Summarising How a Software API is Used Using Machine Learning and Natural Language Processing Techniques

Nikolaos Katirtzis

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2016

Abstract

During the last years, there has been an ever increasing trend in reusing functionality provided by third-party libraries, most of which are accessible through their *Application Programming Interface* (API). However, several studies so far have pointed out the lack of proper documentation of these APIs, which seems to be a great obstacle for using them. With the purpose of facilitating the use of APIs, several systems exploit the information that is now available in software repositories, in order to mine usage examples that show popular usages of the APIs (*API Usage Mining*). This dissertation introduces a novel approach to the problem of API usage mining, in an attempt to mine snippets that are concise, readable, precise, and that cover several usages of the target API. Upon setting the background needed, and providing an overview of the state-of-the-art systems in this field, we define four main features that a system which performs API usage mining should exhibit. On the basis of these features, we then present our approach to the problem, as well as the actual implementation of the system. Our system leverages powerful clustering techniques, such as the *k*-medoids or the HDBSCAN algorithms, in order to cluster similar usage examples, based on their API call sequences. With the aim of presenting the most representative snippet from each cluster, it also considers the structure of the source code, by making use of a tree edit distance metric. Additionally, it exploits the power of a novel summarisation algorithm, that has been implemented as part of this dissertation, in order to produce concise and readable snippets, which it presents to the users, in an order that indicates their popularity. Finally, we conduct appropriate experiments, with the purpose of evaluating five different hypotheses, using different versions of the system. The results are both interesting and promising, with the key hypothesis of the dissertation being confirmed, and showing moreover that our system successfully confronts the problem described in the current dissertation.

Acknowledgements

It is a genuine pleasure to express my deep sense of thanks and gratitude to the people that have supported me throughout the elaboration of this dissertation.

First and foremost, I owe a gratitude to my supervisor, Dr. Charles Sutton, for giving me the opportunity to accomplish this dissertation, as well as for his keen interest on me, and on my previous work on source code mining, since our first meeting. His support and continuous optimism concerning this project have been essential during this work.

I would also like to express my gratitude to the PhD candidate from the Aristotle University of Thessaloniki, Mr Themistoklis Diamantopoulos, for the kind cooperation and support throughout this work. Without his valuable assistance, the completion of this project would seem impossible.

I am also grateful to Dr. Jaroslav Fowkes, and to the PhD candidate Mr. Miltiadis Allamanis, for providing me with the datasets used in this work, as well as for their invaluable suggestions.

I also owe a great debt of gratitude to my parents, as well as to my girlfriend Christina, for their understanding and patience throughout this demanding year.

Last but not least, I would like to thank my friends for their support throughout all these years.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Nikolaos Katirtzis*)

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Third-party libraries are an integral part of many software systems. They could facilitate the implementation of a specific component (code reuse), or even offer an additional functionality, that is only available when using them. Most of these libraries are accessible through their *Application Programming Interface* (API), which may consist of numerous classes and methods. However, it is now clear, based on multiple studies, that APIs lack proper examples and documentation and, in general, sufficient explanation on how to be used [1, 2].

In an attempt to overcome the above mentioned limitations, the developers often exploit general-purpose or specialised search engines (*Code Search Engines* or CSEs), or even *Question-and-Answer* services (Q&A) such as Stack Overflow[1], in order to find possible API usages. A noticeable research is conducted in [3], where Hoffmann et al. clearly identify the API-related queries as the most common queries on the web, for the Java language. The results of their research are illustrated in Figure 1.1[2].

1.1 Hypothesis of the Dissertation

The purpose of the current dissertation is to summarise typical use cases of an API, by leveraging powerful statistical techniques from the emerging fields of *Machine Learning* (ML) and *Natural Language Processing* (NLP). A potentially successful system would collect API usage examples from the web, apply unsupervised learning techniques to cluster similar examples, and show a representative example from each cluster, summarising in this way the target API.

[1]`http://stackoverflow.com/`

[2]This figure, as well as small parts of this chapter, have been presented as part of the *Informatics Research Proposal* (IRP) document.

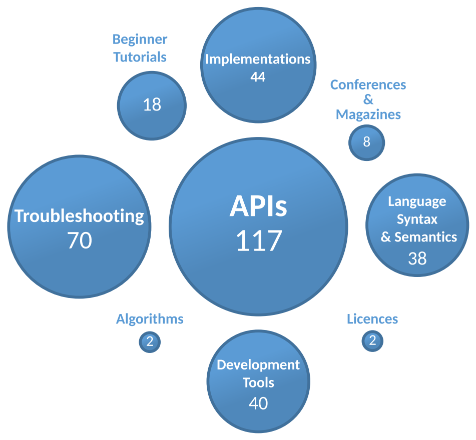


Figure 1.1: Most common types of queries initiated by developers on the web.

Interpreting the purpose of this dissertation, our intuition tells us that an API can be effectively summarised using statistical techniques, that may be successfully applied to source code. Although such techniques have been well applied to natural language, their application to source code is quite challenging, as we have to take into account both its semantic context and its structure. Moreover, a key hypothesis of the dissertation is that the presentation of snippets would be of greater value to the developers, than that of API method call sequences.

1.2 Motivation of the Dissertation

The benefits of such a system are numerous for the users as well as for the developers of an API. In [4], Spinellis identifies *time*[3] and *quality* as two of the three main factors that may lead a software project to success[4]. A system that efficiently summarises an API could improve both of them. More specifically, the presentation of usage examples to the users would lead to time saving, allowing the developers to spend more time in more useful tasks, while it could also help in improving the software's quality by considering popular usage examples of the API. Additionally, the developers of the library might receive useful statistics about the common usages of their library's API, and proceed to enhancements (e.g. enrich it with more examples, in case they notice that the existing ones are not adequate, or even provide a better documentation).

[3]Spinellis implies several aspects of time, however, we are mainly interested in the development time in this dissertation.

[4]With *cost* being the third factor.

Furthermore, we see a possible interesting connection of such a system with a new feature that was added to the popular Q&A website Stack Overflow. The *Documentation*[5] tool, which is still in Beta mode, is described as a “community-curated, example-focused developer documentation”, and aims to collect examples that could be used for documentation purposes in different technologies. We argue that a system that performs API usage mining could act as a driving force for the tool, by bootstrapping the examples. Then, the users of the Documentation could improve them, while adding the appropriate descriptive text.

Finally, there is an additional motivation, which stems from a novel evaluation approach presented in [5]. There, Fowkes and Sutton evaluate the quality of the patterns -in the form of API call sequences- mined by systems that perform API usage mining, using the `examples` directory that possibly exists in the repository of the target library. This approach has not been applied to source code yet, though, and we are interested in doing so in this project.

1.3 Possible Usage Scenarios

We may think of several usage scenarios for the system to be developed. Below we analyse two typical use cases, as well as a brainstorming one.

- The most typical usage scenario involves the expansion of an API’s documentation. For instance, the *APIMiner*[6] tool augments the documentation of the Android API with source code examples.

- Another common scenario is that of a CSE, that is specialised to APIs. As an example, we mention the *Codota*[7] and the KodeBeagle[8] CSEs. Such a CSE could also propose similar examples, or even API methods that are often invoked together.

- A more brainstorming scenario has been described in Section 1.2, and makes use of the new StackOverflow’s *Documentation* feature. That is, the system could be exploited with the purpose of augmenting the Documentation with examples.

[5]`https://stackoverflow.com/documentation`
[6]`http://apiminer.org/`
[7]`http://www.codota.com/`
[8]`http://kodebeagle.com/`

1.4 Structure of the Dissertation

The rest of this report is organised as follows: In Chapter 2 we set the appropriate background, by introducing terms and techniques that are going to be mentioned during the analysis of our implementation. Then, in Chapter 3, we define the problem faced in this dissertation, while providing an overview of the state-of-the-art systems that perform API usage mining, with the aim to present usage examples to the users. Chapter 4 describes the approach and the methodology followed in order to solve the problem, as well as in order to tackle any obstacles, before analysing the actual implementation of the system in Chapter 5. There, we extensively analyse each component of the system, and any algorithms implemented and used from our system. That chapter is followed by Chapter 6, where we describe the evaluation methodology, in terms of the metrics used and the experiments conducted, while presenting and interpreting the results. Finally, in Chapter 7 we present the contributions and limitations of this work, and suggest possible enhancements. For the sake of a better understanding of the implementation, as well as of the experiments' results, additional material, including indicative snippets that are mined by our system, are presented in the appendices.

Chapter 2

Background

2.1 Overview

During the last 15 years, there has been an increasing number of publications which harmonically combine the *Software Engineering* field with that of *Data Mining*. The combination of these two fields has led to the generation of a new research area, that of *Mining Software Repositories* (MSR). More interestingly, an international workshop on MSR[1] has been organised as part of the *International Conference on Software Engineering*[2] (ICSE), since 2004. The organisers of this workshop consider the MSR field as the one that "analyses the rich data available in software repositories to uncover interesting and actionable information about software systems and projects."

2.2 Mining Software Repositories

Since the official birth of the MSR field, in 2004, there have been a few publications that aim to provide an overview of this field. Hassan [6] identifies historical repositories, such as source-control[3] and bug repositories, as the main type of software repositories, to be mined for revealing interesting information about software projects.

Regarding the methodology used for the purposes of MSR, Kagdi et al. mention, among others, the static source code analysis, which is used extensively for bug finding and fixing, or even Information Retrieval methods, which are used for classification and clustering of similar documents. They also discuss frequent pattern mining, with the

[1]`http://msrconf.org`

[2]`http://www.icse-conferences.org/`

[3]Commonly known as *version* or even *revision control* repositories. Popular examples include *Git* (with *GitHub* being the most prevalent service of this type), and *Subversion* controls.

purpose of mining usage patterns, referring to the popular *Dynamine* system, presented in [8]. This has been one of the first systems to mine usage patterns, with its key hypothesis being that “violations of useful patterns are potential sources of errors”.

After an overview of surveys such as the one conducted by Kagdi et al., we see that the early systems in the MSR field mainly targeted bug identification. Although this purpose is still popular, the large amount of source code that is now available in software repositories, as well as the increasing trend in leveraging third party libraries, that aim to facilitate the coding process, introduced a new aspect for mining software repositories; that of mining usage patterns that target the APIs of popular third-party libraries.

2.3 Mining API Usage Patterns

Indeed, Khatoon et al. [9] identify *API usage* as one of the three primary source code mining scopes[4]. ISHAG et al. [10] define the problem of API usage pattern mining as “the process of finding the proper usage sequence or order of a group of reusable code elements within an API”. This requires searching for API source code inside client code -located in local software repositories or in the web- which, at a method level, may be reduced to API methods (referred to as *API method calls* or even as *method calls*), inside *client methods*, as these are defined below:

Client code A source code file, that may contain usages of the target API.

Client method A method of the client code.

API method call A method of the API that is invoked inside a client method.

The first step in order for a collection of source code files to be mined for existing -frequent- patterns, is to determine on the preprocessing approach of the available source code. According to ISHAG et al. [10] there are three paths of mining techniques based on the way the datasets are preprocessed; namely *itemsets*, *sequences* and *graphs*.

In this section we are going to explain both of them. For the sake of a better understanding of the differences between these three paths of minings techniques, we firstly provide a list of indicative snippets, in Figure 2.1.

[4]With *programming rules* and *copy-paste detection* being the other two.

```
void client_m1(){
  ...
  Object a;
  ...
  a.API_m1();
  ...
  a.API_m2();
  ...
  a.API_m3();
  ...
  ...
  ...
}
```

(a)

```
void client_m2(){
  ...
  Object a;
  ...
  a.API_m1();
  ...
  a.API_m1();
  ...
  a.API_m2();
  ...
  a.API_m3();
  ...
}
```

(b)

```
void client_m3(){
  ...
  Object a;
  ...
  a.API_m2();
  ...
  a.API_m3();
  ...
  a.API_m1();
  ...
  a.API_m2();
  ...
}
```

(c)

Figure 2.1: Client method examples where the API methods are called (a) once in a fixed order, (b) more than once in a fixed order, and (c) more than once in a random order.

2.3.1 Itemset Mining

Generally speaking, an *itemset* is a collection of unique items, where the order in which the items show is not preserved. The most popular use of this representation is linked to the *market-basket analysis*[5], which describes many-to-many relationships between baskets (also called *transactions*, and defined in Definition 1) and itemsets [11]. For instance, a basket may relate to the itemset $\{cheese, bacon, bread\}$.

Definition 1 *Using a set of items $I = \{i_1, i_2, ..., i_n\}$, a transaction T can be represented as a tuple in the form (id, items), where id uniquely identifies T, while items indicates a subset of items in I.*

Itemset mining is then linked to *association rule mining*, where an association rule indicates a frequent relation in the form $\{cheese,bacon\} \to \{bread\}$, which means that a client that buys the set of items on the left hand would probably also buy the set of items on the right hand. In addition to the market basket data domain, this concept has been applied to a large number of other domains, including Web mining, bioinformatics, and medical diagnosis [12].

In terms of source code and more specifically of method calls, we can think of a client method as a transaction, and of the API method calls in it as its itemset. Table 2.1 shows the transactions generated when the API method calls of the client methods

[5]`http://snowplowanalytics.com/guides/recipes/catalog-analytics/market-basket-analysis-identifying-products-that-sell-well-together.html`

presented in Figure 2.1 are represented as itemsets. As we can see, every transaction consists of a "bag" of API calls, thus omitting any information about multiple invocations of the same API method, and about the order on which the API methods are called.

Table 2.1: Transactions generated when the API method calls presented in Figure 2.1 are represented as itemsets.

Transaction	Itemset
$client\_m_1$	$\{API\_m_1, API\_m_2, API\_m_3\}$
$client\_m_2$	$\{API\_m_1, API\_m_2, API\_m_3\}$
$client\_m_3$	$\{API\_m1, API\_m_2, API\_m_3\}$

A large number of general-purpose algorithms that mine itemsets have been proposed, including the popular *Apriori* [13] or the *FP-growth* [14] algorithms, while there are several publications on mining frequent (or interesting) itemsets that target source code, aiming to facilitate API usage [8, 15][16]. The definition of a frequent itemset is given in Definition 2.

Definition 2 *Given a support threshold min_sup, an itemset I with support $sup(I)$ is a frequent one, if $sup(I) \geq min\_sup$. The support of an itemset indicates the number of transactions that contain the itemset.*

Note that, although the *interestingness* of an itemset/sequence had at first been linked to its frequency in a given database, most recent approaches use statistical models to mine interesting itemsets/sequences.

2.3.2 Sequence Mining

Similarly to an itemset, a *sequence* is a collection of items, however, in this collection, the order in which the items show is preserved, while duplicates may exist.

Table 2.2 shows the transactions generated when the API method calls of the client methods presented in Figure 2.1 are represented as sequences.

Sequence mining is a well studied problem; there is a great research in mining DNA sequences, or even time-series, while there are numerous algorithms that perform sequence mining. Some striking examples include the *SPADE* algorithm [17] that mines frequent sequences (defined in Definition 3), and the *BIDE* algorithm [18] which

Table 2.2: Transactions generated when the API method calls presented in Figure 2.1 are represented as sequences.

Transaction	Sequence
$client\_m_1$	$\{API\_m_1, API\_m_2, API\_m_3\}$
$client\_m_2$	$\{API\_m_1, API\_m_1, API\_m_2, API\_m_3\}$
$client\_m_3$	$\{API\_m_2, API\_m_3, API\_m_1, API\_m_2\}$

is used extensively in systems that conduct API usage mining in order to mine frequent closed sequences (defined in Definition 4). Moreover, algorithms that mine interesting sequences rather than frequent ones have emerged during the last years [19].

Definition 3 *Similarly to a frequent itemset, a sequence S with support $sup(S)$ is frequent if, given a support threshold min_sup, it holds that $sup(S) \geq min\_sup$.*

Definition 4 *A frequent closed sequence is a frequent sequence for which there is no super-sequence with the same support.*

2.3.3 Graph Mining

The most common way to represent a source code file is using an *Abstract Syntax Tree* (AST), which provides a convenient way to preserve useful nodes, while excluding others (e.g. parentheses, semicolons, etc.). An AST is an intermediate tree representation of the source code, and is the result of the reduction of a *parse tree*, produced by a parser generator. The concept behind this reduction is that a parse tree usually contains numerous nodes, most of which are furthermore redundant, in the sense that they do not contain any structure information. An indicative example of the AST representation used in the current project is presented in Appendix A.1, where we also illustrate the AST for a given source code, in Figure A.3.

ASTs tend to be a simpler source code representation structure than the *Abstract Semantic Graphs* (ASG), which provide a higher level of abstraction than the first ones, and that are typically constructed from an AST by a process of enrichment and abstraction.

Interestingly, the first step in order to mine source code in -almost- any case is to extract an AST for each of the source code files. Taking this into account, one could take advantage of as much information as possible stored in these trees. That is, although itemset and sequence mining have been proven efficient in terms of computational complexity, both of these representations lead to loss of information when they are used to represent structured text, such as source code.

In general, a more efficient way in that case would be to use a directed graph, where nodes represent statements in the source code, and edges indicate possible transitions between the statements. Common graph-based structures used in source code mining include *Control Flow Graphs* (CFGs), which tend to represent the paths that might be traversed through a program during its execution, or even the *Program Dependency Graphs* (PDGs), that capture dependencies between statements.

However, this huge quantity of information comes with a great computational cost. Indeed, working with CFGs and PDGs may seem an infeasible solution when mining large repositories.

Regarding API usage mining, the API method calls of a client code could be well represented either by a tree structure, as shown in Figure 2.2b, where the representation mainly captures the different levels on which the API methods are invoked, or even by a simplified graph, as the one illustrated in Figure 2.2c, which depicts a sequence of API calls, preserving loop and condition statements.

There has been a large number of frequent subgraph algorithms implemented so far [20, 21]. However, in the field of API mining, frequent subtree mining seems to be more common, and thus frequent subtree mining algorithms are usually applied to solve the problem of mining API usage patterns, with the data being represented using a tree structure.

With the purpose of leveraging a frequent subtree algorithm, one has to generate a database of transactions, similarly to the itemset mining problem. This can be achieved quite easily by serialising the tree using a delimiter, and a character that indicates level change.

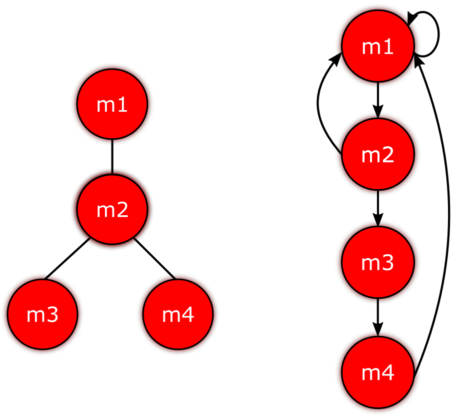
For instance, the tree illustrated in 2.2b can be well transformed to the transaction below:

$$\{m_1, m_2, m_3, -1, m_4, -1, -1, -1\}$$

where the comma (“,”) character is used as the delimiter between the API methods, and the number -1 indicates a level-up move[6].

[6]A careful reader may observe that such a transaction is quite similar to the ones used in sequence mining. This is true and stems from the fact that we have serialised the tree in a sequence-form, however, here we include the information of the nodes’ level, as well.

```
void client_m4(){
  ...
  Object a;
  ...
  while(...a.API_m1()...){
    ...
    if(...a.API_m2()...){
      ...
      a.API_m3();
      ...
      a.API_m4();
      ...
    }
  }
  ...
}
```



(a) (b) (c)

Figure 2.2: Figures illustrating (a) a Java source code example, (b) an AST-based representation of the example that captures the different levels on which the API methods are invoked, and (c) a graph-based representation of the example with respect to its API calls.

Similarly to the task of frequent itemset/sequence mining, the disadvantage of mining frequent subtrees is that, in a large database, there might be a huge amount of them, especially if the specified support is low. Based on this, it is quite common to mine closed or even maximal frequent subtrees, defined in Definition 5 and Definition 6, respectively.

Definition 5 *A frequent closed subtree is a frequent subtree where none of its super trees has the same support.*

Definition 6 *A frequent maximal subtree is a frequent subtree where none of its supertrees is frequent.*

As pointed out by Chi et al. in [22], mining frequent subtrees is an emerging field, with practical applications in domains including computational biology, Web mining, XML document mining, and computer networks.

