bc}) by performing the decompilation and disassembly transformation operations.
   Design decision: Why are the decompilation and disassembly transformations appropriate?
   (c) Train the PVA models M_{sc} and M_{bc} on F_{sc} and F_{bc}, respectively, obtained in the previous step.
   Design decision: Why use PVA, and what should be the PVA tuning-parameters for obtaining optimal results in our task?
   (d) For each source file $f \in F_{sc}$ and the bytecode record $b \in F_{bc}$, obtain the corresponding vector representations (ϕ_{sc}, ϕ_{bc}) using suitable PVA models trained in the previous step. ϕ_{sc} and ϕ_{bc} obtained for each source code and bytecode instance are stored in the database D.
2. **Determining if an Input TPL (J) is a Bloat-TPL or not for a Given Application.**
   (a) Compute the vector representation $\langle \phi'_{bc}, \phi'_{sc} \rangle$ for the bytecode and source code representations of J.
   (b) Obtain all the vectors $\langle \phi_{bc}, \phi_{sc} \rangle \in D$, such that the respective similarity scores between $\langle \phi'_{bc}, \phi_{bc} \rangle$ and $\langle \phi'_{sc}, \phi_{sc} \rangle$ are above specific threshold values ($\hat{\alpha_{bc}}$ and $\hat{\alpha_{sc}}$).
   Design decision: What are the optimal values of similarity thresholds ($\hat{\alpha_{bc}}$ and $\hat{\alpha_{sc}}$)?
   (c) Determine whether J is a bloat-TPL or not for the given application.
   Design decision: How is the nature of J determined?

3.2 Design Considerations in Our Approach

In this section, we address the design decisions taken while implementing our approach.

3.2.1 Collecting TPLs from MavenCentral

The libraries used for training our models (named Jar2Vec) were taken from MavenCentral. We choose MavenCentral as it is a public host for a wide variety of popular Java libraries. MavenCentral categorizes the Java libraries based on the functionality provided by the libraries. However, our method is not dependent on MavenCentral; the TPLs could be sourced from reliable providers. To collect the Java libraries, we perform the following steps:

1. Crawl the page https://mvnrepository.com/open-source, and download the latest version of a JAR file for each of the top k libraries listed under each category. For our experiments, we choose k=3.
2. Extract and save in a database table the metadata associated with the downloaded JAR. The metadata includes details of the TPL, such as the category, tags, and usage stats.

3.2.2 Rationale for Choosing PVA for Training Models

We train Jar2Vec models using PVA on the source code and bytecode textual forms of the .class files obtained by the decompilation and disassembly of various TPLs. The key reasons for choosing PVA are i) It allows us to compute the fixed-length vectors that accurately represent the source code samples. Keeping the length of vectors same for every source code sample is critical for implementing an efficient and fast system. ii) Recent works such as (Alon et al., 2019), 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.

Tuning Parameters for PVA: Performance, in terms of accuracy, efficiency, and speed of PVA, is determined by its input parameters such as β, γ, and ψ (see Table-1). Therefore, one of the major tasks is to select the optimal values of β, γ, and ψ that can result in the best performing Jar2Vec models ($\hat{M}_{bc}$ and $\hat{M}_{sc}$). The performance of the Jar2Vec models is evaluated by measuring their accuracy in detecting the similarity in TPLs. The experiments' details to determine β, γ, and ψ are provided in the Appendix.

3.2.3 Rationale for using the Decompilation and Disassembly Transformations

It is necessary to derive a "normalized" and obfuscation-resilient textual form of the .class files to compute a reliable vector representation. The normalization applies a consistent naming of symbols while preserving the semantics and structure of the code. We use the decompilation (*giving a source code text*) and disassembly (*giving a bytecode text*) as transformations to extract such normalized textual forms of .class files.

3.2.4 Employing the Use of Vector Representations for Performing Similarity Detection between TPLs

To determine the similarity between libraries efficiently, we create a database (D) of vectors. These vectors correspond to the .class files present in a target repository of libraries (such as MavenCentral, or an in-house repository maintained by an organization). We obtain the vector representations for both the source code and bytecode of .class files present in TPLs using suitably trained PVA models and store them in D. The PVA vectors enable fast and efficient detection of TPL similarity.

3.2.5 Computing the Threshold Similarity Measure $\hat{\alpha}$

Our method detects two libraries' similarity by inferring the similarity scores for .class files contained in those libraries. To check if two .class vectors are similar or not, we compute their cosine similarity[4]. An important design decision in this context is:

For reliably detecting a library, what is the acceptable value of Jar2Vec similarity scores for decompiled source code and bytecode inputs?

