Dependency Families in the Maven Ecosystem

An Analysis of Software Dependency Graphs

Wojciech Graj

Supervisors: Dr. Sebastian Proksch, Cathrine Paulsen

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfillment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Wojciech Graj
Final project course: CSE3000 Research Project
Thesis committee: Dr. Sebastian Proksch, Cathrine Paulsen, Dr. Georgios Iosifidis

Abstract

The Maven ecosystem relies heavily on dependencies to provide functionality, but the relationships between these dependencies are not well understood. This paper introduces the concept of dependency families, where a group of dependencies are owned by the same entity and designed to be used together. We develop a method to detect these families using a combination of structural and statistical techniques, and apply it to the Maven Central repository. Our analysis reveals insights into the structure and trends of dependency families, including their size distribution, usage patterns, and version homogeneity. Specifically, we find that most families are composed of a small core of frequently used dependencies alongside many supplemental ones; that releases without code changes are surprisingly prevalent; and that while many dependencies in a family share version numbering, this is not consistent enough for developers to always rely on. Our findings have implications for developers, maintainers, and users of dependencies in the Maven ecosystem.

1 Introduction

Modern software often reuses functionality provided by existing software libraries instead of building everything from the ground up. The management of these libraries (dependencies) on which certain software depends can be done through a build automation tool. Apache Maven is one of the main build automation tools used for software development in Java [7], and as such, presents a large and interesting dataset from which insights into development practices and trends can be gleaned, which might also be representative of the wider Java ecosystem.

A key characteristic of Java dependencies is their specialization, wherein a single dependency is often scoped to a small subset of functionality. These specialized dependencies are designed to be co-used with a subset of other specialized dependencies, with the goal of providing all the functionality needed for a specific use case while minimizing the total amount of code being included and, therefore, minimizing the amount of unused code in downstream software. [9]

These collections of specialized dependencies that provide complementing functionality are termed ”dependency families” in this paper. Dependency families are an interesting subject of study as they are a key aspect of Java's approach to dependencies, but are absent or less widespread in many other programming languages that often favor feature flags in a single dependency [3].

As an example, consider `org.apache.lucene>lucene-core`, `org.apache.lucene>lucene-queryparser`, and `org.apache.lucene>lucene-queries`. All of these dependencies provide additional tightly-coupled lucene-related functionality, and would therefore belong to the same family, but a downstream user can chose to only depend on the subset of them that is actually needed.

Dependency families are an unstudied facet of the Java ecosystem. Furthermore, very little structural analysis has been performed on entire repositories of Java dependencies, with the maximal extent being the generation and usage of dependency graphs [5] [1], without analysis of trends present in this data. Additionally, while prior work has suggested how best to handle dependency management at the individual package level [8], there is little guidance or tooling support for developers grappling with families of interrelated dependencies. As such, this paper will shed light on dependency families within the Maven ecosystem, their structural features, trends pertaining to them, and how developers can best leverage them.

The results of this research are pertinent to maintainers and developers of dependencies, as trends present among families could inform best practices that reflect the general consensus of the ecosystem, but are currently informal and unwritten. Additionally, it is valuable for downstream dependency users, as it will explain patterns they can generally expect their dependencies to follow, as well as things to look out for. Furthermore, demonstrating the crucial role that dependency families play within the Maven ecosystem could facilitate and guide lobbying for the development of new functionality within Maven that will help facilitate the use and development of dependency families, benefiting the ecosystem as a whole. Finally, provided with this paper is an up-to-date dataset that improves on that provided by Benelallam et al. [1] and Jamie at al. [5], which can be used by other researchers to fill the gap in structural insights about the Maven ecosystem.

This paper attempts to answer the following questions.

1. How can we detect which dependencies belong to the same dependency family?
2. What are some common patterns among dependency families?
   (a) How are dependency family sizes distributed, and how much of the Maven ecosystem to they account for?
   (b) How is the frequency of use of individual dependencies in any family distributed?
   (c) How often are versions out-of-sync and how frequently are releases without code changes published to keep their versions in sync?

Question 2(a) serves to identify the pervasiveness of dependency families, justifying the importance of other results. Question 2(b) evaluates whether dependency families are well-specialized, and could indicate how effort should be prioritized by maintainers of dependency families. Question 2(c) answers whether developers can rely on dependencies being in-sync, and could justify the improvement of Maven's handling of identical releases.

The paper is split into three overarching sections. Section 3 focuses on the dataset and how it was obtained, 4 discusses the detection of dependency families, while section 5 is about the common patterns among dependency families. Each section has methdology, results, and when applicable, discussion subsections.

2 Definitions of Terms

- **Artifact**: A software package which can have multiple distinct versions. Artifacts are identified by the following:
  - **Group ID**: The organization that owns the artifact in reverse DNS notation, used as a namespace.
  - **Artifact ID**: The name of the specific software package in a format that is both machine- and human-readable.
- **Dependency**: A reference to a different artifact that is required by an artifact.
- **Dependency Family**: A group of dependencies owned by a single entity that are designed to be used together to provide additional related functionality.

