Classification of APIs by Hierarchical Clustering

Johannes Härtel, Hakan Aksu, and Ralf Lämmel
University of Koblenz-Landau, Germany
Software Languages Team

ABSTRACT

APIs can be classified according to the programming domains (e.g., GUIs, databases, collections, or security) that they address. Such classification is vital in searching repositories (e.g., the Maven Central Repository for Java) and for understanding the technology stack used in software projects. We apply hierarchical clustering to a curated suite of Java APIs to compare the computed API clusters with preexisting API classifications. Clustering entails various parameters (e.g., the choice of IDF versus LSI versus LDA). We describe the corresponding variability in terms of a feature model. We exercise all possible configurations to determine the maximum correlation with respect to two baselines: i) a smaller suite of APIs manually classified in previous research; ii) a larger suite of APIs from the Maven Central Repository, thereby taking advantage of crowd-sourced classification while relying on a threshold-based approach for identifying important APIs and versions thereof, subject to an API dependency analysis on GitHub. We discuss the configurations found in this way and we examine the influence of particular features on the correlation between computed clusters and baselines. To this end, we also leverage interactive exploration of the parameter space and the resulting dendrograms. In this manner, we can also identify issues with the use of classifiers (e.g., missing classifiers) in the baselines and limitations of the clustering approach.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; *Abstraction, modeling and modularity*;

KEYWORDS

APIs. Hierarchical clustering. Feature modeling. Maven Central Repository. GitHub. Clustering exploration.

ACM Reference Format:
Johannes Härtel, Hakan Aksu, and Ralf Lämmel. 2018. Classification of APIs by Hierarchical Clustering. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension, May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196321.3196344

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 the author(s) 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.
ICPC '18, May 27–28, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196344

1 INTRODUCTION

The use of APIs (application programming interfaces) or, in fact, the use of the corresponding libraries and frameworks, is an essential element of today's software development. Each API addresses a certain programming domain, e.g., GUIs, databases, collections, or security; an API facilitates the reuse of functionality, protocols, tools, and designs related to the addressed domain. The use of APIs not only facilitates program quality by relying on reusable, tested, established, and well-encapsulated functionality—it also improves program comprehension by being able to find traces of a well-defined and commonly known interface across many projects.

In practice, developers face the following question time and again: Should some concern be implemented 'from scratch' or rather with the help of an existing API and, if so, what API options do exist and how to use the API eventually? In addition to this issue of API adoption, developers also face related issues of API migration or retirement. In all these contexts, developers rely on fundamental techniques for searching APIs [32, 42, 46], classification of APIs [22, 36, 45, 46], API documentation [21, 25], and mining API usage including patterns thereof [12, 23, 30, 40, 46].

In the present paper, we focus on the classification problem and we study many alternatives of API clustering and discuss the correlation between computed clusters and available baselines for classification. We also examine the influence of different parameters of API clustering on the correlation. The simpler of our two baselines is a suite of 60 APIs with manual tags for (programming) 'domains' available in previous work [36]. We develop a more significant baseline (Section 3) as a curated suite of APIs drawn from the *Maven Central Repository* (*MCR*). This repository features two kinds of crowd-sourced (community-authored) classifiers: more specific 'categories' (e.g., XML processing) and more general 'tags' (e.g., XML). Figure 1 illustrates browsing one of *MCR*'s categories.

Our effort on API clustering is based on a feature model (Section 4) which captures the parameters of a versatile hierarchical clustering approach. Our approach does not only allow us to find a configuration with the maximum correlation to a given baseline (Section 5), we can also explore the parameters (Section 6) and the corresponding clustering (i.e., the dendrograms of hierarchical clustering) interactively, subject to a web application. Figure 2 visualizes API clusters for the XML APIs of the smaller baseline. One can observe that the XML domain is detected and reproduced by this clustering because all present XML APIs are combined eventually and non-XML APIs only appear further up in the merging process not shown in Figure 2. It is interesting to notice that SAX and JDOM show stronger similarity when compared to JDOM and dom4j, even though this grouping does not correlate with the underlying, more refined classification of XML parsing approaches (push, pull, or in-memory).

In this manner, our approach can also be used to validate existing classifiers, e.g., those available on *MCR*, because we can answer the

Figure 1: Browsing *MCR*'s category 'XML Processing'.

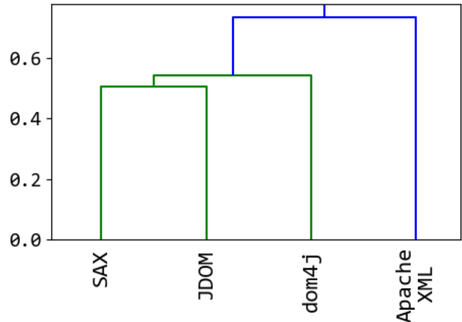

Figure 2: Exploring clustering of XML APIs. The vertical axis shows the threshold for similarity between merged APIs.

questions whether the right classifiers are applied to all APIs and how to usefully recommend classifiers; see Section 6 for details. Our approach to API clustering and the visual exploration in particular has been inspired by related work on clustering metamodels [4].

Summary of contributions.

- We develop a threshold-based approach for creating a curated suite of relevant APIs and versions thereof by taking into account popularity (usage) on *GitHub* and API versioning. To this end, we gather *GitHub* projects dependencies on APIs in the *Maven Central Repository* (*MCR*) from 2.5 million POM files on *GitHub*. We also analyze the API changes between consecutive versions. *MCR*'s metadata for categories and tags (for classification) is added to the curated API suite.
- We develop a feature model for the variability in API clustering and a method for evaluating configurations by means of measuring the correlation of clustering with a given baseline for classification. In this manner, we can determine the configuration with maximum correlation, subject to the enumeration of all configurations generated by the feature model. Our model admits 2592 configurations.
- We propose an interactive exploration mechanism for API clustering, subject to appropriate visualization. In this manner, we can also identify issues with the use of classifiers (e.g., missing classifiers) in the baselines and limitations of the clustering approach.

The curated API suite, the feature model, the correlation results, other elements of an overall dataset, and the web application for exploring API clustering are available online.[1]

Roadmap of the paper. Section 2 recaps hierarchical clustering. Section 3 summarizes the methodology for creating the curated API suite as a baseline. Section 4 introduces the feature model for variability in API clustering. Section 5 describes the evaluation of API clustering for all feature model configurations. Section 6 demonstrates interactive exploration of API clustering. Section 7 discusses threats to validity. Section 8 discusses related work. Section 9 concludes the paper.

2 HIERARCHICAL CLUSTERING

In this paper, we exercise hierarchical clustering of API versions in two instances: first for reducing the number of versions to be considered and second for computing manifestations of API classifiers.

Generally, hierarchical clustering [2, 5, 15] takes a set of elements as source and constructs a hierarchy of clusters. The hierarchy is represented as a dendrogram, i.e., essentially a binary classification tree; see Figure 2 for an illustration. Each node in the tree corresponds to a subset of the source elements; the leaf nodes represent singleton element sets of the source; a non-leaf node merges the element sets of its two child nodes; the root represents the complete set of elements. In Figure 2, the leaf nodes are the APIs such as {SAX}, {JDOM}, {dom4j}, and {ApacheXML}. The vertical position of a node in a dendrogram represents the similarity between the two merged clusters. In Figure 2, SAX and JDOM are the most similar APIs.