We deem two .class files as highly similar or identical when the similarity score for the files is higher than a threshold value $\hat{\alpha}$. The value of $\hat{\alpha}$ is determined by running several experiments to measure similarity scores for independent testing samples. The details of the experiments are discussed in the Appendix.

3.2.6 Determining the Nature of an Unseen JAR File J for a Given Application A

To determine if a given JAR file (J) is "bloat-TPL" for an application (A), we leverage the best performing Jar2Vec models ($\hat{M}_{bc}$ and $\hat{M}_{sc}$) and the vectors database D.

If J contains .class files depicting considerable similarity ($\geq \hat{\alpha}$) with the "available TPLs," it is deemed to be as "likely-to-be-non-bloat" for A. If for at least N .class files in J, the similarity scores are $\geq \hat{\alpha}$, we label J as a *likely-to-be-non-bloat* TPL for A, else a *bloat-TPL*. In our experiments, we take N as half of the count of .class files present in J. For a more strict matching, a higher value of N can be set. The complete steps for the detection procedure are listed in Algorithm 1.

Selection of the Top-similar "Available TPLs": We explain it with an example. Suppose we have four "available TPLs", with $C := \{$M1.jar, M2.jar, M3.jar,

[4]https://bit.ly/2RZ3W5L

M4.jar}, such that these contain 15, 6, 10, and 10 .class files, respectively. Now, D will contain the PVA vectors corresponding to the source code and bytecode representations of all the .class files present in all the JARs in C. Next, suppose we want to test a JAR file Foo.jar that contains ten .class files, and that we have the following similarity scenarios:

1. All ten .class files of *Foo.jar* are present in M1.jar.
2. All six .class files of *M2.jar* are present in Foo.jar.
3. Seven out of ten .class files of *M3.jar* are present in Foo.jar.
4. For *M4.jar*, none of these files is identical to those present in Foo.jar, but they have similarity scores higher than the threshold.

Which of the JAR files (M1 – M4) listed above will be determined as the most similar to Foo.jar?

Our approach determines the most-similar JAR file by measuring the total number of distinct .class file matches. So with this logic, the similarity ordering for Foo.jar is M1, M4, M3, M2.

In this setting, determining the similarity of two JARs is akin to comparing two sets for similarity. Here the items of the *sets* would be the PVA vectors representing .class files. We apply the following approach to determine the TPL-similarity:

1. For each .class c in Foo.jar, find each record $r \in D$ such that the similarity score between c and r is higher than a threshold. Let $R \subset D$ denote the set of all such matched records.
2. Find the set Y of distinct JARs to which each $r \in R$ belongs.
3. Sort Y by the number of classes present in R.
4. Select the top-k items from Y as similar JARs to Foo.jar.

Algorithm-1 presents the above logic in more detail.

3.3 Implementation Details

The logical structure of the proposed system is shown in Figure 2. All components of the system have been developed using the Python programming language. Details of each of the components are as follows:

1. **JAR File Collector:** We developed a crawler program to collect the JAR files and the metadata associated with each JAR file. The files were downloaded from www.mavencentral.com, a reputed public repository of Open Source Software (OSS) Java libraries. MavenCentral has about 15 million indexed artifacts, which are classified into about 150 categories. Some examples of the categories include JSON Libraries, Logging Frameworks, and Cache Implementations. The metadata about each JAR includes the following main items: License, Categories, HomePage, Date of Availability, Files, Used By (count, and links to projects using a library). The JAR file categories' complete details, metadata collected, and the specific JAR files chosen can be found at *https://bit.ly/2WFALXf*.

2. **Transformation Handler:** This module provides the transformations and preprocessing of the .class files present in the input JAR files. Two types of transformations implemented are a) De-

Algorithm 1: Steps for determining the nature of a TPL J.

```
1:  Input: $J :=$ A TPL file provided as input by an end-user.
    $\hat{M_{bc}}, \hat{M_{sc}} :=$ The best performing Jar2Vec models.
    $\hat{\alpha_{sc}}, \hat{\alpha_{sc}} :=$ Threshold similarity scores for source code and bytecode.
    $\phi_{sc}^{ref}, \phi_{bc}^{ref} :=$ Reference PVA vectors for source code and bytecode.
    $D :=$ Database containing the vector representations of .class files in C.
2:  Output: $d :=$ Decision on the nature of $J$.
    /*Please see Table 1 for notation*/
3:  $S_{sc} := S_{bc} := NULL$
4:  for all .class files $f \in J$ do
5:      Obtain the PVA vectors $\phi'_{sc}$ and $\phi'_{bc}$ using $\hat{M_{bc}}$ and $\hat{M_{sc}}$.
6:      Query the database $D$ for top-k most similar vectors to $\phi'_{sc}$ and $\phi'_{bc}$.
7:      $\alpha_{sc}, \alpha_{bc} :=$ Compute the cosine similarity between $\langle \phi'_{sc}, \phi_{sc}^{ref} \rangle$ and $\langle \phi'_{bc}, \phi_{bc}^{ref} \rangle$.
8:      $S_{sc} := S_{sc} \cup \langle \alpha_{sc} \rangle$
9:      $S_{bc} := S_{bc} \cup \langle \alpha_{bc} \rangle$
10: end for
11:
```