A few examples of algorithms that mine frequent, as well as closed and maximal frequent subtrees include the *FREQT* [23], and the *CMTreeMiner* [24] algorithms.

2.4 Unsupervised Learning Techniques

Instead of mining association rules, ISHAG et al. [10] identify clustering techniques as an alternative to find reusable components. As we are going to see in the analysis of systems that perform API usage mining, clustering techniques are used heavily in this field.

Pang-Ning et al. [25] define *Clustering* as “the analysis that divides data into groups (clusters) that are meaningful, useful, or both”. The main objective is to form groups where the objects within them are similar (with respect to a similarity metric) to one another, and different from (or unrelated to) the objects in other groups. Clustering is one of the most common tasks in the *Unsupervised Learning* field, which involves mining useful information based on unlabeled data. The fact that there is no ground truth -in contrast to the *Supervised Learning* field- hinders the evaluation of the results, as well as the selection of the parameters used in the clustering algorithms, as we will note during the analysis of our implementation.

2.4.1 Types of Clustering Algorithms

There is a variety of types of clustering techniques, as well as of the clusters they generate[7]. Pang-Ning et al. [25] provide an overview of the different types of clustering algorithms, which can be summarised to the ones analysed below:

Partitional versus Hierarchical A partitional-based algorithm generates non-overlapping clusters, while the clusters of an hierarchical algorithm can be illustrated using a tree structure, where a cluster may contain subclusters.

Exclusive versus Overlapping versus Fuzzy An exclusive algorithm assigns each data point to a single cluster, in contrast to an overlapping one, where a single data point may belong to multiple clusters. Moreover, in a fuzzy clustering, every object is assigned a probability for each cluster, which indicates the probability of the point to belong to that cluster.

Complete versus Partial In contrast to complete clustering, where all the data points are assigned to a cluster, in a partial clustering, only the points that fulfil a defined criterion are being clustered.

[7]There seems to be a misunderstanding on this separation; many textbooks confuse the criterion used to assign data points into clusters with the structure of the generated clusters, separating, for instance, the density-based from the hierarchical algorithms.

There is also a separation with respect to the clusters generated by the various algorithms. With respect to this separation, we may have the below mentioned categories:

Well-separated cluster Any point in a cluster is closer (or more similar) to every other point in the cluster, than to any point not in the cluster.

Center-based cluster Each cluster is well represented by its center point which, ideally, summarises its cluster. Thus, data points that belong to a cluster are closer to their cluster's center, than to any other clusters' center.

Density-based cluster Clusters are generated based on dense areas of data points, which are, ideally, surrounded by sparse areas of data points.

Contiguous cluster Also called a *nearest-neighbour* cluster. A point in a cluster is closer (or more similar) to one or more other points in the cluster, than to any point not in the cluster.

2.4.2 k-means Clustering

One of the most popular clustering techniques is the *k-means* algorithm. It is a partitional, center-based technique, where a predefined number of clusters are created by assigning data points to clusters, based on their -euclidean- distance from the clusters' center points (called *centroids*). The pseudocode in Algorithm 1 summarises the process followed when clustering using the k-means algorithm.

Algorithm 1 k-means

```
1: procedure k-MEANS(k)
2:     Initialise k centroids
3:     repeat
4:         Compute distances between points and centroids
5:         Assign each point to its closest centroid's cluster
6:         Update centroids
7:     until centroids do not change
8: end procedure
```

As demonstrated, the first step of the algorithm is to select the initial centroids. This task may seem trivial, as one could select the first k points, or even random points from the dataset. Nevertheless, taking into consideration that, in many cases, this initialisation highly affects the clustering results, several techniques have been implemented in order to improve it. The most common technique is the k++ one, which selects the clusters' centroids incrementally, by making use of probabilities [26].

The k-means algorithm presumes data points in the Euclidean space[8], and its primary goal is to minimise the *Sum of the Squared Error* (SSE), as this is defined in Equation (2.1).

$$SSE = \sum_{i=1}^{k} \sum_{p \in C_i} dist(p, c_i)^2 \tag{2.1}$$

where k is the number of clusters, while c_i indicates the centroid of the cluster C_i.

An illustration of the k-means clustering is shown in Figure 2.3.

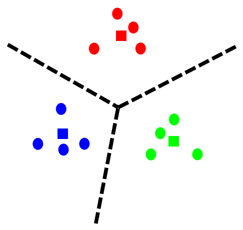


Figure 2.3: Clustering using the k-means algorithm.

2.4.3 k-medoids Clustering

Based on the fact that the k-means algorithm computes the centroids as the mean values of the clusters, it is plain to see that the algorithm is quite sensitive to possible outliers. Indeed, as Han et al. [27] point out, this effect is particularly exacerbated due to the use of the squared-error function.

A solution to this problem is to use data points as the centers of the clusters (also called *medoids*). This makes the *k-medoids* algorithm more robust to noise and outliers, compared to k-means, because the first minimises a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances.

One of the advantages of the k-medoids technique is that it can be used to cluster data not in the Euclidean space, which means that the user can use alternative distance metrics, instead of Euclidean ones, used when clustering with the k-means algorithm[9].

An illustration of the k-medoids clustering is shown in Figure 2.4.

[8]After all, a centroid is a multivariate mean, which inevitably relates to the Euclidean space.

[9]One could claim that we could transform a non-Euclidean distance into a Euclidean one. The hidden point here is that we have to ensure that our distance function can be minimised by the *mean*, in order for

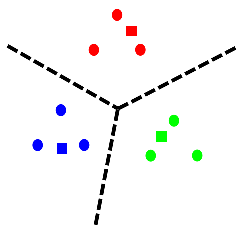


Figure 2.4: Clustering using the k-medoids algorithm.

2.4.4 DBSCAN Clustering

The algorithms analysed above are heavily used in the literature. However, they have a few limitations that may make them inefficient. For example, they are both appropriate for finding circular clusters, but they cannot identify arbitrarily shaped clusters. In addition to that, they cannot eliminate noisy data, which they try to assign to clusters, thus reducing clusters' quality.

The *DBSCAN* clustering technique comes as a solution to the aforementioned problems. The algorithm needs two input parameters that are described below:

Eps The maximum distance between two samples for them to be considered as in the same neighbourhood.

MinPoints The number of samples in a neighbourhood for a point to be considered as a core point.

Based on the values of the parameters, the DBSCAN algorithm classifies the data points into three categories, which are described below:

Core points A point p is a *core point*, if there exist at least $MinPoints$ points, whose distance from it is $\leq Eps$.

Border points A point p is a *border* (or *reachable*) point, if it falls within the neighbourhood of a core point, without being a core point itself.

Noise points Any point that is not a core or a border point (also called *outlier*) is classified as a *noise point*.

A nice simplified version of the algorithm that uses center-based densities is presented in [25] and is described in Algorithm 2, while an illustration of the DBSCAN

the k-means algorithm to be able to converge in a finite number of iterations. Here comes the k-medoids algorithm, which is based on the fact that there is a finite number of potential medoids.

clustering is depicted in Figure 2.5, where we can see that arbitrarily-shaped clusters have been successfully identified by the algorithm.

Algorithm 2 DBSCAN

```
1: Classify points as *core*, *border* or *noise* points
2: Remove noise points
3: Connect core points with distance $\leq Eps$ to each other with an edge
4: Create a cluster for each pair of connected core points
5: Assign border points to one of the clusters of its associated core points
```

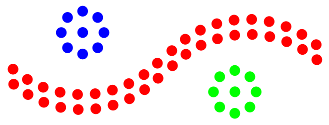


Figure 2.5: Clustering using the DBSCAN algorithm.

2.5 Document Summarisation

The problem of API summarisation could be linked to the document summarisation one. In fact, it could be viewed as a multi-document summarisation task where, according to Aliguliyev [28], the intention is to create a compressed version of a given collection of documents, that provides useful information to the users. Multi-document text summarisation has been an emerging field throughout the last 15 years. The *Document Understanding Conference*[10] (DUC), organised since 2001[11], has been the major forum for comparing summarization systems, while it also included the *multi-document summarization task*, with the purpose of creating fixed-length summaries, for a set of given documents.

When summarising an API, the collection of documents refers to the source code repository that is going to be mined, while the compressed version is in the form of API call sequences, or even better of snippets. A common approach in multi-document summarisation, is to leverage unsupervised techniques, in order to cluster similar sentences.

For instance, Wang et al. make use of semantic analysis, with the purpose of computing similarities between sentences, which they then cluster. This is followed by the

[10]`http://www-nlpir.nist.gov/projects/duc/index.html`

[11]DUC is now part of the *Text Analysis Conference* (TAC).

retrieval of the most informative sentences from each cluster, which are going to form the summary. Similarly, Boros et al. [30] use a combination of hierarchical and non-hierarchical clustering techniques, with the aim of partitioning a set of sentences into clusters, with each of them containing sentences covering only a single topic. A more recent approach is the one presented in [31], where Liu et al. build a deep learning model, in an attempt to efficiently solve the multi-document summarisation problem.

In addition to the multi-document summarisation problem, we are also going to consider the single-document summarisation task in this dissertation. This task aims to produce a summary of a single document. According to Aliguliyev [28], such a summary could be either *extractive* or *abstractive*. The first one includes these sentences that are considered important, in order for someone to understand the given document, while, according to Aliguliyev, the generation of the second one “usually needs information fusion, sentence compression and reformulation”.

2.6 Source Code Summarisation

Most of the systems that perform API mining so far present sequences of API method calls, rather than snippets, to the users. Although such sequences may seem interesting, they cannot adequately describe a source code example efficiently. On the other hand, presenting long snippets, such as the entire source code of the client methods that contain the mined sequences, may hinder the understanding of the target API, as these snippets usually contain several non-API relevant statements. This has prompted the authors of most recent publications in API usage mining to try to summarise usage examples [32, 33], or even to synthesise them, by combining information of similar snippets [34].

In contrast to text summarisation which is a well-studied field, source code summarisation techniques have only emerged during the last years. These two fields have many similarities, though. Ramanujam and Kaliappan [35] identify three characteristics of a good text summary; *coverage*, *coherence* and *redundancy*. We could claim that both characteristics apply to source code, too. What differentiates, however, the process of text summarisation from that of source code summarisation is that a source code, in contrast to free text, contains semantic as well as structural information. Hence, simple approaches that leverage text summarisation techniques (e.g. tf/idf), and that are based on keywords, cannot be efficiently applied to source code as they neglect the semantic context.

Similarly to a single text document summary, the major goal of a code fragment summary is to present the main ideas in the original fragment in less space. There have been interesting studies that try to define what a good source code example, and by extension summary, is. For instance, Nasehi et al. [36] introduce the notion of the *concise code*, which usually contains less than four lines of code, while replacing unnecessary details with placeholders (comments or ellipses). Moreover, Ying and Robillard [37] identify common summarisation practices (e.g. shortening identifiers or formatting code for readability) by conducting user studies.

Ying and Robillard [38] introduce a novel way to summarise code fragments, by exploiting ML techniques. Their system trains a classifier, using feature vectors, that contain features which are either syntactic or even related to the query, in order to decide on whether a line of the given source code should be in the summary or not. This is an interesting approach, but a general-purpose one.

Instead, taking into consideration that, in the case of API usage mining, we are mainly interested in statements that contain API-relevant information, Kim et al. [33] exploit *slicing*[12] techniques, to extract only the semantically relevant -to the given API method- lines. In the same direction, Montandon et al. [32] devise a summarisation algorithm, which uses *forward* and *backward slicing*[13] techniques, with the purpose of excluding the non-API statements. This seems a promising task-specific approach but, as we are going to explain during the analysis of our implementation, this could still result in undesirable redundancy.

[12]There have been almost 35 years since Weiser [39] defined *program slicing* as a method for isolating only the part of a program that is of interest to the user, based on a slicing criterion. Here, we are primarily interested in *static slicing*, which does not take into consideration the execution of the program.

[13]The purpose of *backward slicing*, is to find these statements that contribute to the -information used in the- defined criterion, while *forward slicing* identifies the statements that are affected by that.

Chapter 3

Existing Work on API Summarisation

3.1 Defining the Problem

As mentioned in Section 1.1, the problem we are facing in general is that of mining API usage patterns, which has been clearly described by ISHAG et al. in [10]. However, here we specifically define four main features that a system that performs API usage mining should exhibit. That is, given a -local- repository of Java source code files that are relevant to the target API, the system to be developed should be able to:

- ✓ Consider the structure of the source code files
- ✓ Cluster similar usage examples
- ✓ Summarise the examples into concise and readable snippets
- ✓ Present a ranked list of snippets to the users

As revealed from the above features, our primary goal is to mine common usage snippets that are concise and readable, instead of mining and presenting frequent sequences of API method calls. In addition to that, we are going to explore ML and NLP techniques in order to approach the problem efficiently, many of which have not been applied to any systems before.

The most striking examples of similar systems that conduct API usage mining are analysed in the next two sections. We divide them into two categories; the first one consists of systems that present sequences, and the second one of these that present snippets to the users.

3.2 Systems that Output Sequences

One of the very first systems to mine API usage patterns has been the *MAPO* system [40]. Its objective has been to generate usage patterns, that could then act as an index for the recommendation of code snippets. MAPO is an ordinary example of a system that performs frequent sequence mining. As regards the representation of the source code, the system firstly extracts API method call sequences from the given source code files. After that, MAPO clusters these sequences, on its most recent version [41], with the aim of improving the quality of the mined patterns. On its next step, the system combines the mined frequent call sequences -extracted using the *SPAM* sequence miner [42]- from each cluster, to produce patterns that are eventually presented to the users. The latest version of the MAPO system, accompanies the API sequences with their associated snippets. However, it is still more of a sequence-based approach, as it shows the source code of the client method without proceeding to any summarisation, while it also avoids to consider the structure of the snippets.

In [43] the authors argue that MAPO outputs a large number of usage patterns, many of which are moreover redundant. To overcome these issues, Wang et al. define *scalability*, *succinctness* and *high-coverage* as the main characteristics by which an API miner should abide. Based on these characteristics, their *UP-Miner* system focuses on qualitative usage patterns. Their approach includes a two-step clustering, as well as mining of frequent closed sequences -using the *BIDE* algorithm [18]- in order to mine usage patterns of single API methods. Although the UP-Miner tool is able to extract even more rear patterns, which cannot be extracted by the MAPO system, we believe that the use of probabilistic graphs of API method calls, in order for the first system to present the results may confuse the users, who definitely prefer simpler ways for the presentation of such results (e.g. ranked lists).

A brand new system that introduces several novelties with the purpose of presenting API usage patterns, in the form of sequences, is presented in [5]. Fowkes and Sutton point out that most of the heavily used frequent sequence mining methods require multiple user-defined parameters, that need to be hard-coded, and which are additionally hard to tune. With the aim of avoiding manually given parameters, they present a near parameter-free probabilistic algorithm (*PAM*), which is able to cluster method sequences, and to extract "the most informative API call patterns". In this way, Fowkes and Sutton apply techniques used in statistics and in the ML field to source code. This is the first system to mine API calls at GitHub scale, and to evaluate using handwritten examples that exist in libraries' `examples` directory, as *gold standards*[1]. Using examples written by the developers of a library to evaluate API miners seems a promising concept, that would enable the drawing of more objective conclusions, compared to the ones drawn from user studies. Additionally, using the libraries' `examples` directory allows the use of automatic evaluation methods, which would not usually be possible. We will dive more into this concept in Chapter 6. Although the outcome of this system is a list of method sequences rather than of usage snippets, we are going to leverage techniques used in this paper.

3.3 Systems that Output Snippets

Instead of presenting API-relevant sequences to the users, many systems during the last years output snippets. A possible usage scenario of such systems includes the enrichment of the APIs' documentation.

Upon improving the documentation of a target API, Kim et al. [33, 44, 45] firstly exploit slicing techniques, in order to summarise snippets retrieved by the *Koders*[2] CSE. Their summarisation algorithm preserves the lines of the source code that are relevant to the target API, using common backward and forward slicing. The next step is to represent the summarised snippets using feature vectors. For this task the system makes use of the *DECKARD* clone detection algorithm [46], which is a tree-based similarity detection algorithm, that proposes vectors for approximating the semantic context of ASTs. Kim et al. feed this algorithm with Java source code elements, while they also include additional features, such as the frequency of the query API method, and the lines of code in the snippets. Having extracted the feature vectors, they then experiment with three different approaches, in order to organise the generated examples; the *eXoaCluster* algorithm, that uses the k-means clustering technique to cluster similar usage examples, the *eXoaRank* algorithm, which ranks the examples based on their probability/frequency, and the *eXoaHybrid* algorithm, that combines both of the aforementioned approaches. A disadvantage of the *eXoaDocs* system we identify is that the system is not really capable of mining frequent patterns that include multiple API method calls. This is based on the fact that their feature vectors do not include any such information. Thus, the system mainly targets usage examples of single API methods.

[1]A *gold standard* (also known as "oracle example") is supposed to be an ideal example, based on which the quality of other examples can be evaluated.

[2]Renamed to *Black Duck Open Hub Code Search*, before being discontinued in June 2016.

Another approach that is not quite common is that followed in [47], where Wang et al. make use of the *Google Search* web search engine, in order to find useful examples. The *APIExample* tool extracts code snippets, as well as their surrounding text from webpages, with the purpose of forming usage examples. It then clusters[3] these examples, and ranks them using intra-cluster and inter-cluster ranking heuristics. However, this tool does not summarise the usage examples, while we could argue that the clustering technique used does not lead to significant reduction in the number of the collected examples.

A more recent system that is presented in [32] is the *APIMiner*[4]. One of the highlights of the first version of this system is the summarisation algorithm that has been implemented by the authors, which uses backward and forward slicing, in order to preserve only the API-relevant statements of a source code file. In its most recent version, which is based on the work presented in [48], the APIMiner leverages association rule techniques, and uses an improved version of the summarisation algorithm, with the aim of resolving variable types, or adding abstractive comments. However, it does not cluster similar usage examples, and additionally, our investigation of the system, revealed that most of the examples show the usage of a single API method.

Even when slicing is employed in the aforementioned systems, mined examples often contain extraneous statements, as pointed out by Buse and Weimer [34]. Therefore, the authors introduce a novel system that synthesises representative and well-typed usage examples. To the best of our knowledge, this in the first system to synthesise abstract examples, which it then presents to the users. This system combines *path sensitive data flow analysis*[5], clustering, and pattern abstraction, in order to present usage examples that are quite complete and abstract. The mined snippets include abstract naming, as well as helpful code, such as `try` and `catch` statements. However, the fact that the source code is represented using graphs makes the system really complex, and probably inefficient as pointed out in [43].

[3]Here, only client methods that contain the same API calls are clustered together.

[4]`http://apiminer.org/`

[5]According to Winter et al. [49], in path sensitive data-flow analysis (or path-sensitive DFA), path information that reveals whether a path is feasible or not is collected, with the aim to report bugs from feasible paths only.

Chapter 4

Conceptual Design Work

4.1 Overview

In this chapter we present our approach to the problem defined in Section 3.1, while justifying any design decisions made at any particular stage. At first, we clearly describe the input and output of the system. Then, we provide an overview of our approach, before explaining each step and decision separately.

4.2 Input and Output of the System

In contrast to the majority of the systems that perform API usage mining, where the input of the system is usually a repository of source code files, as well as a class or a method of the target API, our system needs a slightly different input as this is described below:

Input A repository of Java source code files (client code), as well as a `.arff` file, consisting of client methods, and of their associated API call sequences. Each row in the `.arff` file is in the form presented in Figure 4.1, where the `client_method` (also called "caller") and any $\texttt{API\_call}_{\texttt{i}}$ (also called "call") are fully qualified names. A sample row in the `.arff` file for the `Twitter4J` API is shown in Figure 4.2.

Output A ranked list of Java snippets. An example is shown in Figure C.11.

As revealed by the input of our system, we aim to mine usage example for the entire target API, rather than for a single method or a class of it. However, an input in the latter forms would only involve filtering of the mined snippets.

```
'client_method','API_call₁ ... API_callₙ'
```

Figure 4.1: General form of a transaction in a `.arff` file.

```
'com.gsbina.android.adot4j4a.ADOT4J4A.writeToken','twitter4j.auth.AccessToken.
    getToken twitter4j.auth.AccessToken.getTokenSecret'
```

Figure 4.2: Transaction in the `twitter4j.arff` file.

4.2.1 Justifying the Form of the Input

Our decision to use a different input from that used in the literature, stems from the fact that our team has already created a well-formed local corpus of the top Java projects in GitHub[1] [50]. This, at first, prompted us to use this corpus as the source of our system.