Hierarchical clustering may work in to ways: i) in a top-down manner where the root node of the tree is repeatedly divided according to a pluggable second clustering algorithm; ii) in a bottom-up manner starting from the leaf nodes and repeatedly merging nodes by employing a similarity function between the clusters to decide what to merge next. We leverage bottom-up clustering, as summarized below.

Data: elements E, similarities S
Result: dendrogram D
Initialize Cs with singleton clusters for elements E;
while $|Cs| > 1$ **do**
  Merge the most similar clusters in Cs according to S;
  Insert the merge into the dendrogram D;
end

Algorithm 1: Bottom-up hierarchical clustering.

The similarity computations are explained once we discuss the instances of hierarchical clustering in more detail. An important operation on dendrograms is to flatten a dendrogram's hierarchy into a set of mutually exclusive clusters. This is done by cutting the

[1] https://github.com/softlang/apiclustering

Table 1: Top 15 used API versions

GroupId	ArtifactId	Version	#Usages
junit	junit	4.12	156189
junit	junit	3.8.1	130676
junit	junit	4.11	113152
javax.servlet	servlet-api	2.5	90144
javax.servlet	jstl	1.2	90048
log4j	log4j	1.2.17	80550
javax.servlet	javax.servlet-api	3.1.0	70727
commons-io	commons-io	2.4	53062
jstl	jstl	1.2	49412
javax.servlet.jsp	jsp-api	2.1	43652
commons-dbcp	commons-dbcp	1.4	41093
javax.inject	javax.inject	1	39440
junit	junit	4.10	39388
javax.servlet	javax.servlet-api	3.0.1	39388
commons-lang	commons-lang	2.6	36020

tree horizontally at a given similarity threshold. In Figure 2, when cutting at 0.6, the resulting flat clusters are {SAX, JDOM, dom4j} and {ApacheXML}; we get two flat clusters here because we 'cut' two vertical lines.

3 A CURATED API SUITE

We aim at a baseline for API clustering which includes currently relevant APIs with classifiers. To this end, we use a curation approach which selects popular (Java) API versions in the sense of being recently and often used in open-source projects hosted on *GitHub*. Also, we use the crowd-sourced API classification for so-called categories and tags, as available on *Maven Central Repository* (*MCR*). In this manner, we expect to improve on previous baselines in published research, such as the 60 APIs used in work on API-usage analysis [36] which were manually tagged by the authors. Improvement concerns numbers of APIs, currentness of APIs considered, less subjective classification data, and consideration of API versions. The availability of classification data is essential to our evaluation approach for API clustering, as developed in Section 5.

3.1 A Raw API List

MCR suggests coordinates *groupId*, *artifactId* and, *version* for referring to particular versions of an API. The API JARs and metadata including categories and tags for classification can be retrieved from *MCR*.[2] The curated suite should only contain APIs with reasonable degree of importance. The curated suite should not be swamped with redundant content, i.e., nearly equal versions of the same API. To this end, we have developed a threshold-based filtering approach for identifying valuable candidates based on data available on *GitHub* and *MCR*, as described below.

In a first phase, the *GitHub* search API is used to extract the location of all recently indexed pom.xml files on *GitHub*. (For what it matters, the API search limit is circumvented by recursive query segmentation which splits a query by setting an upper and lower bound in file size based on the initially returned number of total results. This process may miss some results.) We only consider heads of default branches. (Due to limitations in the search API, we only consider files smaller than 384 KB.) The resulting 2.5 million POM URLs are streamed into an XML parser for dependency extraction. (For simplicity, only simple placeholders are extracted; parent POM files and version ranges are not considered.) In this manner, we determine over 10 million API usages. The most used coordinate triples are shown in Table 1.

3.2 Analysis of API Elements

In a second phase, the version-specific API JARs, as downloaded from *MCR*, are analyzed in terms of content shared by different API versions or added and removed in each revision. As the basic unit for such analysis, we select method names qualified by class names. We call such a unit '*API element*'. For instance, JUnit version 4.11 contains an API element `Assert#assertEquals`, i.e., the `assertEquals` in the `Assert` class. We decided against package-qualified names so that, for example, package reorganization does not affect similarity. We accept the clashes due to missing package qualification. We stick to this simple representation of source code, as opposed to sophisticated representations, for example, [34].

Accordingly, we consider sets of API elements Es_{n-1} and Es_n to compute the measures 'ADD' and 'REMOVE' for consecutive API versions. We use the Jaccard coefficient to combine added and removed elements. The evolution of JUnit's ADD and REMOVE measure is shown at the bottom of Figure 3.

$$\mathrm{ADD}(n) = \frac{Es_n \setminus Es_{n-1}}{Es_{n-1} \cup Es_n} \quad \mathrm{REMOVE}(n) = \frac{Es_{n-1} \setminus Es_n}{Es_{n-1} \cup Es_n}$$

$$\mathrm{JC}(n) = \frac{Es_{n-1} \cap Es_n}{Es_{n-1} \cup Es_n}$$

3.3 The Curation Strategy

We submit the following curation principles to be met ultimately by a curation strategy which is meant to select API versions for inclusion into the curated API suite and for which to evaluate API clustering:

(a) An API version is of interest, if the overall API is used frequently.
(b) Frequently used versions are more interesting than infrequently used ones.
(c) Versions with significant differences ('ADD', 'REMOVE') are more interesting than nearly identical ones.
(d) Consecutive versions that are very similar do not have to be considered both.
(e) Overall, the size of the suite should be controllable so that a manageable suite can be considered for evaluation.

We considered three strategies. In a first attempt, we ranked the API versions according to *usage* combined with the version's ADD and REMOVE measure. This approach turned out to be too local in that it selected interesting breakpoints in the version history, but it neglected the overall context of available versions; see principle (c). In a second attempt, we used top-down hierarchical clustering to recursively divide the version history at significant points and thereby addressing principle (c). However, we struggled to