$$d := \begin{cases} \text{non-bloat} & \text{if for at least } N \text{ .class file records in both } S_{sc} \text{ and } S_{bc} \text{ individually, } \alpha_{sc} > \hat{\alpha_{sc}} \text{ and } \alpha_{bc} > \hat{\alpha_{bc}} \text{ respectively.} \\ \text{bloat} & \text{otherwise} \end{cases}$$

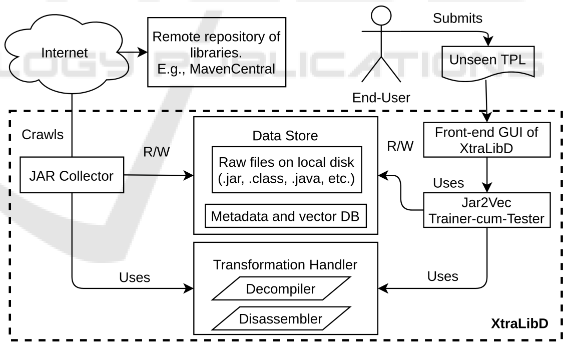
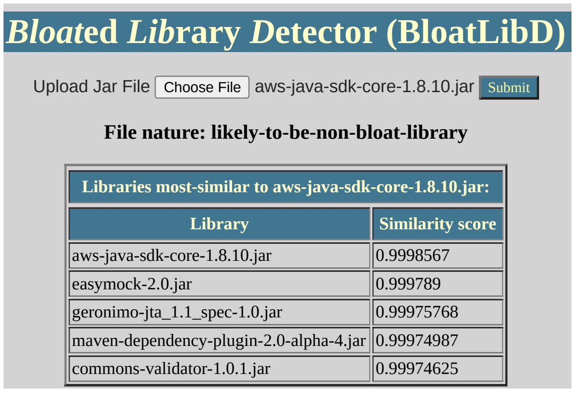


Figure 2: Logical structure of the proposed system.

compilation of the .class file to produce a corresponding Java source and b) Disassembling the .class files into human-readable text files containing Java Virtual Machine (JVM) bytecode instructions for the .class files.

We used the Procyon (Strobel, 2019) tool for performing the decompilation and disassembling of the .class files. The respective transformation output is further preprocessed to remove comments and adjust token whitespaces before storing it as a text file in a local repository. The preprocessing was done for decompiled Java source to ensure that the keywords and special tokens such as parentheses and operators were delimited by whitespace. The preprocessing provides proper tokenization of the source into meaningful "vocabulary words" expected by the PVA.

3. **Jar2Vec Trainer-cum-tester:** We use an open-source implementation of the PVA – called Gensim (Dai et al., 2015), to train our Jar2Vec models. The Jar2Vec trainer-cum-tester module's primary task is to:
   (a) Train the Jar2Vec models using bytecode and Java source files produced by disassembling and decompiling the .class files.
   (b) Infer the vectors for unseen .class files' bytecode and source code by using the respective models.
4. **Metadata and the Vectors' Database:** The information about libraries fetched from MavenCentral is stored in a relational database. The following are the essential data items stored in the database:
   (a) Name, category, version, size, and usage count of the library.
   (b) Location of the library on the local disk as well as a remote host.
   (c) For each .class file f in a JAR:
      i. The fully qualified name of the f.
      ii. Sizes of f, and the textual form of its decompiled Java source code (f_{sc}) and the disassembled JVM bytecode (f_{bc}).
      iii. Inferred PVA vectors $\langle \phi_{sc}, \phi_{bc} \rangle$ for the above files.
      iv. Cosine similarity scores α_{sc}^{ref} and α_{bc}^{ref} between $\langle \phi_{sc}, \phi_{sc}^{ref} \rangle$ and $\langle \phi_{bc}, \phi_{sc}^{ref} \rangle$, respectively. The values α_{sc}^{ref} and α_{bc}^{ref} are scalar.
5. **BloatLibD's GUI:** The user interface of BloatLibD is a web-based application. End-user uploads a TPL using this GUI, which is then processed by our tool at the server-side. The tool requires the TPLs in JAR format as input. Figure 3 displays the results for a test file[5] submitted to our tool. As shown by the figure, BloatLibD displays the input file's nature and the top-k (k=5) similar essential libraries along with the corresponding similarity scores. As we achieve higher accuracy with the source code Jar2Vec models than the bytecode models (discussed in the Appendix), we use the best performing source code Jar2Vec model for developing our tool.


