# Uncovering the Secrets of the Maven Repository

## Analysis of Library Sizes in Maven Central

**Niels Tomassen**[1]

**Supervisor(s): Sebastian Proksch**[1]**, Mehdi Keshani**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Niels Tomassen
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Mehdi Keshani, Soham Chakraborty

# 1 Abstract

This research explores the size variations of artifacts in Maven Central, a repository containing a large collection of Java artifacts. This analysis sheds light on the coding habits and dependency management ecosystems within Maven Central, emphasizing the importance of managing artifact sizes effectively. It also provides valuable insights to library maintainers and clients who want to download libraries. For example, we can determine the average amount of space required to download 100 libraries.

The analysis is done by selecting a single version for each artifact in Maven Central and extracting metadata from the corresponding files.

The results reveal that the average size of an artifact is 1447 KB, although this average is heavily influenced by a few exceptionally large artifacts. Approximately 86% of the artifacts have a size smaller than 400 KB, indicating that the majority of artifacts are relatively lightweight. The large artifacts identified in the analysis are predominantly attributed to two categories. The first category contains extensive projects with a substantial number of files, while the second category includes machine learning or big data projects that include massive data files.


# 2 Introduction

Maven Central is a repository that contains a large number of common libraries used in Java. It serves as a place where developers can store and share projects written in Java or other languages that compile to JVM bytecode. Maven has gained popularity for simplifying the build process of Java projects, especially regarding dependencies. It was estimated in 2018 that every week over 100 million artifacts are downloaded from Maven Central [1].

Maven dynamically downloads Java packages from Maven Central and stores them in a local cache, the .m2 directory, for easy access. The packages uploaded to the Central Maven Repository contain a wealth of useful information about the coding habits of library maintainers and statistics about Java projects. Unfortunately, the libraries uploaded to Maven Central may include nonessential or unnecessary components, such as repackaged dependencies, unused lines of code, and bloated dependencies (dependencies specified but not needed) [2], among other issues.

To classify and bring structure to the extraneous elements contained in libraries, we conducted a data analysis on a dataset from Maven Central. Specifically, we examined why certain libraries occupy more space than others. The insights obtained from this analysis provide valuable information for developing guidelines to reduce the amount of undesirable content in libraries, whether they are intended for Maven or other repositories. Additionally, knowing how big libraries are on average is very valuable information. If a software engineer wants to install 100 libraries, an estimate can be given for the amount of space required. This knowledge is very beneficial in aspects of software design such as resource planning, infrastructure design, performance considerations and more.

While earlier research has analyzed Maven repositories, there has been insufficient investigation into the causes of large artifact sizes (refer to the related works section for more information). Our research serves as a brick in the metaphorical "house of software development knowledge", and enhances our understanding of the most popular repository to distribute and use Java artifacts [3], Maven Central.

This research aims to provide insights into the factors contributing to the varying space requirements of libraries. To address this objective, the research is structured around three sub-questions:

1. How big is an average library on Maven?
2. How is the space requirement of the libraries distributed in the ecosystem?
3. What are the reasons for the larger sizes?

The first research question serves as a baseline to determine the average size of libraries, enabling the identification of outliers based on their significant deviation from this average. The second question is answered by means of a distribution graph illustrating how the size requirement deviates from the average in Maven Central. The last research question involves a more comprehensive approach. Initially, correlation graphs are plotted between relevant metadata of libraries and their sizes. This metadata consists of the number of files in the library and the number of direct and transitive dependencies. For further details, refer to the methodology section. Finally, individual outliers within these correlation graphs are manually analyzed to determine the causes behind their deviation from the expected result.

Our findings show that on average libraries in Maven Central are 1447KB. However, the median is 25.9KB. This means that the average is skewed by a small amount of very large artifacts. Our manual analysis revealed that these very large artifacts are almost all related to machine learning or big data.

This paper will be structured as follows: first there is a section on related works. Section 4 describes the methodology used in this research. After that, the results are presented in Section 5. Section 6 provides a discussion of the findings, with Section 7 on responsible research. Finally, Section 8 concludes the paper.

# 3 Related work

Our research aims to provide insight into why some libraries require more space than others. Here, related works in the field of analysing dependency manager ecosystems are discussed.

