LEVERAGING FILE SIGNIFICANCE IN BUS FACTOR ESTIMATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Vahid Haratian
January 2025

Leveraging File Significance in Bus Factor Estimation
By Vahid Haratian
January 2025

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Eray Tüzün(Advisor)

Uğur Doğrusöz

İsmail Sengör Altıngövde

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

LEVERAGING FILE SIGNIFICANCE IN BUS FACTOR ESTIMATION

Vahid Haratian
M.S. in Computer Engineering
Advisor: Eray Tüzün
January 2025

Software projects often face developer turnover for various reasons. Since developers are key sources of knowledge in these projects, their absence inevitably leads to some degree of knowledge loss. The Bus Factor (BF) is a metric used to assess the impact of this knowledge loss on a project's continuity. Traditionally, BF is defined as the smallest group of developers whose departure would result in a loss of more than half of the project's knowledge. Current state-of-the-art methods calculate developers' knowledge based on the number of files they have authored, using data from version control systems (VCS). However, numerous studies have highlighted that not all files in software projects hold the same level of significance. In this study, we investigate the impact of weighting files based on their significance on the performance of two widely used BF estimators. Significance scores are calculated using five established graph metrics derived from the project's Dependency Graph: PageRank, In-/Out-/All-Degree, and Betweenness Centralities. Additionally, we introduce BFSig, a prototype implementing our approach. Lastly, we present a new dataset featuring BF scores reported by software practitioners from five prominent GitHub repositories. Our findings show that BFSig surpasses the baseline methods, achieving up to an 18% reduction in Normalized Mean Absolute Error (NMAE). Additionally, BFSig reduces False Negatives by 18% when identifying potential risks linked to low BF. Furthermore, our respondents validated BFSig's versatility, highlighting its capability to evaluate the BF of individual project subfolders. In conclusion, we believe that when estimating BF from authorship, software components of greater significance should be given higher weight.

Keywords: Bus Factor, Truck Factor, File Significance, Knowledge Management, Intelligent Collaboration Tools, Dependency Graph, Code Referencing.

ÖZET

DOSYA ÖNEMİNİN OTOBÜS FAKTÖRÜ TAHMİNİNDEKİ ROLÜ

Vahid Haratian
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Eray Tüzün
Ocak 2025

Yazılım projeleri, çeşitli nedenlerden dolayı geliştiricilerin ayrılmasıyla karşı karşıya kalır. Yazılım projelerinde geliştiriciler ana bilgi kaynaklarından biri olduğundan, onların yokluğu kaçınılmaz olarak belli bir düzeyde bilgi kaybına yol açmaktadır. Otobüs Faktörü (OF), bu bilgi kaybının projenin sürekliliğini nasıl etkileyebileceğini değerlendiren bir metriktir. Geleneksel olarak, OF, projeden ayrıldıklarında toplam bilginin yarısından fazlasını kaybettiren en küçük geliştirici kümesi olarak hesaplanır. Mevcut en son yaklaşımlar, geliştiricilerin bilgisini sürüm kontrol sistemi bilgilerini kullanarak yazılmış dosya sayısıyla ölçmektedir. Ancak, birçok çalışma yazılım projelerindeki dosyaların farklı öneme sahip olduğunu göstermiştir. Bu çalışmada, dosyaların önemine göre ağırlıklandırılmasının, iki yaygın BF tahmin algoritmasının performansına olan etkisi incelenmiştir. Önem skorları, projenin Bağımlılık Grafiğinden türetilen PageRank, Giriş-/Çıkış-/Tüm-Derece ve Ara Merkezilik gibi beş iyi bilinen grafik metriği hesaplanarak elde edilmiştir. Ayrıca, yaklaşımın bir prototipi olan BFSig tanıtılmıştır. Son olarak, beş önde gelen GitHub deposundan yazılım uzmanlarıyla yapılan anketlerle toplanan BF skorlarını içeren yeni bir veri kümesi sunulmuştur. Sonuçlarımız, BF-Sig'un, Normalleştirilmiş Ortalama Mutlak Hata (NMAE) açısından %18'e kadar bir azalma sağlayarak, temel yaklaşımlardan daha iyi performans gösterdiğini göstermektedir. Ayrıca, BFSig, düşük BF ile ilişkili potansiyel riskleri belirlemede %18 daha az Yanlış Negatif üretmektedir. Bunun yanı sıra, BFSig'un esnekliğini projenin alt klasörlerinin BF'sini değerlendirme yeteneği doğrulanmıştır. Sonuç olarak, OF'nü yazarlık temelinde tahmin etmek için, daha yüksek öneme sahip yazılım bileşenlerine daha yüksek ağırlık verilmesi gerektiğine inanıyoruz.

Anahtar sözcükler: Otobüs Faktörü, Kamyon Faktörü, Dosya Önem Derecesi, Bilgi Yönetimi, Akıllı İşbirliği Araçları, Bağımlılık Grafiği, Kod Referansı.

Acknowledgement

First and foremost, I would like to acknowledge that this work would be impossible if it were not for the unwavering belief and backing I had of my supervisor Dr. Eray Tüzün. I hope I can repay the faith and patience he has displayed in me through the highs and lows during my time as a student and as a researcher, the freedom and confidence he afforded me at every step, and pass on the values he instilled in me and which I hold so dear.

This thesis would also not be possible without the help of our friends at the Intelligent Collaboration Tools Lab at JetBrains N.V.: Vladimir, Misha, Nikolai, Egor, and everyone else, but most of all, Pouria who remained undeterred with his efforts despite serious difficulties at just about every turn.

I'm also eternally grateful to Bilkent University, the administrators, and the admissions committee(s) who took a gamble on me not once, but twice, paving the way for me to receive a quality education that would otherwise have been simply out of reach.

This would also not have been possible without the support of my family, especially my mother, who fought against all odds to afford me the best possible education we could afford, and my father who spent his life's earnings on the same, far beyond his means.

This study was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) 3501 program (Project Number: 121E584).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development is a knowledge-driven process that demands a broad range of skills and expertise, especially given the rapid pace of technological progress [1]. Typically, software projects are created and maintained by teams of developers [2]. In large-scale software projects, it is common for developers to have varying levels of familiarity with different parts and components of the system [3]. As a result, each component often has key developers who possess deeper insights into its underlying logic.

It is important to identify key developers in a project because their departure can threaten the continuity of the project and leave certain parts unmaintainable by the remaining team. This is not a hypothetical risk: Avelino et al. reported [4] that 16% of the open-source projects they studied experienced the departure of all key developers. Such situations disrupt the maintenance process and may lead to project abandonment unless new key developers are brought in [4]. This problem is commonly referred to as the "bus factor problem" [5].

The Bus Factor (BF), also known as the Truck Factor, indicates how knowledge is distributed among developers in a project. It reflects the risk of knowledge loss and the project's vulnerability in case certain developers leave. By definition, BF is the smallest number of people whose departure would prevent the rest of the

team from maintaining the project [6].

BF is often calculated alongside a ranked list of key developers [7, 8, 9, 10, 5], who are referred to as BF developers. Understanding the BF helps reduce turnover risks and allows project managers to identify potential bottlenecks in the development process, preventing future issues. For example, the development process can slow down if only a few engineers are responsible for critical components. These developers might be unavailable due to other work, vacations, or sick leave [11, 5].

Several studies have explored the bus factor. A common approach, used by Zazworka et al. [8], Cosentino et al. [9], Rigby et al. [12], and Avelino et al. [7], is to analyze version control history to estimate code authorship. Jabrayilzade et al. [5] extended this method by incorporating additional indicators, such as code reviews. They also conducted surveys with developers, showing that while the BF is recognized as an important risk, it is rarely monitored in practice.

All BF studies we reviewed consider all project files or lines of code equally important when estimating the bus factor. However, Srinivasan et al. [13] found that some components within a project are more structurally significant and play a critical role in the project's overall structure. Additionally, architectural decisions can result in certain components carrying most of the business logic of the project [14]. Mens et al. [15] also observed that some files are harder to maintain due to a high number of associated bugs or insufficient documentation. These files should be given more weight in BF risk estimation since they have a greater impact on the project's maintainability. The significance of a file (FS) can be determined based on the importance of its components in relation to the rest of the system.

Several studies have attempted to identify structurally significant components by analyzing a project's source code [16, 17, 13, 18, 19, 20]. These studies typically construct a project dependency graph and apply vertex-rating methods to identify the most influential nodes. In our study, we use PageRank [21], In-/Out-/All-Degree, and Betweenness Centrality [22] as the primary vertex-rating algorithms.

This enables us to evaluate the potential impact of incorporating the structural significance of components into BF estimation.

Our goal is to improve bus factor estimation by assigning greater importance to structurally significant files, which are more likely to cause issues if abandoned. BFSig[1] computes several graph-based metrics from the dependency graph of a software project. To assess the effect of weighting files by their significance, we compare the predictions of our approach with human assessments of the bus factor. Unfortunately, although Jabrayilzade et al. [5] found that engineers are interested in BF data for both projects and their components, publicly available datasets only provide BF estimates for entire projects. To our knowledge, the study by Cury et al. [23] is the only one that evaluates knowledge ownership at a finer level of granularity. However, this study does not provide a dataset for reproducing results or validating other BF estimation methods. Since no existing dataset contains BF information for individual project components, we conducted a survey with 12 practitioners from five popular GitHub repositories to collect fine-grained ground truth data on the BF of these projects.

To calculate these scores, we first need to construct the dependency graph of the given project. However, we found that there is no widely reusable tool available for extracting dependency graphs. Although many open-source and commercial tools exist for this purpose, they are often difficult to use. Some of these tools are outdated and no longer maintained, making them incompatible with newer language features and prone to syntax errors [24, 25, 26]. For example, we observed that most Python tools, such as PyCG [25] and Pyan [24], are not compatible with the latest Python version (3.12 at the time). As a result, these tools were unable to analyze actively maintained popular GitHub repositories. While we tried using older versions of the repositories, these tools were highly sensitive to the Python version, and we could only identify one usable revision in an experimental repository.

Additionally, some tools work well only for small projects and are challenging to configure for larger ones [27, 28, 29, 30]. The definition of a small project

[1] `https://github.com/JetBrains-Research/file-importance`

can vary, but we encountered difficulties using these tools on well-known GitHub repositories. For instance, many Java tools rely on analyzing built JAR or WAR files. However, large projects like ElasticSearch do not compile into a simple WAR file that these tools can process. Moreover, this approach excludes code that is not part of the final built artifacts, such as test cases, build scripts, and documentation.

Tools that evaluate source code directly also presented significant configuration challenges. For example, Sonargraph [28] requires manual configuration of the build tools to perform its analysis. While this is manageable for small projects, configuring such tools for large projects with multiple levels of submodules is far more complex. For instance, ElasticSearch's intricate build structure made it difficult to configure Sonargraph probes, and despite our efforts, we were unable to complete the setup.

Among the tools that directly analyze source code, there are some that require minimal configuration. However, their accuracy is often too low to be useful for most cases [31]. These tools typically rely on explicit imports at the beginning of files to identify references. This approach can fail in complex project structures where the target of an import is difficult to locate. Additionally, it excludes a significant portion of references that occur within packages and files.

To address this gap and simplify dependency extraction, we introduce RefExpo [2] in this study. RefExpo leverages IntelliJ IDEA, a robust Integrated Development Environment (IDE). IDEs like IntelliJ are designed with sophisticated dependency analysis to enable precise code navigation and reference detection. They remain compatible with both the latest and older language versions by working directly with the compiler provided to them. Their user-friendly interfaces also make project import and management significantly easier compared to experimental tools. For most projects, the importing phase is automatic and straightforward. Even for large open-source projects with complex structures, documentation is often provided to guide IDE configuration for full code recognition. For example, projects like ElasticSearch and Apache Flink include instructions for importing

[2] `https://github.com/vharatian/RefExpo`

their codebases into IntelliJ.

The main contribution of this study is evaluating the impact of files' structural significance on BF estimation. We found that incorporating significance scores improves performance by up to 18% in terms of Normalized Mean Average Error (NMAE) and reduces false negatives by up to 18% when identifying bottlenecks related to low BF. RefExpo performs best when Betweenness Centrality is integrated into the tool developed by Jabrayilzade et al. [5]. Additionally, we created a new BF dataset that includes data on project subfolders. Using this dataset, we demonstrate that RefExpo effectively estimates the BF of project subfolders. Consistent with the findings of Jabrayilzade et al. [5], we observed that engineers are interested in knowing the BF of both the entire project and its components. Our findings and dataset provide researchers with resources to develop BF estimation tools that better align with the needs of practitioners.

The contributions of this study can be summarized as follows:

- Introducing a novel BF estimator that incorporates FS.
- Demonstrating the effectiveness of BF estimation for project subfolders.
- Collecting BF data for five projects and their subfolders into a dataset.
- Introducing RefExpo, an easy-to-use tool for generating dependency graphs.
- Compiling an initial dataset [2] of 20 popular GitHub projects, including Java and Python, with plans for future expansion.

This thesis builds upon our previous publications, BFSig[32] and RefExpo[33], which have been integrated and expanded to form the basis of this work.

Chapter 2

Background

In this chapter, we provide an overview of the related works on the Bus Factor (BF) calculation and the software component significance.

2.1 Bus Factor Calculation

Various studies have sought to assess developer knowledge by analyzing their contributions to the development process. These studies introduce metrics based on version control system (VCS) data as proxies for actual knowledge. Girba et al. [34] define file ownership as the proportion of lines modified by a developer relative to the total lines in a file. Fritz et al. [35, 36] present the degree of authorship (DOA), which evaluates a developer's interaction with a file in comparison to interactions by other developers.

Building on these concepts, some studies estimate the Bus Factor (BF) by analyzing developer knowledge derived from VCS data. The methodologies in these studies typically follow a common pattern. They begin by calculating a quantitative measure of each developer's knowledge of various software components (usually files). Then, they determine the smallest group of developers whose departure would reduce component ownership below a predefined threshold

(commonly 50%). In this framework, a file is considered abandoned if no developer retains sufficient authorship over it. When the number of abandoned files exceeds a specified threshold, the project is classified as unmaintainable or "dead."

The foundational work in this domain by Zazworka et al. [8] employs a greedy iterative algorithm to simulate the removal of the most knowledgeable developers and assess its effect on ownership coverage. Cosentino et al. [9] investigate various interaction metrics, such as commit frequency and code churn, to identify optimal strategies for evaluating developer expertise. Rigby et al. [12] propose utilizing a random approach instead of a greedy one to pinpoint the smallest group of BF developers.

