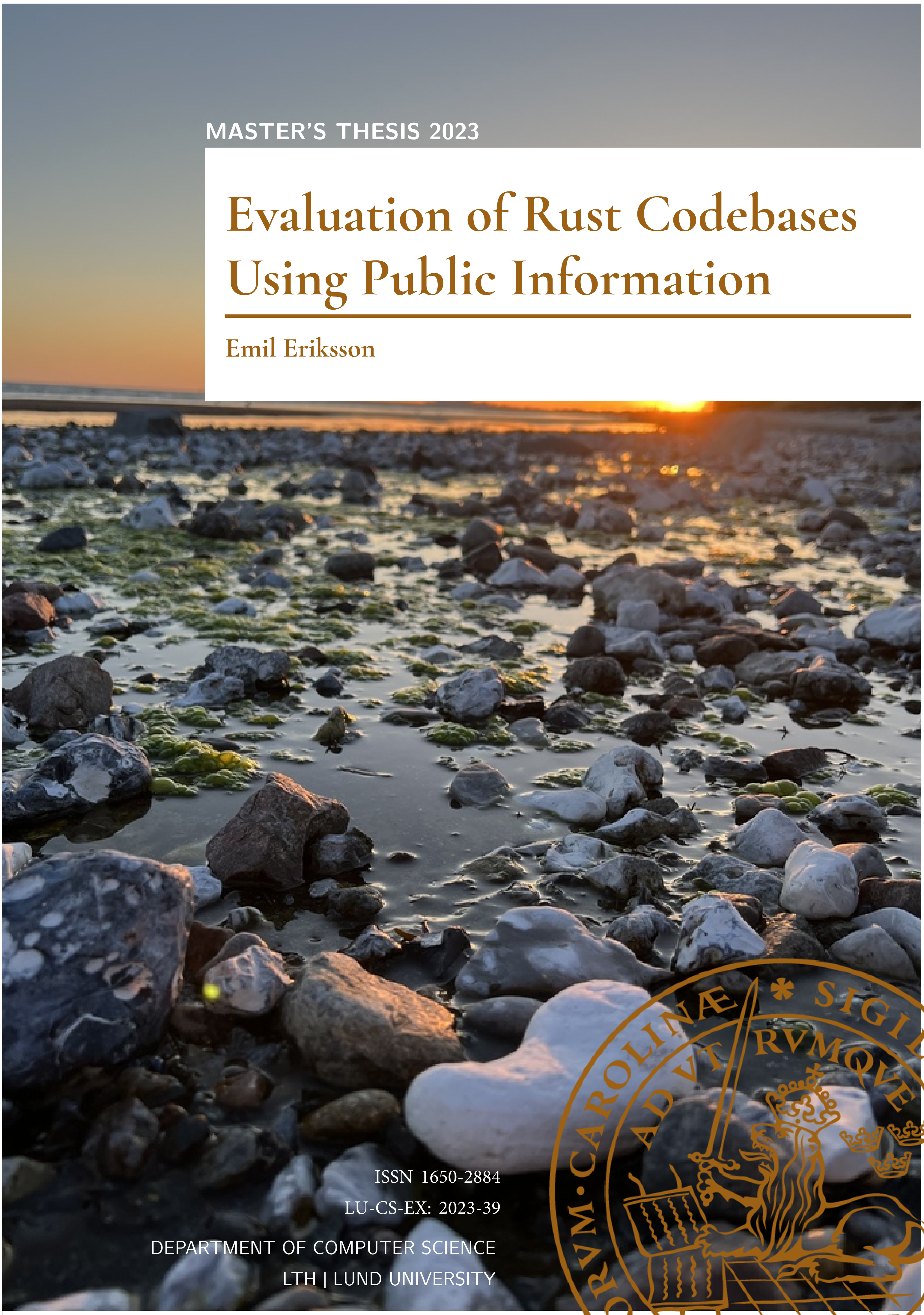


MASTER'S THESIS 2023

Evaluation of Rust Codebases Using Public Information

Emil Eriksson

ISSN 1650-2884
LU-CS-EX: 2023-39

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

LU-CS-EX: 2023-39

Evaluation of Rust Codebases Using Public Information

Utvärdering av kodbaser i Rust utifrån publik information

Emil Eriksson

Evaluation of Rust Codebases Using Public Information

Emil Eriksson
em5184er-s@student.lu.se

June 15, 2023

Master's thesis work carried out at Volvo Car Sverige AB.

Supervisors: Alexandru Dura, alexandru.dura@cs.lth.se
Nikolaos Korkakakis, nikolaos.korkakakis@volvocars.com

Examiner: Christoph Reichenbach, christoph.reichenbach@cs.lth.se

Abstract

Understanding the content of a software project is a complex endeavour. While the Rust programming language promises developers a safer programming language, a program may still include vulnerable code through its dependencies.

In this thesis we present a CLI[1] tool, `cargo-indicate`, to query the dependency tree of Rust projects using standard GraphQL. This tool aggregates data from a variety of sources, such as program analysis tools (cargo-geiger), source control platforms (GitHub), and package registries (crates.io) and exposes them in a schema.

We use this tool to collect data about popular Rust packages, and describe their distribution. We employ a clustering strategy to identify categories of Rust projects.

We conclude that some, but not all, data contain useful information that can help developers understand their dependency tree. We describe three categories of Rust packages, and have reason to believe that project marketing is a significant factor in separating projects.

We believe that our tool provides a novel approach to aggregate data about the Rust ecosystem from different sources, with an interface that can easily be extended. For developers this tool is a possible stage in a future CI pipeline, and for researchers it provides a way of analyzing the Rust ecosystem.

Keywords: Rust, Code Evaluation, Open Source, Crates.io, Cargo, GraphQL, Query, Dependencies

[1]Command-Line Interface

Acknowledgements

This thesis would not have been possible without the contributions, guidance, and insights of a few individuals.

First of all, I owe a debt of gratitude to my supervisor Alexandru Dura. He has gone above and beyond in helping me, and his suggestions have been invaluable in writing the thesis you are currently reading. I am deeply thankful for the time he has taken to help me, and for the insightful discussions we had in his office next to the ever growing stack of coffee cups.

I would also like to express my appreciation for Nikolaos Korkakakis and Julius Gustavsson at Volvo Cars, whose expertise and experience greatly contributed to the quality of this project. Their daily feedback and suggestions significantly enriched this thesis. I am also grateful to Philippe Burlion at Volvo Cars for giving me the opportunity to work with this project, and for his continuous support.

The tool presented in this work would not be possible without Predrag Gruevski, the author of `trustfall`. His interest in my project has been truly motivating, and I want to thank him for his patience in answering my questions. With this I also want to give a wider thanks to the Rust community, for which I wrote this thesis. Without the community and the open source contributors this thesis would not be possible.

Finally, thanks to Lena Eriksson for the cover image.

Contents

Chapter 1
Introduction

In a world where the importance and impact of software rises exponentially, the correctness of that software becomes increasingly important in our lives. The ever increasing number of services being provided to replace the old require no IT professional to notice; In Sweden alone the usage of cash is ever decreasing, important mail is sent via online services, and car owners may find that the most powerful computer in their household is standing on their driveway.

It is no surprise that more and more people rely on software in their everyday life, and when that software fails, or worse, is exploited by malicious actors, things start to fall apart. Historically, many of most severe cases of software failures is due to developer error; Bugs such as Heartbleed[1] went undetected for years, coming into public view only when the effects became much too clear.

While these software bugs may be created by human developers, the mistake is an easy one to make. Memory is notoriously difficult to manage, and programming languages such as C and C++ allow mistakes to happen. While they provide excellent performance and low level control, they require a skilled developer that makes no mistakes to be safe.

The Rust programming language aims to help developers by providing performance characteristics similar to that of C and C++, while ensuring that memory mismanagement is difficult. It also provides a modern and comfortable ecosystem, where third-party code can easily be integrated in their own project. On the other hand, can Rust developers really be sure that the code they do not write themselves holds up to their standards? While a lot of information about projects is publicly available, many developers adds third-party code without much thought. After all, a project that simply helps you parse some specific file format cannot be dangerous, or can it?

This thesis aims to provide a tool that can analyze Rust packages using information that is available to the public, from various sources, some of which are unique to the Rust ecosystem. Developers should be able to use the tool to for example identify dependencies of their project

[1] https://heartbleed.com/

that should be considered for further, perhaps manual, investigation.

This is to be done by identifying *signals* that may indicate the quality or risk associated with a Rust third-party package from the Rust ecosystem. Signals here define any source of data that is available to the public, and may be sampled at any time to provide some value. While the data sampled may in some cases also contain historical data, such as the commit history of a project, some signals do not and simply provide a snapshot of the project at that time. With this definition, some signals that do not inherently contain historical data will need continuous sampling to provide a time series.

The initial goal is not to provide a tool for static code analysis, but rather to collect, compose, and evaluate signals from various sources available to projects written in Rust, and make it possible to write queries to search through a Rust dependency tree.

To ensure the quality of the tool, the signals are to be evaluated. If deemed feasible, the tool should provide some way of filtering a project (for example in a CI[2] environment) on some characteristics.

These goals lead to the following research questions:

- **RQ1**. *What signals contain information that can differentiate a package against another package? (and which do not)?*
- **RQ2**. *Do the selected signals correlate in any way, and if so, how?*
- **RQ3**. *Can we find clusters of Rust packages based on their signals?*
- **RQ4**. *Does combining signals in multiple dimensions provide benefits for analysis of a Rust package?*
- **RQ5**. *Are Rust packages of the same category likely to have the same signal characteristics, and if so, what are they?*
- **RQ6**. *How can we use signals to identify Rust packages in need of manual developer attention?*

[2]Continuous Integration

Chapter 2
Background

This chapter aims to provide an introduction to the Rust programming language (Section 2.1), a motivation for the work done (Section 2.2), as well as presenting related work (Section 2.3).

2.1 The Rust Programming Language

This section aims to provide some background to the Rust programming language, and can be skipped by those familiar with the language. Interested readers are recommended to read [15].

The Rust programming language is a relatively new programming language, first announced by Mozilla in 2010. Described as a language intended to provide developers with performance, reliability, and productivity, the language has grown considerably and was marked as stable in 2015. It has reached widespread adoption, at companies such as AWS, Facebook and Volvo Cars.

At the center of the Rust programming language is an *ownership*-system, used to guarantee memory-safety and thread-safety at compilation. This removes the need for a garbage collector at runtime, while preventing common errors related to memory safety.

This safety, while still providing performance on par with C/C++, made Rust the first language in addition to C and assembly to be accepted for use in the Linux kernel as of version 6.1 [8].

2.1.1 The Ownership Model & The Borrow Checker

The Rust ownership system provides some guarantees against common memory errors at compile-time. While an in-depth explanation will not be provided here (again [15] provides a good explanation), we will provide an overview to provide context for the thesis.

Rust provides several tools to prevent programmer mistakes, or writing code that could contain memory-safety issues. The Rust compiler guarantees memory-safety and thread-safety, which in turn protects the developer against common bugs.

Unlike other languages such as Java or Go, Rust does not need a garbage collector at runtime, to clear unused memory. Instead Rust uses the concept of *ownership* of values.

Listing 2.1: An example of a Rust program that will not compile due to attempt to borrow moved value

```rust
fn foo(x: String) {
    println!("x␣is␣\'{x}\'");
    // ...
} // ‘x‘ is automatically dropped at the end of the scope

fn main() {
    let x = String::from("Hello,␣world!");
    foo(x);
    println!("{x}"); // Will cause compile-time error!
}
```

Listing 2.2: An example of a C program that will compile, but contains a bug

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void foo(x: *char) {
    printf("x␣is␣\'%s\'\n", x);
    // ...
    free(x);
}

int main() {
    char* x = (char*) malloc(sizeof("Hello,␣world!"));
    strcpy(x, "Hello,␣world!");
    foo(x);
    printf("%s\n", x); // ‘x‘ is used after free
    return 0;
}
```

Listing 2.3: An example of a Rust program where borrowing is used

```rust
fn foo(x: &str) {
    println!("x␣is␣\'{x}\'");
    // ...
}

fn main() {
    let x = String::from("Hello,␣world!");
    foo(&x);
    println!("{x}"); // ‘x‘ was never moved, so this is fine!
}
```

At any one time, a value in Rust has one single owner. Other parts of the program can *borrow* a reference to the value to using the `&` operator. A Rust *reference* is similar to a C pointer, but with an important requirement: A Rust reference may *never* point to invalid memory, and will *always* point to a valid instance of the type of the reference. Ownership can also be *moved*. A value may not be used after it is moved, preventing a freed value to be accessed. When the owner of a value goes out of scope, the value is automatically dropped (freed).

In Listing 2.1, the `x` variable has ownership of the `String` value. When passed to the `free` function, the ownership is moved and the value is then immediately dropped, since the block of the `free` function is empty. However, when the `x` is attempted to be printed in the `println!` macro, it is attempted to be used after a move, which is not allowed by the Rust compiler.

Writing the same program in C is trivial (although odd), and will compile. One such program can be seen in Listing 2.2. Note that lines 12-13 in the C example corresponds to line 7 in the Rust example.

It is possible to write a program that performs the operation above in a memory safe way in Rust using references, as seen in Listing 2.3.

While these programs are simple, and the C program is especially poorly written, for larger programs it is not as trivial to keep track of pointers to values. To guarantee that they are never used after free, or containing other memory problems, is a challenge even for experienced C programmers. Rust provides this guarantee without any runtime penalties, since the checks are made at compile-time.

2.1.2 Unsafe Rust

> THE KNOWLEDGE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF UNLEASHING INDESCRIBABLE HORRORS THAT SHATTER YOUR PSYCHE AND SET YOUR MIND ADRIFT IN THE UNKNOWABLY INFINITE COSMOS.
>
> (Disclaimer at the start of *The Rustonomicon*, the book for Unsafe Rust)

While providing several benefits in regards to safety and correctness of Rust programs, the Rust compiler can for some applications be too restrictive. This is deliberate design decision, but does create a problem where Rust would not be able to solve some problems. The solution used in Rust can be seen as dividing the language in two parts: *Safe Rust* (used by default) and *Unsafe Rust* (denoted by the `unsafe` keyword) [2]. In this context, *safe* refers to the guarantees given by the Rust compiler for safe Rust such as type-safety and memory-safety. This means that there can be no dangling pointers, use-after-frees, or any other kind of undefined behavior in safe Rust [2].

Unsafe Rust is in many ways similar to Safe Rust, with some important additions often referred to as *unsafe superpowers*. These superpowers include the following [15]:

1. Dereference a raw (C-like) pointer

2. Call an unsafe function or method

3. Access or modify a mutable static variable

4. Implement an unsafe trait

5. Access fields of `unions`

An interesting aspect is that calling foreign functions via an FFI (Foreign Function Interface), such as C library functions, is always unsafe. This means that some calls to important system libraries can only be made in unsafe Rust.

It is also the case that some parts cannot be safe due to their nature; Code that relies on hardware interfaces may fail, although the code itself is correct. Listing 2.4 provides an example where Unsafe Rust is required, as it calls a native C function.

Listing 2.4: Rust code calling inheritly unsafe C functions using the `libc` package

```
 1 fn main() {
 2     let m = std::mem::size_of::<i64>();
 3
 4     // Call to unsafe function
 5     let p: *const i64 = unsafe { libc::malloc(m) } as *const i64;
 6
 7     // ‘p‘ is never checked against ‘libc::PT_NULL‘ to ensure
 8     // memory allocation succeeded
 9
10     // Unsafe dereference of raw pointer
11     println!("Found␣{}␣at␣allocated␣memory", unsafe { *p });
12
13     // ‘p‘ is never freed!
14 }
```

This example presents both benefits and dangers of Unsafe Rust; While C functions can easily be called using the FFI provided by the `libc` package, the code contains dangers. The call to `malloc` inside the `unsafe` block is non-problematic, however the handling of the resulting raw pointer is not. Not checking if the allocation succeeded results in a dangerous memory access at line 11, and `libc::free` is never called on the raw pointer.

A common Unsafe Rust design pattern is to wrap unsafe code in a safe API, to provide functionality not possible with standard Rust [2]. This is the case for several parts of the Rust standard library, such as `RefCell<T>`, which will guarantee Rust borrowing rules at runtime instead of at compile-time.

This wrapping of unsafe code in a safe API can lead to a problem that is referred to as *unsoundness*. Unsound code is not unsafe, but if a Rust library is not sound, it is possible to use it in such a way that it leads to, for example, memory unsafety. Say that an arbitrary C function `hates8` takes an integer as input. For all inputs except `8` it works fine, but if a user were to pass `8` to it, the function will immediately start doing all kinds of memory unsafe procedures. While it probably will not summon eldritch horrors the likes of which will drive any person to insanity, it may cause security problems. If a (safe) Rust function where to call `hates8` via an FFI and it does not ensure the input is not `8` in some way, the function is considered unsound. It presents itself as safe, but it is possible for a caller to reach memory unsafety while using it.

To avoid the invocation using the forbidden number, the library designer may use several different techniques, or just pass the danger to the library consumer by marking the function as `unsafe` and add a disclaimer in the documentation that `8` is not to be used as a parameter.

Unsafe functions are instead functions that require that the caller fulfills some requirements described by the documentation to be safe [15].

2.1.3 The Rust Ecosystem

The foundation of the Rust ecosystem is the `cargo` package manager. While it is possible to build Rust projects without `cargo`, such as using `make`, this thesis will only discuss projects using `cargo`. Together with the crates.io code registry, `cargo` provides software developers with an easy way to publish code to the community, and to use published code as dependencies. The `cargo` command-line interface provide Rust developers with built-in tools to interact with their code, including the Rust compiler `rustc`. To download dependencies, compile the package and link it, the user simply has to use the `cargo build` command.

We must here make a distinction we have so far avoided; *crates* and *packages*. *The Book* provides a definition of both which we will provide here, however the term package is used by our tool. A *crate* is the smallest amount of code that the Rust compiler considers at a time. A crate can either be a *library crate*, intended to be used by other crates, or a *binary crate*, a program that is executable. A *package* is a set of at most *one* library crate and any number of binary crates [15]. Since only library crates can be dependencies, dependencies in Rust are often called packages, even though the true dependency is the library crate in that package, which may or may not share the name of the package. To complicate the matter, it is also possible to split a Rust project into several packages, where one package contains only the binary crate, and one contains only the library crate. We have decided to use packages as our lowest point of interest, since there may be several crates available in the `src/` directory of a package, and a `Cargo.toml` file (see Listing 2.5) refers to a package (or workspace). While this may lose some granularity, and some confusion in the case where a library crate does not share the name of its package, we consider this description to be more clear. By convention it is unlikely that this will cause problems, but it may result in unexpected behavior and results when a package has an advanced configuration.

The Cargo Book[1] contains a great overview of `cargo` and the Rust ecosystem. We will provide a brief summary here to provide a background for the rest of the thesis.

Each package can have several versions, and versioning is made using semantic versioning [1]. Semantic versioning can be summarized as a scheme where versions are split into three parts; MAJOR.MINOR.PATCH, with the following definitions[2]:

- MAJOR versions are incremented on incompatible API changes
- MINOR versions are incremented on adding functionality in a backward compatible manner
- PATCH versions are incremented on backward compatible bug fixes

In a Rust package handled by *cargo*, code dependencies are listed in a `Cargo.toml` file. An example `Cargo.toml` file can be seen in Listing 2.5.The user may specify which version(s) of a dependency is to be used, and can define intervals (like *All MINOR versions above X.Y.Z*). The actual versions used when the package is first compiled will be written to a `Cargo.lock` file, so the same versions are reused as long as they fulfill the versions requirements defined in `Cargo.toml`. It is also possible to define dependencies with a specific path on the local filesystem, or to use a git repository (with an optional revision hash).

[1]https://web.archive.org/web/20230422160738/https://doc.rust-lang.org/cargo/

[2]https://web.archive.org/web/20230528123304/https://semver.org/

Listing 2.5: An example `Cargo.toml` file

```toml
[package]
authors = ["Emil Jonathan Eriksson <eje1999+cargo-indicate@gmail.com>"]
name = "cargo-indicate"
version = "0.2.0"
edition = "2021"
description = "Crate for running GraphQL queries on Rust dependency trees"
keywords = ["cargo", "cli", "search", "dependencies", "graphql"]
categories = ["command-line-utilities", "development-tools::cargo-plugins"]
readme = "../README.md"

[[bin]]
name = "cargo-indicate"
path = "src/main.rs"

[dependencies]
clap = { version = "4.1.4", features = ["wrap_help", "derive"] }
indicate = { path = "../indicate", version = "^0.2.0" }
serde = { version = "^1.0", features = ["derive"] }
serde_json = "1.0.93"

[dev-dependencies]
trycmd = "0.14.12"
test-case = "3.0.0"
```

When `cargo` builds a package locally, the source code of the dependencies are downloaded to the `$CARGO_HOME` directory on the users machine, by default `$HOME/.cargo`. While the Cargo Book notes that the internal structure of `$CARGO_HOME` is not stabilized, we note that `cargo` keeps a cache for each `git` repository and package index used here, making it able to easily access the source code when needed to build a package. This aspect, that `cargo` keeps local copies of the source code, is important for the thesis, as it ensures that we can access the source code of dependencies as easily as the source code of our target package.

It is not uncommon that Rust packages share repositories with other (related) Rust package; in those cases each crate have their own `Cargo.toml` file, but they may also share a *workspace* `Cargo.toml` file higher up in the directory tree. This makes it possible for crates to share `target/` directories, and already compiled dependencies. These crates may also use a `path` specifier for their dependencies, in which case they can refer to crates in the same repository or workspace, which we found several instances of in our data collection.

A Rust crate or its versions cannot be removed from crates.io once published with the exception of use deemed illegal or malicious by crates.io administrators[3]. To provide a way of marking a version containing known bugs or vulnerabilities, a package author can mark individual versions as *yanked* (marked that they should not be used). Yanked versions are only used if they already appear in the `Cargo.lock` file, but will not be selected by `cargo` otherwise.

We can also note an important functionality in Listing 2.5, namely the `features` key. The Cargo Book describes it as a mechanism to provide a form of conditional compilation and optional dependencies. A package author may define that some dependencies are only needed when some features are enabled, and also that some items in the code are enabled (or disabled) when a feature is enabled. An example of a function for which the implementation changes depending on a feature can be seen in Listing 2.6

In addition to the built-in functionality provided by `cargo`, users are free to write their own extensions, which is how the software was implemented in this thesis.

[3]https://web.archive.org/web/20230422160710/https://crates.io/policies

Listing 2.6: Example of a Rust program for which the implementation of `foo` changes depending if compiled with the `worldwide` feature or not (`rustc <file>` vs `rustc --cfg 'feature="worldwide"' <file>`)

```rust
#[cfg(feature = "worldwide")]
fn foo() {
    println!("Hello world!");
}

#[cfg(not(feature = "worldwide"))]
fn foo() {
    println!("Hej Sverige!");
}

fn main() {
    foo();
}
```

2.2 Motivation

This sections intends to provide motivation for the work, both from an industry standpoint and from an academic standpoint.

2.2.1 Rust at Volvo Cars

At Volvo Cars, Rust is increasingly being adopted as a programming language for embedded software development. The language has gained popularity in the organization, and users are often advocates of further adoption by others within the organization. Perceived benefits include the built-in ecosystem in `cargo`, the safety inherent in the language, as well as a rich type system allowing for faster development.

Open source software, as published on crates.io, is frequently used in development at Volvo Cars. While allowing faster development, and reuse of code for common tasks, this leads to problems regarding the quality, safety, and risk associated with using outside code. This problem is amplified by the dependencies listed in a Volvo Cars `Cargo.toml` file may itself depend on other crates. Internal processes aim to minimize these risks, but this does not mean that the practice is risk-free.

Given this dependency tree, it quickly becomes very hard, if not impossible, for a team of developers to manually verify for quality and correctness. For this reason, the need for a tool to automatically warn a developer of problems in their dependency tree becomes apparent. It would also be helpful for a developer to easier get insight in their dependency tree without doing a manual investigation of all metrics for all dependencies.

2.2.2 Basis for Further Research

The open source software community is ever evolving, producing more and more software of varying amounts of quality. While studies in this field continue to make interesting discoveries in the relationship between different data sources, these may often focus on smaller subsets of signals such as repository information or simply analysis of the code.

However, modern software development includes the production of much more information than just the code itself; The history of the software, the people creating it, the website hosting it, the historical issues are all pieces of information that may paint a bigger picture. Accessing this information in a way that benefits both developers and academics provide a challenge, that we aim to solve in part for the Rust ecosystem with the tool we provide with this thesis.

2.3 Related Work

This section provides examples of work in related domains, and especially Rust projects, that aims to solve the same or neighboring problems as presented in the motivation section. We will start by introducing the Google project deps.dev, followed by existing Rust tooling that we have found during our research. Finally we will present studies looking at similar areas of research.

2.3.1 deps.dev and the OpenSSF Scorecard

Open Source Insights, a service by Google, provides some of the functionality presented in this thesis. Its website deps.dev provides developers with the ability to search for any open source project in several ecosystems, including `cargo`. The service provides developers with a detailed dependency graph, as well as an *OpenSSF Scorecard*.

The OpenSSF Scorecard is a project[4] by the Open Source Security Foundation. The scorecard aims to provide a template for best security practices for open source projects. It currently lists 16 criteria, such as branch protection, CI-tests, if it is maintained (has recent commits) etc. It then aggregates this to a final score. Our tool can provide some of the information used to create the scorecard, but does not provide a score to the user. Instead, it provides more metrics that can then be evaluated according to the needs of the user. The OpenSSF scorecard can thus be a suitable tool for initial screening, while more in-depth analysis, including more Rust-specific metrics, of a project can be done with our tool.

While our tool contains overlap in what it can provide for the user, we believe that our tool provide novel functionality. While deps.dev and the OpenSSF scorecard provides information in a easy-to-use way, making the information accessible, it is limited in its functionality by focusing mainly on best practices found in a repository. We also note that our tool focuses entirely on Rust, while deps.dev supports multiple ecosystems. This provides us with the ability to take Rust-only parameters into account. We note that our tool is extensible, both in adding more data sources, but also because the user can write their own queries that can investigate new relationships, similar to how one would write queries to a database.

[4]http://web.archive.org/web/20230508021952/https://github.com/ossf/scorecard

2.3.2 Rust Software

There exist several Rust crates and packages intended to make it easier to evaluate the dependencies used by a project. Such work includes, but is not limited to:

- `cargo-geiger`, a tool for finding usage of unsafe Rust in a project, including how it uses unsafe Rust in its dependencies
- `cargo-audit` and the RustSec Advisory Database, which lists and audits known vulnerable crate versions
- `cargo-crev`, which allows for cryptographically verifiable code review system
- `cargo-vet`, which ensures third-party crates have been audited by a trusted source
- `cargo-deny`, which allows for listing blocked dependencies in a `deny.toml` file and linting the dependency graph, which is similar to the tool that is presented in this thesis

While these projects provide valuable information, and could be considered for any Rust project, they themselves does not provide a way to query the results, or combine the different metrics. We do note that some of these may be more easily integrated into a CI pipeline, such as using `cargo-deny` for preventing some known issues.

2.3.3 Studies

Software repositories stored on services like GitHub and GitLab provide much of the foundation for this report. One study by Schueller et al. used an approach collecting information similar to the tool we provide in this paper [21]. They found that in the Rust ecosystem, a vast majority of projects could be cloned from either GitHub or GitLab, with a majority being available on GitHub. They provide a data processing pipeline to collect data, process it, and then make it available in a database. The data includes information about the repositories, such as stars, package versions, pull requests, issues etc. The declared purpose of the database is to give a comprehensive view of several dimensions of the ecosystem.

While the processing of the data in this pipeline is more rigorous than the one provided by our tool, the presentation of data is not immediately accessible in the same way to Rust developers. Tools are provided to recreate the dataset, and one version of it is available on GitHub. We believe that our tool could use a more stable source of information for repositories in the future, and that a database with a rigorous data pipeline could be one such source. Nothing prevents such an extension, but the current implementation has benefits of being up-to-date and providing ease of use.

`GrimoireLab` is another project providing insights into multiple data sources, which include GitHub among others [11]. The data can be anything from tweets to answers on StackExchange sites. This data can then be used to create software development analytics dashboards and reports. Even with the capabilities of aggregating data from different sources, the problem of assessing FOSS (Free and Open Source Software) project health is very complex, and no set of metric has been found to reveal the health of a project. It should also be noted that our work focuses exclusively on the Rust ecosystem, with the added benefits of focusing on Rust-specific metrics (such as the usage of Unsafe Rust).

There has been several more works in the area of mining data from software repositories, and the International Conference on Mining Software Repositories[5] is a source of several articles.

Software Bills of Material is another aspect that is related to this thesis. SBoM can be described as a inventory of all components that make up a software project, and is described by the American Cybersecurity & Infrastructure Security Agency (CISA) as a key building block in software security and software supply chain management[6]. However, a recent review of the usage in practice [25], based on interviews and surveys of SBoM practitioners, reveals that the practice is still in early stages, and that while beneficial, there is need for standardization. The tooling is currently held down by a lack of maturity. While practice may vary between projects and organizations, it reveals underlying problems present in the practice. We do not claim that our tool provide a SBoM, especially not to be held to strict requirements, but the adaptive capabilities makes it easier for developers to create queries that takes all components of a software component into account.

In our analysis we look at several scalar metrics in the Rust ecosystem, to attempt to create a baseline for what users of our tool might expect to find. In [23], the authors attempt to find indicators of software vulnerabilities. Looking at complexity, code churn, and developer activity, the authors found that they could reduce the number of files and lines of code to inspect by significant amount for the Mozilla Firefox web browser and the Red Hat Enterprise Linux kernel. The authors used a binary classification technique, measuring how well different metrics managed to identify known vulnerable files, and found that several metrics that are also available using our tool (such as lines of code) are indications of vulnerabilities. We should note that the paper focuses on file detection in two projects, whereas we focus on detecting complete packages that are of interest (vulnerabilities or otherwise)

There have been several studies on the ability of Rust to prevent bugs. One such study [26] looked at all Rust CVEs (Common Vulnerabilities and Exposures) regarding memory-safety, and found that unsafe Rust was present in all but one of the bugs. This analysis focuses on code-level, which is not our goal, but the results provides support for our assumption that Rust unsafety is a signal worth investigating.

There has also been further work to understand how software developers use Rust, and in particular unsafe Rust. [10] looked at publicly available Rust code, and used the call graphs to determine how unsafe Rust was used by that code. It was found that less than 30% of Rust libraries use unsafe, but that unsafe usage was sometimes hidden in the call chain. [19] used another approach, and performed manual close inspection of 850 unsafe usages and 170 bugs in five open source Rust projects. It was found that unsafe code was unavoidable in many cases. No cases were found of memory-safety issues that did not also include unsafe code. [6] used a combination of automatic code inspection, intermediate representation inspection, type information provided by the Rust compiler, and manual inspection. The results agree with the other papers, and found unsafe usage to be a source of bugs, but also that sometimes API providers hide unsafe usage from the API consumers.

These studies support usage of unsafe Rust as an important factor in identifying memory-related issues with projects, which is a unique feature of Rust and something our tool sup-

[5]http://www.msrconf.org/

[6]https://web.archive.org/web/20230518045746/https://www.cisa.gov/sbom

ports.

While the Rust compiler provides static analysis of programs, several works have attempted to extend this verification with additional resources; [12] implements a static code checker using the Viper verification infrastructure developed at ETH; [17] provides a technique to mitigate the security threat of unsafe Rust by using a heap allocator separating the memory of safe and unsafe Rust; [16] provide and evaluate a bug detection framework for the Rust mid-level intermediate representation (MIR). These example papers all focus on a static code checking, and we believe our approach to be a complement to this focus on the code itself.

To summarize, there exists past and ongoing research into the field of identifying vulnerable Rust code. Several studies and projects exist to analyze Rust code statically. There exists studies in GitHub and GitLab usage, and also specific research into usage of these services for Rust packages. We consider our tool to provide a novel approach to provide this information with a unified interface, and with a combination of focus on stats about source code as well as information from other sources.

Chapter 3
Method

This chapter will start by providing a black-box description of the tool, `cargo-indicate`, and its underlying library `indicate`, that we created. We will then provide a more in-depth description of the inner workings and its underlying technologies. We will then present a way of collecting data using `cargo-indicate`, followed by the methods used to analyze the data to answer our research questions and evaluate the usefulness of our tool.

Some implementation details are omitted but are included in Appendix B.

3.1 `cargo-indicate` Overview

Listing 3.1: Invocation of `cargo-indicate` with a simple query; `cargo indicate` is the software, the `-q` flag accepts one or more queries, or the paths to files containing queries, `-p` is the target package name, and the path after `--` is where `cargo-indicate` should look for a Rust package

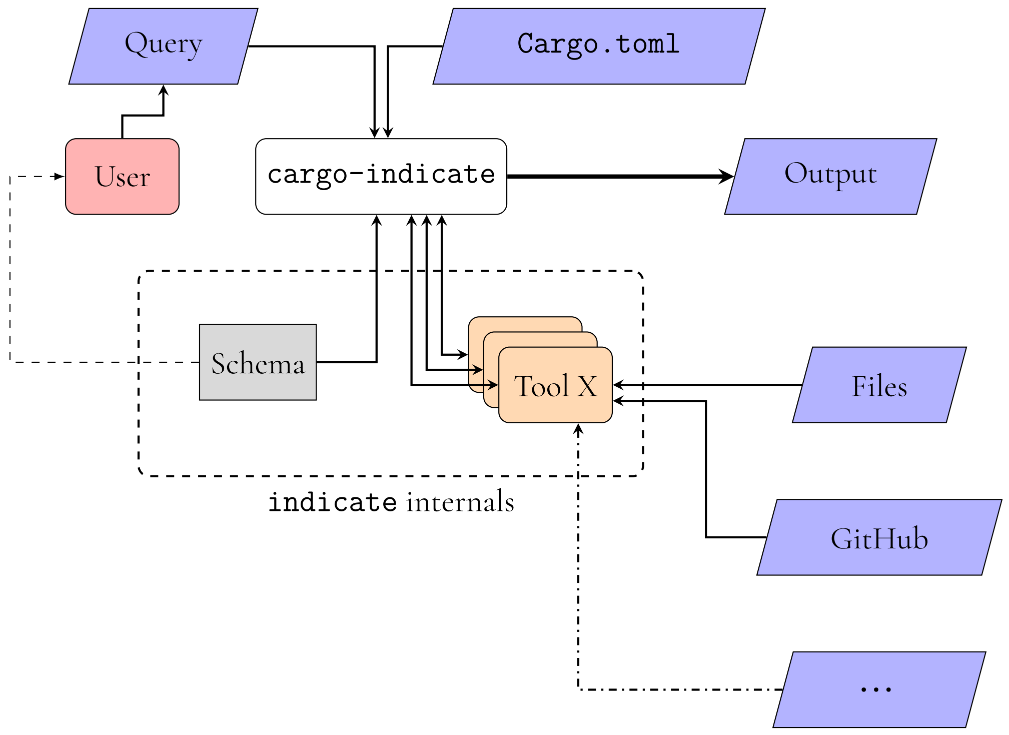
```
# With query inline
cargo indicate                              \
  -q '{ RootPackage { name @output } }'    \
  -p cargo-indicate                         \
  -- $SOME_DIR/