  For example, all dependencies with a group ID prefixed by the following are considered to belong to their respective families: `org.apache.lucene`, `org.junit`, `app.cash.sqldelight`. Further, `org.apache.camel` and `org.apache.maven` belong to separate families, as they provide completely different non-complementing functionality.
- **Project Object Model (POM)**: A per-artifact configuration file that specifies the artifact ID, version, dependencies, and other information used by Maven.
- **Release**: A published version of an artifact. Artifacts can have multiple releases.

3 Dataset

Before families can be identified and analysis can be performed on them, a dataset is needed. This section describes how the dataset was gathered and provides some key statistics about it.

3.1 Methodology

The Maven Central Repository was chosen as the primary data source, as it is the largest and de facto primary repository of Maven packages, and regularly publishes an index of all hosted files. The index is provided as a list of files in Apache Lucene format, which can then be migrated into a relational database, normalized, and processed.

The OSGi specification defines a system for a dynamic component model which, in summary, contains a list of packages published by specific releases, and a list of packages required by them that should be sourced from different releases. Apart from data about the release each file belongs to and certain file metadata, the Maven index contains OSGi metadata for the subset of releases that can make use of it. This OSGi metadata can be used to determine the dependency relationships between artifacts.

Given the limited adoption of OSGi (see 3.2), using the dependencies of each release declared in its POM would provide a far larger dataset spanning a greater number of artifacts. The Maven index doesn't include the releases' POM files, hence these were individually downloaded from the central repository, and then had their dependencies extracted from them.

Furthermore, Maven allows dependencies to be declared in the `dependencyManagement` section of POM files, providing a hint as to which versions would be compatible, without actually having to depend on the artifacts. This mechanism is typically used within dependency families to ensure compatibility for downstream users that depend on multiple artifacts in a family.

Finally, POM files can inherit certain data from parent POMs, declared in the `parent` section. This can be used as an indicator of some relationship between them.

3.2 Results

The two data sources used are the Maven index, and the POM files of all releases. Only 50 883 of 688 201 (7.39%) of all artifacts in the Maven index have OSGi bundles associated with them, and 15 985 044 (98.1%) releases have POM files.

4 Dependency Family Detection

Prior to conducting any analysis on trends within dependency families, they must first be detected. This section details how the dataset obtained in section 3 was used to identify dependency families and provides a discussion on the quality of the results.

4.1 Methodology

Maven doesn't provide any metadata that trivially identifies artifacts as members of dependency families, and each artifact's family will have to be calculated based on what data is available. Creating groups of similar entities is a task to which community-detection algorithms are well-suited, provided the data can be expressed as a graph. The nodes in such a graph would represent individual artifacts, and the edges between them should be weighted by some manner of similarity metric. Given that dependency families are defined as groups of dependencies that are designed to be used together, a metric representing their pairwise co-use is a natural fit.

This subsection first describes the graph's construction, then discusses evaluation methods.

Graph Construction

Community detection on a graph consisting of the entire central maven index would be prohibitively computationally expensive, hence a hybrid statistical approach with structural constraints was chosen.

All dependencies in a family must be owned by the same organization or person, hence communities can be identified within each organization. It is impossible to identify an organization from an artifact's group ID based solely on structural data, as `org.apache.maven` and `org.apache.felix` both belong to `org.apache`, whereas `io.github.vipcxj` and `io.github.riseclipse` have two distinct owners and shouldn't be in the same family. However, the first two period-delimited segments of the group ID (e.g. `io.github` for `io.github.vipcxj`) will always have to match for two artifacts to have the potential of having the same owner, given that they represent the domain name that hosts the project in reverse-DNS notation. [1] As such, community detection can be performed within each root group ID separately.

As previously mentioned, dependency families are designed to be used together, hence the co-use rate between pairs of dependencies should influence the graph's edge weights. Given that many families consist of a few dependencies that are used many orders of magnitude more often than the rest of the family, a normalized metric that isn't influenced by the absolute usage rate of individual artifacts must be used. The overlap coefficient matches this criteria, as it represents the ratio between the overlap of two sets, and the cardinality of the smaller of the two, yielding a value in range $[0, 1]$ [10].