Bloated dependencies are packages that are specified as dependencies, but are not required to build or run an application. Bloated dependencies unnecessarily increase the dependency set of an application. Soto-Valero et al. (2021) studied these bloated dependencies in the Maven ecosystem. In particular, they quantified the amount of bloat in Maven artifacts. They concluded that, on average, 75.4% of dependencies are bloated in an artifact, either declared directly or through inheriting from other dependencies [2]. Furthermore, they released a tool called `DEPCLEAN` which analyzes Java applications that are packaged with Maven and can automatically remove any bloated dependencies. This tool is very

useful if one wants to create guidelines to prevent bloat in future Maven artifacts. Additionally it can be used in further research on package managers to detect bloat in Java applications. While bloated dependencies can be one of the main causes for a large dependency set, our research differs from this research because it examines individual package size rather than the dependencies of packages.

`FindBugs` is a tool that statically analyzes Java bytecode and can detect various types of software bugs. The distribution of these types of bugs across software ecosystems provides insights into what causes developers to have software bugs in their code. Mitropoulos et al. (2014) analyzed a dataset obtained from the Central Maven Repository by using `FindBugs`. In their paper, it is explained how their dataset was carefully constructed from a snapshot and had to go through data cleaning to yield the final dataset. This dataset has been uploaded on GitHub, which can be used when doing further research on the Maven ecosystem. The results of their research are not limited to the distribution of bugs in the Maven Ecosystem, they also calculated the correlation between artifact size and bug count. They found that a greater size indicates a higher likelihood of bugs, especially in the categories: style, performance, bad practice and malicious code [4]. While their research offers great insights into issues that arise as the artifact size increases, the underlying reason for large sizes is what is explored in our research.

There are several benefits to using packages to reuse code. For example, it can improve software quality and increase productivity. There are packages that implement very basic functionality, called trivial packages, which are used by a lot of software developers. Abdalkareem et al. (2017) researched these trivial packages to find out why people use trivial packages, when the simple functionality they serve can easily be implemented by hand. Moreover, they examined how common trivial packages are and what the drawbacks of using them are. Their findings indicate that trivial packages make up 16.8% of their dataset of npm packages. The main reason people cited for using trivial packages is that they believe the functionality is well implemented and tested. However, their research showed only 45.2% even have tests [5]. Further analysis showed that 11.5% of trivial packages have over 20 dependencies. When most people use trivial packages for simple functionality, that could be written in a few lines of code, adding all these dependencies unnecessarily bloats the dependency set of your project. Abdalkareem et al. have studied the prevalence of trivial packages and their usage in the npm ecosystem. Our research explores the size of packages in the Maven ecosystem, trivial packages are an interesting subset of these packages, which are expected to have a small size.

When using a dependency manager like Maven, one of the main benefits is that it can automatically download new versions of libraries, as they are released, without any action required from the developer. Newer versions often improve performance or fix problems in the previous versions so updating is desirable. However, occasionally, newer versions introduce defects that were not present in previous versions. When this happens, developers strive to fix these issues within the same day and release another version, a so-called same-day release. Cogo et al. (2021) have researched these same-day releases. Specifically they used data from the npm ecosystem to examine the frequency, the code changes, and the adoption rate of same-day releases. Their findings indicate that despite the time constraint accompanying same-day releases, the changes are important. Furthermore, they found 96% of the popular packages have at least one same-day release [6]. As part of their results they also noticed that same-day releases are often error-prone as 39% of same-day releases were followed up by another same-day release. The results of their research are valuable for software developers, Cogo et al. illustrated the error-prone nature of same-day releases, so developers should carefully assess the trade-off between the value and risk of adopting them. Furthermore, their research showed that it would be wise for popular packages to have a release pipeline that can properly handle same-day releases. Our research differs because we analyze the size of library releases, but as evident from the results of Mitropoulos et al. (2014) an increase in size corresponds to a higher likelihood of bugs which can be the culprit for same-day releases.

Analyzing large software ecosystems such as Maven can unlock valuable insights into decision-making regarding adoption of libraries, or updating to newer versions. In order to model popularity, adoption and diffusion of libraries within a software ecosystem, Kula et al. (2017) introduced the Software Universe Graph (`SUG`). The `SUG` is a way to represent software ecosystems, and its nature allows it to be mined for insights about popularity, adoption and diffusion of libraries. Further contributions of their research include the formal definition of metrics related to adoption, popularity and diffusion which provide a way to compare and contrast these categories among libraries or even software ecosystems. Furthermore, they built `SUG` models for both the Maven and CRAN software ecosystems and show how the `SUG` can be used to illustrate differences between these dependency managers. They found that Maven users are more reluctant to update to newer releases of dependencies whereas CRAN users exhibit a more always-updated approach to dependency management [7]. The `SUG` model can be used in further research into the popularity of libraries, or used for comparisons of software ecosystems. Our research uses a different approach to analyze a software ecosystem as it uses a more traditional data analysis, because the nature of the `SUG` model does not lend itself well to our specific research questions.