# With query in a file and more precise package path
cargo indicate -q query.graphql -- $SOME_DIR/cargo-indicate/
```

We built `cargo-indicate`, a `cargo` add-on to provide a way to write queries on a local Rust package and its dependencies. After being installed, it can easily be invoked by calling `cargo indicate`. A high-level overview of `cargo-indicate` can be seen in Figure 3.1.

`cargo-indicate` uses a subset of GraphQL to make it possible to execute queries on a package and its dependency tree. The `indicate` library provides a schema, accessible

Figure 3.1: A high-level overview of `cargo-indicate`

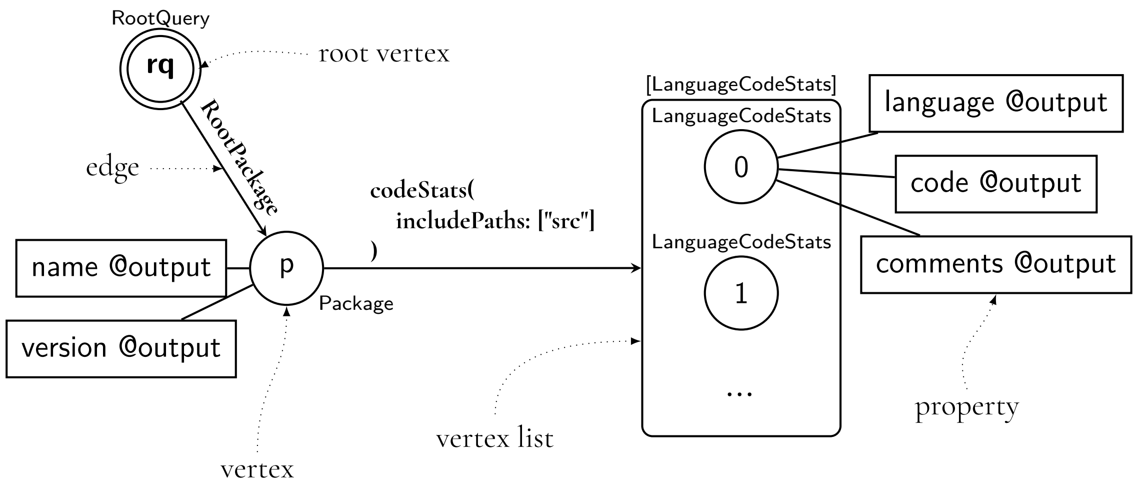


using the `--show-schema` flag (see Appendix C). By providing `cargo-indicate` with a query and the path to a Rust package, it can resolve the query across the dependency tree of the target package. `cargo-indicate` then uses underlying tools, one for each signal, to sample data. Listing 3.2 shows a simple query counting the number of lines in a package. Figure 3.2 provides a graphical representation. In this figure, circles are vertices, arrows are edges, rectangles are properties, and vertices enclosed in a box are a list of vertices. In Rust code, the vertices in this query are variants of a `Vertex` enum with a counted reference to a struct with fields corresponding to the properties. The edges are more abstract, and we refer to Appendix B and the code itself for more implementation details. Note that the root of the query is `RootPackage`, which means that the query will start at the root of the dependency tree.

Listing 3.2: An example `cargo-indicate` query counting the number of lines of code and comments in the `src/` directory of a package

```
{
    RootPackage {
        name @output
        version @output
        codeStats(includedPaths: ["src"]) {
            language @output
            code @output
            comments @output
        }
    }
}
```

Figure 3.2: A graphical representation of the query in Listing 3.2 (duplicate properties omitted)



The results are JSON lists that can be printed to standard output or saved in files. Fields marked with the `@output` query directive are included in this output, and can be renamed by using `@output(name: "foo")`. By checking if the results of a query are an empty list (`[]`), filters can be created. This makes it possible to disallow any characteristics that are undesirable. A sample output for Listing 3.2 can be seen in Listing 3.3.

Listing 3.3: Sample result for the `indicate` package for the query in Listing 3.2

```
[
  {
    "code": 187,
    "comments": 47,
    "language": "GraphQL",
    "name": "indicate",
    "version": "0.1.0"
  },
  {
    "code": 2661,
    "comments": 120,
    "language": "Rust",
    "name": "indicate",
    "version": "0.1.0"
  }
]
```

Listing 3.4: An example `cargo-indicate` query counting the number of lines of code and comments in the `src/` directory of a package, filtering out results with less than the argument `minLoc`.

```
{
    RootPackage {
        name @output
        version @output
        codeStats(includedPaths: ["src"]) {
            language @output
            code @output @filter(op: ">=", value: ["$minLoc"])
            comments @output
        }
    }
}
```

Listing 3.5: Sample result for the `indicate` package for the query in Listing 3.4, with the additional flag `--args '{ "minLoc": 500 }'`

```
[
  {
    "code": 2661,
    "comments": 120,
    "language": "Rust",
    "name": "indicate",
    "version": "0.1.0"
  }
]
```

We can modify this query to take arguments. We can for example only output languages with more than `minLoc` lines of code, see Listing 3.4.

Arguments can be set using the `--args` flag, such as `--args '{ "minLoc": 500 }'`. This would result in the result in Listing 3.5.

It is also possible to bundle arguments and queries in a file of a supported format (such as `.ron`[1] files), which is supported by the `-Q/---query-with-args` flag.

All queries can be made on the target package or its dependencies, or a combination of both. There are some exceptions to some signals, and these are presented together with the signals themselves.

Queries are very flexible, and have access to filters, recursion, folding (converting results of some depth n to a list of n elements), optionals, tags, and renaming outputs. Listing 3.6 shows an example query listing all dependencies using the semver trick[2]. The semver trick is a workaround for an issue that can occur in the Rust ecosystem. Rust considers a type from two different minor semantic versions to be different, even if the type has not changed between those versions (so type `A` in `liba 0.2` is not the same as type `A` in `liba 0.3`). When a large number of Rust projects then reuse type `A` in their public API, the version of `liba` those projects use becomes important, and upgrades a difficult coordination problem between library developers. The solution is to release a patch version of `liba`, `liba 0.2.1`,

[1]Rusty Object Notation

[2]http://web.archive.org/web/20230320044313/https://github.com/dtolnay/semver-trick/

that depends on a future version of itself (`liba 0.3`). The unchanged types can then be re-exported from `0.3` to `0.2.1`, so any library using `A` from `liba 0.2.1` in their public API actually use `A` from `liba 0.3`.

Listing 3.6: An example `cargo-indicate` query listing all dependencies using the semver trick, by checking if a dependency depends on itself (a true check would also ensure that the version it depends on is newer). Note that bound values using the `@tag` directive are prefixed with `%`

```
{
    Dependencies(includeRoot: false) {
        name @tag(name: "parentName") @output
        dependencies {
            name @filter(op: "=", value: ["%parentName"])
        }
    }
}
```

Figure 3.3: A graphical representation of the query in Listing 3.6

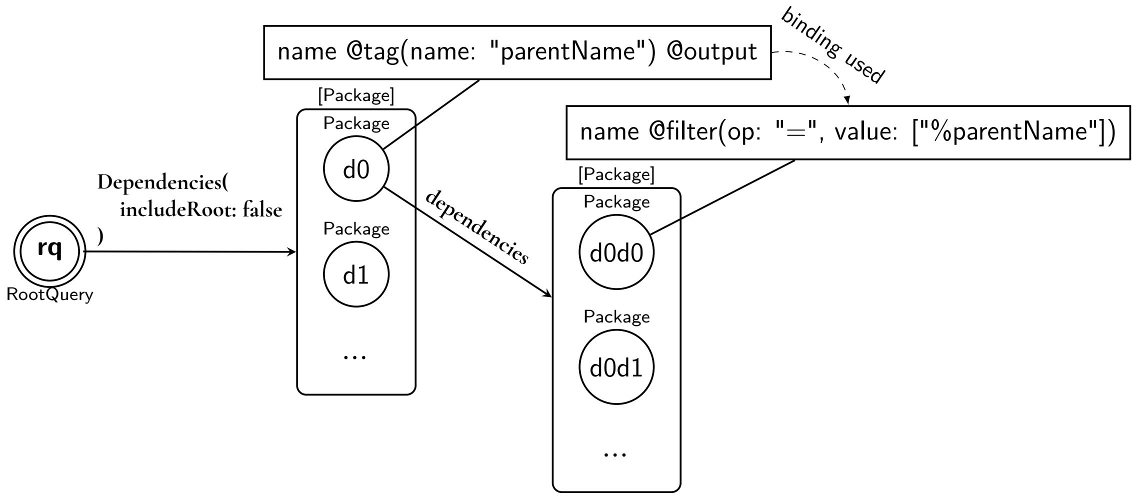


Figure 3.3 provide an overview of the query execution; The query engine start at an "invisible" `RootQuery` vertex, and then follow the `Dependencies` edge to retrieve a list of all dependencies of a project, not including the root package according to the edge parameter. The dependencies are a list of `Package` vertices. The name of each package is bound to `"parentName"` using the `@tag` query directive. The engine then follows the `dependencies` edge of each package, to access the list of direct dependencies for that package. It then goes through the list of packages, comparing the name of each direct dependency to the value bound to `"parentName"` (the `%` prefix is for tagged values). The filter ensures that only those packages (`d0`, `d1`, ...) that have at least one dependency (`d0d0`, `d0d1`, ...) with the same name as that package are included in the output.

`cargo-indicate` resolves queries lazily, that is, results are only fetched and computed when required by the query. This has several benefits for the user:

1. Only request third-party APIs when required, especially important due to rate limits

2. Adding more signals do not impact old queries

3. Some parts of the schema can effectively be ignored by the user, without any downsides. This is especially useful if one signal requires API keys or special configurations; the user does not have to configure what they do not use

3.2 An In-Depth Look at `cargo-indicate`

This section will provide a more in-depth look at how `cargo-indicate` queries the dependency tree of a Rust project, and aims to provide insights in its strengths and weaknesses. We will introduce the query engine behind `cargo-indicate`, `trustfall`, and describe how it is used to query our signals. We will end with some implementation details of the `indicate` library, describing how it may further developed to include more signals.

`cargo-indicate`

An in-depth overview of `cargo-indicate` is presented in Figure 3.4. We will not discuss the details of command parsing, which is handled by the excellent `clap` package.

We note that the user input is parsed into four distinct input types, namely:

1. One or more queries, to be answered by `cargo-indicate`

2. Optional sets of arguments to the provided queries

3. Optional options, which can be related to how `cargo-indicate` uses tools etc.

4. The package path, i.e. the path that is used to resolve the manifest file (`Cargo.toml`) of the target root package

`indicate` uses the builder pattern[3] to make it easy to control how the `IndicateAdapter` is built, and thus how the tools are used. The only requirements to create an `IndicateAdapter` instance is a valid path to a `Cargo.toml` file. It is then this `IndicateAdapter` instance that is used by the `trustfall` engine, presented in Section 3.2.1, to resolve the queries.

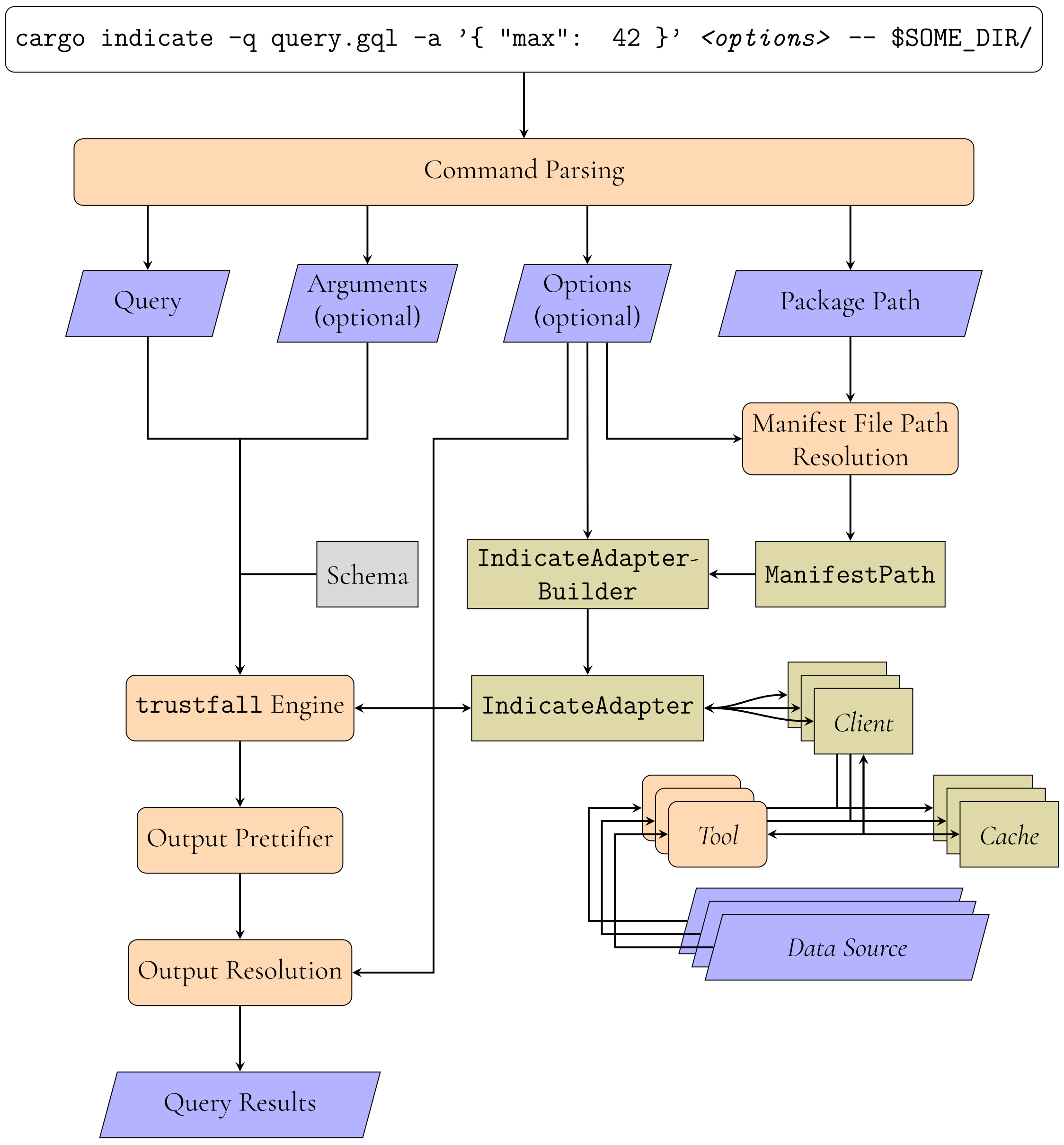
While several tools, such as API clients and `cargo-geiger`, are used by `cargo-indicate`, these are accessed through so-called *clients*. These are wrapper types to provide an interface suitable for `IndicateAdapter`, and to add caching functionality. The implementation details differ, and some clients do little more than to add a cache and expose underlying methods in the tool. Note that not all clients have caches, and that a cache might be both in-memory and in storage (to be used across executions).

`IndicateAdapter` does not create these clients from the onset. Since a set of queries to resolve does not necessarily need the usage of all clients, initialization is deemed unecessary. Thus, the `OnceCell` package[4] is used to provide lazy initialization of clients and their cache. This is made possible by the `trustfall` engine, which does not request edges and vertices not required by the query.

[3]https://web.archive.org/web/20230406145753/https://rust-unofficial.github.io/patterns/patterns/creational/builder.html

[4]To be stabilized in the Rust standard library

Figure 3.4: An in-depth overview of how `cargo-indicate` resolves a query



3.2.1 The Trustfall Project

`trustfall` is a query language and engine written in Rust under an Apache-2.0 license by Predrag Gruevski[5]. The project is used as the backend for the `cargo-semver-checks` project, which ensures that semantic versions defined in `Cargo.toml` matches the changes in the project.

This is done by running `trustfall` queries on the generated `rustdoc` documentation; A set of queries are run to compare the old and the new documentation, and if there are differences that are known to require a semver update the developer is informed.

The underlying goal of `trustfall` is to combine perceived benefits of the GraphQL query language, and SQL. It is described as a way to combine data sources such as third-party APIs, local files, and other databases.

While the `trustfall` project is still in early development, it does provide a framework from which it is possible to reach the requirements of the software developed in this thesis.

3.2.2 Trustfall Data Structure

In `trustfall` (and GraphQL), one may think about data as a graph. Queries start at a finite set of entry points (root queries); These entry points are a set of edges leading to vertices in the schema. From these initial vertices a query may request the scalar properties (such as `id` and `name` of a vertex of an arbitrary `User` type), or an edge leading to another vertex. An edge may take some parameters, and leads to another vertex of some type with its own scalar properties and edges.

`trustfall` requires three things to execute a query:

1. A *schema* to describe how data is related and can be queried
2. An *adapter* to resolve the fields described by the schema
3. The *query* itself

At the time of writing, both schemas and queries are written in a subset of GraphQL[6]. A very simple GraphQL schema can be seen in Listing 3.7.

Listing 3.7: A very simple GraphQL schema

```
type RootQuery {
    user(name: String): User
}

type User {
    name: String!
    age: Int!
}
```

This schema can be interpreted as the following, where (*RootQuery* has a special meaning in `trustfall` schemas): *There is one entry point to the schema (user), this entry point takes an optional argument (name) and will return a nullable object of type user with two properties (name and*

[5]https://web.archive.org/web/20230425042416/https://github.com/obi1kenobi/trustfall

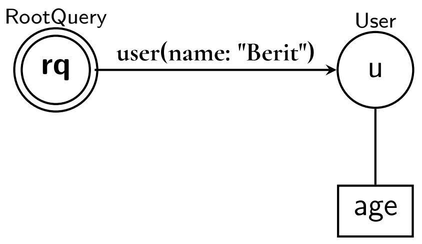
[6]https://graphql.org/

age). Note that a `!` suffix indicates that a property is non-nullable. To retrieve the age of all users with the name *"Berit"*, the query in Listing 3.8 could be used.

Listing 3.8: A query for the age of all users with the name *"Berit"*

```
{
    user(name: "Berit") {
        age
    }
}
```

Figure 3.5: A graphical representation of the query in Listing 3.8



The `trustfall` schema and query would be equivalent, with the exception that `age` would be annotated with the `@output` query directive to be part of the output.

Query directives such as `@output` provides a lot of the power of `trustfall` and are built into the query engine. At the time of writing, using `trustfall 0.4`, the following query directives are available (these are also defined in the `indicate` schema in Appendix C):

1. `@filter`, for filtering on a property using a set of predefined operations such as `>`, `=`, `has_substring` etc.
2. `@tag`, for binding a property to a name, to be reused later in the query
3. `@output`, for making a property part of the output
4. `@optional`, for making a property optional
5. `@recurse`, for recursing up to a `depth` number of times along an edge
6. `@fold`, for folding all outputs from a scope into parallel lists
7. `@transform`, for transforming, like counting the members of a `@fold`

These query directives are a current focus of development for `trustfall`, and further updates will directly benefit `cargo-indicate` and the queries that can be written.

While a GraphQL API would usually provide a resolver for its schema as part of an API, connected to a single database (SQL or otherwise), `trustfall` is explicitly developed to provide a way to query across data sources (so that while the `age` property may come from a file, another property may come from an external API).

This is done using the `BasicAdapter` trait[7], providing four methods to be implemented:

[7]Rust traits are similar to a Java interface or a Haskell typeclass

1. `resolve_starting_vertices`
2. `resolve_property`
3. `resolve_neighbours`
4. `resolve_coercion`

All of these operate on iterators of a associated type; `Vertex`. This is a Rust type that in some way (decided by the user of the `trustfall` API) can represent all the different vertices as described in the schema. Since Rust iterators are lazily evaluated, a neighbor or property is only evaluated when explicitly requested by the `trustfall` engine as long as `collect` is not called on it (a Rust standard library method to force the evaluation of an iterator).

The methods to be implemented for `BasicAdapter` provide a way of mapping the names of vertices, their edges, and their properties to Rust values and functions. Each function takes string parameters that make it possible for Trustfall to traverse the graph. This requires the programmer to ensure that all possible combinations are handled by the methods.

The implementation of `BasicAdapter` is described further in Appendix B (Section B.1).

3.2.3 Signals as Vertices

The `trustfall` graph approach allows for an extensible amount of signals, and also allows for handling dependencies in a flexible way.

At the core of the `indicate` schema is the `Package` type. This type represents a singular package as published on crates.io, and may represent both a root package (the package being queried), or any of its dependencies. Using the properties and edges of this vertex, it is possible to access information and signals as they are required.

The entry point to the graph is provided by a root package. The dependency tree of this package, along with scalar properties about it and its dependencies, are provided with the help of the `cargo_metadata` crate. It provides a programmatic interface to interact with a Rust package.

By providing the `Package` type with a `dependencies: [Package!]!` edge (non-nullable list of non-nullable packages), queries can be run across the whole dependency tree. The `trustfall` `@recurse` query directive allows for recursing the operation, although currently limited by the requirement of a defined `depth`.

As more signals are added, they can simply be defined as new properties and edges on the `Package` type. This means as long as these are only ever added, all new updates to the tool will be fully backwards compatible with old queries, as old queries will simply use a strict subset of the new schema.

New signals implementation details may be vastly different (such as calling a third-party API, parsing code, or querying a database), but as long as they provide an arm in the `BasicAdapter` they can be resolved by the engine.

The general implementation of a signal follows some steps in the `indicate` library:

1. Updating the `schema.trustfall.graphql` to include the new edges and properties
2. Adding one or more new variants in the `indicate::vertex::Vertex` enum, variants either holding information directly or holding data that may retrieve information

3. Depending on the signal, implementing a new client. A client here simply describes a Rust type that can sample information from some signal, providing functionality such as caching

4. Adding the new arms in the methods of the `IndicateAdapter` implementation of `BasicAdapter`

5. New test queries to ensure the correct implementation

Since signals vary, `cargo-indicate` does not provide any traits that describe common client behavior. Further development of `cargo-indicate` could possibly include a standardized implementation of the adapter-client relationship, based on a design evaluation of the existing implementations, and performance profiling.

Further implementation details for the adapter-client relationship can be found in Appendix B (Section B.2).

3.3 Selecting Signals

Due to the nature of this thesis, it is not possible to say which signals have the highest impact, and which ones contain little information, from the start. This does not mean that we have to approach the issue of selecting signals completely blind though; There have been studies made on Rust and Rust bugs [6][19][26], as well as for other programming languages [23], and thus some indication of what might be significant.

The rest of this section will list the signals implemented by `indicate`, with a motivation for their initial selection, the underlying tool(s), possible quirks, and any limitations. The validity of data provided by them will be evaluated later.

To easier categorize signals, and to be able to reason about their correlation with each other, we will here present some behavior found in the presented signals.

1. *Code-dependant*; Signals that depend on the source code of a project to be sampled

2. *Version-dependant*; Signals that provide granularity only on the semantic version level

3. *Snapshot*; Signals that cannot be guaranteed to be reproducible, i.e. might differ between samples, since they only provide point-in-time data, with no way of checking earlier values

4. *Historical*; Signals that provide historical data when sampled

The signals are summarized in Table 3.1, and are further expanded upon in the following sections.

We refer to Appendix C for details about how the signals appear in the schema.

Table 3.1: Summary of the signals collected by `cargo-indicate`

Signal	Tool	Sample Scalar Values
The Dependency Tree	`cargo_metadata`	Number of dependencies etc.
Unsafe Rust	`cargo-geiger`	Lines of unsafe expressions, unsafe trait implementations etc.
Code Stats	`tokei`	Lines of code, lines of comments, comments to code ratio etc.
GitHub	`octorust`	Stars, amount of followers of repository owner etc.
RustSec Advisory Database	`rustsec`	Number of different type of advisories etc.
crates.io	`crates_io_api`	Number of downloads, amount of yanked versions etc.

3.3.1 The Dependency Tree

Listing 3.9: The dependency tree of `serde_json` v1.0.96 created using `cargo-tree` (only normal dependencies included)

