LiFUSO: A Tool for Library Feature Unveiling based on Stack Overflow Posts

Camilo Velázquez-Rodríguez
Vrije Universiteit Brussel
Brussels, Belgium
camilo.ernesto.velazquez.rodriguez@vub.be

Eleni Constantinou
Eindhoven University of Technology
Eindhoven, The Netherlands
e.constantinou@tue.nl

Coen De Roover
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

***Abstract*—Selecting a library from a vast ecosystem can be a daunting task. The libraries are not only numerous, but they also lack an enumeration of the features they offer. A feature enumeration for each library in an ecosystem would help developers select the most appropriate library for the task at hand. Within this enumeration, a library feature could take the form of a brief description together with the API references through which the feature can be reused. This paper presents LiFUSO, a tool that leverages Stack Overflow posts to compute a list of such features for a given library. Each feature corresponds to a cluster of related API references based on the similarity of the Stack Overflow posts in which they occur. Once LiFUSO has extracted such a cluster of posts, it applies natural language processing to describe the corresponding feature. We describe the engineering aspects of the tool, and illustrate its usage through a preliminary case study in which we compare the features uncovered for two competing libraries within the same domain. An executable version of the tool is available at https://github.com/softwarelanguageslab/lifuso and its demonstration video is accessible at https://youtu.be/tDE1LWa86cA.**

***Index Terms*—software ecosystems, features, libraries, Stack Overflow**

I. Introduction

Reusing functionality provided by third-party libraries has become common practice [1]. For most programming languages, libraries nowadays form vast ecosystems supported by central repositories such as Maven Central and NPM Registry. As the number of libraries grows, some are bound to offer similar features [2]. For example, Maven Central has at the moment of writing, 73 different libraries that provide reusable implementations of collection data types.[1] From the brief description of each library (e.g., README files or introductory texts on websites of libraries), one can infer that they provide different kinds of collections (e.g., maps, sets, queues), but the range of features offered is not immediately clear (e.g., specific utility methods for sorting, reversing, filtering or transforming collections), neither how these features should be used (e.g., a single static method call, an instance method of a class that needs to be instantiated, three methods needed to be called together), nor how each library compares to others on the offered features.

At selection time, developers need to select one or more libraries from a solution domain (e.g., PDF generation, mocking). Current tool support is often based on popularity metrics (e.g., number of stars or downloads) [3] instead of on the actual features that libraries have to offer [4]. As a consequence, less popular libraries that offer more features or features that are easier to use might never get selected.

As a foundation for tools that support developers in feature-based library selection, we have proposed a data-driven approach to uncovering library features from Stack Overflow (SO) posts [5]. SO posts with library usages represent a unique opportunity for API mining because of their focus on a particular task. Additionally, these posts contain natural language descriptions around code snippets, adding meaning to the referenced code. Our approach computes features as clusters of API references labelled by a description of the functionality they implement. In this follow-up tool paper, we present LiFUSO, the first tool to combine a prototype implementation of the approach with a graphical user interface. We explore the engineering aspects of LiFUSO and present a case study illustrating the intended usage of the tool.

II. LiFUSO

This section describes the architecture of LiFUSO and the main design choices made in its implementation. We refer the reader to Velázquez-Rodríguez et al. [5] for a detailed description and evaluation of the approach implemented by LiFUSO. Figure 1 depicts a summary of how the approach computes the features for a given library, and how LiFUSO visualises them. Section II-B will give insights into the implementation of these steps.

A. Summary of the Approach

The input of LiFUSO is a target library for which features need to be computed. Our tool requires that the *groupID* and *artifactID* (e.g., `com.google.guava` and `guava` respectively) of at least one version of the library is available from the Maven Central repository. The name of the target library also needs to be a valid SO tag.[2]

The first step *collects* the names of API elements (i.e., public classes, methods, fields) as well as answers from SO that might contain a usage of these elements by downloading all versions of the library from Maven and processing their bytecode.