Furthermore, taking into account the limited time for the elaboration of this dissertation, and the fact that there has already been some previous work on API usage mining from our team in [5], we decided to use the EXAMPLE dataset presented in that work. This would allow us to spent more valuable time on the main task of mining patterns, rather than on that of finding relevant files in the GitHub Java corpus. This dataset includes popular libraries and frameworks that also contain an `examples` directory, which is moreover going to enable the comparison of our system's mined snippets with these handwritten examples. For each library, Fowkes and Sutton have created a directory of source code files that import a class belonging a (sub)package of the library (client code), while they have also generated the aforementioned `.arff` files[2].

We point out that, ideally, there would exist a mapping between the `.arff` transactions and their associated source code files. However, as this information was not available for the EXAMPLE dataset, we created a script that uses a best-effort approach, in order to do this. Our approach uses the fully qualified name of each client method, with the aim of finding its source code file. When there are duplicate class files, (these are labelled: `Class.java`, `Class_2.java`, `Class_3.java`, etc), we parse the package declaration of each class file, and match it to that of the client method's one.

[1] `http://groups.inf.ed.ac.uk/cup/javaGithub/`

[2] The best-effort approach used to extract the API call sequences, and to generate the `.arff` file subsequently, is described extensively in [5], and is out of the scope of this dissertation.

4.3 Overview of the Proposed Methodology

Our methodology may be summarised in the following points:

1. **Clean the `.arff` file**
   This step filters the `.arff` file, by removing any sequences that are not of interest, as we are going to explain in Section 4.4.

2. **Cluster the sequences in the cleaned `.arff` file**
   In this step, we leverage clustering techniques, in order to cluster the API call sequences in the `.arff` file. We analyse the process in Section 4.5.

3. **Select a fixed number of sequences from each cluster**
   This is a clustering postprocessing step, that selects the top sequences from each cluster, as briefly explained in Section 4.6.

4. **Generate a summarised snippet for the source code associated with each selected sequence**
   In this step we retrieve the source code files that are associated with the sequences that have been selected in the previous step, and generate a summarised snippet for each of them, by leveraging a summarisation algorithm we implemented. We explain the process followed, as well as the decision to implement our own summarisation algorithm, in Section 4.7.

5. **Select a single summarised snippet from each cluster**
   In this step we use a tree edit distance metric, in order to select a single snippet from each cluster, as explained in Section 4.8.

6. **Rank the selected snippets, based on their support**
   This is the final step, where we rank the snippets in order of decreasing support. This is described in Section 4.9.

4.4 Cleaning the `.arff` File

A crucial step that precedes the application of clustering techniques to the data is the preprocessing step, which basically refers to the appropriate cleaning, that aims to remove the data that could break down even a powerful clustering algorithm. However, data cleaning has not only to do with the clustering quality; it also refers to the removal of any data that is not of interest. In our case, we proceed to the following removals:

- **Remove callers that refer to different versions of the same source code file**
  An identical caller name indicates a different version of the same file. We remove multiple versions of the same source code file, as we noticed that most of them are identical in terms of API call sequences, and that they would affect the clustering process. An example is shown in Figure 4.3.

- **Remove singleton sequences**
  These are the sequences that contain only a single API call. We claim that singletons do not contain any useful information, as they do not show any interaction between different API methods. An example is shown in Figure 4.4.

- **Remove pseudo-singleton sequences**
  These are the sequences that contain only a single API method, which is invoked multiple times. An example is shown in Figure 4.5.

- **Removes unique[3] sequences, if specified so**
  An interesting decision here is that of whether to remove the sequences that are unique. Although the removal of the unique sequences would result to more tight clusters, we would probably miss rare patterns, while the fact that they are unique does not mean that they do not share API calls with other sequences. Based on this, we decided to have this feature as a parameter of the preprocessing step, and consider this on the evaluation of the system.

```
'com.oreilly.android.otweet.layouts.StatusListItem.setStatus','twitter4j.
    Status.getUser twitter4j.User.getScreenName twitter4j.Status.getText
    twitter4j.User.getProfileImageURL'
'com.oreilly.android.otweet.layouts.StatusListItem.setStatus','twitter4j.
    Status.getUser twitter4j.User.getScreenName twitter4j.Status.getText'
'com.oreilly.android.otweet.layouts.StatusListItem.setStatus','twitter4j.
    Status.getUser twitter4j.User.getScreenName twitter4j.Status.getText'
```

Figure 4.3: Example indicating multiple versions of the same source code file.

```
'com.bourke.finch.lazylist.LazyAdapter.onClick','twitter4j.Status.getUser'
```

Figure 4.4: Example indicating a singleton sequence.

```
'twickery.web.SiteStreams.apply','twitter4j.User.getId twitter4j.User.getId'
```

Figure 4.5: Example indicating a pseudo-singleton sequence.

[3]We define a *unique sequence* as this sequence for which there is no other sequence that contains the same API calls, invoked in the same order. That is, there is no other identical sequence.

4.5 Clustering the Sequences

In this section we analyse any design decisions that are related to the clustering process. For instance, we justify our decision to apply clustering techniques at this stage, as well as our decision to cluster the sequences using a distance matrix instead of a feature vector. In addition to that, we visualise the `Twitter4J` dataset, in order to investigate whether there is a clear notion of clusters. Finally, we provide a brief overview of the clustering techniques we implemented, which are going to be analysed extensively in the next chapter.

4.5.1 Why Clustering at This Stage?

Our decision to use clustering techniques, and more specifically at this stage, is quite simple. At first, looking at the client code in Figures 4.6 and 4.7, we see that these two snippets are quite similar, while they contain the same API calls (these are highlighted appropriately). A clustering technique that is based on the API call sequences of the client code would cluster these two snippets together. Taking into account the large number of files in the repository, this seems a more efficient approach than a clustering technique that would consider the structure of the client code, too. After all, we take into consideration the structure of the clustered snippets at a next stage, as described in Section 4.8. Hence, this approach is a balance between time overhead and quality improvement.

```
{
   Editor editor;
   AccessToken accessToken;
   editor.putString(string, accessToken.getToken());
   editor.putString(string, accessToken.getTokenSecret());
}
```

Figure 4.6: Sample client code where the API calls are highlighted.

```
{
   Editor editor;
   AccessToken token;
   if (token != null) {
      editor.putString(string, token.getToken());
      editor.putString(string, token.getTokenSecret());
   }
}
```

Figure 4.7: Sample client code that contains the same API calls with the client code presented in Figure 4.6.

4.5.2 Feature Vector vs Distance Matrix

The common input of a clustering algorithm is a feature vector that is generated from the data. While clustering using a feature vector has the advantage that one can apply almost every clustering technique on the vector, we identify a few problems here; one of them is that the designer should decide on the features to be used. In our case, a feature vector could be a boolean vector, where each future would represent a unique API method[4]. As we understand, in this way we would represent the data as itemsets, rather than as sequences. The latter could be achieved by defining additional features that reveal the order in which the API methods are invoked.

Instead of generating a feature vector, probably the most common way to cluster sequences is to compute a distance/similarity matrix, using a sequence distance/similarity metric, and then cluster the sequences using this matrix. This approach cannot leverage algorithms that are based on Euclidean distances (such as the *k*-means one), but the majority of clustering techniques can be modified in order to receive a distance matrix instead of a feature vector as an input[5]. In order to compute the similarity between any two sequences we tried several metrics, and we are going to analyse the most consistent of them in the next chapter. Our conclusion is that there is no significant difference in the results when using the one or the other, and thus we decided not to consider this as a parameter during the evaluation of the system.

4.5.3 Visualising the Data

Considering that we now have a representation of our data, it would be interesting to try to visualise this. Taking into account the high-dimensional nature of the data, this would be tricky enough, though. Dimensionality reduction methods like the popular *PCA* or the *Isomap* kernels would not work in our case, as the data seem quite sparse. A technique that is usually applied in high-dimensional sparse data is this of the *t-SNE* method [51], which converts similarities between data points to joint probabilities. Trying to visualise the data[6] using the implementation of the t-SNE algorithm, which

[4]Notice that we are talking about API methods and not about API calls; an API method may be called multiple times.

[5]We point out here that, while pre-computing a distance matrix would result to higher memory usage, it would also lead to lower computational/time complexity, as any clustering algorithm would compute this information at some point.

[6]At this step, we do not remove unique sequences in order to have a comprehensive picture of our data.

is part of Python's `scikit-learn` library[7], we noticed that, although there were a few dense areas, most of the sequences are unique and similar to multiple other sequences, a fact that may confuse the t-SNE algorithm. A possible visualisation of the data is shown in Figure 4.8. We should point out here that the visualisation is highly affected by several factors, such as the sequence similarity metric, and the parameters used in the t-SNE algorithm. In any case, this plot indicates that it is difficult enough to predict the number of clusters -even with the use of popular metrics such as the *silhouette coefficient* or the *Elbow* method as we ascertained later- as there are no clear, with respect to cohesion and even more to separation, clusters in the data.

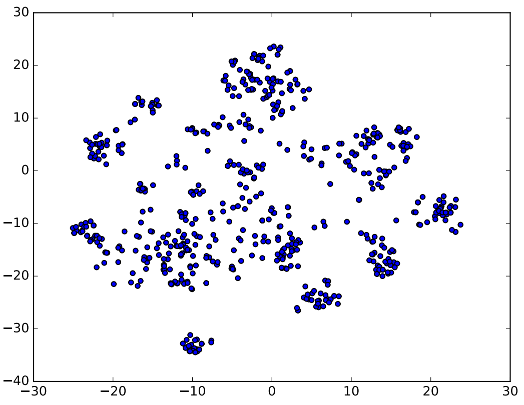


Figure 4.8: Plotting the sequences of the `Twitter4J` API using the *t-SNE* algorithm.

4.5.4 Applying Clustering Techniques

Regarding the clustering techniques that have been investigated in the current project, although we tried several algorithms, two of them have been proven promising, and thus we decided to evaluate two different versions of the system, with respect to the clustering algorithm used. The first one uses an implementation of the k-medoids algorithm, the basic version of which has been described in Section 2.4.3, while the second one is a hierarchical version of the DBSCAN algorithm (named *HDBSCAN*, and introduced in [52]). We also tried different approaches for the initialisation of the medoids for the k-medoids technique, and implemented the k++ technique which has been proven more efficient than a random initialisation for this task. Regarding the number of clusters, we tried to predict them using popular methods, including the

[7]`http://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html`

Elbow or the silhouette coefficient ones, however, none of them showed any cut-off point, and thus we decided to hard-code this parameter, using our intuition.

In any case, the result of the clustering process will be a clustered version of the `.arff` file. An example is shown in Figure 4.9, where the callers whose sequences have been clustered together are highlighted using the same colour.

```
'com.gsbina.android.adot4j4a.ADOT4J4A.writeToken','twitter4j.auth.AccessToken.
    getToken twitter4j.auth.AccessToken.getTokenSecret'
'org.jclouds.demo.tweetstore.controller.StoreTweetsController.apply','
    twitter4j.Status.getId twitter4j.Status.getText twitter4j.Status.getUser'
'jp.senchan.android.wasatter.task.TaskSetOAuthToken.doInBackground','twitter4j
    .auth.AccessToken.getToken twitter4j.auth.AccessToken.getTokenSecret
    twitter4j.TwitterException.printStackTrace'
'org.mule.twitter.TwitterConnector.setOauthVerifier','twitter4j.Twitter.
    getOAuthAccessToken twitter4j.auth.AccessToken.getToken twitter4j.auth.
    AccessToken.getTokenSecret'
'ru.abishev.wiki.example1.FunctionalDetector.onStatus','twitter4j.Status.
    getUser twitter4j.Status.getText twitter4j.Status.getText'
'de.bsd.zwitscher.OneTweetActivity.speak','twitter4j.Status.getUser twitter4j.
    Status.getText'
```

Figure 4.9: Clustered version of the `.arff` file.

4.6 Selecting the Most Representative Sequences

The next step after clustering the sequences, is to retrieve the source code of the most representative sequences of the clustering. A simple solution would be to select the most representative sequence of each cluster (the “medoid” sequence in the case of the *k*-medoids algorithm, or probably the sequence with the highest intra-cluster support for the HDBSCAN algorithm), and then retrieve and present its associated source code to the user. However, we decided to consider more than one sequences from each cluster; this includes any sequence that is identical to the most representative one. This would allow us to take into account the structure of the source code files that contain the same API calls, in order to select the one with the most common structure, as we are going to explain in Section 4.8, and mainly in Section 5.9. We decided to use a fixed number for the maximum number of sequences to be retrieved from each cluster, which is a balance between time overhead and quality improvement.

4.7 Generating Summarised Snippets

For each of the selected sequences, we retrieve its associated source code, and generate a summarised version of it. The main steps of this process are shown below:

1. Extract the body of the client method associated with the mined sequence.
2. Summarise the source code of the previous step, using the summarisation algorithm that has been implemented as part of the current dissertation.

Regarding the first step, our initial approach has been to get the subtree of the *Longest Common Ancestor* (LCA) of the statements where the API methods are invoked, and present this to the user. However, we decided to go even further and implement a summarisation algorithm, which is analysed extensively in the next chapter. Our decision on proceeding to such an implementation is explained in Section 4.7.1.

4.7.1 Implementing a Novel Summarisation Algorithm

After investigating summarisation algorithms that have been implemented as part of systems that perform API usage mining, we realised that the majority of them fails either in terms of quality, or in terms of time complexity.

More specifically, general-purpose algorithms, like the one presented in [38] cannot efficiently produce a summarised version of a client code. The primary reason is that they mainly include syntactic features, without giving the appropriate weight to the statements that are relevant to the API. Additionally, most of them summarise a code fragment based on lines rather than on statements, a fact that could lead to incomplete snippets.

Even the algorithms that are claimed to be specialised in the task of API usage mining, could result to high computational complexity, while the summarised code may still contain numerous statements, most of which can be considered redundant. For instance, the algorithm presented in [32] makes use of backward and forward slicing techniques, which inevitably lead to a high time overhead, taking into account that the backward and forward slicing is executed multiple times. Additionally, a client code in which the variables used in statements that are relevant to the API, are also used in several statements that are not relevant to the API, could not be efficiently summarised by this algorithm. As an example, the algorithm would not remove -almost- any of the statements in the client code presented in Figure B.9.

Based on the above, we proceeded to the implementation of a novel summarisation algorithm, which is going to be analysed in the next chapter. Our algorithm produces the summarised snippet presented in Figure B.10, when it is applied to the client code in Figure B.9. As we can see, it successfully produces a summarised version of the original client code, which is both readable and concise.

4.8 Selecting the Most Representative Snippets

Until this stage we have generated summarised snippets for the top files of each cluster. However, we have not yet taken into account the structure of the snippets. In an attempt to do so, we decided to leverage a tree edit distance metric[8] on this step. Using such a metric, we are then able to select the most representative snippet among the summarised snippets associated with the top files of each cluster. This can be achieved by creating, for each cluster, a distance matrix that stores the tree edit distance between any two top snippets of the cluster, and then selecting the snippet with the minimum sum of distances in its cluster's matrix.

Analysing the concept above, assuming N top snippets from each cluster, an $N \times N$ distance matrix is created, as shown in Table 4.1. The tree edit distance between the i_{th} and j_{th} snippet is stored in position ij of the matrix. Then, the sum of distances associated with the most representative snippet of this cluster is highlighted.

Table 4.1: Distance matrix that stores the tree edit distance between any two top snippets of a cluster. The index associated with the most representative snippet is coloured blue.

	0	1	2	3	4
0	0	12	11	9	15
1	12	0	18	17	3
2	11	18	0	0	21
3	9	17	0	0	20
4	15	3	21	20	0
sum	47	50	50	46	59

The computation of the tree edit distance between any two Java snippets is analysed in the next chapter.

4.9 Ranking the Selected Snippets

The order in which the results are shown to the users plays a crucial role for systems that perform API usage mining. Presenting a bag of snippets would lead to users dissatisfaction, as the number of these snippets may be quite large. This would require

[8]Pawlik and Augsten [53] define the *tree edit distance* between two trees as the minimum-cost sequence of node edit operations that transform one tree into another.

the users to spend enough time, in order to find for instance the most popular snippets. Considering that most of the systems use the support of the sequences/snippets in order to rank them accordingly, we worked in the same way.

Taking into account that there is no trivial way to compute the support for a source code file, we consider the API call sequence of each mined snippet. More specifically, if the API call sequence of a mined snippet is a subsequence of the API call sequence of a file in the repository, then we claim that the latter file supports the mined snippet. An example that shows the API call sequence of a mined snippet, and that of a file in the mined repository that supports it, is shown in Figure 4.10.

```
twitter4j.Status.getUser
twitter4j.Status.getText
```

(a)

```
twitter4j.Paging.<init>
twitter4j.Paging.sinceId
twitter4j.Paging.setCount
twitter4j.Status.getUser
twitter4j.Status.getText
```

(b)

Figure 4.10: API call sequence (a) of a mined snippet, (b) of a file in the repository that supports the mined snippet. The API call sequence of the mined snippet is a subsequence (highlighted API calls) of the file in the repository.

Having computed the support of each mined snippet, we then sort the snippets in descending order of support.

Chapter 5

Implementation

Having briefly analysed our approach to the problem, as well as the appropriate decisions we made, we can now proceed to the analysis of the actual implementation of our system.

5.1 System Overview

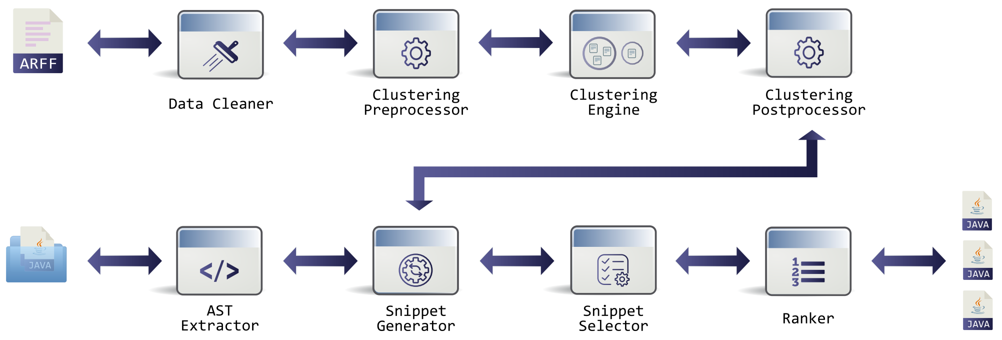


Figure 5.1: The architecture of the system.

The architecture of the system is illustrated in Figure 5.1. As depicted, the system comprises the below mentioned components:

AST Extractor Extracts the ASTs of the files in the repository.

Data Cleaner Cleans the `.arff` file and keeps only the API call sequences that are going to be clustered.

Clustering Preprocessor Creates a distance matrix for the sequences, using a sequence distance metric.

Clustering Engine This is the primary component of the clustering process. It clusters the sequences using the distance matrix and a clustering technique, and stores the results in data structures and files.

Clustering Postprocessor This component receives the clustering results and identifies the most representative sequences of each cluster. It then retrieves their associated ASTs, which have been extracted by the AST Extractor component, and feeds them to the *Snippet Generator* component.

Snippet Generator Generates summarised snippets for the files retrieved by the *Clustering Postprocessor* component, by leveraging the summarisation algorithm.

Snippet Selector Selects the most representative snippet among the generated snippets of each cluster, based on a tree-edit distance.

Ranker Ranks the selected snippets based on their support.

5.2 AST Extractor

This component extracts the ASTs of the source code files in the repository, which are then used by the *Snippet Generator* component, in order to generate summarised snippets. In order to extract the ASTs, we make use of the `srcml`[1] tool, which converts Java source code to the *srcML* format. The srcML format is an XML representation for source code, where the mark-up tags identify elements of the abstract syntaxt for the language. An example of the AST extracted by the srcml tool, for a given Java source code, is presented in Appendix A.1.

5.3 Data Cleaner

This component receives the content of the `.arff` file which, as described in Section 4.2, comprises of client methods and their associated API call sequences, and proceeds to the following modifications, as described in Section 4.4:

- Removes callers that refer to different versions of the same source code file.
- Removes singleton and pseudo-singleton sequences.
- Removes unique sequences, if specified so by the appropriate parameter.

[1]`http://www.srcml.org/about-srcml.html`

The result of this component is an updated version of the original dataset.