For sets $\mathbf{D}_x$, $\mathbf{D}_y$ representing the releases that depend on artifacts x and y respectively, the overlap coefficient is calculated as c in equation 1. Given the high computational cost of calculating $|\mathbf{D}_x \cap \mathbf{D}_y|$, it can be approximated by assuming that for each artifact, the number of releases that only depend on one of x or y is minimized. This is shown in equation 2, where $\mathbf{A}$ denotes the set of all artifacts and $\mathbf{V}_a$ denotes the set of releases for artifact a.

$$c(x, y) = \frac{|\mathbf{D}_x \cap \mathbf{D}_y|}{\min\left(|\mathbf{D}_x|, |\mathbf{D}_y|\right)} \tag{1}$$

$$|\mathbf{D}_x \cap \mathbf{D}_y| \approx \sum_{a \in \mathbf{A}} \min\left(|\mathbf{D}_x \cap \mathbf{V}_a|, |\mathbf{D}_y \cap \mathbf{V}_a|\right) \tag{2}$$

For the sake of simplicity, $\mathbf{D}_x$ includes both releases that depend on x, and those that mention it in the `dependencyManagement` POM section.

Apart from the overlap coefficient, the existence of a parent-child relation between any two releases of two artifacts is another good indicator of a potential family. Let this be represented by function $p : \mathbf{A} \times \mathbf{A} \to \{0, 1\}$.

A potentially better indicator could be whether there is a sibling or descendant relationship between artifacts, however both of these would require up to $\frac{|\mathbf{A}_r|(|\mathbf{A}_r|-1)}{2}$ edges, where $\mathbf{A}_r$ is the set of artifacts in root group ID r, which is infeasible for the largest root group IDs.

Given that two metrics exist for the weight of an edge, combining them could potentially lead to better communities. As such, the function w in equation 3 used as the edge weight is a linear combination of the c and p metrics, with $\alpha \in [0, 1]$ determining the bias towards each individual metric. Linear combinations are used due to their simplicity, but other methods of combining the two metrics could potentially be more effective. The edge weight is always normalized to $[0, 1]$, which allows for the same parameters to be used for all cardinalities of $\mathbf{A}_r$.

$$w\left(x, y, \alpha\right) = \alpha p\left(x, y\right) + \left(1 - \alpha\right) c\left(x, y\right) \tag{3}$$

The Leiden [11] and Louvain [2] algorithms are the leading community detection algorithms, and are the algorithms used to identify the families. The Leiden algorithm produces better connected communities and doesn't face the same resolution limit on modularity [11], however preliminary results indicated that the improvement isn't large enough to justify only testing the Leiden algorithm. Additionally, in the case of a graph consisting only of unweighted edges, an algorithm that groups all connected components of the graph is used.

The Leiden and Louvain algorithms both have multiple parameters that can tweak their output. We anticipated, based on prior work, that resolution would have the largest effect, so our search focused on this parameter. Initial results showed that increasing their iteration count has diminishing returns, and the Leiden algorithm's randomness parameter has little impact on the final result. As such, finding the parameters that provide the best communities was done by performing multiple grid searches over combinations of values for the resolution and α parameters.

Community Quality Evaluation

The quality of any given set of communities can be measured by computing their similarity to a set of manually-identified families. A set of 26 families of differing cardinalities, differing cardinalities of their root group IDs, and differing usage rates was used for this purpose to minimize cherry-picking bias. The families were chosen by selecting random packages within larger root group IDs as well as from among the most popular Maven packages, and identifying the family they would belong to. Any packages with ambiguous dependency families were excluded. See appendix B for the list of manually-identified families.

The quality of the communities was calculated as the mean Jaccard index of each community when compared to its corresponding manually-identified family. The detected community corresponding to each manually-identified community is determined to be the detected community with the largest number of nodes in the given manually-identified community.

The Jaccard index, which is the proportion of the overlap of two sets to their union, can be found in equation 4. It was chosen because it is normalized to $[0, 1]$ and can be easily intuitively understood. Additionally, it assigns equal value to false-positives and false-negatives, as there is no obvious reason to punish one of these metrics more harshly. [4]

$$J\left(\mathbf{A}, \mathbf{B}\right) = \frac{|\mathbf{A} \cap \mathbf{B}|}{|\mathbf{A} \cup \mathbf{B}|} \tag{4}$$

The mean false-discovery rates (FDR) and false-negative rates (FNR) are also provided, as calculated in equations 5 and 6 respectively using the count of false positives (FP), false negatives (FN), and true positives (TP). The false-positive metric could not be provided, as there isn't a sensible way to measure the number of true negatives.

$$\text{FDR} = \frac{\text{FP}}{\text{FP} + \text{TP}} \tag{5}$$

$$\text{FNR} = \frac{\text{FN}}{\text{FN} + \text{TP}} \tag{6}$$

Finally, an isolation rate is also provided, equaling the proportion of nodes without any edges, as shown in equation 7.

[1]Certain older artifacts have a one-segment group ID. For these artifacts, the singlular segment is treated as the root group ID.

U is the set of unconnected nodes, and **N** is the set of all nodes in a graph.

$$\text{IR} = \frac{|\mathbf{U}|}{|\mathbf{N}|} \tag{7}$$

4.2 Results

There exist parent-child relations specified in the POM files between 315 705 artifact pairs constrained within their root group IDs, and 15 346 680 pairwise overlap coefficient values between artifacts. 3 704 862 104 edges could exist given the root group ID constraint if all graphs were complete.