[1] https://mvnrepository.com/open-source/collections

[2] https://stackoverflow.com/tags

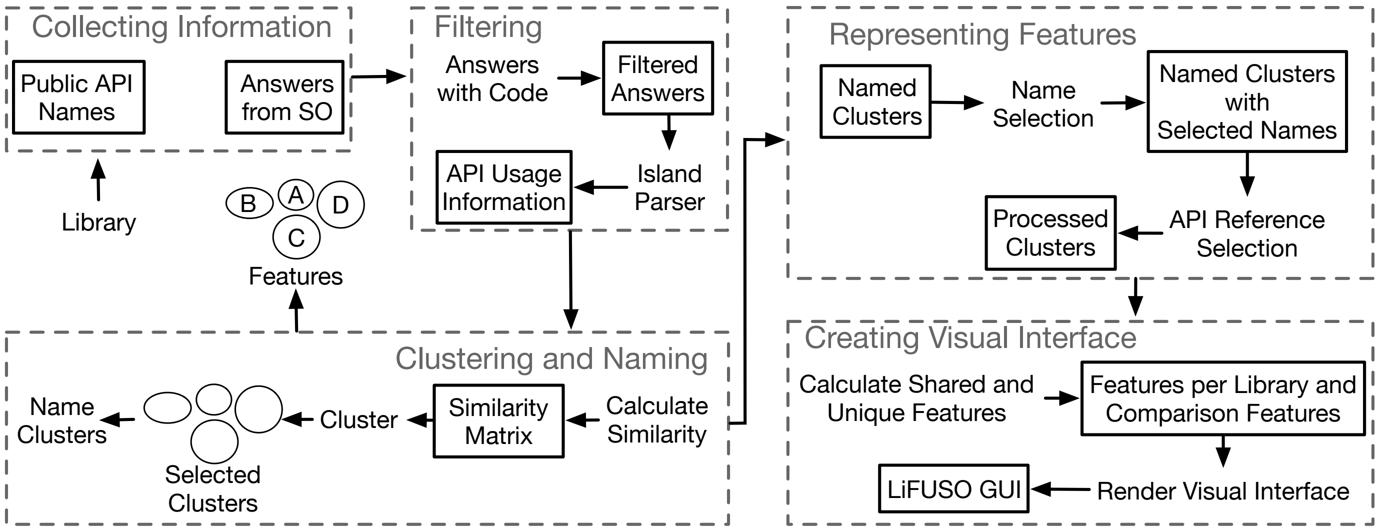


Fig. 1. Summary of the approach implemented by LiFUSO.

SO answers are obtained from the SOTorrent dataset [6][3] by searching posts with the name of the library among their tags.

The subsequent *Filtering* step excludes answers without any code snippets as we are interested in showing code usages with an associated name to potential users. As SO code snippets are not necessarily complete nor syntactically correct, LiFUSO relies on a robust parser generated by a custom-built island grammar to this end. Our custom parser focuses on the syntactic constructs in which method invocations can occur. For a single invocation within a variable declaration or expression statement, it produces the name of the statically declared type of the receiver expression followed by the name of the invoked method (e.g., `JSoup.parse`). For a chain of successive invocations, common for libraries with a fluent API, it produces the statically declared type of the first receiver expression followed by the names of the successively invoked methods (e.g., `Document.body.text`). In case a code snippet contains API references from the library under analysis, the extracted usages, as well as their surrounding text, are stored for further processing.

The *Clustering and Naming* step first constructs a matrix of the Jaccard similarity between all SO answers based on their usage of the library's API elements. Next, it applies hierarchical clustering to the matrix and by means of a dynamic cut tree technique [7] selects the optimal cutting point to form clusters. The resulting clusters are groups of which the elements are close to one another yet far from the elements in other clusters. Using the local outlier factor (LOF) [8] technique, the most frequent API elements within the cluster are identified (i.e., the local outliers). In case LOF cannot determine such elements, the cluster is discarded.