5.4 Clustering Preprocessor

This component creates a distance matrix, where the distance between any two sequences is stored. The distance metrics that have been implemented in order to compute the distance between any two sequences S_1 and S_2 are analysed below. Note that both distances are normalised in the range $[0.0, 1.0]$.

A. *LCS distance*

This metric makes use of the *Longest Common Subsequence* (LCS) between two sequences, in order to compute their distance. The formula of the *LCS distance* is shown in Equation (5.1).

$$LCS\_dist\,(S_1, S_2) = 1 - 2 \cdot \frac{|LCS\,(S_1, S_2)|}{|S_1| + |S_2|} \tag{5.1}$$

where $|LCS\,(S_1, S_2)|$ is the length of the LCS between the two sequences.

As an example, given two sequences $S_1 = BAACD$ and $S_2 = AACBDE$, it holds that $LCS(S_1, S_2) = AACD$, and thus $|LCS\,(S_1, S_2)| = 4$. Then, the LCS distance is:

$$LCS\_dist\,(S_1, S_2) = 1 - 2 \cdot \frac{4}{5+6} \simeq 0.27$$

B. *SeqSim distance*

The *SeqSim* metric is a sequence similarity metric that is based on *n-grams*[2], which has been introduced in [43]. At first, the authors define the n-gram set $G(S)$, for a sequence $S(s_1 s_2 \ldots s_n)$, as the collection of *unigrams*, *bi-grams*, ... , *n-grams* of S, as shown in Equation (5.2):

$$G(S) = \{s_1, s_2 \ldots, s_n,\ s_1 s_2, s_2 s_3 \ldots, s_{n-1} s_n,\ \ldots,\ s_1 s_2 \ldots s_{n-1}, s_2 s_3 \ldots s_{n-1} s_n, \\ s_1 s_2 \ldots s_{n-1} s_n\} \tag{5.2}$$

Then, the distance between S_1 and S_2 is computed using the formula in Equation (5.3).

$$SeqSim\_dist\,(S_1, S_2) = 1 - \frac{\sum_i Weight(g^i_\cap)}{\sum_i Weight(g^i_\cup)} \tag{5.3}$$

where $G_\cap = G(S_1) \cap G(S_2)$, $G_\cup = G(S_1) \cup G(S_2)$, $g^i_\cap \in G_\cap$, $g^i_\cup \in G_\cup$, and $Weight(g^i_\cap)$, $Weight(g^i_\cup)$ are equal to the length of $g^i_\cap$ and $g^i_\cup$, respectively.

[2]Given a text sequence, an *n-gram* is a contiguous sequence of n items from the given sequence. According to the n value, we may have *unigrams* ($n = 1$), *bigrams* ($n = 2$) and so on.

As an example, given two sequences $S_1 = BAACD$ and $S_2 = AACBDE$:

$$\begin{aligned}
G(S_1) &= \{B,A,...,D,\ BA,AA...,CD,\ ...,\ BAAC,AACD,\ BAACD\} \\
G(S_2) &= \{A,A,...,E,\ AA,AC...,DE,\ ...,\ AACBD,ACBDE,\ AACBDE\} \\
G_{\cap} &= \{A,B,C,D,\ AA,AC,\ AAC\} \\
G_{\cup} &= \{A,B,...,E,\ BA,AA,...DE,\ ...,BAACD,AACBD,ACBDE,\ AACBDE\}
\end{aligned}$$

which leads to:

$$\begin{aligned}
SeqSim\_dist\,(S_1,S_2) &= 1 - \frac{Weight(A) + Weight(B) + ... + Weight(AAC)}{Weight(A) + Weight(B) + ... + Weight(AACBDE)} \\
&= 1 - \frac{1+1+...+3}{1+1+...+6} \simeq 0.86
\end{aligned}$$

C. Jaccard distance

We have also implemented the *Jaccard* distance. This is basically an itemset distance metric, a fact that makes it not so accurate as the two aforementioned metrics, but it is far more efficient, in terms of computational complexity, and thus it may be used for large datasets. Its formula is presented in Equation (5.4).

$$Jaccard\_dist\,(S_1,S_2) = 1 - \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{5.4}$$

As an example, given two sequences $S_1 = BAACD$ and $S_2 = AACBDE$:

$$Jaccard\_dist\,(S_1,S_2) = 1 - \frac{|\{A,B,C,D\}|}{|\{A,B,C,D,E\}|} = 1 - \frac{4}{5} = 0.2$$

Table 5.1 shows a few indicative examples that point out the differences between the aforementioned metrics. As we can see, the *Jaccard_dist* metric may overestimate the similarity between the sequences, while the *SeqSim_dist* may underestimate it, a fact that could lead to a quite dense or sparse distance matrix respectively. It seems that the *LCS_metric* provides a balance between the other two metrics. For this reason, we decided to use this metric in the versions of the system that are going to be evaluated.

The outcome of this component is an $N \times N$ table, where N is the number of the sequences. Each sequence is assigned a unique id and, as a result, the value stored in the index i, j of the table is the distance between the i_{th} and the j_{th} sequences[3].

[3]Although we use a *redundant* (also called *full* or *complete*) distance matrix, we only need the upper triangular part -or even the lower one, as the first one is the transpose of the second one- of the distance matrix, taking into account that this is a symmetric matrix. A more efficient approach would use a *condensed* distance matrix, which does not contain any redundancy.

Table 5.1: Examples indicating the differences between the various sequence distance metrics that have been implemented.

S_1	S_2	*LCS_dist*	*SeqSim_dist*	*Jaccard_dist*
ABC	ABC	0.00	0.0	0.00
AB	CDE	1.00	1.00	1.00
ACBD	ABDC	0.25	0.82	0.00
BAACD	AACBDE	0.27	0.86	0.20
AAAB	AAB	0.14	0.44	0.00

5.5 Clustering Engine

The *Clustering Engine* component implements the clustering techniques that could be used in order to cluster the sequences. Although we have integrated/implemented several clustering algorithms, we are going to analyse only the two that are used in the actual implementation. These include an implementation of the k-medoids algorithm, as well as the HDBSCAN[4] algorithm [52].

This component receives the distance matrix as input, and clusters the sequences using one of the implemented clustering techniques, which are analysed in this section.

5.5.1 k-medoids Implementation

The implementation of the k-medoids algorithm is based on the one presented in [54]. We have also implemented the concept behind the k++ initialisation, in order to predict the initial medoids more efficiently. A simplified pseudocode of the k-medoids algorithm that has been implemented is shown in Algorithm 3, while the pseudocode of the k++ initialisation technique is presented in Algorithm 4.

We have implemented the k-medoids algorithm in a similar way to that used in the `sklearn.cluster` class of the Python's `scikit-learn` library, which includes several implementations of clustering techniques. That is, we created a class named `KMedoids`, which includes a `fit` method, that receives the distance matrix as a parameter and calls the `static` `k_medoids` function. The class can be instantiated and initialised using several parameters, including the number of clusters, the maximum number of iterations, and the initialisation technique to be used (either the k++ tech-

[4]`https://github.com/lmcinnes/hdbscan`

Algorithm 3 *k*-medoids

```
1: procedure k-MEDOIDS($k$, $max\_iter$)
2:     Initialise $k$ medoids using the $k$++ initialisation technique
3:     repeat
4:         Assign each point to its closest medoid's cluster
5:         Update medoids by computing the mean of each cluster and getting the data point whose distance to the mean is the minimum one
6:     until medoids do not change or $max\_iter$ reached
7: end procedure
```

Algorithm 4 *k*++

```
1: procedure k++($k$)
2:     Initialise the first medoid randomly
3:     repeat
4:         Create a distance matrix $D$ which stores, for each data point, its distance from its closest medoid
5:         Choose the data point $i$ as the next medoid, uniformly, at random, with probability $\frac{D(i)}{\sum_{i \in data} D(i)}$
6:     until $k$ medoids have been chosen
7: end procedure
```

nique, or a random, or even a fixed initialisation that makes use of a given array). The algorithm returns the medoids and the labels that indicate the clusters to which the data points have been assigned.

The main differences between our implementation and the one presented in [54] are summarised below:

- We have implemented the *k*++ initialisation in order to better predict the initial medoids, while we also provide the option of a predefined initialisation, given an array which comprises of the initial medoids.
- Our algorithm avoids the creation of empty clusters. This is mainly a problem when the random initialisation is passed as an option, as the implementation in [54] does not prevent from selecting identical data points as the initial medoids. Such a selection would lead to empty clusters and we avoid this by excluding identical data points from the random selection.

- We have implemented an additional criterion in order to decide on whether the algorithm has converged or not; this checks whether the members of the clusters have been assigned to a different cluster during the last iteration, rather than whether the medoids have been modified during that iteration. However, this option does not to show any real difference in the results for the datasets we use, and thus we decided to exclude it from the evaluation.

5.5.2 HDBSCAN Implementation

As regards the *HDBSCAN* algorithm [52], it is a hierarchical version of the DBSCAN algorithm, which has been described in Section 2.4.4, that varies the Eps parameter, and integrates the result to find a clustering that gives the best stability over this parameter. This allows HDBSCAN to find clusters of varying densities (unlike the basic DBSCAN algorithm), and to be more robust to parameter selection.

There is already a high performance implementation of the HDBSCAN algorithm written in Python by Leland McInnes[5], and thus we decided to make use of this implementation.

Algorithm 5 summarises the basic steps of the algorithm, as these are described in the original paper, as well as in the documentation of the implementation we use[6]. For the definition of the terms used in Algorithm 5, as well as for a deeper analysis of the algorithm, we encourage the reader to consider the original paper, as any further explanation is beyond the scope of this dissertation.

Algorithm 5 HDBSCAN

```
1: Compute the mutual reachability distance between all data points
2: Create the Mutual Reachability Graph (MRG), a weighted graph with the data points as
   vertices, and an edge between any two points with weight equal to the mutual reachability
   distance of those points
3: Compute the Minimum Spanning Tree (MST) of the MRG via Prim's algorithm
4: Convert the MST into an hierarchy of connected components
5: Condense the MST using the $min\_cluster\_size$ parameter
6: Compute the stability of each cluster and extract a flat, non-overlapping partition, that
   maximises the sum of these stabilities.
```

[5] https://github.com/lmcinnes/hdbscan

[6] http://nbviewer.jupyter.org/github/lmcinnes/hdbscan/blob/master/notebooks/How%20HDBSCAN%20Works.ipynb

Regarding the complexity of the algorithm, Campello et al. note that, in case a data matrix is provided as input (as in our case), both the time and the space complexity are reduced to $O(n^2)$.

Although this implementation does not have any required parameters, we set the `min_cluster_size` parameters to two. This parameter indicates the minimum number of data points that could form a new cluster (namely, the minimum size of each cluster).

5.6 Clustering Postprocessor

The next component receives the clustering results and retrieves a maximum number of files from each cluster, in order to generates snippets. In our actual implementation, we use a fixed number of maximum files to be retrieved, which has been set to $n = 5$, while we only retrieve the medoid of each cluster, and at most $n-1$ sequences that are identical to it. This would allow us to have more precise patterns, as choosing sequences that are not identical could probably result to quite dissimilar sequences.

This component feeds the *Snippet Generator* component with the top files to be used in order to generate snippets.

5.7 Snippet Generator

Having clustered the sequences and selected the top files of each cluster, the next step is to generate succinct snippets for these files. This component makes use of the ASTs extracted by the *AST Extractor* component. The process followed is analysed below:

- For each of the top files of the clusters, received by the *Clustering Postprocessor* component, we retrieve its associated `.xml` file, that contains its AST representation.

- We extract the part of the `.xml` file that refers to the client method of the top file, and more specifically, we extract the body of the client method. However, we also store any class variables, as well as the method's parameters, in order to be able to resolve variable types at a later stage.

- For each snippet of the previous step, we generate a summarised snippet, using the summarisation algorithm that has been implemented. Actually, for each `.xml` file of the previous step, which includes the body of the client methods, we generate a summarised `.xml` file, and convert it back to Java source code, using the srcml tool.

As an example, the `.xml` file associated with the caller `twickery.web.code.UserCode.name` is the one presented in Figure A.1, with the body of the client method associated with the caller being highlighted. Regarding the generated summarised snippet, we are going to analyse this process in the next subsection, and thus it is better to avoid to present any examples here[7].

5.8 Summarisation Algorithm

The summarisation algorithm receives as parameters the root of the `.xml` file, the API calls invoked in the snippet that is associated with the `.xml` file, and the variables that have been defined in a previous part of the source code (class variables and method's parameters). It then summarises the snippet, using the process presented in Algorithm 6, which is further analysed in this section.

Algorithm 6 Summariser

```
 1: procedure SUMMARISE($tree$, $API\_calls$, $decl\_vars$)
 2:     $tree \leftarrow$ Preprocess($tree$)
 3:     $API\_stmts$, $non\_API\_stmts \leftarrow$ ClassifyStmts($tree$, $API\_calls$)
 4:     $decl\_vars \leftarrow$ GetLocDeclVars($tree$, $decl\_vars$)
 5:     $tree \leftarrow$ RemNonAPIStmts($tree$, $non\_API\_stmts$)
 6:     $decl\_vars \leftarrow$ FilterDeclVars($tree$, $decl\_vars$)
 7:     $API\_decl\_vars \leftarrow$ GetAPIDeclVars($API\_stmts$)
 8:     $API\_vars\_not\_read \leftarrow$ CheckAPIDeclVarsRead($API\_decl\_vars$, $API\_stmts$)
 9:     Add a declaration statement for each $var \in decl\_vars$ in $tree$
10:     Add a descriptive comment for each $var \in API\_vars\_not\_read$ in $tree$
11:     Add descriptive comments for empty blocks in $tree$
12: end procedure
```

- *Line 2* firstly removes any comments. Moreover, it replaces any `literal` nodes in the srcML format with their srcML type. This includes nodes of srcML type `string`, `char`, `number`, `null`, and `boolean`. Such replacements lead to more abstractive snippets.

[7]After all, the presented example contains only a single statement and it does not make sense to present a summarised version of it.

- *Line 3* distinguishes between API statements and non-API statements, based on whether an API method is invoked or not in these statements, and stores the corresponding nodes in appropriate lists. An extensive analysis, as well as the pseudocode of this function is presented in Appendix B.1 (Algorithm 7).
- *Line 4* retrieves all the variables that are declared locally.
- *Line 5* removes all non-API statements by iterating the tree in reverse order (reverse bottom-up pre-order traversal).
- *Line 6* filters the list of the variables that are declared either locally or as class variables, by checking whether these are used in the summarised tree, as well as if they have already been declared at a previous point. In the latter case, there is no need to re-declare them. Note that this function is called after removing the non-API statements, in order to iterate over the summarised tree, rather than on the original one.
- *Line 7* retrieves the variables that are declared in API statements.
- *Line 8* checks whether the variables retrieved in the previous line are used only in non-API statements after that point, and keeps only these for which this statement holds true.
- *Line 9* adds declarations statements for all the variables retrieved in Line 6.
- *Line 10* adds descriptive comments in the form "`Do something with var`", where `var` is one of the variables retrieved in Line 8.
- *Line 11* finds empty blocks (as a result of no existing API statements in their body), and adds descriptive comments in the form "`Do something`", to improve the readability of the source code.

5.9 Snippet Selector

The *Snippet Selector* component receives the generated snippets of the clusters and selects one of them from each cluster, to be presented to the users. The process followed in order to select the most representative snippet of each cluster has been analysed in Section 4.8. However, there we intentionally did not explain the way the tree edit distance between two Java snippets is computed, as we use a tool for this purpose.

That is, in order to compute the tree edit distance between any two snippets, we leverage the APTED[8] tool, which implements the AP-TED algorithm, introduced in [53]. The AP-TED algorithm reduces the time complexity of the first-naive tree edit distance implementation, proposed in [55], from $O(n^6)$ to $O(n^2)$, while also achieving an $O(n^2)$ space complexity.

The APTED tool computes the tree edit distance between two trees, which are represented in a form similar to that presented in Section 2.3.3. For instance, a tree with a root node A, that has two children B and C, where B has a child D, is represented by the transaction below:

$$\{A\{B\{D\}\}\{C\}\}$$

In order to create such a transaction for an `.xml` file which is in the srcML format, we take into account the elements' tags in the `.xml` file. An indicative example that illustrates the whole process is presented in Appendix A.2.

As regards the parameters of the tool, one can set the cost for any insertion, deletion or renaming operation. However, we use the default values for these parameters, where the cost is 1 for any of these operations.

5.10 Ranker

The previous component selects a single snippet from each cluster. Then, we need to rank the mined snippets, in order to present them to the users in an order that reveals their importance. The *Ranker* component sorts the snippets based on their support, with respect to the files in the mined repository, as described in Section 4.9.

Finally, we apply the *Artistic Style*[9] formatter to the ranked snippets, in order to improve their readability. This formatter mainly fixes any improper indents and spacings, while ensuring consistent spacing (e.g. before all brackets) in the snippets. An example of the application of this tool to Java source code is presented in Appendix A.3.

[8] `http://tree-edit-distance.dbresearch.uni-salzburg.at/`
[9] `http://astyle.sourceforge.net/`

Chapter 6

Evaluation

6.1 Overview

As we have seen so far, several parameters are used in our implementation. For instance, one can enable or disable the preprocessing step that removes the unique sequences, or do the same for the summariser. Moreover, we have implemented two different clustering algorithms that may lead to different results, in terms of the sequences that are clustered together. This prompts us to evaluate different versions of the system which, however, should vary as little as possible with respect to the different conditions. Ideally, when evaluating between different versions of a system, these versions may only differ by a single parameter, in order to draw appropriate conclusions.

In the next section we are going to explain our decision to evaluate the system's results using the examples written by the developers of the libraries, rather than writing our own gold standard examples.

After that, we present the different versions that aim to be used for the evaluation of the system, as well as the dataset used. Then, we divide the evaluation process into several experiments where, for each one, we clearly point out the hypothesis that is going to be tested. This step-by-step evaluation process will enable us to evaluate different aspects of the system and of the results. Additionally, the last experiment intends to evaluate the key hypothesis of the current dissertation.

We decided to use only the `Twitter4J` dataset in the conducted experiments, as we noted that, even when using this single dataset, we draw numerous interesting conclusions. Moreover, the hypotheses defined in the experiments may be well evaluated using this dataset.

To avoid any confusion, we firstly describe three main terms that are going to be used in the evaluation process.

Sequence A sequence refers to an API call sequence. When this term is used in the evaluation metrics, this indicates the API call sequences of the snippets mined by our system.

Snippet A snippet refers to a Java code fragment mined by our system. The mined snippets are the results of our system.

Example An example refers to a handwritten Java code fragment, which is part of the `examples` directory of the target API.

6.2 Evaluating Using the `examples` Directory

In the majority of the publications in the field of API usage mining, the authors evaluate their systems with the aid of user studies. However, as we mentioned in Chapter 1, this inevitably leads to subjective results. Furthermore, Young and Pezze [56] correctly point out that "the human eye is a slow, expensive, and unreliable instrument for judging test outcomes". This prompts us to use a more automatic way of evaluating our system. Taking into account that writing our own gold standard examples would also lead to subjective results, while it is moreover time consuming, a better approach would be to look for such gold standard examples in the internet. Fortunately, the developers of several APIs include such examples in their repositories. We believe that this is a really valuable information that could be usefully utilised for the evaluation of systems that perform API usage mining. In addition to that, the evaluation using handwritten examples that exist in the `examples` directory of the target libraries, has already been tested in [5], where the system under evaluation mines API call sequences. Therefore, we decided to use a similar methodology, with the intention of evaluating snippets rather than sequences. To the best of our knowledge, this is the first system to be evaluated using this methodology, and we hope that the outcome would be interesting enough, in the hope that this methodology will be used for the evaluation of similar systems in the future. After all, many popular authors in the field of IR insinuate that this field seems to lack of efficient and objective evaluation techniques [57], and our approach may be helpful on that.

6.3 Versions of the System Under Evaluation

Table 6.1 illustrates the different versions of our system that are going to be considered during the evaluation process.

Table 6.1: Different versions of the system that are going to be considered under the evaluation process.

