SPYSE - A Semantic Search Engine for Python Packages and Modules

Shiva Krishna Imminni, Mir Anamul Hasan, Michael Duckett
Puneet Sachdeva, Sudipta Karmakar, Piyush Kumar, Sonia Haiduc
Florida State University, Tallahassee, FL 32303, USA

ABSTRACT

Code reuse is a common practice among software developers, whether novices or experts. Developers often rely on online resources in order to find code to reuse. For Python, the Python Package Index (PyPI) contains all packages developed for the community and is the largest catalog of reusable, open source packages developers can consult. While a valuable resource, the state of the art PyPI search has very limited capabilities, making it hard for developers to find useful, high quality Python code to use for their task at hand.

We introduce SPYSE (Semantic PYthon Search Engine), a web-based search engine that overcomes the limitations of the state of the art, making it easier for developers to find useful code. The power of SPYSE lays in the combination of three different aspects meant to provide developers with relevant, and at the same time high quality code: code semantics, popularity, and code quality. SPYSE also allows searching for modules, in addition to packages, which opens new reuse opportunities for developers, currently not supported.

TOOL URL: <https://pypi.compgeom.com>

VIDEO URL: <https://youtu.be/Praglw-vS50>

CCS Concepts

•Software and its engineering → Software maintenance tools; Reusability; •Information systems → Search engine architectures and scalability;

Keywords

Search Engine; Code Reuse; Python; Recommender Systems

1. INTRODUCTION

In order to streamline and speed up development many programmers reuse existing code, especially third party libraries [5]. Searching for code is a frequent activity, and research has shown that developers perform up to twelve code searches on average every day [4]. Code reuse can reduce

coding time and effort, and can lead to higher quality software. Therefore it is important to offer tools that make the search for code as easy and flexible as possible for developers.

For Python, the main resource where developers can find code to reuse is the Python Package Index (PyPI)¹. PyPI contains a database of almost 70,000 Python packages that users can query using a simple keyword-based search. To find a package, developers need to know its exact name or a specific keyword that matches its description, which limits the code that can be retrieved. In addition, the current search functionality in PyPI does not accommodate cases where a developer may only be looking for a code snippet, such as a Python module, instead of a whole package.

Other code search engines exist, such as, Krugle², searchcode³ and BlackDuck⁴, which accommodate multiple programming languages, including Python. While these allow for searching code at a smaller granularity level than PyPI, such as a module, they require the developer to know exactly (part of) the name of the module they are looking for. Also, they do not support more “semantic” searches where the developer is looking for a good package for a particular domain (e.g., “3d graphics”), without knowing its name.

We propose SPYSE (Semantic PYthon Search Engine)⁵, a free, web-based search engine for Python code that aims to overcome the current limitations of the state of the art and offer developers more flexibility, as well as guidance in finding good Python packages and modules for reuse.

SPYSE’s power stands in its combination of various metrics and heuristics. The first type of heuristics are related to semantics. SPYSE uses two search engines: Elasticsearch, and Bing, in order to leverage the semantic information found in the metadata (e.g., name, description) and source code (i.e., identifiers and comments) of the Python packages and modules found in the PyPI repository. Therefore, when a developer formulates a specific query, SPYSE will employ the search engines to find relevant packages or modules.

While research in software engineering has also led to a few large scale search engines [1, 3], which are more semantics-oriented than the commercial search engines, none has indexed Python code. More than that, SPYSE goes beyond any existing code search engine in industry and academia by employing two other types of metrics which guide developers towards high quality code. These metrics include the number

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-4205-6/16/05...\$15.00

DOI: <http://dx.doi.org/10.1145/2889160.2889174>

¹<https://pypi.python.org/pypi>

²<http://www.krugle.com/>

³<https://searchcode.com/>

⁴<https://code.openhub.net/>

⁵<https://pypi.compgeom.com>

of downloads for a package, indicating popularity (used as a proxy for quality), and code quality metrics, such as the number of warnings and lines of comments to code ratio. This last set of metrics represents a unique feature of SPYSE, which is, to our knowledge, the first code search engine that, in addition to finding relevant code to a query, tries to guide developers towards higher quality code to reuse. We believe this feature can be particularly beneficial for developers that are new to Python and the packages available in PyPI.

2. ARCHITECTURE

SPYSE is organized into three separate components: Data Collection, Data Indexing and Data Presentation. They are described in detail below.

2.1 Data Collection

2.1.1 Package Level Search

PyPI allows access to the metadata of a package via an http request which returns a JSON file with keys such as description, author, package_url, downloads.last_month, etc. We use this to collect metadata for all packages in PyPI.

We also extracted data related to the code quality of each Python package. First, we check if a package is following the PEP 0008 – Style Guide for Python Code⁶. PEP8⁷ is a simple Python module that validates any Python code against the PEP 8 style guide. Pylint⁸ is another such tool that checks for line length, variable names, unused imports, duplicate code and other coding standards against PEP 8. We used Prospector⁹, which brings together both the functionality of PEP8 and Pylint and also adds the McCabe¹⁰ code complexity metric. When a package is processed with Prospector, we get a count of errors, warnings, messages and their description. We use this information as an approximation of code quality.