```
serde_json v1.0.96
|-- itoa v1.0.6
|-- ryu v1.0.13
`-- serde v1.0.163
    `-- serde_derive v1.0.163 (proc-macro)
        |-- proc-macro2 v1.0.59
        |   `-- unicode-ident v1.0.9
        |-- quote v1.0.28
        |   `-- proc-macro2 v1.0.59
        |       `-- unicode-ident v1.0.9
        `-- syn v2.0.18
            |-- proc-macro2 v1.0.59
            |   `-- unicode-ident v1.0.9
            |-- quote v1.0.28
            |   `-- proc-macro2 v1.0.59
            |       `-- unicode-ident v1.0.9
            `-- unicode-ident v1.0.9
```

A projects *dependency tree* is the tree of all the projects dependencies, and the dependencies of those dependencies, and so forth. An example dependency tree can be seen in Listing 3.9, where we can see that some dependencies like `unicode-ident` can appear multiple times in the same tree.

When describing this tree, the *direct dependencies* of a Rust package are the dependencies listed under `[dependencies]` in its manifest file (`Cargo.toml`). In Listing 3.9 `itoa`, `ryu`, and `serde` are direct dependencies of `serde_json`. While other dependency types, like development dependencies, are also direct dependencies, `cargo-indicate` does not include them. Further development may see a flag to change this behavior though.

The other type of dependencies, *transitive dependencies*, are the dependencies of a package that are not explicitly listed under `[dependencies]`. These are the dependencies of dependencies, dependencies of those and so forth.

Note that a transitive dependency can also be a direct dependency, if it appears both in the manifest file and lower in the dependency tree. In this case, `cargo-indicate` considers it to be both.

While providing a backbone for further queries, as each node in this tree is a `Package` and can be queried as such, the dependency tree may also contain information on its own. For example, the amount of direct dependencies might suggest how open the author is to include third party code, while the amount of transitive code perhaps does not (as it is not under direct control).

As stated earlier, `cargo_metadata` is used to gain programmatic access to the dependency tree. In `indicate`, the `IndicateAdapter` struct uses types from this crate to identify and handle packages.

3.3.2 Unsafe Rust

Unsafe Rust has been found to be a significant indicator for bugs [19], and also that unsafe Rust usage is widespread [6]. Due to these factors, unsafe Rust usage is very likely to contain information valuable for our purposes.

The Rust `cargo-geiger` crate provides a way to check how much of the binary of a Rust project is unsafe Rust, including calls to dependencies. This approach is not perfect, and the `cargo-geiger` repository[8] contains issues where `cargo-geiger` is unable to fully describe the unsafe usage. We found false positives where `cargo-geiger` included unsafe code that should have been excluded in the analysis, since conditional compilation via Rust *features* should have excluded the code. We also found instances where Rust procedural macros (a form of preprocessing) impacted the analysis.

While not providing a stable interface, `cargo-geiger` provides the option to output data in a JSON format, which can then be easily parsed to Rust data types using the `serde_json` crate. The implementation of this in `indicate` relies on replicating the structure of JSON data in Rust structs, and then providing methods to those data types for computing data such as percentage of code compiled being unsafe. One important aspect here is that `cargo-geiger` may, for a number of reasons, fail to resolve the unsafe usage for some package in the query. By examining the logs for our data collection script, we noted that this often were the case when some dependencies were only used for the Microsoft Windows OS. In this case, the edge for geiger unsafety will simply be `null`.

The implementation of geiger unsafety in the `cargo-indicate` comes with a downside; It is very possible to write queries retrieving information that simply holds no value. For example, it is possible to write a query that looks at the unused code, and how much of that code is unsafe out of all unused code. One such query can be seen in Listing 3.10

This may provide *some* information (such as in a context where a developer has decided

[8]https://github.com/rust-secure-code/cargo-geiger

what *not* to use), but a better alternative is to see what is actually used, or how unsafe the dependencies are in total.

Another important aspect is what we mean by `used`; in the `cargo-indicate` schema, it means used in the context of the *root crate*. This means that running the same query lower in the dependency tree will represent something different. While being a step away from the requirement that all queries can be run anywhere in the dependency tree, it follows the underlying logic of `cargo-geiger`.

Listing 3.10: A `cargo-indicate` query retrieving the percentage unsafe code not used, out of all unused code