There has been some research into the ecosystems of dependency managers, but there has been too little research into the reasons for large artifact sizes. The importance of this research is evident as the result of several researches have illustrated the correlation between large artifact size and a higher bug count.

## 4 Method

The infrastructure we use can be divided into four main components. Figure 1 shows a high level overview of how these parts fit together. The indexer reader (1) reads the maven central index and stores the result in a table. The data selector (2) runs on the table created by the indexer reader and selects which packages to use for analysis. The resolver (3) receives

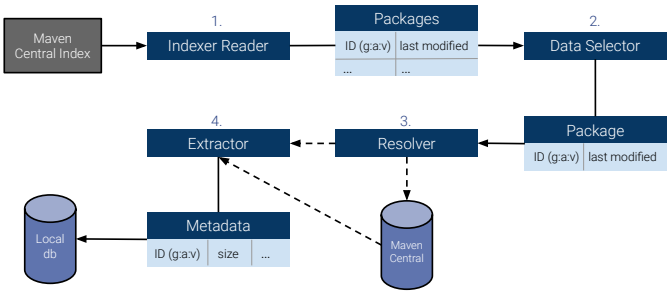


Figure 1: Component diagram of software architecture we use for the data analysis. The four main components are: 1. Indexer Reader, 2. Data Selector, 3. Resolver, and 4. Extractor.

the ID of a package from the data selector and resolves the files required for analysis. The extractor (4) uses the files which were resolved by the resolver and extracts metadata from these specific files.

### 4.1 Data Specification

On Maven Central, there are approximately 11 million indexed packages. Each package consists of a group ID, artifact ID and version. When libraries update, they release a new version. This means most libraries have many versions which are very similar to each other. Analyzing multiple versions of the same packages increases the time and resources required for the analysis and biases trends towards packages with many versions released. This is why in our research, we analyze only one version for each group ID and artifact ID. This is done through random selection to get an accurate distribution of packages per year of Maven Central. This is needed to analyze trends throughout the years. Only selecting the latest versions could result in the exclusion of packages from earlier years, which is why we opted for the random selection.

After pruning all the other versions, our reduced dataset contains 479,915 packages, which each have an unique combination of group ID and artifact ID. Figure 2 shows the distribution of packages released per year for the entire dataset while Figure 3 shows the distribution of the reduced dataset. These are very similar which means the dataset we analyze is a good sample to draw conclusions about trends throughout the years for the entirety of Maven Central.

With the total population size and the sample size we can calculate the error margin and confidence level of our sample. The error margin reflects the level of precision or risk a researcher is willing to accept [8]. The confidence level represents the level of confidence one has in the sample accurately representing the entire population. From a sample size of 479,915 and a population size of 10,333,041 we derive an error margin of 0.18% and a confidence level of 99%.

### 4.2 Reading the Maven Central index

The described sampling method is applied to the index file of Maven Central. The index file is a comprehensive listing of the majority of artifacts stored in Maven Central, along with metadata such as the date of the last modification. In the selection process, the index file is read, and the package