Bloated Library Detector (BloatLibD)

Upload Jar File [Choose File] aws-java-sdk-core-1.8.10.jar [Submit]

File nature: likely-to-be-non-bloat-library

Libraries most-similar to aws-java-sdk-core-1.8.10.jar:

Library	Similarity score
aws-java-sdk-core-1.8.10.jar	0.9998567
easymock-2.0.jar	0.999789
geronimo-jta_1.1_spec-1.0.jar	0.99975768
maven-dependency-plugin-2.0-alpha-4.jar	0.99974987
commons-validator-1.0.1.jar	0.99974625

Figure 3: Top similar TPLs detected by BloatLibD.

4 EXPERIMENTAL EVALUATION

The primary goal of our experiments is to validate the correctness and accuracy of our tool – BloatLibD. The efficacy of our tool depends on its accuracy in performing the task of detecting similar TPLs. BloatLibD achieves this by detecting the similarity between the PVA vectors of the .class files present in the TPLs. The Jar2Vec models used by BloatLibD are responsible for generating different PVA vectors. Therefore, we perform various parameter-tuning experiments to obtain the best performing Jar2Vec models (discussed in the Appendix). To evaluate the performance of BloatLibD, we develop a test-bed using the TPLs collected from MavenCentral (discussed in §4.1) and perform the following experiments:

1. Test the performance of Jar2Vec models (and thus BloatLibD) in performing the TPL-detection task (discussed in the Appendix).
2. Compare the performance of BloatLibD with the existing TPL-detection tools (discussed in §4.2).

[5]https://bit.ly/2yb2eHY

Table 2: TPL Data summary.

Item	Count
Downloaded JAR files	450
JAR files selected for experiments	97
JAR files used for training	76 + 1 (`rt.jar`)
JAR files used for testing	20
.class files used for training	30427
.class files used for generating test pairs of bytecode and source code	4033
Unique pairs of bytecode files used for testing	20100
Unique pairs of source code files used for testing	20100

4.1 Test-bed Setup

We crawled https://mvnrepository.com/open-source?p=**PgNo**, where **PgNo** varied from 1 to 15. Each page listed ten different categories from the list of most popular ones, and under each category, the top-three libraries were listed.

We started by downloading one JAR file for each of the above libraries. That is, a total of $15 \times 10 \times 3 = 450$ JAR files were fetched. In addition to the above JAR files, we also included the JDK1.8 runtime classes (`rt.jar`). After removing the invalid files, we were left with 97 JAR files containing 38839 .class files.

We chose random 76 JAR files out of 97 plus the `rt.jar` for training the Jar2Vec models, and the remaining 20 JAR files were used for testing. We used only those .class files for training whose size was at least 1kB since such tiny .class files do not give sufficient Java and byte code, which is necessary to compute a sufficiently unique vector representation of the .class contents. The training JARs had 33,292 .class files, out of which only 30427 were of size 1kB or bigger. We chose the minimum file size as 1kB because we observed that the files smaller than 1kB did not significantly train an accurate Jar2Vec model. The testing JARs had 4,033 .class files. A summary of the TPL data is shown in Table 2. Note: the training and testing of Jar2Vec models were performed on the source code and bytecode extracted from the respective number of .class files.

4.2 Performance Comparison of BloatLibD with the Existing TPL-detection Tools

To the best of our knowledge, no work leverages the direction of using code similarity (in TPLs) and the vector representations of code to detect the bloat-TPLs for Java applications. We present our tool's performance comparison (BloatLibD) with some of the prominently used tools, viz., LiteRadar, LibD, and LibScout. The details about these tools have been discussed in §2.

4.2.1 Objective

To compare the performance of BloatLibD with the existing TPL-detection tools. Through this experiment, we address the following:
How does BloatLibD perform in comparison to the existing TPL-detection tools? What is the effect on storage and response time? Is BloatLibD resilient to the source code obfuscations?