```
{
  RootPackage {
    geiger {
      unused {
        # total = unsafe functions + unsafe traits + ...
        total {
          percentageUnsafe @output
        }
      }
    }
  }
}
```

3.3.3 Code Stats

Code stats is here defined as statistics about some codebase, like number of lines of code, lines of comments, etc. The `tokei` crate provides an API to quickly access this information.

It is not guaranteed that stats like these do provide insights in a package, however large amounts of uncommented code is harder to maintain than small packages with a lot of comments. On the other hand, very small dependencies could likely be implemented directly.

In [23], several such metrics were investigated, and it was found that code line count was significant in detecting files with vulnerabilities, while comment density was not.

To be able to calculate this information, `tokei` needs access to the source code. Since all source code for dependencies is downloaded by `cargo` and stored locally in a registry directory, accessing the source code in the adapter can simply be done with the source path provided by `cargo_metadata`.

3.3.4 GitHub

GitHub is a popular way of hosting Rust packages, and provides an API that can be used to retrieve information about a repository, or its owner. We use the `octorust` crate to access it. It is possible to gain insights into several metrics provided by GitHub, such as number of (git) forks, number of stars (akin to social media likes or favorites), if the repository is marked as archived or not, etc.

The motivation for these data points is as an indication of popularity, which in turn may increase the scrutiny of the code. This follows Linus's law; *Given enough eyeballs, all bugs are* *shallow* [20], assuming GitHub metrics would provide insight in the amount of eyeballs, i.e. the amount of developers investigating a project, finding issues, and fixing them.

In fact, studies on the code review and decision making structures of GitHub repositories have revealed significant differences in behavior between groups of developers, that may in turn be important when selecting which FOSS projects that are likely to be sustainable over an extended period of time [27].

Contrary to information from crates.io, metrics from GitHub generally require more conscious decisions from a user; while downloads in crates.io count even those done without the user knowing (as when downloading dependencies), GitHub stars for example require a form of effort to produce.

According to [7], stars are considered by practitioners as the most useful measure of popularity on GitHub, ahead of forks and watchers. The authors selected a sample of 400 Stack Overflow users, and e-mailed them asking them to rate the three metrics according to how useful they are to assess the popularity of a GitHub project. However, this study also revealed that stars may favour projects with larger marketing campaigns, than projects with higher quality. It also revealed that the starring is done mainly to show appreciation to the projects (52.5%), to bookmark the projects (51.1%), and because they used the projects (36.7%). The same study also found that a majority of developers (73.0%) considered the number of stars before using or contributing to a project. Among these, 29.3% also considers other factors such as code quality, license, and documentation.

On the other hand, using metrics from GitHub requires both that the API returns reliable information, and that the client used collects this information accurately. Perhaps most pressing is that even if the data is collected and correct, it may still not be useful to us as the data might not provide any new insights about project quality or health. One study found several perils with mining GitHub data, many relating to the nature of some projects [14]. While we avoid some perils by focusing on popular Rust projects, some are harder to avoid, such as patterns in commits and the project structure.

The initial signal includes only information about the repository and the owner, and a more developed signal may include detailed information about issues and pull requests. Due to the extensible nature of `cargo-indicate`, this should be considered an easy target for development.

3.3.5 RustSec Advisory Database

The RustSec Advisory Database is a repository of security advisories against Rust crates published on crates.io[3]. It aims to provide a way for the Rust community to report advisories for crates in a centralized location, using GitHub pull requests as a way of reporting advisories.

Using a client provided by RustSec, `rustsec`, it is possible to query this database using several parameters, such as the severity of the vulnerability, the versions affected, etc.

While `advisory-db` at face value provides a lot of information, many of the fields provided are nullable. This means that we cannot rely on always being able to compare advisories using all available fields. Severity has also historically been a measure that is hard to keep objective, and sources of advisory information may conflict in their rating [9].

One must also consider one major issues with advisories; The creators of code that are very likely to contain future advisories are perhaps the least likely to report it. While others may choose to do so, severe vulnerabilities may go unreported for some packages while other developers may report even the most miniscule issues. This is a similar behavior that can become a problem with yanked crate versions, as we will see later.

Neuhaus et al. [18] showed that past reported exploits may not even correlate with future issues by analyzing the Mozilla vulnerability history. While interesting, it would still be interesting to see how advisory reporting correlate to other data points.

3.3.6 crates.io

The Rust package registry itself provides an API with information about crates, such as information about the number of downloads for a specific crate. We use the `crates_io_api` crate to access it.

This information can in some way provide insight into the popularity of a project. Here it is important to remember that libraries that often appear lower in a dependency tree, further from the root, are likely to have more downloads. If they provide some core functionality, or utility that is useful, they are more likely to have more downloads as transient dependencies than conscious downloads. Nonetheless, downloads provides a real insight in the usage of a project.

One caveat is that there may exist alternative package registries with other stats. It is possible to provide a registry that `cargo` can interface with, and this is described in the Cargo Book. It may be the case that a company host a private copy of the crates.io registry for any number of reasons, such as not wanting their package usage to be public. It is a possibility that such usage skews the data, but we do not take it into account in this thesis.

3.3.7 Signal Characteristics

In the table below, the characteristics of the signals are listed. Note that while it is possible to create historical data for some signal (with multiple queries, against different targets etc.) historical in this context means if the sample is a series of historical data points.

Signal	Code-dependant	Version-dependant	Snapshot	Historical
Dependency Tree	For root package	For dependencies	Yes	No
Unsafe Rust	For root package	For dependencies	Yes	No
Code Stats	For root package	For dependencies	Yes	No
GitHub	No	No	Yes	No
RustSec Advisory DB	No	Yes	No	Yes
crates.io	No	Yes	Yes	No

An important point to note here is the time discrepancy between signals that provide at-best version granularity (crates.io and RustSec Advisory Database), and those that can instead look directly at the code. While for example Code Stats can be checked on any source code, only download stats for specific versions (or sums of them) can be checked with crates.io. This creates a discrepancy when the time between a version release and the code being queried. While one could rely on only the latest releases from crates.io, this would still create a discrepancy as the GitHub signal always provides only the current data on the master branch, which may not be representative of the latest released version.

3.4 Data Collection

While the project structure set out in Sections 3.1 and 3.2 provides a framework for running queries on Rust projects, it does not provide any guidance on *what* queries to use, or how they are to be interpreted.

Our goal is to explore if there exist trends in the Rust ecosystem that can be applied when writing queries or filters using `cargo-indicate`. Since queries being run on one package can be reused on the dependencies of another, we can attempt to set a baseline for what is to be expected by running queries on popular Rust packages.

This section sets out to present a semi-automated environment that uses the CLI developed to run different queries on open source Rust projects, and to save the results as a form of baseline for further evaluation. We will first present the selection process for target Rust projects, and then how we ran queries on these projects.

3.4.1 Package Selection

We explored two ways of selecting target packages to be used in the data analysis; one manual and one semi-automated. We found that manual selection process removes some of the identified issues with the semi-automated approach, but had other drawbacks.

Initially, manual selection from crates.io was used to manually find target crates. crates.io provides functionality for sorting crates by recent downloads, all downloads, keywords, and categories. As a result, crates with a lot of recent downloads in different keywords were added to the package list, and their selection criteria was logged.

However, as the initial data from these crates was analyzed, we concluded that it would be preferable to scale up the data collection to provide a larger basis for further analysis. While manually selecting packages from crates.io provides the possibility of filtering some crates (such as only including a library and not a CLI using that library), it is a very slow process.

Thus, we created a program that relies on the crates.io public API; this in turn relies heavily on the Rust `consecrates` crate, allowing a simple interface for querying the most popular categories as listed on crates.io

This revealed a classification difficulty with Rust packages; authors are allowed to list their packages with several different categories and keywords in the `Cargo.toml` file of their project, meaning that it becomes entirely subjective on how crates are categorized. This could be avoided when using a manual package list to collect data, as we could be more consistent. However, we decided to instead rely on the self-reported categories and keywords of Rust packages, and these two properties were simply added to the `trustfall` schema to be part of the query results. We chose 28 categories from the list of 52 possible categories on crates.io categories[9] at random, using the `consecrates` built-in selection of categories.

Using the `consecrates` crate, the top[10] 100 crates of 28 categories were written to a new package list, and the duplicate entries were removed (going by name, as names are unique on crates.io).

To also include the most popular crates, the script also ensured to include the 500 most popular crate by downloads, both with regards to recent downloads and downloads for all

[9]https://web.archive.org/web/20230323232002/https://crates.io/categories

[10]Top by amount of recent downloads

time. Duplicate entries, including duplicates from the top-by-category, was removed.

It should be noted that the usage of downloads as a metric of popularity may favor libraries, as these are downloaded each time they are added as dependencies. Binary crates that are used as standalone tools may only be downloaded when used directly.

There is also the downside that some data may be queried twice. It is common for Rust projects to be divided across multiple packages. This is necessary in some cases, as when providing procedural macros, as Rust currently does not support them to be defined in the same crate as they are used. This means that the hypothetical crates `A` and `A_macros` both are included in the data collection (after all, if `A_macros` is needed to build `A`, it must have *at least* the same amount of downloads as `A`). In some cases, such as when `A` relies heavily on `A_lib` to provide its logic, the result may be almost duplicate, since they share many characteristics. They might even be hosted in the same repository. In our data collection, we found several such cases. Out of 2435 analyzed packages, 692 or 28.4% did not have a repository that was unique in the data set.

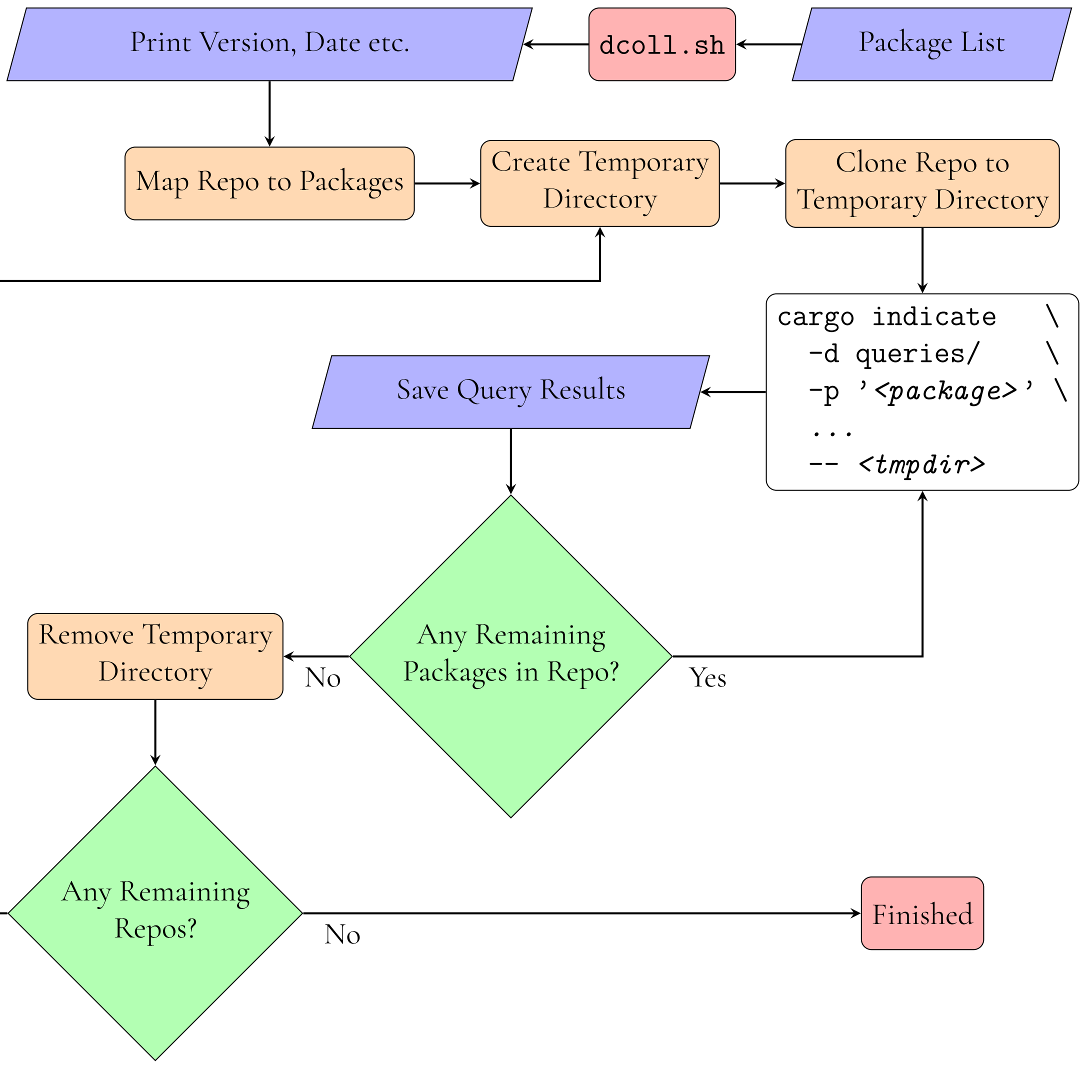
3.4.2 Automated Querying

Since some signals, such as `cargo-geiger`, require access to the source code locally, and since the CLI tool is intended to be run locally, a simple test setup was constructed where the latest version of a Rust crate repository was cloned using `git` into a temporary directory, and then had a set of queries run on it in series. The script executing these functions is called `dcoll.sh`. This script took a file with a list of packages and their repository links as an input. The script was executed in a instance provided by a cloud service provider over the course of several hours. This allowed for some expensive queries to be run, with the possibility of waiting for third-party API quotas if they were reached (such as the GitHub 1000 requests/hour quota). The workflow of `dcoll.sh` is described in Figure 3.6

To find a package in a repository required a special flag for `cargo-indicate`, `--package`, which ensured that the name listed in the initial found `Cargo.toml` file matched the package listed in the package list, and if not, recursively searched the directory for a `Cargo.toml` file matching that package name.

Due to the nature of the data collection procedure, only the latest version as available in a repository will be the basis for signal sampling. In our case, the information was collected on 2023-04-21. This is only significant for the *code-dependent* signals, i.e. unsafe Rust usage and code stats. This can in some cases lead to misleading data, as `cargo` reports the version provided in the `Cargo.toml` file, while a code-dependant signal on the latest commit may in fact report data on a commit made later than when the manifest file and crates.io was last updated. An alternative approach may be to instead download a tarball of the source code for a specific version, and to run code-dependent queries against that. This does not provide a perfect representation of a project either, as signals without easy access to the time series (such as GitHub API responses), more accurately reflect the current status of the repository than any specific version.

Figure 3.6: The workflow of the `dcoll.sh` script, collecting data using `cargo-indicate`. The package list contains package names and a link to their git repository



3.5 Statistical Analysis

Due to the amount of different features collected for each package in the data collection, we want to identify which data points from each signal provide information, how signals correlate etc. To be able to answer the research questions as described in the introduction (Chapter 1) we must therefore analyze the data.

Even though `cargo-indicate` makes it possible to run any query anywhere in the dependency tree of a project, we only focus on root packages (i.e. the packages in our list described in Section 3.4.1) for the analysis. Due to our way of collecting data, this includes popular dependencies, since the popular packages likely appear in the dependency tree of many other packages. Using the scalar data provided by a thorough root-level query will then provide the insight that can later be used when constructing queries on the dependency tree.

For this section, we will focus on analysis on this root-level scalar data. As such, some parts of the capabilities of `cargo-indicate` will not be subject to thorough analysis.

One problem when working with data of the type presented here is that we may inadvertently affect our analysis by our own presumptions; for example, we have already discussed that we believe that unsafe Rust is useful in distinguishing between package characteristics. However, this can skew our view of our data. Therefore, we will suggest an approach whereby models will instead provide much of the interpretation we seek. This does come with the caveat that we cannot and do not state desireable outcomes; we simply do not know what analyzed packages may be of interest.

The rest of this section will provide different concepts that will aid us in finding value, or lack thereof, in the data provided by `cargo-indicate`.

3.5.1 Structuring Data

The basis for our work is the Python `pandas` library, a project providing many capabilities useful when working with the output of `cargo-indicate`. A pandas `DataFrame` is a data structure that provides a large amount of functionality, and in our case can be seen as a matrix of dimensions $p \times n$, where p is the packages, and n is the collected scalar values, or *features*. We note that the values in this matrix, $\mathbf{D}$, are not necessarily floating point values. Instead they can also be integers, booleans, dates etc. Pandas columns use the `dtype` property to describe which type a column may be. Some values in $\mathbf{D}$ may also be `None` or `NaN`, indicating no value present. This can be the case when `cargo-indicate` failed to retrieve some data for a package (such as GitHub stars for a package that is not hosted on GitHub). The *count* column in Table 4.1 provides some insight in which values where missing; generally information from `cargo-geiger` is more likely to be missing. This could be the case when `cargo-geiger` fails to compile the package in question.

Another important aspect is that `DataFrame` provides ways of filtering out rows that contain `None`/`NaN`, and to select columns with some data types only. For some calculations, only floating point operations make sense (and for these, pandas often ignore columns that have `dtype`s that does not make sense to include). Equation 3.1 shows an example of $\mathbf{D}$ (transposed for clarity).

$$\mathbf{D}^T = \begin{array}{r c} & \begin{array}{ccc} p_1 & p_2 & \cdots \end{array} \\ \begin{array}{r} \textbf{starsCount} \\ \textbf{files} \\ \textbf{usedTotalPercentageUnsafe} \\ \vdots \end{array} & \begin{bmatrix} 13 & 37 & \cdots \\ 4 & 20 & \cdots \\ \texttt{None} & 69.0 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \end{array} \tag{3.1}$$

3.5.2 Finding Correlations & Reducing Redundancy

This section provides two goals; To find which features (columns in $\mathbf{D}$) correlate with each other, and to which degree, and to eliminate duplicate information to provide a new DataFrame, without this redundancy. This is related to answering **RQ1** (*What signals contain information that can differentiate a package against another package?*) and providing insight into **RQ4** (*Does combining signals in multiple dimensions provide benefits for analysis of a Rust package?*).

We first must note that the columns in **D** have vastly different scales and distributions in its features. According the documentation provided by `scikit-learn`, most of the models require that the data is standardized, i.e. that they look like normally distributed data; Gaussian with zero mean and unit variance [4]. This is provided by the `sklearn.preprocessing` module. We will use `StandardScaler` for the purpose of this thesis, but note that we will first look at the distribution of the features to ensure normal distributions, and make necessary transformations.

A first approach is to plot the *correlation matrix* for **D**, using the `DataFrame` `corr` method. This matrix represent how much two columns correlate for each other, in our case for a scale from `0.0` to `1.0`, where `1.0` would mean that two features correlate directly to each other for all packages p_n in **D**. While a plot of this provides an intuitive understanding of the relationship between features, the same matrix can be used to prune unnecessary columns. We can set a limit `limit` of the highest amount of correlation between a pair of columns we accept before dropping one of the columns in a pair to remove the redundancy. A simple list interpretation can then produce our redundancy-free DataFrame. The `corr` method takes an optional parameter we can specify; method of correlation. The default Pearson method assumes that the relationship between variables are linear. An option is the Spearman method, which does not rely on the features being linear. It was decided to use the Spearman method after initial investigations of the distributions of features, which we discuss in Section 4.1.