Version	Unique Sequences Removal	Clustering Algorithm	Summariser
RemUniqNaivNoSum	ON	Naive	OFF
RemUniqNaivSum	ON	Naive	ON
KeepUniqNaivSum	OFF	Naive	ON
KeepUniqKMedoidsSum	OFF	*k*-medoids	ON
KeepUniqHDBSCANSum	OFF	HDBSCAN	ON

As might be seen, the first parameter is related to the preprocessing step, and indicates whether the unique sequences are going to be removed or not. The second parameter is the clustering algorithm to be used, with the aim of clustering the API call sequences. The *naive* algorithm has not been analysed in Chapter 5, as it purely clusters identical sequences together. In fact, there was no need to implement a different algorithm, as this could be seen as a *k*-medoids version, where the number of clusters equals the number of the non-identical sequences after the preprocessing step. The versions that use this algorithm are going to reveal whether a more powerful technique, that clusters similar rather than identical sequences, leads to better results. Finally, the summariser that leverages the implemented summarisation algorithm can be enabled or disabled in advance.

The values of the input parameters for the *k*-medoids and the HDBSCAN clustering algorithms are going to be presented in the experiments.

6.4 Dataset

As regards the dataset used, we make use of the `Twitter4J` dataset, which is part of the EXAMPLE dataset, used in [5]. This dataset targets the `Twitter4J` API, and summary statistics for the client code, as well as for the `examples` that are related to this dataset are presented in Table 6.2. Taking into account that we are going to evaluate the results of the system using the handwritten examples in the `examples` directory as our gold standards, we decided not to split the dataset into a training and a test set. That is, the training set is the entire client code, while the test set includes the source code in the `examples` directory.

Table 6.2: Summary of the `Twitter4J` dataset, which is part of the EXAMPLES dataset presented in [5].

Attribute	Value
No. client files	549
Client LOCs	96,010
No. API classes in client files	81
No. API methods in client files	475
No. transactions in the `.arff` file	1,066
No. singleton sequences in the `.arff` file	367
No. pseudo-singleton sequences in the `.arff` file	39
No. unique sequences in the `.arff` file	585
No. example files	89
Example LOCs	5,458

6.5 Evaluation Metrics

In this section we present the metrics used in an attempt to evaluate the system. We make use of popular metrics related to the precision of the system, as well as to the coverage of the API methods in the dataset. Moreover, we define a metric that would enable us to evaluate whether presenting snippets rather than sequences is of more value to the developers. Furthermore, we make use of two quantitative metrics used in [34], with the purpose of evaluating the summarisation algorithm. We point out that the expression $|x|$ is a shortcut for the expression “number of x”.

6.5.1 Precision Metrics

Regarding the overall precision of the system, we define the *sequence_precision* and the *snippet_precision*, in Equations (6.1) and (6.2), respectively.

$$sequence\_precision = \frac{|\text{sequences contained in at least one example}|}{|\text{sequences}|} \tag{6.1}$$

$$snippet\_precision = \frac{\sum_{i=1}^{|snippets|} \frac{|tokens\_snippet_i \cap tokens\_gold_{snippet_i}|}{|tokens\_snippet_i|}}{|\text{snippets}|} \tag{6.2}$$

where $tokens\_gold_{snippet_i}$ are the tokens in the example that best matches to $snippet_i$.

The *sequence_precision* metric is computed based on the API call sequences between a mined snippet and a handwritten example, while the *snippet_precision* takes into account the entire snippets. More specifically, the latter metric computes the overlap between a snippet and its most similar example, in terms of Java tokens. The process followed in order to extract the tokens and to compute the *snippet_precision* metric, is extensively analysed in Appendix C.1.

We also define the *matched_snippet_precision*, in Equation (6.3), with the intention of eliminating the effect of the snippets that have not been matched to any of the examples, to the value of the *snippet_precision* metric.

$$matched\_snippet\_precision = \frac{\sum_{i=1}^{m} \frac{|tokens\_snippet_i \cap tokens\_gold_{snippet_i}|}{|tokens\_snippet_i|}}{m} \tag{6.3}$$

where m is the number of snippets that match to at least one example.

6.5.1.1 Evaluating the Precision of Ranked Results

Taking into account that our system returns a ranked list of snippets, we also make use of the *precision at top k* (*precision@k*), which is also known as *precision at cut-off k*. This metric indicates the precision calculated for the set of the top k documents returned, and is heavily used for the evaluation of ranked documents[1]. Such a metric would enable us to plot the precision for different values of k, while it would facilitate the comparison between versions of the system that return a different number of snippets. Based on this, and similarly to the precision metrics presented above, we firstly define the average sequence precision at top k (*avg_seq_prec@k*), in Equation (6.4)[2].

$$avg\_seq\_prec@k = \frac{\sum_{i=1}^{k} seq\_prec@i}{k} \tag{6.4}$$

[1] `http://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-ranked-retrieval-results-1.html`

[2] Usually, there is a confusion on how the average at top k is computed. In this evaluation, we use the formula in a similar way to that presented in `https://www.kaggle.com/c/avito-prohibited-content/forums/t/9584/average-precision-at-k-ap-k/50976#post50976`

where:

$$seq\_prec@i = \frac{|\text{matched sequences at top } i|}{i}$$

In a similar manner, the average snippet precision at top k ($avg\_snip\_prec@k$) is defined in Equation (6.5).

$$avg\_snip\_prec@k = \frac{\sum_{i=1}^{k} snip\_prec@i}{k} \tag{6.5}$$

where:

$$snip\_prec@i = \frac{\sum_{j=1}^{i} \frac{|tokens\_snippet_j \cap tokens\_gold_{snippet_j}|}{|tokens\_snippet_j|}}{i}$$

Another metric used in order to evaluate the quality of the sequences, as well as of the snippets, is the *Normalised Discounted Cumulative Gain*[3], which uses the graded relevance as a measure of usefulness.

The formula of this metric is shown in Equation (6.6).

$$nDCG_k = \frac{DCG_k}{IDCG_k} \tag{6.6}$$

where:

$$DCG_k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

and $IDCG_k$ is the maximum possible (ideal) DCG. Note that the value of rel_i indicates the relevance of the i_{th} recommended entity to the query.

Based on the above, we define the $sequence\_nDCG_k$ and the $snippet\_nDCG_k$. Regarding the value of rel_i, when computing the $sequence\_nDCG_k$, this is set to 1.0 if the snippet's sequence is contained in at least one example, and to 0.0 otherwise. This reveals that we consider it as a boolean variable. On the other hand, for the $snippet\_nDCG_k$, the value of rel_i indicates the similarity of a snippet to the example it best matches to, and lies in the range [0.0,1.0]. The $IDCG_k$ is in both cases set to 1.0.

[3] `https://www.kaggle.com/wiki/NormalizedDiscountedCumulativeGain`

6.5.2 Metrics Used for the Evaluation of the Key Hypothesis

With the purpose of evaluating the key hypothesis of the current dissertation, which is that presenting snippets rather than sequences would be of greater value to the developers, we additionally define the $addit\_snippet\_info$ metric, in Equation (6.7). This metric indicates the amount of information that is revealed when presenting snippets instead of sequences. In other words, it shows how much information is suppressed when presenting sequences instead of snippets. In order to compute this information, we can see an API call sequence as a sequence of tokens, where a token is the name of an API method, and refer to them as the *sequence-tokens* of a mined snippet. Then, the Java tokens extracted using the process in Appendix C.1 are referred to as the *snippet-tokens* of the mined snippet.

$$addit\_snippet\_info = \frac{\sum_{i=1}^{m} \frac{|tokens\_snippet_i \cap tokens\_gold_{snippet_i}|}{|tokens\_sequence_{snippet_i} \cap tokens\_gold_{snippet_i}|}}{m} \tag{6.7}$$

where $tokens\_sequence_{snippet_i}$ are the tokens of the sequence associated with $snippet_i$ (*sequence-tokens*), and m is the number of snippets that match to at least one example.

6.5.3 Coverage Metrics

On top of the aforementioned metrics, we also compute the percentage of the API methods covered by the mined snippets. This is a popular evaluation metric in the field of API usage mining, which is also considered in one of the few recent publications that aim to build API usage example metrics [58]. This metric is going to facilitate the decision on the best version of the system, as we will be able to investigate whether there is a trade-off between the precision of the mined snippets and the coverage of the API, in terms of the API's methods. Its formula is shown in Equation (6.8).

$$coverage\_of\_API\_methods = \frac{|\text{API methods covered by the mined snippets}|}{|\text{API methods in the dataset}|} \tag{6.8}$$

Note that the coverage is computed over the API methods in the dataset (`.arff` file), rather than over the total number of the methods defined in the API. This decision is based on the fact that, in case an API method does not exist in any of the files in our local repository, there is no way to cover this method. In this way, we focus on the approach followed to solve the problem, rather than on whether the dataset used covers the API.

6.5.4 Size and Readability Metrics

In order to evaluate the summarisation algorithm, we make use of two quantitative metrics, which have been used for the evaluation of similar systems [34]; the *Physical Lines of Code* (PLOCs) and the *readability* metric, which are analysed below.

In order to count the PLOCs of a source code file, we use the *loc-counter* tool[4], which counts any line that has more than three non-blank characters. It is clear that this is not a normalised metric (e.g. in the range [0.0, 1.0]).

As regards the readability of a source code file, we use the appropriate tool from Buse and Weimer[5], which is based on the authors' work in [59]. This metric is based on human studies and has been proven efficient enough to agree with a large set of human annotators. Given a Java source code file, the tool outputs a value in the range [0.0, 1.0], where a higher value indicates a more readable snippet.

Based on the above, we define the *avg_snippet_readability* in Equation (6.9), as well as the *avg_snippet_PLOCs* in Equation (6.10).

$$avg\_snippet\_readability = \frac{\sum_{i=1}^{|snippets|} \text{readability of } snippet_i}{|\text{snippets}|} \tag{6.9}$$

$$avg\_snippet\_PLOCs = \frac{\sum_{i=1}^{|snippets|} \text{PLOCs of } snippet_i}{|\text{snippets}|} \tag{6.10}$$

Having defined and explained the evaluation metrics, we can now go on with the experiments that have been conducted in order to evaluate the system.

6.6 Experiment 1 - Evaluating the Summarisation Algorithm

In the experiment of this section, we are going to investigate whether the integration of the implemented summarisation algorithm improves the results of the system. For this purpose, we use the first two versions presented in Table 6.1, namely the *RemUniqNaivSum* and the *RemUniqNaivNoSum* versions. In both of them the unique sequences are removed before the clustering process, while they both use the *naive* clustering technique, where only identical sequences are clustered together. The hypothesis under evaluation is described below:

[4]`https://java.net/projects/loc-counter/pages/Home`
[5]`http://www.arrestedcomputing.com/readability`

Hypothesis 1 *The summarisation algorithm leads to more concise and readable snippets, while improving their precision with respect to the handwritten examples.*

6.6.1 Quantitative results

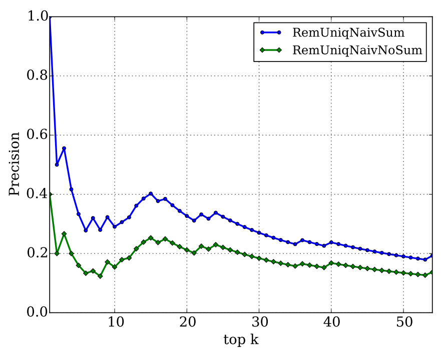
Table 6.3 shows the values of the metrics used for the evaluation of the summariser. It is clear that the precision of the system, with respect to the handwritten examples, has been improved. This is mostly revealed by the spectacular increase in the $matched\_snippet\_precision$ metric, where only the matched snippets affect the metric's value. Interpreting this value, well above than $2/3$ of the content of any mined snippet is used by its associated handwritten example, when using the summarisation algorithm.

Table 6.3: Evaluation metrics for the version that does not make use of the summariser ($RemUniqNaivNoSum$), and for this that does leverage it ($RemUniqNaivSum$).

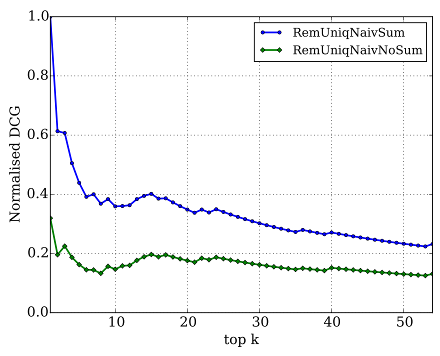
Metric	$RemUniqNaivNoSum$	$RemUniqNaivSum$
$snippet\_precision$	0.14	0.19
$matched\_snippet\_precision$	0.49	0.69
$avg\_snippet\_readability$	0.17	0.37
$avg\_snippet\_PLOCs$	13.13	8.09

Regarding the readability of the snippets, this is more than doubled when leveraging the summariser. We should point out here that, although the readability may seem low, even when using the summariser, this is mainly due to the fact that the readability metric considers features such as the lines' length or even the identifiers' length, as discussed in its designers' paper [59].

In addition to the readability of the snippets, there is a decrease in the number of PLOCs. This is definitely a result of the non-API statements removal, especially if we take into account that the summariser also resolves any variable types; a fact that should lead to longer snippets rather than to shorter ones. This decrease in the PLOCs would be more clear if we exclude the additional statements that aim to resolve variable types. In that case, we find out that the value of the $avg\_snippet\_PLOCs$ is further decreased to almost 5 PLOCs. Recalling the notion of the *concise code*, as this has been defined in [36], and mentioned in Section 2.6, we could claim that the summarisation algorithm successfully leads to concise snippets.



(a)



(b)

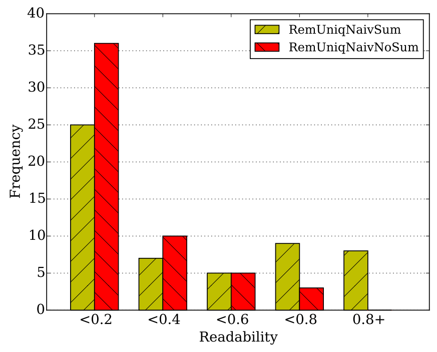
Figure 6.1: Figures illustrating (a) the $snippet\_precision@k$, and (b) the $snippet\_nDCG_k$ for both versions.

6.6.2 Visualising the Results

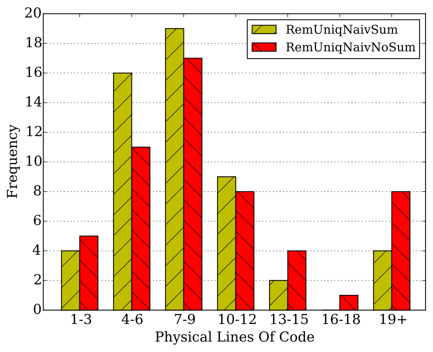
In order to better interpret the results, we firstly plot the sequence precision at top k, and the $snippet\_nDCG_k$, in Figures 6.1a and 6.1b, respectively.

It is clear from Figure 6.1a that the version that uses the summariser mines more precise snippets than the one that does not use it, for any value of k. The difference between the two versions is furthermore illustrated in Figure 6.1b, where the relevance is graded, with respect to the ranking. Both plots indicate a downward trend in the precision metric, which is justified by the fact that the mined snippets that are placed in lower positions are more complex; they normally contain a large number of API calls.

We also plot the distribution of the mined snippets in terms of their readability, in Figure 6.2a, as well as in terms of their PLOCs, in Figure 6.2b. As depicted in Figure 6.2a, the version that leverages the summariser generates almost 10 snippets with readability ≥ 0.8, out of the 54 that are mined in total, while the version that does not make use of the summarisation algorithm does not mine any snippet with a readability in that range. Moreover, almost half of the snippets generated by the $RemUniqNaivSum$ version have a readability ≥ 0.5. Inspecting the readability of each snippet manually, showed that there are four snippets with a readability value < 0.01 that heavily impact on the value of the $avg\_snippet\_readability$ metric. For this reason, Figure 6.2a illustrates the results in a more representative way than the aforementioned metric.



(a)



(b)

Figure 6.2: Figures illustrating (a) the readability distribution of the snippets, and (b) the corresponding PLOCs distribution, for both versions of the system.

As regards the distribution of the PLOCs, which is shown in Figure 6.2b, we point out that the majority of the snippets generated by the *RemUniqNaivSum* version contain less than 10 PLOCs. The reader may notice a strange behaviour in the plot for the lowest range $(1-3)$, where the *RemUniqNaivNoSum* version mines more snippets with PLOCs in this range than the *RemUniqNaivSum* version. Inspecting the results manually, we find out that this is due to the additional statements that aim to resolve variable types. In any case, Figure 6.2b makes clear that the snippets generated by the *RemUniqNaivSum* version are smaller in size than these mined by the *RemUniqNaivNoSum* version.

In addition to the quantitative metrics and the plots presented in the last two sections, we present a few indicative mined snippets of the two versions in Appendix C.2.

6.7 Experiment 2 - Evaluating the Preprocessing Step

In the experiment of this section, we are going to investigate whether the removal of the unique sequences, during the preprocessing step, improves the results of the system. In order to do so we use the *RemUniqNaivSum*, as well as the *KeepUniqNaivSum* versions of the system. In both of them the summariser is enabled, while they use the *naive* clustering technique, where only identical sequences are clustered together. Although these two versions result to a quite different number of clusters, and to a different number of mined snippets consequently, the number of clusters is computed using the same methodology, based on the number of the non-identical sequences in each case. The hypothesis under evaluation is described below:

Hypothesis 2 *Unique sequences often occur in handwritten examples, while removing them may additionally lead to limited coverage of the API.*

6.7.1 Quantitative results

The removal of the unique sequences reduces the number of sequences to be clustered from 660 to 175. Then, the number of clusters identified by the naive clustering technique, in the *RemUniqNaivSum* version, is 54. On the other hand, the *KeepUniqNaivSum* version, which does not remove the unique sequences, identifies 539 clusters, using the naive clustering technique. This huge difference in the number of clusters reveals that the removal of the unique sequences may lead to limited snippets, which do not sufficiently cover the API.

Table 6.4: Evaluation metrics for the version that removes the unique sequences (*RemUniqNaivSum*), and for this that does not remove them (*KeepUniqNaivSum*).

Metric	*RemUniqNaivSum*	*KeepUniqNaivSum*
sequence_precision	0.28	0.14
snippet_precision	0.20	0.08
avg_seq_prec@50	0.28	0.40
avg_snip_prec@50	0.19	0.24
$sequence\_nDCG_{50}$	0.34	0.42
$snippet\_nDCG_{50}$	0.23	0.25
coverage_of_API_methods	14.53	88.63

Indeed, as shown in Table 6.4, well less than 15% of the API methods are covered by the snippets mined by the *RemUniqNaivSum* version. This, compared to the fact that the handwritten examples cover around 38% of the API methods, indicates that *the removal of unique sequences actually leads to limited coverage of the API.*

As regards the number of snippets that match to handwritten examples, we find out that the *RemUniqNaivSum* version includes 15 of them, out of the 54 mined in total. On the other hand, 73 snippets out of the 520 mined by the *KeepUniqNaivSum* version match to at least one handwritten example. This shows that *unique sequences often occur in handwritten examples*.

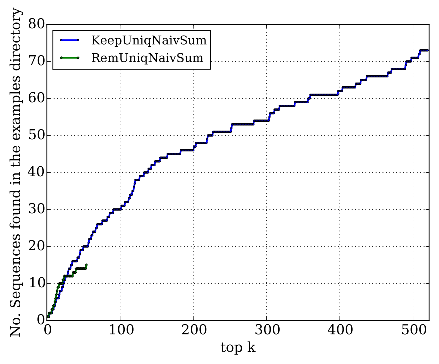
Looking at the precision metrics, we firstly present the *sequence_precision*, and the *snippet_precision*, which do not take into account the ranking, while they consider the total number of the results. However, a more fair comparison would take into consideration the same number of results for both versions. Such a comparison is possible using the next four metrics presented in Table 6.4, which show that *preserving the unique sequences does not only ensure a higher coverage in API methods, but it also ensures a higher precision among the top results*. We could claim that we expected such a behaviour, as the fact that a sequence is unique does not necessarily mean that it is not supported by other sequences; indeed, although a sequence may not be frequent on itself, it may be a subsequence of frequent sequences. In the latter case, the ranker plays a major role on the results of the metrics presented here, as it ranks the snippets based on their sequence's support in the dataset which, as explained in Section 5.10, takes into account its super-sequences.

