Object-Level Dependency Analysis of Python Modules

Tom Aarsen, s1027401
Radboud University, Netherlands
tom.aarsen@ru.nl

ABSTRACT

Developers of open-source products are currently unable to know which sections of their products are used by others. As a consequence, these developers might mismanage their often limited manpower by working on sections that are rarely used. This work focuses on Python code and introduces a general approach to gathering knowledge about the usage of objects from a given module. The approach involves gathering files that may use objects from a given module, and then extracting the object-level dependencies of that given module from every file. Furthermore, we introduce and publish a Python module entitled `module_dependencies` that implements this approach. It is able to find hundreds of thousands of uses of objects from any Python module in a matter of minutes, giving developers critical insight about which areas of their code should be prioritized.

1 INTRODUCTION

Open source software development may be considered a bit of a one way street: the developer of a system may frequently inform the system users of new additions and changes, but users will rarely return information to the developers. This happens almost exclusively when an issue is discovered by a user, and thus a system developers will never have a good grasp of what functionality of their system is actually being used.

This is a shame, as this knowledge is crucial for determining which sections of code have urgency and priority. If this knowledge was readily accessible, developers would be able to determine where their often limited manpower must be directed for the biggest benefit.

This work will tackle this gap in knowledge by producing a general approach of extracting this information. Furthermore, this work introduces a Python [12] module entitled `module_dependencies` which applies this approach in practice. The scope of this product is limited to Python modules, and with object-level this work refers to functions, classes and variables. This work answers the following research question and sub-questions:

RQ	How to determine how frequently objects from a given Python module are used in practice?
Sub-RQ1	How to gather source code of open-source projects that rely on objects from a given Python module?
Sub-RQ2	How to determine the usage of imported objects from a given Python file?

Note that the RQ may be answered by answering both Sub-RQ1 and Sub-RQ2, and applying the two resulting approaches sequentially.

To summarize, this work contains the following contributions:

- A method of gathering source code of open-source projects that rely on objects from a given Python module. (Sub-RQ1)
- A technique for extracting the used objects from a given Python modules from a given Python file using static analysis. (Sub-RQ2)
- An open-source Python module called `module_dependencies`[1] that may be used to determine the usage of objects from a given Python module. (RQ)

The `module_dependencies` module introduced in this work is a tool designed to be used for gaining knowledge about the usage statistics of Python modules. The following sections quickly illustrate situations in which the tool is (not) recommended.

1.1 When to use `module_dependencies`?

The `module_dependencies` module searches for, downloads, and parses unpredictable text (i.e. code) extracted via GitHub and GitLab. It does so in order to count how frequently not only the module as a whole, but also sections (classes, functions, etc.) of that module are used. Consequently, this tool is highly recommended when attempting to answer the following (or similar) text mining questions:

1.1.1 How frequently are specific sections of code of Python module X used? For example, how common are `MaxPool1D` layers relative to `AveragePooling1D` layers in Tensorflow? Or, which functionality of my Python module is most frequently used, and thus would benefit the most from improvements?

1.1.2 How common is Python module X relative to Python module Y?. For example, when trying to determine the market share of different modules in the same sphere, such as comparing different machine learning frameworks.

1.2 When not to use `module_dependencies`?

The `module_dependencies` module only counts number of uses up to once per file, and thus does not prove useful when the frequency of use within the same file is important. Furthermore, it does not consider the context of the usage, e.g. a function called with a string versus an integer. Beyond that, multiple different Python modules could be imported with the same name, and in those situations `module_dependencies` is unable to differentiate them, causing improper results.

Lastly, `module_dependencies` relies on the Sourcegraph API (see Section 3.1), which does not allow paginated requests. As a result, `module_dependencies` can only send one big request, which may time out for very large requests of above 250,000 imports of a module. See Section 4 for more information about the validation of the tool, which may influence whether this tool is right for your text mining task.

TxMM'21-'22, Radboud University, Nijmegen, Netherlands

[1] https://github.com/.../module_dependencies

2 RELATED WORK AND BACKGROUND

2.1 Python Dependency Analysis

A core aspect of extracting the object-level dependencies from a Python file (Sub-RQ2) involves determining which Python modules are dependencies for a given file. Prior research has been carried out for this task.