Developers also care about how well the code is commented. The ratio of the number of comment lines to the total number of lines of code, number of code lines to the total number of lines and number of warnings to the number of lines, represent other metrics to indicate aspects of code quality that we use. We used CLOC¹¹ to get the total number of files, number of comments, number of blank lines and number of code lines for each Python package. We collect all this information to form a JSON document for each Python package.

2.1.2 Module Level Search

A module is a file with the extension “.py” containing Python definitions and statements¹². It is a collection of multiple programming units such as classes, methods and variables. Some developers are interested in searching for these programming entities in a module, so we built SPYSE to support these use cases. To enable this type of search, we have to extract useful information from the modules of each package, i.e., get a list of programming entities for each module. We are interested in classes, functions and variables in classes, global functions, recurring inner functions inside

global functions, variables inside global functions and global variables. We maintain each type using a separate key so more weight can be given to certain entities if needed.

We used Bandersnatch¹³ to clone an entire PyPI repository. To collect the required information, we iterate through all packages in a repository and within each package we iterate through all modules. For each module, we construct and traverse the Abstract Syntax Tree using the ast¹⁴ module.

Similar to packages, we collect code quality metrics for modules using Pylint and CLOC (we could not use Prospector on modules). We store all information for a module in a JSON document, one for each Python module in a package.

2.2 Data Indexing and Searching

We used Elasticsearch (ES)¹⁵, an open source search engine built on top of Apache Lucene to index our data and query the indexed data for both package level search and module level search. FileSystem River (FSRiver)¹⁶, an ES plugin, is used to index the documents.

2.2.1 Data Indexing

For package level search, the extracted data for each Python package is considered a document and indexed. All fields in a document are indexed, but the following fields are first analyzed using the ES Snowball analyzer¹⁷: name, summary, keywords and description. The analyzer normalizes the text by generating tokens, removing English stop words, transforming everything to lowercase, and stemming the remaining words. This improves the matching during a search. The other fields are not analyzed before indexing either because they are numbers or are of no interest with respect to the search query.

For the module search, the extracted data for each module in a Python package is considered as a document and indexed. All fields except the module_path are normalized using a custom analyzer before they are indexed, to improve the search. The analyzer generates tokens using a custom tokenizer that splits identifiers by underscore, camel case and numbers, and preserves also the original token. Then it uses the ES Snowball filter¹⁸. The module_path field is not analyzed as it is not used for matching a search query.

2.2.2 Data Searching

We define our search query according to ES Query DSL to look for matches in the index for user queries. ES uses a combination of the Boolean model and the vector space model to find matches for a query, but also adds modern features like a coordination factor, field length normalization, and term or query clause boosting¹⁹.

For package level search, ES looks for matches for the user query in the following fields: name, author, summary, description and keywords. Based on the matches, it ranks the results and returns name, author, summary, description,

¹³<https://pypi.python.org/pypi/bandersnatch>

¹⁴<https://docs.python.org/2/library/ast.html>

¹⁵<https://www.elastic.co/products/elasticsearch>

¹⁶<https://github.com/dadoonet/fsriver>

¹⁷<https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-snowball-analyzer.html>

¹⁸<https://www.elastic.co/guide/en/elasticsearch/reference/1.4/analysis-snowball-tokenfilter.html>

¹⁹<https://www.elastic.co/guide/en/elasticsearch/guide/current/scoring-theory.html>

⁶<https://www.python.org/dev/peps/pep-0008/>

⁷<https://pypi.python.org/pypi/pep8>

⁸<http://www.pylint.org/>

⁹<https://github.com/landscapeio/prospector>

¹⁰<https://pypi.python.org/pypi/mccabe>

¹¹<http://cloc.sourceforge.net/>

¹²<https://docs.python.org/3/tutorial/modules.html/>

Figure 1: SPYSE: Homepage.

version, keywords, number of downloads in the last month for each top n ranked Python package, where n is the number of matching packages requested. The summary and description of a matched package are returned and the matches to the query are highlighted in the user interface while displaying the results.

For module level search, ES looks for matches to the user query in the following fields: class, class_function, class_variable, function, function_function, function_var and variable of a module document. Different weights can be assigned to matches in different fields based on the fields' importance. The weights assigned to each field can be configured in SPYSE. In our current implementation, a match in the class field is considered more important (i.e., having a higher weight) than a match in the function field in a module. Also, a match in the function defined within a class(class_function) or another function(function_function) field is considered less important than a match in the global function(function) field in a module. Also, a match in a variable defined in a class(class_variable) or another function(function_var) will have less weight compared to a match in the global variable(variable) field in a module. Based on the matches, SPYSE ranks the results and returns the path to the module(module_path) where the match occurred. Using this information, we retrieve the module .py file and extract the relevant code segments for each matching module.