6.7.2 Visualising the Results

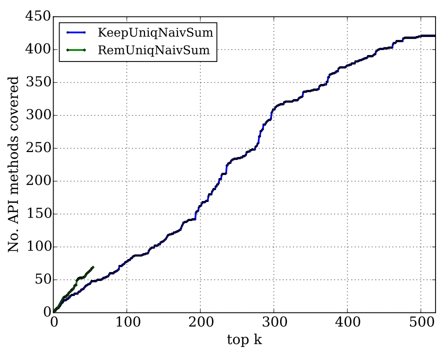
The analytical results of the metrics presented in Table 6.4 are plotted in this subsection.

Figure 6.3a indicates that it is quite possible that rare snippets -that contain unique sequences- exist in the `examples` directory. Although the relationship presented in this figure is not linear (it seems to be linear up to a threshold $k = 100$), it still shows that unique snippets -with respect to their API call sequences- are often part of the documentation of an API. Moreover, Figure 6.3b makes clear that, although the removal of unique sequences leads to higher coverage for the same value of k, up to a threshold (e.g. $k = 50$), this results to limited coverage of the API methods in general, for larger values of k.

Regarding the precision metrics, we illustrate the precision at top k in Figure 6.4, and the $nDCG_k$ in Figure 6.5, for both the snippets and their associated sequences. An interesting point shown in all these figures is that the *RemUniqNaivSum* version seems to ensure only a limited number (≤ 25) of high quality results, in terms of their precision to the handwritten examples. That is, if we were only interested in e.g. the top 20 snippets of the API, then probably this version could mine sufficient snippets of high precision. However, we believe that this number would only be acceptable for someone who is looking for a limited number of the top snippets of an API, rather than on snippets that contain specific methods or classes of the API.

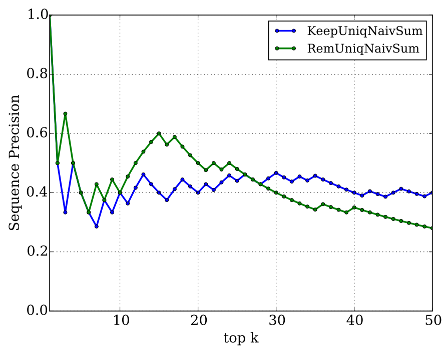


(a)

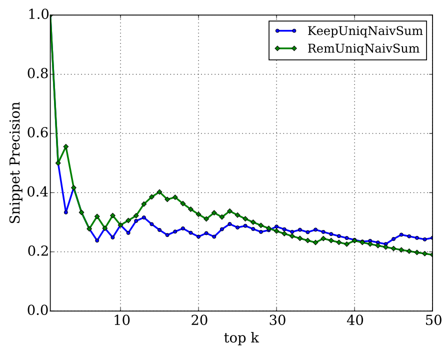


(b)

Figure 6.3: Figures illustrating (a) the number of snippets whose sequences match to at least one example, and (b) the number of API methods covered by the mined snippets, using the top k mined snippets for both versions.

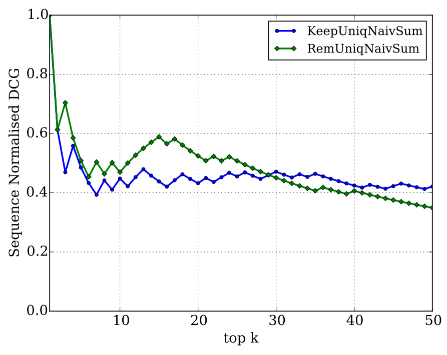


(a)

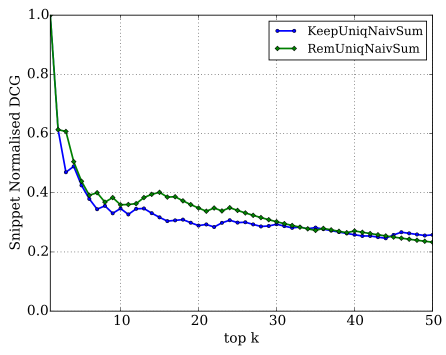


(b)

Figure 6.4: Figures illustrating (a) the sequence precision, and (b) the snippet precision at top k, for both versions.



(a)



(b)

Figure 6.5: Figures illustrating (a) the $sequece\_nDCG_k$, and (b) the $snippet\_nDCG_k$, for both versions.

6.8 Experiment 3 - Evaluating the Clustering Techniques

In the following experiment, we are going to investigate whether the application of different clustering techniques leads to different results. For this purpose we make use of the *KeepUniqKMedoidsSum*, as well as of the *KeepUniqHDBSCANSum* versions of the system. Both of them preserve the unique sequences in the preprocessing step, while they leverage the summariser. In order for the comparison between the k-medoids and the HDBSCAN algorithm to be fair, we use the same number of clusters for both of them. In order to ensure this, we set the *min_cluster_size* parameter of the HDBSCAN algorithm to 2, which leads to 110 clusters for the `Twitter4J` API. Then, we use the same number of clusters for the k-medoids algorithm. The hypothesis under evaluation is described below:

Hypothesis 3 *Different clustering techniques lead to similar clusterings, and mined snippets consequently, for the same number of clusters.*

The first part of the hypothesis could be evaluated based on qualitative examples, as well as based on clustering evaluation metrics. However, we are mainly interested in the second part of the hypothesis. This means that, even though the clusters formed by a clustering technique may be characterised by low *cohesion* and *separation*, or even *silhouette coefficient* values, the technique may lead to valuable mined snippets.

6.8.1 Quantitative Results

Evaluating the algorithms using clustering evaluation metrics, we find out that the clusterings of both algorithms show a quite low silhouette coefficient value. More specifically, the silhouette score for the *KeepUniqKMedoidsSum* version is 0.13, while the corresponding score for the *KeepUniqHDBSCANSum* is even lower (< 0.1). However, this is an effect of the quite noisy data, which prompts us to avoid using popular clustering evaluation metrics, for the evaluation of this part of the system. After all, based on this experiment's hypothesis, we are mainly interested on whether different clustering techniques lead to different clusterings and mined snippets, rather than on evaluating the quality of the clusters.

Table 6.5 presents the metrics used in order to decide on whether the different clustering techniques result to different mined snippets. At first, we see that the sequence-based precision metrics do not show any real difference. The HDBSCAN algorithm seems to perform slightly better than the k-medoids algorithm, however, the difference between the two algorithms is negligible enough to proceed to such a conclusion. In addition to that, both versions achieve similar coverage of the API. These two facts indicate that the two clustering techniques mine similar sequences. Indeed, the reader may find the top 10 sequences, with respect to their support in the dataset, mined by the two algorithms, in Appendix C.4, where it is shown that there is only a slight difference between the sequences mined by the two versions.

Table 6.5: Evaluation metrics for the version that uses the k-medoids clustering technique, and for that which uses the HDBSCAN algorithm.

Metric	*KeepUniqKMedoidsSum*	*KeepUniqHDBSCANSum*
sequence_precision	0.17	0.20
snippet_precision	0.11	0.12
matched_snippet_precision	0.65	0.62
$sequence\_nDCG_{110}$	0.25	0.26
$snippet\_nDCG_{110}$	0.16	0.16
coverage_of_API_methods	37.47	39.80

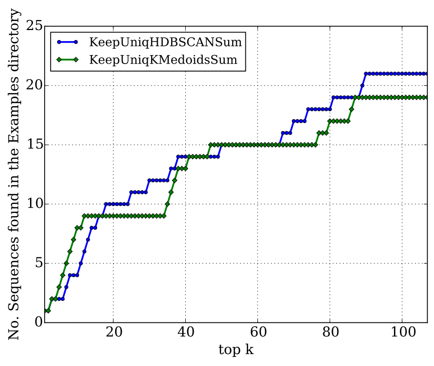
We also present the snippet-based precision metrics in Table 6.5, although these are not really relevant to this experiment, as the clustering techniques take into account only the API call sequences of the files, and not their source code, which is considered at a later stage.

6.8.2 Visualising the Results

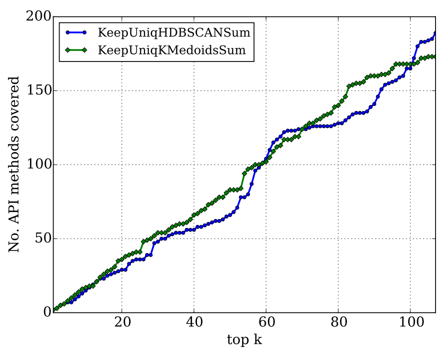
The analytical results of the sequence-based metrics presented in Table 6.5 are plotted in this subsection.

As depicted in Figure 6.6, the number of sequences found in the handwritten examples follows a similar trend for both versions, while the same number of methods are being covered by the two versions, for similar k values. Combining these plots with the metrics presented in the previous section, we see that the two clustering techniques lead to similar clusterings[6].

[6]A better interpretation is that both clustering techniques lead to similar clusters' centres, as actually the clusters' centres are used in order to mine a sequence from each cluster and then retrieve its source code file.

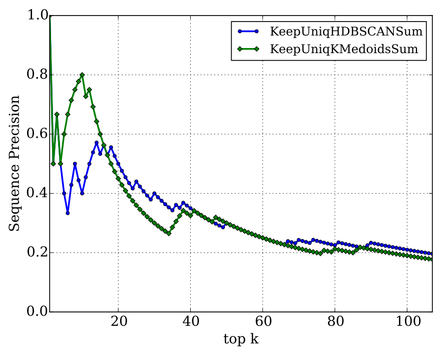


(a)

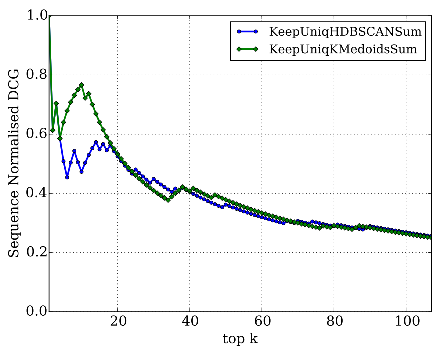


(b)

Figure 6.6: Figures illustrating (a) the number of snippets whose sequences match to at least one example, and (b) the number of API methods covered by the mined snippets, using the top k mined snippets, for both versions.



(a)



(b)

Figure 6.7: Figures illustrating (a) the sequence precision at top k, and (b) the $sequence\_nDCG_k$, for both versions.

Figure 6.7 shows a slightly increased sequence precision, for the k-medoids version, for the top 20 sequences. However, after a manual inspection, we find out that this can be explained as follows; the top sequences of the HDBSCAN algorithm are more complex sequences than these mined by the k-medoids technique, containing more than two API calls. Moreover, checking the support of these sequences, we surprisingly find out that the sequences mined by the k-medoids algorithm have lower support than these mined by the HDBSCAN technique. This interesting point, combined with all the previous results shown in this experiment leads us to one more conclusion; it is quite hard to determine on the best clustering technique for systems that conduct

API mining, without having an indication of the system's results, with respect to the handwritten examples.

Taking into account that there is no real difference between the two clustering techniques, we could claim that the HDBSCAN algorithm seem a more preferable solution, as this algorithm is almost parameter-free. This is based on the fact that we could, in any case, set the *min_cluster_size* parameter to 2, and the algorithm would then estimate the number of clusters.

6.9 Experiment 4 - Comparing Naive with Powerful Clustering Techniques

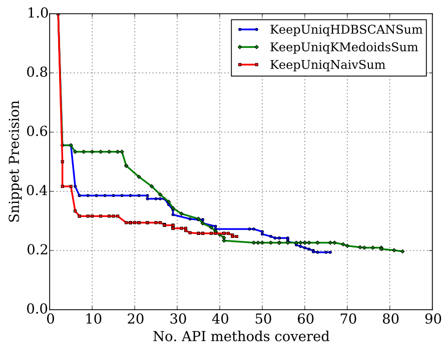
In the experiment presented in this section, we are going to investigate whether the application of clustering techniques that cluster similar rather than identical sequences improves the results. For this purpose we use the *KeepUniqNaiveSum*, as well as the *KeepUniqKMedoidsSum*, and the *KeepUniqHDBSCANSum* versions of the system. The first version clusters only identical sequences, and leads to a large number of clusters, while the other two versions cluster similar sequences, and output the most representative ones, thus leading to less clusters. Similarly to the previous example, we use the same number of clusters for the *KeepUniqKMedoidsSum* and the *KeepUniqHDBSCANSum* versions ($k = 110$). The hypothesis under evaluation is described below:

Hypothesis 4 *More powerful clustering techniques, that cluster similar rather than identical sequences, lead to more valuable snippets.*

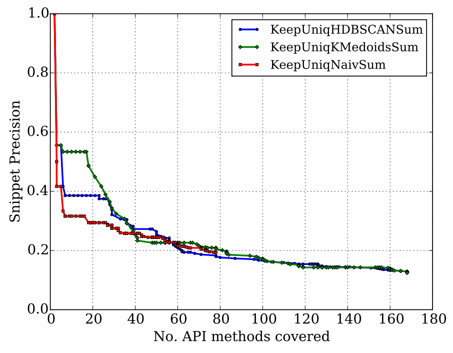
We believe that an efficient way in order to evaluate the hypothesis, is to plot the precision against the coverage of the API, in a similar manner to that used in the precision against recall figures. Such an illustration would reveal the value of the snippets to the developers. That is, if a system mines snippets that are precise and cover a large part of the API concurrently, then this system would be of great value to the developers.

A common way in order to eliminate the saw-tooth shape of the precision against recall curve, is to plot the *interpolated precision against recall curve*, where the highest precision found for any recall value (or level) $\geq$ current recall is shown[7]. In our case, we use the same concept in order to plot the precision against coverage curve. In

[7]http://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-ranked-retrieval-results-1.html



(a)



(b)

Figure 6.8: Figures illustrating the average interpolated snippet precision against the API methods coverage for three different versions of the system, that leverage different clustering techniques (a) using their top 50 mined snippets, and (b) using the top 100 mined snippets.

such a plot, the curve shows a decreasing trend and facilitates the drawing of possible conclusions.

In Figure 6.8a we illustrate the precision against coverage of the API for the top 50 mined snippets, while Figure 6.8b is generated using the top 100 mines snippets, and is obviously an extension of Figure 6.8a. It is noteworthy to say that being up and to the right is better.

These plots seem really interesting. The coverage in API methods, achieved by the *KeepUniqNaiveSum* version is quite low, which is related to the fact that this version clusters only identical sequences together. This means that it contains several similar snippets that use the same API calls (which is an indication of a high cohesion and a low separation). On the other hand, the *KeepUniqKMedoidsSum* and the *KeepUniqHDBSCANSum* versions achieve the best results here, and this indicates that there is a high separation between the clusters. More specifically, the *KeepUniqKMedoidsSum* version achieves way better results for the first 25 methods covered, as shown in Figure 6.8a. This increased precision has been explained in the previous experiment, but here we also notice a difference in the methods covered, where the *KeepUniqKMedoidsSum* version covers almost 85 API methods in its top 50 mines snippets, with the *KeepUniqHDBSCANSum* version covering well less than 65, for the same value of k. However, this difference seems to be eliminated in Figure 6.8b, which leads us to the conclusion that the *KeepUniqKMedoidsSum* version performs better than the *KeepUniqHDBSCANSum* version for smaller values of k.

In conclusion, this experiment indicates that the application of a clustering technique that clusters similar rather than identical sequences, leads to a better trade-off between the precision and coverage metrics which, in its turn, shows that *the mined snippets of more powerful clustering techniques are of greater value to the developers*.

6.10 Experiment 5 - Evaluating the Presentation of Snippets Rather Than of Sequences

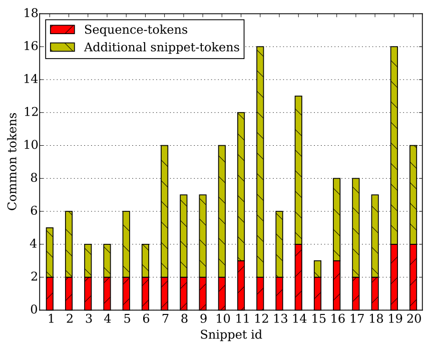
In this experiment we are going to investigate the key hypothesis of the current dissertation, which is that an API miner that presents snippets rather than sequences may help the users learn the target API easier. The hypothesis under evaluation is described below:

Hypothesis 5 *Snippets are more useful to developers than sequences of API calls.*

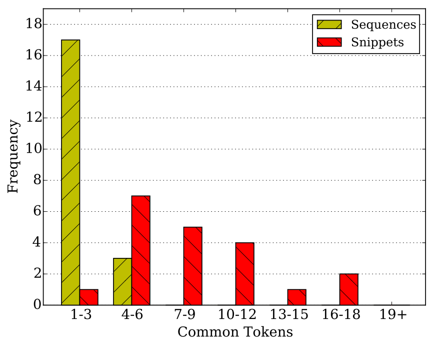
In an attempt to evaluate the hypothesis, we are going to make use of the *KeepUniqHDBSCANSum* version of the system, although the clustering technique does not play a key role in this evaluation. The evaluation metric that would help us to conclude is the $addit\_snippet\_info$ one, which computes the additional information, in terms of tokens, that is shown to the users, when presenting snippets instead of sequences.

Computing the value of this metric for the total number of the results (110 results), we find that the ratio between the snippets-tokens and the sequence-tokens, that are shared between the mined snippets and their associated examples, is 3.4. This means that the presentation of snippets instead of sequences leads to 3.4 times more information to the developers. This value is noticeably high, especially if we take into account that we have not used any techniques in order to predict identifier names, which would lead to more common snippet-tokens between the mined snippets and their associated examples.

Plotting the additional information revealed by the snippets in Figure 6.9a (this is coloured green) we see that, almost in any case, presenting snippets instead of sequences reveals at least twice as much valuable information as this revealed by the sequences. More interestingly, there are even cases where the number of common tokens between a mined snippet and its associated example have been increased by eight times (e.g. in *Snippet id*= 12), when the presentation of the associated sequences would only reveal a few common tokens (less than 4 in the usual case).



(a)



(b)

Figure 6.9: Figures illustrating (a) the additional information -in terms of Java tokens- revealed when mining snippets instead of sequences, for the top 20 mined snippets of the *KeepUniqHDBSCANSum* version of the system, and (b) the distribution of common tokens between the mined snippets and their associated examples, when presenting snippets and sequences.

In addition to Figure 6.9a, we also illustrate the distribution of the common tokens between the snippets and their associated examples, when these are presented as snippets and as API call sequences, in Figure 6.9b. This figure makes clear that the presentation of sequences leads to limited information about the usage of the API, concealing valuable information, which can be revealed when presenting snippets. It is quite interesting that there are mined snippets with more than 15 common tokens with their associated examples.

Furthermore, we present two indicative mined snippets in Appendix C.5, which show the additional information revealed when presenting snippets, as well as the power of our system to mine snippets that do not exist in the `examples` directory, which are however valuable.

Indubitably, both plots show that *much more information is revealed when presenting snippets instead of sequences*. This conclusion confirms our initial hypothesis, while it also shows that *our system successfully mines snippets that are valuable to the users, facilitating the use of the target API.*

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In accordance to Chapter 1, the primary goal of this dissertation has been to design a system that effectively summarises typical usages of a target API. We firstly described the problem faced in the current dissertation, in Section 3.1, where we moreover presented four main features that a system that performs API usage mining should exhibit. After that, we analysed our approach to the problem, in Chapter 4. There, we justified any design decisions on which we proceeded, while we also identified the decisions that would probably impact the results of the system, and which should be evaluated in the evaluation process. Then, in Chapter 5, we extensively analysed the actual implementation of the system, by providing an overview of the system's architecture, and describing the functionality of each component in its own section. Furthermore, in the same chapter, we analysed the algorithms that have been implemented and/or used by the system.

Taking into account that there were several design decisions that could impact the results of the system, we decided to evaluate five different versions of the system, by conducting experiments where, a pair of different versions each time, would evaluate a clear hypothesis. Hence, in Chapter 6, we clearly defined five different hypotheses, with the first four of them being related to the different algorithms that have been implemented, and the last one being the key hypothesis of the current dissertation, which is that the presentation of snippets would be of more value to the developers than that of API call sequences. The outcomes of the first four experiments were really interesting, showing the efficiency of the novel summarisation algorithm we implemented, as well as any possible differences between the various clustering techniques that we leveraged. In addition to that, the last experiment evidently revealed that the key hypothesis is confirmed by the system. Moreover, we believe that the indicative mined snippets of the system, presented in the appendices, are concise, readable, precise, and cover several usages of the target API. This means that they could be efficiently used for documentation purposes.