Community detection yielded the best results with edge weights being a linear combination of the existence of a parent-child relation and the overlap coefficient, as opposed to a single one of these two metrics.

Table 1: Best hyperparameters and corresponding isolation rate, Jaccard index, false-discovery rate, and false-negative rate for community detection for parent-only, overlap coefficient-only, and mixed edge weights.

Algorithm	Resolution	α	IR	J	FDR	FNR
Connected Components	-	1.0	0.546	0.236	0.275	0.544
Leiden	0.06	0.00	0.115	0.432	0.131	0.522
Louvain	**0.003**	**0.96**	0.0413	0.575	0.114	0.365

As can be seen in table 1, relying solely on parent-child relations led to communities that were overly-large, exemplified by the high false-discovery rate, and didn't match the manually-identified communities well, with a Jaccard index of only 0.236. While the results were better when relying only on co-use, yielding a higher Jaccard index and lower false-discovery rate, a Jaccard index of 0.432 is still low. Both of these approaches had a high false-negative. When $\alpha = 1$, this is solely attributable to the high isolation rate, or high number of nodes without

Optimal results were obtained with $\alpha = 0.96$, where the parent-child relations accounted for the vast majority of any given edge's weight. Despite the overlap coefficients having only a minor influence on edge weight, the Jaccard index is significantly improved at 0.575, and the false-discovery rate has its lowest value.

It could be valuable to research whether different functions for combining edge weights could yield better results. This is due to how odd it is for the overlap coefficient was weighted so lightly, since it is an empirical measurement of co-use, which is what dependency families are definitionally meant to maximize.

The results could potentially be improved by splitting the overlap coefficients for `dependencyManagement` and `dependencies` POM sections. Further, calculating the similarity of artifact names and using full resolved dependency trees instead of only direct dependencies would both be computationally expensive, but might drastically improve results.

4.3 Discussion

The biggest challenge during the detection of dependency families was determining which data would be most appropriate for determining the graph's edge weights. As discussed in the methodology, dependency relations in the OSGi metadata appeared to be a good candidate, as it came included in the Maven index. However, only 7.39% of artifacts have OSGi metadata, which greatly limits its usefulness.

A different metric that was considered is the pairwise similarity of artifacts' names, since artifacts in the same family often share a common prefix in their artifact IDs. The issue is that approximately $3.7 \cdot 10^9$ pairwise comparisons would have to be performed, meaning that a very efficient comparison method would have to be used. The primary issue that prohibited this from being used is the lack of standarization in artifact naming. While some dependency families are well-behaved, the location of this shared prefix can differ (e.g. `org.slf4j>slf4j-api` and `org.slf4j>jcl-over-slf4j`), a subset of dependencies in a family can lack it (e.g. `com.itextpdf>itext7-core` and `com.itextpdf>pdftest`), and some contain a potentially mismatched group ID in their artifact IDs (e.g. `io.github.riseclipse>fr.centralesupelec.edf.riseclipse.main` and `io.github.riseclipse>fr.centralesupelec.edf.riseclipse.iec61850.nsd`). It is difficult to account for such nuance while maximizing computation speed.

The final dependency families had a false-discovery rate of 0.114 and a false-negative rate of 0.365. This means that the identified dependency families, on average, included excess dependencies that belonged in separate families less often than they failed to include dependencies that should have been in them. In this case, it is better for the FNR to be greater than the FDR, as it means that dependency families are not overly-broad, which makes it more feasible to calculate statistics like their version homogeneity. The relatively high false negative rate likely due to the fact that many dependencies are depended upon only a couple times, leading to inaccurate overlap coefficient values that are often low.

Given that a grid search over a wide range of parameters was performed to achieve this community detection result, it is unlikely to get a significantly better result without somehow sourcing more dependency usage data or changing how edge weights are calculated. However, the Jaccard index of 0.575, table 2, and figures such as figure 4 indicate that the quality of detected dependency families is satisfactory.

5 Family-Based Insights

Having identified dependency families sufficiently accurately, insights about trends within them can be found. This section will discuss their pervasiveness, usage rates, and the extent to which versions are kept in-sync within families. These results have implications for manintainers of both dependencies and Maven itself, as well as developers, as discussed in subsection 5.3.

5.1 Methodology

Most of the figures presented in the results subsection are sufficiently simple that they do not require explanation beyond that provided in said subsection. The two notable exceptions are the method of measuring version homogeneity, and the method of identifying empty releases.