[2] *MCR* access URL: http://mvnrepository.com/artifact/*groupId*/*artifactId*

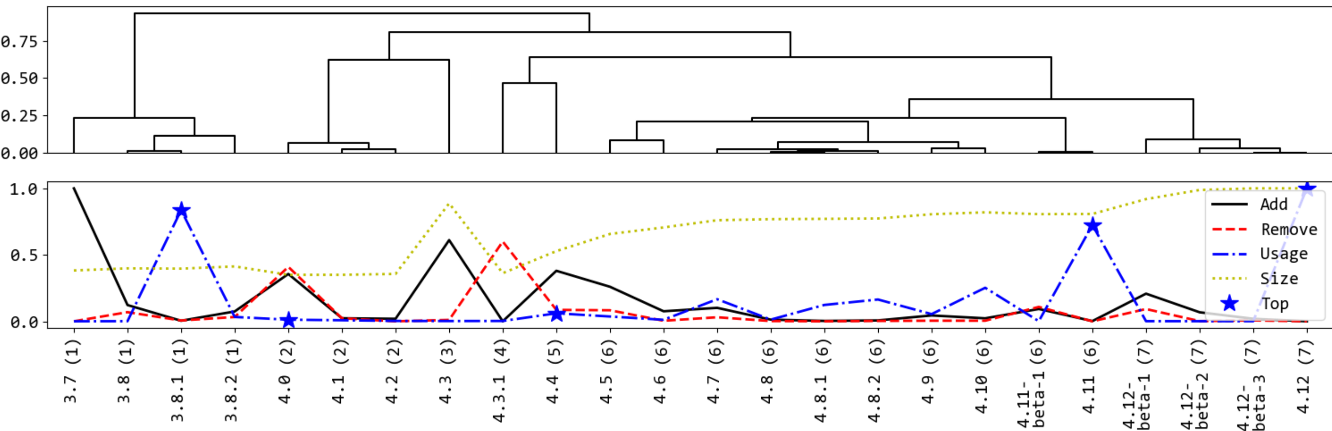

Figure 3: The evolution of JUnit: The upper part depicts the dendrogram merging versions based on similarity. The lower part shows the version history of JUnit in terms of a version's relative size, the usage, and the measures ADD and REMOVE. A star highlights a version that is part of the curated suite. The number behind a version, e.g., *(1)* in *3.8.1 (1)*, denotes a version's cluster index. The threshold for flattening is 0.3, i.e., 30% of the API elements differ.

Table 2: Top 15 highest rated API versions in the curated API suite ('Cls.' abbreviates cluster, 'usg.' usage and 'Vers.' version)

GroupId, ArtifactId & Version	#Es	#API usg.	API rank	#Vers. usg.	Cls. rank	Category	Tags	Vers. rank	#Cls.
junit:junit:4.11	1115	552625	0	113152	0	Testing Frameworks	junit testing	0	7
mysql:mysql-connector-java:5.1.38	5969	183490	1	16039	0	MySQL Drivers	database connector driver mysql	0	6
org.springframework:spring-webmvc:4.2.5.RELEASE	3385	175226	2	7520	0	Web Frameworks	spring framework web mvc	0	6
org.slf4j:slf4j-api:1.7.5	158	167060	3	27329	0	Logging Frameworks	slf4j logging api	0	4
org.springframework:spring-context:.1.6.RELEASE	3500	164921	4	6978	0	Dependency Injection	spring dependency-injection	0	7
log4j:log4j:1.2.17	1990	157603	5	80550	0	Logging Frameworks	logging	0	2
org.springframework:spring-core:4.2.5.RELEASE	4392	130559	6	5897	0	Core Utilities	spring	0	8
org.slf4j:slf4j-log4j12:1.7.5	36	122700	7	16546	0	Logging Bridges	slf4j logging bridge	0	5
org.springframework:spring-web:4.2.5.RELEASE	3194	119705	8	5496	0	Web Frameworks	spring framework web	0	9
org.springframework:spring-test:4.1.6.RELEASE	2191	112174	10	4657	0	Testing Frameworks	spring testing	0	5
javax.servlet:servlet-api:2.5	370	108760	11	90144	0	Java Specifications	standard specs javax servlet api	0	3
commons-io:commons-io:2.4	768	103973	12	53062	0	I/O Utilities	io	0	6
org.springframework:spring-jdbc:4.2.5.RELEASE	1675	102520	13	4694	0	JDBC Extensions	spring jdbc sql	0	3
javax.servlet:jstl:1.2	1704	97483	14	90048	0	Java Specifications	standard specs javax servlet	0	2
com.fasterxml.jackson.core:jackson-databind:2.6.3	5213	95269	15	4929	0	JSON Libraries	binding json	0	2

reasonably define the point at which to split and we actually discovered principle (d) which was not addressed so far. In the third and ultimately preferred attempt, we used bottom-up hierarchical clustering, thereby immediately addressing principle (d). Clustering is applied to the sorted version list of each API. Consecutive versions are merged based while using the lowest Jaccard coefficient (JC) for the similarity function. We slightly deviate from the basic algorithm in that linear (temporal) order of the elements is respected because we want clustering to align with the version history showing ADD and REMOVE, thereby enabling an intuitive view of the evolution of an API in combination with the clustered version ranges. For instance, Figure 3 depicts the similarity of JUnit versions in the upper part with the leaves at the bottom corresponding to JUnit's version history.

In our approach, a dendrogram is flattened at a given threshold to separate clusters of fundamentally different versions. Three components are summed up to produces a final version rating. The formula that we utilize reflects a very basic ranking schema. We aim at enumerating the versions based on the usage rank of an API ('API rank'), the usage rank of a cluster ('Cluster rank'), and the usage rank of a version ('Version rank'). Enumeration is based on a combined rank defined as the sum of the following components:

- **API rank** represents an API's global usage rank over the raw list. Thereby, we address principle (a).
- **Cluster rank $*$ #APIs** represents the usage rank of a cluster among all the clusters for the given API multiplied with the number of available APIs. Thereby we ensure that each API is exercised once before any second cluster for an API is selected. Thereby, we address principle (b).
- **Version rank $*$ #Clusters** represents the version's usage rank in the given cluster multiplied with the overall number of clusters. Thereby we ensure that each cluster is exercised once before any second version is drawn from a cluster. Thereby, we address principles (c) and (d).

For the sake of manageability (principle (e)) and to remain with this simple formula, we only consider the top 100 used APIs (Section 3.1). We consider all the versions of these 100 APIs for clustering. We apply a flattening threshold 0.3 for the clusters, i.e., 30% of the API elements differ. The versions are sorted by the summed-up components, as described above. We cut at an index of 300. The highest rated versions of the curated API suite are shown in Table 2.

4 VARIABILITY IN API CLUSTERING

We aim at a feature model for variability in API clustering such that it can be searched for configurations with particular cluster properties, most notably correlation with the classification of a given baseline. The model at hand does not capture all conceivable ways in which API clustering might work, but it combines a considerable range of options exercised in related work. The mandatory top-level features are shown in Figure 4.

We restrict the clustering to bottom-up hierarchical clustering. Representative-based clustering methods (e.g., k-means) or yet alternative methods are not evaluated in this work.

4.1 Data Representation

Since the representation of API data constrains nearly every processing components, we decided to restrict our exploration to one prominent format, i.e., a vector as part of a vector-space model [39]. We prefer this basic, non-dependency-oriented, quantitative, multi-dimensional representation format for APIs, since it is compatible with a variety of standard analytical processing steps that follow. Processing features that we miss are dependency-oriented formats, most notably, strings or trees.

4.2 Data Extraction

The initial step of extracting data[3] bridges the gap between API (implementation) and vector. A standard way, when working with source code, is to extract islands of natural language. The resulting text is converted to vectors of plain natural language terms where each term is a vector component with the number of its occurrences in the text. There are different ways how abstract syntax trees can be mapped to vectors. We only extract declared class and method names from APIs as islands of natural language; we do not explore the alternatives of extracting data from the body of a method or comments. Using identifiers for classification is successfully applied in previous work [18, 19, 43]

4.3 Alternative Features

In the following, we describe the variability for the top-level mandatory features in Figure 4. The order of presentation reflects the order of processing steps. The sequential execution of the processing steps could be interrupted, in principle, by several feedback mechanisms. We stick to the linear process instead of exploring more complex orchestrations.