It should be now clear that the system we implemented exhibits the four features described in Section 3.1. More specifically, it clusters similar usage examples, by taking into account their API calls, while it also takes into consideration their structure at a later stage. Furthermore, the summarisation algorithm ensures concise and readable snippets, which are presented to the users as a ranked list, that indicates their popularity.

7.2 Future Work

Although we have tried several different techniques in the different components that have been implemented, including for instance various sequence similarity metrics, and clustering techniques, there are still several lines of improvement in the system. Most of them, however, have to do with the approach followed on each step of the implementation. In this section, we describe a number of possible enhancements, that are either related to the limitations of the system, or even to the decisions made on each step.

We believe that the most important component to be integrated to the system is the parser, that would enable the extraction of the `.arff` file without the need to provide this as an input. This would also facilitate possible further evaluation of the system, using additional libraries. We clearly justified the decision to avoid implementing this component in this dissertation, in Section 4.2.1.

Regarding the clustering techniques used in the implementation, it would be convenient enough to be able to predict the number of clusters for the *k*-medoids algorithm. Although, as explained in Section 4.5.4, this could not be easily achieved using popular techniques used for this task, we could probably try to define a metric such as the one used in [43], where Wang et al. define a dissimilarity metric between the mined usage patterns. In a similar manner, we could either make use of a sequence similarity metric, to compute the dissimilarity between the mined sequences, or even better of a tree edit distance metric, in order to compute the dissimilarity between the mined snippets. Then, we could restart the clustering process using a different *k* value, until achieving the highest possible dissimilarity between the mined snippets.

In addition to the aforementioned improvements, we could also extend the approach used to retrieve the top mined sequences from each cluster. That it, instead of retrieving a fixed number of sequences from each cluster, we could use a two-stage clustering approach where, after clustering based on the API call sequences of the snippets, we could further cluster the results of the formed clusters, using a tree edit distance metric. This would enable the retrieval of snippets that use the same API call sequence, but which differ in their structure.

Finally, a future work could definitely include the use of additional datasets, as well as the comparison of our system with that presented in [34]. The reason why we did not proceeded to such a comparison is that the version of this system that is available (a) does not support snippets that use features introduced in Java 1.8, and (b) there is additional misbehaviour of the system when outputting the mined snippets. Our attempt to fix these issues did not lead to a stable version of the system. Thus, the only way to retrieve its mined snippets is by manually proceeding to some cleaning, for which we would have to spent valuable time.

Appendix A

Tools Specifics

In this appendix we provide indicative examples for the most important tools that have been used in the implementation of our system. This includes the following tools:

`srcml` This tool converts source code files into the *srcML* format, which is basically an XML representation of the source code's AST.

`APTED` This tool computes the Tree Edit Distance between two Java source code files.

`Artistic Style` This is an automatic formatter for source code files, which supports Java source code files, too.

A.1 `srcml` Example

In the context of this dissertation, we could characterise the srcml tool as a powerful AST extractor, which is able to parse even uncompilable and incomplete source code files. One of its key characteristics, which differentiates it from other similar tools, is that it preserves the original source code in the srcML format, used to represent the AST of the source code. This is really helpful for tasks such as the source code synthesis one, as it is possible to retrieve the XML elements needed, using XPath[1] expressions. The srcml tool is also powerful in the sense of translation speed, parsing approximately 3.000 files per minute, as mentioned in the tool's website[2].

An indicative example that shows the transformation of a Java source code file into its srcML format is presented in Figure A.1 and in Figure A.2. As we can see, the entire source code is preserved in the `.xml` file, which can then be transformed back to the

[1]XPath is a query language for selecting nodes from an XML document.

[2]`http://www.srcml.org/about-srcml.html`

original Java source code, using the same tool. Note that the subtree that corresponds to the body of the `name` function is highlighted, and can be extracted using an XPath expression.

```java
import twitter4j.User;

public class UserCode {
  private User user;

  public UserCode(User user) {
    this.user = user;
  }

  String name() {
    return user.getName();
  }

}
```

Figure A.1: An example of a Java source code, to be transformed into the srcML format, using the srcml tool.

```xml
<import>import <name><name>twitter4j</name><operator>.</operator><name>User</name></name>;
     </import>

<class><specifier>public</specifier> class <name>UserCode</name> <block>{
  <decl_stmt><decl><specifier>private</specifier> <type><name>User</name></type> <name>user</
       name></decl>;</decl_stmt>

  <constructor><specifier>public</specifier> <name>UserCode</name><parameter_list>(<parameter><
       decl><type><name>User</name></type> <name>user</name></decl></parameter>)</
       parameter_list> <block>{
    <expr_stmt><expr><name><name>this</name><operator>.</operator><name>user</name></
         name> <operator>=</operator> <name>user</name></expr>;</expr_stmt>
  }</block></constructor>

  <function><type><name>String</name></type> <name>name</name><parameter_list>()</
       parameter_list><block>{
    <return>return <expr><call><name><name>user</name><operator>.</operator><name>getName
</name></name><argument_list>()</argument_list></call></expr>;</return>
  }</block></function>

}</block></class>
```

Figure A.2: AST of the file presented in Figure A.1, in the srcML format, extracted using the srcml tool.

A possible AST representation for the highlighted fragment in Figure A.2, which corresponds, as mentioned before, to the body of the `name` function (which is also highlighted in Figure A.1), is illustrated in Figure A.3.

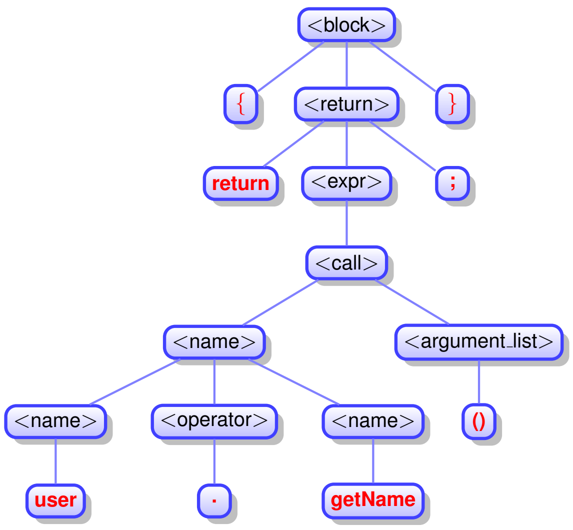


Figure A.3: Illustration of the AST of the highlighted fragment of the code presented in the Figure A.2. The text nodes that correspond to source code are coloured red.

A.2 `APTED` Example

Another tool used in our implementation is the `APTED`[3]. The tool considers three edit operations, namely the *insertion*, *deletion* and *renaming* (also called *substitution* in string edit distances) of a tree node. In Figures A.4 to A.9 we present an example of two Java source code files (Figures A.4 and A.5), that are transformed to their srcML format (Figures A.6 and A.7), and then to appropriate transactions in the form used by the APTED tool (Figures A.8 and A.9). As we can see, the transaction in Figure A.9 contains 8 additional nodes, with respect to the one in Figure A.9 (these are highlighted). Hence, the tree edit distance between these two trees, computed by the APTED tool, is 8, as there are 8 addition operations.

```
{
   Editor editor;
   AccessToken accessToken;
   editor.putString(string, accessToken.getToken());
   editor.putString(string, accessToken.getTokenSecret());
}
```

Figure A.4: Summarised snippet to be compared with the snippet in Figure A.5 using the APTED tool.

[3]`http://tree-edit-distance.dbresearch.uni-salzburg.at/` one, which computes the tree edit distance between two trees. The designers of the tool define the tree edit distance as “the minimum-cost sequence of node edit operations that transform one tree into another”

```
{
   Editor editor;
   AccessToken token;
   if (token != null) {
      editor.putString(string, token.getToken());
      editor.putString(string, token.getTokenSecret());
   }
}
```

Figure A.5: Summarised snippet to be compared with the snippet in Figure A.4, using the APTED tool.

```
<block>{
   <decl_stmt><decl><type><name>Editor</name></type> <name>editor</name></decl>;</decl_stmt
       >
   <decl_stmt><decl><type><name>AccessToken</name></type> <name>accessToken</name></decl
       >;</decl_stmt>
   <expr_stmt><expr><call><name><name>editor</name><operator>.</operator><name>putString</
       name></name><argument_list>(<argument><expr><literal type=”string”>string</literal></expr><
       /argument>, <argument><expr><call><name><name>accessToken</name><operator>.</
       operator><name>getToken</name></name><argument_list>()</argument_list></call></expr></
       argument>)</argument_list></call></expr>;</expr_stmt>
   <expr_stmt><expr><call><name><name>editor</name><operator>.</operator><name>putString</
       name></name><argument_list>(<argument><expr><literal type=”string”>string</literal></expr><
       /argument>, <argument><expr><call><name><name>accessToken</name><operator>.</
       operator><name>getTokenSecret</name></name><argument_list>()</argument_list></call></
       expr></argument>)</argument_list></call></expr>;</expr_stmt>
}</block>
```

Figure A.6: AST of the file presented in Figure A.4, in the srcML format, extracted using the srcml tool.

```
<block>{
   <decl_stmt><decl><type><name>Editor</name></type> <name>editor</name></decl>;</decl_stmt
       >
   <decl_stmt><decl><type><name>AccessToken</name></type> <name>token</name></decl>;</
       decl_stmt>
   <if>if <condition>(<expr><name>token</name> <operator>!=</operator> <literal type=”null”>null</
       literal></expr>)</condition><then> <block>{
      <expr_stmt><expr><call><name><name>editor</name><operator>.</operator><name>
          putString</name></name><argument_list>(<argument><expr><literal type=”string”>string</
          literal></expr></argument>, <argument><expr><call><name><name>token</name><
          operator>.</operator><name>getToken</name></name><argument_list>()</argument_list></
          call></expr></argument>)</argument_list></call></expr>;</expr_stmt>
      <expr_stmt><expr><call><name><name>editor</name><operator>.</operator><name>
          putString</name></name><argument_list>(<argument><expr><literal type=”string”>string</
          literal></expr></argument>, <argument><expr><call><name><name>token</name><
          operator>.</operator><name>getTokenSecret</name></name><argument_list>()</
          argument_list></call></expr></argument>)</argument_list></call></expr>;</expr_stmt>
   }</block></then></if>
}</block>
```

Figure A.7: AST of the file presented in Figure A.5, in the srcML format, extracted using the srcML tool.

```
{block{decl_stmt{decl{type{name}}{name}}}{decl_stmt{decl{type{name}}{name}}}{
    expr_stmt{expr{call{name{name}{operator}{name}}{argument_list{argument{
    expr{literal}}}{argument{expr{call{name{name}{operator}{name}}{
    argument_list}}}}}}}}{expr_stmt{expr{call{name{name}{operator}{name}}{
    argument_list{argument{expr{literal}}}{argument{expr{call{name{name}{
    operator}{name}}{argument_list}}}}}}}}}
```

Figure A.8: Transaction in the form used by the APTED tool, for the file presented in Figure A.6.

```
{block{decl_stmt{decl{type{name}}{name}}}{decl_stmt{decl{type{name}}{name}}}{
    if{condition{expr{name}{operator}{literal}}}{then{block{expr_stmt{expr{
    call{name{name}{operator}{name}}{argument_list{argument{expr{literal}}}{
    argument{expr{call{name{name}{operator}{name}}{argument_list}}}}}}}}{
    expr_stmt{expr{call{name{name}{operator}{name}}{argument_list{argument{
    expr{literal}}}{argument{expr{call{name{name}{operator}{name}}{
    argument_list}}}}}}}}}}}}
```

Figure A.9: Transaction in the form used by the APTED tool, for the file presented in Figure A.7.

A.3 `Artistic Style` Example

In order to increase the readability of our system's generated snippets, we make use of the *Artistic Style* formatter. This is a commonly used formatter, that supports several languages, including the Java language. We set its `style` option to the `java` one, which is used for Java source code files. The way this tool works is straightforward, and thus we only provide an example in Figures A.10 and A.11, without any further analysis.

```
{
      List<SavedSearch> searches;
      Twitter twitter;
      try {
         searches = twitter.getSavedSearches();
            // Do something with searches
      } catch (TwitterException e) {
         e.printStackTrace();}}
```

Figure A.10: Snippet before the application of the Artistic Style formatter.

```
{
   List<SavedSearch> searches;
   Twitter twitter;
   try {
      searches = twitter.getSavedSearches();
      // Do something with searches
   } catch (TwitterException e) {
      e.printStackTrace();
   }
}
```

Figure A.11: The snippet presented in Figure A.10, after the application of the Artistic Style formatter to it.

Appendix B

Summarisation Specifics

B.1 Summarisation Pseudocodes

One of the basic steps of the summarisation algorithm is the one where the statements are classified to API and to non-API ones. In order to decide on whether a statement belongs to the first ones or to the latter ones, the algorithm needs to be fed with the name of the API methods, that are called in the snippet to be summarised. As we see in Algorithm 7, the `ClassifyStmts` function then takes the tree of the source code to be summarised, as well as these names, as its input parameters, in order to classify the statements. The process followed is analysed below:

- In *Lines 2* and *3* the function initialises the two lists that are going to store the API and the non-API statements, respectively.
- In *Line 4* the function iterates over the tree in document order (also called *pre-order traversal*[1]).
- *Line 5* checks whether a node[2] is the root of a *statement*, as these are defined by the srcML format (e.g. `decl_stmt`, `return`, etc.).
- In *Lines 6* and *7* the function checks whether the nodes that satisfy the condition in *Line 5* contain an API call in their subtree. Based on the latter decision, they are appended to the appropriate list.
- In *Line 11* it checks whether a node is the root of a *condition*, as these are once more defined by the srcML format (e.g. `condition`, `control`, etc.).

[1] `https://en.wikipedia.org/wiki/Tree_traversal`

[2] In this section, we use the term "node" to indicate element nodes in the xml file.

- *Line 12* checks whether the parent node of a node that satisfies the condition in *Line 11*, contains an API call. If so, the node is added to the list where the API statements are stored. Justifying this part of the algorithm, we firstly point out that each node that is a root of a condition statement (e.g. `condition`, `control`, etc.), has a parent that is a control[3] statement (e.g. `if`, `for`, etc.), in the srcML format. In addition to that, if -the subtree of- a control statement contains an API call, then we should also keep -the subtree of- its condition(s)[4] in the summarised code. Based on this, we may classify the condition statements using their associated control statements classification information. Moreover, our decision to exclude the `else` statement of the condition in *Line 12* stems from the fact that there is no need to add a condition statement into the corresponding list, in case it is a non-API statement, as the first is going to be removed, too, in case its associated control block is removed.

- Finally, *Lines 15-17* check whether -the subtree of- a control statement contains an API call, and if not, this is added to the appropriate list. Note that there is no need to add a control statement into the corresponding list, in case it contains an API call, as all the API statements of its subtree are going to be added to the list, as either *statement* or *condition* nodes.

[3]To avoid confusion, here we are talking about decision-making and looping statements (see `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html`), rather than about the `control` element, used in the srcML format.

[4]A hidden point here is that the srcML format handles the `case` statements of a `switch` statement in a similar manner to `conditions`, and hence we consider them as conditions, too.

Algorithm 7 Statements Classifier

```
1: procedure ClassifyStmts($tree$, $API\_calls$)
2:     $API\_stmts \leftarrow 0$
3:     $non\_API\_stmts \leftarrow 0$
4:     for all $node \in tree$ do                    ▷ Iterate over the $tree$ in document order
5:         if $node$ is a $statement$ then
6:             if $node.subtree$ contains an $API\_call \in API\_calls$ then
7:                 $API\_stmts \leftarrow API\_stmts \cup node$
8:             else
9:                 $non\_API\_stmts \leftarrow non\_API\_stmts \cup node$
10:            end if
11:        else if $node$ is a $condition$ then
12:            if $node.parent.subtree$ contains an $API\_call \in API\_calls$ then
13:                $API\_stmts \leftarrow API\_stmts \cup node$
14:            end if
15:        else if $node$ is a $control$ then
16:            if $node.subtree$ does not contain any $API\_call \in API\_calls$ then
17:                $non\_API\_stmts \leftarrow non\_API\_stmts \cup node$
18:            end if
19:        end if
20:    end for
21:    return $API\_stmts$, $non\_API\_stmts$
22: end procedure
```

B.2 Summarised Snippets

In this section we are going to show the effect of the summarisation algorithm to the Java snippets it is applied to. Indicative examples are shown for features including the variables type resolution, the literals replacement, as well as the addition of descriptive comments, which are added either in case of empty blocks, or in snippets where variables that are declared in API statements are used in non-API statements (which are removed from the summarised snippet). We also provide an example which mainly shows the removal of any non-API statements.

A. Variables Type Resolution

The summarisation algorithm is able to resolve the type of -almost[5]- any variable used in the snippet, in case this information is available in the original source code file. An example of this feature is shown in Figures B.1 and B.2.

```
import it.unibz.dao.TwitterDao;
import twitter4j.Twitter;
import twitter4j.TwitterException;
import twitter4j.auth.AccessToken;
import twitter4j.auth.RequestToken;

public class TwitterServiceImpl implements TwitterService {
  private AccessToken accessToken = null;
  private TwitterDao twitterDao;

   public void login(Twitter t, String parameter, RequestToken rt, long userId
       ) throws TwitterException {
      AccessToken a = t.getOAuthAccessToken(rt, parameter);
      twitterDao.storeAccesstoken(a.getToken(), a.getTokenSecret(), userId);
   }
}
```

Figure B.1: Java snippet with unresolved variables.

```
{
   TwitterDao twitterDao;
   String parameter;
   Twitter t;
   long userId;
   RequestToken rt;
   AccessToken a = t.getOAuthAccessToken(rt, parameter);
   twitterDao.storeAccesstoken(a.getToken(), a.getTokenSecret(), userId);
}
```

Figure B.2: Summarised snippet of the source code presented in Figure B.1, after the type resolution.

[5]A limitation of our approach when resolving variables type is that we do not take into account the scope of a variable. Although this may lead to an incorrect type resolution, this occurs in rare cases only.

B. Literals Replacement

One of the features of the summarisation algorithm is that it replaces the literals, as these are defined in the srcML format, with their types. This includes numbers, strings or even booleans. For instance, the snippet presented in Figure B.3 is summarised to the one shown in Figure B.4.

```java
q.setQuery(queryParams[0]);
if (lang.equals("") == false) {
  q.setLang(lang);
}
```

Figure B.3: Java snippet, before the literals replacement.

```java
q.setQuery(queryParams[number]);
if (lang.equals(string) == boolean) {
  q.setLang(lang);
}
```

Figure B.4: Summarised snippet of the source code in Figure B.3, after the replacement of literals with their srcML type.

C. Addition of Descriptive Comments inside Empty Blocks

A feature of the summarisation algorithm that aims to improve the readability of the mined snippets, is the addition of descriptive comments, in case of empty blocks. The idea behind that is analysed in [48], while an example that shows the effect of this feature is presented in Figures B.5 and B.6.

```java
try {
} catch (TwitterException e) {
  e.printStackTrace();
}
```

Figure B.5: Java snippet with an empty block, before adding a descriptive comment.

```java
try {
  // Do something
} catch (TwitterException e) {
  e.printStackTrace();
}
```

Figure B.6: Summarised snippet of the source code presented in Figure B.5, after the addition of a descriptive comment inside the empty block.

D. Addition of Descriptive Comments for Variables Declared in API Statements that are Used Only in non-API Statements

This is a feature that adds a novelty to our summarisation algorithm. It is based on a similar feature introduced by Buse and Weimer in [34]. However, in our case, we make use of the API statements, as well as of the non-API statements of the source code to be summarised. That is, we check whether an API statement declares a variable that is then used only in non-API statements (which are removed in the summarised version). If so, we add a descriptive comment for the declared variable. The effect of this feature is shown in Figures B.7 and B.8.

```
if (twe.getCreatedAt().getTime() + 1000 < lastMaxCreateTime) {
  breakPaging = true;
}
else {
  String userName = twe.getFromUser().toLowerCase();
  JUser user = userMap.get(userName);
  if (user == null) {
    user = new JUser(userName).init(twe);
    userMap.put(userName, user);
  }
}
```

Figure B.7: Java snippet without descriptive comments. The API method calls are highlighted.

```
if (twe.getCreatedAt().getTime() + number < lastMaxCreateTime) {
  // Do something
} else {
  String userName = twe.getFromUser().toLowerCase();
  // Do something with userName
}
```

Figure B.8: Summarised snippet of the source code presented in Figure B.7, after the addition of descriptive comments. The API method calls are highlighted.