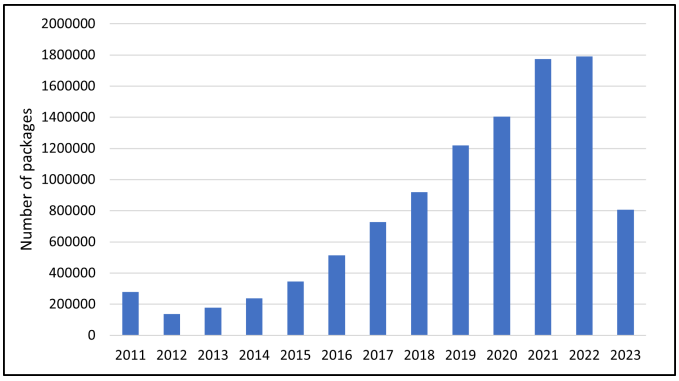


Figure 2: Distribution of packages released per year in the Maven Central index

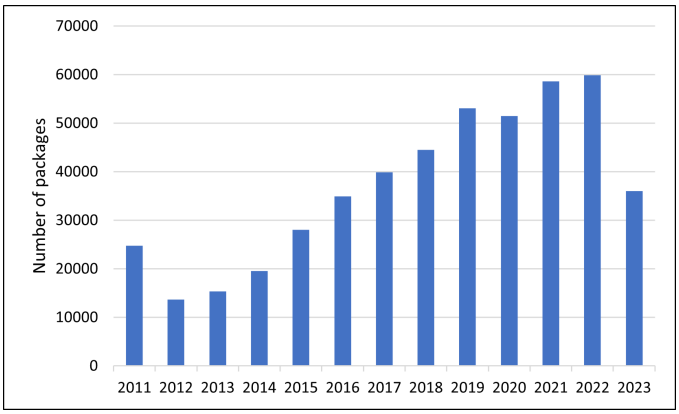


Figure 3: Distribution of packages per year in the sample we obtain from our data selection

ID (groupid:artifactid:version) and the last modified date are stored in a database table. After that, the sampling method is employed to create a new reduced table including only the packages that have been selected through the selection process.

### 4.3 Artifact Resolution

Once the IDs of the selected packages are stored in a table, the next step is artifact resolution. This involves checking whether the local .m2 folder contains the JAR and POM files corresponding to each artifact ID in the table.

A JAR ("Java archive") file is a file that aggregates many class files, meta data and resources into a single compressed archive. It is similar to a ZIP file but used primarily for distributing Java libraries. Our pipeline also works with other packaging types such as EAR, WAR or any generic other type. For simplicity, we will refer to any such packaging type as a JAR file.

A POM (Project Object Model) file is a configuration file used by Maven. The POM file, named "pom.xml" is an XML file that describes the structure and configuration of a Maven project. It tells Maven how to build the project and which dependencies are included.

If either or both of the files (JAR and POM) are not present

in the .m2 folder, the resolver automatically downloads them from Maven Central and saves them in the .m2 folder.

By the end of this process, all of the selected packages will have their corresponding JAR and POM files available in the local cache.

### 4.4 Extraction of Metadata

After the artifact resolution, the next step involves extracting metadata from both the JAR and POM files. This process begins by parsing the actual files into Java JAR and POM objects. Subsequently, two separate extractors are executed to extract the desired metadata from the files.

The first extractor is the "Size Extractor", which extracts the following information for each JAR file:

1. Total number of files (excluding directories)
2. Size of the JAR file in bytes
3. Different file extensions contained within the JAR file, and for each extension:
   (a) Number of files with that extension
   (b) Average size of files with that extension
   (c) Smallest size among files with that extension
   (d) Largest size among files with that extension
   (e) Median size among files with that extension

The Size Extractor provides a comprehensive analysis of file sizes and distributions within the JAR files.

The second extractor is the "Dependency Extractor", which extracts the following information from each POM file:

1. Number of direct dependencies
2. Number of transitive dependencies

The Dependency Extractor provides insights into the dependency structure of each project, including the counts of direct and transitive dependencies specified in the POM files.

## 5 Results

In this section, we present the results of running the extractors on the selected data sample. Out of the 479,915 packages we analyze, 63,250 do not have an artifact (JAR, EAR, WAR, etc.). These packages are left out of the plots and averages we compute in this section because they do not have any files or an artifact with a size. Most of these packages are parent poms that specify the dependencies or other configurations of a child artifact. Also, for an additional 6,419 packages, the POM file could not be resolved because the parent is likely hosted on a different repository than Maven Central.

**Research Question 1**

The first research question is: How big is an average library on Maven? After analyzing the sizes of all the packages in the sample, we calculate the average size to be approximately 1447KB. Out of the 410,246 artifacts we consider 354,755 were smaller than 400KB. Which means more than 86% of packages are less than 400KB yet the average is 1447 KB. The median of the size is 25.9 KB, with the largest package being 986,867 KB.

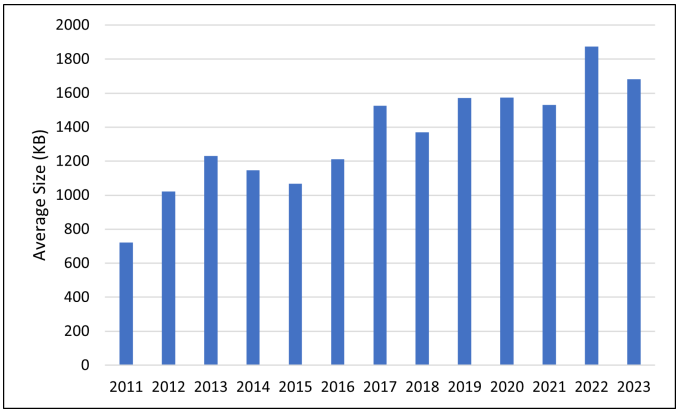


Figure 4: Average size of an artifact per year. The figure shows that the average artifact size increases as the years go by.