Source. Two alternative sources (baselines) may be considered. This can either be the 60-APIs baseline (feature *Sixty APIs*), as extracted from the data set of previous work [36] or the curated API suite (feature *Curated API Suite*), developed in Section 3.

[3] One also speaks of 'feature extraction', but we use 'data extraction' in this paper to avoid confusion with features in the sense of variability (feature modeling).

Table 3: Sequentially applied normalization steps

Normalization steps in sequential order	Sample	*None*	*CCSW*	*All*
Chars that are not letters are removed from the extracted names.	assertEquals	×	×	×
Camel-case occurrences in the text are split by inserting whitespace between the separate words.	assert Equals		×	×
Every char in the text is made lower-case.	assert equals	×	×	×
Words from a stop-word list are removed from the text.	assert equals		×	×
The porter stemming algorithm [27] is applied to all words.	assert equal			×

Sampling. *Sampling* takes the API list according to the *Source* feature and samples its content. Feature *None* does not alter the list, feature *API* (sampling) removes half of the available APIs, and feature *Class* (sampling) removes half of the available classes for each API.

Data Selection. Data selection for clustering can be done by *filter* and *wrapper models* [1]; they both optimize the selected subset of content that is used for clustering. A filter model is based on a criterion to select content to feed into the similarity function used for clustering. A wrapper model applies clustering on a subset of the content; afterwards, the result is used to evaluate the selection [1]. Wrapper models are not applicable since we do not yet cover feedback mechanisms in our approach. We use a domain-specific filter that accepts method and class names based on visibility: Feature *All* extracts all available names whereas *Visible* only keeps those that belong to classes and methods that can be seen from outside the API packages, as this may correspond better to API usage.

Normalization. Normalization is applied to the extracted islands of natural language; in our case, to method and class names. The output is the source for the construction of vectors in the vector-space model. Normalization reflects domain-specific knowledge and conventions. When processing software artifacts, camel-case can be split. When processing natural language artifacts, stemming can be applied. We decided on the sequentially applied normalization steps (inspired by [3]) as shown in Table 3. Each of the alternative features that we explore in the feature model bypasses a certain selection of such steps, thereby slightly decreasing the relevant combinations. These are the options for processing chains for normalization: *None* for doing the minimum of necessary normalization; *CCSW* for including programming-related conventions and stop-word filtering; *All* for including also a natural language processing step. We used the Mallet stop-word list.[4]

Document Granularity. We extract text (class and method names) from the class files (JARs) of an API, selected by visibility,

[4] http://mallet.cs.umass.edu/

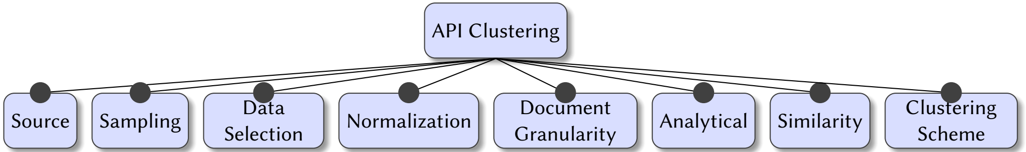

Figure 4: Top-level features of API clustering.

and normalized by different strategies. The *Document Granularity* defines how the text of the API classes are aggregated to form the vectors (*document unit*). We consider these granularity levels: *API* (granularity) aggregates all text of an API, *Package* (granularity) groups by package, and *Class* (granularity) takes class files as document unit.

Analytical. The *Analytical* feature exposes dimensionality reduction and weighting techniques side-by-side. We integrated dimensionality reduction and weighting techniques in the same position of the processing chain, as both are altering the vectors in dimensionality and/or changing the components. For LSI we combine the weighting scheme and dimensionality reduction comparable to [20]. Feature *None* bypasses the analytical processing; *IDF* performs weighting based on *inverse document frequency* to decrease the influence of very common terms; *LSI* applies latent semantic analysis [10] on the vectors (previously weighted analogue to feature *IDF*); and *LDA* performs latent dirichlet allocation [7].

Latent semantic indexing (LSI) is the application of a singular value decomposition on the textual domain; it projects the original data into a smaller axis system. Afterwards, the truncated representation of the original data is often better suited for similarity computations, since the noise is reduced in the documents, and phenomena like synonymy or polysemy of features are resolved [1]. We set the dimension parameter to 200. Latent dirichlet allocation (LDA) is mostly considered to be a topic-modeling technique rather than a technique for reducing dimensionality. While LSI uses a reduced amount of linear independent dimensions to represent documents in a compressed way, LDA employs a stochastic sampling process producing a set of topics that can afterwards be used in a similar manner to represent and compare documents. Our LDA instance is configured with 25 topics; beta and alpha are both assigned to 0.1; it is configured with a random seed and memorized once computed; and we use an EM algorithm implementation of Scala Apache Spark[5]. LSI is taken from the same library.

By setting the parameters for LSI and LDA to specific values, we exclude alternative topic counts, dirichlet distributions, or dimension counts, thereby suggesting directions for future work. Also, we do not explore several known effects of weighting techniques on topic models or dimensionality reduction (e.g., [20, 24, 44]).

Similarity. A similarity function is the essential input for hierarchical clustering. We consider two definitions that are applicable for similarity computation between API vectors. If document granularity is not chosen to be *API*, i.e., the document vectors do not directly correspond to an API, then the class- or package-sized documents for each API are aggregated by summing up the vectors to form the overall API vector. Prominent alternatives for similarity are *Cosine* and *Jaccard coefficient*. There are several other measures that we do not explore since our data might be sparse (no dimensionality reduction is applied); it might contain negative components (when processed by LSI); or the document vector can represent a stochastic distribution (when processed by LDA). We do not explore several alternatives for similarity, for example, the lp-norm or Kullback Leibler divergence.

Clustering Scheme. Bottom-up hierarchical clustering works on a set of clusters and keeps on merging (pairs of) clusters until only one cluster remains. We permit different linkage schemes needed to lift the similarity between two APIs to the level of two sets of APIs. There are the following options. The *Single* scheme merges those clusters first with the minimum distance in between any of the APIs contained in the two set, no matter how far the clusters' centers are apart from each other. The *Complete* scheme leverages the maximal distance in between the two sets. The *Average* scheme leverages the mean similarity.

5 EVALUATION OF API CLUSTERING

The general idea behind the feature model evaluation is that every configuration has a fixed correlation with an external criterion. In this manner, configurations (correlations) can be compared. Ideally, API versions that are classified the same by a given baseline would be grouped in the same cluster. Also, all versions of a API should be grouped together by clustering. First, we describe how correlation between clusters and classification can be measured. Afterwards, we present the concrete evaluation in terms of i) correlation curves, ii) maximum correlation, iii) feature dependencies, iv) generalization and v) top configurations.

5.1 Methodology

The enumeration of configurations is straightforward because the feature model is flat with all top-level features being mandatory and directly refined into leaf alternatives. There are 2592 configurations based on combining these options: *Source* (2), *Sampling* (3), *Feature Selection* (2), *Document Granularity* (3), *Normalization* (3), *Analytical* (4), *Similarity* (2) and *Clustering Scheme* (3).