A semantic tree [9] is computed for the sentences in the title of and in the text surrounding each SO answer. For each noun and verb, the direct typed dependencies on another verb or noun are analysed to extract pairs of the form noun-verb or verb-noun. The most frequent verbs and nouns in the pairs (using LOF once again) are retained as candidates for the name of the feature.

Although the clusters have now been computed according to our previously-published approach, the LiFUSO tool still needs a way to represent them to the developer. In step *Representing Features*, LiFUSO selects the top-5 most frequent pairs as the representation of the name of the feature. We found that selecting more frequent pairs (e.g., top-10, top-20) hinders discerning interesting features among several displayed next to one another, and encumbers understanding individual features displayed in isolation. As the representation for the API references in the clustered snippets, LiFUSO displays only those extracted through the LOF technique in a previous step. As such, clusters with the most frequent name pairs and API references are outputted.

The actual visualisation is produced in the *Creating Visual Interface* step. Depending on the view selected by the developer, the features common to a set of libraries as well as features unique among a set of libraries are computed and displayed. Individual features can be inspected in detail too.

B. Implementation

LiFUSO implements the above approach using a series of components, each responsible for a separate data processing step. All steps up to and including the computation of the shared and unique features are pre-computed and their results are persisted. The tool's graphical user interface (GUI) uses the pre-computed information, so navigation is swift. The following choices were made in the implementation of each component:

Collecting Public API Names Implemented in Scala, this component uses the *requests* library[4] to query Maven

[3]November 16th, 2020 version from https://zenodo.org/record/4287411

[4]https://github.com/com-lihaoyi/requests-scala

Central for all versions of a library. Once their JAR files have been downloaded, the names of the public API elements are extracted by processing their bytecode using the *Apache BCEL*[5] library.

Collecting SO Answers with Code The Scala implementation of this component uses the *JSoup*[6] library to process the HTML of each SO answer that has the name of the library among its tags. Answers without a `code` block are discarded.

Extracting Natural Language Terms The *Java* implementation of this component uses the *Stanford NLP Toolkit*[7] to extract the natural language terms from the title of and the text surrounding the remaining SO answers.

Extracting API Usage Information We constructed an island parser with the help of the *parboiled*[8] library. Using the API usage information extracted by the parser for each SO answer, we used the Jaccard similarity to construct a similarity matrix.

Clustering To compute the clusters given the similarity matrix, we used the `hclust` function in the *stats* package from the *R* standard library and relied on the dynamic tree cut feature of the third-party *dynamicTreeCut*[9] library.

Cluster Selection and Naming To discard low-quality clusters without frequent API references, we used the LOF implementation provided by the *scikit-learn*[10] library of *Python*. We rely on *reticulate*[11] for the interoperability between *R* and Python. All other data between components is exchanged through CSV and TXT files. The previously extracted natural language terms are now used to name each remaining cluster.

Extracting Feature Representations As mentioned, the top-5 most frequent verb-noun pairs are extracted as the name representation whereas API references filtered by LOF are selected for the code part of the feature. The usage frequency of the API references in all clustered code snippets is also calculated and included in the feature information. We implemented these frequency calculations using the *R* programming language.

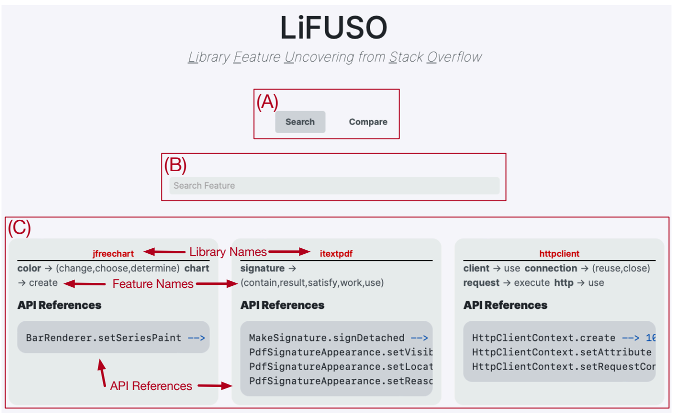
Calculating Common and Unique Features Every pair of two libraries in the ecosystem is considered and their features are compared. The comparison focuses on the names produced for their features: if two features from different libraries have a common name (i.e., a verb-noun pair), then these features are included in those considered shared by the two libraries. Conversely, if no other library has a feature with the same name, then that feature is considered unique to the library under analysis. We implemented this comparison in the *Julia* programming language.