4.2.2 Procedure

To perform this experiment, we invited professional programmers and asked them to evaluate our tool. One fundred and nine of them participated in the experiment. We had a mixture of programmers from final year computer science undergraduates, postgraduates, and the IT industry with experience between 0-6 years. The participants had considerable knowledge of Java programming language, software engineering fundamentals, and several Java applications. The experiment was performed in a controlled industrial environment. We provided access to our tool for performing this experiment by sharing it at https://bit.ly/2V80NCT. The tools' performance was evaluated based on their *accuracy*, *response time*, and the *storage requirement* in performing the TPL-detection task. We compute the tool's storage requirement of the tools by measuring the memory space occupied in storing the relevant "reference TPLs." The TPL-detection tools – LibD, LibScout, and LibRadar, require the inputs in an Android application PacKage (APK) format. Therefore, APK files corresponding to the JAR versions of the TPLs were generated using the Android Studio toolkit[6] (listed in Step 12 of Algorithm 2).

The programmers were requested to perform the following steps:

1. Randomly select a sample of 3-5 JAR files from the test-bed developed for the experiments (discussed in §4.1).
2. Test the JAR file using Algorithm 2.
3. Report the tools' accuracy and response time, as observed from the experiment(s).

[6]https://developer.android.com/studio

Algorithm 2: Steps for performing the comparison.

```
 1: Input: $L$ = Set of TPLs downloaded as JAR files ($J$)
    randomly from MavenCentral.
    $X$ = Set of XML files required as input by LibScout.
    $\hat{M_{bc}}, \hat{M_{sc}}$ := The best performing Jar2Vec models.
    $\phi_{sc}^{ref}, \phi_{bc}^{ref}$ := Reference PVA vectors for source code
    and bytecode.
    /*Please see Table 1 for notation*/
 2: Output: Terminal outputs generated by LibD, LiteRadar, and LibScout.
 3: $F'_{sc} := F'_{bc} := NULL$
 4: for all JAR files $J \in L$ do
 5:     for all .class files $f_u \in J$ do
 6:         $f_{bc}^{u}, f_{sc}^{u}$ := Obtain the textual forms of the byte-
            code and source code present in $f_u$.
 7:         $f'_{bc}, f'_{sc}$ := Modify $f_{bc}^{u}$ and $f_{sc}^{u}$ using various trans-
            formations listed in §4.1.
 8:         $F'_{sc} := F'_{sc} \cup \langle f'_{sc} \rangle$
 9:         $F'_{bc} := F'_{bc} \cup \langle f'_{bc} \rangle$
10:     end for
11: end for
12: $Y$ := Convert $F'_{sc}$ into the corresponding APK files us-
    ing Android Studio.
13: Test with LiteRadar, LibD, LibScout using $X$ and $Y$.
14: Test $F_{sc}$ and $F_{bc}$ with BloatLibD using Algorithm 1.
```

4.2.3 Evaluation Criteria

In the context of the TPL-detection task, we define the accuracy as:

$$Accuracy = \frac{Number\ of\ TPLs\ correctly\ detected}{Total\ number\ of\ TPLs\ tested} \tag{1}$$

4.2.4 Results and Observations

Table 3 lists the *accuracy*, *response time*, and *storage space requirement* values observed for the tools. We now present a brief discussion of our results.

Accuracy of the TPL-detection Tools: Some of the key observations from the experiments are:

1. LiteRadar cannot detect the transformed versions of the TPLs and fails in some cases when tested with the TPLs containing *no transformations*. For instance, it cannot detect exact matches in the case of zookeeper-3.3.0.jar library[7] and kotlin-reflect-1.3.61.jar library[8].
2. LibScout detects the *TPLs without any transformations* but suffers from package-name obfuscations as it cannot detect the modified versions of TPLs containing package-name transformations.
3. LibD substantially outperforms LibRadar and LibScout in capturing the similarity between the TPLs but does not comment on their nature, i.e., ⟨likely-to-be-bloat-TPL, likely-to-be-non-bloat-TPL⟩. It also comes with an additional cost of manually comparing the TPLs with the "reference set."
4. For a given input file, BloatLibD labels it as ⟨ likely-to-be-bloat-TPL, likely-to-be-non-bloat-TPL⟩, and lists the top-k similar libraries and the respective similarity scores shown in Figure 3.
5. BloatLibD detects the TPLs for 99% of the test cases. As observed from the table values, BloatLibD outperforms LiteRadar, LibScout, and LibD with the improvement scores of 30.33%, 74.5%, and 14.1%, respectively. As BloatLibD performed equally well on the obfuscated test-inputs, the results validate that it is resilient to the considered obfuscation types.

[7]http://bit.ly/2VymUmA

[8]http://bit.ly/32MvkZe

Table 3: Performance comparison of various TPL-detection tools.