GitHub code snippets (called gists) are often uploaded without specific dependencies listed, resulting in gists that cannot be executed. For determining whether a certain gist falls in this category, Gistable [6] has been developed. This tool builds an Abstract Syntax Tree (AST) of the source code from the gist, and traverses this tree to determine the imported modules. The Gistable algorithm is able to successfully tackle non-trivial types of importing in Python.

Since then, more research has been carried out to analyse dependencies. DockerizeMe [7] was developed to combat a naive assumption from Gistable. Furthermore, V2 [8] was developed using the same dependency analysis technique as Gistable and DockerizeMe, but now focusing on differences in dependency versions. It also supports dependency analysis of Jupyter notebooks. Similar work has been performed resulting in SnifferDog [16]. This tool also uses AST analysis to attempt to ensure the executability of Jupyter notebooks. Beyond gists and Jupyter notebooks, similar work was done in order to determine the executability and reproducability of Stack Overflow code snippets [9].

All of the aforementioned works rely on AST analysis in order to determine the dependencies of a code snippet. In particular, the technique used by Gistable, DockerizeMe and V2 has been used as the basis behind the `module_dependencies` tool presented in this work.

2.2 Abstract Syntax Tree analysis

As suggested in Section 2.1, Abstract Syntax Tree (AST) analysis is an important technique for analysing source code. Such an AST is a tree representation of source code, which can easily be parsed and analysed. The Python standard library contains a lightweight `ast` module [11] which allows parsing Python source code into an AST. This module also implements a convenience visitor class that allows for simple traversing of any AST.

However, as with the majority of Python standard library modules, it is not particularly extensive. If more features are desired, then an extension of `ast` called `astroid` [10] may be better suited. Crucially, `astroid` can statically infer various Python constructs. As a result, if a module `foo` is installed, and some program using an object deep within `foo`, then `astroid` may be able to infer that the object originated from `foo`.

However, `astroid` has some limitations and drawbacks that make it less desirable than `ast` for the purposes of answering Sub-RQ2. See Section 3.2 for the motivation of this choice.

2.3 Large-Scale Code Gathering

Determining how frequently objects from a given Python module are used in practice (RQ) involves gathering source code that might depend on that module (Sub-RQ1). This source code can then be analysed further to determine the objects that are used. However, this data collection task is not a trivial problem. After all, a lot of data must be considered before a useful usage distribution can be determined.

Several tools have been developed for gathering this kind of information, such as GitPython [13] and later PyDriller [15]. Both tools allow for downloading the full source code of a project, but seem unable to only downloading certain files that contain dependencies of a module of interest. This makes them somewhat inefficient for the task.

An alternative option are offline mirrors of GitHub data, such as GHTorrent [3, 4], which contains around 18TB of compressed data. The main benefit of this information is that it can be accessed without rate limits, unlike with the real GitHub data. That said, GHTorrent does not store code. This disqualifies it as an option for this work.

3 APPROACH

Figure 1 shows a high-level overview of the approach that will be described and motivated in this section. First and foremost, the figure shows 3 crucial steps performed:

(1) `Module Name` ⟶ `Files`.
(2) `Files` ⟶ `Dependencies`.
(3) `Dependencies` ⟶ `Usage Distribution/Plot`.

The first two steps correspond precisely with Sub-RQ1 and Sub-RQ2. These three steps have been combined in order to develop `module_dependencies`[2], a Python module allowing users to determine the usage of objects from any Python module. Beyond that, these three steps will be elaborated on in the following sections.

3.1 Gathering Source Code

This section will describe the methodology of gathering files with Python source code that may contain uses of a given Python module (Sub-RQ1). This module will be denoted by m from this point onwards. It will represent the name of the module that is used when importing said module. Some examples include `import nltk` and `from nltk import word_tokenize`. In both of these examples, m = `'nltk'`.

We consider three strategies for gathering relevant source code.