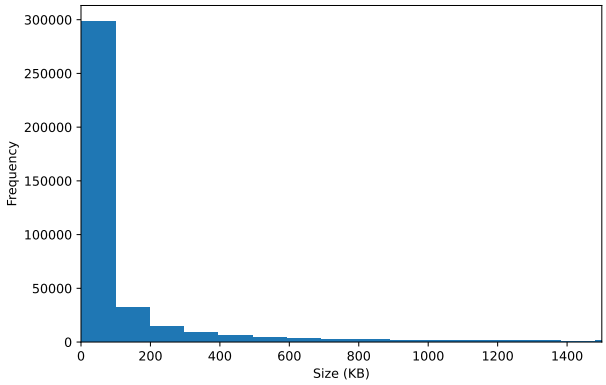


Figure 5: Distribution of size of artifacts in Maven Central. The x-axis has been cut off at 1500KB for readability. We see the majority of packages are significantly smaller than the mean of 1447KB

In Figure 4 the changes in size requirements over the years are depicted. In 2011, the average size of a library was approximately 723 KB and in 2023 this has increased to 1683 KB. The figure shows a distinct pattern, with package size increasing as the years go by.

**Research Question 2**

The second research question explores the distribution of space requirements in Maven Central. Figure 5 presents a distribution graph depicting the distribution of space requirements within our sample from Maven Central. There are outliers that take an enormous amount of space, but for readability, the graph has been cut off at 1500 KB. This distribution is as expected, most packages are in the 0-100KB range, in accordance with the median of 25.9KB. This figure shows most of the packages are significantly smaller than the average size computed in research question 1.

**Research Question 3**

The last research question investigates the reasons behind larger library sizes. Figures 6, 7 and 8 illustrate the correlations between artifact size and the number of files, direct

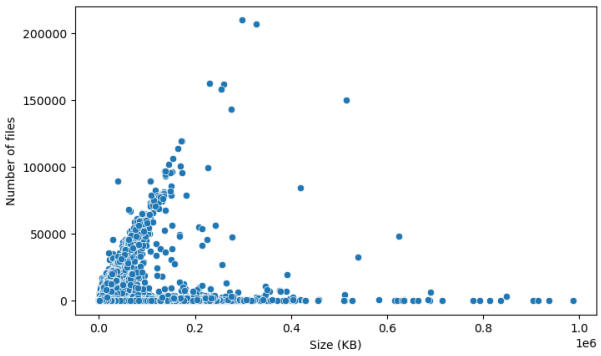


Figure 6: Scatter plot of the number of files included in a JAR and total artifact size.

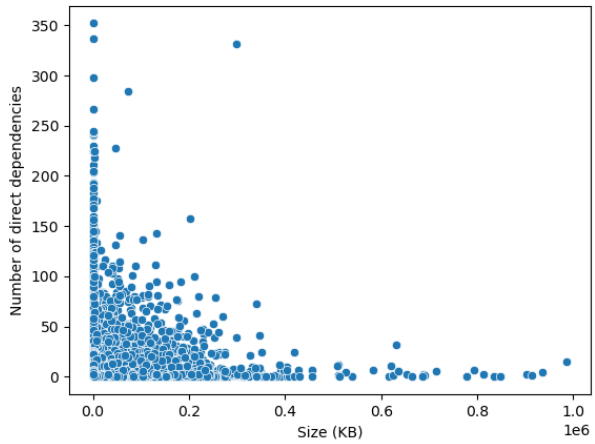


Figure 7: Scatter plot of the number of direct dependencies and total artifact size.

dependencies, and transitive dependencies, respectively. Figure 6 shows two distinct categories of libraries. Those who slightly increase in size when they add more files and those who don't have that many files to begin with and are usually small but can become very large too. In the subsection manual analysis of outliers, we investigate the libaries that have a small number of files but a large total artifact size to find a reason for the larger size.

For the transitive dependencies, an additional 40,396 artifacts are not taken into account in the analysis. This is because the library we use to determine the transitive dependencies, ShrinkWrap[1], has dependency conflicts with a part of our core infrastructure, the resolver. Because of this, the parent pom files cannot be resolved for certain artifacts, which is needed to resolve the dependencies of the child pom. Nevertheless, we still obtain the number of transitive dependencies for 369,723 artifacts, which are shown in Figure 8. This does not impact Figures 6 and 7, which include 410,246 artifacts.