TPL detection tools	Performance Metrics values		
	Accuracy (in %)	Response Time (in seconds)	Storage requirement (in MBs)
LiteRadar	68.97	12.29	1.64
LibScout	25.23	6.46	3.93
LibD	85.06	100.92	12.59
BloatLibD	99	38.98	1.52

Response Time of the TPL-detection Tools: BloatLibD achieves 61.37% improvement in the response time over LibD while delivering higher response times than LiteRadar and LibScout.

Storage Requirement of the TPL-detection Tools: BloatLibD leverages the PVA vectors to detect the similarity among the TPLs, while the tools used for comparison, viz., LibD, LibScout, and LiteRadar, use the "reference lists" of TPLs. These tools contain the "reference lists" of TPLs as files within their tool packages. As observed from the storage requirement values, BloatLibD has the lowest storage requirement due to the vector representation format. BloatLibD reduces the storage requirement by 87.93% compared to LibD, 61.28% compared to LibScout, and 7.3% compared to LiteRadar.

4.3 Threats to Validity

For developing our Jar2Vec models and D, we utilize a subset of Java TPLs (i.e., JAR files) present in the MavenCentral repository. We assume that these TPLs cover a reasonably wide variety of Java code such that the Jar2Vec models that we train on them will be accurate. However, there could still be many other

Java code types that could have improved the Jar2Vec models' accuracy. The approach's efficacy also depends on the completeness and update status of the reference libraries. For training the Jar2Vec models, we obtain the normalized textual forms of the source code and bytecode representations of the .class files present in the JAR files. We obtain the source code and bytecode by using the decompilation and disassembly operations. Therefore, our Jar2Vec models' accuracy is subject to the accuracy of the decompilation and disassembly operations.

Next, we treat an unseen TPL that shows considerable similarity with the set of "available TPLs" as *likely-to-be-non-bloat-TPLs*. Thus, the labeling of a TPL as *likely-to-be-non-bloat-TPL* or *bloat-TPL* is strongly dependent on its use in the considered application. We do not consider the TPL-usage as per now, but have included it as part of our future work. While training the Jar2Vec models, we consider only the .class files of size 1kB or larger. However, there may exist Java libraries where the average class size is lower than this limit. Excluding such a group of TPLs from the training might give inaccurate results when the input TPL being checked happens in such a group. The main reason for excluding such tiny .class files is that they do not give sufficient Java and byte code, which is necessary to compute a sufficiently unique vector representation of the .class contents.

By reviewing the literature (Ma et al., 2016; Backes et al., 2016; Li et al., 2017), we realized that there are a significant amount of TPL-detection tools designed for Android Applications, requiring the input file in an APK format. To the best of our knowledge, no tool performs the TPL-detection for software applications existing in JAR formats. Therefore, we converted our TPLs present from JAR to APK format using the Android Studio toolkit and choose LibD, LibRadar, and LibScout – some of the popular TPL-detection tools for our comparison. However, due to the fast advances of research in this area, we might have missed some interesting TPL-detection tool that works with the JAR file formats.

5 CONCLUSIONS

We have proposed a novel application of the well-known PVA to train the Jar2Vec models to detect the similarity of JAR files. The uses of the PVA have been mostly in the domain of natural language processing. To the best of our knowledge, we are the first to apply it for detecting the similarity of Java libraries and leverage it to build our tool – BloatLibD. Our work's significant insight is that the PVA can compute dependable vector representations of various software code forms.

Our experiments with a large corpus of .class files have demonstrated that Java binaries' similarity can be reliably detected using Jar2Vec. Another key idea that we have successfully leveraged in our approach is using the modern, semantics-preserving Java decompilers to transform the binary .class files into an obfuscation-invariant textual form. This transformation allowed us to leverage PVA for detecting binary code similarity while also allowing our method to be resilient against the obfuscated input. We have verified our approach's efficacy by testing it with more than 30000 .class files, where we have achieved detection accuracy above 99% and an F1 score of 0.968. BloatLibD outperforms the existing TPL-detection tools, such as LibScout, LiteRadar, and LibD, with an accuracy improvement of 74.5%, 30.33%, and 14.1%, respectively. Compared with LibD, BloatLibD achieves a storage reduction of 87.93% and a response time improvement of 61.37%.

As part of the future work, we plan to extend our idea of utilizing the source code similarity to detect software bloat for software written in other programming languages. Further, we plan to explore the direction of actual TPL-usage within the application to detect the unused parts of TPL-code. The idea can be leveraged to develop various software artifacts for automating the SDLC activities, such as software code review, source code recommendation, and code clone detection.

REFERENCES

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019). Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29.