[5]https://commons.apache.org/proper/commons-bcel/
[6]https://jsoup.org/
[7]https://nlp.stanford.edu/software/
[8]https://github.com/sirthias/parboiled
[9]https://cran.r-project.org/web/packages/dynamicTreeCut/
[10]https://scikit-learn.org/stable/index.html
[11]https://rstudio.github.io/reticulate/

Graphical User Interface We used the *Nuxt*[12] framework for *JavaScript* to implement the tool's graphical user interface. Through this interface, users can explore features present in an ecosystem and compare libraries to one another. Currently, the interface only supports selecting two libraries for comparison. Future improvements will support comparing a selection of multiple libraries.

The polyglot nature of our implementation is due to our decision to select a programming language and the third-party libraries for each component separately.

C. Graphical User Interface


LiFUSO
Library Feature Uncovering from Stack Overflow
(A)
Search
Compare
(B)
Search Feature
(C)
jfreechart
Library Names
itextpdf
httpclient
color → (change,choose,determine) chart → create
Feature Names
signature → (contain,result,satisfy,work,use)
client → use connection → (reuse,close) request → execute http → use
API References
API References
API References
API References

```
BarRenderer.setSeriesPaint -->
```

```
MakeSignature.signDetached -->
PdfSignatureAppearance.setVisib
PdfSignatureAppearance.setLocat
PdfSignatureAppearance.setReasc
```

```
HttpClientContext.create --> 1€
HttpClientContext.setAttribute
HttpClientContext.setRequestCor
```

Fig. 2. Front page of LiFUSO with the Search functionality tab activated.

Figure 2 depicts the main interface through which users can explore the features of a library. The tab bar surrounded by the rectangle labelled *A*, enables switching between the feature *Search* and the library *Compare* features of the tool. The *Search* tab is active by default and contains the elements with labels *B* and *C*. The search box with label *B* enables searching through the features uncovered for the ecosystem using natural language queries, which are matched against the feature names. Example queries include "collection", "filter collection", "create chart", or "extract image".

Features that match the query are displayed in a grid of feature views labelled *C*. Each individual feature view depicts the following information:

Library Name Centered in red at the top of the box (e.g., *jfreechart*).

Feature Name Collection of frequent verb-noun pairs describing the feature. Each noun is rendered in a bolder font, and followed by the verbs that are commonly used together with that noun.

API References The most frequent API references from SO posts to that feature are shown in a darker colour at the bottom of the box. The percentage shown next to each reference corresponds to the frequency at which the reference occurs within the SO posts clustered together for the feature.

[12]https://v3.nuxtjs.org

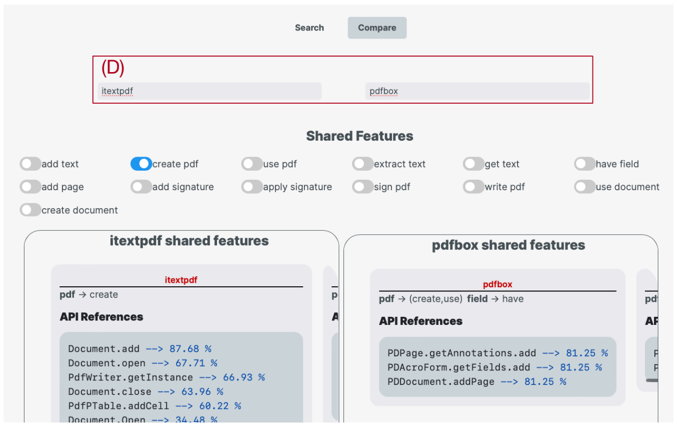
Figure 3 shows the *Compare* tab activated and its widgets. The two combo boxes in the rectangle labelled *D* are populated with the libraries in the ecosystem, and enable selecting the two libraries of which the features need to be compared.