[1]https://github.com/shrinkwrap/resolver

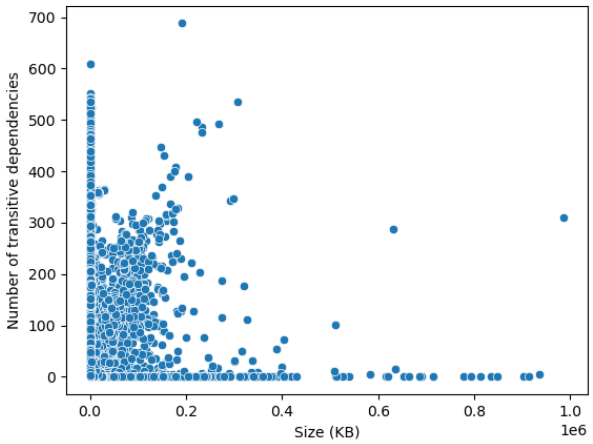


Figure 8: Scatter plot of the number of transitive dependencies and total artifact size..

### Manual analysis of outliers

The analysis of outliers that belong to the category of small number of files and large artifact size from Figure 6 provides insight into other dimensions that contribute to library size. Specifically, we looked at packages that have less than 200 files and are larger than 600,000 KB ( 600 MB). In our sample, it turns out there are 11 such packages. The artifacts that were analyzed can be found in the README file of the repository. As expected, most of them are data analysis and machine learning related projects that include massive data files such .data, .csv, .db, .nt and .rnn. Some of the packages are analyzing natural language so they had many large .ol files describing part of a language features. An example of such a file is: 'bg-hyphenation.hfst.ol' which appears to be related to hyphenation patterns or rules for the Bulgarian language.

An interesting case is one package that had three massive .so files. Shared object (so) files are dynamically linked libraries that contain compiled code and data in Unix-like operating systems. By linking to shared object files, programs can access functions, data, and other resources contained within those files without including them directly in the JAR.

### File extensions

Figure 9 displays the file extensions with the largest average size, highlighting the specific types of files that, when included, contribute significantly to the overall size of artifacts in Maven Central. Many of the largest files are used for storing data, like files such as .rnn (recurrent neural network), .box (used for storing e-mails), .bigmodel, etc. These files can take up a significant amount of space.

On the other hand, Figure 10 illustrates the average distribution of file extensions within a JAR file, providing insights into the composition of file types typically found in JAR files. Class files are by far the most common files in a JAR file hosted on Maven Central. Applications often consist of numerous class files, each representing different functionalities and components of the software. The number of class files can go into the thousands for larger applications, the artifact

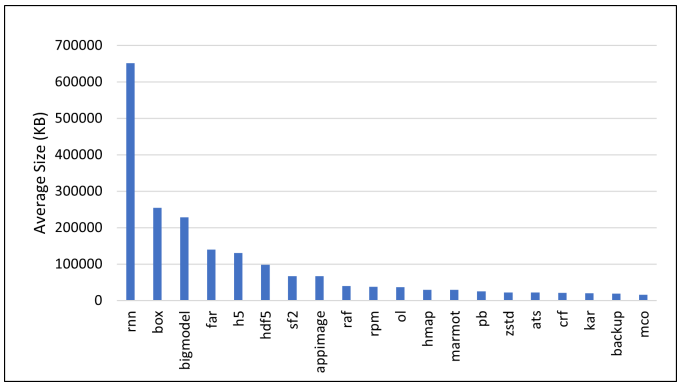


Figure 9: File extensions with the largest average size.

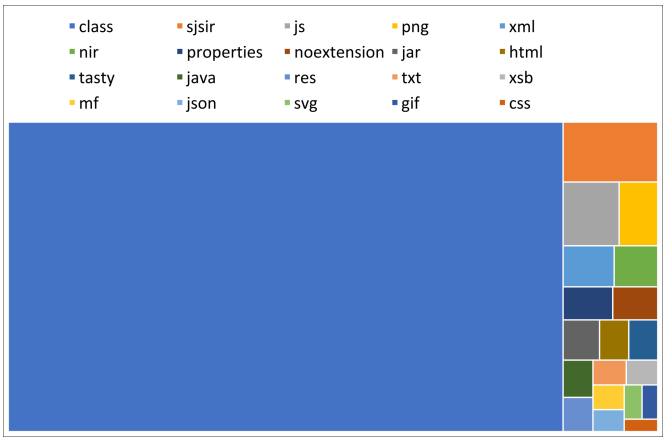


Figure 10: Distribution of number of occurrences of file extensions found in JAR files hosted on Maven Central. Class files are the most common, being 30.3 times more common than the number two: .sjsir files.

with the most class files in our sample has a total of 209,729 class files.[2]

Number two and three are .sjsir and .js files which are similar to class files but used for Scala and JavaScript projects respectively. Interesting is .png files which appear higher on the list than we expected. After further analysis, we find that png files are only included in 17,238 artifacts but on average they are included 70.4 times. This high number explains why they are found so high on the list even though they are not that common to include in a JAR file.