Anthony Desnos, Geoffroy Gueguen, S. B. (2018). Welcome to androguard's documentation!

Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. (2012). Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM.

Backes, M., Bugiel, S., and Derr, E. (2016). Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM.

Book, T., Pridgen, A., and Wallach, D. S. (2013). Longitudinal analysis of android ad library permissions. *arXiv preprint arXiv:1303.0857*.

Chen, K., Liu, P., and Zhang, Y. (2014). Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186. ACM.

Dai, A. M., Olah, C., and Le, Q. V. (2015). Document embedding with paragraph vectors. In *NIPS Deep Learning Workshop*.

Feichtner, J. and Rabensteiner, C. (2019). Obfuscation-resilient code recognition in android apps. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, page 8. ACM.

Grace, M. C., Zhou, W., Jiang, X., and Sadeghi, A.-R. (2012). Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM.

Le, Q. and Mikolov, T. (2014). Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196.

Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., Xue, R., and Huo, W. (2017). Libd: scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346. IEEE.

Liu, B., Liu, B., Jin, H., and Govindan, R. (2015). Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th annual international conference on mobile systems, applications, and services*, pages 89–103. ACM.

Ma, Z., Wang, H., Guo, Y., and Chen, X. (2016). Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th international conference on software engineering companion*, pages 653–656. ACM.

Mitchell, N. and Sevitsky, G. (2007). The causes of bloat, the limits of health. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 245–260.

Strobel, M. (2019). Procyon: A suite of java metaprogramming tools focused on code generation, analysis, and decompilation.

Wang, H., Guo, Y., Ma, Z., and Chen, X. (2015). Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 71–82. ACM.

Xu, G., Mitchell, N., Arnold, M., Rountev, A., and Sevitsky, G. (2010). Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 421–426.

Zhang, Y., Dai, J., Zhang, X., Huang, S., Yang, Z., Yang, M., and Chen, H. (2018). Detecting third-party libraries in android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152. IEEE.

APPENDIX

Objective: The objective here is to seek an answer to our questions:

1. *For reliably detecting a library, what is the acceptable value of Jar2Vec similarity scores for source code and bytecode inputs?*
2. *Does the threshold similarity score ($\hat{\alpha}$) vary with the input parameters (β, γ, and ψ) of PVA?*
3. *What are the optimal values for the PVA tuning-parameters β, γ, and ψ?*

Please refer to Table-1 for notation definitions.

Table 4: Scenarios for training Jar2Vec models using PVA.

Parameters varied			
Epochs β	**Vector size γ**	**Training samples ψ**	**Models**
Fixed at 10	Fixed at 10	Vary 5000 to- `CorpusSize` in-steps of 5000	`CorpusSize` ÷ 5000
Vary 5- to 50 in-steps of 5	Fixed at 10	Fixed at- `CorpusSize`	10
Fixed at 10	Vary 5- to 50 in-steps of 5	Fixed at- `CorpusSize`	10

Test-bed Setup: Using the test partition of the test-bed developed in §4.1, we generate a test dataset (Y) containing $same, different$ file pairs in 50:50 ratio. Further, to check if our tool is resilient to source code transformations, we test it for the following three scenarios:

1. *Package-name Transformations:* Package names of the classes present in TPLs are modified.
2. *Function (or Method) Transformations:* Function names are changed in the constituent classes' source code, and function bodies are relocated within a class.
3. *Source Code Transformations:* Names of various variables are changed, source code statements are inserted, deleted, or modified, such that it does not alter the semantics of the source code. For instance, adding print statements at various places in the source file.

We test Jar2Vec models' efficacy in detecting similar source code pairs (or bytecode pairs) using Y.

Procedure: The salient steps are:

1. F_{bc}, F_{sc} := Obtain the textual forms of bytecode and source code present in source files of training JARs of the test-bed (developed in §4.1).
2. For each parameter combination $\pi \in Z$ (listed in Table-4):

   (a) $S_{sc}^{\pi} := S_{bc}^{\pi} := NULL$

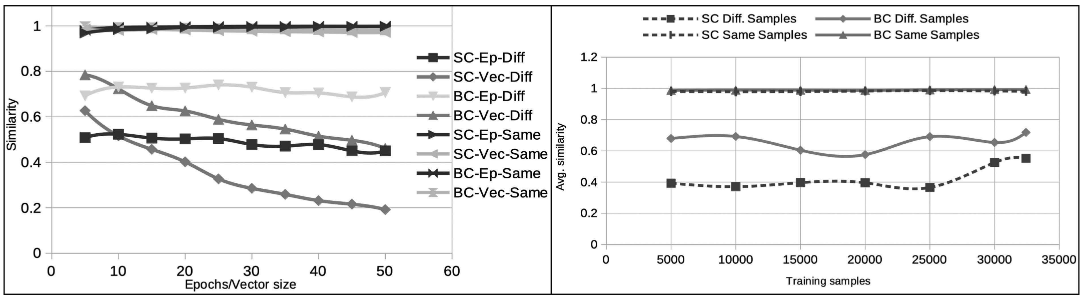


Figure 4: Variation of *average similarity* with PVA tuning-parameters.