Avelino et al. [7] introduced a widely adopted method for estimating the Bus Factor (BF), leveraging the degree of authorship (DOA) metric initially proposed by Fritz et al. [36]. Their approach begins by calculating the DOA for each developer across project files. Next, it filters out minor contributions, retaining only significant ones classified as DOA contributions. The method then simulates the iterative removal of the most significant contributors, evaluating the project's authorship coverage at each step. When this coverage falls below 50%, the process halts, and the removed developers are identified as BF developers, with their count representing the BF value. Authorship coverage is defined as the proportion of files with at least one major contributor to the total number of files.

Avelino et al. applied this methodology to 133 open-source projects and gathered feedback from practitioners for 67 of them. The results revealed that 53% of practitioners fully or partially agreed with the findings. Additionally, they reported 91% accuracy for projects with a BF of one or two.

Recently, Jabrayilzade et al. [5] refined the algorithm proposed by Avelino et al. [7] by integrating two additional dimensions of authorship information: data from code reviews and time spent in meetings. Their enhanced algorithm also incorporates the concept of knowledge decay, which reduces the influence of older contributions over time. For instance, the algorithm assigns twice the weight to the authorship of a recent commit compared to one made five months earlier.

Algorithm 1 Modified Algorithm with Structural Changes

```
procedure DetermineImpactLevel(Contributors, Artifacts)
    ContributorsData ← ComputeContributorMetrics(Contributors, Artifacts)
    SortedContributors ← RankByInfluence(ContributorsData)
    ImpactLevel ← 0
    for Contributor ∈ SortedContributors do
        Influence ← MeasureOverallImpact(SortedContributors, Artifacts)
        if Influence < 0.5 then
            break
        RemoveContributor(SortedContributors, Contributor)
        ImpactLevel ← ImpactLevel + 1
    return ImpactLevel

procedure ComputeContributorMetrics(Contributors, Artifacts)
    Metrics ← {}
    for Contributor ∈ Contributors do
        InfluenceScore ← CalculateRatio(Contributor.Artifacts, Artifacts)
        Metrics[Contributor] ← InfluenceScore
    return Metrics

procedure MeasureOverallImpact(Contributors, Artifacts)
    RelevantCount ← 0
    for Artifact ∈ Artifacts do
        if IsContributorRelevant(Artifact, Contributors) then
            RelevantCount ← RelevantCount + 1
    return RelevantCount / Length(Artifacts)
```

This approach ensures that developers who contributed to a file in the distant past may no longer be regarded as its primary authors as time progresses.

To validate their algorithm, Jabrayilzade et al. conducted experiments on 13 commercial projects from JetBrains, gathering practitioner feedback through surveys. The evaluation revealed a marginally improved mean absolute error of 5.46, compared to 5.80 achieved by the tool from Avelino et al.

For the remainder of this paper, we will refer to the tools developed by Avelino et al. and Jabrayilzade et al. as ABF (Avelino) and JBF (Jabrayilzade), respectively.

2.2 Software Component Significance

Numerous studies investigate methods to pinpoint structurally critical components in software projects. A prevalent technique involves analyzing dependency graphs. A dependency graph represents software components mathematically, where nodes denote components and edges indicate dependencies between them. Inoue et al. [16] introduce an algorithm inspired by Google's PageRank [21] to identify key nodes within a dependency graph. Building upon this idea, Srinivasan et al. [13] propose the `Discrete-Time Markov Chain-based` Component Ranking algorithm. They benchmark their approach against three established graph metrics: Betweenness, Closeness, and Eigenvector centralities [18]. Their findings reveal that their algorithm aligns most closely with expert evaluations of critical components. Qing et al. [17] examine the connection between a node's degree in a dependency graph and its significance within a software project. Their approach identifies complex classes by considering both the number of methods and lines of code.

Several advanced methodologies are available for identifying significant components. For instance, Pan et al. [19] leverage k-core decomposition to compute centrality and pinpoint key nodes within a dependency graph. Similarly, Wang et al. [20] employ dynamic metrics, such as runtime data and method invocations,

to identify critical components.

Many studies suggest that nodes with greater significance in a dependency graph are inherently more structurally important. This often leads to a focus on components with higher reusability potential. It also facilitates analysis of the ripple effect, which evaluates how changes to one component influence others. Nevertheless, no existing research, to our knowledge, examines the relationship between component significance and the risks posed by uneven knowledge distribution. In this work, we evaluate knowledge distribution by treating files as the fundamental unit, aligning with baseline approaches. We define FS as the cumulative significance of the components within a file. Given the limited availability of BF datasets, our analysis concentrates on a select set of graph metrics, as described below.

Graph Metrics for the Dependency Graph. Our study aims to enhance bus factor evaluation by deriving FS from the dependency graph. We use the following metrics on the dependency graph as estimators for FS.

1) **PageRank (PG):** Initially introduced by Google [21] to assess the significance of web pages, PageRank has been repurposed in software engineering research to examine various types of graphs. For example, Suzuki et al. [37] employed PageRank to investigate commit histories, identifying commit types more prone to introducing bugs. This algorithm iteratively assigns scores to nodes in a directed graph based on the quantity and quality of incoming links. In web contexts, a page linked by highly ranked nodes achieves a higher PageRank score.

2) **Degree Centrality (DC):** Degree centrality measures how connected a node is within a graph [22]. It is determined as the ratio of nodes directly connected to a given node, accounting for incoming and outgoing links in directed graphs. For the dependency graphs used in this analysis, degree centrality scores lie between 0 and 1. This metric has two variants:

   – **In-Degree Centrality (IDC):** Reflects the proportion of nodes with edges directed toward the node.

- **Out-Degree Centrality (ODC):** Represents the proportion of nodes to which the node directs edges.

3) **Betweenness Centrality (BC):** Betweenness centrality quantifies a node's importance by calculating the fraction of shortest paths that pass through it [22].

$$BC(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2.1}$$

Here, σ_{st} represents the total number of shortest paths between nodes s and t, while $\sigma_{st}(v)$ denotes the paths passing through node v. Unlike other metrics, Betweenness Centrality is influenced by the graph's size and can exceed 1.

2.3 Dependency Graph Extraction

2.3.1 Java Tools

Most existing tools [27, 29] designed for Java evaluation focus on built artifacts, such as JAR or WAR files. However, we were unable to find any updated, ready-to-use tools for analyzing Java source code directly. Some tools, like Sonargraph [28], support source code evaluation but require additional configuration to build the artifacts before performing their analysis. Other tools, such as OPAL [38], demand substantial custom implementation, making them less user-friendly and not immediately usable out of the box.

Dependency Finder. DependencyFinder [29] is a tool designed for analyzing Java applications, particularly focusing on single artifacts like JAR or WAR files. It offers detailed analysis by identifying dependencies within the code, including method and property accesses, providing a granular understanding of code interactions. Despite its strong features for extracting and browsing dependencies, DependencyFinder struggles to handle large projects that produce complex artifacts not contained within a single file.

Jarviz. Jarviz [27] is a tool tailored for analyzing Java applications, specifically JAR files, with support for evaluating multiple JAR files simultaneously. It requires explicit specification of dependencies for accurate analysis. One of its key features is the ability to detect method-level dependencies, providing insights into method interactions and dependencies across JAR files. However, Jarviz does not detect property accesses unless they occur via method calls (e.g., getters and setters), as its focus is primarily on method-level analysis.

OPAL. OPAL [38] is an advanced static analysis platform designed to help developers improve the precision, soundness, and performance of their static analyses. It evaluates and decompiles Java-compiled outputs, such as class files and JAR files, to provide detailed analyses. A notable strength of OPAL is its ability to execute analysis modules efficiently, ensuring scalability—an area where many scientific tools face challenges. While OPAL does not directly support dependency graph (DG) construction, it offers fundamental features for tracing dependencies, making it a versatile tool for static analysis. It allows flexibility in evaluation and can track references at various levels, such as class and method, based on custom implementations.

RefExpo. The primary advantage of RefExpo for evaluating Java projects is its simplicity and ease of use. Unlike many tools, RefExpo does not require a JAR file, removing restrictions on project size. Additionally, no prior knowledge of the underlying build system is needed, as IntelliJ can automatically read and process all project structures. RefExpo evaluates all files in a repository, including test files and other supplementary files, offering comprehensive analysis.

2.3.2 Python Tools

Generating dependency graphs for Python projects differs significantly from generating them for strictly typed languages due to Python's flexibility and its reliance on runtime reference resolution. Unlike Java, Python does not produce compiled bytecode like JAR files that encapsulate most references in the code. Instead, all references are resolved at runtime. Although dynamic call analysis

tools exist [39], static dependency analysis for Python faces unique challenges, particularly in reference resolution [40].

Pyan. Pyan [24] is a tool designed for analyzing Python code, focusing on tracking classes and functions directly from source code. This capability makes it a valuable solution for Python code analysis. However, Pyan has limitations in handling more complex structures, such as nested functions or function pointers. Additionally, its primary drawback is its compatibility, as it supports only Python versions up to 3.6.

PyCG. PyCG [25] is another tool for Python code analysis, offering a practical approach to constructing call graphs and evaluating security vulnerabilities. It analyzes code statistically, focusing on time efficiency. PyCG introduces a novel method for evaluating variable assignment chains and function passing, enabling the generation of more accurate call graphs. A key achievement of PyCG is its exceptional accuracy in call graph generation, boasting a 99.2% success rate within its specific microsite. Additionally, PyCG provides a micro suite to assess tool performance across various aspects of the Python language. However, PyCG is limited to Python versions 3.6 and 3.7 and has been officially archived.

RefExpo. One of the primary advantages of RefExpo over other Python-specific tools is its modular integration with IntelliJ, allowing it to work with any Python interpreter. This compatibility ensures that RefExpo can support all Python versions, both old and new. In contrast, many existing tools have built-in interpreting logic that restricts them to specific Python versions. Another advantage of RefExpo is its robust foundation in IntelliJ, which is backed by JetBrains, a well-established company, and a large community. This backing ensures ongoing maintenance and compatibility with future advancements.

However, a notable limitation of RefExpo compared to the previously mentioned tools is its inability to extract dynamic references. Dynamic references are not explicitly defined in the code but are resolved at runtime based on inputs, which RefExpo currently cannot capture.

2.3.3 Multi-Lingual tools

Sonargraph. Sonargraph [28] is a software architecture management tool designed to analyze code quality in projects written in `Java`, `C#`, and `Python`. One of its key features is Dependency Extraction, which scans the code structure to identify and visualize dependencies between packages, classes, and methods. This feature provides second-level evaluations of code dependencies, helping developers detect and resolve complex issues such as cyclic dependencies. While Sonargraph can evaluate Python code directly from source, its evaluation of Java projects requires configuration within build tools to analyze the build artifacts. As a result, it functions as a semi-source-based tool for Java projects.

RefExpo. Multilingual tools typically require extensive configuration before they can begin evaluations. In contrast, RefExpo eliminates the need for manual configuration by automatically detecting all the programming languages used in a project. Additionally, it supports advanced and framework-specific evaluations, such as analyzing Angular directives in HTML files, provided the appropriate plugins are configured. RefExpo also handles projects that use multiple programming languages, addressing a limitation of many other tools. The modular architecture of IntelliJ allows it to support modules with different programming languages or SDKs and even accommodate multiple languages within a single module.

Chapter 3

RefExpo

RefExpo creates detailed dependency graphs for projects written in various programming languages. Built on IntelliJ, RefExpo supports multi-language projects, including Java, Python, JavaScript, Go, Ruby, HTML, and framework-specific syntax. For example, Python support can be enabled by installing the Python plugin provided by JetBrains. Additionally, adding framework-specific plugins, such as for Angular, allows RefExpo to generate advanced dependency graphs tailored to custom framework implementations. The use of IntelliJ as the core engine enables RefExpo to handle large-scale projects with minimal or no configuration. However, for large projects with unique structures, some configuration might be necessary. Fortunately, the widespread use of IntelliJ often means guides are available for configuring such projects in this IDE.

The tool is available as a plugin that can be installed via the IntelliJ Marketplace[1] or utilized in IntelliJ's headless mode through the command line by using the source code [2]. To leverage RefExpo, users must first import their project into IntelliJ, ensure proper indexing of the source code, and then access the RefExpo toolbox. Although building the project is not mandatory, it is recommended to confirm successful project import and indexing by attempting a build before

[1] `https://plugins.jetbrains.com/plugin/23684-refexpo`,
[2] `https://github.com/vharatian/RefExpo`

starting the evaluation. Figure 3.1 illustrates the RefExpo toolbox interface. To streamline analysis and improve usability for large projects, RefExpo provides configurable filters. These filters enable users to apply regular expressions for evaluating files, classes, and methods. Furthermore, RefExpo identifies circular references and includes options to exclude them at the file, class, or method levels. While filtering can be applied post-evaluation using the generated CSV file, pre-application of filters helps prevent the creation of excessively large output files. Moreover, pre-filtering can significantly reduce execution time when working on extensive projects by limiting the scope of the evaluation.

RefExpo generates a CSV file that lists all detected references. The graph structure is encoded in the CSV file as edges, defined by a source and a target locator. Each locator is characterized by three primary attributes: file, class, and method. Table 3.1 details the attributes associated with each locator in the CSV file. In the literature [41, 42], graphs that incorporate dynamic links are often referred to as call graphs, whereas those restricted to static references are termed dependency graphs. Throughout this work, these terminologies are used consistently. A comprehensive usage guide, including a brief demonstration video, is available in the replication package.

3.1 Implementation

To comprehend the workings and nuances of RefExpo's inspection mechanism, it is crucial to explore its two-phase approach.

RefExpo operates with two inspection phases, both resembling each other closely. This two-phase strategy arises due to IntelliJ's lazy indexing policy, which defers indexing until a file is accessed or indexing becomes necessary, thereby avoiding delays for the user. However, RefExpo requires complete indexing to be performed before evaluation. As a result, in the first phase, RefExpo systematically traverses all accessible files and performs a comprehensive pass over their Abstract Syntax Tree (AST) using the PSI API provided by the IntelliJ plugin SDK [43].