The selection of `limit` is not trivial, but as we will see later it can be set quite high $(= 0.99)$ and then adjusted to allow for other parts of the analysis if deemed necessary.

One problem with the dataset as presented is that even with the most correlated features being removed, it remains highly dimensional, so it is not possible to represent these dimension in an intuitive way so that we may make conclusions. This problem could be solved with techniques reducing the dimensionality of the data, while preserving the maximum amount of information in the original data. We will here present models that may help us, but note that all models must be tested against the data set and against each other to ensure their suitability.

The models we will look at are

1. PCA, Principal Component Analysis
2. t-SNE, *t*-distributed stochastic neighbor embedding

PCA aims to reduce the amount of dimensions by projecting the data to a lower dimensional space; an in-depth review is presented in [13]. For visualization, two or three dimensions are suitable, but we may also let the implementation attempt to guess the amount of dimensions. This can be suitable for removing redundant features. PCA is a linear method relying on finding *principal components*. Each principal component is a unit vector, orthogonal to all others in the set. The first principal components follows the same direction as a line that best fits the data. The next principal component then follows the second-best line that is orthogonal to the first, and so on. The line that best fits the data here means the line that minimizes the average perpendicular distance from the points in the data set to the line.

By iterating through these principal components up to the desired amount of dimensions, we can explain the most amount of variance in the data using the first component, and then the second component explains the most amount of variance in what it is left, and so on.

We must here note that PCA is linear, that is, if the data we are reducing are not linear, the principal components may fail to represent the data accurately. Since we wish to discover relationships and trends in **D**, we must be carful not to make assumptions about relationships in the data.

t-SNE is another tool that can be used. It is described by [4] to be useful in data visualization, but may need PCA to reduce noise in the original data of the amount of features high (where 50 is suggested as a reasonable amount). The principle is to produce a map in lower dimensions (e.g. two or three), and then place points that preserves the local structure of the higher dimensions. This is done by first constructing a probability distribution over pairs of high-dimensional objects, such that similar objects are assigned high probability. A similar distribution is then defined for the lower dimension, minimizing the KL divergence between the higher and lower dimension distributions. The KL divergence is a measure of how one probability distribution differs from another probability distribution. The goal of this is to represent the distribution in higher dimensions accurately in the lower dimension. However, we note that t-SNE can create different results with different initializations [4], and that hyperparameters passed to it can drastically change the resulting graphs [24]. We decided to test t-SNE to explore if it provided any additional insights into the data we collect, but as we will discuss later were not able to find any additional insights by using t-SNE.

3.5.3 Finding Outliers

In answering **RQ6** (*How can we use signals to identify Rust packages in need of manual developer attention?*), we need understanding of what makes Rust crates stick out. Developers likely want to find crates in their dependencies that deviate from the normal, and thus it would be helpful to identify what is not normal.

One way of finding these outliers is the Mahalanobis distance $d_M(\vec{x}, Q)$ for some package data $\vec{x}$ for our dataset Q, as described by [5]. We only consider numerical values, i.e. `float` and `int`. This distance is a measure of the distance of a package to the distribution of packages we are analyzing, if we consider the data set as a probability distribution. Mahalanobis distance is defined as in Equation 3.2 for a distribution Q on $\mathbb{R}^N$ with a mean $\vec{\mu} = (\mu_1, \mu_2, \dots, \mu_N)$, for some point $\vec{x} = (x_1, x_2, \dots, x_N)$. Here, S is the positive-definite covariance matrix.

$$d_M(\vec{x}, Q) = \sqrt{(\vec{x} - \vec{\mu})^T S^{-1} (\vec{x} - \vec{\mu})} \tag{3.2}$$

To identify outliers, we can then assume a normal distribution of the distances, defining outliers as those with a distance d_M of equal to or greater than 3 standard deviations from the mean distance. This can be calculated using the z-score, described in Equation 3.3.

$$z = \frac{x - \mu}{\sigma} \tag{3.3}$$

Equation 3.3 means that $|z| > 3$ would meet our requirements for outliers.

This strategy avoids influencing what is to be defined as "deviating", but comes with the caveat that we do not inherently know what makes these points outliers. Here we must instead look at the features present in these points, and attempt to categorize them.

We must also here be careful of our input features, to not provide features we have earlier deemed redundant. We must also ensure that the underlying distributions are normal. We attempted to do so by looking at the distributions of our features, and using the $\log_{10}$ value of features that appear to have a log-normal distribution. While this make several features appear normal, we note that the approach is not perfect and that some features appear to not be normal, which may impact our results.

3.5.4 Finding Clusters

In the previous section we discussed finding correlations in our data set. In answering **RQ2** (*Do the selected signals correlate in any way, and if so, how?*) and **RQ3** (*Can we find clusters of Rust packages based on their signals?*), we want to find sets of characteristics that describe packages that, using all available features that do not overlap over a certain limit, can be said to share characteristics. We want to create labels for Rust packages, without knowing beforehand what those labels might be. This is a common use case for k-means clustering [22].

k-means can be described in four steps [22]:

1. Randomly select k centroids in a d-dimensional space, where d is the amount of features, and $1 \leq k \leq n$ for n data points
2. Assign each data point to its closest centroid
3. Move the centroids to the average location of the data points assigned to it
4. Repeat steps 2-3 until the assignments do not change (or change very little)

Since the initial selection of centroids is random, it is recommended to run the initialization multiple times to ensure that the solution is stable. There exists some additional known issues with k-means [22]:

- There is no exact way of selecting a 'correct' value for k
- There may exist more than one unique solution
- It is up to the researcher to interpret the resulting clusters

It is noted by [22] that the last issue often is the biggest problem. In our case, we wish to be able to describe what each cluster means in the context of the Rust ecosystem. Preferably we would like to identify if some cluster is especially dangerous to be a dependency, and what characterizes it.

One major benefit of using k-means, being an unsupervised learning algorithm, is that we as researchers do not set the labels, but they are instead found by the algorithm. This avoids the dangers of assuming that something 'should' be present in a dataset, when it actually is not.

We also note that k-means can be combined with dimension reduction techniques like PCA to present the clusters in 2D.

`scikit-learn` provides implementations of k-means which we use.

3.5.5 Using Existing Categorical Data

To answer **RQ5** (*Are Rust packages of the same category likely to have the same signal characteristics, and if so, what are they?*), we can use the data provided by crate authors themselves. The `Cargo.toml` file can contain two fields that provide categorical data; `categories` and `keywords`. Both of these are exposed as properties of the `Package` type in `cargo-indicate`s schema, and can thus be part of **D**, as list of strings. We will limit ourselves by only focusing on `categories`.

There exists some issues with this data however, that we must take note of. Firstly, we are not interested in how categories and keywords affect other parameters, instead we seek to find out how they appear in clusters. Thus they must be excluded in our previous analysis, to be added at a later stage (here `pandas` provides excellent functionality to exclude these columns). Our goal is to see if our clusters, describing similar signal characteristics, align with the categories provided by developers.

Secondly, we must transform the data to a usable format. We note the following about these features, as described by *The Cargo Book*:

1. A category can have any number of subcategories, denoted with a separating `::` (like `math::addition`)
2. We assume that a crate author is likely to reuse categories in the set of already existing one, like the ones provided by crates.io, but they may provide others
3. A crate can have up to 5 categories on crates.io, but an author may add more in the manifest file
4. Some categories are equal (like `no-std` and `no_std`)

For the first issue, we suggest several *subcategory renaming strategies*:

- `TOP_PARENT_CATEGORY`: For a subcategory `"top::mid::bot"`, use `["top"]`
- `BOTTOM_SUBCATEGORY`: For a subcategory `"top::mid::bot"`, use `["bot"]`
- `FULL_SPLIT`: For a subcategory `"top::mid::bot"`, use `["top", "mid", "bot"]`
- `NONE`: For a subcategory `"top::mid::bot"`, use `["top::mid::bot"]`

We note that this is not an exhaustive list of possible strategies, and others are possible. It is also possible to combine the existing categories. We limit ourselves to these strategies, and do not consider combinations of them.

For all strategies, duplicates are removed (`-` and `_` are considered to be equivalent (as they would on crates.io), and case is ignored).

There are benefits and drawbacks of all strategies. `TOP_PARENT_CATEGORY` loses some information, but also reduces the amount of strategies while giving an overview. `BOTTOM_SUBCATEGORY` retains detailed information, but can remove important context (consider the categories `"database::cache"` and `"http::cache"`). `FULL_SPLIT` keeps all parts, but still loses context and might be misleading. `NONE` retains all information, but can lead to large amounts of categories, and might actually reduce readability as subcategories that are very closely related are considered separate.

Considering this, we decided that `TOP_PARENT_CATEGORY` provided the best balance of maintaining specificity while still being easy to use. While it loses some specificity, other categories may be directly misleading (or in the case of `NONE`, too noisy).

With regards to making this information easily available, the `pandas.DataFrame get_dummies` method provides the needed functionality. This method breaks out the full set of categories for all packages in our data set (from the categories column), and creates a new set of boolean columns; each column represents a category and simply holds information if a package was listed with that category or not. The original category column can then be discarded. The following examples show a transformed matrix following a renaming strategy combined with application of `get_dummies`. These examples only show the category column(s); In reality the matrix would also contain all data features. We also note that using `NONE` would be the same as just using `get_dummies`.

Figure 3.7, 3.8, and 3.7 provide examples how package the category column may be split into several one-hot encoded columns for three example packages.

Figure 3.7: Transformation where `TOP_PARENT_CATEGORY` is used. Note how there is no overlap in categories in the resulting matrix

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \end{array} \begin{array}{c} \textbf{categories} \\ \left[\begin{array}{c} \{"database :: cache", "speed"\} \\ \{"http :: cache", "web\_programming", "web\_programming :: speed"\} \\ \{"no\_std"\} \end{array}\right] \end{array}$$

$$\downarrow \texttt{TOP\_PARENT\_CATEGORY} + \texttt{get\_dummies}$$

$$\begin{array}{c|ccccc} & \textbf{database} & \textbf{http} & \textbf{no\_std} & \textbf{speed} & \textbf{web\_programming} \\ p_1 & 1 & 0 & 0 & 1 & 0 \\ p_2 & 0 & 1 & 0 & 0 & 1 \\ p_3 & 0 & 0 & 1 & 0 & 0 \end{array}$$

Figure 3.8: Transformation where `BOTTOM_SUBCATEGORY` is used. Note that p_1 and p_2 overlap in categories, although these subcategories represent different things.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \end{array} \begin{array}{c} \textbf{categories} \\ \left[\begin{array}{c} \{"database :: cache", "speed"\} \\ \{"http :: cache", "web\_programming", "web\_programming :: speed"\} \\ \{"no\_std"\} \end{array}\right] \end{array}$$

$$\downarrow \texttt{BOTTOM\_SUBCATEGORY} + \texttt{get\_dummies}$$

$$\begin{array}{c|cccc} & \textbf{cache} & \textbf{no\_std} & \textbf{speed} & \textbf{web\_programming} \\ \hline p_1 & 1 & 0 & 1 & 0 \\ p_2 & 1 & 0 & 1 & 1 \\ p_3 & 0 & 1 & 0 & 0 \end{array}$$

Figure 3.9: Transformation where `FULL_SPLIT` is used. There is now a lot of overlap in p_1 and p_2, but they retain information.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \end{array} \begin{array}{c} \textbf{categories} \\ \left[\begin{array}{c} \{"database :: cache", "speed"\} \\ \{"http :: cache", "web\_programming", "web\_programming :: speed"\} \\ \{"no\_std"\} \end{array}\right] \end{array}$$

$$\downarrow \texttt{FULL\_SPLIT} + \texttt{get\_dummies}$$

$$\begin{array}{c|cccccc} & \textbf{cache} & \textbf{database} & \textbf{http} & \textbf{no\_std} & \textbf{speed} & \textbf{web\_programming} \\ \hline p_1 & 1 & 1 & 0 & 0 & 1 & 0 \\ p_2 & 1 & 0 & 1 & 0 & 1 & 1 \\ p_3 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

Chapter 4
Results

In this chapter we will present the results of the data collected to evaluate `cargo-indicate` and its signals. We will start by presenting the raw data, its distribution across all target packages. We will then present correlations found in the data set, and our reduction of redundancy. Following that, we present identified outliers. These are then put into context as we present our clustering analysis, incorporating knowledge gained from the distributions and marking outliers.

`cargo-indicate` only provides the underlying data in this chapter, that is, JSON files with query results. The visualization and other parts of the presented results were created using a separate Python program.

4.1 Distribution of Raw Data

Table 4.1 presents an overview of the raw data. We note that the *count* is significantly lower for the features relying on `cargo-geiger`, making it likely that the tool was not able to compile some packages. A short description of each feature is available in Appendix A.

The distribution of data can be seen in Figure 4.1-4.9. We note that some values are better described on a log scale, such as `lines`, `cratesIoTotalDownloads`, etc. This was found after comparing the distributions on a linear scale and a log scale, selecting the scale that provided the visualization that most resembled a known distribution (in most cases a normal distribution). These are provided on a log scale. To decrease variance and ease scaling, these values will be replaced with their $\log_{10}$ for the rest of the analysis, when nothing else is mentioned. Which values are replaced by their logarithm is also described in Appendix A. We found this to improve the results of clustering and outlier detection, in accordance with theory (lower variance is useful for k-means, and Mahalanobis relies on the normal distribution of the input features). These figures also provide insights into the for some features multimodal distribution, which we will discuss further in Section 5.1. They also include information about the amount of packages with `NaN` as a value, as well as when the package result was above 0.

Table 4.1: Raw data used for analysis. *count* is the amount of non-`NaN` data points for this feature, and **25%**, **50%**, and **75%** are the 25th, 50th, and 75th percentiles respectively

	count	mean	min	25%	50%	75%	max	std
ghUnixCreatedAt	2136	2015-03-09	2008-02-25	2012-03-30	2015-01-17	2018-03-11	2023-01-15	-
ghFollowersCount	2136	734.2	0	20	87	433.2	3.4e+04	2683.7
starsCount	2136	1229.5	0	22	114.5	608.5	6.2e+04	4636.1
forksCount	2136	99.3	0	4	19	89	2352	233.2
openIssuesCount	2136	41.5	0	1	8	33	1427	95.0
watchersCount	2136	1229.5	0	22	114.5	608.5	6.2e+04	4636.1
files	2211	19.6	1	2	6	16	4736	114.7
lines	2211	6537.8	0	441.5	1339	3953	3.0e+06	7.0e+04
blanks	2211	475.0	0	49	156	436	1.2e+05	2592.3
code	2211	5888.0	0	353	1104	3307.5	3.0e+06	6.9e+04
comments	2211	174.9	0	7	34	134	1.2e+04	602.3
commentsToCode	2209	0.1	0	0.0	0.0	0.1	6.4	0.3
directDepsCount	2211	4.8	0	1	3	6	56	6.3
totalAdvisoryCount	2211	0.1	0	0	0	0	7	0.4
cratesIoTotalDownloads	2211	1.1e+07	92	3.1e+04	4.1e+05	6.0e+06	2.1e+08	2.6e+07
cratesIoRecentDownloads	2211	1.7e+06	87	7650	8.7e+04	1.2e+06	3.3e+07	3.9e+06
cratesIoVersionDownloads	2034	2.1e+06	0	1471.8	3.6e+04	4.6e+05	1.1e+08	7.7e+06
cratesIoVersionsCount	2211	23.5	1	7	14	29	401	30.2
cratesIoYankedVersionsCount	2211	1.6	0	0	0	1	148	6.4
cratesIoYankedRatio	2211	0.1	0	0	0	0.0	0.9	0.1
usedExprsPercentageUnsafe	1844	6.2	0	0	0	1.3	100	16.2
usedExprsSafe	1844	2093.6	0	124.8	464	1549	1.9e+05	8247.9
usedExprsTotal	1844	2245.6	0	133	516.5	1600.2	1.9e+05	8498.2
usedExprsUnsafe	1844	152.0	0	0	0	10	3.6e+04	1272.9
usedFunctionsPercentageUnsafe	1844	4.1	0	0	0	0	100	15.3
usedFunctionsSafe	1844	22.8	0	1	6	18.2	2429	100.0
usedFunctionsTotal	1844	25.4	0	1	7	20	2429	113.6
usedFunctionsUnsafe	1844	2.7	0	0	0	0	1315	39.2
usedItemImplsPercentageUnsafe	1844	1.5	0	0	0	0	100	7.8
usedItemImplsSafe	1844	58.9	0	3	15	45	9719	262.3
usedItemImplsTotal	1844	60.0	0	3	15	46	9719	262.9
usedItemImplsUnsafe	1844	1.0	0	0	0	0	128	6.0
usedItemTraitsPercentageUnsafe	1844	1.2	0	0	0	0	100	8.7
usedItemTraitsSafe	1844	3.3	0	0	1	3	304	9.9
usedItemTraitsTotal	1844	3.4	0	0	1	3	304	10.0
usedItemTraitsUnsafe	1844	0.1	0	0	0	0	16	0.8
usedMethodsPercentageUnsafe	1844	1.4	0	0	0	0	100	7.5
usedMethodsSafe	1844	131.3	0	5	33	106	1.1e+04	445.6
usedMethodsTotal	1844	134.8	0	6	33	109	1.2e+04	463.9
usedMethodsUnsafe	1844	3.5	0	0	0	0	665	33.0
usedTotalPercentageUnsafe	1844	5.6	0	0	0	1.3	100	15.0
usedTotalSafe	1844	2309.9	0	146	542	1704	1.9e+05	8554.5
usedTotalTotal	1844	2469.2	0	157	611.5	1772.5	1.9e+05	8844.9
usedTotalUnsafe	1844	159.3	0	0	0	11	3.6e+04	1317.2

Figure 4.1: Distributions of numerical `cargo-indicate` features used in analysis. Log scales added where appropriate. On log scales zero values have been removed, but are visible on left bar plot, `NaN` values are never present in distribution plot

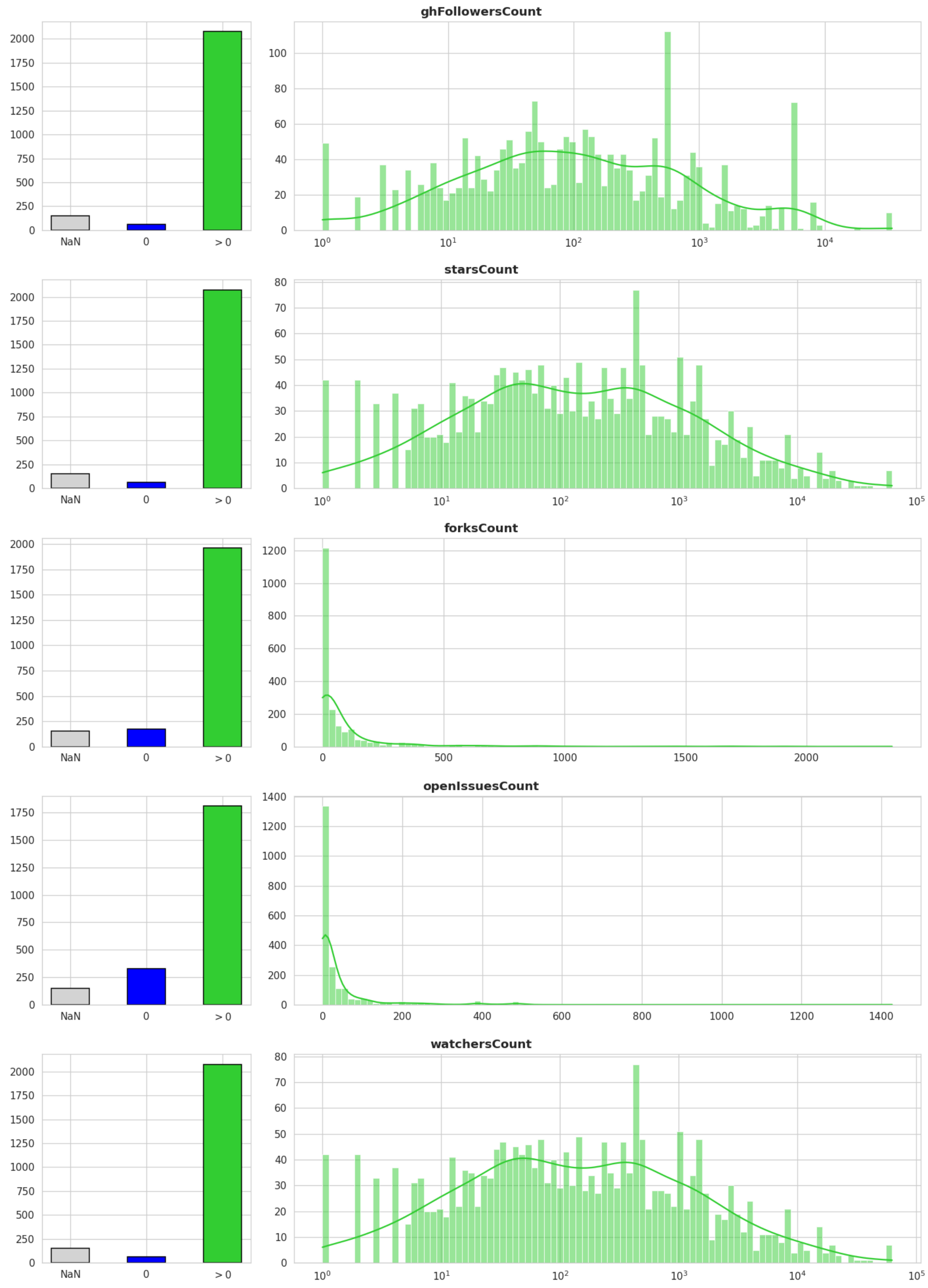


Figure 4.2: Distributions of numerical `cargo-indicate` features used in analysis. Log scales added where appropriate. On log scales zero values have been removed, but are visible on left bar plot, `NaN` values are never present in distribution plot

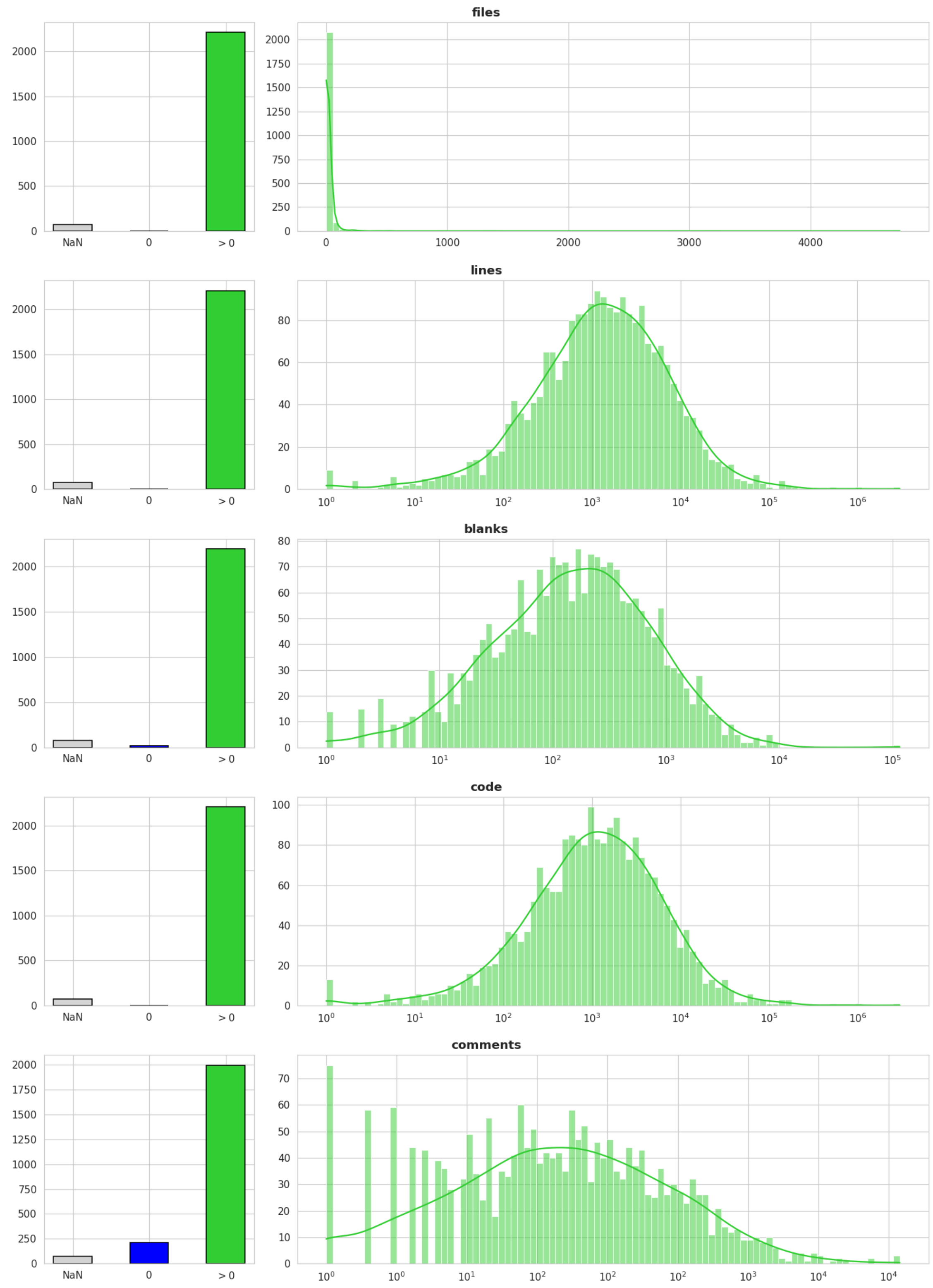


Figure 4.3: Distributions of numerical `cargo-indicate` features used in analysis. Log scales added where appropriate. On log scales zero values have been removed, but are visible on left bar plot, `NaN` values are never present in distribution plot

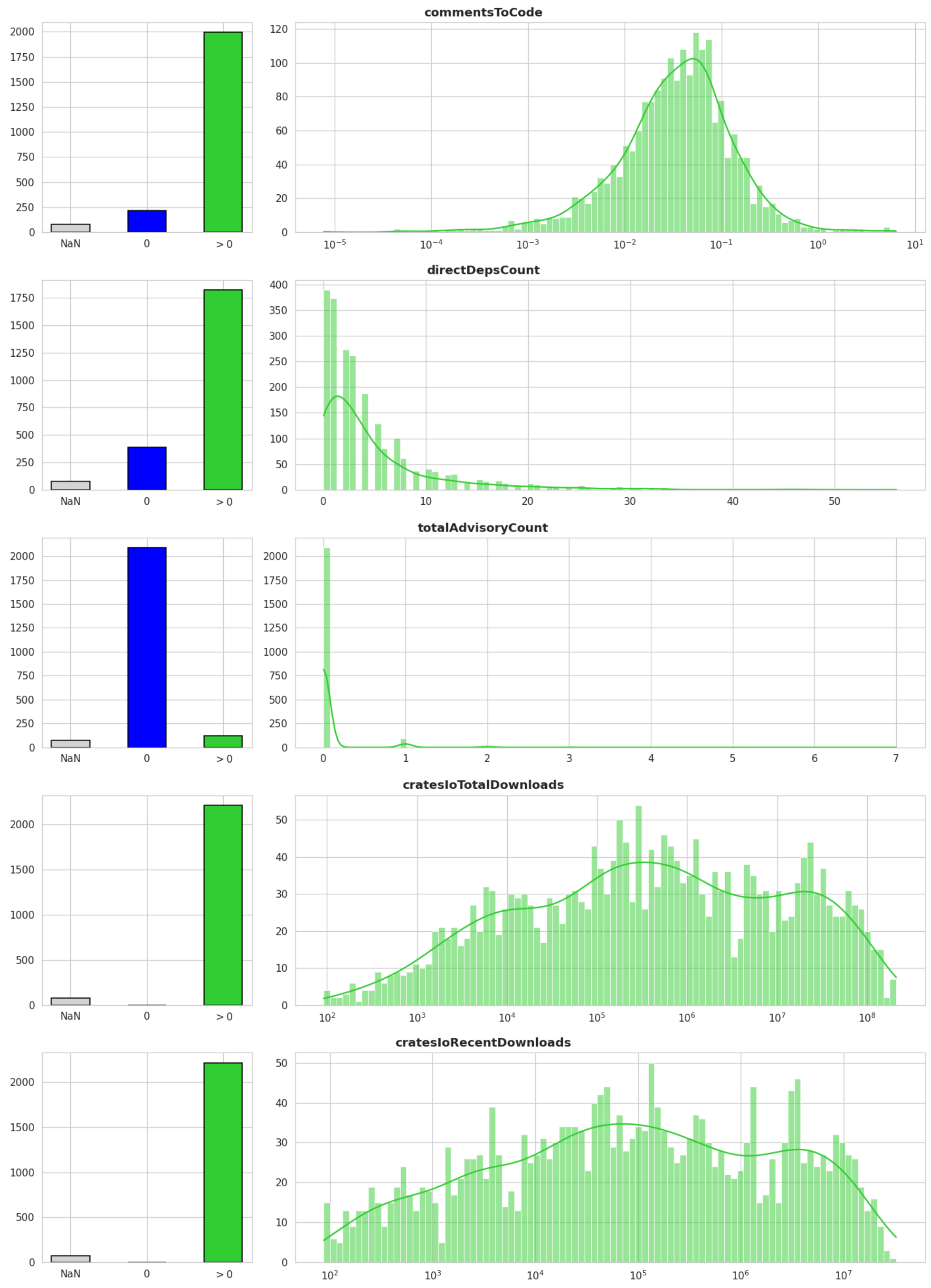


Figure 4.4: Distributions of numerical `cargo-indicate` features used in analysis. Log scales added where appropriate. On log scales zero values have been removed, but are visible on left bar plot, `NaN` values are never present in distribution plot

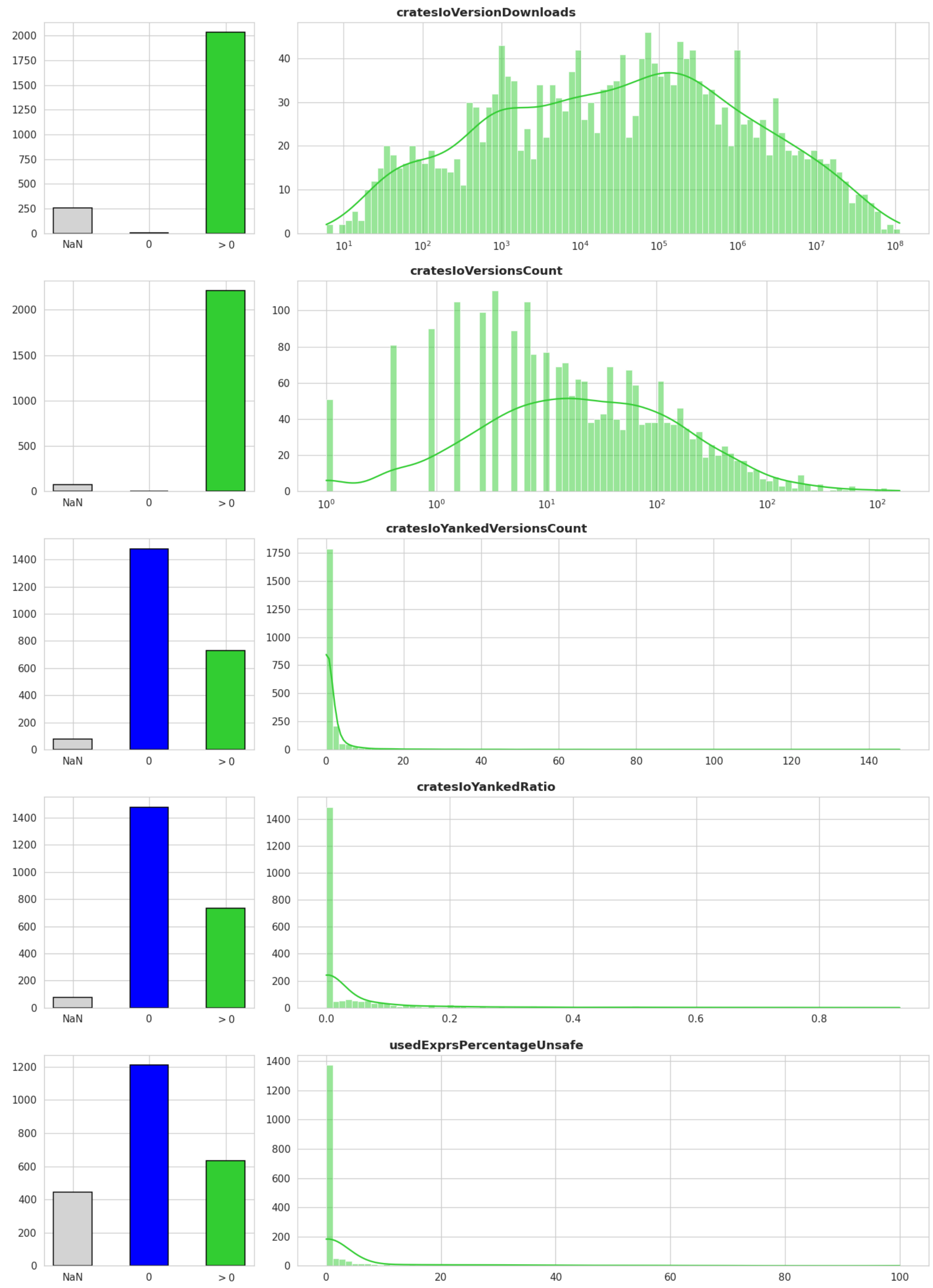


Figure 4.5: Distributions of numerical `cargo-indicate` features used in analysis. Log scales added where appropriate. On log scales zero values have been removed, but are visible on left bar plot, `NaN` values are never present in distribution plot

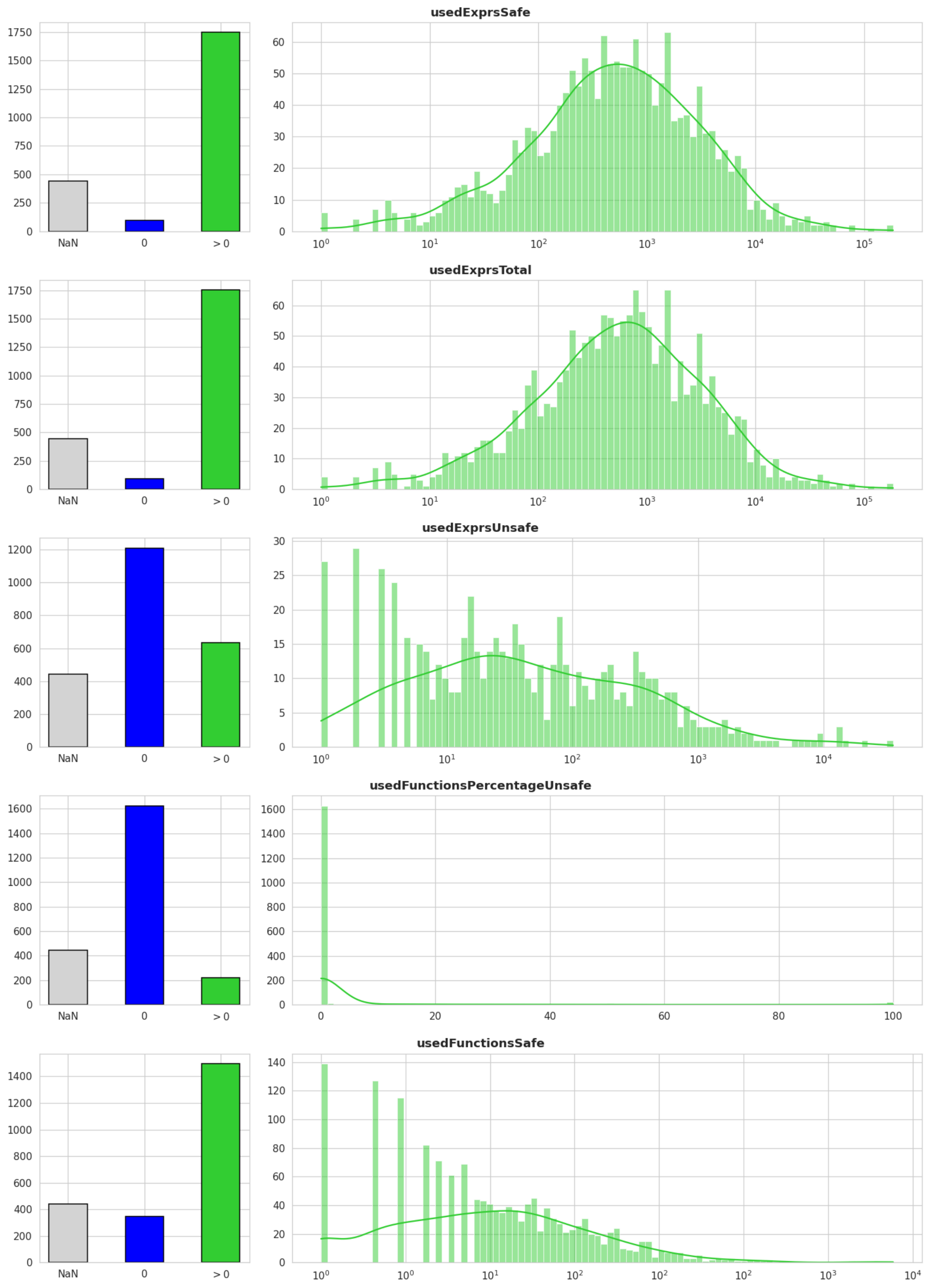


Figure 4.6: Distributions of numerical `cargo-indicate` features used in analysis. Log scales added where appropriate. On log scales zero values have been removed, but are visible on left bar plot, `NaN` values are never present in distribution plot