(1) The first strategy is to rely on the GitHub Search API [2] to find files that contain references of m. Then, download these files directly through GitHub. This approach is straight-forward, yet crippled by the GitHub rate limit. The GitHub search API has a rate limit of 30 requests per minute for authenticated accounts, while some Python modules are used in hundreds of thousands of files on GitHub. Merely collecting the data to analyse such modules using this approach would take several days at the minimum.
(2) As mentioned in Section 2.3, some alternative strategies exist for skirting around this rate limit [3, 4, 13, 15]. However, these programs tend to either not store source code (GHTorrent), or require downloading entire repositories at once (GitPython, PyDriller). Furthermore, these tools would still require the GitHub Search API to determine

[2]https://github.com/.../module_dependencies

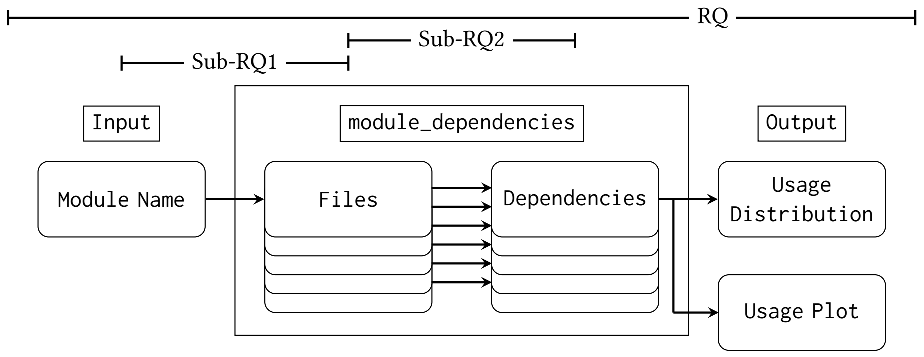

Figure 1: Overview of the approach taken by this work, including a representation on how the research question and subquestions relate to the work.

which repositories must be downloaded. Although this approach does not use the GitHub API to download the files, the rate limit would still be hit for searching. In short, none of these tools would result in a time-efficient program.

(3) A third strategy is to rely on a third-party code searching engine. One such program is Sourcegraph [14]. Sourcegraph allows users to query millions of repositories using GraphQL [5] queries. Such queries are powerful, allowing users to search for specific terms, regular expressions, programming languages, repositories, files and so on. Furthermore, the queries allow users to specify which information they wish to collect, including file content. Using Sourcegraph, the content of files containing import statements of *m* can be fetched very rapidly.

In summary, Sourcegraph supports gathering exactly the desired files through detailed searching. The data is gathered very efficiently - multiple orders of magnitude quicker than the alternatives. As we consider the most important use case of `module_dependencies` to be the *relative* usage of functionality the considerable efficiency of Sourcegraph is preferred, even if it may not capture exactly as many uses of *m* as strategy 1. Furthermore, if this approach is to be used as a single step in a larger text mining task, then time-efficiency is certainly appreciated. In Section 4 we will validate the performance of the chosen strategy 3 in contrast to using the GitHub API.

The Sourcegraph GraphQL API accepts requests using `JSON` payloads with two keys: `query` and `variables`. The value for the former key must be the actual GraphQL query, which specifies the output that we wish to receive. The query that was used in this work has been attached in Listing 2. The second key, `variables`, is a `JSON` object mapping the variables used in the aforementioned query to the values that they should have. This search query is a space-separated string containing several search options. The most important one is the `'content'` parameter, while the remainder of the parameters can be seen in Table 1. This `'content'` field contains the search term itself. For the term, this work uses a regular expression which matches all different ways that *m* can be imported:
`^\s*(import|from) +`*m*`[\s\.,$]`
For fetching Jupyter notebooks, the first character `^` is replaced with `\"`, as code in Jupyter notebook files always start with some optional whitespace followed by a double quote symbol.

Note that this work is interested in the usage of objects from *m*, and thus searching for imports as opposed to usage seems counterintuitive. However, any usage of *m* must require an import of *m*. That said, *m* can be imported without being used in a file.

To summarize, this is why we believe this methodology will have a high recall and a relatively high precision. Step 2 from Section 3 will eliminate any false positives left over from this step. Furthermore, we believe this methodology to be an extremely time-efficient alternative to existing solutions.