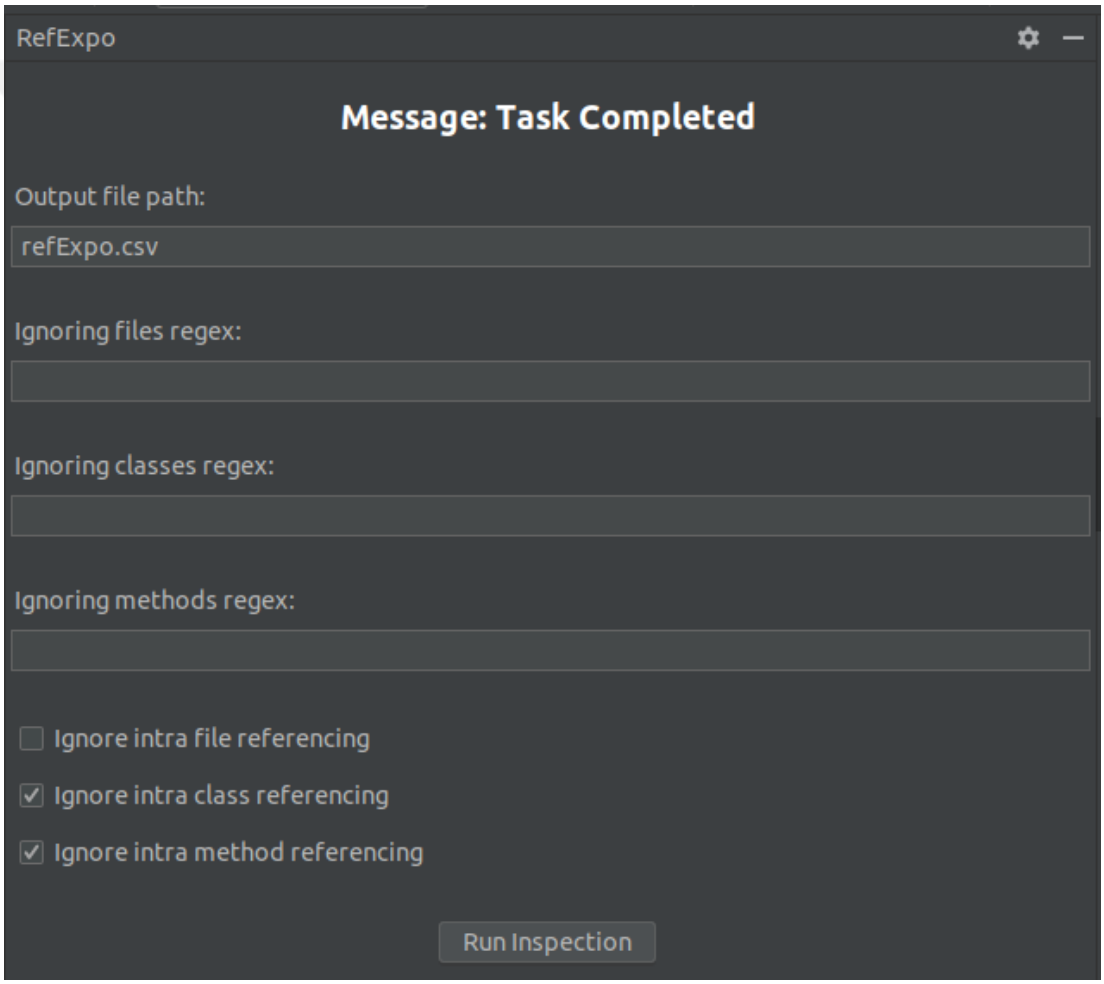


Figure 3.1: RefExpo toolbox screenshot

For each AST element, IntelliJ identifies and provides a list of references where the statement is used. In this initial phase, RefExpo resolves these references, prompting IntelliJ to index the target files and retain this information.

Subsequently, RefExpo executes a second traversal, revisiting all files and elements. During this pass, it evaluates the resolved references and logs their occurrences. Figure 3.2 illustrates the overall structure of RefExpo's inspection process.

RefExpo focuses exclusively on internal references, disregarding external ones. For example, while it is common for Java projects to reference `java.lang.String`, RefExpo filters out such references. To achieve this, RefExpo initially narrows the scope of project files to those managed under the version control system (VCS). This filtering step excludes build artifacts and dependency files, as IntelliJ can analyze references within compiled CLASS and JAR files. Ultimately, only references with targets included in the VCS are considered valid within the project.

As an output, RefExpo produces a CSV file cataloging all detected references. Each reference entry includes three locators—File, Class, and Method—for both the source and target. The presence of these locators depends on the reference type. For instance, references involving HTML files, which lack classes or methods, are recorded with only the file name, leaving the class and method columns empty in the CSV.

3.2 Evaluation

In this chapter, we present the evaluation of RefExpo at both micro and macro levels to assess its performance in producing dependency graphs. This evaluation is crucial to ensure that RefExpo accurately identifies dependencies and performs reliably in various contexts. The micro-level evaluation involves detailed testing with specific data suites, while the macro-level evaluation examines RefExpo's effectiveness across a range of popular GitHub repositories. This approach ensures

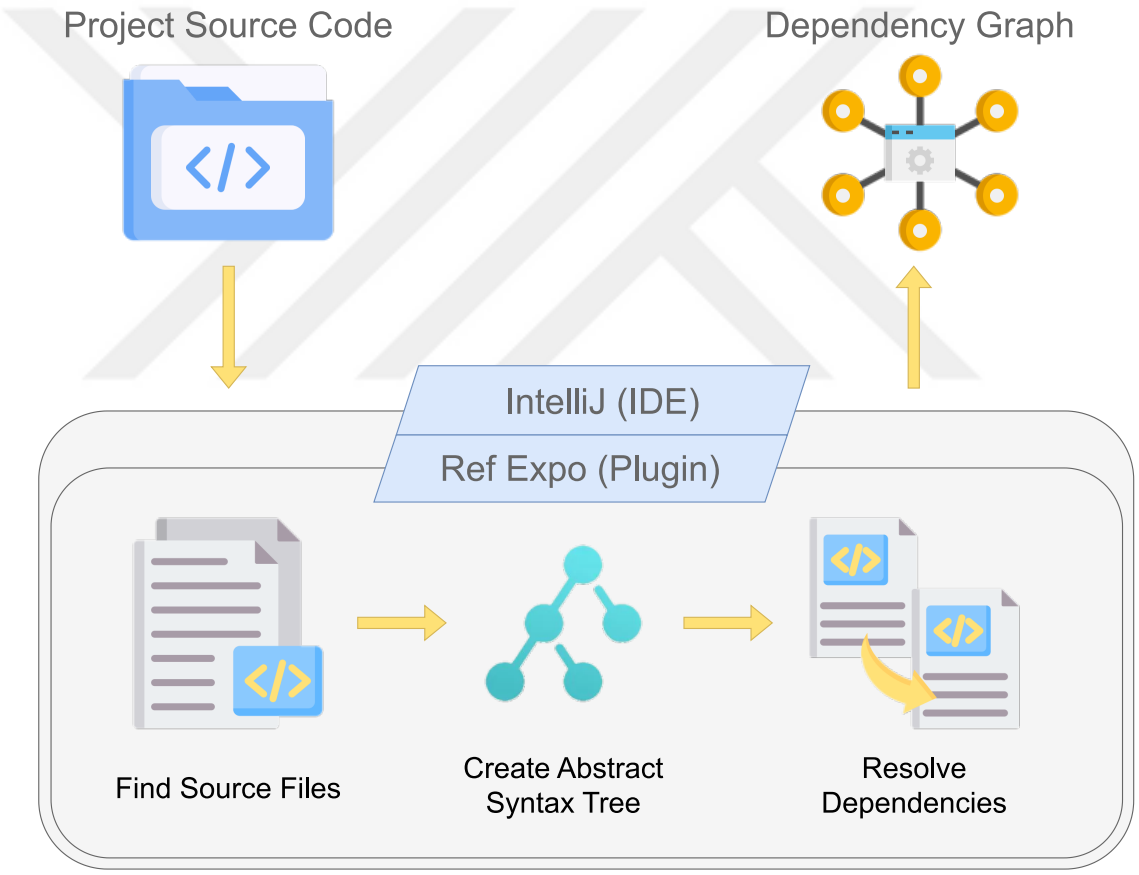


Figure 3.2: RefExpo general structure

a comprehensive assessment of RefExpo's accuracy and robustness.

3.2.1 Micro Level

To validate RefExpo performance, we inspected two available data suites, Judge [44] and PyCG [25]. Although these test suites were originally designed for constructing call graphs which are different from dependency graphs(Described in Chapter 3.2.1.1), we found them helpful in gauging the accuracy of RefExpo. During the micro evaluation phase, we ran RefExpo inspection repeatedly and compared the outputs of it with the expected graph to identify and fix bugs to ensure accurate results.

Since the test suits are initially designed for call graphs, we refine them to ensure they provide accurate measurements for RefExpo. We remove all test cases that fall outside of the intended evaluation scope of RefExpo. Nevertheless, we provided the performance analysis for both the initial and cleaned-up versions of the test suits to remove any confusion regarding the legitimacy of our evaluation. During the cleaning process, we remove the following test cases:

- ***Dynamic Links.*** A complete description of why we decided to remove these links is provided in.
- ***External References.*** We have removed all the references to external logic, such as references to `java.lang.Integer`. IntelliJ excels in reference mining, indexing external references, evaluating their internal logic, and providing decompilation. Thus, including them in the evaluation would result in an explosion in the output size. Therefore, we decided to exclude them from the RefExpo evaluation scope.

Although the test cases provided in the PyCG test suite are placed in separate files, Judge test cases are defined in `.md` files. Therefore, we process them one step further to place them in the right files and folders with small tweaks for fixing

package names to be consistent with Java's required structure. Nevertheless, both test suites are placed in our replication repository.

3.2.1.1 Call Graph vs Dependency Graph

In this chapter, we clarify the distinctions between call graphs and dependency graphs, which is essential for understanding our evaluation method of RefExpo. While RefExpo produces dependency graphs, we evaluated its performance against tools that generate call graphs since there are not many updated tools for dependency graphs. This explanation aims to elucidate these concepts and justify our evaluation approach.

Call graphs are only concerned about the possible paths and routes that control flow might navigate during execution as a result of function invocation [41, 45] In contrast, Dependency graphs consider all possible components referenced by a given component, including method calls. Although the definitions might sound simple, there might be some confusion involved. Code example 3.1 demonstrates a logic where `func3` obtains an instance of `func1` by calling `func2`. Figure 3.3a and Figure 3.3b present two possible graphs for this logic. The difference between these graphs is that one has an edge between `func3` and `func1`, and the other one does not. The most accurate way of constructing call graphs is to evaluate the runtime information [41, 45]. Therefore, since this logic will result in a function call from `func3` to `func1` during execution, this edge should be included in a call graph. Hence, Figure 3.3a can represent the call graph. Nevertheless, the dependency graphs are flexible in this regard. For instance, if one is concerned with coupling evaluation, depending on their scope, they might want to include or remove the dynamic call edges [46]. Thus, some researchers might find those edges beneficial for their purposes or disturbing [47]. If we look at the exact implementation of the logic in Code Example 3.1 in type strict languages, it is more doubtful that such a link should be included. Since it works against object-oriented design patterns and creates abstraction. Code Example 3.2 shows an equivalent implementation of Code Example 3.1 in Java language. In Java, the language structure suggests the usage of design patterns and interfaces to break this link. Therefore, in these cases,

mining back these links and putting them into the graph might even be a bad idea and be considered as fighting against the language structure. The language provides all these features to increase the modularity of the code and weaken links between unrelated logic. Code Example 3.3 shows a test case example from Judge test suits. As it is obvious, the method `castToTarget` is designed to be ignorant about the class type that is passed to it. Then, tracing back the dependencies and establishing the dynamic links sounds like fighting against a designed principle called separation of concerns [48]. With all said, there is still no general consensus about these dynamic links, and they might not be beneficial for the majority of static code analysis studies. Considering the complexity of mining dynamic links and the doubt about their usefulness in the context of static code analysis, we choose not to include them in the RefExpo evaluations.

Code 3.1: Python code for method referencing

```python
def func1():
    return "Hello from the first function!"

def func2():
    return func1

def func3:
    func = func2()
    print(func())
```

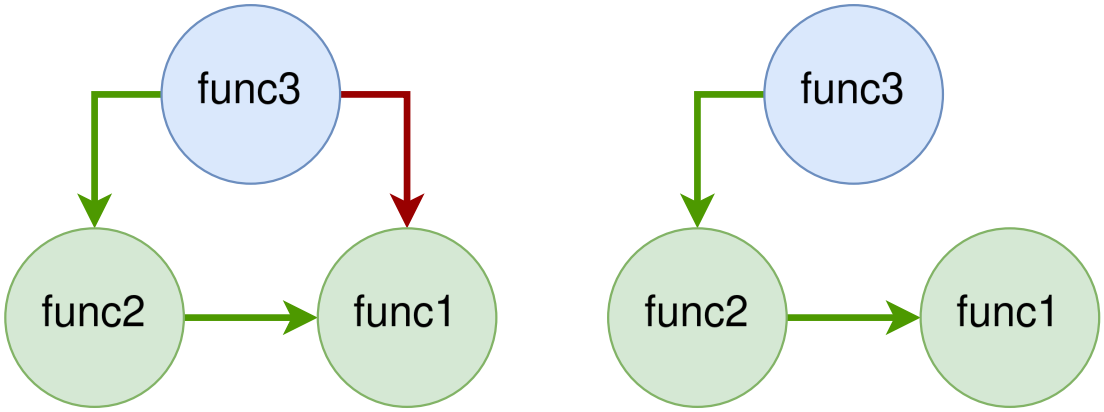
Code 3.2: Java code for interface Implementation

```java
interface Interface {
    void func();
}

class InterfaceImpl implements Interface {
    public void func() {
        System.out.println("Hello World");
    }
}
```

```java
public static Interface generator() {
    return new InterfaceImpl();
}

public static void executeFunc() {
    Interface instance = generator();
    instance.func();
}
```

Code 3.3: Java example of dynamic linkage from Judge test suite

```java
static <T> void castToTarget(Class<T> cls, Object o) {
    T target = cls.cast(o);
    target.toString();
}
```



(a) Call Graph (b) Dependency Graph

Figure 3.3: Graphs for example, Code 3.1

3.2.1.2 Measurements

For measuring the correctness of RefExpo evaluations, we decided to use Recall. Because originally the datasets were designed to construct a call graph, we observe that RefExpo results in low accuracy. This is because there are lots of other

references in the test cases that should not be included in a call graph. Thus, we find Recall a better indicator for the effectiveness of RefExpoṘecall is simply calculated as the ratio of the number of results correctly retrieved (True Positives) to all related results (Specified in the test suite).

$$Recall = \frac{Number\ of\ correctly\ identified\ edges}{Number\ of\ all\ edges\ specified\ in\ the\ test\ suite} \tag{3.1}$$

3.2.1.3 Clean Up

Table 3.2 summarizes the cleanup process over Judge and PyCG test suites. Initially, the Judge test suite contained 112 test cases and a total of 112 edges. 17 (15%) cases were removed, resulting in the final 95 test cases. Among those, 11 (10%) were removed due to being an external link and six (5%) due to being a dynamic link.

The PyCGe test suite initially contains 104 cases and a total of 223 edges. 60 (27%) edges were removed, resulting in the final 163 edges. Among those, 15 (7%) were removed due to being an external link and 45(20%) due to being a dynamic link.