We use an *external criterion* [13] for cluster evaluation, that is, a pre-existing classification. Our methodology for computing the correlation between clusters and classification is inspired by related work on clustering metamodels [4]. We construct matrices M_x for several relationships x (0 and 1 in the cells) between the clustered API versions as the rows and columns.

$$M_x(i,j) = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are related according to } x \\ 0 & \text{otherwise} \end{cases}$$

[5] https://spark.apache.org/mllib/

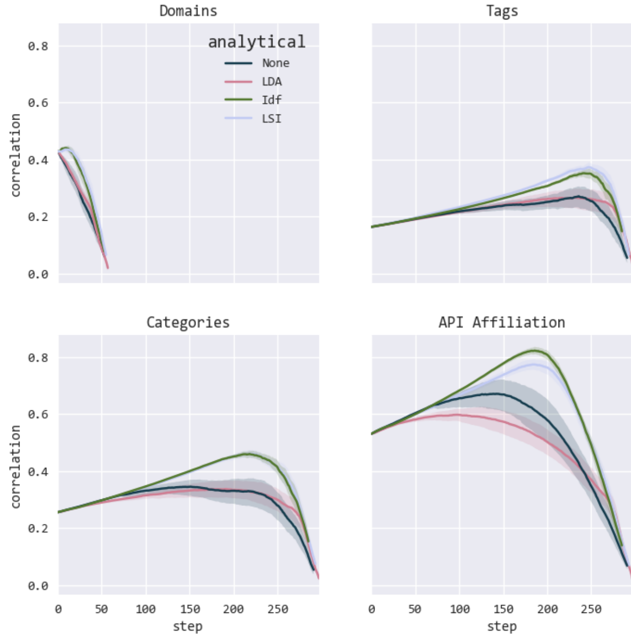

Figure 5: Correlation curves for the analytical models.

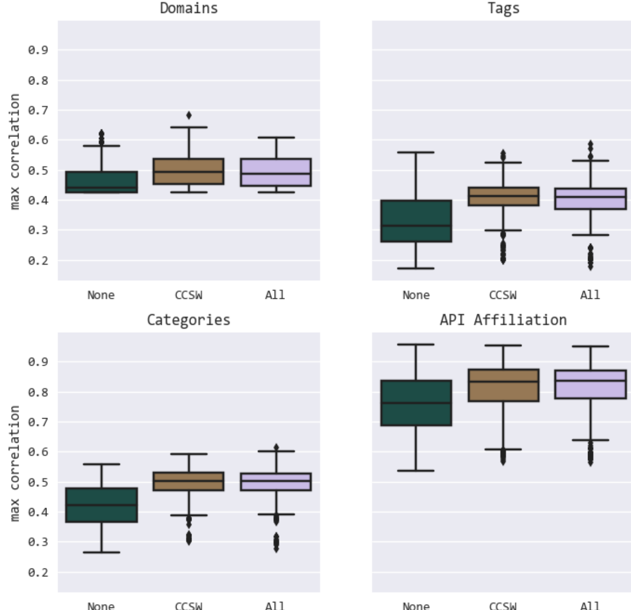

Figure 6: Influence of normalization on correlation.

There are these options for x: cls — for belonging to the same cluster; tag — for sharing a tag (according to *MCR*); cat — for being in the same category (according to *MCR*); dom — for being in the same domain (according to Sixty APIs); api — for being versions of the same API which we also refer to as 'API affiliation'. The matrices tag, cat, dom, and api correspond to external criteria.

We describe now the computation done for each configuration. This computation employs the created dendrogram. We apply the MATLAB corr2 function[6] to compare M_{cls} and any external criterion matrix; the function returns a correlation measure between -1 for negative correlation, 0 for no correlation, and +1 for positive correlation. We do not want to select a threshold for cutting the dendrogram, as it is prone to high variations in the underlying similarity. Thus, we flatten the dendrogram at each merging step and compute M_{cls} such that each matrix reflects one additional cluster merge.

5.2 Correlation Curves

The sequence of corr2 applications between the sequence of M_{cls} matrices and the external criterion can be depicted as a curve showing the evolution of correlation over all merging steps with the criterion at hand. For instance, Figure 5 illustrates main trends of the stepwise correlation grouped by the *Analytical* feature with a notion of uncertainty, depicted by the shaded area around the main curves. There are four curves in such a figure: one for each external criterion.

The plot in Figure 5 shows that *IDF* performs best at API affiliation (i.e., grouping of API versions) at nearly every merging step; *LSI* performs best in reproducing tags; *LDA* and no analytical precessing (*None*) tend to misclassify APIs on average but both also share the highest amount of uncertainty; *IDF* and *LSI* both perform well on the domain criterion. API affiliation can only be evaluated for *Curated API Suite* because the corpus for *Sixty APIs* contains only one version per API. A side effect of different versions is that the tag and category curves rise during the first 200 steps simply because versions of the same API are assigned to the same classification and are very likely to be merged. The domain criterion can only be tracked on the *Sixty APIs* dataset with 60 APIs and hence, only 60 merging steps are shown in this case.

This type of plot is also available online for the other top-level features of our feature model. In this manner, we cover all 2592 configurations at some level of abstraction. That is, we group the 2592 configurations by features and plot the set of correlation curves for each group as done in Figure 5.

5.3 Maximum Correlation

Feature-specific influences are illustrated by simplifying the plots on depicting the distribution of maximum correlations, i.e., the peak of the correlation curves. Figure 6 shows such distributions for the alternatives of the *Normalization* feature. Introducing stemming (*All*) does not increase performance when compared to camel-case splitting and stop-word filtering (*CCSW*). Tags, category, and domain classification are negatively effected by missing normalization (*None*); API affiliation (grouping of API versions) is affected less. Such comparison of one particular feature is reasonable because each alternative's distribution reflects all other features equally measured, e.g., each distribution of *None*, *CCSW*, and *All* bears the same amount of configurations employing the feature *LDA*.

5.4 Feature Dependencies

We may also examine the influence of features on each other. Dependency between features is illustrated by the distribution of

[6] http://de.mathworks.com/help/images/ref/corr2.html

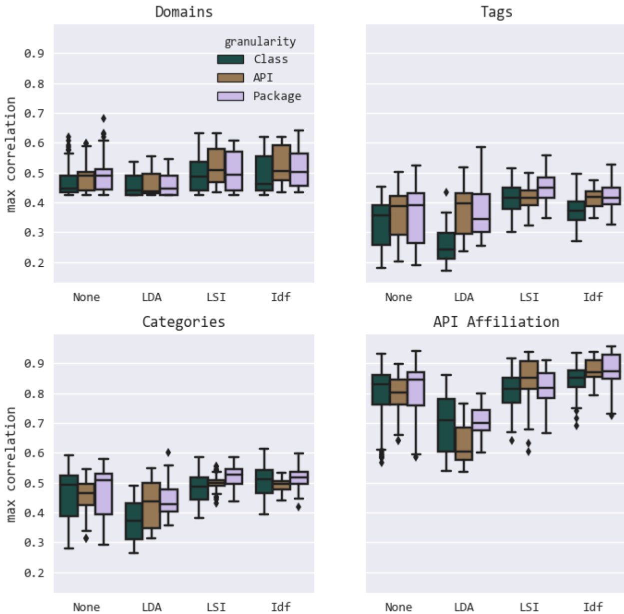

Figure 7: Influence of document granularity and the analytical model on correlation.