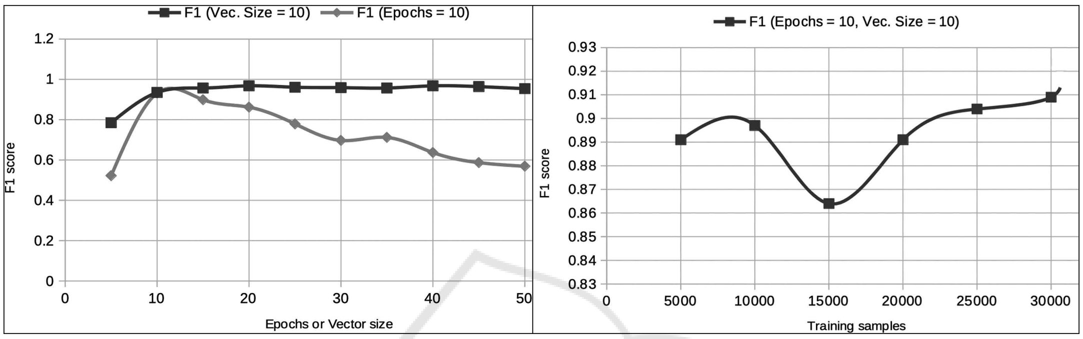


Figure 5: Performance metrics with different PVA models trained on *source code.*

(b) $M^{\pi}_{bc}, M^{\pi}_{sc}$:= Train the Jar2Vec models using F_{bc}, F_{sc}.

(c) Save M^{π}_{bc} and M^{π}_{sc} to disk.

(d) For each file pairs $\langle p_i, p_j \rangle \in Y$:

i. ϕ_i, ϕ_j := Obtain PVA vectors for p_i, p_j using $M(\pi)$

ii. $\alpha_{i,j}$:= Compute cosine similarity between $\langle \phi_i, \phi_j \rangle$

iii. if $p_i == p_j$: $S_{same} = S_{same} \cup \langle \alpha_{i,j} \rangle$

iv. else: $S_{different} = S_{different} \cup \langle \alpha_{i,j} \rangle$

(e) $\hat{\alpha^{\pi}_{bc}}, \hat{\alpha^{\pi}_{sc}}$:= Obtain the average similarity scores using S^{π}_{bc} and S^{π}_{sc} and save them.

(f) Using the $\hat{\alpha^{\pi}_{bc}}, \hat{\alpha^{\pi}_{sc}}$ as thresholds, compute the accuracy of M^{π}_{bc} and M^{π}_{sc}.

(g) Plot the variation of $\hat{\alpha_{bc}}$, $\hat{\alpha_{sc}}$, the accuracy of PVA models for different values of β, γ, and ψ used in the experiment, and analyze.

Results and Observations: Figure 4 and 5 show the effect of PVA tuning-parameters on the *average similarity* and the model performance metrics values, respectively. The legend entry BC-Ep-Diff represents the similarity variation w.r.t epochs for bytecode case when two samples were different. SC-Vec-Same indicates the variation w.r.t vector size for source code case when two samples were identical. The following are the salient observations:

1. Effect of increasing the epochs beyond 10 seems to have a diminishing improvement in the accuracy scores.
2. A noticeable decrease in similarity scores was observed by increasing the vector count beyond 5, and the epochs count beyond 10.
3. As anticipated, the accuracy (indicated by F1 scores[9]) improves with the size of training samples.

Therefore, we take $\hat{\alpha_{sc}} = 0.98359$ and $\hat{\alpha_{bc}} = 0.99110$ as the similarity threshold values for source code data and bytecode data, respectively. Further, the best accuracy (99.48% for source code and 99.41% for bytecode) is achieved with the Jar2Vec model trained using 30427 samples, 10 epochs, and the vector size of 10. The precision and recall values, in this case, were 99.00% and 99.00%, respectively, resulting in an F1 score of 99% for the source code case. As we achieve the highest accuracy scores at $\beta = \gamma = 10$, we take these as the optimal input parameter values for PVA.

[9]https://bit.ly/3kHqkNg