Judge Test Suite. Table 3.3 summarizes RefExpo evaluation results over the Judge test suite, designed to evaluate support for different features of Java. For 95 cases, all of the specified edges were included in the RefExpo output. This resulted in 85% accuracy over the initial test suite and 100% over the cleaned version. In both cases, the resulting accuracy is higher than the maximum accuracy reported by the Judge evaluation, which OPAL yielded as 83% [44, 38].

Micro Evaluation Java

RefExpo yield 100% recall during micro evaluation over the Judge test suite, which is 17% higher than the best recall specified in the Judge evaluation.

PyCG Test Suite. Table 3.3 summarizes RefExpo evaluation results over PyCG test suite. RefExpo resulted in a 97% recall over the cleaned test suite and 68% over the uncleaned one. The result is 5% higher than the maximum accuracy

yielded by PyCG which is 92% [25]. However, we should take into account that their evaluation is provided over the original version of the test suite, not the cleaned one.

Micro Evaluation Python

RefExpo yield 97% recall during micro evaluation over the cleaned version of the PyCG test suite, which is 5% higher than the 92% accuracy reported for PyCG.

3.2.2 Macro Level

We compared the results of RefExpo against five existing tools over four popular GitHub repositories.

Table 3.5: Macro Evalution (Python): Output comparison between RefExpo Pyan, and PyCG. (All Shared: edges that have been identified by all the tools, Two Shared: edges that are only identified by two of the tools, not the other)

Projects	Total Edges	All shared	Two Shared	Shared	Total			Shared			Unique		
					RefExpo	Pyan	PyCG	RefExpo	Pyan	PyCG	RefExpo	Pyan	PyCG
TheAlgorithms/Python	1733	356(21%)	409(24%)	765(44%)	1405(81%)	961(55%)	488(28%)	761(44%)	733(42%)	392(23%)	644(37%)	228(13%)	96(6%)
wting/autojump	566	96(17%)	88(16%)	184(33%)	487(86%)	214(38%)	145(26%)	184(33%)	155(27%)	125(22%)	303(54%)	59(10%)	20(4%)
aboul3la/Sublist3r	236	22(9%)	69(29%)	91(39%)	234(99%)	34(14%)	81(34%)	91(39%)	32(14%)	81(34%)	143(61%)	2(1%)	0(0%)
oarriaga/face_classification	191	19(10%)	25(13%)	44(23%)	182(95%)	44(23%)	28(15%)	44(23%)	44(23%)	19(10%)	138(72%)	0(0%)	9(5%)
Average	682	14%	21%	35%	90%	33%	26%	35%	27%	22%	56%	6%	4%

Table 3.6: Macro Evalution(Java): Output comparison between RefExpo Jarviz, Dependency Finder, and Sonargraph. (All Shared: edges that have been identified by all the tools, Two Shared: edges that are only identified by two of them, not the others)

Projects	Total Edges	All shared	Two Shared	Three Shared	Shared	Totals				Shared				Unique			
						RefExpo	Jarviz	DF	Sonargraph	RefExpo	Jarviz	DF	Sonargraph	RefExpo	Jarviz	DF	Sonargraph
google/guava	9,249	335(4%)	3,726(40%)	2,650(29%)	6,711(73%)	8,382(91%)	4,329(47%)	2,200(24%)	4,369(47%)	6,592(71%)	4,302(47%)	1,941(21%)	3,907(42%)	1,790(19%)	27(0%)	259(3%)	462(5%)
ReactiveX/RxJava	10,738	473(4%)	6,894(64%)	-	7,367(69%)	10,210(95%)	6,969(65%)	1,399(13%)	-	7,367(69%)	6,924(64%)	916(9%)	-	2,843(26%)	45(0%)	483(4%)	-
square/retrofit	712	36(5%)	160(22%)	-	196(28%)	657(92%)	166(23%)	121(17%)	-	193(27%)	161(23%)	74(10%)	-	464(65%)	5(1%)	47(7%)	-
FastXML/jackson-core	899	175(19%)	90(10%)	337(37%)	602(67%)	818(91%)	252(28%)	555(62%)	563(63%)	599(67%)	241(27%)	491(55%)	560(62%)	219(24%)	11(1%)	64(7%)	3(0%)
Average	5,400	8%	34%	48%	60%	92%	41%	29%	55%	59%	40%	24%	52%	34%	1%	5%	3%

Metrics. To compare different tools, we create a list of the edges that the tools identify as edges of the graph, and we evaluate the overlaps. We visualize the overlap analysis outcome by drawing Venn diagrams that show the exact number of overlaps between different tools. Afterward, we define the ratio of unique results count to the total number of edges as a measure of the uniqueness of each tool's results. Similarly, we define the ratio of shared and total edges. It should be noted that all these ratios are calculated based on the total number of edges that have been identified by all the tools. Hence, for each tool the ratio of unique edges plus the ratio of the shared edges is not equal to one but the ratio of its total edges.

Tool Selection. For this study, we have concentrated on evaluating tools that offer assessments for the Java and Python programming languages. We compare the performance of RefExpo against all language-specific tools discussed in Chapter 2.3.

Project Selection. To remove any bias from our evaluation we tried to select popular projects from GitHub. Initially, we used the same methodology that we used to construct our dataset. During our evaluation, we never encountered any project that RefExpo can not evaluate. However, due to limitations of other tools we needed to change our approach accordingly.

Python. Since PyCG and Pyan are only compatible with Python 3.6 and 3.7, we faced a barrier in selecting our repositories. We observed that almost all popular GitHub projects are under active development. Thus, they are updated and are using new technologies. Consequently, we could not run both PyCG and Pyan over any of the most popular repositories in their current version at the time. However, we could successfully run PyCG and Pyan over one of the older revisions of `TheAlgorithms/Python` project. The tools were able to provide an evaluation for the latest version of the project published before February 2019. This is because Python 3.8 was introduced in the same year and the developers started to integrate new language features.

Although it is not a real production project and was developed for educational

purposes, it still contains 1407 files and gained 175 starts. Moreover, we found out we could run PyCG and Pyan over three out of five repositories evaluated in the original PyCG study. They are substantially smaller than the projects we aimed to evaluate and are not maintained for years, which makes them compatible with both tools. Despite, their size and lack of maintenance, they gained more than 5k stars which makes them legitimate projects for evaluation. In the end, we came up with a set of four projects described in the replication package [3], `TheAlgorithms/Python`, `wting/autojump`, `aboul3la/Sublist3r`, `oarriaga/face_classification`.

As demonstrated in Table 3.5 and Figure 3.4 on average RefExpo covered 90% of the identified edges, Pyan 33% of them, and PyCg 26%. Among those edges that have been covered by RefExpo 35% of them were shared, and 56% of them were unique. Similarly, Pyan and PyCG yielded 27% and 22% of shared edges and 6% and 4% of unique edges. We have observed that Pyan is the second best-performing tool regarding the ratio of shared and unique edges. Our results show that there is a good number of edges that are identified by only two of the tools (21%) while the majority of these edges are between RefExpo and one of the Pyan or PyCG. Usually, the intersection area between PyCG and Pyan is below 1% meaning that RefExpo is more agreeable compared to both of them.

Macro Evaluation Results for Python

RefExpo yield superior results regarding the unique and shared number of edges which are 50% and 8% higher than the next best-performing tool, Pyan. RefExpo is the most agreeable among the three tools by having more one-to-one overlaps.

Java. We have better options when it comes to Java evaluation since the tools are more mature and are compatible with recent Java versions. Nevertheless, we could not pick the top most popular repositories since some of the tools only can provide evaluation for JAR files. Still, some of the most popular repositories have their JAR file built and ready to use over Maven. Even though one of the projects that is listed as the top repositories, Jackson, does not build into a single JAR file, we could run our evaluation over a submodule of it `jackson-core` which its JAR

[3] `https://github.com/vharatian/RefExpo`,

file is available on public repositories. Besides, the test codes are not available in the built JAR files available in the public repositories. Hence, we needed to manually delete all the test files to make a fair comparison between the tools. We further removed all the sample codes included in the repository for the same reason. Finally, we could not configure Sonargraph on two of the projects since it was constantly crashing and we could not find the right configuration. However, we decided to keep the project since other tools were able to provide evaluations. This results in four final projects described in the replication package [4], `google/guava`, `FastXML/jackson-core`, `ReactiveX/RxJava`, `square/retrofit`.

As demonstrated in Table 3.6 and Figure 3.5 on average RefExpo covered 92% of the identified edges, Jarviz 41%, Dependency Finder 29%, and Sonargraph 55%. Among those edges that have been covered by RefExpo 59% of them were shared, and 34% of them were unique. Similarly, Jarviz, Dependency Finder, and Sonargraph yielded 40%, 24%, and 52% of shared edges and 1%, 5% and 3% of unique edges. We have observed that Sonargraph is the second best-performing tool regarding the ratio of shared edges. Nevertheless, Dependency Finder yielded the second-best results regarding unique results. Our results show that there is a good number of edges that are identified by only two or three of the tools (52%) while the majority of these edges are identified by RefExpo. Usually, the intersection areas that fall outside of RefExpo circle have a value less than 1%.

Macro Evaluation Results for Java

RefExpo yield superior results regarding the unique and shared number of edges which are 31% and 7% higher than the next best-performing tools, Sonargraph and Dependency Finder. RefExpo is the most agreeable among the four tools by having more overlaps.

[4] `https://github.com/vharatian/RefExpo`

3.3 Dataset

To facilitate researchers in accessing an extracted dependency graph efficiently, we evaluated 20 repositories written in Python and Java [5]. Table 3.7 provides an overview of the projects included in the initial release of our dataset. The selection process involved identifying prominent repositories on GitHub using its search API and ranking them by their star count. Subsequently, we filtered repositories based on their primary programming languages, focusing specifically on Java and Python. Repositories deemed as small code samples were manually reviewed and excluded from the dataset. Additionally, we incorporated projects excluded during filtering but later evaluated as part of our macro evaluation phase. These projects were initially chosen for their compatibility with other tools used in our comparison with RefExpo results. As detailed in our replication package [3], the dataset encompasses projects of varying sizes, ranging from 11 files to 15K files. This method ensures a versatile dataset aimed at supporting researchers with diverse needs. We intend to continuously expand this dataset and welcome collaborations with researchers, offering to customize it to address specific requirements.

[5] `https://github.com/vharatian/RefExpo`

Parameter Name (source, target)	Description
Path	Path to the file where the location is defined
Line	Specific line number within the file
Class	Name of the class where the location is found
ClassFull	Fully qualified class name, including its package path
Method	Name of the method where the location is found
MethodFull	Fully qualified method name, including the class and package
Structure	Hierarchical structure of the location in the file, such as a method defined inside another method or a class nested in another class

Table 3.1: Locator Attributes in RefExpoś CSV output

Table 3.2: Micro test suites clean up summary

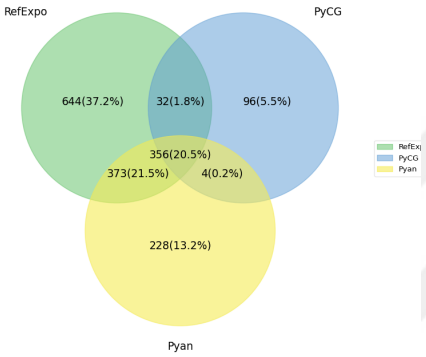
	Judge	PyCG
Test Cases	112	104
Total Edges	112	223
External Links	11	15
Dynamic Links	6	45
Total Clean edges	17	60
Cleanup	15%	27%

Table 3.3: Micro evaluation for Java over Judge test suite. '*' indicated the numbers for the cleaned data suite

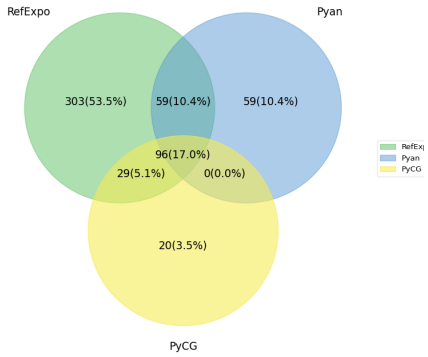
Feature	Cases	Cases*	Corrects	Recall	Recall*
Class Loading	4	0	0	0	0
Dynamic Proxies	1	1	1	1	1
Java 8 Interface Methods	7	7	7	1	1
Java 8 Invoke Dynamics	11	11	11	1	1
JVM Calls	5	4	4	0.8	1
Library	5	4	4	0.8	1
Modern Reflection	8	8	8	1	1
Non Virtual Calls	5	5	5	1	1
Reflection	20	20	20	1	1
Serialization	14	7	7	0.5	1
Signature Polymorphic Methods	7	6	6	0.86	1
Static Initializers	8	8	8	1	1
Types	6	5	5	0.83	1
Unsafe	7	7	7	1	1
Virtual Calls	4	2	2	0.5	1
Total	112	95	95	0.85	1

Table 3.4: Micro evaluation for Python over PyCG test suite. '*' indicated the numbers for the cleaned data suite

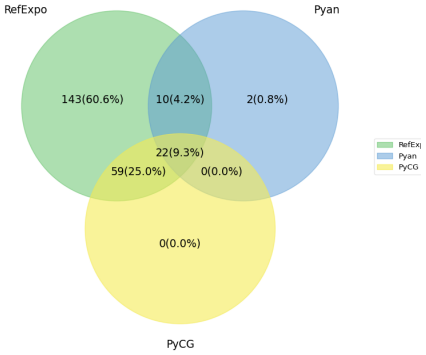
Feature	Tests	Edges	Edges*	Corrects	Recall	Recall*
returns	4	12	8	8	0.67	1
lambdas	5	14	9	9	0.64	1
classes	22	52	42	42	0.81	1
args	6	14	7	7	0.5	1
decorators	7	22	15	15	0.68	1
mro	7	16	14	11	0.69	0.79
dicts	12	19	13	13	0.68	1
exceptions	3	3	3	3	1	1
dynamic	1	1	0	0	0	1
imports	14	14	14	14	1	1
assignments	4	15	15	15	1	1
direct_calls	4	10	5	5	0.5	1
builtins	3	10	4	4	0.4	1
generators	6	18	16	7	0.39	0.44
functions	4	4	4	4	1	1
external	6	11	2	2	0.18	1
Total	104	223	163	151	0.68	0.93



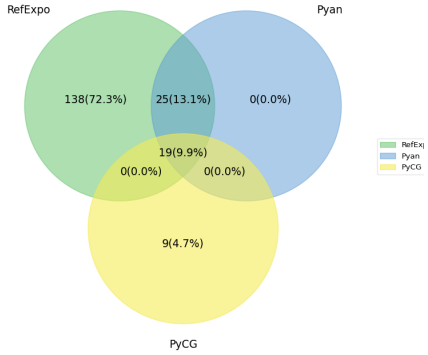
(a) Macro evaluation over TheAlgorithms/Python