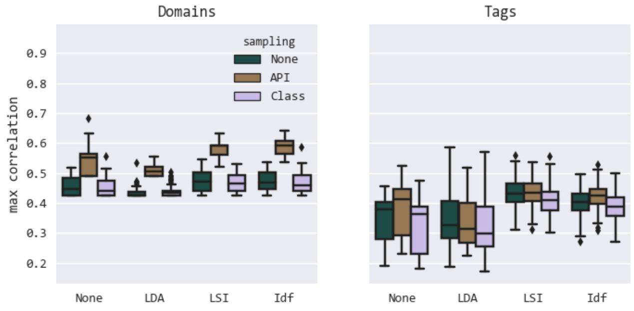

Figure 8: Influence of sampling and the analytical model on correlation.

maximum correlation when grouping all configurations for two features *a* and *b*. The distribution is depicted on the vertical axis, the alternatives of *feature a* are listed on the horizontal axis, and *feature b* defines the colors of the midspread in the boxplot. For the features *Document Granularity* and *Analytical* this is shown in Figure 7. The plot shows that, except for criterion API affiliation, *LDA* is much better when working with the coarse-grained document sizes *Package* and *API*. This is comparable to insights of other research, e.g., in the context of analyzing twitter documents [28]. All other analytical models are affected less by document granularity.

5.5 Generalizability

We may leverage *Sampling* strategies for checking the generalizability of statements in our work. Figure 8 shows a grouping by *Sampling* and *Analytical*. The analytical method can be interpreted as the treatment and different sampling strategies corresponds to the subject of a treatment. The maximum correlation with respect to an external criterion is the measure that we want to observe; hence, we examine the distributions. Currently, our sampling strategy is not strong enough to make claims about significance. A drawback of the corr2 function is that the diagonal of both compared matrices holds ones which happens to increase the ratio of perfectly correlating diagonal entries when matrix dimensionality decreases. Therefore, the correlation cannot be reliably compared for configurations on different sized API lists.

5.6 Top Configurations

We show the top combinations for the Curated API Suite in Table 4. No sampling is applied. The list is constructed such that it contains each available feature in its highest ranked position. The top-most occurrence of each concrete feature is highlighted in bold. We decide to rank according to the maximum category correlation. *LDA* produces the best clusters. The second configuration on the ranking is one that uses the analytical model *IDF* and is the best configuration when having *Class* document granularity fixed. The third configuration employs *LSI* it is the first configuration which benefits from bypassing stemming (feature *CCSW*). The *Jaccard coefficient* performs best when selecting all methods of an API instead of the visible ones (feature *All*). The two highest ranked configurations in Table 4 are explored in the following section.

6 EXPLORATION OF API CLUSTERING

An exploration of the clustering results, subject to setting parameters of the configuration, may help developers to validate existing classifiers in terms of answering the questions whether the right classifiers are applied for all APIs and how to usefully recommend classifiers. Of course, limitations of the clustering approach may also be revealed in this manner. That is, an overall maximum correlation may not necessarily imply that the corresponding configuration works best for clustering (categorizing) a specific subset of APIs. We demonstrate these aspects below.

In our implementation, we provide a panel (Figure 9) permitting the user to select an alternative for each top-level API clustering feature. Such a multi-parametric way of exploring clustering is inspired by previous work on clustering metamodels [4].

Figure 10 shows a dendrogram for merging the MySQL and PostgreSQL APIs. This dendrogram is computed for the top-most configuration in Table 4. This *LDA*-based clustering does not separate the different APIs; it separates older and newer versions, thereby illustrating a limitation of our approach.

Figure 11 shows another dendrogram for merging the MySQL and PostgreSQL APIs. This dendrogram is computed for the second ranked configuration in Table 4. This *IDF*-based configuration is more appropriate for grouping versions of the same API together. If we are interested in such grouping, we should consider configurations with a higher correlation with API affiliation. This is indeed the case for the shown configuration.

Figure 12 points out a missing classification in *MCR*. In the overall view, this part of the dendrogram is located above the previously shown MySQL and PostgreSQL clusters in Figure 10. Clearly,

Table 4: Top configurations with maximum category correlation

Analytical	Similarity	Normal.	Data Selection	Document Granularity	Clustering	Max Category	Max Tags	Max Api
LDA	**Cosine**	**All**	**Visible**	**Package**	**Single**	0.602	0.587	0.671
Idf	Cosine	All	Visible	**Class**	Single	0.599	0.439	0.844
LSI	Cosine	**CCSW**	Visible	Package	Single	0.579	0.501	0.79
Idf	**JC**	All	**All**	Class	Single	0.571	0.387	0.794
Idf	JC	All	Visible	Package	**Average**	0.568	0.467	0.871
None	Cosine	All	Visible	Class	Single	0.566	0.414	0.852
LSI	Cosine	**None**	All	Package	Single	0.554	0.559	0.725
LSI	Cosine	CCSW	All	Package	**Complete**	0.551	0.462	0.834
LSI	Cosine	CCSW	Visible	**API**	Single	0.539	0.471	0.845

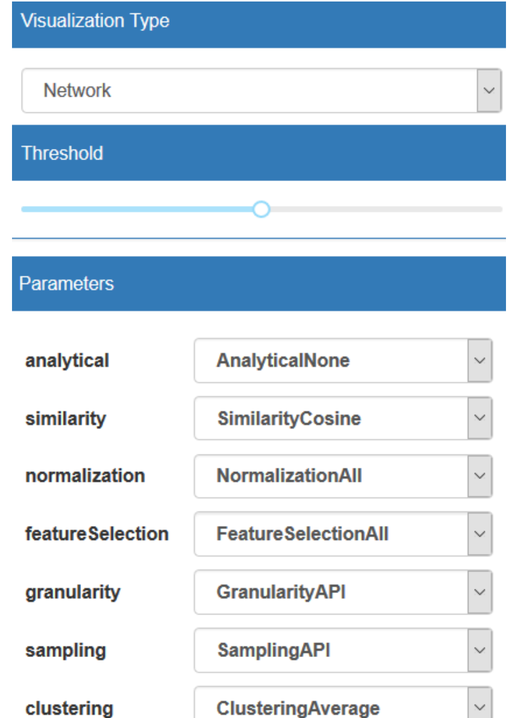

Figure 9: Parameter panel for exploration.

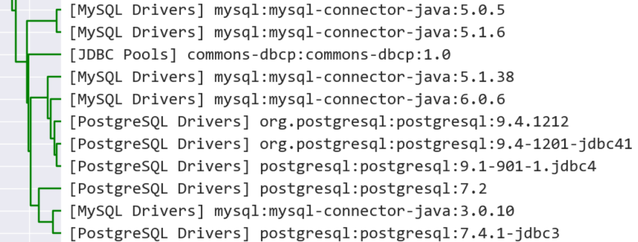

Figure 10: MySQL and PostgreSQL clustering.