III. Case Study

We now report on a small case study in which we use LiFUSO for a feature-based comparison of the *ITextPDF* and *PDFBox* libraries. While these libraries differ in their actual API, we expect them to offer similar features as both are intended for manipulating PDF files. We should therefore be able to spot shared natural language terms for the names of the features computed by LiFUSO, as well as some that are unique to each library.

A. Use of LiFUSO in the Case Study

We configured LiFUSO with the appropriate *groupID* and *artifactID* for the two libraries. The number of initial SO answers with at least one tag with the name of the libraries is 8,511 and 2,643 for *ITextPDF* and *PDFBox* respectively. The remaining number of answers after the processing described in Section II is 3,004 and 945 respectively.



Shared Features

add text, create pdf, use pdf, extract text, get text, have field, add page, add signature, apply signature, sign pdf, write pdf, use document, create document

itextpdf shared features

itextpdf
pdf → create
API References

```
Document.add --> 87.68 %
Document.open --> 67.71 %
PdfWriter.getInstance --> 66.93 %
Document.close --> 63.96 %
PdfPTable.addCell --> 60.22 %
Document.Open --> 34.48 %
```

pdfbox shared features

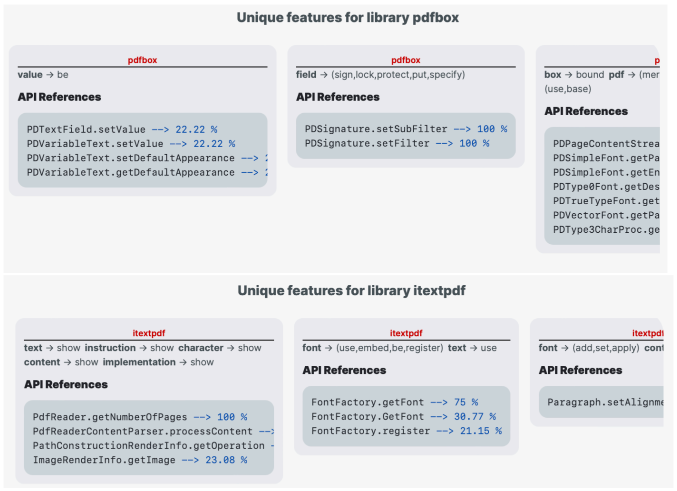
pdfbox
pdf → (create,use) field → have
API References

```
PDPage.getAnnotations.add --> 81.25 %
PDAcroForm.getFields.add --> 81.25 %
PDDocument.addPage --> 81.25 %
```

Fig. 3. Shared features for the studied libraries.

The tool was able to compute 70 features for *ITextPDF* and 36 features for *PDFBox*. We used the *Compare* tab of the GUI to select these two libraries. Figure 3 depicts features in common to each of the libraries. First, a list of checkboxes is displayed, each of which corresponds to a shared feature. The number of features being shared is 13. Upon any checkbox activation, the corresponding features in each library with the same name are shown in the view below the checkboxes. The left and right sides of the view depict the shared features of the library selected in the left and right combo box respectively. This view enables seeing how two competing libraries offer the same feature in different ways, and to inspect their API differences. In the case of Figure 3, the feature `create pdf` is shared by both libraries and is therefore displayed on both sides of the view. *ITextPDF* has four features that include `create pdf` as name, whereas *PDFBox* has two.

Several features are unique to *ITextPDF* and *PDFBox*, as depicted in Figure 4. In particular, the features about `font register`, `font use` and `show instruction` seem to be unique to the *ITextPDF* library. Alternatively, *PDFBox* appears to have some unique functionalities about the set of values for text fields and variables (e.g., `value be`) and the specification of a signature for a field (e.g., `sign field`, `put field`, `sign field`, `specify field`, etc.). Other interesting unique features of *ITextPDF* are `add cell`, `set appearance` and `read bookmark`. Conversely, `convert image`, `merge file`, and `replace image` seem unique to `PDFBox`.