(b) Macro evaluation over wting/autojump

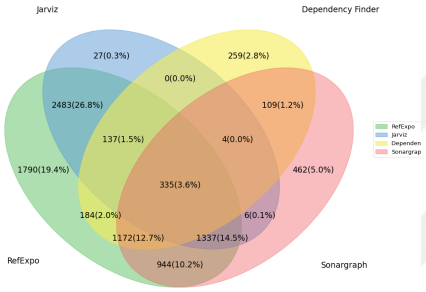


(c) Macro evaluation over aboul3la/Sublist3r

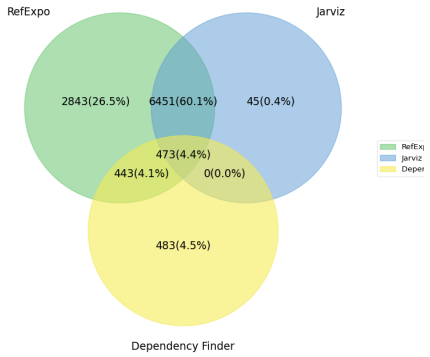


(d) Macro evaluation over oarriaga/face_classification

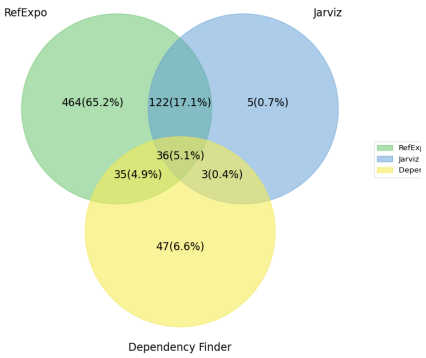
Figure 3.4: Macro evaluations of Python projects



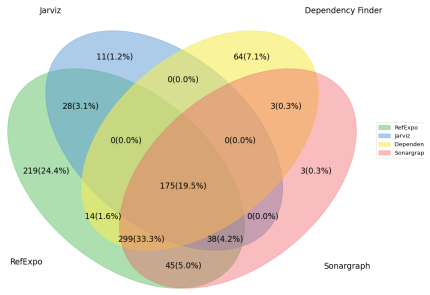
(a) Macro evaluation over google/guava



(b) Macro evaluation over ReactiveX/RxJava



(c) Macro evaluation over square/retrofit



(d) Macro evaluation over FastXML/jackson-core

Figure 3.5: Macro Evaluation of Java projects.

Table 3.7: List of projects included in the initial version of our datasets

Project	Language	Stars	files	commits	First Commit
TheAlgorithms/Python	Python - 99%	176K	1,407	3.302	2016
ytdl-org/youtube-dl	Python - 99%	127K	978	18,801	2008
huggingface/transformers	Python - 99%	120K	4,044	15,079	2018
django/django	Python - 97%	75K	6,786	32,399	2005
tiangolo/fastapi	Python - 99%	68K	2,136	3,765	2018
home-assistant/core	Python - 99%	67K	15,441	72,258	2013
pallets/flask	Python - 99%	66K	250	5,243	2010
wting/autojump	Python - 99%	16K	41	841	2008
aboul3la/Sublist3r	Python - 99%	9K	11	138	2015
oarriaga/face_classification	Python - 99%	6K	72	148	2017
spring-projects/spring-boot	Java - 98%	72K	9,761	47,298	2012
elastic/elasticsearch	Java - 99%	67K	27,743	75,176	2010
spring-projects/spring-framework	Java - 98%	54K	10,137	29,213	2008
google/guava	Java - 99%	49K	3,306	6,335	2009
ReactiveX/RxJava	Java - 99%	48K	1,960	6,079	2012
square/retrofit	Java - 95%	42K	428	2,185	2010
apache/dubbo	Java - 99%	40K	4,390	7,385	2011
skylot/jadx	Java - 91%	38 K	2000	2,140	2013
dbeaver/dbeaver	Java - 99%	36K	8,492	25,307	2012
FastXML/jackson-core	Java - 99%	2K	414	2,776	2011

Chapter 4

Methodology

The methodology follows a structured approach comprising five key steps:

1) Eight notable Java and Kotlin repositories are selected from GitHub to establish the ground truth dataset.

2) Using BFSig, five unique significance scores are computed for each file based on the topological properties of the dependency graph.

3) These significance scores are integrated into existing baseline techniques, ABF and JBF, via BFSig.

4) A survey of 12 contributors from the selected repositories is conducted to gather ground truth data, where participants provide insights into Bus Factor (BF) metrics for the entire project and specific subfolders, referred to as targets in this study.

5) The impact of incorporating FS on the performance of baseline methods is evaluated using the collected ground truth data.

Figure 4.1 illustrates the sequential framework of our study.

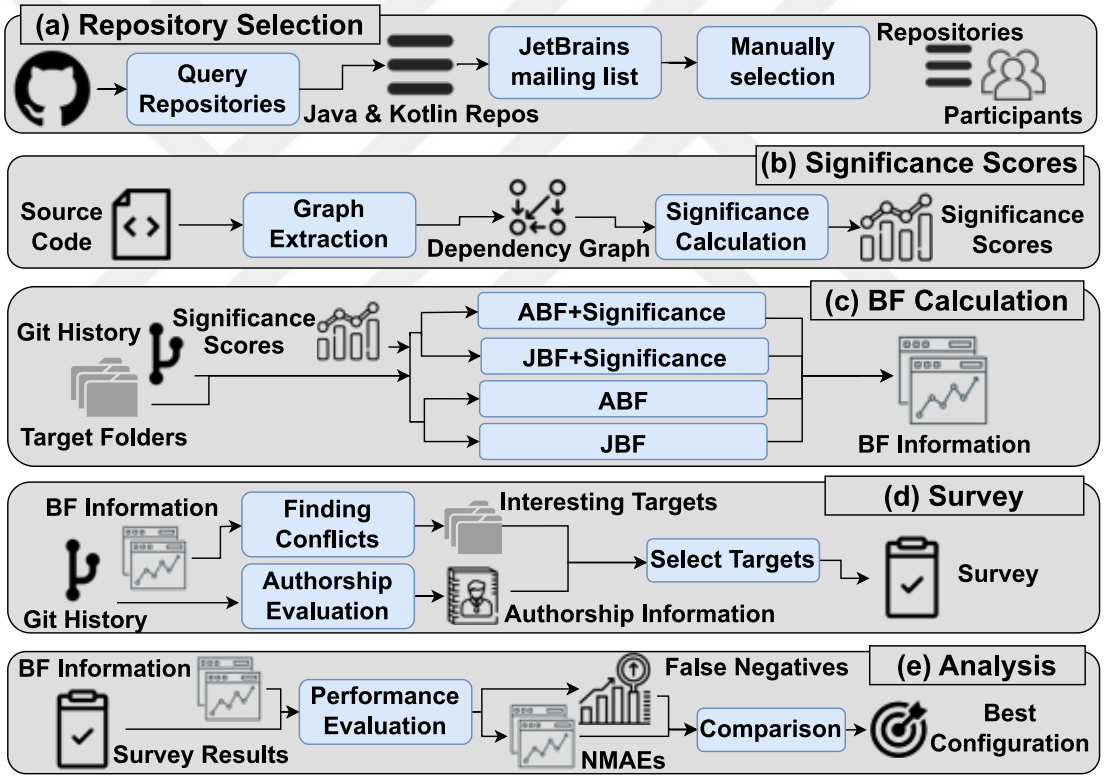


Figure 4.1: BFSig Study Structure

4.1 Research Questions

In this research, we investigate the following questions:

RQ1: Does accounting for FS improve BF estimation?

The primary distinction between BFSig and previous baseline methods is its capacity to consider file significance. To assess the impact of FS on improving BF estimation, we compare the performance of ABF and JBF against BFSig, using the newly collected ground truth data from this study.

RQ2: Are BF algorithms applicable to assess folders' BF?

This research represents the first attempt to estimate the Bus Factor (BF) at both the overall project level and the level of individual subfolders. As a result, it is crucial to evaluate whether BF estimation techniques (including BFSig and the baseline methods) can reliably assess the BF of subfolders. Specifically, we examine whether the error in estimating the BF for subfolders significantly differs from the error associated with estimating the BF for complete projects.

RQ3: Does BF estimation accuracy depend on the BF_{actual}?

Avelino et al. [7] noted a significant decline in the performance of ABF for projects where the BF exceeds two. This research question investigates the accuracy of various algorithm variants across different BF ranges and evaluates the impact of incorporating FS on the assessment.

RQ4: Does accounting for FS improve identifying bottlenecks related to low BF?

The primary objective of BF assessment is to support decision-makers in addressing potential risks. One notable risk is the emergence of bottlenecks caused by a large number of development requests directed at a module with a low BF. This situation aligns with the "code red" community smell described by Palomba et al. [11] and recognized by Jabrayilzade et al. [5] as one of the most common

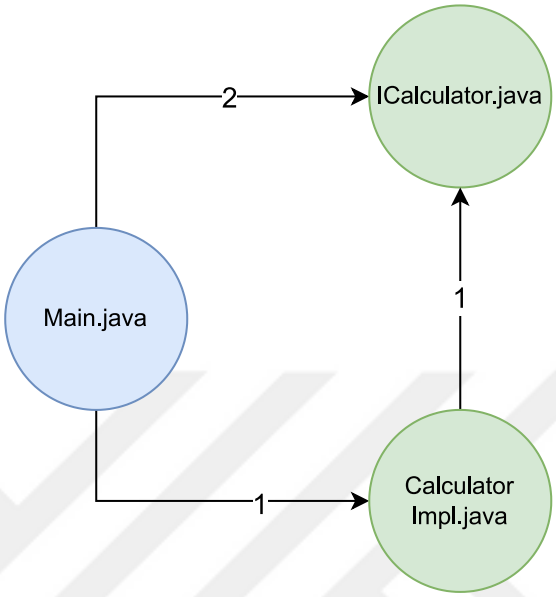


Figure 4.2: Dependency Graph for Sample Code 4.1

community smells. This highlights the need to evaluate whether incorporating FS enhances bottleneck detection when compared to baseline methods.

4.2 Significance Calculation

As shown in Figure 4.1b, BFSig creates a dependency graph (DG) for a project to calculate significance scores for its files. The DG is a directed and weighted graph where references to file components are treated as dependencies. In a typical DG, nodes can represent various software elements such as modules, files, classes, or methods. In our case, since the focus is on computing file significance scores, we model files as nodes in the graph. Given the directional nature of component references, the graph employs directed edges. These edges are weighted to capture the frequency of references between components. Self-referencing edges, representing intra-file references, are excluded as they are irrelevant to the metrics computed. For example, the DG generated from the code in Code 4.1 is depicted in Figure 4.2. This graph comprises three nodes: `Main.java`, `IAdder.java`, and `AdderImpl.java`, along with three edges: (`Main.java`, `IAdder.java`, 1),

(`Main.java`, `AdderImpl.java`, 2), and (`AdderImpl.java`, `IAdder.java`, 1).

Code 4.1: Sample code resulting in DG shown in Figure 4.2

```
Main.java:
public class Main {
    public static void main(String[] args) {
        IAdder adder = new AdderImpl();
        System.out.println(adder.addNumber(5.0, 0.6));
    }
}

ICalculator.java:
public interface ICalculator {
    float addNumber(float a, float b);
}

CalculatorImpl.Java
public class AdderImpl implements IAdder {
    @Override
    public float addNumber(float first, float second) {
        return fist + second;
    }
}
```

To generate dependency graphs, we employ RefExpo, a tool built using the JetBrains IntelliJ plugin SDK [49]. IntelliJ, a widely used Integrated Development Environment (IDE), provides a scripting API that facilitates access to its internal operations, such as code indexing and reference resolution. With this API, RefExpo analyzes all files within a Git repository by applying filters, parsing file elements, and resolving inter-file references. A detailed discussion of RefExpo is presented in chapter **??**.

Once the dependency graph is created, BFSig computes significance scores

for each file using algorithms such as PageRank, In-Degree, Out-Degree, All-Degree, and Betweenness Centrality. These algorithms are collectively termed as Significance Indicators (SI) throughout this thesis. The resulting scores capture different topological characteristics of each node within the graph. For these calculations, BFSig utilizes Python in combination with the NetworkX library [50].

4.3 Baselines Adjustments

Figure 4.1c provides an overview of this process. ***Incorporating Significance Scores:*** ABF and JBF calculate developer authorship in a project as the proportion of files authored by a developer to the total number of files. Authorship coverage is similarly computed, representing the percentage of authored files relative to the total file count. To incorporate significance scores, we enhanced ABF and JBF by summing the significance scores of files instead of simply counting them during authorship and coverage calculations. Based on our SI definition, each file's significance score is within the $[0, 1]$ range, except for Betweenness Centrality (BC). Thus, these scores are utilized in both overall and individual authorship coverage calculations. When all file significance scores are set to 1, the algorithms behave identically to their original baseline implementations. Algorithm 2 details the modifications made to the original ABF process described in Algorithm 1.

For each SI, we construct a set of scores that quantify the significance of each file. We adapted ABF and JBF to incorporate these SI sets and their respective scores into their calculations. The revised tools then produce distinct BF estimates for each SI.

BF for Folders. Jabrayilzade et al. [5] reported that 61% of engineers express interest in acquiring BF insights for high-risk project modules. Nevertheless, since a single module may span multiple directories, identifying modules automatically within a software project remains a complex task. In this study, we evaluate BF information at the subfolder level within a project and adapt ABF and JBF to support this analysis.

Algorithm 2 Modified Algorithm with Structural Changes

```
1: procedure ComputeContributorImpact(Contributors, Artifacts)
2:     ContributorImpact ← {}
3:     TotalImpactScore ← Sum(CalculateImpact(Artifacts))
4:     for all Contributor ∈ Contributors do
5:         ImpactScore ← Sum(CalculateImpact(Contributor.Artifacts))
6:         Coverage ← ImpactScore / TotalImpactScore
7:         ContributorImpact[Contributor] ← Coverage
8:     return ContributorImpact
9:
10: procedure AssessOverallImpact(Contributors, Artifacts)
11:     TotalImpactScore ← Sum(CalculateImpact(Artifacts))
12:     RelevantArtifactImpact ← 0
13:     for all Artifact ∈ Artifacts do
14:         if IsContributorRelevant(Artifact, Contributors) then
15:             RelevantArtifactImpact ← RelevantArtifactImpact + CalculateImpact(Artifact)
16:     return RelevantArtifactImpact / TotalImpactScore
```

Minor Contributors. During our analysis, we identified situations where a small group of primary contributors authored a target, and their removal led to a knowledge loss slightly below the 50% threshold. To compensate, JBF incorporates minor contributors to meet the 50% mark, often leading to substantial overestimations. Conversely, we noted that ABF circumvents this issue by excluding authors contributing less than 10% of the total authorship. Private communication with the ABF authors revealed that this pruning strategy was introduced to mitigate overestimation, although it was not explicitly mentioned in their original work [7]. To tackle this challenge, we modify the JBF algorithm by omitting authors whose contributions are less than 10% of the target's total authorship compared to the top contributor, thereby reducing overestimation effectively.