An indicative example that reveals the efficiency of the summarisation algorithm in case of large snippets is presented in Figures B.9 and B.10.

```
private void showUserView(ViewHolder holder, User currentEntity) {
      /* Get the entity text. (If the user is protected, the status may be
       * null, so account for that) */
      String text = "";
      if (currentEntity.getStatus() == null) {
         // TODO: add to strings.xml
         text = "You need to follow this user to see their status.";
      } else {
         text = ((User)currentEntity).getStatus().getText();
      }
      holder.text_tweet.setText(text);
      holder.text_tweet.setTypeface(mTypeface);
      Linkify.addLinks(holder.text_tweet, Linkify.ALL);
      Linkify.addLinks(holder.text_tweet, screenNameMatcher,
             Constants.SCREEN_NAME_URI.toString() + "/");

      /* Set the tweet time Textview */
      Date createdAt = new Date();
      if (currentEntity.getStatus() != null) {
         createdAt = ((User)currentEntity).getStatus()
             .getCreatedAt();
      }
      holder.text_time.setText(new PrettyDate(createdAt).toString());
      holder.text_time.setTypeface(mTypeface);

      /* Set the screen name TextView */
      String screenName = currentEntity.getScreenName();
      holder.text_screenname.setText("@"+screenName);
      holder.text_screenname.setTypeface(mTypeface);

      /* Set the profile image ImageView */
      imageLoader.displayImage(screenName, holder.image_profile);
   }
```

Figure B.9: Client code without the summariser. The API calls are highlighted.

```
{
   User currentEntity;
   String text;
   Date createdAt;
   if (currentEntity.getStatus() == null) {
      // Do something
   } else {
      text = ((User)currentEntity).getStatus().getText();
   }
  // Do something with text
   if (currentEntity.getStatus() != null) {
      createdAt = ((User)currentEntity).getStatus()
      .getCreatedAt();
   }
  // Do something with createdAt
   String screenName = currentEntity.getScreenName();
   // Do something with screenName
}
```

Figure B.10: Summarised version of the client code presented in Figure B.9. The API calls are highlighted.

Appendix C

Evaluation Specifics

C.1 Computing the Snippet-Based Metrics

The process followed in order to compute the snippet-based metrics, is analysed below:

1. For each source code file (mined snippets and examples), we extract a sequence of tokens, using a Java code tokeniser, that has been implemented as a previous work of our team. The Java Tokenizer makes use of the Eclipse JDT, in order to extract a list of tokens from a serialised version of a Java source code. We have proceeded to a few modifications on the existing version of the Tokenizer, in order to be able to tokenise multiple files, and to write the output to a `.json` file. An example that shows the tokens extracted from a Java source code is presented in Figures C.1 and C.2.

2. Having extracted a `.json` file where the tokens of the mined snippets/examples are stored, we proceed to some basic preprocessing; this includes the removal of any symbols (e.g. brackets, semicolons, etc.), as well as of any comments and Javadocs. Moreover, we replace any literals (e.g. Strings, numbers, etc.) with their srcML type, in order to be able to fairly evaluate the summarisation algorithm, which includes this step. The revised list of tokens for the example presented in Figure C.1 is shown in Figure C.3.

3. For each snippet, we check whether an example contains the same API calls (it may contain additional ones, but this is not a problem). If so, we use the snippet-based metrics defined in Section 6.5. In case multiple examples match to the snippet, with respect to its API calls, we match the snippet to the one that maximises the number of their common tokens.

```
{
   int page;
   long sinceId;
   int count;
   Paging paging = new Paging(page, count);
   if (sinceId > 0) {
      paging.setSinceId(sinceId);
   }
}
```

Figure C.1: Java snippet to be tokenised by the Java Tokenizer.

```
["{","int","page",";","long","sinceId",";","int","count",";","Paging","paging"
    ,"=","new","Paging","(","page",",","count",")",";","if","(","sinceId","=",
    "0",")","{","paging",".","setSinceId","(","sinceId",")",";","}","}"]
```

Figure C.2: Java tokens extracted from the Java source code presented in Figure C.2, using the Java Tokenizer.

```
["int","page","long","sinceId","int","count","Paging","paging","new","Paging",
    "page","count","if","sinceId","number","paging","setSinceId","sinceId"]
```

Figure C.3: Revised list of Java tokens for the Java source code presented in Figure C.2, after the preprocessing step.

C.2 Experiment 1

```
{
  super.onCreate();
  oAuthHelper = new OAuthHelper(this);
  twitter = new TwitterFactory().getInstance();
  oAuthHelper.configureOAuth(twitter);
}
```

Figure C.4: Snippet ranked first by the *RemUniqNaivNoSum* version of the system.

```
{
   Twitter twitter;
   twitter = new TwitterFactory().getInstance();
   // Do something with twitter
}
```

Figure C.5: Snippet ranked first by the *RemUniqNaivSum* version of the system. This snippet is a summary of the one presented in Figure C.4.

```
{
   Twitter twitter = new TwitterFactory().getInstance();

   // Get the key info from system properties
   String key = System.getProperty("twitter-consumer-key");
   String secret = System.getProperty("twitter-consumer-secret");

   try
   {
      twitter.setOAuthConsumer(key, secret);

      //Make a request token to access twitter service to request for user
          details
      RequestToken token = twitter.getOAuthRequestToken();

      //Make a session attribute(Some servlet background will help here!)
      HttpSession session = request.getSession();

      //we need these in callback servlet
      session.setAttribute("requestToken", token);

      //Thwe twitter login url
      String loginURL = token.getAuthenticationURL();

      //Redirect to twitter
      response.sendRedirect(loginURL);
   } catch (TwitterException e) {
      //TODO: log4j
   }
}
```

Figure C.6: Random snippet mined by the *RemUniqNaivNoSum* version of the system.

```
{
   String key;
   String secret;
   Twitter twitter = new TwitterFactory().getInstance();

   try
   {
      twitter.setOAuthConsumer(key, secret);
      RequestToken token = twitter.getOAuthRequestToken();
      String loginURL = token.getAuthenticationURL();
      // Do something with loginURL
   } catch (TwitterException e) {
      // Do something
   }
}
```

Figure C.7: Random snippet mined by the *RemUniqNaivSum* version of the system. This snippet is a summary of the one presented in Figure C.6.

C.3 Experiment 2

```
{
   Twitter twitter;
   twitter = new TwitterFactory().getInstance();
   // Do something with twitter
}


{
   Twitter mTwitter;
   final String CONSUMER_KEY;
   final String CONSUMER_SECRET;
   mTwitter = new TwitterFactory().getInstance();
   mTwitter.setOAuthConsumer(CONSUMER_KEY, CONSUMER_SECRET);
}


{
   BasicDBObject tweet;
   Status status;
   tweet.put(string, status.getUser().getScreenName());
   tweet.put(string, status.getText());
}


{
   String mConsumerKey;
   Twitter mTwitter;
   AccessToken mAccessToken;
   String mSecretKey;
   if (mAccessToken != null) {
      mTwitter.setOAuthConsumer(mConsumerKey, mSecretKey);
      mTwitter.setOAuthAccessToken(mAccessToken);
   }
}


{
   Twitter mTwitter;
   String token;
   String secret;
   AccessToken at = new AccessToken(token, secret);
   mTwitter.setOAuthAccessToken(at);
}
```

Figure C.8: The top 5 snippets, with respect to their support, mined by the *RemUniqNaivSum* version of the system.

```
{
   Twitter twitter;
   twitter = new TwitterFactory().getInstance();
   // Do something with twitter
}


{
   final TwitterFactory factory;
   GlobalSetting gs;
   Twitter twitter = factory.getInstance();
   twitter.setOAuthConsumer(gs.getTwitterConsumerKey(),gs.
       getTwitterConsumerSecret());
}


{
   Twitter mTwitter;
   final String CONSUMER_KEY;
   final String CONSUMER_SECRET;
   mTwitter = new TwitterFactory().getInstance();
   mTwitter.setOAuthConsumer(CONSUMER_KEY, CONSUMER_SECRET);
}


{


   BasicDBObject tweet;
   Status status;
   tweet.put(string, status.getUser().getScreenName());
   tweet.put(string, status.getText());
}


{
   String mConsumerKey;
   Twitter mTwitter;
   AccessToken mAccessToken;
   String mSecretKey;
   if (mAccessToken != null) {
      mTwitter.setOAuthConsumer(mConsumerKey, mSecretKey);
      mTwitter.setOAuthAccessToken(mAccessToken);
   }
}
```

Figure C.9: The top 5 snippets, with respect to their support, mined by the *KeepUniqNaivSum* version of the system.

C.4 Experiment 3

```
TwitterFactory.<init>
TwitterFactory.getInstance

TwitterFactory.<init>
TwitterFactory.getInstance
Twitter.setOAuthConsumer

Status.getUser
Status.getText

TwitterFactory.<init>
TwitterFactory.getInstance
Twitter.setOAuthConsumer
Twitter.setOAuthAccessToken

auth.AccessToken.getToken
auth.AccessToken.getTokenSecret

TwitterStreamFactory.<init>
TwitterStreamFactory.getInstance

Paging.<init>
Status.getId

http.AccessToken.getToken
http.AccessToken.getTokenSecret

Twitter.getOAuthAccessToken
TwitterException.printStackTrace

Twitter.updateStatus
TwitterException.printStackTrace
```

(a)

```
TwitterFactory.<init>
TwitterFactory.getInstance

TwitterFactory.<init>
TwitterFactory.getInstance
Twitter.setOAuthConsumer

Status.getUser
Status.getText

Twitter.setOAuthConsumer
Twitter.setOAuthAccessToken

auth.AccessToken.<init>
Twitter.setOAuthAccessToken

TwitterFactory.<init>
TwitterFactory.getInstance
Twitter.setOAuthConsumer
Twitter.setOAuthAccessToken

auth.AccessToken.getToken
auth.AccessToken.getTokenSecret

TwitterStreamFactory.<init>
TwitterStreamFactory.getInstance

ConfigurationBuilder.<init>
ConfigurationBuilder.setOAuthConsumerKey
TwitterFactory.<init>
TwitterFactory.getInstance

User.getId
User.getScreenName
```

(b)

Figure C.10: The top 10 sequences, with respect to their support, mined (a) by the *KeepUniqKMedoidsSum*, and (b) by the *KeepUniqHDBSCANSum* version of the system, that leverage the k-medoids and the HDBSCAN clustering techniques, respectively.

```
{
   Twitter twitter;
   twitter = new TwitterFactory().getInstance();
   // Do something with twitter
}

{
   Twitter mTwitter;
   final String CONSUMER_KEY;
   final String CONSUMER_SECRET;
   mTwitter = new TwitterFactory().getInstance();
   mTwitter.setOAuthConsumer(CONSUMER_KEY, CONSUMER_SECRET);
}

{
   BasicDBObject tweet;
   Status status;
   tweet.put(string, status.getUser().getScreenName());
   tweet.put(string, status.getText());
}

{
   String mConsumerKey;
   Twitter mTwitter;
   AccessToken mAccessToken;
   String mSecretKey;
   if (mAccessToken != null) {
      mTwitter.setOAuthConsumer(mConsumerKey, mSecretKey);
      mTwitter.setOAuthAccessToken(mAccessToken);
   }
}

{
   Twitter mTwitter;
   String token;
   String secret;
   AccessToken at = new AccessToken(token, secret);
   mTwitter.setOAuthAccessToken(at);
}
```

Figure C.11: The top 5 snippets, with respect to their support, mined by the *KeepUniqHDBSCANSum* version of the system.

```
{
   Twitter twitter;
   twitter = new TwitterFactory().getInstance();
   // Do something with twitter
}

{
   Twitter mTwitter;
   final String CONSUMER_KEY;
   final String CONSUMER_SECRET;
   mTwitter = new TwitterFactory().getInstance();
   mTwitter.setOAuthConsumer(CONSUMER_KEY, CONSUMER_SECRET);
}

{

   BasicDBObject tweet;
   Status status;
   tweet.put(string, status.getUser().getScreenName());
   tweet.put(string, status.getText());
}

{
   String CONSUMER_KEY;
   AccessToken a;
   String CONSUMER_SECRET;
   Twitter twitter;
   twitter = new TwitterFactory().getInstance();
   twitter.setOAuthConsumer(CONSUMER_KEY, CONSUMER_SECRET);
   twitter.setOAuthAccessToken(a);
}

{
   Editor e;
   AccessToken mAt;
   e.putString(string, mAt.getToken());
   e.putString(string, mAt.getTokenSecret());
}
```

Figure C.12: The top 5 snippets, with respect to their support, mined by the *KeepUniqKMedoidsSum* version of the system.

C.5 Experiment 5

```
{
   AccessToken accessToken;
   String oauthToken;
   String oAuthVerifier;
   Twitter twitter;
   try {
      accessToken = twitter.getOAuthAccessToken(oauthToken,oAuthVerifier);
      // Do something with accessToken

   } catch (TwitterException e) {
      e.printStackTrace();
   }
}
```

Figure C.13: A snippet mined by the *KeepUniqHDBSCANSum* version of the system, that has been matched to a handwritten example. The common tokens between the snippet and the handwritten examples it matches to are highlighted; the sequence-tokens are only coloured, while the additional snippet-tokens are moreover encircled, and show the additional information revealed to the developers when presenting snippets instead of sequences.

```
{
   Twitter mTwitter;
   final String CONSUMER_KEY;
   final String CONSUMER_SECRET;
   mTwitter = new TwitterFactory().getInstance();
   mTwitter.setOAuthConsumer(CONSUMER_KEY, CONSUMER_SECRET);
}
```

Figure C.14: A snippet mined and placed in the second position by the *KeepUniqHDBSCANSum* version of the system. It has not been matched to any handwritten example, although it is supported by 70 source code files in the mined dataset. In fact, there is no handwritten example that covers the `setOauthConsumer` method of the `Twitter4J` API, which is considered as one of the most popular methods of the API. This shows that our system may be well used in order to augment the documentation of an API with new examples.

Bibliography

[1] Martin P. Robillard. What makes apis hard to learn? answers from developers. *Software, IEEE*, 26(6):27–34, 2009.

[2] Gias Uddin and Martin P. Robillard. How api documentation fails. *Software, IEEE*, 32(4):68–75, 2015.

[3] Raphael Hoffmann, James Fogarty, and Daniel S Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22. ACM, 2007.

[4] Diomidis Spinellis. *Code quality: the open source perspective*. Adobe Press, 2006.

[5] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining at github scale. *arXiv preprint arXiv:1512.05558*, 2015.

[6] Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.

[7] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007.

[8] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 296–305. ACM, 2005.

[9] Shahida Khatoon, Arif Mahmood, and Guohui Li. An evaluation of source code mining techniques. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, volume 3, pages 1929–1933. IEEE, 2011.

[10] MUSA IBRAHIM M ISHAG, HYUN WOO PARK, DINGKUN LI, and KEUN HO RYU. Highlighting current issues in api usage mining to enhance software reusability. 2016.

[11] Anand Rajaraman, Jeffrey D Ullman, Jeffrey David Ullman, and Jeffrey David Ullman. *Mining of massive datasets*, volume 1. Cambridge University Press Cambridge, 2012.

[12] Pang-Ning Tan, Steinbach Michael, and Vipin Kumar. Chapter 6. association analysis: Basic concepts and algorithms. *Introduction to Data Mining. Addison-Wesley. ISBN*, 321321367, 2005.

[13] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[14] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM Sigmod Record*, volume 29, pages 1–12. ACM, 2000.

[15] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[16] Jaroslav Fowkes and Charles Sutton. A bayesian network model for interesting itemsets. *arXiv preprint arXiv:1510.04130*, 2015.

[17] Mohammed J Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1-2):31–60, 2001.

[18] Jianyong Wang and Jiawei Han. Bide: Efficient mining of frequent closed sequences. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 79–90. IEEE, 2004.

[19] Jaroslav Fowkes and Charles Sutton. A subsequence interleaving model for sequential pattern mining. *arXiv preprint arXiv:1602.05012*, 2016.

[20] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724. IEEE, 2002.

[21] Jun Huan, Wei Wang, Jan Prins, and Jiong Yang. Spin: mining maximal frequent subgraphs from graph databases. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 581–586. ACM, 2004.

[22] Yun Chi, Richard R Muntz, Siegfried Nijssen, and Joost N Kok. Frequent subtree mining–an overview. *Fundamenta Informaticae*, 66(1-2):161–198, 2005.

[23] ABE Kenji, Shinji Kawasoe, Hiroshi Sakamoto, Hiroki Arimura, and Setsuo Arikawa. Efficient substructure discovery from large semi-structured data. *IEICE TRANSACTIONS on Information and Systems*, 87(12):2754–2763, 2004.

[24] Yun Chi, Yirong Yang, Yi Xia, and Richard R Muntz. Cmtreeminer: Mining both closed and maximal frequent subtrees. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 63–73. Springer, 2004.

[25] Tan Pang-Ning, Michael Steinbach, Vipin Kumar, et al. Introduction to data mining. In *Library of congress*, volume 74, 2006.

[26] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[27] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[28] Ramiz M Aliguliyev. Clustering techniques and discrete particle swarm optimization algorithm for multi-document summarization. *Computational Intelligence*, 26(4):420–448, 2010.

[29] Dingding Wang, Tao Li, Shenghuo Zhu, and Chris Ding. Multi-document summarization via sentence-level semantic analysis and symmetric matrix factorization. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 307–314. ACM, 2008.

[30] Endre Boros, Paul B Kantor, and David J Neu. A clustering based approach to creating multi-document summaries. In *Proceedings of the 24th annual international ACM SIGIR conference on research and development in information retrieval*, 2001.

[31] Yan Liu, Sheng-hua Zhong, and Wenjie Li. Query-oriented multi-document summarization via unsupervised deep learning. In *AAAI*, 2012.

[32] João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. Documenting apis with examples: Lessons learned with the apiminer platform. In *WCRE*, pages 401–408. Citeseer, 2013.

[33] Jinhan Kim, Sanghoon Lee, Seung-Won Hwang, and Sunghun Kim. Enriching documents with examples: A corpus mining approach. *ACM Transactions on Information Systems (TOIS)*, 31(1):1, 2013.

[34] Raymond PL Buse and Westley Weimer. Synthesizing api usage examples. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 782–792. IEEE, 2012.

[35] Nedunchelian Ramanujam and Manivannan Kaliappan. An automatic multidocument text summarization approach based on naïve bayesian classifier using timestamp strategy. *The Scientific World Journal*, 2016, 2016.

[36] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 25–34. IEEE, 2012.

[37] Annie TT Ying and Martin P Robillard. Selection and presentation practices for code example summarization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 460–471. ACM, 2014.

[38] Annie TT Ying and Martin P Robillard. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 655–658. ACM, 2013.

[39] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[40] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.

[41] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.

[42] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 429–435. ACM, 2002.

[43] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 319–328. IEEE Press, 2013.

[44] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Adding examples into java documents. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 540–544. IEEE, 2009.

[45] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Towards an intelligent code search engine. In *AAAI*, 2010.

[46] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[47] Lijie Wang, Lu Fang, Leye Wang, Ge Li, Bing Xie, and Fuqing Yang. Apiexample: An effective web search based usage example recommendation system for java apis. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 592–595. IEEE Computer Society, 2011.

[48] Hudson Silva Borges. Extracting examples for API usage patterns. Master's thesis, Federal University of Minas Gerais, Brazil, 2014.

[49] Kirsten Winter, Chenyi Zhang, Ian J Hayes, Nathan Keynes, Cristina Cifuentes, and Lian Li. Path-sensitive data flow analysis simplified. In *International Conference on Formal Engineering Methods*, pages 415–430. Springer, 2013.

[50] Miltiadis Allamanis and Charles Sutton. Mining Source Code Repositories at Massive Scale using Language Modeling. In *The 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE, 2013.

[51] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[52] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 160–172. Springer, 2013.

[53] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[54] Christian Bauckhage. Numpy/scipy recipes for data science: k-medoids clustering. Technical report, Technical Report, University of Bonn, 2015.

[55] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433, 1979.

[56] Michal Young and Mauro Pezze. Software testing and analysis: Process, principles and techniques. 2005.

[57] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. Evaluation in information retrieval. *Introduction to information retrieval*, pages 151–175, 2008.

[58] Stevche Radevski, Hideaki Hata, and Kenichi Matsumoto. Towards building api usage example metrics. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 619–623. IEEE, 2016.

[59] Raymond PL Buse and Westley R Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130. ACM, 2008.