Measuring Version Homogeneity

A common pattern within dependency families is for all of their latest version numbers to be equal (see 5.2), allowing a user to easily use multiple dependencies from a family without having to deduce which versions are compatible with each other. The version homogeneity metric, as seen in equation 8, represents the ratio of distinct latest version numbers $\mathbf{N}_t$ to the number of artifacts with an existing latest release $\mathbf{R}_t$ at any point in time t for a given dependency family. A higher value represents a family with more homogeneous versions.

$$H(t) = 1 - \frac{|\mathbf{N}_t| - 1}{|\mathbf{R}_t| - 1} \tag{8}$$

A dependency family's version homogeneity score can be averaged over its entire existence $\mathbf{T}$, as seen in equation 9. A score close to 1 indicates a family whose versions are kept in-sync with each other.

$$H = \frac{1}{|\mathbf{T}|} \sum_{t \in \mathbf{T}} \left(1 - \frac{|\mathbf{N}_t| - 1}{|\mathbf{R}_t| - 1}\right) \tag{9}$$

Identifying Empty Releases

Each release has multiple files associated with it in the Maven index. Most artifacts have a `sources.jar` file, which contains only `.java` source files, and lacks compiled `.class` files. As such, an identical source JAR file can be expected to be produced any time a release is compiled iff the source code hasn't changed, and the releases' author has enabled reproducible builds by settings the `project.build.outputTimestamp` property or the timestamps on the source files have not been changed[2]. Releases can also choose to include the POM file in the sources JAR, which will mark a release as having had its sources changed if the release's version number is included in it verbatim. This is not accounted for, as it is non-trivial to detect.

For consecutive releases, an identical sha1 hash of the sources JAR identifies a release as unchanged.

5.2 Results

This subsection will cover three key insights, namely the cardinalities (sizes) of dependency families, the usage rates of their individual dependencies, and the homogeneity of their versioning.

Cardinalities and Pervasiveness

Figure 1 shows how many dependency families with any given cardinality were identified. There is approximately a negative power-law relationship between the cardinalities of dependency families, and how many instances of a family with such a cardinality exist. As such, most families are very small, consisting of only a few dependencies. The modal cardinality is 2, and only 338 of 59 300 (0.570%) communities have a cardinality $>$ 100.

The three largest dependency families are presented in table 2. They are all coherent despite their large size, and the two for which communities could be manually identified exhibit a low false-discovery rate, indicating they are not overly broad.

[2]https://maven.apache.org/guides/mini/guide-reproducible-builds.html

Figure 1: Frequency of dependency family cardinalities.

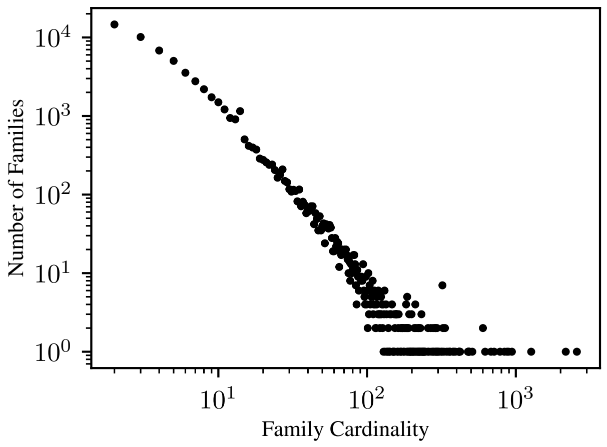


Table 2: Three largest dependency families, their cardinalities, and false-discovery rates.

Dependency Family	Cardinality	FDR
`org.apache.camel`	2574	0.0703
`com.liferay`	2167	-
`io.openliberty.features`	1269	0.000

With the detected dependency families, 522 899 (76.0%) of all artifacts belong to a dependency family.

Usage Rates

Figure 2 exemplifies how often a dependency is depended upon if it the second- through n^{th} most depended-upon dependency in a family. There is an approximate negative power-law relationship between the frequency rank of a dependency within a family, and how often it is used compared to the most-used dependency in its family. The trend breaks down past a frequency rank of around 100 because most families are of a small size, and the sample size therefore drastically decreases.

Dependencies that are the second-most often depended upon within a family still see fairly common use, but this decays as their rank increases. As such, it is clear that dependency families typically consist of up to a few main dependencies and many peripheral dependencies that have very situational use-cases.

Figure 3 shows the proportion of a dependency family that is directly depended-upon. Most often, only a small subset of any dependency family is directly used by any release. There is a significant spike in the bucket encompassing 0.500, and a smaller one in the 0.333 bucket, which is attributable to cases where one dependency is used in a family with cardinality 2 or 3 respectively. The skew in the results is noticeable because these are the most common cardinalities.

Figure 2: Usage rate of dependencies by frequency rank in family, normalized against most used dependency in given family.