3.2 Extracting Functionality Dependencies

This section will describe the methodology of extracting functionality from *m* from a given file (Sub-RQ2). This source code from this file will be denoted by *f* from now onwards. Note that this work is interested in functionality dependencies on an object-level, e.g. variables, classes and functions. For example, given Listing 1, this step wishes to extract `nltk.word_tokenize` and `nltk.pos_tag`.

```python
from nltk import word_tokenize, pos_tag
sentence = "Please tag and tokenize this"
tagged = pos_tag(word_tokenize(sentence))
```

Listing 1: Python code snippet using objects from the Natural Language Toolkit (NLTK) [1].

We believe that AST analysis is the most effective method of analysing source code, and thus we consider two strategies for extracting object-level functionality dependencies that both involve AST analysis as described in Section 2.2.

(1) The first strategy relies on the `ast` standard library. This module contains a relatively efficient parsing function, but is limited in functionality. The strategy involves parsing a file using `ast` into an AST, to then use a visitor to traverse this AST. When encountering specific nodes, e.g. `ast_Import`, `ast_ImportFrom` and `visit_Name`, the visitor can keep track of imported objects.

(2) The second strategy uses the more extensive `astroid` module instead, as introduced in Section 2.2. This module is an extension of `ast`, and allows for inference of objects. For example, if Listing 1 was parsed with `astroid`, then

it can infer that `word_tokenize` is a function defined under `nltk.tokenize.word_tokenize`, and that `pos_tag` is a function defined under `nltk.tag.pos_tag`.

The `astroid` module is able to infer by performing analysis on *m* itself. This marks one important distinction between the two strategies: it requires *m* to be installed. Another difference is related to overhead. Parsing using `astroid` and performing inference is significantly slower than parsing using `ast`. Lastly, applying `astroid` in practice has shown that a noticeable amount of Python modules use "hacks" to load their objects in a cached or lazy way. These can cause `astroid` to no longer be able to infer them effectively.For example, any use of `nltk.corpus` will be inferred as `nltk.lazyimport.LazyModule`. As a result, `astroid` is a considerably less attractive option, and thus this work uses strategy 1 instead.

Whenever the file to extract dependencies from is a Jupyter notebook, that file is first converted to a regular Python file. This is done by simply reading the `JSON` contents of the notebook, and appending each of the cells together into a Python file. Note that this work attempts to parse each cell, and discards cells that raise errors. This combats the common pattern of having code cell blocks containing non-code contents, such as email addresses.

To summarize, this work relies on `ast` and a visitor originally inspired by DockerizeMe [7]. This visitor is extended to track the usage of objects. Then, filtering is applied to ensure that only objects from *m* remain. This methodology of extracting dependencies from files is repeated for every file resulting from Step 1 from Section 3.

3.3 Post-processing

As mentioned in Section 3.2, this work relies on `ast`, which may for example return either `nltk.word_tokenize` or `nltk.tokenize.word_tokenize` for each usage of `word_tokenize`. This hampers the ability to appropriately aggregate all dependencies, and prevents analysis on usage of content derived from `nltk.tokenize`.

To combat this, this work will optionally extend all dependencies resulting from Step 2 in Section 3 to the most frequent potential origin of that dependency. In short, `nltk.word_tokenize` will be extended to `nltk.tokenize.word_tokenize`. Then, all uses of objects are aggregated, resulting in a number that represents the number of files in which each object was used.

4 RESULTS AND ANALYSIS

4.1 Validation of Gathering Source Code

This section will discuss the results of the approach from Section 3.1 corresponding to Sub-RQ1. The goal of this approach was to collect source code containing uses of *m*. Recall will be the most important metric to measure the success of this approach, because as many of the files that use *m* must be collected as possible. Following recall is (time) efficiency, as the files must be collected as quickly as possible. Beyond that, precision is not as important, since files without uses of *m* can always be filtered away later. However, a low precision can be devastating for the efficiency of the remainder of the steps.