4.4 Validation Survey Methodology

To obtain ground truth data for evaluating the impact of file significance on baseline methods, we organized a survey involving practitioners. The survey design was a collaborative effort among all authors, refined through multiple

discussions. To enhance its validity, two pilot sessions were conducted with internal engineers engaged in diverse projects before distributing it to the final participants. The responses were gathered using Google Forms [51]. A publicly accessible printed version of the survey questions is available online[1].

Recruitment and Project Selection. Figure 4.1a provides a high-level depiction of this step. BFSig supports reference analysis across various programming languages, including Java, Python, JavaScript, and HTML. Nonetheless, the current design is tailored for strictly typed programming languages. Considering that Java and Kotlin are among the most prominent strictly typed languages, we focused our analysis on popular GitHub repositories primarily written in these languages. In one case, where a repository contained a small amount of Python code, we extended our analysis to the Python files as well.

To identify relevant repositories, we initially compiled a list of 161 repositories from GitHub using the keywords "Java" and "Kotlin." After a manual review, we excluded unrelated repositories, such as tutorials, documentation, and JavaScript-based projects, narrowing the list to 73 repositories. To encourage participation while adhering to GDPR regulations, we cross-referenced the contributors of these repositories with JetBrains' mailing database. This database includes individuals who have opted in to receive marketing and analytics-related communications from JetBrains. Through this process, we selected eight repositories with the highest number of contributors in the database who demonstrated significant authorship in the project. Table 4.1 summarizes the selected repositories and their associated metadata. Finally, we reached out to all contributors in the database linked to these repositories, resulting in a participant pool of 92 individuals.

Informed Consent and GDPR. The survey includes a detailed informed consent form. Furthermore, we consulted GDPR specialists to verify that our research approach complies with the "legitimate interest" clause outlined in the GDPR framework.

[1] `https://figshare.com/articles/conference_contribution/Bus_Factor_Estimation_Survey/22818053`

Survey Protocol. Our survey is organized into two main sections. The first section collects general demographic information about participants, such as their "Years of experience in the IT industry" and their "Current role within the project." The second section provides a list of eight targets, asking participants to rate their familiarity with the code in these targets and the duration of their involvement, using a 1-5 Likert scale. Participants are also asked to estimate the BF and identify key engineers for each target. Unlike Avelino et al. [7], who shared their findings with participants for validation, we directly gather respondents' opinions on the BF value. This method aims to reduce response bias, as sharing preliminary results could influence participants' views if the findings are not entirely accurate.

In the survey, participants are given a predefined set of subfolders. Figure 4.1d illustrates this process. Our objective is to select meaningful subfolders that represent distinct project modules large enough to evaluate the BF comprehensively. Subfolders are selected for each participant based on their authorship coverage within the folder, ensuring they are not tasked with evaluating unfamiliar targets. The initial set of subfolders is created by exporting JBF authorship data for each contributor. We also prioritize subfolders where BF estimates from different SIs show higher levels of disagreement. When all estimators agree on a target's BF value, any inaccurate estimate affects all SIs equally, limiting insights into their relative performance.

Finally, we ask participants if incorporating knowledge distribution across project subfolders could improve project health evaluations. While Jabrayilzade et al. [5] posed a similar question in their surveys, we believe participants' views on the relevance of BF for subfolders may change after evaluating the BF of the selected targets.

4.5 Analysis

Error Measurement. Mean Absolute Error (MAE) is a commonly utilized metric for evaluating the accuracy of estimation models [52]. Although metrics like Root Mean Squared Error (RMSE) are also available, MAE is deemed more suitable for our analysis. This is because, for an actual BF of five, an estimate of $BF = 100$ is not significantly different from $BF = 500$. MAE minimizes the over-penalization of estimators for extreme outliers.

However, MAE alone does not fully represent the context of our study, as the impact of an error varies depending on the actual BF value. For example, an error of one unit is negligible for a target with $BF = 100$, but it is critical for $BF = 1$. To address this, we normalize the absolute error (AE) and derive the Normalized Absolute Error (NAE):

$$NAE = \frac{|BF_{actual} - BF_{estimate}|}{BF_{actual}} \tag{4.1}$$

When BF_{actual} is zero, we substitute one as the normalization value to avoid division by zero.

We categorize data points into three groups based on their BF values: BF=0,1; BF=2,3; and BF¿3. While the risk level linked to BF ideally depends on the target's size and the number of contributors, we consider these three categories as representing high risk (BF=1), moderate risk (BF=2,3), and low risk (BF¿3). This categorization also ensures a well-balanced distribution of survey responses, as demonstrated in Table 5.1.

To prevent excessive penalization of estimators in scenarios with low performance, we establish an upper bound for the NAE. For instance, if the actual BF is two, estimating it as 10, 12, or 14 results in negligible practical differences. We identify such cases by computing the Z-Score [53] of the NAEs, which measures the relative deviation using the mean and standard deviation.

$$Z-Score = \frac{Data - Mean}{Standard\ Deviation} \tag{4.2}$$

We identify the minimum value such that less than 1% of the data achieves a higher Z-Score: $Z{-}Score_{max}$. Subsequently, we establish the upper limit for NAE:

$$NAE_{max} = Z{-}Score_{max} * Standard\ Deviation + Mean \tag{4.3}$$

$$NAE = \begin{cases} NAE_{max}, & \text{if } value > NAE_{max} \\ value, & \text{if } value \leq NAE_{max} \end{cases} \tag{4.4}$$

Afterwards, we average all NAE_i to come up with NMAE, which is an indicator of the overall normalized error of an algorithm.

$$NMAE = \frac{1}{N} \sum^{N} NAE_i \qquad N = Size(Dataset) \tag{4.5}$$

Bottleneck Identification. Beyond NMAE, we assess the estimators' ability to detect bottlenecks by analyzing the proportions of accurate, overestimated, and underestimated results. Overestimations correspond to false negatives, reflecting missed detections of bottlenecks caused by low BF, while underestimations lead to false positives and unnecessary alarms. Since BF is pivotal in identifying potential workforce bottlenecks, minimizing false negatives is emphasized to reduce the risk of fostering a false sense of security. Thus, when two estimators exhibit comparable accuracy, the one with fewer false negatives is deemed more reliable.

Bias Removal. For all metrics, we initially calculate the average results for each project individually and then compute the overall mean across projects. This approach addresses potential bias toward projects with higher respondent counts, as shown in Table 4.1. Such bias may originate from differences in participant numbers per project (see Table 4.1). Additionally, factors like team size, team structure, and source code organization may introduce further bias.

Data Confusion. To address inconsistencies in respondents' answers, we treat each response as a separate data point. We argue that this method provides a more precise evaluation for NMAE than averaging conflicting values to calculate the error. For instance, in cases with two differing actual BF estimates, NMAE applies equal penalties to estimators that provide a value between the two estimates, while penalizing those outside this range more severely.

Table 4.1: Summary of Metadata and Respondents for Repositories

Repository	**Resp.**	**Contrib.**	**Commits**	**Stars**	**Files**	**First Commit**
elastic/elasticsearch	1	1,812	68,720	63.7K	24,257	2010
quarkusio/quarkus	5	799	35,150	11.7K	20.412	2018
apache/flink	2	1,128	33,301	21.2K	21,319	2010
mybatis/mybatis-3	1	208	4,620	18.5K	1,880	2010
JetBrains/compose-multiplatfor	3	112	1,828	11.7K	2,021	2020
openjdk/jdk	0	687	73,835	15.9K	66,096	2007
apache/dubbo	0	498	6,716	38.9K	3,625	2011
spring-projects/spring-boot	0	988	42,697	67.2K	8,751	2012

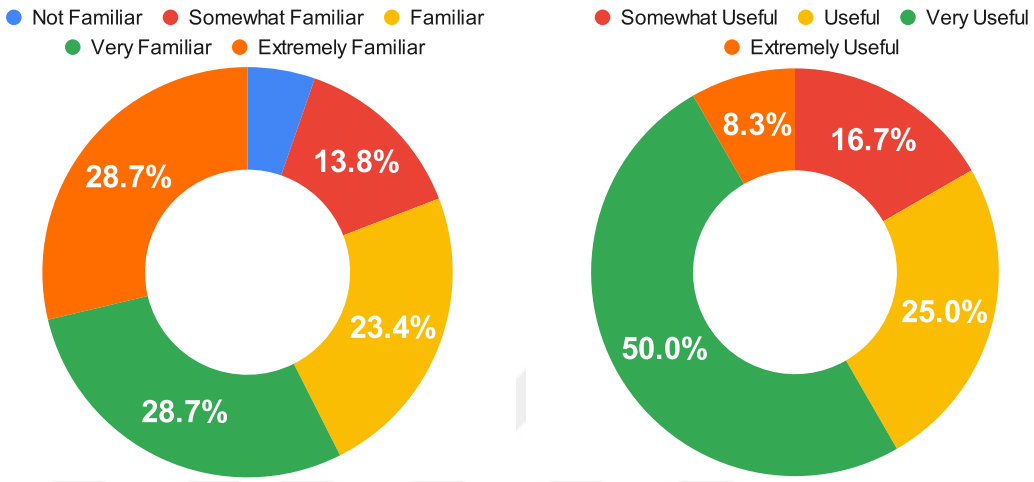
Chapter 5

Results

We collected 12 responses spanning five projects, resulting in a total of 95 data points. One target was excluded because a respondent described it as trivial: "NA: this module is not so important and rather trivial." The raw dataset is publicly available here[1]. Table 4.1 summarizes the projects included in the study. Among the participants, four identified as team leads, while the others described their roles as developers, programmers, or software engineers. On average, respondents reported approximately 11 years of experience in the IT industry and around three years of involvement with the respective project. Additionally, they indicated being adequately familiar (scoring 3+ out of five on a Likert scale) with roughly 81% of the targets, as depicted in Figure 5.1a.

As shown in Figure 5.1b, 83% of participants agreed that evaluating the bus factor for individual folders enhances understanding of knowledge loss risks within a project, and none rated BF data for folders as "Not Useful." This highlights the significance of assessing BF at the folder level and creating a dataset that offers ground truth on BF for both entire projects and their components.

[1] `https://figshare.com/articles/dataset/Bus_Factor_Estimation_Survey_Resultskajhdfj/22820471`



(a) Familiarity with Targets (b) Usefulness of Subfolders' BF

Figure 5.1: Overview of Respondents' Feedback

5.1 RQ1: Does accounting for FS improve BF estimation?

Table 5.1 presents the NMAE for baselines with and without (None) SIs across five evaluation groups of BF values. The rationale behind selecting these groups is explained in Chapter 5.2. We set the Z-Score threshold at 3.84, which flags 0.96% of the data points as over-penalizing. Our findings indicate that BFSig generally outperforms the baselines across nearly all configurations within the evaluation groups. However, there are a few instances where incorporating certain SIs results in slightly worse performance compared to the baselines. Figures 5.2 and 5.3 illustrate a comparison of NMAEs for various algorithm variants.

SI Impact on BF Estimation

Incorporating certain SIs enhances BF estimation across all evaluation groups, with JBF under BC demonstrating the best performance.

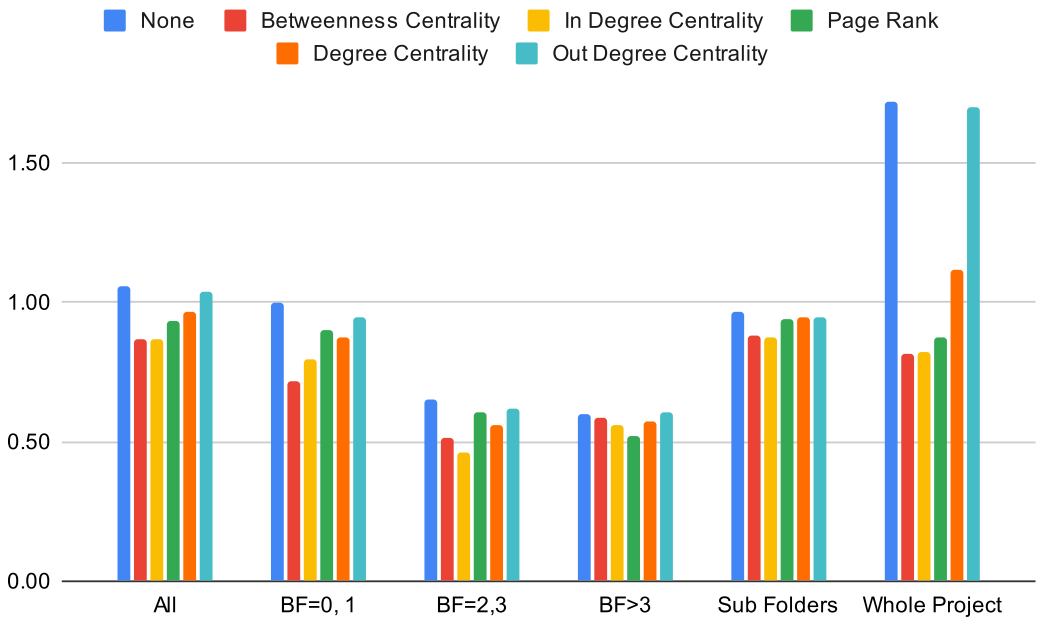


Figure 5.2: Comparison of NMAE values illustrating the influence of SIs on ABF (Avelino)

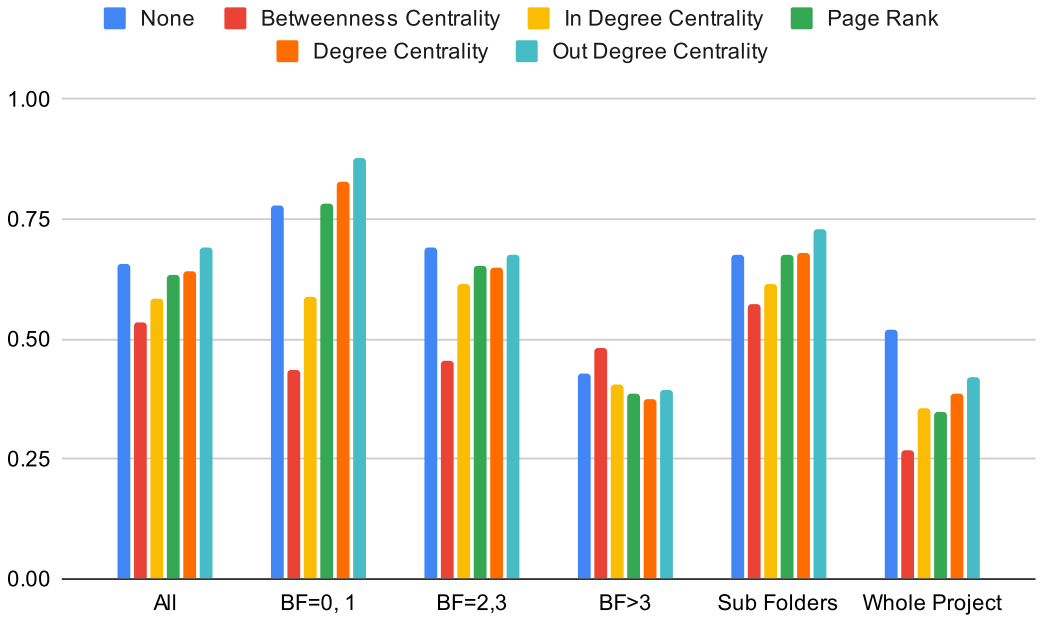


Figure 5.3: Comparison of NMAE values demonstrating the effect of SIs on JBF (Jabrayilzade)

5.2 RQ2: Are BF algorithms applicable to assess folders' BF?

As shown in Table 5.1, ABF performs better when estimating the BF of project subfolders, whereas JBF achieves better results for the whole project. However, the best-performing SIs provide more accurate BF estimates for the entire project than either baseline. Figure 5.4 highlights the performance improvement of the top SIs, IDC and BC, in estimating BF for the whole project compared to subfolders. The results indicate that BFSig demonstrates significantly better performance improvement when estimating BF for the entire project. Incorporating file significance leads to up to 9.5 times more improvement for ABF and 1.5 times for JBF in whole-project BF estimation compared to subfolder estimation.