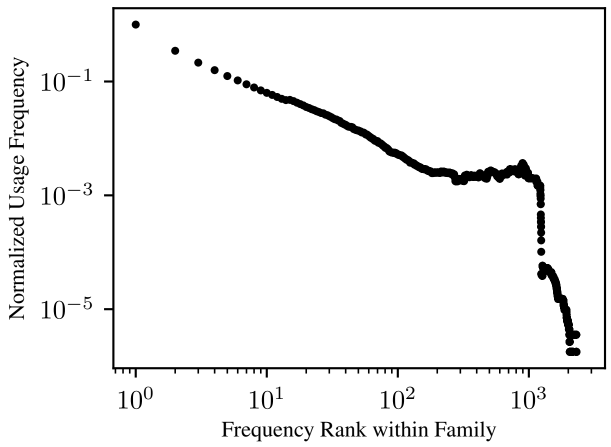
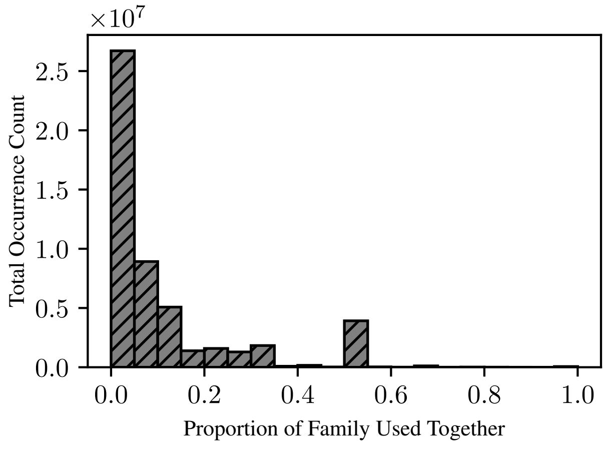


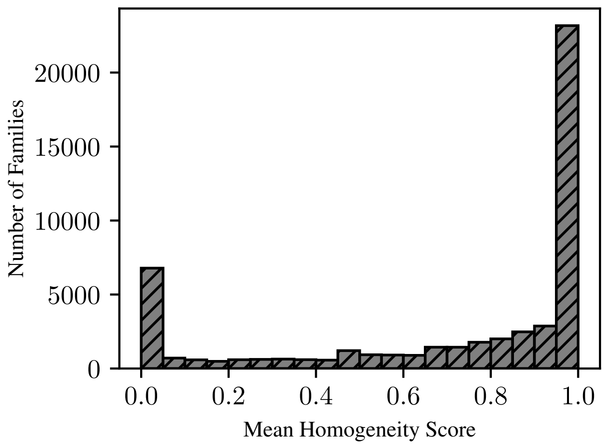
Figure 3: Proportion of dependency families directly depended upon.



Version Homogeneity and Empty Releases

Figure 4 gives an overview of the version homogeneity scores for dependency families. The homogeneity scores appear to have a distribution biased towards higher scores and a spike at very low homogeneities, as the two most common brackets are $0.95 - 1.0$ and $0.0 - 0.05$. 55.3% of all dependency families have a version homogeneity ≥ 0.85.

Figure 4: Mean homogeneity score of dependency families.



As indicated by table 3, 7.98% of releases inside of dependency families are identical, whereas 9.28% of them are identical outside of dependency families.

Table 3: Identical releases inside and outside of dependency families.

	Total Source JAR Releases	Identical Releases	Identical Releases (%)
Inside Family	11793235	941398	7.98
Outside Family	1704253	158127	9.28

As explained in the methodology, it may be valuable to consider releases with very similar sizes to have had no significant code changes. The importance of this is emphasized by the fact that only 45 402 (6.60%) artifacts have reproducible builds enabled. Figure 5 shows the frequency with which various source JAR size differences occur between consecutive releases for size differences $\leq$ 32 B. Based on this graph, a cutoff value of $\leq$ 4 B was chosen to indicate releases that are likely to have had no significant code changes. Table 4 shows that 60.9% of releases inside families fit this criterion, while 56.4% of releases outside of them do.

Tables 3 and 4 suggest there isn't a significant difference in the amount of empty releases inside and outside of dependency families.

To account for minor changes such as timestamps and version numbers, the sizes of the source JARs of consecutive releases can be compared. While code changes could also result in only small changes to the size of the source JARs, they

Figure 5: Absolute difference in source size between consecutive releases for size differences ≤ 32 B.

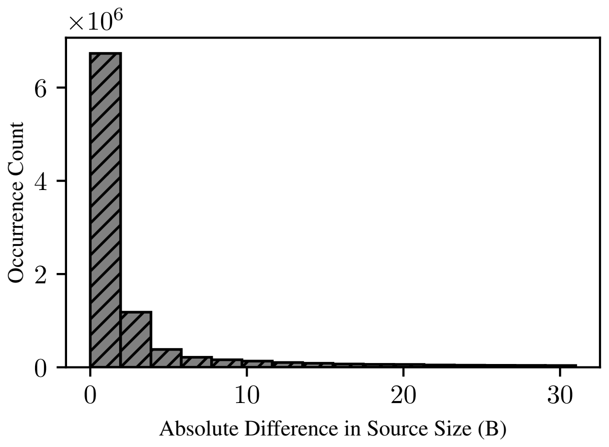


Table 4: Releases inside and outside of dependency families with a size difference ≤ 4 B.