'com.alibaba:druid' seems to provide SQL functionality, as we confirmed manually, but such a classification is missing, thereby illustrating the potential of our approach to help with correcting classifications.

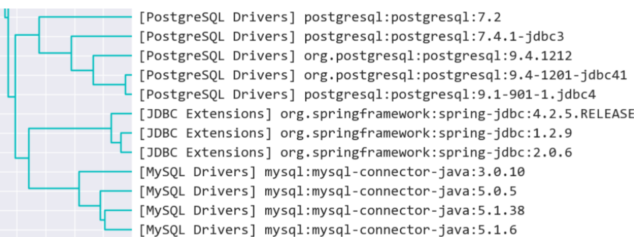

Figure 11: An alternative MySQL and PostgreSQL clustering.

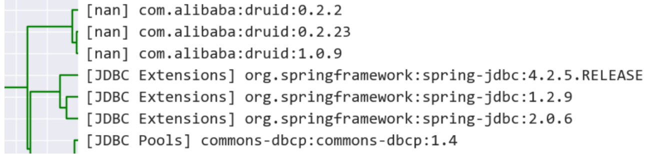

Figure 12: Missing classification on Maven Repository.

7 THREATS TO VALIDITY

Curated list. An issue on internal validity is that only the most recent POM files are taken into account regardless of any timestamp. We argue here that the most recent state of *GitHub* is our focus indeed. We exclude version ranges from the analysis, as we prefer analyzing 'explicitly' chosen API usage. A last point that we omit to cover are accidentally packaged dependencies in API JARs. This is difficult to handle. For instance, naming conventions of the group and artifact id are not strictly met in the package structure. In the future, we aim to include explicit API dependency information.

Feature model. The main threats to internal validity are related to the restricted and biased feature model as already outlined in Section 4. We keep close to related work to address this threat.

Evaluation. A main issue on external validity, already pointed out in Section 5, is that we do not make significance statements about the generalizability of a configuration. We partially handle external validity applying a configuration on differently sampled data. The outliers produced by LDA are connected to the lack of samples that are input to our approach, as LDA is a stochastic process that produces different results on each run. Especially when computing statistics on LDA averaging the results becomes necessary [41].

This threat can be handled in future work by increasing the amount of samples.

8 RELATED WORK

Categorization and Clustering. In [11], a comparable approach to automated categorization of new software projects or libraries is exercised: BUCS (Bytecode-based Unsupervised Categorization of Software). Unsupervised dirichlet process clustering is used. The authors motivate the usability with missing source code (only class files are available) and the unsupervised nature of the classification. We align with this research in that we add a well-defined baseline and explore variability not present in BUCS. Such research can be seen as a potential profiteer of insight gained by our evaluation, for example, in that stemming is not crucial for the shape of API clusters. In [45], API clustering in the domain of service oriented-computing applies a k-means variant in combination with LDA. The work exceeds ours in providing and evaluating a ranking model for services that is aware of categories. In [4], metamodel repositories are clustered hierarchically. Four alternative similarity functions are used and compared. We explore more alternatives and combinations. This work exceeds ours in also examining structural similarity between metamodels. In [18, 19, 43], software archives or open-source repositories or source-code archives are automatically categorized. We cluster APIs using hierarchical clustering and by aiming at good correlation with existing baselines. In [20], program code is hierarchically clustered in a more fine-grained manner. Our feature model covers parts of the described methods, i.e., LSI, cosine similarity, and average linkage scheme. In [26], various software clustering algorithms are evaluated in the context of program comprehension. First, many comprehension decompositions of the system are produced that are then used for external evaluation of three linkage schemes of hierarchical clustering, k-means, and ACDC. We consider several more dimensions of alternatives described by a feature model. We conduct our research on APIs.

Parameter Exploration and Optimization. In [31], the methods Jensen-Shannon (JS), VSM, LSI, and LDA are used for traceability recovery and evaluated on the small-scale EasyClinic and eTour corpora. We conduct the evaluation of clustering on the larger, crowd-sourced baseline *MCR*. Multi-dimensionality of alternatives is readily touched in this work in terms of exploring the effects of six different LDA topic counts. In [6], LDA-specific parameters and differences in the normalization are examined when used for source-code analysis. The research focuses on the illustration of parameter influence on internal criteria executed on a synthetic corpus. The authors aim to aid in LDA parameter understanding. We execute a broader analysis of alternatives on a crowd-sourced external criterion. In [29], configuration parameters in software systems are analyzed, as motivated by poor default configurations. The proposed technique uses statistic methods to generate a graph-based plot on parameter interaction and is exemplified for Apache Hadoop configuration. We also depict dependencies between parameters. In [33], LDA parameter optimization relies on genetic algorithms. We may want to enhance our work accordingly to lift our feature model exploration to the level of search-based software engineering [14]. This work uses the combinatorial exploration of configurations for the evaluation of the generic algorithm. In [8], a method called combinatorial interaction testing (CIT) is described that enumerates combinations of parameter assignments to test a system. The amount of tests can be reduced by only enumerating all possible n-way combinations for the parameters. In [9], the generation of combinations is pruned by constraints leading to a cost-effective coverage. We see the introduction and constrains or other means of reducing the number of combinations as a useful extension of our approach. In [16], combinatorial interaction testing is applied to generate coverage arrays for a feature model used to test a industrial scale software product line. This is facilitated by an improved algorithm. We share the notion of feature models.

Searching. In [32], LibFinder is presented; it helps revealing missed reuse opportunities of APIs in a system. The mechanism is driven by used APIs, co-usage of APIs, and the semantic similarity in between the system and an API. This work is relevant in that it also employs API usage data mined from Maven and GitHub. In [42], search functionality that is suited for API classes is illustrated that can be seen as an alternative to category-based browsing that we facilitate.

Usage Patterns. In [40], a dataset of API usage is presented. This work, just like ours, relies on mining usage data with the help of parsing POM files located on *GitHub*. This dataset exceeds our usage data since it considers fine-grained API method calls and version history. In [30], usage examples are automatically mined and visualized. This work is interesting in terms of the varied granularity of the produced 'clusters' of practical API usage. In [37], usage patterns are extracted only considering the API and not the client projects. The work is related in that is also uses a clustering algorithm, i.e., DBSCAN, for detecting usage patterns. In [38], the previous work is extended by combining client-based and library-based usage patterns. In [12], a deep learning-based approach focused on API-usage sequences is illustrated.

9 CONCLUSION

We have developed an automated approach for computing a curated API suite based on *Maven Central Repository* and *GitHub*. We have used the suite as a baseline for exploring clustering configurations. The suite may also serve other purposes in the future, for example, for mining API migration patterns [17], in which case however we should extend the suite by method calls in client projects along the version history, as outlined for a different data set in related work [40].