We also see a lot of extensions like .mf and .xml files. A manifest file (.mf) is a text file that contains metadata about a JAR file and its content, it is not a surprise that this is a common file as almost any JAR includes such a file. XML files can be used for different use cases but a common one is for the POM.xml which is present in almost any Maven project. This explains why so many JAR files hosted on Maven Central include xml files.

[2]ID: org.apache.servicemix.bundles:org.apache.servicemix.bundles.aws-java-sdk2:2.17.257_1

# 6 Discussion

In this section, we interpret the results obtained from the analysis. Additionally, we discuss the implications and limitations of our research. Finally, we conclude with recommendations for further research in this area.

## 6.1 Interpretation

From the results, it is evident the average is getting skewed by a few very large artifacts. This can be seen in Figure 5. The large majority of artifacts are smaller than 400 kilobytes, still the average size is 1447 kilobytes. The median of the sample, which is 25.9 KB, indicates that most artifacts are extremely lightweight. This means that the 14% remaining artifacts are so big they massively contribute to the average size.

**Average size per year**

Figure 4 shows the average size of an artifact per year. It is evident that there is a distinct pattern with package size increasing as the years go by. We suspect that that this is caused by an increase in the machine learning and big data industry. As seen in the results section on file extensions, many of the largest extensions are files like .rnn and .bigmodel which are used for machine learning or data purposes. The increase in these industries can be explained by a general increase in the storage and computing power of modern computers. Moore's law states that the number of transistors on a microchip doubles approximately every two years, while the cost of the chip remains the same or decreases. Because of this, most modern high-end computers have the capacity to store over a terabyte (TB) of data in their main memory [9].

Another explanation is that most packages have multiple versions, with newer versions adding extra features or fixing bugs. This could cause an increase in size in the later years as there is a higher probability of a package being in a later stage of its development cycle, and thus having a larger size as more features got added.

**Reasons for large artifact size**

Looking at Figure 6, it is clear that although the number of files do increase artifact size slightly, they are not the main culprit for the largest libraries. A similar thing can be stated for the number of direct and transitive dependencies. It seems in fact that the largest libraries have a low number of files, direct and transitive dependencies.

Our manual analysis of outliers reveals that the largest files are often large due to a few very large files used to store data needed in the rest of the application. The artifacts that used these files were all related to big data or machine learning in some way. Bigger libraries are not preferable for distributing purposes. It takes a lot of space to store them in a database like Maven Central, and users have to download a massive artifact to be able to use it. In most cases it would be better practice to not include such large files that store data in the JAR and instead upload them to a separate online storage system and write in a README how to connect to these files.

The manual analysis also revealed one case where .so files are included in the JAR. It is striking that these files are included in the artifact's JAR, since their main purpose is to separate reusable code from the program, like a dependency from Maven.

**The big picture**

From our analysis, it has become clear that the number of files and the number of direct and transitive dependencies are not the underlying reasons for the largest artifacts. We found that the largest artifacts are related to machine learning or big data in some way. These libraries include massive files that are used for storing data. This is supported by the fact that average library size seems to have increased along with an increase in the machine learning and data industry. These large library are undesirable for distributing purposes. Hence, we urge that the software community considers size when implementing and distributing libraries. One thing we propose is the separation of these massive files from the actual code, which will drastically decrease the size of these artifacts.

### 6.2 Threats to validity

It is important to acknowledge the limitations of this study, for example, we only select one version per package. We believe this results in the most fair analysis but it is important to keep in mind. In addition, there are 6,419 packages for which the POM file could not be resolved because the parent is hosted on a different repository.

The way we detect outliers is also a limiting factor. We manually query the data for packages that have less than an arbitrary number of files and are larger than a certain threshold. Changing these thresholds leads to different packages being selected for manual analysis which might reveal different explanations for large artifacts.