	Total Releases	Similar Releases	Similar Releases (%)
Inside Family	14378803	7180341	49.9
Outside Family	1918902	961663	50.1

are significantly more likely to be larger than a simple version number change, which should at most change the size by a few bytes.

5.3 Discussion

Cardinalities and Pervasiveness

Dependency families account for the majority of the Maven ecosystem, with 76.0% of all dependencies belonging to a family. This number is likely to be higher, given that the false-negative rate is higher than the false-discovery rate when detecting dependency families.

It is also interesting to note that most dependency families are small, with the modal family cardinality being 2, and their cardinalities decreasing approximately based on the power law.

Usage Rates

The trend seen in figure 2 wherein the frequency with which dependencies are used has a negative power-law relationship with their frequency rank approximately corresponds with Zipf's law [12], which states that such a distribution is generally expected when comparing the occurrence count and frequency rank of events in a wide variety of phenomena [6]. This trend only holds until a frequency rank of around 100, which is likely explainable by the fact that only 0.570% of dependency families have a cardinality > 100, and such a low sample size can easily introduce bias.

Figure 3 indicates that typically only a small subset of any given dependency family is directly depended upon. This could be seen as a success of Maven's dependency specialization, since it means that downstream releases do not include the majority of a dependency family's code that they don't need, but that would bloat the final compiled binary.

It is also interesting to note that many families have a few primary dependencies that are used very often, while the rest have a more niche use-case and are used significantly less often. Once again, this could be considered an indicator of the success of dependency specialization.

Version Homogeneity and Empty Releases

High version homogeneity is the clear trend within dependency families, as 55.3% of families have a homogeneity score ≥ 0.85. 0.85 was chosen as the cutoff discriminating homogenous from inhomogenous families by accounting for the FDR of 0.114, and the fact that even for families that have simultaneous releases for all of their dependencies, the minor time differences between these releases render a score of 1.00 unlikely. Developers who use dependency families can often safely assume that various dependencies they use from the same family will have the same version, but this is by no means a given.

Detecting empty releases by simply checking the sha1 hashes of the source JARs of consecutive releases is not very effective, as only 6.60% of artifacts have reproducible builds enabled. This means that the source JARs will have slight changes despite the source code not having changed.

Considering releases with a size difference of ≤ 4 B as having no code changes yields a very high figure of 49.9% of releases within families being empty, and 50.1% of releases outside of them being empty. One could expect there to be far more empty releases within dependency families to keep versions in-sync, but because of the imperfect outcome of dependency family detection, the similarity between these figures is likely attributable to the false-positives decreasing this value for releases in families, and false-negatives increasing it outside of families. These high values indicate that, while similarly sized releases are pervasive, we cannot easily determine whether they actually contain no code changes, and are solely used to keep versions in-sync.

The very high number of empty releases suggests that it would be beneficial to optimize artifact storage in Maven towards them. It is probable that most of these similar releases also have a high byte-level similarity, and by storing the binary diffs between such releases, the size of the Maven Central repository could be noticeably decreased. Furthermore, this result highlights the importance of reproducible builds, as identical source JARs could be trivially deduplicated by certain file systems. Maven should consider making reproducible builds opt-out instead of opt-in, or should at the very least simplify how easy they are to enable.

6 Responsible Research

Ethical implications and reproducibility are both important aspects of research to consider. While we have been unable to identify any potential ethical concerns with analyzing and drawing conclusions from the gathered data, sourcing this data ethically is important.

Instructions and tooling maintained by Apache were used to download the Apache-owned Maven index, which implies their endorsement of such actions, but downloading all of the

89.9 GB of POM files falls outside of a typical use-case. It could be considered ethically questionable to use excess resources provided to the public for free by a non-profit corporation, hence the POM files were sourced from a mirror distributed by Google. Google arguably has sufficient capital and resources to allow for such an operation without undue cost.

With such an empirical study, reproducibility is of high importance. As such, all the source code used to obtain this paper's figures, along with instructions on how to use it, is made available under the Affero General Public License version 3[3]. The choice of a strong copyleft license ensures that any improvements made to the code must also be made available to users of any derived software. By providing all source code, the results can be replicated by other researchers, and the methodology can be scrutinized.

7 Conclusions and Future Work

In this study, we proposed a method for identifying dependency families based on a hybrid structural and statistical approach and evaluated it as quite effective. We note that most dependency families consist of a few frequently used dependencies and many supplemental ones, indicating that dependency specialization is generally successful. Further, we point out that releases without code changes are very common and could be stored more optimally, and justify enabling reproducible builds as a more sensible default. Finally, we note that it is common for the latest releases within any family to have the same version, but this doesn't occur often enough for developers to be able to safely make this assumption without checking.