Unique features for library pdfbox

pdfbox
value → be
API References

```
PDTextField.setValue --> 22.22 %
PDVariableText.setValue --> 22.22 %
PDVariableText.setDefaultAppearance --> 2
PDVariableText.getDefaultAppearance --> 2
```

pdfbox
field → (sign,lock,protect,put,specify)
API References

```
PDSignature.setSubFilter --> 100 %
PDSignature.setFilter --> 100 %
```

p
box → bound pdf → (mer
(use,base)
API References

```
PDPageContentStrea
PDSimpleFont.getPa
PDSimpleFont.getEn
PDType0Font.getDes
PDTrueTypeFont.get
PDVectorFont.getPa
PDType3CharProc.ge
```

Unique features for library itextpdf

itextpdf
text → show instruction → show character → show content → show implementation → show
API References

```
PdfReader.getNumberOfPages --> 100 %
PdfReaderContentParser.processContent -->
PathConstructionRenderInfo.getOperation -
ImageRenderInfo.getImage --> 23.08 %
```

itextpdf
font → (use,embed,be,register) text → use
API References

```
FontFactory.getFont --> 75 %
FontFactory.GetFont --> 30.77 %
FontFactory.register --> 21.15 %
```

itextpdf
font → (add,set,apply) cont
API References

```
Paragraph.setAlignme
```

Fig. 4. Unique features for the studied libraries.

To verify whether the unique features of a library are not implemented in the library being compared to, we still need to consult the developers of *both* libraries or to manually inspect their documentation. We leave this for our immediate future work. Note, however, that we have already compared the features computed for *PDFBox* against online tutorials for the library during the actual quantitative evaluation of the approach underlying the LiFUSO tool [5].

IV. Conclusion

We presented LiFUSO as a data-driven tool for enumerating the features offered by the libraries in a software ecosystem. To this end, the tool processes Stack Overflow posts, including the API references within their snippets and the natural language terms in the surrounding sentences. LiFUSO extracts the API references by means of a tailor-made island parser. Similarities between the API references within the snippets are calculated to finally return features as clusters, named according to the most frequent noun-verb pairs in the surrounding sentences. In this tool paper, we described the engineering aspects of the tool and reported on a small case study in which we conducted a feature-based comparison of two libraries in the same domain.

Acknowledgements

This research was carried out in the context of the Excellence of Science project 30446992 *SECO-ASSIST* financed by FWO-Vlaanderen and F.R.S.-FNRS.

References

[1] V. Bauer, L. Heinemann, and F. Deissenboeck, “A structured approach to assess third-party library usage,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 483–492.

[2] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik, *Studying Evolving Software Ecosystems based on Ecological Models*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 297–326. [Online]. Available: https://doi.org/10.1007/978-3-642-45398-4_10

[3] R. El-Hajj and S. Nadi, “LibComp: An IntelliJ Plugin for Comparing Java Libraries,” *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1591–1595, 2020.

[4] E. L. Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, “Selecting Third-Party Libraries: The Practitioners’ Perspective,” *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 245–256, 2020.

[5] C. Velázquez-Rodríguez, E. Constantinou, and C. De Roover, “Uncovering library features from API usage on Stack Overflow,” in *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022)*, 2022.

[6] S. Baltes, L. Dumani, C. Treude, and S. Diehl, “SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts,” in *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018, pp. 319–330.

[7] P. Langfelder, B. Zhang, and S. Horvath, “Defining clusters from a hierarchical cluster tree: the Dynamic Tree Cut package for R,” *Bioinformatics*, vol. 24, no. 5, pp. 719–720, 2008.

[8] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, “LOF: Identifying Density-Based Local Outliers,” in *The 2000 ACM SIGMOD International Conference on Management of Data*, 2000, pp. 93–104.

[9] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, “The Stanford CoreNLP natural language processing toolkit,” in *The 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2014, pp. 55–60.