We plan to extend the feature model (e.g., for discrete parameters) but also to restrict it (e.g., on a particular interest). This involves an improved evaluation method with respect to scalability and constraining the feature model. Two options that we have in mind are using genetic algorithms, as described in [33], or combinatorial interaction testing of the feature model, as described in [8, 9, 16]. We also want to study features that concern similarity computation for artifacts of different types (e.g., code versus documentation versus model versus textbook). We are interested in interpreting the data that emerges during different clustering configurations with respect to the given domain (i.e., documents, topics and terms). Further, we see potential applications of the curation approach (i.e., how versions are selected) in other contexts, as it might help to boil

down the available amount of interesting versions in those contexts, too, for example, in the context of recommendation systems for software engineering [35].

REFERENCES

[1] Charu C. Aggarwal. 2015. *Data Mining - The Textbook.* Springer.

[2] Nicolas Anquetil and Timothy Lethbridge. 1999. Experiments with Clustering as a Software Remodularization Method. In *WCRE.* IEEE, 235–255.

[3] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. 2000. Information Retrieval Models For Recovering Traceability Links between Code and Documentation. In *Proc. of ICSM (2000).* IEEE, 40–49.

[4] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2016. Automated Clustering of Metamodel Repositories. In *CAiSE (LNCS)*, Vol. 9694. Springer, 342–358.

[5] Pavel Berkhin. 2006. A Survey of Clustering Data Mining Techniques. In *Grouping Multidimensional Data.* Springer, 25–71.

[6] David Binkley, Daniel Heinz, Dawn J. Lawrie, and Justin Overfelt. 2014. Understanding LDA in source code analysis. In *ICPC.* ACM, 26–36.

[7] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *Journal of Machine Learning Research* 3 (2003), 993–1022.

[8] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatiorial Design. *IEEE Trans. Software Eng.* 23, 7 (1997), 437–444.

[9] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2008. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Trans. Software Eng.* 34, 5 (2008), 633–650.

[10] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. 1990. Indexing by Latent Semantic Analysis. *JASIS* 41, 6 (1990), 391–407.

[11] Javier Escobar-Avila, Mario Linares Vásquez, and Sonia Haiduc. 2015. Unsupervised software categorization using bytecode. In *ICPC.* IEEE, 229–239.

[12] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *SIGSOFT FSE.* ACM, 631–642.

[13] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. 2001. On Clustering Validation Techniques. *J. Intell. Inf. Syst.* 17, 2-3 (2001), 107–145.

[14] Mark Harman and Bryan F. Jones. 2001. Search-based software engineering. *Information & Software Technology* 43, 14 (2001), 833–839.

[15] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. 1999. Data Clustering: A Review. *ACM Comput. Surv.* 31, 3 (1999), 264–323.

[16] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *SPLC (1).* ACM, 46–55.

[17] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. Logging library migrations: a case study for the apache software foundation projects. In *MSR.* ACM, 154–164.

[18] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. 2003. Automatic Categorization Algorithm for Evolvable Software Archive. In *IWPSE.* IEEE, 195.

[19] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. 2006. MUDABlue: An automatic categorization system for Open Source repositories. *Journal of Systems and Software* 79, 7 (2006), 939–953.

[20] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. 2007. Semantic clustering: Identifying topics in source code. *Information & Software Technology* 49, 3 (2007), 230–243.

[21] Niraj Kumar and Premkumar T. Devanbu. 2016. OntoCat: Automatically categorizing knowledge in API Documentation. *CoRR* abs/1607.07602 (2016).

[22] Ralf Lämmel, Rufus Linke, Ekaterina Pek, and Andrei Varanovich. 2011. A Framework Profile of .NET. In *WCRE.* IEEE, 141–150.

[23] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. 2011. Large-scale, AST-based API-usage analysis of open-source Java projects. In *SAC.* ACM, 1317–1324.

[24] Seonggyu Lee, Jinho Kim, and Sung-Hyon Myaeng. 2015. An extension of topic models for text classification: A term weighting approach. In *BigComp.* IEEE, 217–224.

[25] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Trans. Software Eng.* 39, 9 (2013), 1264–1282.

[26] Anas Mahmoud and Nan Niu. 2013. Evaluating software clustering algorithms in the context of program comprehension. In *ICPC.* IEEE, 162–171.

[27] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval.* Cambridge University Press.

[28] Rishabh Mehrotra, Scott Sanner, Wray L. Buntine, and Lexing Xie. 2013. Improving LDA topic models for microblogs via tweet pooling and automatic labeling. In *SIGIR.* ACM, 889–892.

[29] Chelsea A. Metcalf, Farhaan Fowze, Tuba Yavuz, and José Fortes. 2016. Extracting configuration parameter interactions using static analysis. In *ICPC.* IEEE, 1–4.

[30] Evan Moritz, Mario Linares Vásquez, Denys Poshyvanyk, Mark Grechanik, Collin McMillan, and Malcom Gethers. 2013. ExPort: Detecting and visualizing API usages in large source code repositories. In *ASE.* IEEE, 646–651.

[31] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2010. On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery. In *ICPC.* IEEE, 68–71.

[32] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M. Germán, and Katsuro Inoue. 2017. Search-based software library recommendation using multi-objective optimization. *Information & Software Technology* 83 (2017), 55–75.

[33] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *ICSE*. IEEE, 522–531.

[34] Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2016. A dataset of simplified syntax trees for C#. In *MSR*. ACM, 476–479.

[35] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. 2010. Recommendation Systems for Software Engineering. *IEEE Software* 27, 4 (2010), 80–86.

[36] Coen De Roover, Ralf Lämmel, and Ekaterina Pek. 2013. Multi-dimensional exploration of API usage. In *ICPC*. IEEE Computer Society, 152–161.

[37] Mohamed Aymen Saied, Hani Abdeen, Omar Benomar, and Houari A. Sahraoui. 2015. Could we infer unordered API usage patterns only using the library source code?. In *ICPC*. IEEE, 71–81.

[38] Mohamed Aymen Saied and Houari A. Sahraoui. 2016. A cooperative approach for combining client-based and library-based API usage pattern mining. In *ICPC*. IEEE Computer Society, 1–10.

[39] Gerard Salton and Michael McGill. 1984. *Introduction to Modern Information Retrieval.* McGraw-Hill.

[40] Anand Ashok Sawant and Alberto Bacchelli. 2015. A Dataset for API Usage. In *MSR*. IEEE, 506–509.

[41] Mark Steyvers and Tom Griffiths. 2007. *Probabilistic topic models.* Laurence Erlbaum.

[42] Jeffrey Stylos and Brad A. Myers. 2006. Mica: A Web-Search Tool for Finding API Components and Examples. In *VL/HCC*. IEEE, 195–202.

[43] Secil Ugurel, Robert Krovetz, and C. Lee Giles. 2002. What's the code?: automatic classification of source code archives. In *KDD*. ACM, 639–644.

[44] Andrew T. Wilson and Peter A. Chew. 2010. Term Weighting Schemes for Latent Dirichlet Allocation. In *HLT-NAACL*. The Association for Computational Linguistics, 465–473.

[45] Bofei Xia, Yushun Fan, Wei Tan, Keman Huang, Jia Zhang, and Cheng Wu. 2015. Category-Aware API Clustering and Distributed Recommendation for Automatic Mashup Creation. *IEEE Trans. Services Computing* 8, 5 (2015), 674–687.

[46] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *ECOOP (LNCS)*, Vol. 5653. Springer, 318–343.