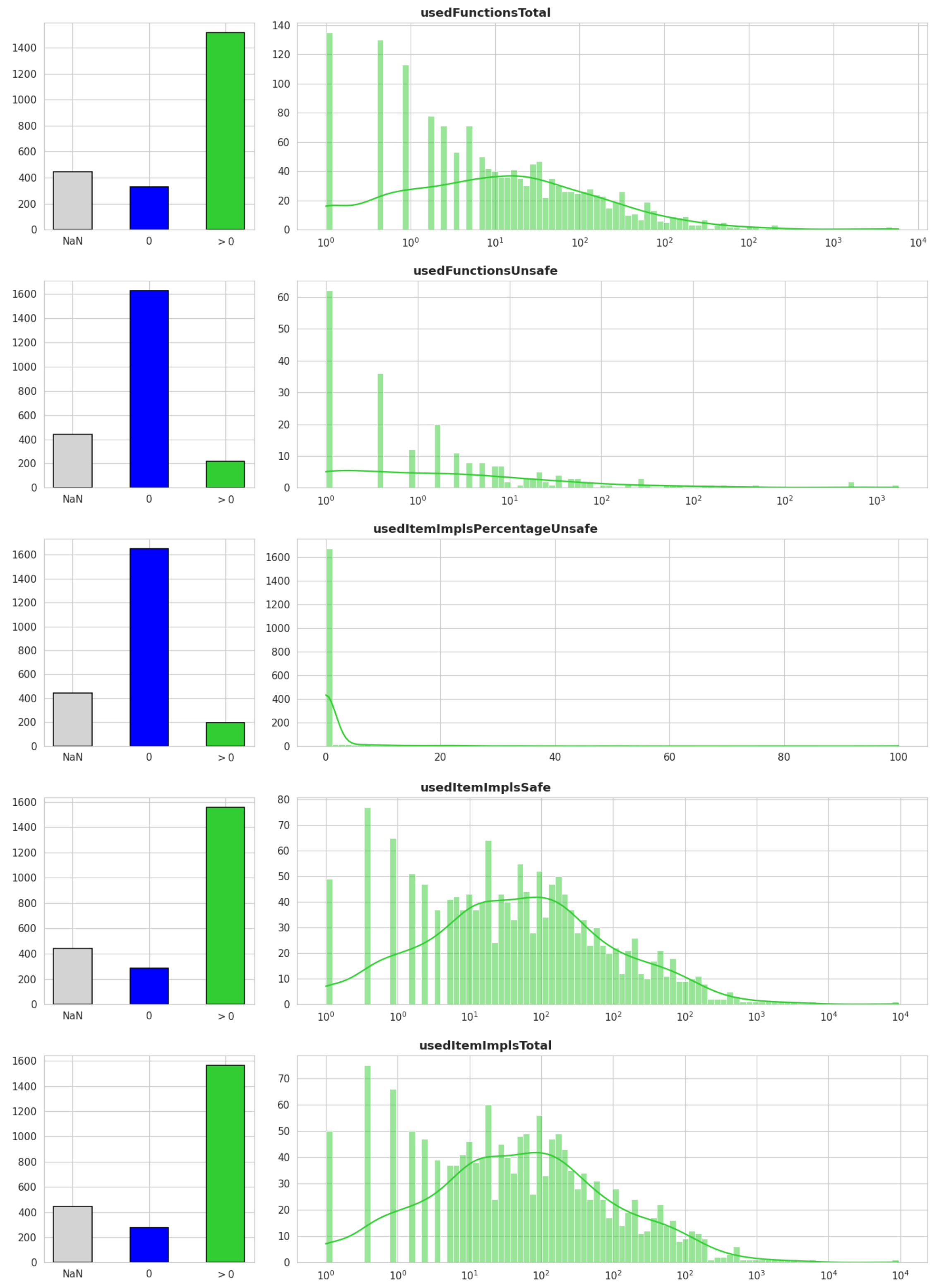


Figure 4.7: Distributions of numerical `cargo-indicate` features used in analysis. Log scales added where appropriate. On log scales zero values have been removed, but are visible on left bar plot, `NaN` values are never present in distribution plot

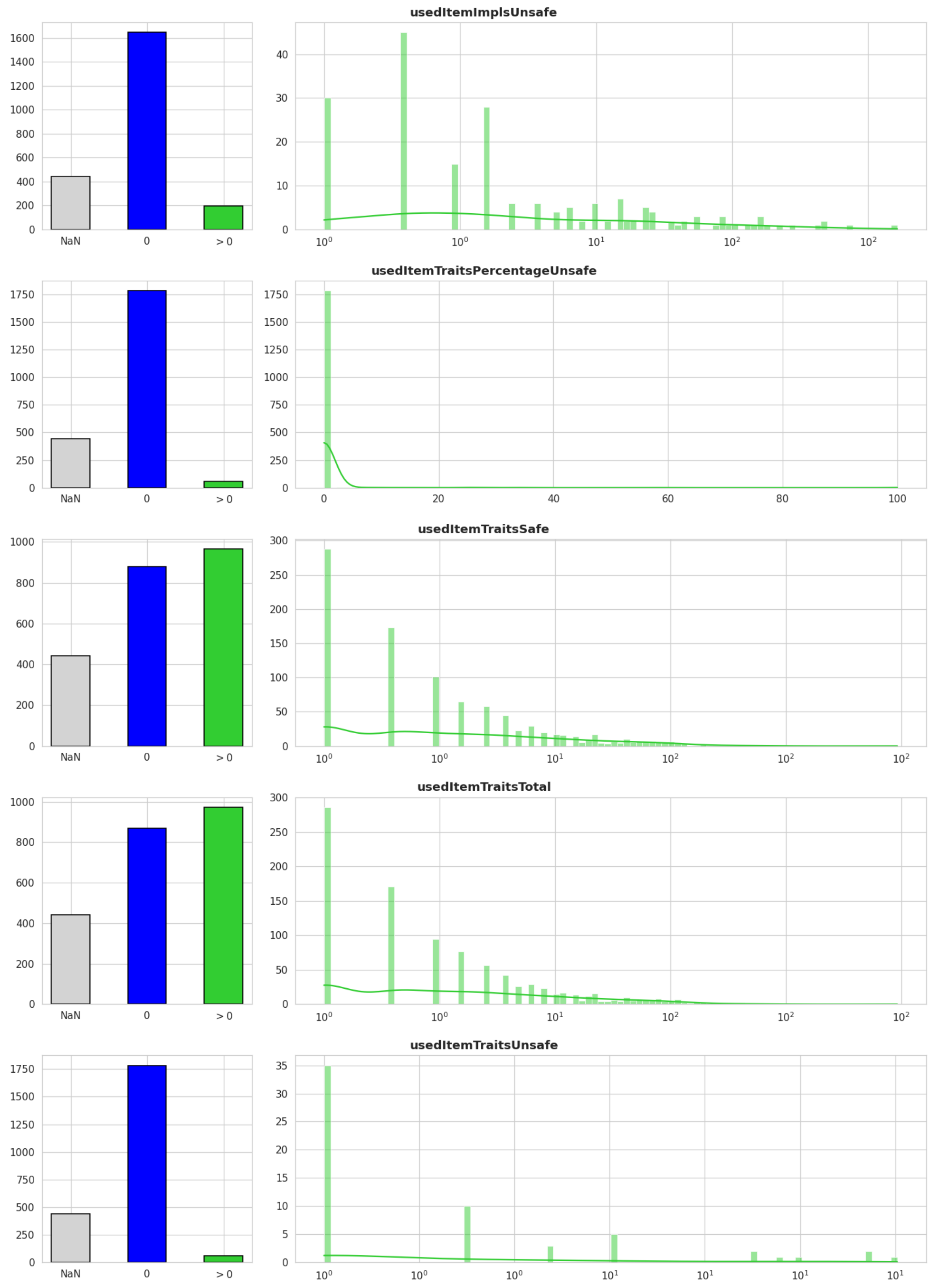


Figure 4.8: Distributions of numerical `cargo-indicate` features used in analysis. Log scales added where appropriate. On log scales zero values have been removed, but are visible on left bar plot, `NaN` values are never present in distribution plot

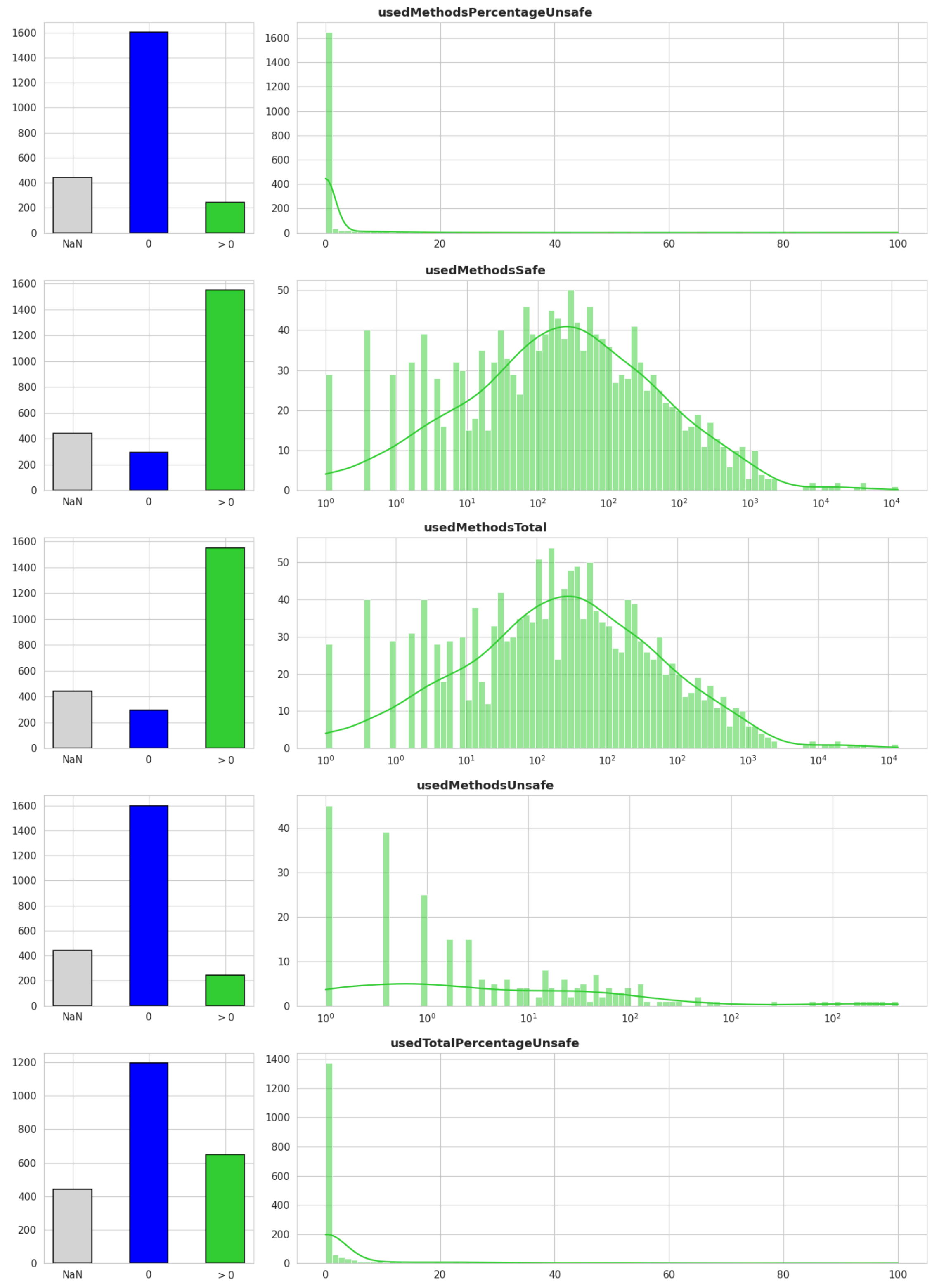
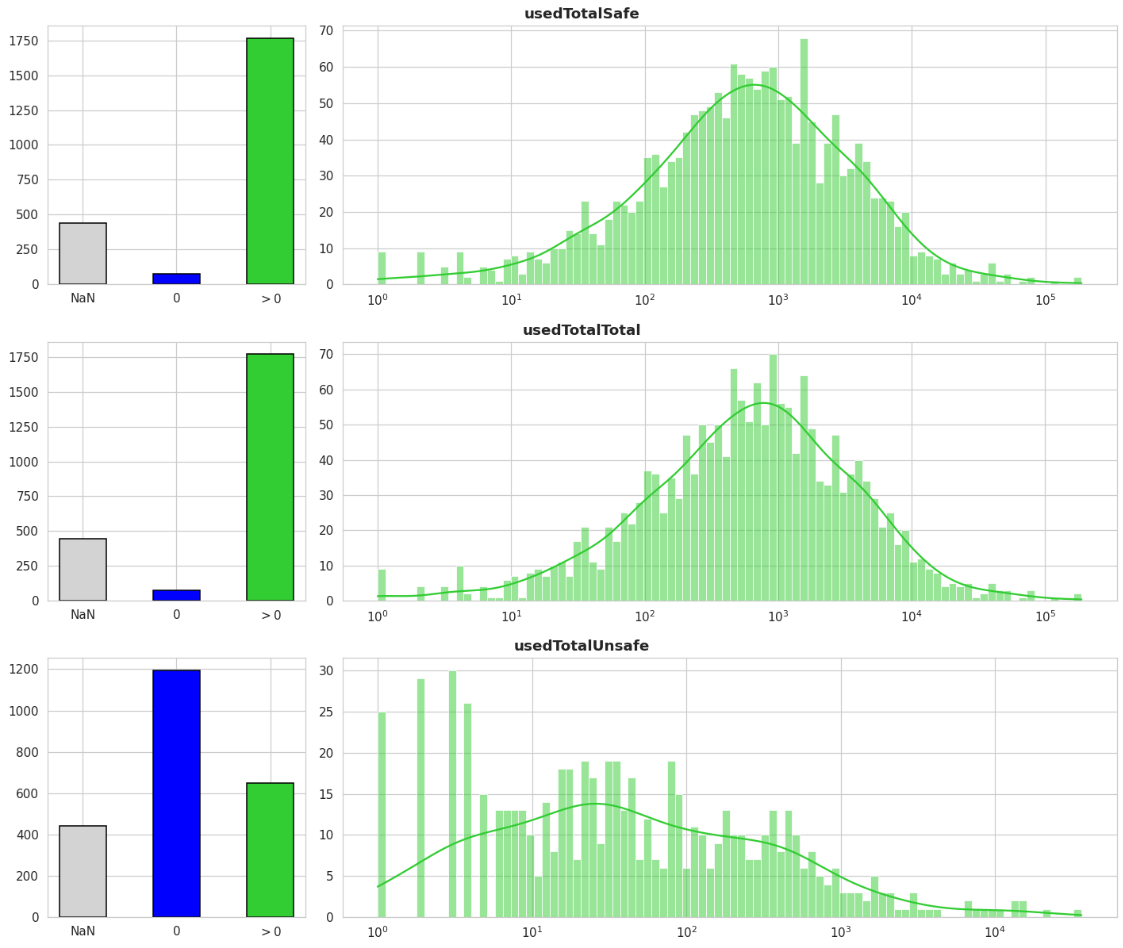


Figure 4.9: Distributions of numerical `cargo-indicate` features used in analysis. Log scales added where appropriate. On log scales zero values have been removed, but are visible on left bar plot, `NaN` values are never present in distribution plot



4.2 Correlations

Figure 4.10 and Figure 4.11 provide the correlation between the numerical and boolean features of the dataset. Note that these values are not scaled, and are not logarithms of the measured values. Since some values have been found to be log-normally distributed, and some not, the `spearman` method was selected for the `DataFrame.corr` method. This method does not rely on linear relationships to find the correlations. Those values that were deemed too similar, with a limit above `0.95`, were excluded from the later analysis. See Table 4.2 for these redundant values.

Table 4.2: Columns dropped due to correlation equal or higher than 0.95 using Spearman coefficient; dropped columns chosen randomly

Removed Column	Correlated Column(s)
cratesIoTotalDownloads	cratesIoRecentDownloads (0.98)
cratesIoYankedVersionsCount	cratesIoYankedRatio (0.98)
lines	code (1.00)
starsCount	watchersCount (1.00)
usedExprsPercentageUnsafe	usedExprsUnsafe (0.98), usedTotalPercentageUnsafe (0.99), usedTotalUnsafe (0.97)
usedExprsSafe	usedExprsTotal (0.98), usedTotalSafe (1.00), usedTotalTotal (0.98)
usedExprsTotal	usedTotalSafe (0.98), usedTotalTotal (1.00)
usedExprsUnsafe	usedTotalPercentageUnsafe (0.97), usedTotalUnsafe (0.99)
usedFunctionsPercentageUnsafe	usedFunctionsUnsafe (1.00)
usedFunctionsSafe	usedFunctionsTotal (0.98)
usedItemImplsPercentageUnsafe	usedItemImplsUnsafe (1.00)
usedItemImplsSafe	usedItemImplsTotal (1.00), usedMethodsSafe (0.96), usedMethodsTotal (0.96)
usedItemImplsTotal	usedMethodsSafe (0.96), usedMethodsTotal (0.95)
usedItemTraitsPercentageUnsafe	usedItemTraitsUnsafe (1.00)
usedItemTraitsSafe	usedItemTraitsTotal (0.99)
usedMethodsPercentageUnsafe	usedMethodsUnsafe (1.00)
usedMethodsSafe	usedMethodsTotal (1.00)
usedTotalPercentageUnsafe	usedTotalUnsafe (0.98)
usedTotalSafe	usedTotalTotal (0.98)

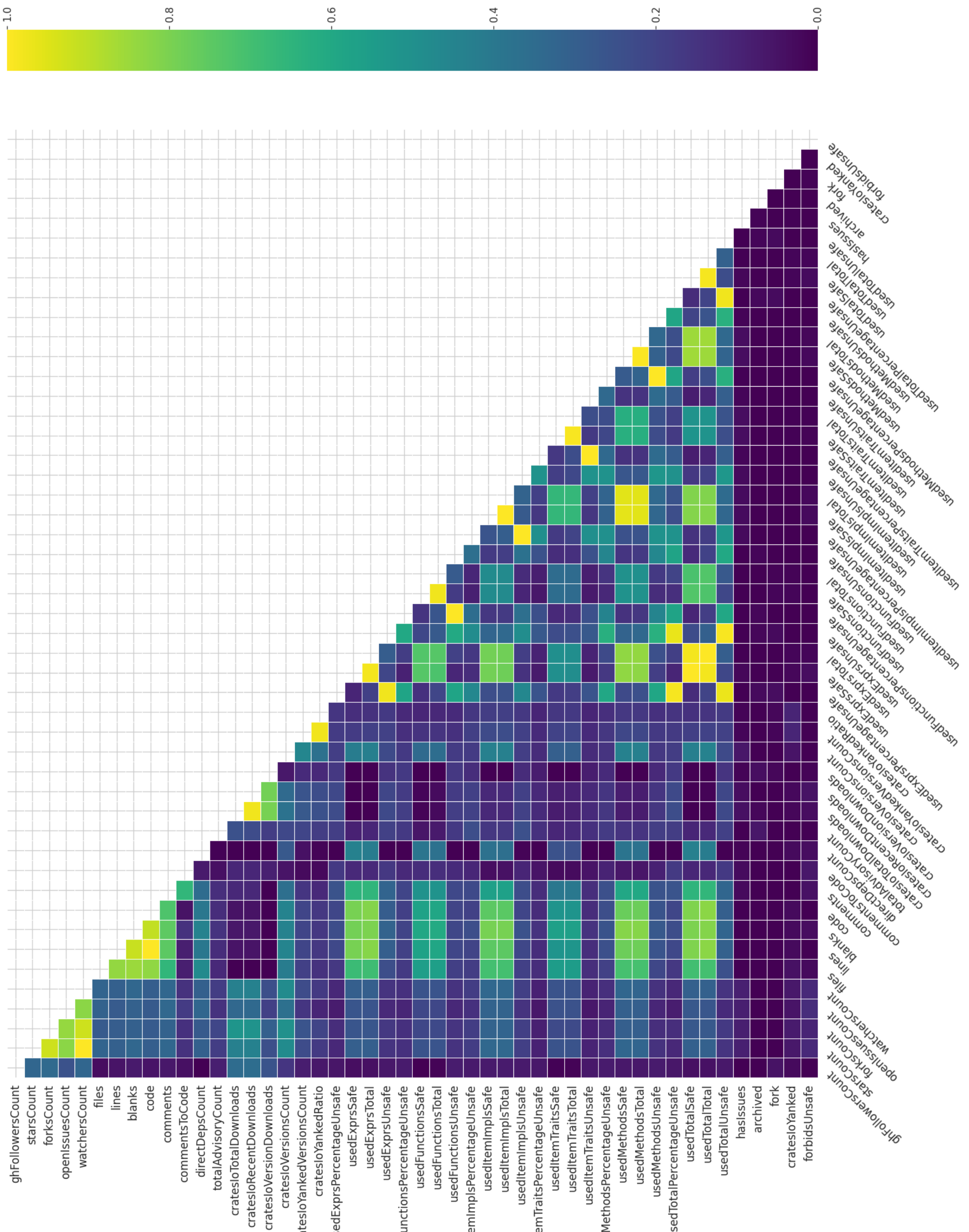
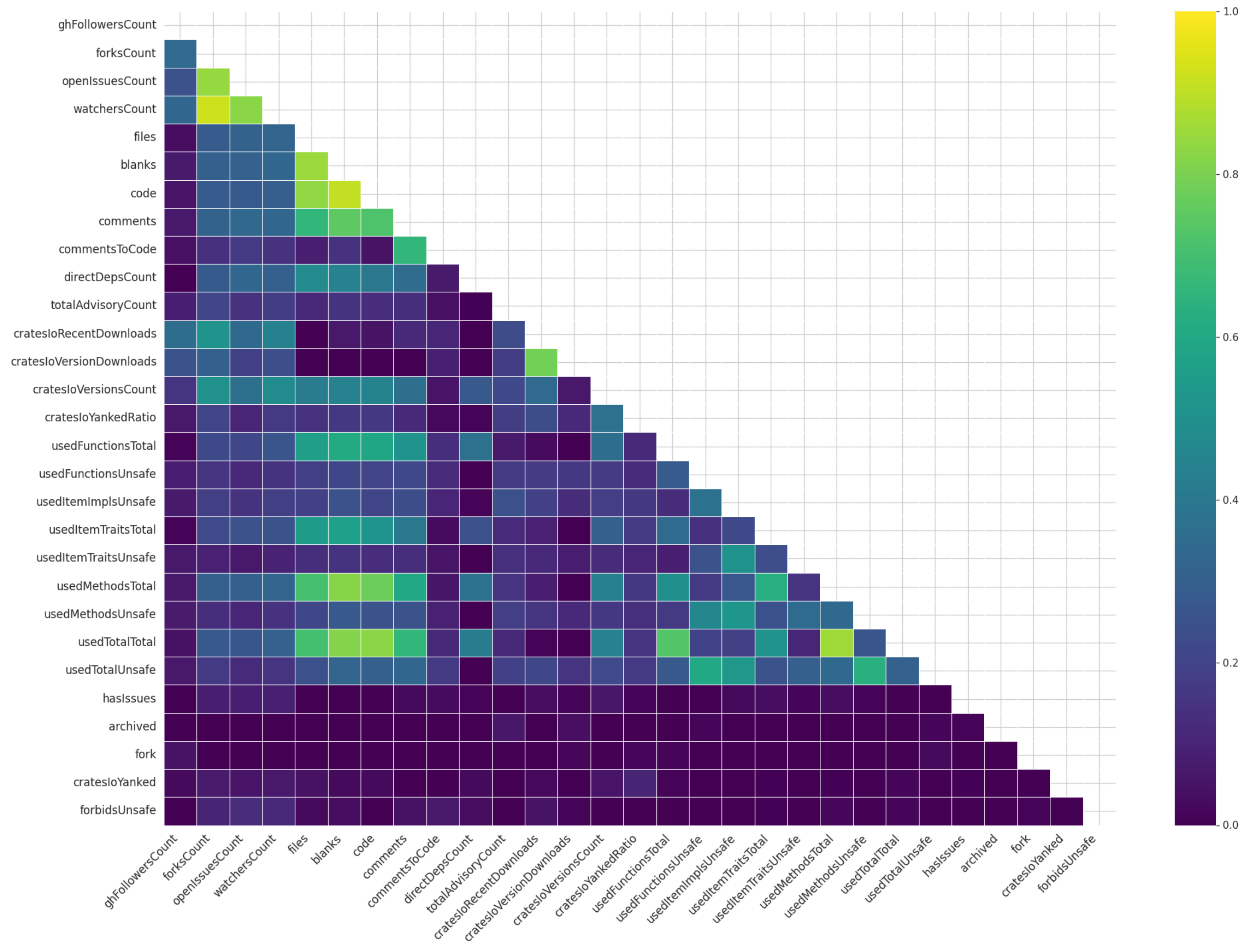


Figure 4.10: Full correlation matrix for numerical and boolean features, redundant values masked, using Spearman's rank correlation coefficient

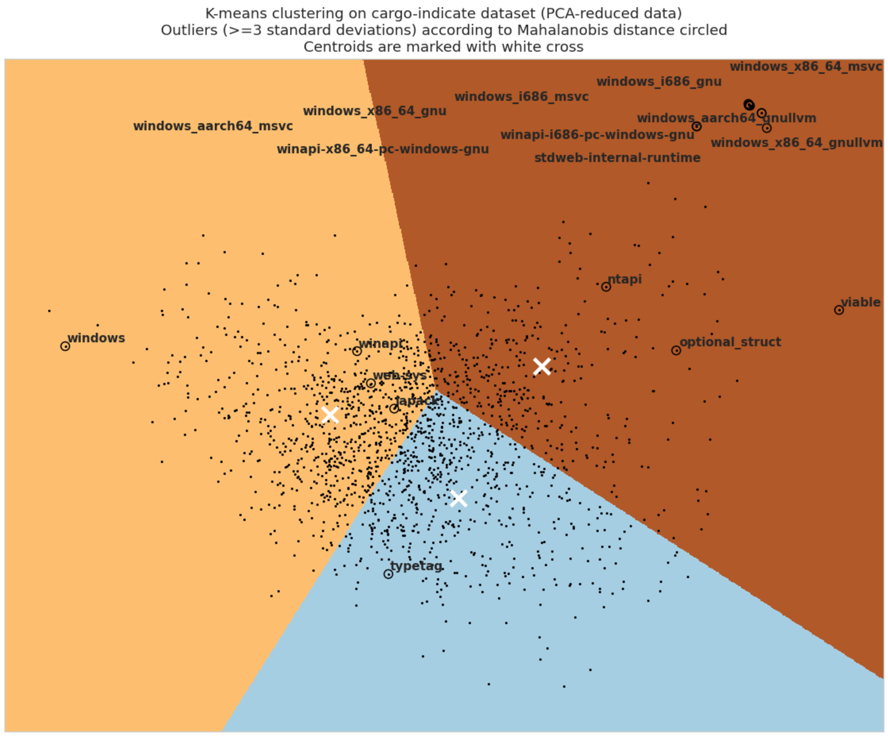
Figure 4.11: Correlation matrix for numerical and boolean features, with a correlation limit of `0.95`, redundant values masked, using Spearman's rank correlation coefficient



4.3 Clustering & Outliers

During the data analysis we decided to use a value of $k = 3$ for k-means clustering. This value provided clear clusters with a significant amount of packages each, while still being easy to describe later. Adding another cluster made the analysis harder, and we decided to use $k = 3$ as we could not provide a better analysis with more clusters. Figure 4.12 gives a representation of the clusters on PCA-reduced data.

Figure 4.12: k-means clustering on PCA-reduced data, with identified log features as logarithms. Colored areas are cluster borders. Descriptions of crates can be found in Table 4.4. Note that several `windows_*` and `winapi_*` packages are stacked on top of each other



A visualization using t-SNE was also attempted, but we failed to produce results that were adequate for analysis or even as a visual aid. We find that the PCA-reduced plot provides adequate support in this regard.

While this gives visual insight in a reduced *k*-means, the following figures instead describes a *k*-means with all data available (except the features identified as redundant). Thus it is not possible to say which cluster in Figure 4.12 corresponds to which label (a, b, or c), since the data used to create the clusters differ.

Table 4.3: Distributions of categories using the `TOP_PARENT_CATEGORY` strategy; majority marked in bold, actual frequency in parenthesis

Category	Total	a	b	c
accessibility	77	27.27% (21)	2.60% (2)	**70.13% (54)**
algorithms	125	**56.80% (71)**	10.40% (13)	32.80% (41)
api_bindings	84	**47.62% (40)**	17.86% (15)	34.52% (29)
ast_implementations	1	**100.00% (1)**	0.00% (0)	0.00% (0)
asynchronous	127	**71.65% (91)**	7.87% (10)	20.47% (26)
authentication	56	**64.29% (36)**	1.79% (1)	33.93% (19)
caching	11	27.27% (3)	27.27% (3)	**45.45% (5)**
cli	103	**57.28% (59)**	0.97% (1)	41.75% (43)
command_line	1	0.00% (0)	0.00% (0)	**100.00% (1)**
command_line_utilities	84	**71.43% (60)**	4.76% (4)	23.81% (20)
compilers	57	**71.93% (41)**	0.00% (0)	28.07% (16)
compression	59	**59.32% (35)**	6.78% (4)	33.90% (20)
computer_vision	1	**100.00% (1)**	0.00% (0)	0.00% (0)
concurrency	52	38.46% (20)	**40.38% (21)**	21.15% (11)
config	80	36.25% (29)	0.00% (0)	**63.75% (51)**
cryptography	132	**74.24% (98)**	6.82% (9)	18.94% (25)
data_structures	140	**52.14% (73)**	24.29% (34)	23.57% (33)
database	62	**67.74% (42)**	14.52% (9)	17.74% (11)
database_implementations	4	**50.00% (2)**	**50.00% (2)**	0.00% (0)
date_and_time	74	40.54% (30)	2.70% (2)	**56.76% (42)**
development_tools	153	**60.13% (92)**	9.15% (14)	30.72% (47)
email	4	**50.00% (2)**	0.00% (0)	**50.00% (2)**
embedded	18	**61.11% (11)**	5.56% (1)	33.33% (6)
emulators	3	**66.67% (2)**	0.00% (0)	33.33% (1)
encoding	134	**66.42% (89)**	10.45% (14)	23.13% (31)
external_ffi_bindings	58	34.48% (20)	5.17% (3)	**60.34% (35)**
ffi	1	**100.00% (1)**	0.00% (0)	0.00% (0)
filesystem	37	**51.35% (19)**	8.11% (3)	40.54% (15)
finance	1	**100.00% (1)**	0.00% (0)	0.00% (0)
game_development	25	32.00% (8)	28.00% (7)	**40.00% (10)**
game_engines	20	30.00% (6)	15.00% (3)	**55.00% (11)**
games	5	**60.00% (3)**	0.00% (0)	40.00% (2)
graphics	61	**57.38% (35)**	18.03% (11)	24.59% (15)
gui	16	**43.75% (7)**	18.75% (3)	37.50% (6)
hardware_support	13	30.77% (4)	30.77% (4)	**38.46% (5)**
history	1	**100.00% (1)**	0.00% (0)	0.00% (0)
internationalization	60	45.00% (27)	1.67% (1)	**53.33% (32)**
localization	17	**52.94% (9)**	0.00% (0)	47.06% (8)
mathematics	81	**66.67% (54)**	1.23% (1)	32.10% (26)
memory_management	18	33.33% (6)	**55.56% (10)**	11.11% (2)
multimedia	71	**73.24% (52)**	15.49% (11)	11.27% (8)
network_programming	89	**68.54% (61)**	11.24% (10)	20.22% (18)

Continued on next page

Table 4.3: Distributions of categories using the `TOP_PARENT_CATEGORY` strategy; majority marked in bold, actual frequency in parenthesis

Category	Total	a	b	c
no_std	270	**58.15% (157)**	19.26% (52)	22.59% (61)
os	83	**49.40% (41)**	18.07% (15)	32.53% (27)
parser_implementations	79	**70.89% (56)**	8.86% (7)	20.25% (16)
parsing	94	**67.02% (63)**	6.38% (6)	26.60% (25)
rendering	40	**52.50% (21)**	17.50% (7)	30.00% (12)
rust_patterns	61	**44.26% (27)**	19.67% (12)	36.07% (22)
science	105	**61.90% (65)**	8.57% (9)	29.52% (31)
simulation	54	**51.85% (28)**	1.85% (1)	46.30% (25)
storage	2	0.00% (0)	0.00% (0)	**100.00% (2)**
template_engine	5	**80.00% (4)**	0.00% (0)	20.00% (1)
testing	1	**100.00% (1)**	0.00% (0)	0.00% (0)
text_editors	29	**65.52% (19)**	3.45% (1)	31.03% (9)
text_processing	125	**56.00% (70)**	12.00% (15)	32.00% (40)
value_formatting	32	**46.88% (15)**	15.62% (5)	37.50% (12)
virtualization	3	**66.67% (2)**	33.33% (1)	0.00% (0)
visualization	12	**75.00% (9)**	0.00% (0)	25.00% (3)
wasm	67	**76.12% (51)**	7.46% (5)	16.42% (11)
web_programming	135	**68.15% (92)**	7.41% (10)	24.44% (33)
All Categories	3413	58.13% (1984)	10.90% (372)	30.97% (1057)

We can also list our outliers this way, as in Table 4.4. It is harder to say what makes these packages stand out, but a more thorough analysis could manually review them to attempt to describe what makes them stand out. We discuss the outliers further in Section 5.3.

Table 4.4: Outliers with an absolute z-score of 3 or higher. Only floats and integers used in the Mahalanobis distance; Correlation limit for columns is 0.93. Relative Mahalanobis distance is distance/mean distance. Descriptions taken from crates.io

	Outlier Placement	Relative Mahalanobis distance	Cluster	Description
ntapi	0	3.7	c	FFI bindings for Native API
web-sys	1	2.8	a	Generated bindings for all Web APIs
winapi-x86_64-pc-windows-gnu	2	2.7	c	Library for Windows API
winapi-i686-pc-windows-gnu	3	2.7	c	Library for Windows API

Continued on next page

Table 4.4: Outliers with an absolute z-score of 3 or higher. Only floats and integers used in the Mahalanobis distance; Correlation limit for columns is 0.93. Relative Mahalanobis distance is distance/mean distance. Descriptions taken from crates.io

	Outlier Placement	Relative Mahalanobis distance	Cluster	Description
optional_struct	4	2.6	c	Macro that will generate a struct with Option fields from another struct
typetag	5	2.6	a	Serde serializable and deserializable trait objects
lapack	6	2.5	a	Wrapper for LAPACK (Fortran)
viable	7	2.3	c	Interop with C++ MSVC VTables through Rust (namespace marked as open to new owner)
stdweb-internal-runtime	8	2.2	c	Internal runtime for the ‘stdweb‘ crate
windows_x86_64_msvc	9	2.2	c	Import lib for Windows
windows_i686_msvc	10	2.2	c	Import lib for Windows
windows_i686_gnu	11	2.2	c	Import lib for Windows
windows_x86_64_gnu	12	2.2	c	Import lib for Windows
windows_aarch64_msvc	13	2.2	c	Import lib for Windows
windows_x86_64_gnullvm	14	2.2	c	Import lib for Windows
windows_aarch64_gnullvm	15	2.2	c	Import lib for Windows
windows	16	2.1	b	Generated Windows API
winapi	17	2.1	a	Raw FFI bindings for all of Windows API

Figure 4.13: Distribution of features for three *k*-means clusters (a,b, and c)

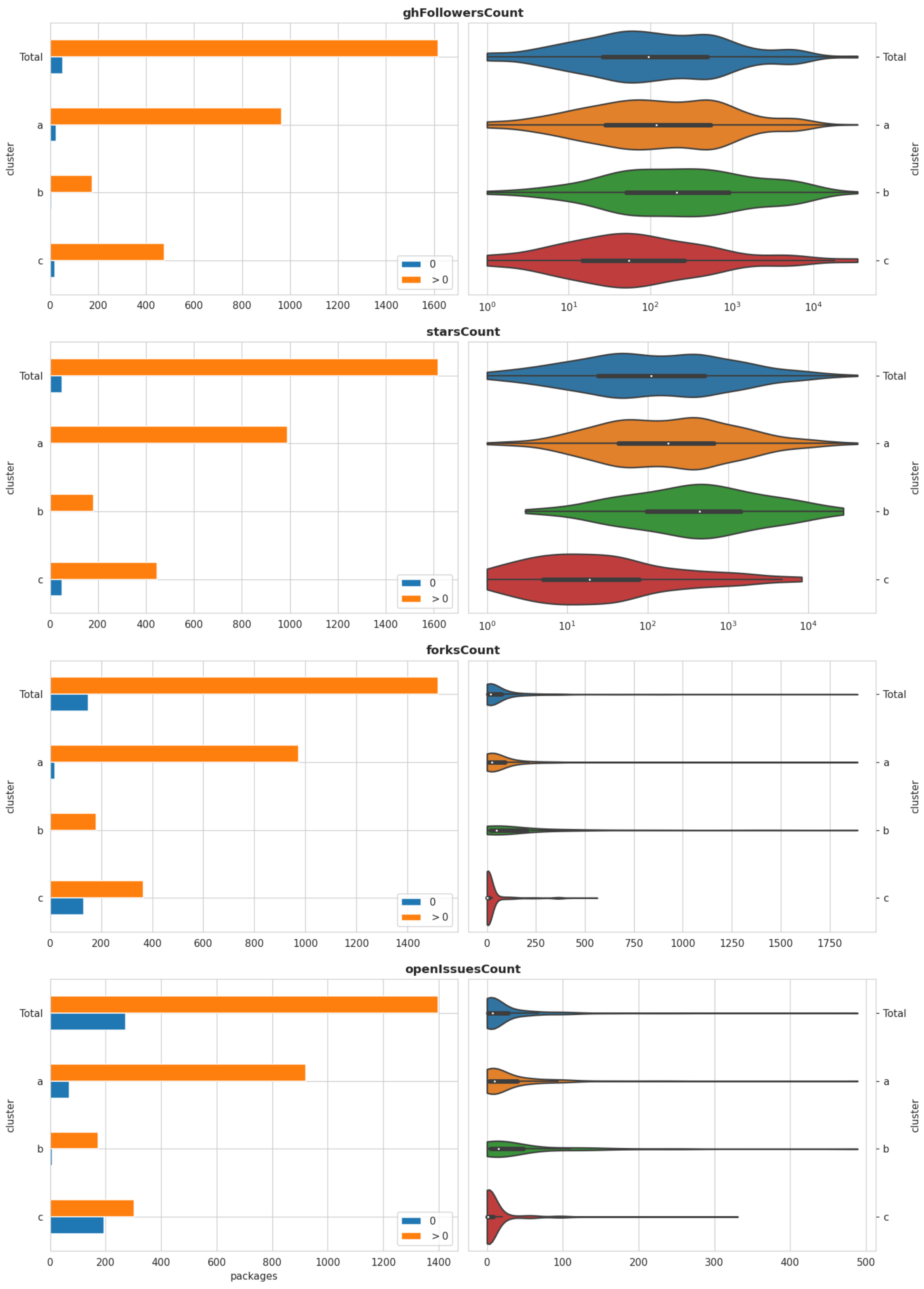


Figure 4.14: Distribution of features for three *k*-means clusters (a,b, and c)

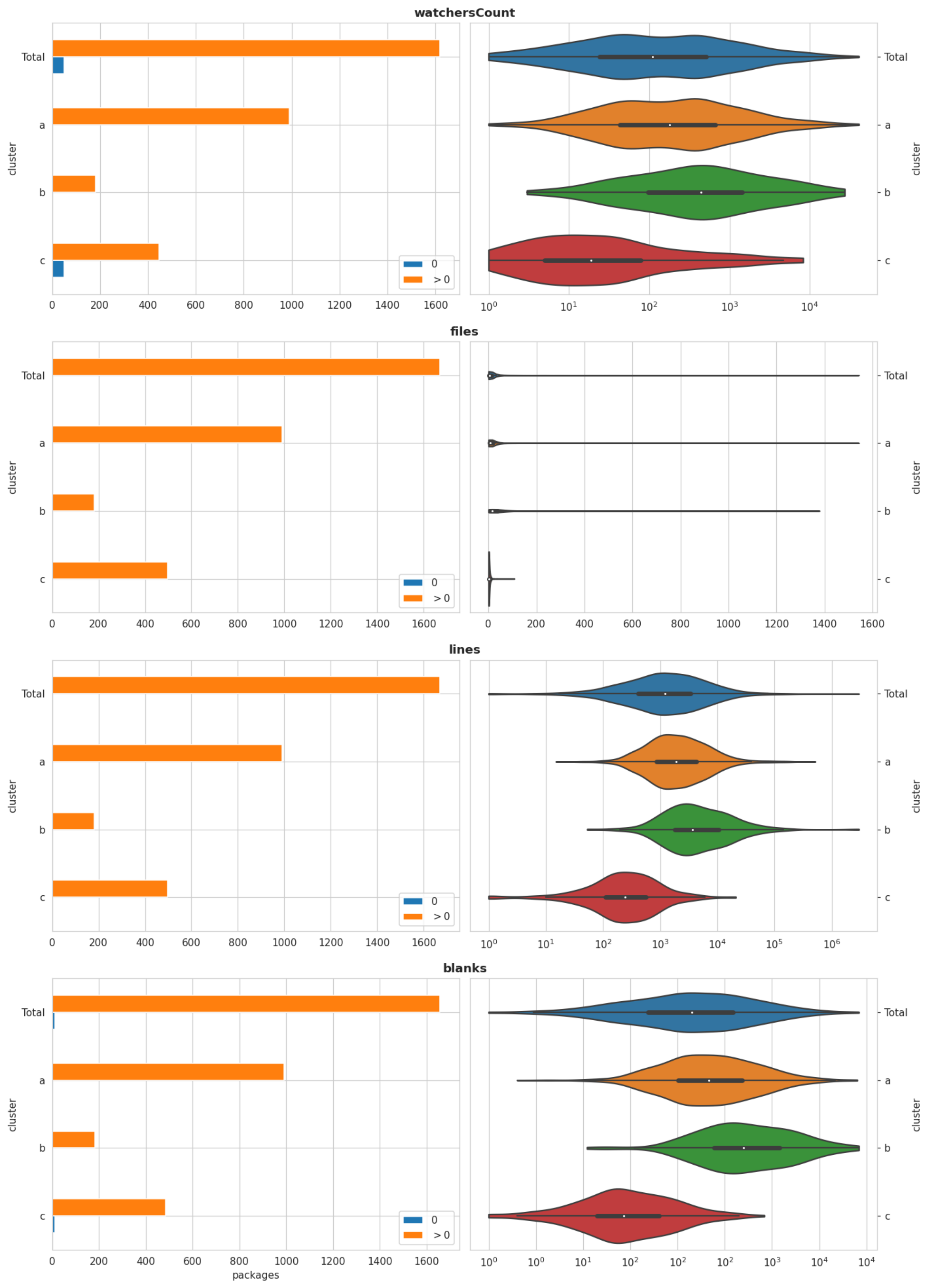


Figure 4.15: Distribution of features for three *k*-means clusters (a,b, and c)

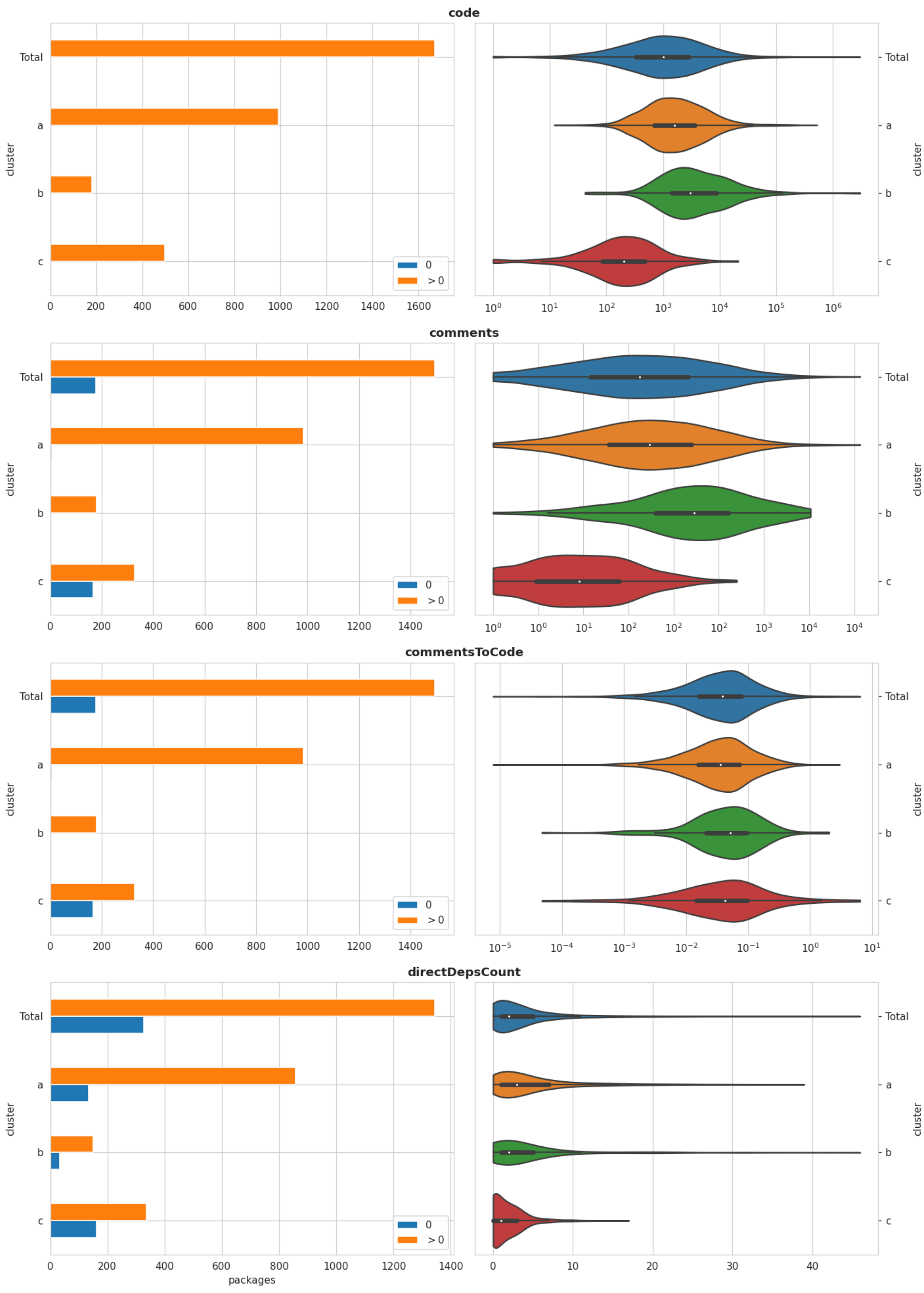


Figure 4.16: Distribution of features for three *k*-means clusters (a,b, and c)

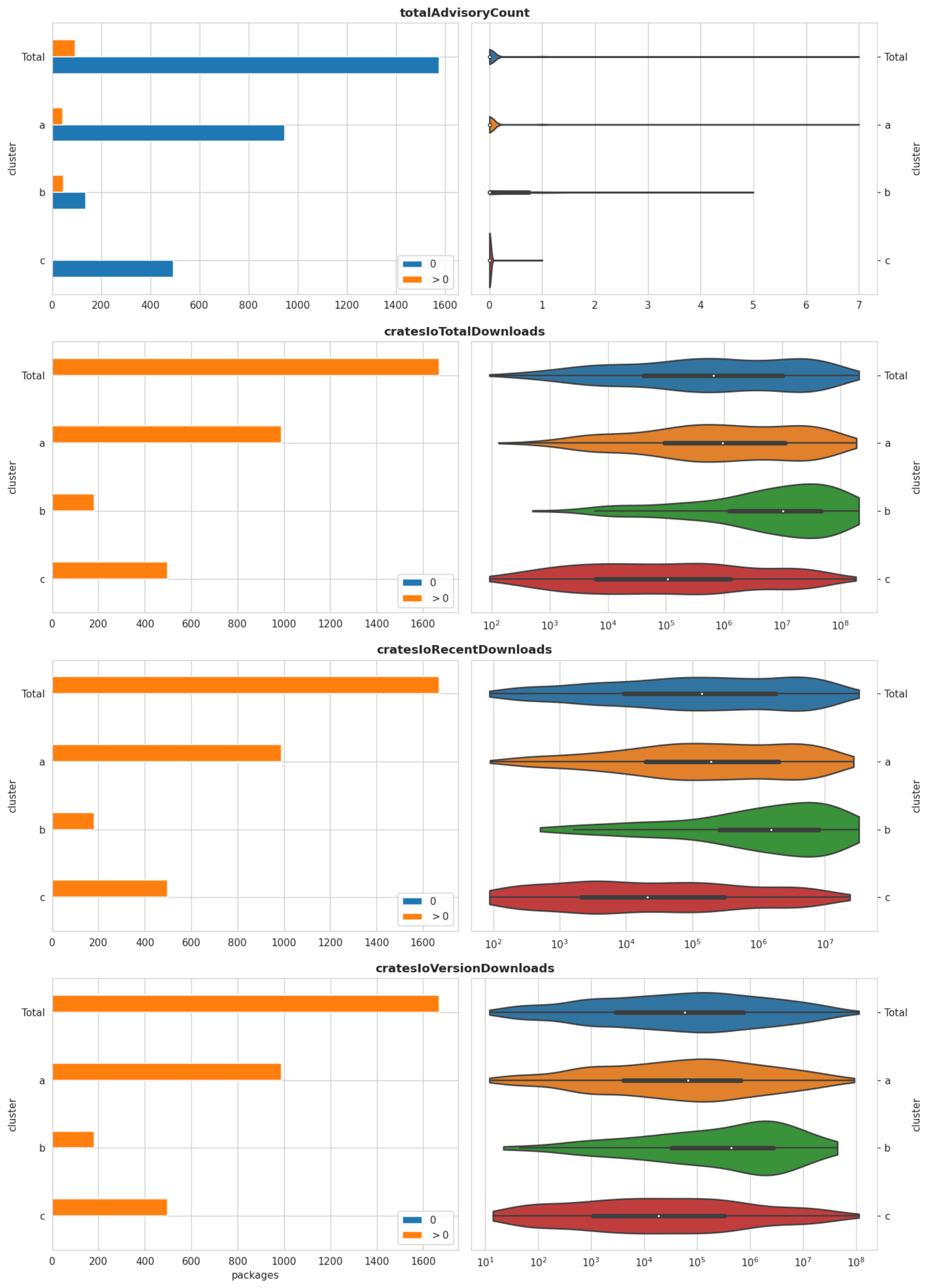


Figure 4.17: Distribution of features for three *k*-means clusters (a,b, and c)

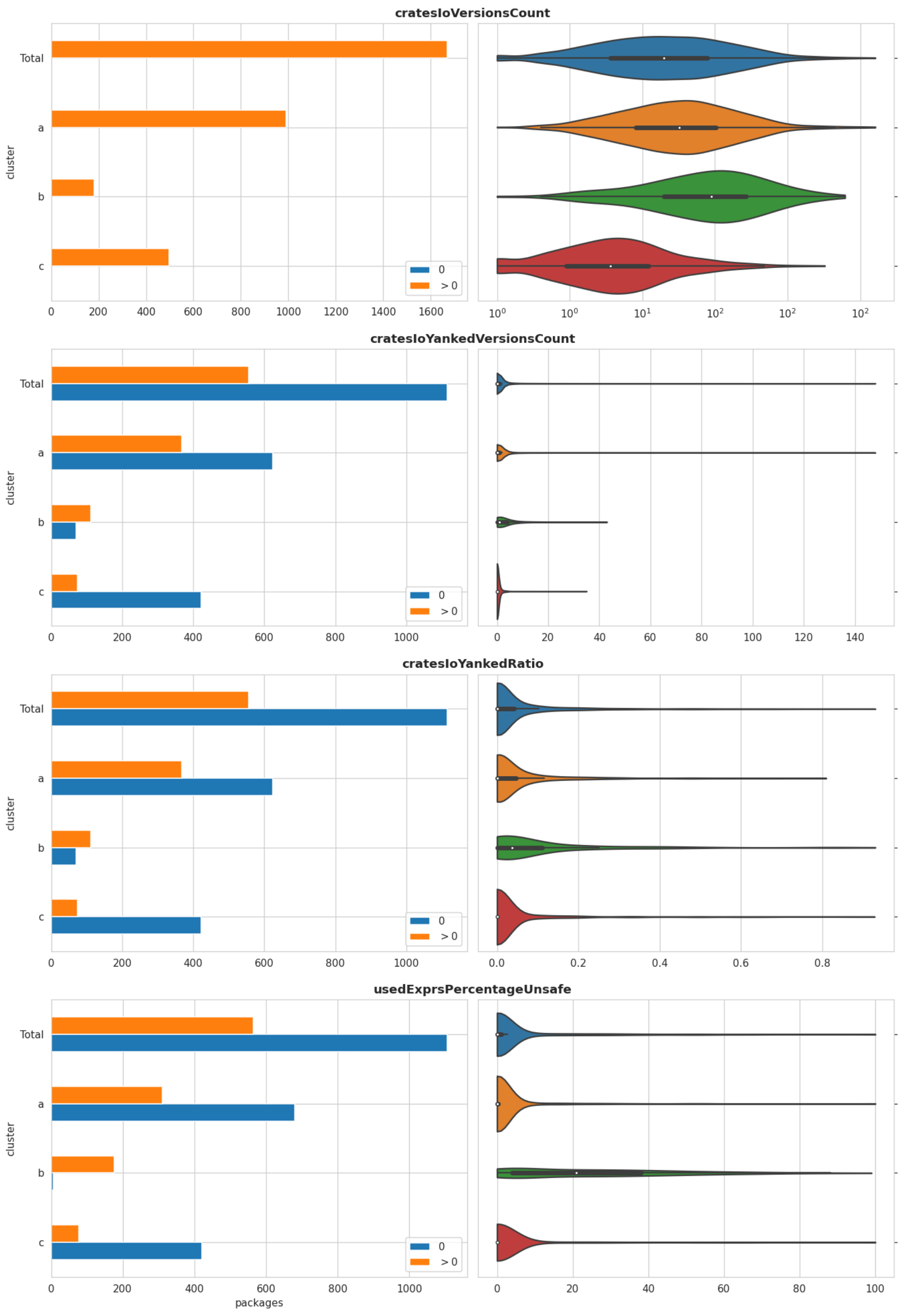


Figure 4.18: Distribution of features for three *k*-means clusters (a,b, and c)

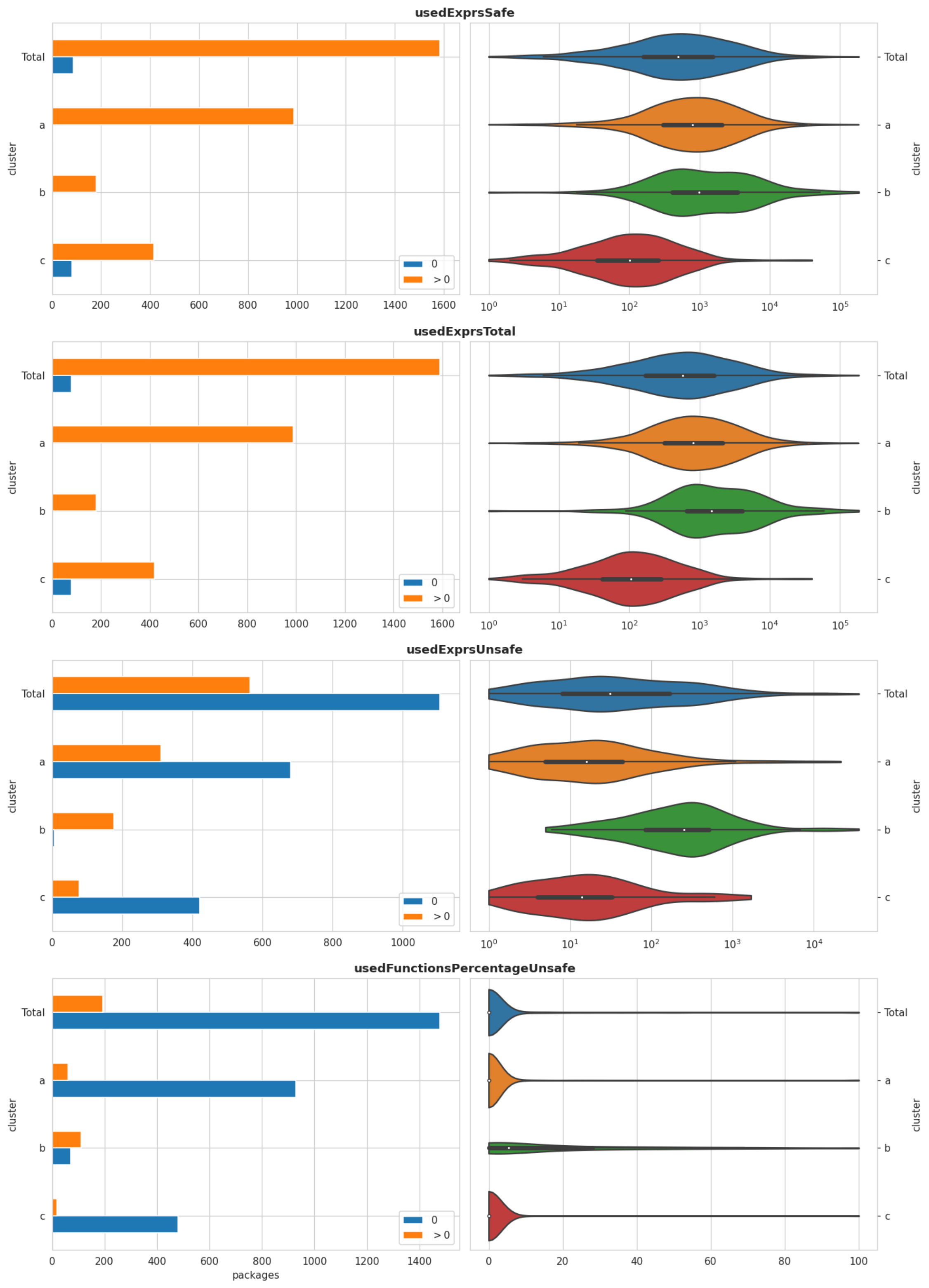


Figure 4.19: Distribution of features for three *k*-means clusters (a,b, and c)

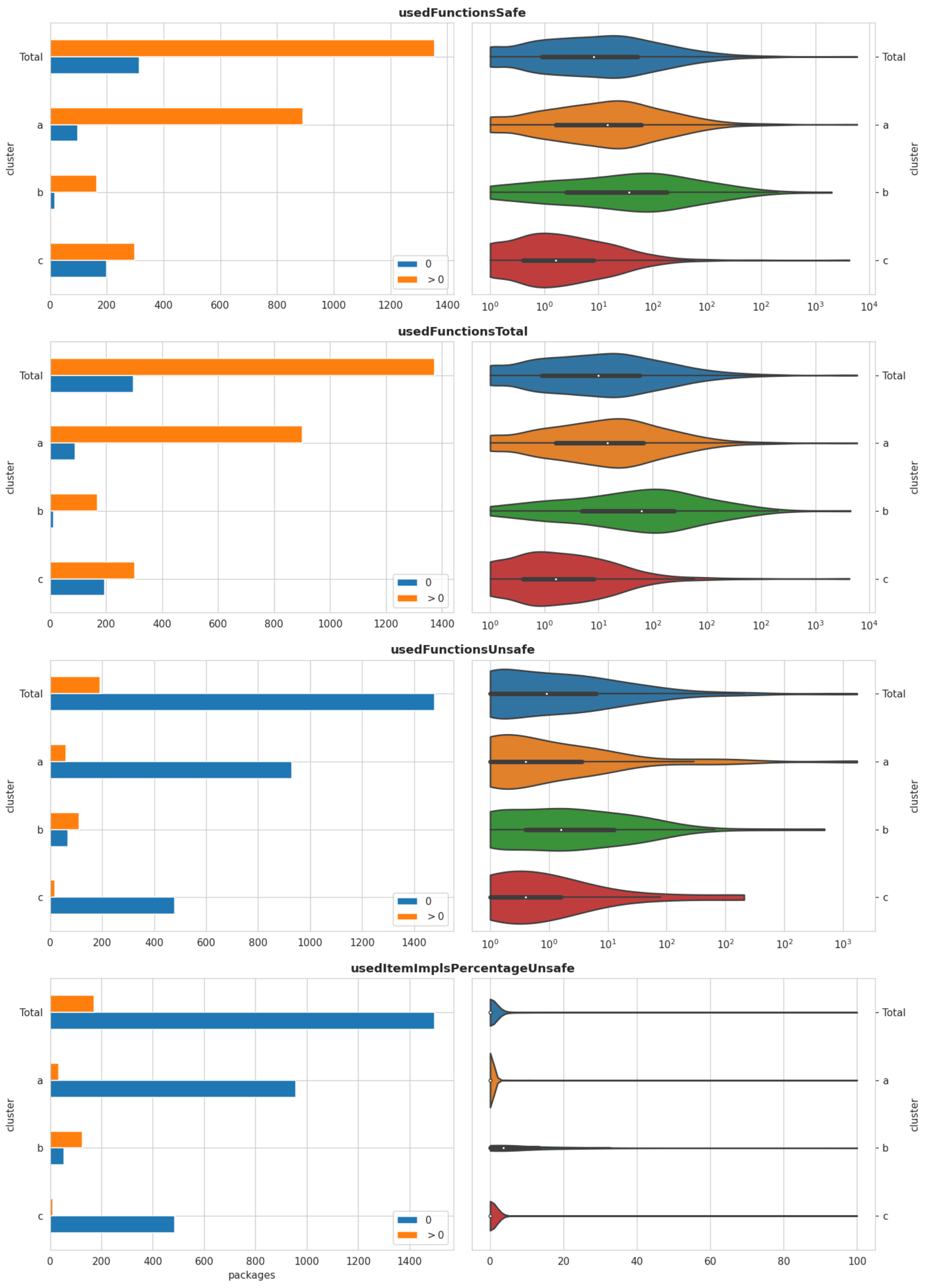


Figure 4.20: Distribution of features for three *k*-means clusters (a,b, and c)

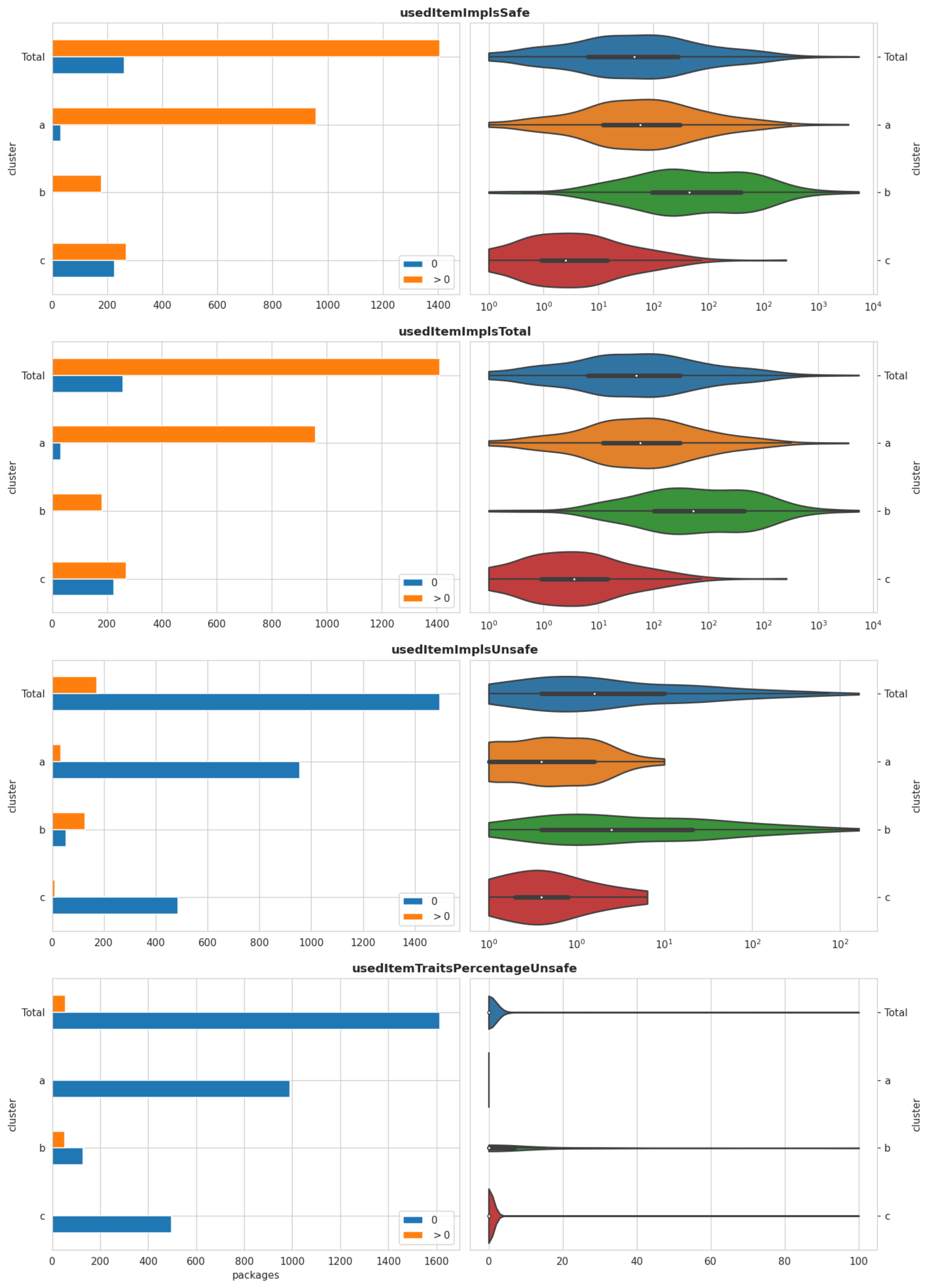


Figure 4.21: Distribution of features for three *k*-means clusters (a,b, and c)

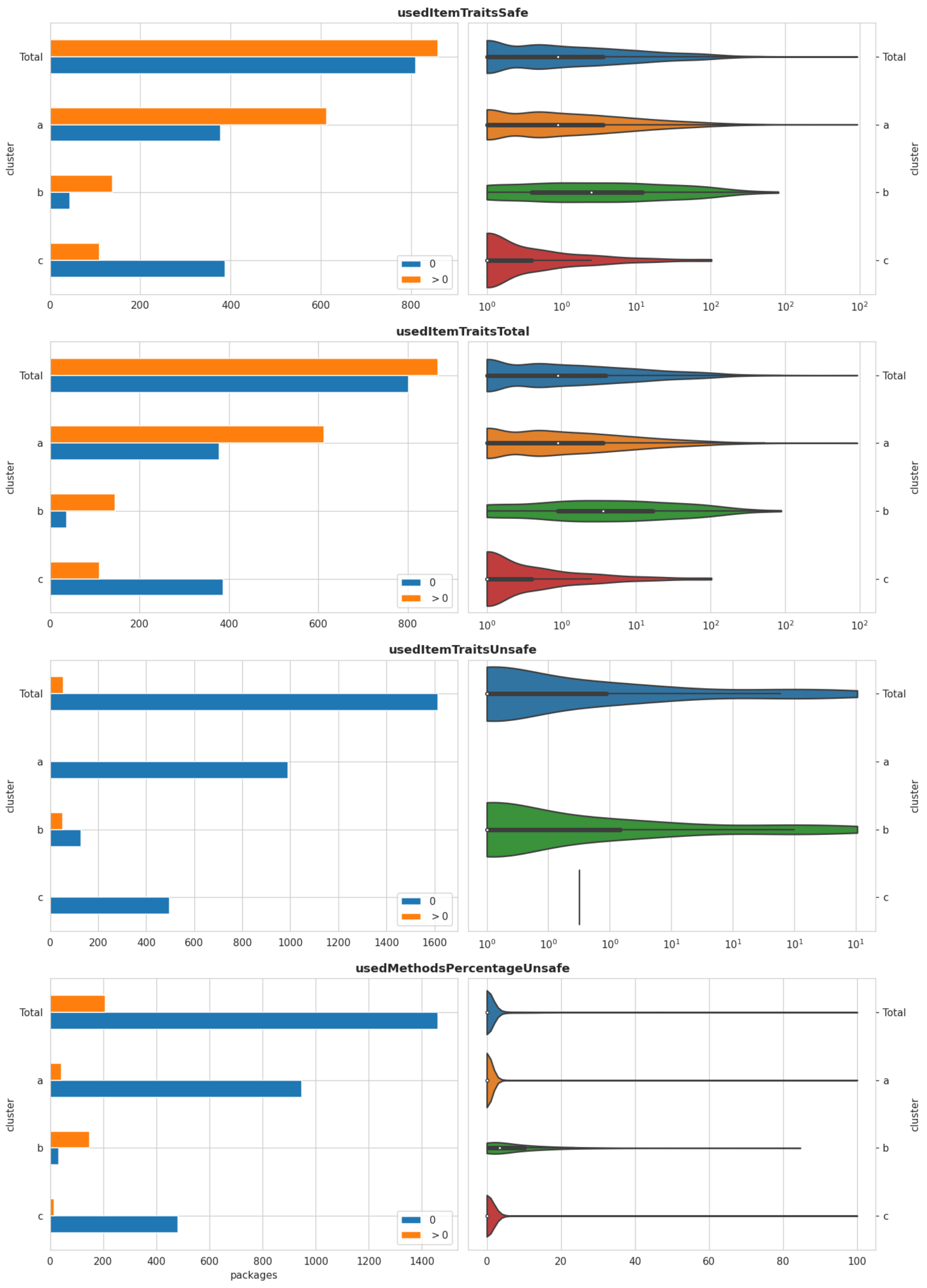


Figure 4.22: Distribution of features for three *k*-means clusters (a,b, and c)

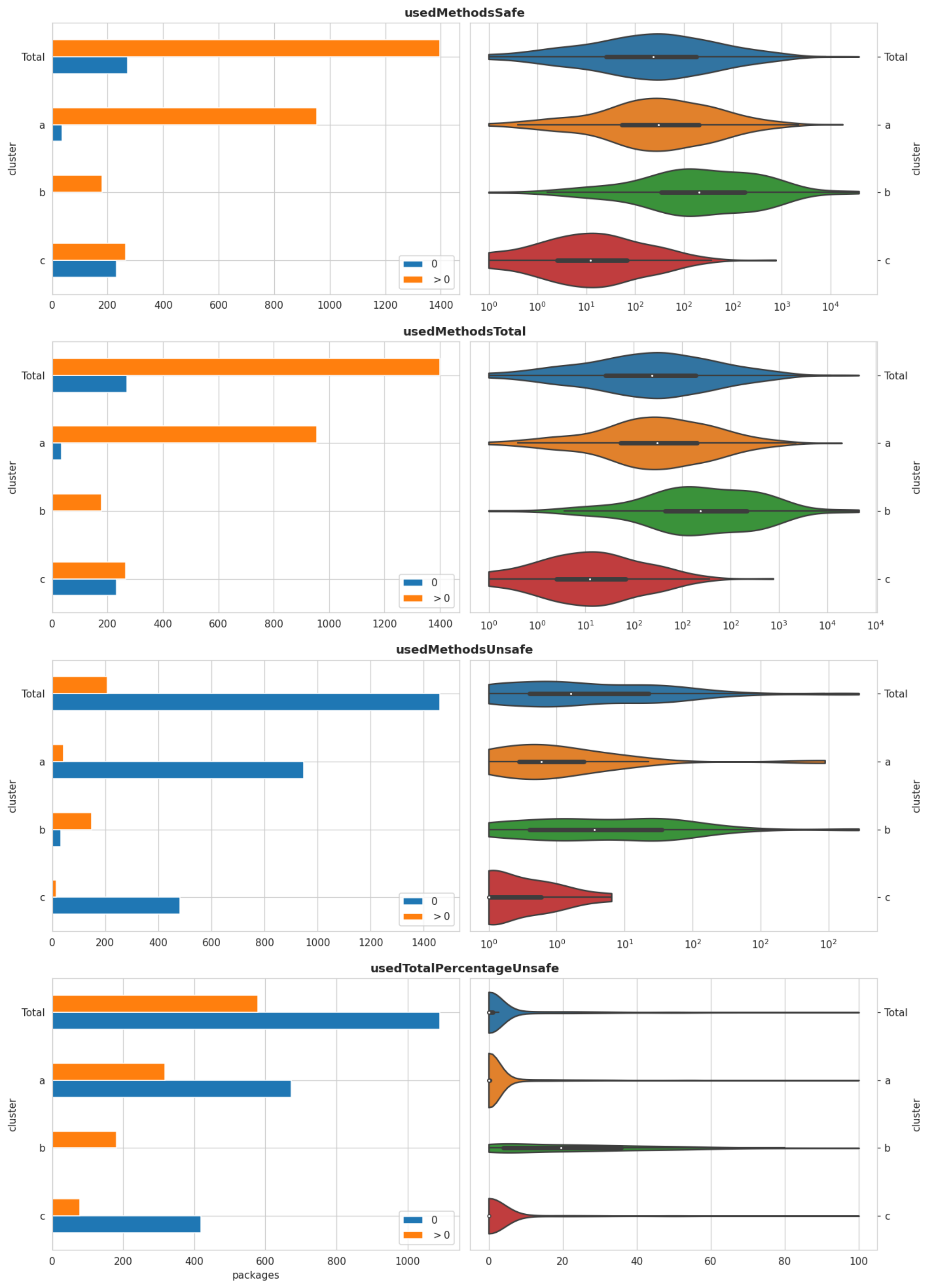
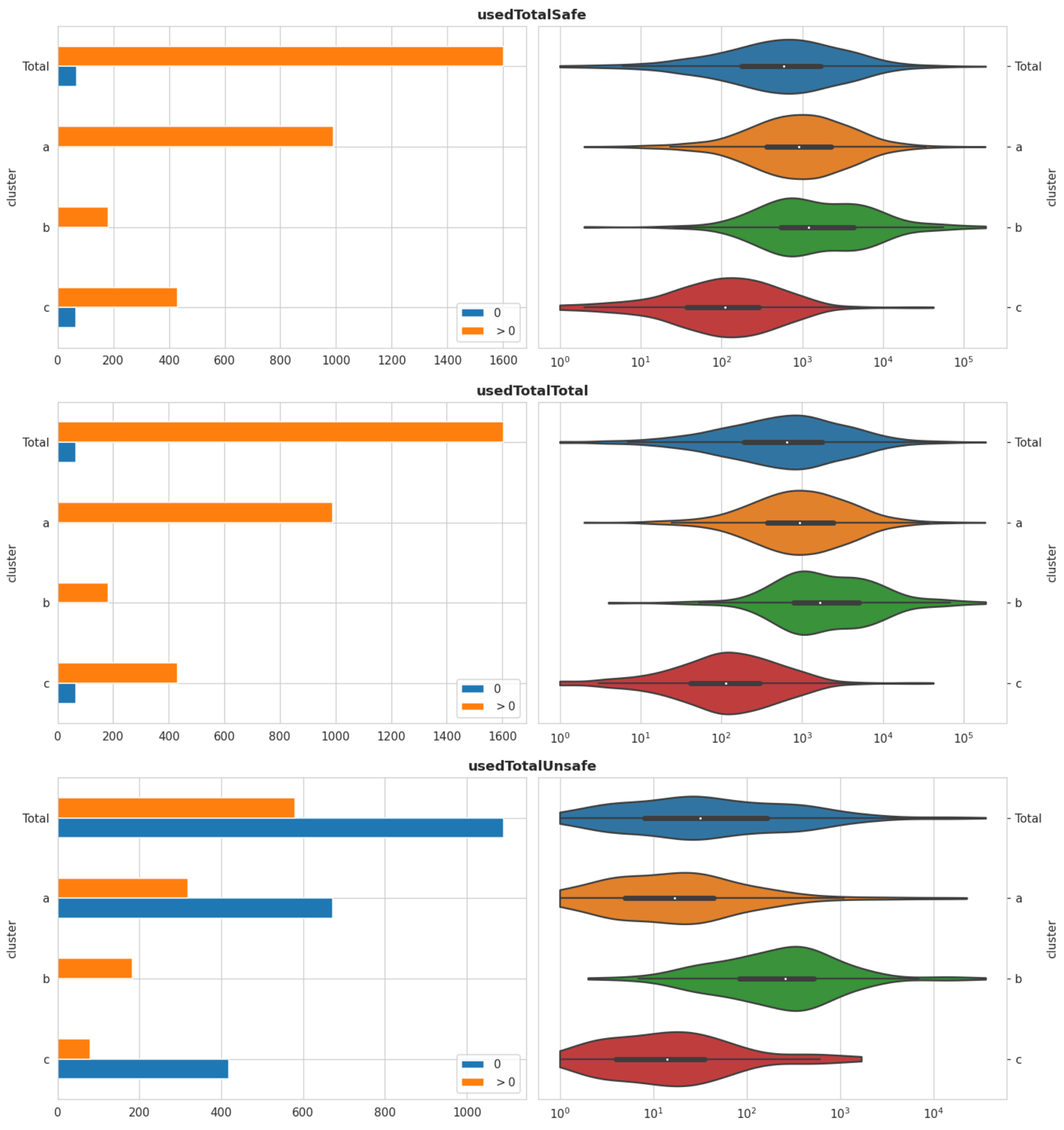


Figure 4.23: Distribution of features for three *k*-means clusters (a,b, and c)



Chapter 5

Discussion

This part will discuss the results presented in the previous chapter, and provides a discussion of implications for `cargo-indicate`. We will also present threats to validity.

5.1 Distribution of Data

To answer **RQ1** (*What signals contain information that can differentiate a package against another package?*), we will look at the distribution of the collected features.

We noted during analysis of the data that several collected features are better described by their logarithm. Some of these, like `code`, show a clearly log-normal distribution. Due to the need for the k-means algorithm to reduce variance in the input features, these features were replaced by their logarithm.

We note that further work should be careful not to use some features directly, but consider using logarithms of their value instead.

We also note that some features, especially regarding unsafe usage, contains a large amount of packages with no presence at all; `usedItemTraitsUnsafe` is one such example. This means that the distribution of projects using unsafe Rust, or features of unsafe Rust, relies on a small sample size. It also means that projects are divided into groups of non-users and users of unsafe Rust.

Also `totalAdvisoryCount` is zero for a vast majority of packages, which makes further analysis of subsets of this data difficult. It also adds to the difficulties already discussed about using vulnerabilities as a signal; The packages reporting them are so few that they may rather describe which authors go through the extra work of reporting them, rather than providing any insight into future vulnerabilities (reported or otherwise). It can even be the case that past advisories may indicate that the project is healthy and is being actively maintained. [18] studied how past vulnerabilities predicted future vulnerabilities in components of major software projects. The authors found empirical evidence that past reported vulnerabilities did *not* predict future vulnerabilities. Based on this assumption, one can argue that projects without vulnerabilities are of interest when identifying dangers in a dependency tree, since past vulnerabilities do not predict future ones, but a *report* of a vulnerability shows investment in the project. This of course relies on that the advisories were not of the kind where the project is reported as unmaintained, however it is possible to write a `cargo-indicate` query that identifies such advisories (such as using a `@filter` directive checking if the advisory description matches `"unmaintained"`). This also relies on the advisory affecting the version in the dependency tree, but in the case where one wants to avoid active advisories being included in dependencies we recommend other projects such as `cargo-deny`[1].

One interesting aspect of the features relating to GitHub popularity, such as `starsCount` and `watchersCount`, is that they appear to have a bimodal distribution. With regards to [7], which tells us that GitHub stars often are a result of marketing, it may be that one distribution is due to the inherent qualities of the projects, while the other describes their marketing. This property, if it does exist, is harder to directly query with `cargo-indicate`, but is something a user should keep in mind as a real possibility.

5.2 Correlations of Signals & Features

To answer **RQ2** (*Do the selected signals correlate in any way, and if so, how?*), we consider Figure 4.10 and Figure 4.11.

Very high correlations can be found between the unsafe Rust features; many of these correlations exist because expressions are the most commonly found type of unsafe usage, and thus correlate highly with the total number of unsafe and safe. We also see correlations between code stats, such as line count, and safe/unsafe usage, which is also to be expected. More lines of code simply mean more code that can be unsafe.

We also see that measures of GitHub popularity, such as stars, forks, and watchers correlate with each other. In fact, `starsCount` and `watchersCount` are found to have a correlation of 1.0 in Table 4.2. This follows the distribution curves found in Figure 4.13 and Figure 4.14. We interpret these findings to mean that these features measure the same thing. As previously discussed, this likely corresponds to GitHub popularity.

Between signals, we see correlations between GitHub popularity (stars, forks) and the amount of downloads on crates.io, which is also to be expected. We can also find some correlation between GitHub popularity and the size of the project (measured in `lines`, `code` etc.). It may be the case that larger projects providing more features are more likely to be popular on GitHub.

We have been unable to find two different signals (not features) that provide close to exactly the same information, and thus find that part of the answer to **RQ4** (*Does combining signals in multiple dimensions provide benefits for analysis of a Rust package?*) is yes, based on that new information can be gained with more of our identified signals. This can be seen in Table 4.2, where the only some features are deemed redundant (correlation of 0.95 or more). We admit that this correlation limit is somewhat arbitrary, but Table 4.2 does not describe any unexpected exclusions based on this limit. We do not believe that minor changes to this value would significantly affect our results. It could be the case that a lower value reduces the risk of similar features measuring the same thing weighing more in the analysis, but we find this unlikely. One must also balance this gain against the risk of removing values that have

[1]https://crates.io/crates/cargo-deny

a very high correlation by chance. Further work could improve upon this by using several different measures of correlations. A Spearman correlation of 1.0 does not necessarily mean that using both values is a bad practice for all applications.

5.3 Clusters & Outliers

The clustering described in Figure 4.13 to Figure 4.23, and in Table 4.4 and Table 4.3 only provides the cluster names `a`, `b` and `c`. To provide better understanding of how we can describe Rust projects, and to answer **RQ3** (*Can we find clusters of Rust packages based on their signals?*), we will attempt to rename them according to their characteristics. We summarize the characteristics in Table 5.1, based on Figure 4.13 to Figure 4.23:

Table 5.1: Summary of characteristics for each cluster identified via *k*-means

Characteristic	`a`	`b`	`c`
GitHub Repository Owner	Bimodal and average	More popular owners	Less popular
GitHub Repository Popularity	Bimodal and average	More popular	Less popular
Code Stats	Average size	Greater than average size	Less than average size, only cluster with projects without comments
Direct Dependency Count	Average	Average	Less than average
Advisory Count	Small amount	Somewhat higher	Almost none
crates.io	Multimodal and average downloads	Most downloaded projects	Spread out, but more projects with few downloads
Yanked Versions	Average	More than average	Less than Average
Unsafe Usage	Average	All projects at least some unsafe, more unsafe	Few projects with unsafe, and those are not very unsafe

We note that not all features presented in Figure 4.13-4.23 were used to find the clusters; Features above a threshold of 0.95 correlation were removed, since they likely measure the same metric in different ways. This can make one metric outweigh others. These pairs are listed in Table 4.2.

While this summary is not perfect in mathematical rigor, it gives us a language to now rename our clusters. We summarize them here:

1. a/average - This cluster is close to the average for all packages in all regards, and it is also the largest cluster
2. b/popular-unsafe - Packages in this cluster have higher values in all regards; They are more popular, they use more unsafe, they yank more versions, and have more advisories

3. c/small-uninvested - Packages in this cluster is the opposite of b/popular-unsafe, and place lower than average in all metrics. Noteworthy is that they don't necessarily have less downloads on crates.io, even though they are less popular on GitHub

We also saw an interesting characteristic of c/small-uninvested; it is the only cluster that contained projects with no comments. At first glance we suspected this to be due to the cluster containing generated packages. This suspicion is based partly on Table 4.4, which reveal generated Windows packages that belong to this category. While it is possible to write a whole package without comments, it is unlikely to be done by humans. However, when we used the pandas `sample` method on the cluster, with a sample of 20 packages out of 497 packages in the cluster, we found no packages that described themselves as generated. When we instead sampled as many packages out of the 168 packages with no found comments, the answer quickly became obvious upon manual inspection: The packages are small library crates, with generally one function or macro implementation. The reason that no comments are found is that all information is contained in Rust doc comments[2], information that is used to generate Rust documentation automatically using `cargo`. This reveals a need to be able to count and quantify these documentations strings as well, in addition to traditional comments.