We also make use of Bing in order to search PyPI for packages and include the results returned by Bing in determining the final results displayed to the developer. The results returned by ES and Bing are combined into a unique list, and then this information is combined with code quality information to create the final list of results.

2.3 Data Presentation

2.3.1 Browser Interface

The interface features a home page with a simple text box for queries and a choice for either package or module level

search. The results page displays the query information at the top, and the results below. The user can modify her query or change the search type from package level search to module level search and vice versa. Refer to Fig. 1 for a screen shot of the homepage.

Figure 2: SPYSE: Package Level Search.

2.3.2 Package Level Search

When a user sends a query to the server for package level search, the query is processed, and a ranked list of packages is sent back. Each package is depicted in the browser as a tile (Fig. 2). The tile provides information about the package that includes the package name, the author of the package, a brief description, the number of downloads and the score assigned by the ranking algorithm. The user has the option to click the tile to view more information about the package and also visit PyPI and other sites related to the package. When the user clicks on the tile, a window is opened (Fig. 3) that contains the detailed description, version, source code homepage, PyPI homepage, score from ranking algorithm, statistics of the package as a bar graph, pie chart and numbers, and other packages from that author.

Figure 3: SPYSE: Package statistics.

2.3.3 Module Level Search

In the module level search (Fig. 4) the server returns a list of code excerpts. Users can then click on a code excerpt, and a window pops up that displays the entire code of the module. This gives users the opportunity to see the entire module should they need more information about it.

We limited the number of code snippets that are returned to 20. Sending more than 20 matches will increase the response time, while people generally do not look at more than the first ten results [2].

Figure 4: SPYSE: Module Level Search.

3. PRELIMINARY AND PLANNED EVALUATION

At this stage of the project we were mostly interested in implementing a fast, stable, and usable version of SPYSE, which can already be used by developers for searching for high quality Python packages and modules. A very preliminary evaluation was performed by one of the authors by comparing the results of SPYSE to those of PyPI’s embedded search functionality. More precisely, five queries representing popular Python package names and five generic queries expressing the purpose of the packages being searched for were executed. The results of the search were analyzed by one of the authors. The results are summarized below. We plan a large scale evaluation with Python developers, both novice and experienced, as the next step in this line of research.

We generally noticed the following:

1. PyPI returns significantly more results than SPYSE, sometimes by orders of magnitude. In one example, when searching for the package “Django”, PyPI returned 11,292 results, representing approximately $\frac{1}{4}$ of the total number of packages on PyPI. On the other hand, SPYSE always returns a maximum of 40 results, in order not to overwhelm the user. Our choice of displaying a limited number of results was based on the

fact that on Google 81% of users view only one results page (10 items) [2].

2. PyPI often assigns the same score for a large number of packages, making it difficult for developers to distinguish the most relevant packages from less relevant ones. For example, when searching for “flask”, PyPI assigned the highest score to 162 packages. SPYSE always assigns the highest score for only one package.
3. For the queries representing exact package names, PyPI retrieved the respective packages on the first position only in 2 out of 5 cases. On the other hand, SPYSE exhibited this behavior in all 5 cases.
4. For the five generic keywords, SPYSE’s results were always more relevant than the ones returned by PyPI. For example, for the keyword “Web Development Framework”, SPYSE returned all unique results with packages like Django and Pyramid (widely used web development frameworks) in the top 5. On the other hand, PyPI returned only 3 unique results (it returned various versions of the same packages; SPYSE always returns unique results), and these were related to web testing rather than web development frameworks.

4. CONCLUSION

SPYSE is designed to be a semantic and scalable search engine for Python code, powered by Elasticsearch. The main goal of SPYSE is to replace PyPI’s search engine, so that it can return high quality search results for Python developers. SPYSE uses various code metrics, the results of two search engines like Bing and Elasticsearch, as well as popularity and code quality metrics to provide high quality search results. It is a complete system that acquires, analyzes, and indexes a large number of Python libraries periodically, enabling fast and semantic searches.

5. ACKNOWLEDGEMENTS

Sonia Haiduc is supported in part by the National Science Foundation grant CCF-1526929. Piyush Kumar is partially supported by CompGeom Inc. and by the Fulbright Scholar Program.

6. REFERENCES

- [1] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comp. Prog.*, 79:241–259, 2014.
- [2] B. Jansen and A. Spink. How are we searching the www? a comparison of nine search engine transaction logs. *Info. Proc. & Mgmt.*, 42(1):248–263, 2006.
- [3] C. McMillan, D. Poshyanyk, M. Grechanik, Q. Xie, and C. Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol.*, 22(4):37, 2013.
- [4] C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: a case study. In *Proc. of ESEC-FSE*, pages 191–201, 2015.
- [5] A. Zaimi, N. Triantafyllidou, A. Mavridis, T. Chaikalis, I. Deligiannis, P. Sfetsos, and I. Stamelos. An empirical study on the reuse of third-party libraries in open-source software development. In *Proc. of the 7th Balkan Conf. on Informatics*, page 4. ACM, 2015.