Support to development of complex software systems with API function call recommendations

Contents

1 Introduction

In recent years, the problem of API recommendations rise more and more in relevance in the complex software systems. When a developer wirtes some code to implement a specific features, he should want some suggestions about useful functions in this context. An API (Application programming interface) is a set of procedures, protocols and objects that gives to the developers the necessary building blocks to implement a specific functionality in easy and understandable way. The API depends on the language that the developer use and it usually composed by libraries with objects and methods useful for the task. The definition is very general and it may change based on the context of the developed project; so, the developer may get confusing about what kind of method or class to use or how to use them in a proper way.
The issues is how to perform a good enough recommendation in this context, balancing possible bias and put the proper hints for the developer. Moreover, the form of the recommendation is also important because in general there are variety of possible suggestion such as code snippet, patterns for the methods, ehance documentation and all things that make a recommendation really usable for the current project.

2 Related Works

Considering the state of the art, there are several approaches that face the problem that I have described in previous section. The key point is to find, on one side, a simple and effective way to extract API functions call and build the recommendation system and on other hand, is necessary to use formal and proper algorithm and techniques that produce very suitable results. In the literature, there are main approaches is first to analyze the code that you are interested in, adopting some similarity metric, optionally followed by clustering techinques and then how to represent the obtained results in form of recommendation that should be clear and effective. In next pages I present a set of approaches that follow the same main skeleton to solve the problem; what is different among them is the techinques, algorithms and produced results but they are all useful and give the flavour of the problem that I'm involved in. Table 1 gives an initial overview on the approaches, with the proposing tool, how they analyze the source code, the similarity metric and algorithm, the eventually clustering technique, the datased considered to validate the results and the given output. In nexts section, I go in deep on all these approaches by describing the overall frameworks and the main idea behind the approaches to have a well defined state of the art before prociding with my approach.

Table 1: Summary of relevant paper

Proposed tool	**Parser for code snippets**	**Similarity**	**Clustering**	**Dataset**	**Output**
MAPO	JDT	Class name, Method name API call sequences	Data-driven	Java projects	Eclipse plugin that shows possible API pattern
UP-Miner	Roselyn AST parser	SeqSim technique similar sequences	Two clustering on frequent sequences with BIDE	C#	Probabilistic graph of API that can be queried
CLAMS	JDT parser	Distance matrix	LCS,HDBSCAN techniques AP-TED for top rank snippet	6 popular Java libraries	Patterns for API
APIRec	GumTree AST parser	association-based model with fine-grained code changes	association-based change inference model	50 Java project randomly from Gtihub	Most frequent API call
Buse-Weimer algorithm	Symbolic execution for path enumeration	Distance matrix	K-medoids algorithm	Java SDK	Human readable documentation
APIMiner extension	see APIMiner	Structural similarity among API	FP-growth with WEKA tool	Android projects	Enhance documentation
JSS approach	No info	Object usages social network with co-existence relation	Co-existence relation with Modularity index method call similarity with Gamma index	Android project	API usage pattern
JIRA extension	JDT	Cosine similarity history and descriptor recommender	Integrator component based on previous weights	Apache projects	Top ranked methods
CodeBroker	Back-end search engine	Latent semantic analysis	Discourse model user model	Java core libraires	Three layerd infomation relevant tasks, signature and JavaDoc