Moreover, we resolve the number of transitive dependencies with ShrinkWrap which conflicts with part of our core infrastructure. Because of this, an additional 40,396 packages are not taken into account for the number of transitive dependencies. This means these packages were not taken into account in the analysis of the impact of transitive dependencies on size.

Finally, there could be other factors that cause a difference in size that were not analyzed in this research. These unexplored factors could potentially contribute to variations in package sizes.

These limitations should be taken into account when interpreting the results.

### 6.3 Future Works

To further enhance our understanding of library sizes in the Central Maven Repository, future research could explore additional factors that contribute to the size variations. Investigating the impact of specific dependencies, coding practices, or build configurations could provide valuable insights. Moreover, considering other dimensions of library quality, such as performance or security, could lead to a more comprehensive analysis.

## 7 Responsible Research

Conducting research in a responsible manner is crucial. As researchers, it is important to prioritize transparency in methodology and results, as well as consider any ethical implications of the research. In our study, we adhere to these principles by ensuring the reproducibility of our results. Moreover, we take precautions to avoid exceeding the rate limit while making requests to Maven Central, in order to prevent excessive strain on this publicly available resource. Respecting usage limits helps maintain the availability and stability of the service for all users. By being mindful of these considerations, we aim to conduct our research responsibly and ethically.

It is important the results are correct and consistent. We ensure code consistency and correctness by conducting extensive testing to verify that our code consistently produces the expected results. We synthesize fake packages and run the extractors on them to validate the functionality and reliability of our code.

### 7.1 Reproducibility

Reproducibility is indeed a crucial aspect of data analysis. It ensures that consistent results are obtained every time the experiment is run. To achieve reproducibility, we take several measures. Firstly, we provide a means to reproduce the data sample that was selected. This is accomplished through the use of a configuration file that where the seed value can be set. The seed value influences which version will be analyzed for each package. By setting this parameter to the same value, the same data sample can be obtained consistently.

Furthermore, our software development and deployment process ensures reproducibility. We have containerized the application using Docker[3]. This ensures that the code can run consistently inside of the container, without depending on the underlying environment of the machine. This effectively eliminates the infamous ”it works on my machine” issue.

By implementing these measures, we strive to ensure the reproducibility of our research and enable others to validate and reproduce our findings.

### 7.2 Reduce strain on Maven Central

To minimize the number of requests made to Maven Central during the resolution step, we leverage the use of a large existing .m2 folder. This folder, created by the FASTEN[4] project, contains a considerable number of packages that have been incrementally downloaded from Maven Central over time. By utilizing this local cache of artifacts, we significantly reduce the number of requests required to fetch packages from Maven Central.

This approach helps optimize the resolution process by relying on the pre-existing collection of artifacts in the .m2 folder, thereby decreasing the number of direct downloads from Maven Central. By leveraging this local cache, we can efficiently access and retrieve the required artifacts while minimizing the strain on the Maven Central server.

## 8 Conclusion

Maven Central contains a wealth of information about Java artifacts that provides insight into coding habits and dependency management ecosystems as a whole. Our analysis is

[3]Docker: an open-source platform for packaging and deploying applications in isolated containers, providing consistency and portability across environments. Available at: https://www.docker.com/

[4]FASTEN: https://www.fasten-project.eu/view/Main/

related to the size of these artifacts and the underlying causes for size variations.

To analyze this we first employed a data selection, by selecting only one version for each artifact. After that, we resolved the POM and JAR files for each package and extracted metadata from both of them.

The results show that the average size of an artifact is 1447 KB, which is increased dramatically by a few very large artifacts. 86% of artifacts are smaller than 400KB. These very large artifacts were either massive projects with a large amount of files, or they were a machine learning or big data project that included enormous files containing data.

## References


[1] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. The maven dependency graph: a temporal graph-based representation of maven central. 2019.

[2] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. 2021.

[3] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. The emergence of software diversity in maven central. 2019.

[4] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. The bug catalog of the maven ecosystem. 2014.

[5] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. 2017.

[6] Filipe R. Cogo, Gustavo A. Oliva, Cor-Paul Bezemer, and Ahmed. E. Hassan. An empirical study of same-day releases of popular packages in the npm ecosystem. 2021.

[7] Raula Gaikovina Kula, Coen De Roover, Daniel M. German, Takashi Ishio, and Katsuro Inoue. Modeling library popularity within a software ecosystem. 2017.

[8] Hamed Taherdoost. Sampling methods in research methodology; how to choose a sampling technique for research. 2016.

[9] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? 2020.