Future work includes investigating the impact of using binary diffs instead of storing compiled binaries in package repositories, especially for similar releases. It is possible that other dependency ecosystems, including non-Java ones, similarly have many releases with few changes and could also stand to benefit from such an approach. Additionally, it would be interesting to investigate whether the usage trends of dependencies in families are similar among downstream non-dependency software, as well as in other dependency ecosystems.

References

[1] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais. The maven dependency graph: a temporal graph-based representation of maven central. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 344–348. IEEE Press, 2019.

[2] V. D. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, October 2008.

[3] P. Hodgson. Feature toggles (aka feature flags). https://martinfowler.com/articles/feature-toggles.html, October 2017. Accessed: 2025-06-16.

[4] P. Jaccard. The distribution of the flora in the alpine zone. *The New Phytologist*, 11(2):37–50, 1912.

[5] D. Jaime, J. E. Haddad, and P. Poizat. Goblin: A framework for enriching and querying the maven central dependency graph. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 37–41, New York, NY, USA, 2024. Association for Computing Machinery.

[6] D. M. W. Powers. Applications and explanations of Zipf's law. In *New Methods in Language Processing and Computational Natural Language Learning*, 1998.

[7] M. Prakash. Build automation tools for software development a comparative study between maven, gradle and bazel. *Computer Science & Information Technology (CS&IT)*, 2022.

[8] P. Siriwardena. *Maven Essentials*. Packt Publishing, 2015.

[9] C. Soto-Valero, D. Tiwari, T. Toady, and B. Baudry. Automatic specialization of third-party java dependencies. *IEEE Transactions on Software Engineering*, 49(11):5027–5045, 2023.

[10] D. Szymkiewicz. Une contribution statistique à la géographie floristique. *Acta Societatis Botanicorum Poloniae*, 11(3), 1934.

[11] V. A. Traag, L. Waltman, and N. J. van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1), March 2019.

[12] G. K. Zipf. *The Psychobiology of Language, an Introduction to Dynamic Philology*. Houghton Mifflin, Boston, 1935.

[3] https://www.gnu.org/licenses/agpl-3.0.en.html

Appendix

A Family Detection Grid Search Hyperparameters

Table 5: Hyperparameters used in grid search to find The best communities.

Algorithm	Resolution	α	Iterations	Randomness
Leiden	$\{0.01n \mid n \in \mathbb{Z}, 1 \leq n \leq 30\}$	$\{0.1n \mid n \in \mathbb{Z}, 0 \leq n \leq 10\}$	70	0.1
Louvain	$\{0.01n \mid n \in \mathbb{Z}, 1 \leq n \leq 30\}$	$\{0.1n \mid n \in \mathbb{Z}, 0 \leq n \leq 10\}$	70	
Leiden	$\{0.001n \mid n \in \mathbb{Z}, 1 \leq n \leq 15\}$	$\{0.1n \mid n \in \mathbb{Z}, 4 \leq n \leq 10\}$	70	0.1
Louvain	$\{0.001n \mid n \in \mathbb{Z}, 1 \leq n \leq 15\}$	$\{0.1n \mid n \in \mathbb{Z}, 4 \leq n \leq 10\}$	70	
Leiden	$\{0.001n \mid n \in \mathbb{Z}, 2 \leq n \leq 8\}$	$\{0.01n \mid n \in \mathbb{Z}, 85 \leq n \leq 98\}$	70	0.1
Louvain	$\{0.001n \mid n \in \mathbb{Z}, 2 \leq n \leq 8\}$	$\{0.01n \mid n \in \mathbb{Z}, 85 \leq n \leq 98\}$	70	

B Manually-Identified Dependency Families

Table 6 shows the 26 manually-identified dependency families used to evaluate the quality of the detected communities. Any artifact with a group ID that started with the value in the table was considered to be in that family.

Table 6: Manually-identified dependency families.

Group ID Prefix
`app.cash.sqldelight`
`app.cash.treehouse`
`app.cash.wisp`
`com.google.errorprone`
`com.google.guava`
`com.squareup.okio`
`com.squareup.retrofit2`
`io.circe`
`io.github.amyassist`
`io.github.jsoagger`
`io.github.llamalad7`
`io.github.panpf.jsonx`
`io.github.panpf.sketch4`
`io.github.panpf.zoomimage`
`org.apache.kafka`
`org.apache.lucene`
`org.apache.xbean`
`org.eclipse.ditto`
`org.eclipse.store`
`org.eclipse.xtext`
`org.jetbrains.exposed`
`org.junit`
`org.scalatest`
`org.slf4j`
`org.wso2.charon`
`org.wso2.msf4j`

C Source Code

All of the source code used for this paper can be found at following address: https://github.com/wojciech-graj/maven-dependency-families