We want to make it clear that these names are not meant to pinpoint some projects worse than other; It is very possible that the community around a c/small-uninvested package is heavily invested and that the code is of the highest quality. It might also be that this cluster contains, sometimes generated, libraries that are heavily relied on, but that lacks the marketing capabilities (or will) of other packages.

We believe that this analysis adds to the answer of **RQ4** (*Does combining signals in multiple dimensions provide benefits for analysis of a Rust package?*); when looking at clusters of Rust projects, we can see trends across multiple dimensions of features.

We must note a current occurrence in the Rust ecosystem here; The Windows API crates that can be seen in Table 4.4 are either official Microsoft support (like `windows`[3]), or they are part of a project to collect Windows FFIs in one project (`winapi`[4]).

In answering **RQ5** (*Are Rust packages of the same category likely to have the same signal characteristics, and if so, what are they?*), we look at Table 4.3 to see how top-level categories correlate to cluster placement. As we expect a majority of available categories (58.13%) belong to the a/average cluster, while c/small-uninvested is at 30.97% and b/popular-unsafe is the smallest cluster at 10.90%. Note that this table describes categories, and not packages (since a package may have multiple categories). We find that the way the Rust ecosystem allows for arbitrary choice of categories makes the analysis harder and the some categories very limited in usefulness. On the other hand it removes constraints, and developers can simply choose to use already popular category names. The only category where b/popular-unsafe is in a significant majority is the `memory_management` category, while c/small-uninvested is dominant in the `accessibility` category, described by crates.io as *Assistive technology that helps overcome disabilities and impairments to make software usable by as many people as possible.* c/small-uninvested also contains a majority for `external_ffi_bindings`. This is a category where one could expect a lot of utility, while it is not as marketable as other projects.

We must also remember that these are renamed categories; only the top category named

[2]Prefixed with `///` or `//!` instead of `//`

[3]https://web.archive.org/web/20230519001001/https://github.com/microsoft/windows-rs

[4]https://web.archive.org/web/20230414074121/http://github.com/retep998/winapi-rs

was used. Since project can have any number of categories, single packages can also be included in many rows in this Table 4.3.

RQ6 (*How can we use signals to identify Rust packages in need of manual developer attention?*) has been hinted at throughout the discussion, and can be summarized as the following statement: *It depends on what the developer is looking for.* However, we will conclude by discussing Table 4.4 describing the identified outliers, which have a Mahalanobis distance that are 3 or more standard deviations from the mean distance for all packages.

One downside with this list is the lack of context, since the Mahalanobis distance does not explain what makes these packages stand out. We do however see a possibility of reusing this strategy when analyzing a dependency tree. It would be of interest to look firstly at the dependencies that deviate the most from the others, but it can also be the case that a developer is only interested in popularity vs. unsafe usage for example. In that case a query could select interesting features, which are then compared to a known distribution of those features.

We note that we set out to use t-SNE as an alternative in visualizing clusters, but found the results to be hard to interpret and use. They have thus been left out of the results and discussion, but we encourage further attempts in cluster analysis of all kinds for similar sets of data.

5.4 Implications for `cargo-indicate`

In answering our research questions, we have also touched on the subject of using `cargo-indicate`, and what to make of its results. We find that `cargo-indicate` provides a useful utility in aggregating and describing several factors in the Rust ecosystem, as way as providing an interface to query these factors. However, we want to caution users against using the features without context, and to not set hard numerical limits. Instead, we believe that further research is needed to establish baselines of values for different kinds of projects. There seems to be a difference in the types of projects and what to expect in results. It is also the case that some values should be normalized before comparison, such as using the logarithm of the values.

We also note that some features of the same signal are close to identical, in accordance to Table 4.2. Using GitHub popularity is a metric that holds value, but measuring both stars and watchers provides no added benefit. The same is true for several unsafe metrics, as well as crates.io downloads.

We also do not recommend using the RustSec Advisory Database as a metric to decide if the dependency tree is "safe", since advisories currently are rare and for many types of projects nonexistent. It is thus unlikely that even if problems exist that they are reported. It is possible that this will change in the future as the database grows in size and popularity.

5.5 Threats to Validity

Here we will present what we identify as threats to the validity of results and conclusions. The section is subdivided into sources of uncertainty.

5.5.1 Underlying Tools

`cargo-indicate` relies on third-party tools to provide its functionality, and thus the results are only as good as these tools. This comes with the benefits that future updates to these tools will benefit `cargo-indicate`, but also means that our dataset may be affected. We cannot guarantee of the dependencies, neither can we of their dependencies (which is interestingly a problem we set out to solve).

`cargo-geiger` is likely the most advanced of the tools, and according to us holds the highest risk of not providing accurate results in all cases. While other tools rely on third-party APIs and files, `cargo-geiger` must handle all ways a Rust package may use unsafe. We found several examples where this was not the case, such as with the `libc` crate. It provides an unsafe interface with the C library, but its reliance on macros makes it harder to detect properly. However, we find that the analysis is still good enough to provide information about unsafe usage on a larger scale, and thus provides insight in this metric.

5.5.2 Validity of the Dataset

The dataset we chose has several benefits, and some downsides. While using popular packages of several categories does ensure that different types of projects, that are also actively used, they may paint a different picture than the whole crates.io registry would. Thus, our conclusions are based on popular packages, where *popular* is relative to the popularity of the included categories. On the other hand, this is likely to exclude unmaintained or empty codebases.

An increase of the sample size would benefit the analysis, but one may also raise the objection that behavior present in popular packages are more desireable than unpopular ones, as popular packages are more likely to appear in a dependency tree analyzed by `cargo-indicate`.

Another issues may be that a type of package is missed by `cargo-indicate`, or the queries we used to collect our data. This can be for any number of reasons, such as failure of underlying tools, unexpected panics, or simply missing data. For example, we found that many popular packages where results were not available was `tree-sitter`-bindings. Tree-sitter is a parser generator tool and an incremental parsing library[5], and we found several bindings prefixed with `tree-sitter-` among projects without results. We were not able to decide why these projects in particular failed.

`cargo-geiger` also provides a challenge in regards to providing results for all targets. Since the tool relies on being able to compile target packages, packages which may require third-party libraries may fail to compile. This means that our dataset for unsafe data may contain a bias against packages that require a specific developer environment. While we attempted to rectify this by installing common requirements, it is possible that some packages with a specific requirement (e.g. a specific C library) are not included. Further work can be more thorough in providing a "complete" developer environment for data collection, and more actively monitor compilation failures. Package configurations are not trivial, and we have ignored the concept of *features* in our analysis. This could be a point of interest in further studies; to look into the usage of features in the ecosystem.

[5]https://web.archive.org/web/20230517055424/https://tree-sitter.github.io/tree-sitter/

5.5.3 Selection of Number of Clusters

In this work we set the amount of clusters in k-means to $k = 3$, in part due to the perceived ease of describing the clusters present when this is the case. While the selection of k has been described as more of an art than a science [22], we cannot escape the possibility that this choice is unwise. During our analysis, we did not find $k = 4$ to provide any benefit for our analysis, but it is possible that another value for k might be chosen in a way that provides a more fine-grained analysis. We attempted using even more clusters, for example $k = 7$ and $k = 9$, but these were even harder to analyze than $k = 4$. While this may be the case, we do not consider this to be detrimental to our work. We have provided a suggestion for clustering with a description of characteristics, and find these abstractions to be useful in understanding the data set.

Chapter 6
Conclusion

This chapter will provide a summary of our contributions, and suggest areas of further work.

6.1 Summary of Contributions

We summarize our contributions in this thesis as the following:

- We created a tool `cargo-indicate` capable of executing complex GraphQL queries on Rust packages and their dependency trees, using several parts of the Rust ecosystem and the code itself as signals
- We collected a dataset using this tool from over 2000 Rust packages
- We described the distributions of some of the collected data
- We found GitHub popularity features to have a bimodal distribution, and hypothesize that this is due to marketing, in addition to more natural project growth
- We found past advisories from `advisory-db` to be a bad measure of future issues due to its limited prevalence, but that it might be a possible good measure of investment in a project
- We described which features correlate and how
- We identified outliers in the set of packages collected using Mahalanobis distances
- We used k-means to classify Rust packages in three clusters
- We analyzed the characteristics of these clusters and described them
- We gave recommendation for potential users of `cargo-indicate` based on our findings

6.2 Further Work

We also believe that a more advanced pipeline could be created, where `cargo-indicate` collects information about a package, that is then set in a context using for example Python that plots its stats on a distribution. We also see the possibility of providing advanced graphs of a dependency tree that could preferably be visualized using another application, similar to deps.dev.

We believe that the general structure of `cargo-indicate` provide several benefits, such as a simple interface that is not implementation or even code-dependant in the form of a GraphQL schema. Further developments, especially in the `trustfall` query engine itself, will directly benefit `cargo-indicate`.

A point of interest is the connection between marketing and popularity on websites such as GitHub. Future studies could analyze marketing for a project on for example social media, developer forums etc. and compare this to the growth in popularity for projects. Another interesting possibility would be to investigate if there exist adverse influence operations, or "fake news", in these contexts; Would it be possible to use marketing to entice developers to use malicious dependencies? Or where the project owners have malicious intent, and intend to publish compromised updates in the future?

There is also the possibility of using Machine Learning to identify projects that are of interest. Here `cargo-indicate` could provide a useful way of collecting information. Difficulties in this regard is the difficulties of labeling projects, but it could be the case that collections of human reviews (such as provided by `cargo-crev`) would aid this endeavour.

References

[1] RFC 1105. https://web.archive.org/web/20230130111106/https://rust-lang.github.io/rfcs/1105-api-evolution.html. Accessed: 2023-02-22.

[2] The Rustonomicon. https://web.archive.org/web/20230108044930/https://doc.rust-lang.org/nomicon/intro.html. Accessed: 2023-02-22.

[3] Rustsec advisory database github repository. https://web.archive.org/web/20230416153313/https://github.com/RustSec/advisory-db/. Accessed: 2023-04-21.

[4] scikit-learn documentation. https://web.archive.org/web/20230425231141/https://scikit-learn.org/stable/modules/preprocessing.html. Accessed: 2023-04-27.

[5] Mahalanobis distance. Encyclopedia of Mathematics, 2023. Accessed on: May 22, 2023.

[6] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):136:1–136:27, November 2020.

[7] Hudson Borges and Marco Tulio Valente. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software*, 146:112–129, December 2018. arXiv:1811.07643 [cs].

[8] Jonathan Corbet. A first look at rust in the 6.1 kernel, Oct 2022. Accessed: 2023-05-22.

[9] Roland Croft, M. Ali Babar, and Li Li. An investigation into inconsistency of software vulnerability severity across data sources. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 338–348, 2022.

[10] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is rust used safely by software developers? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 246–257, Seoul South Korea, June 2020. ACM.

[11] Jesus M. Gonzalez-Barahona, Daniel Izquierdo-Cortazar, and Gregorio Robles. Software Development Metrics With a Purpose. *Computer*, 55(4):66–73, April 2022.

[12] Florian Hahn. Rust2Viper: Building a Static Verifier for Rust. 2016. Medium: application/pdf,1 Online-Ressource Publisher: ETH Zurich.

[13] IT Jolliffe and J Cadima. Principal component analysis: a review and recent developments. *Philos Trans A Math Phys Eng Sci*, 374(2065):20150202, 2016.

[14] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel German, and Daniela Damian. The promises and perils of mining github (extended version). *Empirical Software Engineering*, 01 2015.

[15] Steve Klabnik and Carol Nichols. *The Rust Programming Language, 2nd Edition*. No Starch Press, USA, 2022.

[16] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2183–2196, Virtual Event Republic of Korea, November 2021. ACM.

[17] Peiming Liu, Gang Zhao, and Jeff Huang. Securing UnSafe Rust Programs with XRust. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 234–245, October 2020. ISSN: 1558-1225.

[18] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 529–540, New York, NY, USA, October 2007. Association for Computing Machinery.

[19] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 763–779, New York, NY, USA, June 2020. Association for Computing Machinery.

[20] Eric S. Raymond. *The Cathedral & the Bazaar*. O'Reilly Media, Inc., Feb 2001.

[21] William Schueller, Johannes Wachs, Vito D. P. Servedio, Stefan Thurner, and Vittorio Loreto. Evolving collaboration, dependencies, and use in the Rust Open Source Software ecosystem. *Scientific Data*, 9(1):703, November 2022. Number: 1 Publisher: Nature Publishing Group.

[22] Rachel Schutt and Cathy O'Neil. *Doing Data Science: Straight Talk from the Frontline*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA, USA, 2014.

[23] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, November 2011. Conference Name: IEEE Transactions on Software Engineering.

[24] Martin Wattenberg, Fernanda Viégas, and Ian Johnson. How to use t-sne effectively. *Distill*, 2016.

[25] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead, January 2023. arXiv:2301.05362 [cs].

[26] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael Lyu. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs, February 2021. arXiv:2003.03296 [cs].

[27] Nikolas Zöller, Jonathan H. Morgan, and Tobias Schröder. A topology of groups: What GitHub can tell us about online collaboration. *Technological Forecasting and Social Change*, 161:120291, December 2020.

Appendix A
Descriptions of Package Features

Table A.1: Description of features used in data analysis, values for which their logarithm was used for analysis marked with †

	Description	*Notes*
ghUnixCreatedAt	Timestamp of when the GitHub user owning the repository was created	
ghFollowersCount†	Amount of followers of GitHub repository owner	
starsCount†	GitHub repository stars	
forksCount	GitHub repository forks	
openIssuesCount	Open GitHub issues	GitHub's API counts both issues *and* pull requests as issues; This is the sum of both
watchersCount†	GitHub repository watch-ers	
files	Number of files in the `src/` directory	Only Rust
lines†	Number of lines in the `src/` directory	Only Rust
blanks†	Number of blank lines in the `src/` directory	Only Rust

Continued on next page

Table A.1: Description of features used in data analysis, values for which $\log_{10}$ was used marked with †

	Description	*Notes*
code†	Number of code lines in the `src/` directory	Only Rust
comments†	Number of comment lines in the `src/` directory	Only Rust, does *not* include doc comments
commentsToCode†	**comments / code**	Only Rust
directDepsCount	Number of direct dependencies	Only normal dependencies (listed under `[dependencies]`)
totalAdvisoryCount	Total number of advisories in RustSec's `advisory-db`	Does *not* include withdrawn advisories
cratesIoTotalDownloads†	Total downloads for the package with this name on crates.io	
cratesIoRecentDownloads†	Recent downloads for the package with this name on crates.io	*Recent* as listed by crates.io
cratesIoVersionDownloads†	Total downloads for the package version with this name on crates.io	Version is dependent on the `Package` vertex the `CratesIoStats` is connected to
cratesIoVersionsCount†	Number of released versions for the package with this name on crates.io	
cratesIoYankedVersionsCount	Number of yanked versions for the package with this name on crates.io	
cratesIoYankedRatio	*#yanked versions / #versions*	
used<*item*>PercentageUnsafe	Percentage of used *`<item>`* that are unsafe according to `cargo-geiger`	*Used* refers to used by the root package
used<*item*>Safe†	Used safe *`<item>`* according to `cargo-geiger`	*Used* refers to used by the root package
used<*item*>Total†	Total used *`<item>`* according to `cargo-geiger`	*Used* refers to used by the root package
used<*item*>Unsafe†	Used unsafe *`<item>`* according to `cargo-geiger`	*Used* refers to used by the root package
usedTotalPercentageUnsafe	Percentage of used code items that are unsafe according to `cargo-geiger`	*Used* refers to used by the root package

Continued on next page

Table A.1: Description of features used in data analysis, values for which $\log_{10}$ was used marked with †

	Description	*Notes*
usedTotalSafe†	Total used safe items according to `cargo-geiger`	*Used* refers to used by the root package
usedTotalTotal†	Total used items according to `cargo-geiger`	*Used* refers to used by the root package
usedTotalUnsafe†	Total used unsafe items according to `cargo-geiger`	*Used* refers to used by the root package

Appendix B

Further `cargo-indicate` Implementation Details

B.1 Implementing `BasicAdapter`

Implementations of the methods in the `BasicAdapter` trait, as listed in Section 3.2.2, was solved using a Rust `match`-statement, similar to `switch`-cases in some languages, matching on a string tuple. For example, in the `match`-statement of `resolve_property` for the schema in Listing 3.7, the `("User", "age")` tuple would be used to retrieve an iterator over the age of a user. Using this convention allows for leveraging the powerful pattern-matching system of Rust to allow for inheritance in the schema; in Listing B.1 the `age` property is required in the `Mortal` interface, implemented by both `NonUser` and `User`. To resolve the `age` property for both `User` and `NonUser` in the same way, the `("User" | "NonUser", "age")` pattern can be used.

Listing B.1: Inheritance in a GraphQL schema