The Sourcegraph approach will gather all imports of *m* in Sourcegraph data, resulting in a near perfect recall relative to Sourcegraph. However, the GitHub Search API seems to suggest that it has more data than Sourcegraph, albeit not by a significant margin: we estimate somewhere between 1.5 to 3 times as many results. This suggests that the overall recall of this gathering step is lower than if we had used strategy 3 described in Section 3.1. Consequently, future work can apply strategy 3 in order to improve the recall and collect more files with uses of *m*. However, the Sourcegraph approach will still gather hundreds of thousands of files for well-known modules, which ought to be sufficient if relative usage is most important (e.g. between sections of code, or between different modules).

This strategy performs wonderfully in terms of efficiency. All requests with less than 100,000 imports at a time took Sourcegraph less than 2 seconds, causing the efficiency to be simply a function of download speed. This strategy is able to search for and download hundreds of thousands of files per minute. That said, the Sourcegraph API may start returning a `504 Gateway Time-out` error above 250,000 imports at a time. Such high imports are rarely necessary, as most modules are never used that frequently.

Lastly, the precision of this step is good, but not perfect. This is tricky, as all downloaded files do use *m*, but not all files actually compile. Some contain syntax errors, and depending on the age of *m*, some files may be using Python 2, which cannot compile in Python 3. These unusable files are still downloaded, slowing down the next steps. We encountered precisions ranging between 0.87 (`nltk` with 200,000 imports) and 0.91 (`pandas` with 200,000 imports).

To conclude, the chosen approach using Sourcegraph is able to gather an amount of files within the same order of magnitude as the GitHub Search API approach. However, it can do so orders of magnitude more efficiently. As we are primarily interested in relative usage, not having all data will still allow us to get appropriate results. Furthermore, if this framework of determining how frequently Python objects are used is a part of a larger text mining task, then efficiency is certainly appreciated, for example when comparing the usage between dozens of open-source Python modules. We are more than satisfied with the approach chosen, and are happy to have outlined the alternatives for future work.

4.2 Validation of Extracting Functionality Dependencies

This section discusses the results of the approach from Section 3.2 corresponding to Sub-RQ2. The goal of this approach was to extract functionality dependencies of *m* from *f* on a function, class and variable level. The correctness of this step was validated through unit testing.

The three primary ways of importing objects in Python were considered, as can be seen in Appendix 3. Variations of these import statements were made using Table 2, resulting in a near complete test set of import statements. This functionality extraction approach was able to recognise the uses of nearly all imported objects with two exceptions:

- Objects imported using wildcard imports or using `__import__()` could not be traced back to their module of origin.

- Objects that were imported, but then overridden by some other value, were still wrongfully traced back to their module of origin.

All of these issues could be avoided by using the `astroid` strategy instead, although that brings along other problems of its own. Beyond accuracy, this step is significantly slower than either of the other steps. According to profiling, this is largely attributed to parsing Python files into AST, which means there is not much performance to be gained without moving away from `ast`.

All in all, these flaws correspond to only a small number of cases, and only reduce the precision and recall marginally. Furthermore, these edge cases are all against the Python guidelines. Because these issues should apply with equivalent frequency for all objects, this should not have significant effects on the resulting usage distribution.

That said, we conclude that future work should look into the `astroid` strategy. In contrast to Section 4.1, the difference in time-efficiency is within the same order of magnitude. In practice, people write code that does not follow the Python guidelines, and that code should be parsable too.

4.3 Validation of Post-processing

The optional post-processing applied to extend e.g. `nltk.word_tokenize` into `nltk.tokenize.word_tokenize` is not without flaws. It works perfectly for the large majority of projects, but it fails on projects that have compatibility submodules, like `tensorflow`.

For that module, `tensorflow.float32` will extend into `tensorflow.compat.v1.dtypes.float32`, rather than `tensorflow.python.framework.dtypes.float32`. This issue would also be resolved if `astroid` would be used for functionality dependency extraction, as this post-processing extension would not be necessary.

4.4 Overall results

This work produced `module_dependencies`, which combined both Sub-RQ1 and Sub-RQ2 in order to create a procedure to answer the RQ, by following Figure 1. This module has been published on PyPI[3].