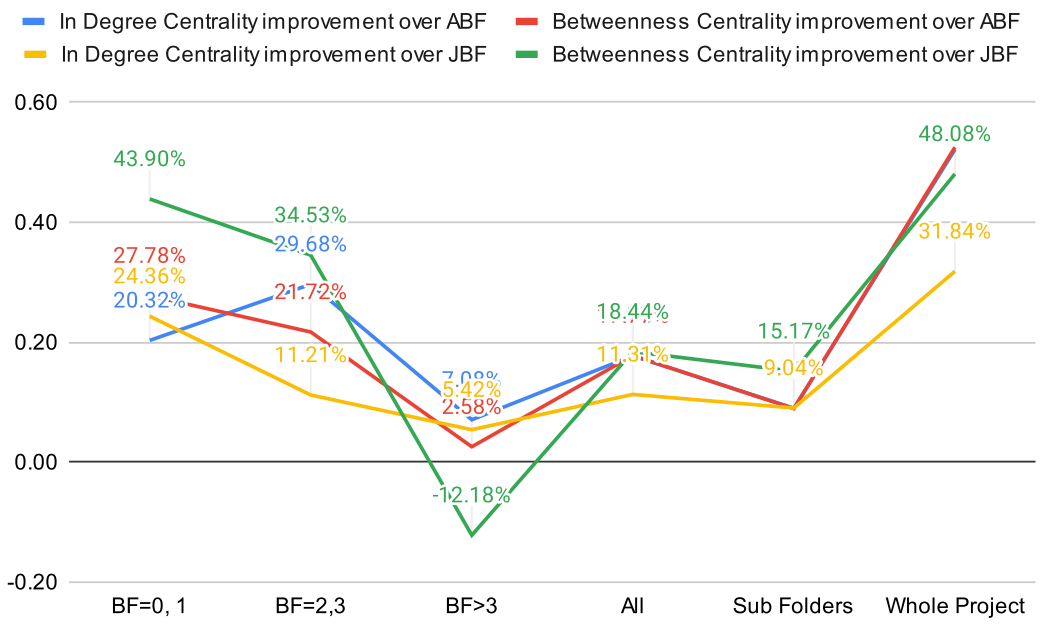


Figure 5.4: Improvement in NMAE over baselines achieved by top-performing SIs

BF for Subfolders

Existing BF algorithms are applicable for evaluating the BF of project subfolders. SIs demonstrate substantial potential in improving BF estimation for the entire project compared to subfolders.

5.3 RQ3: Does BF estimation accuracy depend on the BF_{actual}?

From Table 5.1, we observe that the NMAE of estimator variants generally decreases as the BF increases. However, as the BF value grows, the performance improvement over the baselines diminishes for the best SIs. For instance, BC+JBF performs slightly worse than the baseline for `BF>3`, despite delivering the best overall performance. Figure 5.4 compares the performance gains of BC and IDC across the three BF groups. BFSig demonstrates significant performance improvements for BF=1, a case where Avelino et al. reported ABF performs best.

BF Size and Accuracy

The accuracy of BFSig improves as the BF increases, although its performance gain over the baselines decreases.

5.4 RQ4: Does accounting for FS improve identifying bottlenecks related to low BF?

Figure 5.5 presents a comparison of different algorithm variants in terms of accurate, lower, and higher BF estimations. The results show that BC performs slightly better than the baselines, while other SIs demonstrate slightly worse performance in terms of accurate estimations. However, all SIs generate fewer false negatives compared to the baselines. Among them, BC achieves the best performance, reducing false negatives by up to 18%.

Best SI

BC outperforms other algorithms in terms of NMAE, accuracy, and minimizing false negatives in our dataset.

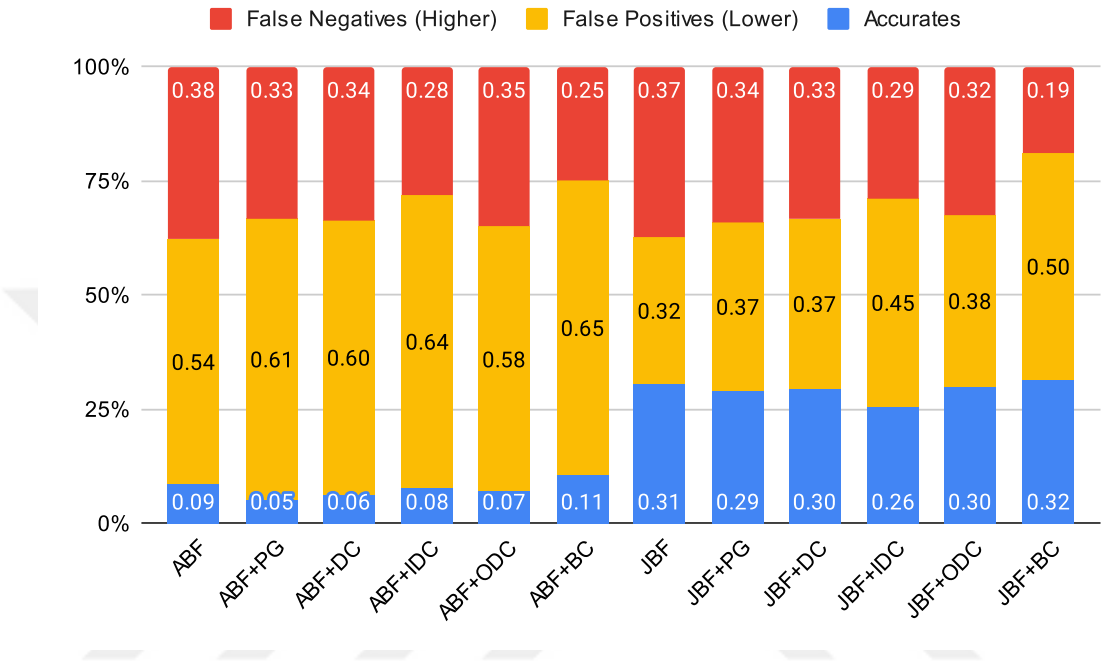


Figure 5.5: Effect of SIs on Accurate estimates, False Positives (lower), and False Negatives (higher) compared to baselines

Table 5.1: Impact assessment of significance indicators on baseline models using NMAE↓. Top-performing values are highlighted and underlined for clarity.

Base Line	Significance Indicator	All	0 >BF >1	2 >BF >3	BF>3	Sub Folders	Whole Project
Data Point Count		95	27	32	36	83	12
ABF (Avelino)	None	1.06	1.00	0.65	0.60	0.96	1.72
	Betweenness Centrality	**0.87**	**0.72**	0.51	0.58	**0.88**	**0.82**
	In Degree Centrality	**0.87**	0.79	**0.46**	0.56	**0.88**	**0.82**
	Page Rank	0.93	0.90	0.60	**0.52**	0.94	0.87
	Degree Centrality	0.97	0.88	0.56	0.57	0.94	1.12
	Out Degree Centrality	1.04	0.95	0.62	0.61	0.94	1.70
JBF (Jabrayilzade)	None	0.66	0.78	0.69	0.43	0.68	0.52
	Betweenness Centrality	**0.54**	**0.44**	**0.45**	0.48	**0.57**	**0.27**
	In Degree Centrality	0.58	0.59	0.62	0.41	0.62	0.35
	Page Rank	0.64	0.78	0.65	0.39	0.68	0.35
	Degree Centrality	0.64	0.83	0.65	**0.38**	0.68	0.39
	Out Degree Centrality	0.69	0.88	0.68	0.39	0.73	0.42

Chapter 6

Discussion

In this chapter, we discuss the broader implications of our findings, providing directions for researchers and practitioners.

6.1 Implications of Results

Despite the significance of the issue, the BF metric is not widely adopted in practice. As Jabrayilzade et al. [5] note, "19% of the respondents have ever worked on a project, where the bus factor was communicated to them." Although most respondents in their study are familiar with the BF concept, the limited usage of BF may be attributed to the lack of an easy-to-use BF estimation tool. Another possible reason could be that managers prefer not to disclose BF information to their teams.

Assessing BF of Subfolders. Both our survey findings and those of Jabrayilzade et al. [5] indicate that practitioners find BF data on specific parts of projects valuable. As we demonstrate that BF algorithms can effectively estimate the BF of project subfolders, we conclude that a BF tool should report data for both the entire project and its components. We encourage other researchers to expand our dataset by including more data points on the actual BF of project parts.

However, selecting meaningful subfolders for surveys is not a straightforward task. For instance, one respondent mentioned that many subfolders in their project "contain examples and tutorials which are much easier to support without prior knowledge about the project." Additionally, subfolders in some projects might not correspond to distinct software components, as they could span multiple folders. As another respondent noted, "Code path is not a good way for the assessment since a module contains codes in multiple code paths. And codes in a certain code path may be involved with multiple modules." Nevertheless, the respondent suggested, "To estimate the bus factor better, I would recommend splitting the project into smaller parts according to the functionalities."

Impact of FS on BF Estimation. We believe incorporating FS into BF estimation offers several advantages. First, our results demonstrate that certain SIs improve BF estimation across all evaluation groups. Additionally, all SI indicators produce fewer false negatives compared to the original algorithms. Since false negatives create a false sense of security, they are considered riskier from the practitioners' perspective. In contrast, false positives, being false alarms, can be dismissed without introducing any actual risk. Furthermore, accounting for FS aligns with engineers' perspectives, as they often recognize that some components are more critical than others [14, 13]. Based on these findings, we believe BFSig, when operating with JBF and BC, can effectively identify risks associated with low BF. It outperforms previously published tools, produces false negatives at a relatively low rate (19%), and offers a two-fold improvement over the baselines in this regard.

BFSig Complexity. BFSig introduces additional complexity compared to baseline approaches, primarily due to the challenges of extracting a project's dependency graph, which requires using supplementary plugins. However, with advancements in technology, such as GitHub's new reference resolution feature [54], integrating this tool with cutting-edge technologies is now more feasible.

6.2 Future Work

This chapter outlines directions for future research to improve bus factor (BF) estimation. We highlight challenges like dataset availability and suggest incorporating factors such as modules, code complexity, bug proneness, and execution time to enhance accuracy and reliability.

Data Sets. The availability of datasets containing BF ground truth data is highly limited. This limitation may arise from the challenges of data collection while complying with GDPR regulations and avoiding biasing the respondents. By gathering BF information at the subfolder level of projects, researchers can address this issue by obtaining multiple data points from each respondent.

Privacy concerns [5] may also discourage practitioners from sharing BF information, leading to a lower response rate. Nevertheless, acquiring such ground truth data can enhance estimator performance and facilitate the adoption of advanced methods like supervised machine learning. Therefore, the creation of such a dataset represents an essential future direction for advancing this domain.

Modules. The existing BF estimators assess the BF of an entire project without factoring in the modular structure within the project. The departure of a module-specific BF engineer halts the development of that module, and for a critical module, this can lead to a BF issue for the entire project. Consider a system with three equally critical modules, each managed by a single engineer. If any of these engineers were to leave, the resulting BF issue would affect not only their respective module but also the entire project. In such a scenario, the BF is effectively one, but conventional BF algorithms would estimate it as two, assuming each engineer has 33% ownership of the project. To address this discrepancy, we aim to incorporate modular structures into BF estimations. As mentioned earlier, identifying modules within projects can be challenging. One potential approach is to use the modules defined by developers in the build scripts as a basis for modular estimation.

Code Complexity. Jabrayilzade et al. [5] highlight that practitioners

seek insights into the "code complexity of the implicated parts" alongside BF information. This observation is consistent with Curtis et al. [55], who investigate the relationship between three complexity metrics and maintenance performance. Their findings indicate that all three metrics are linked to the programmer's ability to maintain a software component independently. Thus, it is clear that code complexity plays a role in assessing the risk associated with modules having low BF. Incorporating code complexity into BF estimation could therefore enhance its accuracy.

Bug Proneness. Bug proneness serves as a potential indicator of software component maintainability. It can be assessed based on the count of reported bugs [56, 57] or the detection of code smells [58]. Given the strong connection between bug proneness and maintainability, incorporating these metrics could enhance the accuracy of BF estimation.

Execution Time. Wang et al. [20] evaluate the significance of software components using dynamic metrics, such as the proportion of control time allocated to each component during production. While these metrics are challenging to compute, their application in assessing component significance for BF evaluation could lead to notable performance improvements.

Chapter 7

Threats to Validity

In this chapter, we address the potential threats to validity, categorized into construct validity, internal validity, and external validity. Construct validity pertains to whether the tools and methods employed effectively measure what they are designed to assess. Internal validity focuses on the reliability of the study's results and the extent to which they are free from biases or inconsistencies. External validity examines the generalizability of the findings beyond the specific context of this study. Each category identifies specific factors that could impact the reliability and applicability of our conclusions.