```
interface Mortal {
    age: Int!
}

type NonUser implements Mortal {
    age: Int!
}

type User implements Mortal {
    name: String!
    age: Int!
}
```

While it is not necessary for these methods to be implemented in this way, it provides a clean mapping between schema and Rust code. Unfortunately, this implementation does not fully exploit the pattern matching restraints in Rust. Rust requires `match`-statements to be *exhaustive*, i.e. all possible patterns must be matched. Since the caller of the `BasicAdapter` methods (the `trustfall` engine) guarantee that the strings provided must be defined in the schema, the developer can be sure that as long as their implementations map 1:1 to their schema, the pattern will be exhaustive, and adding a catch-all pattern at the end can call the `unreachable!` macro (indicating error in the implementation).

For this reason, it was decided to keep a schema file (`schema.trustfall.graphql`) in the library source folder of the project. This file is to act as a single source of truth, and any deviance from the behavior described in it is to be considered a bug.

B.2 Adapter-Client Relationship

Since signal vary significantly in their requirements and limitations, the implementation details vary. For some signals, such as GitHub, the repository will be the same for all versions of a particular package, so the `Vertex::GitHubRepository` variant holds an atomically counted reference to a full repository response from the GitHub API (`Arc<GitHubRepository>`).

In fact, it is possible to divide the work done by the adapter and the clients differently; If the vertex variant contains a reference to the full data (as in the case with `Vertex::GitHubRepository`), the adapter must convert this data to properties itself. If the variant instead hold a unique identifier, it can request a (possibly cached) value from the client instead. In this case the client may do much of the computation, and the graph resolution simply uses the graph as a source of identifiers to be computed (repository links to repository information, name-version tuples to crates.io information etc.). To summarize: We can either store a reference to data in a `Vertex` and let the adapter do computations, or we can store a unique identifier in the `Vertex` and let the client use this information to return the desired value. Variants of this relationship can also be considered, and further additions to `cargo-indicate` should consider ease of implementation, performance, as well as other factors when deciding which model to use.

Appendix C
`cargo-indicate` Schema

Listing C.1: The full `cargo-indicate` schema; This is equivelent to the output of `cargo indicate --show-schema`

```
# This is not truly a GraphQL file; Instead it is a GraphQL representation of
# the types provided by ‘indicate‘.

# _This is the single source of truth for ‘indicate‘. Any deviation from it is
# to be considered a bug._

# This is the currently supported Trustfall directives. They are handled by the
# Trustfall engine.
schema {
    query: RootQuery
}
directive @filter(
    """
    Name of the filter operation to perform.
    """
    op: String!
    """
    List of string operands for the operator.
    """
    value: [String!]
) on FIELD | INLINE_FRAGMENT
directive @tag(
    """
    Name to apply to the given property field.
    """
    name: String
) on FIELD
directive @output(
    """
    What to designate the output field generated from this property field.
    """
    name: String
) on FIELD
directive @optional on FIELD
directive @recurse(
    """
    Recurse up to this many times on this edge. A depth of 1 produces the
    current vertex and its immediate neighbors along the given edge.
    """
    depth: Int!
) on FIELD
directive @fold on FIELD
directive @transform(
    """
    Name of the transformation operation to perform.
    """
    op: String!
) on FIELD

"""
This is the actual types that can be used to create queries.
"""

type RootQuery {
    RootPackage: Package!
    Dependencies(includeRoot: Boolean!): [Package!]!

    """
    Dependencies that are indirect dependencies of the root package;
    excluding direct dependencies that are _only_ direct dependencies, and
    appear nowhere else in the dependency tree
    """
    TransitiveDependencies: [Package!]!
}

# See ‘cargo_metadata::Package‘
type Package {
    id: ID!,
    name: String!,
    version: String!,
    license: String
    keywords: [String!]!
    categories: [String!]!
    manifestPath: String!
    sourcePath: String!

    # This is expensive, due to crates.io crawler policy
    cratesIo: CratesIoStats!

    repository: Webpage

    # All parameters except ‘ignorePaths‘ is exactly the same as ‘tokei::Config‘
    codeStats(
        # If any patterns should be ignored, defaults to an empty list.
        ignoredPaths: [String!],
        # To target only some patterns. Defaults to all. If used,
        # ‘ignoredPaths‘ is still applied
        includedPaths: [String!],
        hidden: Boolean,
        noIgnore: Boolean,
        noIgnoreParent: Boolean,
        noIgnoreDot: Boolean,
        noIgnoreVcs: Boolean,
        treatDocStringsAsComments: Boolean,
        types: [String!] # Types of languages to be included in report
    ): [LanguageCodeStats!]!
    dependencies: [Package!]!

    # For arch and OS, see ‘platforms::target‘
    # For severity, see ‘rustsec::advisory::Severity‘
    advisoryHistory(
        includeWithdrawn: Boolean!,
        arch: String,
        os: String,
        minSeverity: String
    ): [Advisory!]!
    geiger: GeigerUnsafety
}

type CratesIoStats {
    totalDownloads: Int
    recentDownloads: Int
    versionDownloads: Int
    versionsCount: Int
    yanked: Boolean # If this version is yanked from crates.io
    yankedVersions: [String!]
    yankedVersionsCount: Int
    yankedRatio: Float # yanked versions count / versions count
}

# Data from tokei, shared between ‘Language‘ and ‘CodeStats‘
interface CodeStats {
    # Name of the language
    language: String!
    # Total number of files
    files: Int!
    # Total number of lines
    lines: Int!
    # Total number of blank lines
    blanks: Int!
    # Total number of lines of code
    code: Int!
    # Number of lines of comments
    comments: Int!
    # Lines of comments / lines of code
    commentsToCode: Float!
}

# ‘tokei::Language‘
type LanguageCodeStats implements CodeStats {
    # From CodeStats
    language: String!
    files: Int!
    lines: Int!
    blanks: Int!
    code: Int!
    comments: Int!
    commentsToCode: Float!

    # From ‘tokei::Languge::summarize()‘
    summary: LanguageCodeStats!

    # If this language had problem with parsing
    inaccurate: Boolean!

    # Code contained in this
    children: [LanguageBlob!]!
}

# ‘tokei::CodeStats.blobs‘
type LanguageBlob implements CodeStats {
    # From CodeStats
    language: String!
    files: Int!
    lines: Int!
    blanks: Int!
    code: Int!
    comments: Int!
    commentsToCode: Float!

    # Merge this with all child blobs to create new ‘CodeStats‘
    # (‘tokei::CodeStats::summarize()‘)
    summary: LanguageBlob!

    # Blobs of code contained within this one
    blobs: [LanguageBlob!]!
}

type GeigerUnsafety {
    # ‘used‘ refers to code used by the ‘RootPackage‘
    used: GeigerCategories!
    unused: GeigerCategories!

    # used + unused
    total: GeigerCategories!
    forbidsUnsafe: Boolean!
}

type GeigerCategories {
    functions: GeigerCount!
    exprs: GeigerCount!
    item_impls: GeigerCount!
    item_traits: GeigerCount!
    methods: GeigerCount!

    # (functions.safe + exprs.safe + ...), (functions.unsafe + ...)
    total: GeigerCount!
}
```

```

type GeigerCount {
    safe: Int!
    unsafe: Int!

    # safe + unsafe
    total: Int!
    percentageUnsafe: Float!
}

interface Webpage {
    url: String!
}

interface Repository implements Webpage {
    url: String!
}

type GitHubRepository implements Repository & Webpage {
    # From Repository and Webpage
    url: String!

    owner: GitHubUser
    name: String!

    starsCount: Int!
    forksCount: Int!

    # This is the sum of open issues and open PRs
    openIssuesCount: Int!
    watchersCount: Int!

    # If the issues page is available for this repository
    hasIssues: Boolean!
    archived: Boolean!

    # If this is a fork
    fork: Boolean!
}

type GitHubUser {
    username: String!
    email: String!
    unixCreatedAt: Int
    followersCount: Int!
}

# Partly flattened ‘rustsec::advisory::Advisory‘
type Advisory {
    # These fields are flattened out of ‘rustsec::advisory::Metadata‘

    id: String!
    title: String!
    description: String!
    unixDateReported: Int!
    severity: String

    # These are provided by ‘rustsec::advisory::Affected‘
    # They may be empty, so a ‘None‘ means that we do not know
    affectedArch: [String!]
    affectedOs: [String!]
    affectedFunctions: [AffectedFunctionVersions!]

    # These are provided by ‘rustsec::advisory::Versions‘
    patchedVersions: [String!]!
    unaffectedVersions: [String!]!

    # If it was reported in error, this will indicate when it was withdrawn
    unixDateWithdrawn: Int
    #cvss: CvssBase # TODO: Add when Trustfall supports enums
}

# ‘Map<FunctionPath, Vec<VersionReq>>‘ from ‘rustsec::advisory::Affected‘
type AffectedFunctionVersions {
    functionPath: String!
    versions: [String!]!
}
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-06-13

EXAMENSARBETE Evaluation of Rust Codebases Using Public Information
STUDENT Emil Eriksson
HANDLEDARE Alexandru Dura (LTH) & Nikolaos Korkakakis (Volvo Car Sverige AB)
EXAMINATOR Christoph Reichenbach

Att nysta ut höstacken: Vad finns egentligen i ditt mjukvaruprojekt?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Emil Eriksson**

Väldigt sällan skrivs moderna mjukvaruprojekt i ett vakuum. Program kan innehålla flera tusen delar, från ännu fler programmerare, som enkelt laddas ner från nätet. Med mitt program blir det lättare att nysta i dessa delar, och med det har jag undersökt trender för över 2000 populära projekt i programmeringsspråket Rust.

Idag skrivs nästan inga program av en ensam person, team, eller ens företag. Istället förlitar man sig ofta på kod skriven av utvecklare och organisationer som väljer att dela med sig till allmänheten. Detta har bidragit till den stora mängd olika program, hemsidor, och applikationer vi idag tar för givet. Men det krävs stor tillit till att det man använder håller hög kvalité. Detta har historiskt visat sig vara ett minst sagt naivt antagande. `cargo-indicate` är ett verktyg jag utvecklat som gör det lättare att ställa frågor om vad som ingår i ens program. Med hjälp av data från hemsidor, databaser, och koden själv gör det det möjligt att ställa frågor som *Använder jag någon skräpig kod ingen annan verkar använda?* genom sökningar som utvecklarna själva kan skriva.

Detta verktyg har jag sedan använt för att lista ut hur det ser ut för program skrivna i det populära programmeringsspråket Rust. Genom att kolla på data från en stor mängd populära projekt hittade jag flera trender. Till exempel verkar vissa projekt vara populära främst på grund av en mindre PR-kampanj, samtidigt som de innehåller kod som enkelt kan leda till farliga buggar. Det verkar också, något motsägelsefullt, som att ett projekt med många rapporter i databaser över kända problem är ett tecken på att projektet håller *hög* kvalité och inte tvärtom. Det låter konstigt, men en uttråkad programmerare lär knappast ägna speciellt mycket tid att rapportera alla sina misstag.

Verktyget kan komma till användning för både utvecklare och forskare i framtiden som ett sätt att analysera Rust-projekt, och en van programmerare kan utan större besvär utöka det med ännu fler datakällor.