A sample usage has been included in Listing 4, which covers an example text mining task about the relative usage of NLTK objects. The usage of the objects of the Python module `'nltk'` was extracted in just ~5 minutes, by downloading and parsing 41,380 Python files and 9,235 Jupyter notebooks. In total, 113,156 uses of objects from NLTK was discovered, as can be seen in Listing 5 and Figure 2. From Listing 5 we can conclude that `word_tokenize` is the most frequently used object from `'nltk'`, and that the `tokenize`, `corpus` and `stem` submodules are popular. Figure 2 agrees with this sentiment, and shows the large disparity between these popular submodules and some less popular ones like `'classify'` and `'sentiment'`.

With this knowledge that can be extracted by using `module_dependencies` as a step in a text mining task, it will be possible to answer complex text mining questions about the (relative) usage frequency of code objects.

The impressive quantity of data afforded through Sourcegraph (Sub-RQ1) alongside the efficiency of the `ast` approach (Sub-RQ2) results in a very powerful program that is able to efficiently and effectively determine the usage of the objects from a Python module. This shows that the approach from Figure 1 and Section 3 is a very effective solution of the RQ.

[3]https://pypi.org/project/module-dependencies/

5 CONCLUSION

This work has introduced an approach for gathering source code on a large scale, and does so in an efficient and performant way. Although this method does not return as many results as using GitHub as a data source directly, it is in the same order of magnitude, while being considerably more time-efficient. We strongly recommend Sourcegraph as a source of data related to open-source repositories.

This work builds onto existing work using AST analysis using `ast`, which proved to be the most time-efficient approach of the two strategies mentioned in Section 3.2. That said, as this approach does not track which objects exist within our module of interest, this strategy is unable to handle wildcard imports, and may cause issues when naively guessing the origin of an used object.

We challenge future researchers to attempt to tackle the discovered downside of the `astroid` strategy as described in Section 3.2, and use inference to determine the origin of used objects. This would also allow tracking class method use - which the approach from this work cannot do.

This work proudly presents `module_dependencies`[4], a tool that is able to efficiently determine how frequently objects from a given Python module are used. Consequently, this module is useful as a step in text mining tasks, and in practice this tool shows developers exactly which sections of their modules are being used. This knowledge is crucial for determining the urgency and priority of specific sections of code, allowing limited manpower to be applied considerably more effectively.

ACKNOWLEDGMENTS

I would like to thank Steven Bird and Ilia Kurenkov of the NLTK Team for their interest and excitement surrounding this work and its results.

REFERENCES

[1] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit.* O'Reilly Media, Inc.

[2] GitHub. 2021. GitHub. https://github.com/search

[3] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236.

[4] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. 2014. Lean GHTorrent: GitHub Data on Demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. ACM, New York, NY, USA, 384–387.

[5] GraphQL. 2021. GraphQL Specification. https://spec.graphql.org/October2021/

[4]https://github.com/.../module_dependencies

[6] Eric Horton and Chris Parnin. 2018. Gistable: Evaluating the Executability of Python Code Snippets on GitHub. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 217–227.

[7] Eric Horton and Chris Parnin. 2019. DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 328–338.

[8] Eric Horton and Chris Parnin. 2019. V2: Fast Detection of Configuration Drift in Python. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 477–488.

[9] Saikat Mondal, Mohammad Masudur Rahman, and Chanchal K. Roy. 2019. Can Issues Reported at Stack Overflow Questions be Reproduced? An Exploratory Study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 479–489.

[10] Python Code Quality Authority. 2021. *astroid*. https://github.com/PyCQA/astroid

[11] Python Core Team. 2021. *ast — Abstract Syntax Trees*. Python Software Foundation. https://docs.python.org/3/library/ast.html

[12] Python Core Team. 2021. *Python: A dynamic, open source programming language*. Python Software Foundation. https://www.python.org/

[13] Sebastian Thiel. 2021. *GitPython*. https://github.com/gitpython-developers/GitPython

[14] Sourcegraph. 2021. Sourcegraph. https://sourcegraph.com

[15] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python Framework for Mining Software Repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 908–911. https://doi.org/10.1145/3236024.3264598

[16] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring Execution Environments of Jupyter Notebooks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1622–1633.