Construct Validity. We use IntelliJ to export the project's dependency graph, which introduces a potential threat since IntelliJ is not primarily designed for comprehensive inspections. Its optimization prioritizes delivering a smooth user experience and avoiding potential UI freezes by minimizing system resource usage. As a result, IntelliJ does not maintain a complete reference table but resolves references dynamically to conserve memory. Additionally, some indexes are stored on the file system and only loaded into memory as needed. Initially, IntelliJ resolves references using in-memory indexes, then incrementally checks file system indexes, making it impossible to ensure that all references are accounted for. For large projects with tens of thousands of files, keeping all indexes in memory is impractical. While it is challenging to measure the impact of this

limitation—since no dependency graph exists for large projects to serve as a benchmark—IntelliJ remains one of the most robust tools for such analyses. We believe its inspections are sufficiently reliable for the purposes of this study.

Furthermore, the indexing process is incremental, meaning the tool does not index less critical resources unless explicitly accessed. To address this potential limitation, BFSig mitigates the issue by traversing all files and resolving their references before starting the final evaluation. This approach simulates accessing the files, prompting IntelliJ to complete the indexing process.

Internal Validity. First, since our evaluation of significance scores involves identifying the most influential nodes in the dependency graph, files without references are excluded from the graph and the bus factor estimation. These files may vary in importance—ranging from abandoned code, which is irrelevant for BF estimation, to critical components like infrastructure configurations, shell scripts, or documentation. We acknowledge that further research is needed to incorporate files without references into the estimation process, as deeming them trivial could lead to an underestimation of BF. If these files were unimportant, they likely would not have been included in the repository. To address this issue, we asked practitioners about subfolders containing many such files. The respondents indicated that these files are largely unimportant, describing them as auto-generated documentation or sample projects. Thus, the proportion of significant files excluded from the analysis should be minimal for the evaluated projects.

Second, since questions about the bus factor and knowledge distribution could be perceived as assessing individual contributions [5], respondents' answers may not fully reflect their true beliefs. To mitigate this risk, we explicitly stated in the survey description that the results would remain anonymized and that the purpose of the bus factor tool is to evaluate existential risks within the project.

Finally, the data points we collect may contain inconsistencies. To minimize this risk, we included descriptive questions to clarify the concept of BF and reduce

misunderstandings. Nevertheless, a potential issue arises from differing interpretations of the actual BF value. We observed that while practitioners generally agree on the BF for the entire project (with variations of ± 1), their opinions about subfolders vary more significantly. For instance, different respondents provided divergent BF estimates for the same subfolder. To address this issue, we treat each reported point as an independent data point, penalizing estimators if their values fall outside the reported range. However, this approach could artificially inflate the error of the estimators.

External Validity. There is a potential risk of sampling bias, as the repositories we evaluated may not fully represent all projects. Our analysis is limited to open-source repositories primarily developed in Java and Kotlin, which could introduce bias related to specific team dynamics and code structures. Additionally, we focused on high-star repositories on GitHub, which may further skew results, as these are mature projects likely to have well-organized workforces to sustain their success.

Moreover, the final set of repositories and participants was selected based on the number of contributors who consented to receiving marketing emails from JetBrains. This constraint narrowed the pool of potential participants, resulting in the selection of eight repositories. Expanding the dataset while adhering to GDPR requirements was beyond the scope and time frame of this project. However, the dataset we compiled contains 95 data points, which is more extensive than other publicly available datasets with BF scores.

Most of the projects we analyzed include detailed guides for importing the project into IntelliJ, suggesting that these repositories are developed with advanced tools. This is another possible source of bias. Finally, while we acknowledge the potential bias introduced by our recruitment process, it is important to note that collecting ground truth data for BF estimation necessitates focusing on a limited set of projects. As a result, these limitations are intrinsic to studies of this nature.

RefExpo Limitations. RefExpo, like other tools, has its limitations. In terms of tracing dynamic references between components, RefExpo is not as

effective as some other dependency extraction tools. While it is optimized for evaluating dependency graphs, it is not specifically designed to provide the highest accuracy for call graphs. Nonetheless, our evaluations indicate that RefExpo achieves a sufficient level of recall, capturing a significant portion of call graphs. A detailed comparison of dependency graphs and call graphs is provided in Chapter 3.2.1.1.

Additionally, although IntelliJ supports parsing a wide range of programming languages and RefExpo theoretically can evaluate these languages, we have only optimized and tested RefExpo's performance for Python and Java. We enhanced RefExpo's functionality using a micro-evaluation process that leverages various IntelliJ APIs. To achieve optimal results for other programming languages, this optimization and evaluation process would need to be repeated for each language.

Lastly, our dataset is still in its early stages and currently includes 20 projects for Python and Java. We plan to expand the dataset by incorporating more projects and supporting additional languages, such as JavaScript and C#.

Chapter 8

Conclusion

We highlight the key contributions of this article across five main areas. Firstly, we propose a novel approach for estimating the Bus Factor (BF) of a project. This approach leverages file significance scores derived from a project's dependency graph, incorporating metrics such as PageRank, In-/Out-/All-Degree, and Betweenness Centrality. As part of this effort, we developed a prototype tool, BFSig, which integrates these significance scores into two existing baseline tools: Avelino (ABF) and Jabrayilzade (JBF). BFSig offers a practical solution for practitioners to assess the BF of their projects.

Secondly, BFSig introduces the ability to compute BF not only for entire projects but also for individual subfolders. This unique capability enables the collection of a larger dataset, supporting a more comprehensive evaluation.

Thirdly, we gathered a new dataset through a practitioner survey, capturing their insights on the BF of their projects and various subfolders.

Fourthly, we explore IntelliJ's potential as a tool for generating dependency graphs. Building on this exploration, we present RefExpo, a reusable tool built on IntelliJ for dependency graph generation.

Finally, we initiated the creation of a dataset featuring dependency graphs from

20 popular GitHub repositories, encompassing Java and Python projects. This dataset aims to support research in software analytics by providing ready-to-use, high-quality dependency graphs, alleviating the challenges of data extraction for researchers.

We evaluated BFSig's performance using the Normalized Mean Absolute Error (NMAE) and the number of False Negatives in identifying potential bottlenecks. The results demonstrate that BFSig outperforms the baseline tools in terms of accuracy (fewer errors) and bottleneck identification. Additionally, we provided a detailed comparison of BFSig's performance for each significant indicator. Among these, Betweenness Centrality (BC) delivered the best results against our collected ground truth. When using BC, BFSig achieved a 17% and 18% reduction in NMAE compared to ABF and JBF, respectively. It also reduced False Negatives by 13% and 18% relative to the same baselines.

Furthermore, our analysis of the behavior of various significance indicators across different ranges suggests that BFSig's performance could be further improved by combining these indicators into a unified significance score. However, this enhancement falls outside the scope of this study, as it would require a more extensive dataset that could be divided into training, testing, and validation subsets.

In future work, we plan to expand our ground truth dataset, enabling a deeper investigation into the benefits of merging significance indicators. We also aim to explore incorporating additional indicators, such as Code Complexity and Bug Proneness, into our evaluations.

Bibliography

[1] I. Rus, M. Lindvall, and S. Sinha, "Knowledge management in software engineering," *IEEE software*, vol. 19, no. 3, pp. 26–38, 2002.

[2] "Isbsg repository release 10." URL: `https://www.isbsg.org/`, 2007.

[3] M. Nassif and M. P. Robillard, "Revisiting turnover-induced knowledge loss in software projects," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 261–272, 2017.

[4] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik, "On the abandonment and survival of open source projects: An empirical investigation," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–12, 2019.

[5] E. Jabrayilzade, M. Evtikhiev, E. Tüzün, and V. Kovalenko, "Bus factor in practice," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, (New York, NY, USA), p. 97–106, Association for Computing Machinery, 2022.

[6] L. Williams and R. R. Kessler, *Pair programming illuminated*. Addison-Wesley Professional, 2003.

[7] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "A novel approach for estimating truck factors," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–10, 2016.

[8] N. Zazworka, K. Stapel, E. Knauss, F. Shull, V. R. Basili, and K. Schneider, "Are developers complying with the process: An xp study," in *Proceedings*

of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '10, (New York, NY, USA), Association for Computing Machinery, 2010.

[9] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "Assessing the bus factor of git repositories," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 499–503, 2015.

[10] F. Ricca, A. Marchetto, and M. Torchiano, "On the difficulty of computing the truck factor," in *Product-Focused Software Process Improvement* (D. Caivano, M. Oivo, M. T. Baldassarre, and G. Visaggio, eds.), (Berlin, Heidelberg), pp. 337–351, Springer Berlin Heidelberg, 2011.

[11] F. Palomba, D. Andrew Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, "Beyond technical aspects: How do community smells influence the intensity of code smells?," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 108–129, 2021.

[12] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus, "Quantifying and mitigating turnover-induced knowledge loss: Case studies of chrome and a project at avaya," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 1006–1016, 2016.

[13] S. M. Srinivasan, R. S. Sangwan, and C. J. Neill, "On the measures for ranking software components," *Innovations in Systems and Software Engineering*, vol. 13, pp. 161–175, Sep 2017.

[14] M. Richards, *Software architecture patterns*, vol. 4. 2015.

[15] T. Zimmermann, N. Nagappan, and A. Zeller, *Predicting Bugs from History*, pp. 69–88. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[16] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Component rank: relative significance rank for software component search," in *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 14–24, 2003.

[17] Q. Gu, S. Xiong, and D. Chen, "Correlations between characteristics of maximum influence and degree distributions in software networks," *Science China Information Sciences*, vol. 57, no. 7, pp. 1–12, 2014.

[18] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Structural Analysis in the Social Sciences, Cambridge University Press, 1994.

[19] W. Pan, B. Song, K. Li, and K. Zhang, "Identifying key classes in object-oriented software using generalized k-core decomposition," *Future Generation Computer Systems*, vol. 81, pp. 188–202, 2018.

[20] L. Wang, X. Du, B. Jiang, W. Pan, H. Ming, and D. Liu, "Keada: Identifying key classes in software systems using dynamic analysis and entropy-based metrics," *Entropy*, vol. 24, no. 5, 2022.

[21] L. PAGE, "The pagerank citation ranking : bringing order to the web," *Technical Report*, 1998.

[22] R. Diestel, A. Schrijver, and P. Seymour, "Graph theory," *Oberwolfach Reports*, vol. 7, no. 1, pp. 521–580, 2010.

[23] O. Cury, G. Avelino, P. Santos Neto, R. Britto, and M. Túlio Valente, "Identifying source code file experts," in *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '22, (New York, NY, USA), p. 125–136, Association for Computing Machinery, 2022.

[24] "Pyan is a python module that performs static analysis of python code.." URL: `https://github.com/davidfraser/pyan`.

[25] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "Pycg: Practical call graph generation in python," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1646–1657, 2021.

[26] "Pyne: Temporal dependency graph extraction from java git repositories." URL: `https://github.com/darius-sas/pyne`.

[27] "Jarviz: a dependency analysis and visualization tool designed for java applications." URL: https://github.com/ExpediaGroup/jarviz/tree/master.

[28] "Sonargraph is a powerful static code analyzer." URL: https://www.hello2morrow.com/products/sonargraph.

[29] "Dependencyfinder: A suite of tools for analyzing compiled java code.." URL: https://github.com/jeantessier/dependency-finder.

[30] "Ndepend offers an in-depth .net code quality management experience via an interactive web report.." URL: https://www.ndepend.com/.

[31] "Rexdep: Roughly extract dependency relation from source code." URL: https://github.com/itchyny/rexdep.

[32] V. Haratian, M. Evtikhiev, P. Derakhshanfar, E. Tüzün, and V. Kovalenko, "Bfsig: Leveraging file significance in bus factor estimation," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, (New York, NY, USA), p. 1926–1936, Association for Computing Machinery, 2023.

[33] V. Haratian, P. Derakhshanfar, V. Kovalenko, and E. Tüzün, "Refexpo: Unveiling software project structures through advanced dependency graph extraction," 2024.

[34] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, "How developers drive software evolution," *Principles of Software Evolution, International Workshop on*, vol. 0, pp. 113–122, 2005.

[35] T. Fritz, J. Ou, G. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," vol. 1, pp. 385–394, 01 2010.

[36] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill, "Degree-of-knowledge: Modeling a developer's knowledge of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, apr 2014.

[37] S. Suzuki, H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "An application of the pagerank algorithm to commit evaluation on git repository,"

in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 380–383, 2017.

[38] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini, "Call graph construction for java libraries," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), p. 474–486, Association for Computing Machinery, 2016.

[39] "Callgrind is a profiling tool that can construct a call graph for a program's run." URL: `https://www.google.com/search?channel=fs&client=ubuntu&q=callgrind`.

[40] T. Sotiropoulos and B. Livshits, "Static Analysis for Asynchronous JavaScript Programs," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)* (A. F. Donaldson, ed.), vol. 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 8:1–8:29, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

[41] B. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 216–226, 1979.

[42] V. Walunj, G. Gharibi, D. H. Ho, and Y. Lee, "Graphevo: Characterizing and understanding software evolution using call graphs," in *2019 IEEE International Conference on Big Data (Big Data)*, pp. 4799–4807, 2019.

[43] "a platform for building ides." URL: `https://plugins.jetbrains.com/docs/intellij/intellij-platform.html`.

[44] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, (New York, NY, USA), p. 251–261, Association for Computing Machinery, 2019.

[45] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, p. 319–349, jul 1987.

[46] L. Briand, J. Daly, and J. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.

[47] K. Borowski, B. Balis, and T. Orzechowski, "Semantic code graph – an information model to facilitate software comprehension," *IEEE Access*, pp. 1–1, 2024.

[48] G. Booch, *Object oriented design with applications*. USA: Benjamin-Cummings Publishing Co., Inc., 1990.

[49] "Intellij platform sdk." URL: `https://plugins.jetbrains.com/docs/intellij/welcome.html`.

[50] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[51] "Google form." URL: `https://www.google.com/forms/about/`.

[52] T. Chai and R. R. Draxler, "Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature," *Geoscientific Model Development*, vol. 7, no. 3, pp. 1247–1250, 2014.

[53] W. M. Mendenhall and T. L. Sincich, *Statistics for Engineering and the Sciences*. CRC Press, 2016.

[54] "Github editor tool." URL: `https://docs.github.com/en/codespaces/the-githubdev-web-based-editor`, 2023.

[55] B. Curtis, S. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 2, pp. 96–104, 1979.

[56] S. Puranik, P. Deshpande, and K. Chandrasekaran, "A novel machine learning approach for bug prediction," *Procedia Computer Science*, vol. 93, pp. 924–930, 2016. Proceedings of the 6th International Conference on Advances in Computing and Communications.

[57] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller, "If your bug database could talk," in *Proceedings of the 5th international symposium on empirical software engineering*, vol. 2, pp. 18–20, Citeseer, 2006.

[58] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 194–218, 2019.