A APPENDIX

A.1 Sourcegraph GraphQL

```
query ($query: String!) {
  search(query: $query, version: V2) {
    results {
      results {
        __typename
        ... on FileMatch {
          ...FileMatchFields
        }
      }
      repositoriesCount
      limitHit
      cloning {
        name
      }
      missing {
        name
      }
      timedout {
        name
      }
      matchCount
      elapsedMilliseconds
      ...SearchResultsAlertFields
    }
  }
}

fragment FileMatchFields on FileMatch {
  repository {
    name
    description
    stars
    isFork
  }
  file {
    name
    path
    url
    content
  }
}

fragment SearchResultsAlertFields on SearchResults {
  alert {
    title
    description
    proposedQueries {
      description
      query
    }
  }
}
```

Listing 2: SourceGraph GraphQL Query used in `module_dependencies`.

Query Parameter	Description
'context'	This parameter specifies whether the search query considers all code or only code that is owned by the current user. This work uses the 'global' context, corresponding to the first option.
'count'	This parameter specifies the number of results to return. This work uses a default value of 25000 results, but allows users to modify this value to their likings.
'timeout'	The maximum amount of time that Sourcegraph may continue to search for results. This work uses a default value of '10s'.
'patterntype'	Either 'literal', 'regexp' or 'structural', corresponding to the type of search term that is being searched with. This work uses 'regexp'.
'language'	The programming language to filter on. This work uses both '"Python"' and '"Jupyter Notebook"' for this field.
'-file'	This represents the negation of 'file' and means that any matching file will be excluded from search. This work uses 'site-packages/' as a means to exclude files that originate from a virtual environment.

Table 1: Sourcegraph GraphQL Search parameters.

A.2 Import statement types

```
import module
from module import object
from module import *
```

Listing 3: The three primary ways of importing objects in Python

Type of modification	Examples
Relative import	.module or ..module instead of module.
Submodule import	module.submodule instead of module.
Multiple import	object, foo, bar instead of object.
Alias import	object as obj instead of object.

Table 2: Modifications on the main primary of importing objects in Python from Listing 3.

A.3 Usage of `module_dependencies`

```
from module_dependencies import Module
from pprint import pprint

# Attempt to find "all" imports of the "nltk" module
# in both Python files and Jupyter Notebooks each
module = Module("nltk", count="all")
pprint(module.usage()[:15])
print(f"module_dependencies found {module.n_uses()} uses!")
module.plot()
```

Listing 4: Example usage of `module_dependencies`.

```
Fetching Python code containing imports of `nltk`...
Fetched Python code containing imports of `nltk`
Parsing 479,005,314 bytes of Python code as JSON...
Parsed 479,005,314 bytes of Python code as JSON...
Extracting dependencies of 41,380 files of Python code...
Parsing Files: 100% 41380/41380 [03:49<00:00, 180.50files/s]

Extracted dependencies of 41,380 files of Python code.
Fetching Jupyter Notebooks containing imports of `nltk`...
Fetched Jupyter Notebooks containing imports of `nltk`
Parsing 625,320,071 bytes of Jupyter Notebook as JSON...
Parsed 625,320,071 bytes of Jupyter Notebook as JSON...
Extracting dependencies of 9,235 Jupyter Notebooks...
Parsing Files: 100% 9235/9235 [01:18<00:00, 118.07files/s]
Extracted dependencies of 9,235 Jupyter Notebooks.

[('nltk.tokenize.word_tokenize', 11629),
 ('nltk.corpus.stopwords.words', 9520),
 ('nltk.downloader.download', 5493),
 ('nltk.tokenize.sent_tokenize', 4022),
 ('nltk.stem.wordnet.WordNetLemmatizer', 3822),
 ('nltk.tag.pos_tag', 3118),
 ('nltk.stem.porter.PorterStemmer', 3072),
 ('nltk.probability.FreqDist', 1950),
 ('nltk.stem.snowball.SnowballStemmer', 1670),
 ('nltk.tree.tree.Tree', 1566),
 ('nltk.data.load', 1498),
 ('nltk.tokenize.regexp.RegexpTokenizer', 1461),
 ('nltk.compat.python_2_unicode_compatible', 1405),
 ('nltk.corpus.wordnet.synsets', 1288),
 ('nltk.util.ngrams', 1007)]
module_dependencies found 113156 uses!
```

Listing 5: Output of example usage of `module_dependencies` from Listing 4, with the output reformatted for this listing.

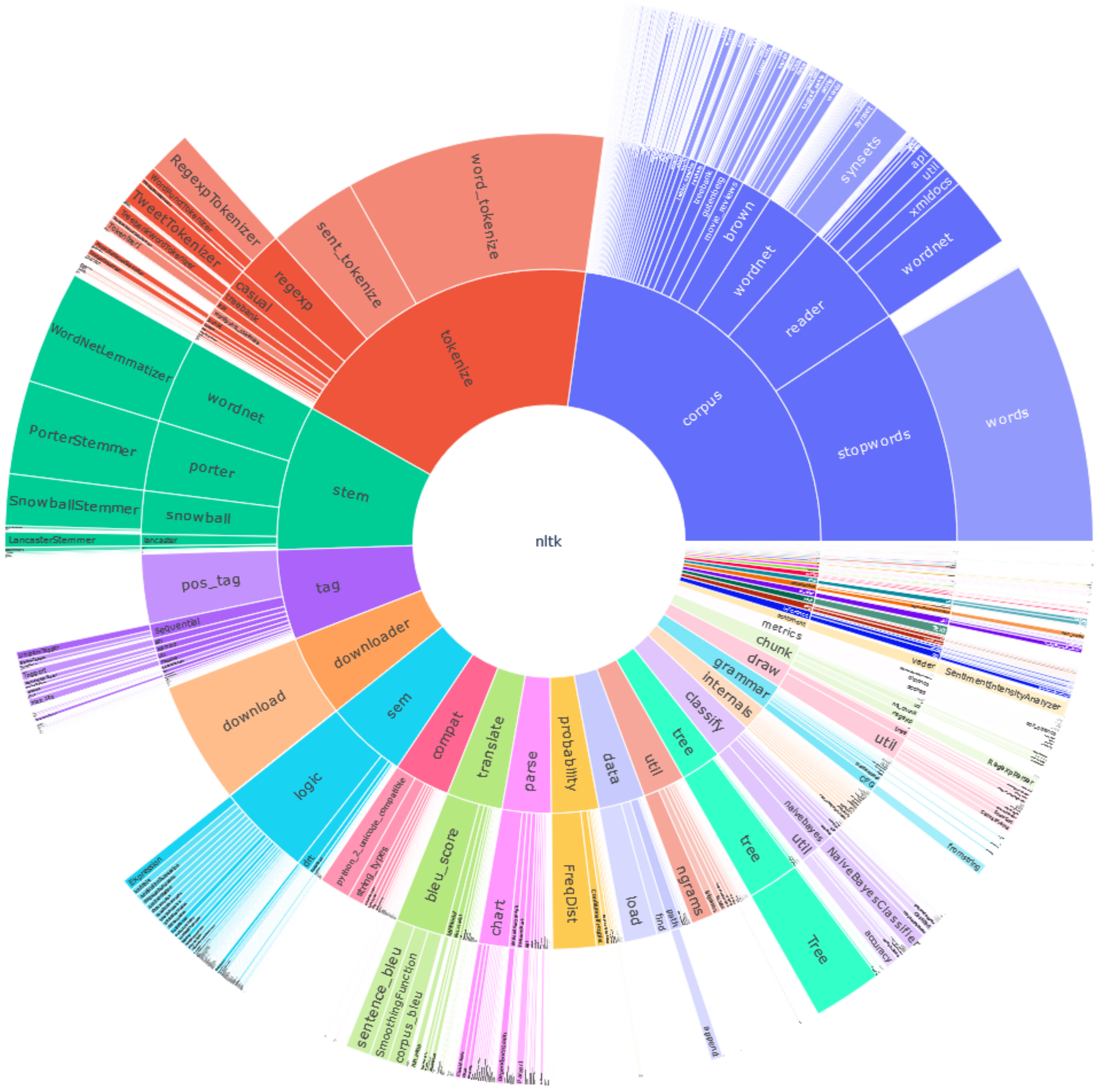

Figure 2: A static picture of the interactive sunburst plot showing the usage of objects from NLTK, as generated with `module_-dependencies`.