2.1 MAPO

In [1] the authors propose MAPO that perform methods extraction using data mining techniques, clustering them to have a more representative data and build a recommender through GEF tool. So, they divided the process into three main steps: analizying the source code, the API miner and API recommender. First of all, MAPO parse the source code coming from Java Github projects that use Eclipse Graphical Editor Framework (GEF), using JDT parser utilities that allow to analyze in deep a Java file and extract classes, interfaces, method invocations and method declarations. For the recommendations, MAPO considers as API method suitable for the recommender only those belong to third-parts libraries, considering the GEF framework as external, class cast and creation associated to external class and the method call that belongs to external class; basically, the authors ignore the libraries and so the internal API of the JDK. Then MAPO collects all source code by using @ as intial mark to identify a method call and # as separator between method and related class. About the conditional statements, such as if, while and so on, the authors not considered the possible relations among multiple conditions and in practice the represent them as flat code in which MAPO considers all braches. As claim by authors, this semplification is necessary otherwise the mining phase is infeasible and the API recommender is not effected by them. Once the method calls are selected, there is the problem of method overwheigth and common-path overweigth that involves respectvely several method call and sequence in common in the source code that can introduce bias in the result; for example, in fragment of code MAPO can select a method call only beacuse it is replicated and not for its real effectiveness. To avoid this problem, the authors select the longest sequence in common that covers also the smaller sequence of method call and, in this way, the reduce the bias. Moreover, MAPO selects only third-parties libraries using inline methodology, that consist to explore the parse tree of classes and idetify the anchestor and so excluding the JDK methods.
Once MAPO have all method call, the next step is mining the sequence to produce recommendation. To do this, is not enough to consider all method sequences because this can be bring incorrect results. So, MAPO using first similarity metric, based on the name and the usage of methods to indetify similar sequence and then clusterize them by using data-driven hierarchical clustering . In particular, the authors consider three level of similatity given by class names, method names and called API method and obtain in this way the similar sequence. Then, by using mathlab tool, MAPO using classical hierarchical clustering to obtain the ranked list of representative sequences, transform them into transaction and put them into a database. Finally, the API recommender is a Eclipse plugin that allow developer to click on the method of interest , excute a query on DB to show possible uses by using sample code and developer can also see details on the proper window tab in which the API method are highligtht. To validate this approach, the authors use a dataset composed by 20 projects that used Eclipse GEF, run the tool and show the effectiveness of their approach by a quantitive comparison. To reinforce the validity, they also conduct an empirical study on by considerig a set of tasks to do and select a group of Java developers that is invoved to solve those tasks.

2.2 UP-Miner

A similar approach is performed by [**?**] in which the authors create UP-Miner tool that improve MAPO in term of accuracy. Going in deep, the aim of the authors is to achive the succitness but also the effectiveness of mined pattern that represent an enhancement of previous approach. Once the authors define API usage mining, that is the optimal number of patterns under a given threshold, they propose a clustering techinques based on BIDE algorithm. As first step, UP-Miner exteracts API pattern sequences using Roslyn AST parser from the projects that compose the dataset. Then, the apply the SeqSim n-gram technique that takes two sequence computes

the simitarity that is based on the shared items between them in term of objects and classes used and on the longer and consecutive subsequences rather than the shorter ones. This phase produce a weighted results that are used for clustering that are conservative on, in sense that the maximum distance between two cluster is the maximum distance between two elemets of those cluster.
Then, UP-Minser use BIDE algorithm, that have the key concept of frequent sequence: a general sequence becames frequent if its supersequences (namely the sequences that contains the considered sequence) is greater or less of the given threshold. Consider this, BIDE algorithm can extract the longest common sequence that are useful to divided the results into different clusters, but it is not sufficient because at this point there may be some redundant cluster. So, it is necessary to apply once again the previous phase considering the cluster as usage pattern and this two-step clustering grants an improvement in term of redundancy considering also two different threshold (one for pre-BIDE and another for post-BIDE application). Until this, UP-Miner adress the problem of coverage but the aim of the authors is to reach also the succitness of patterns. To do this, UP-Miner use a dissimilarity metric that measure the diversity of a usage pattern from another and an utility functions to maximize in order to obtain the better results, decreasing the threshold at each step of the algorithm implemented for this task. Once UP-Miner computes the correct and more succitness as possible usage API pattern, the authors show them to the developer by building a probabilistic graph in which each node is a usage pattern and the edge is weighted with certain probability. To produce this prototype and make some experiments that involves the developers in a similar way as we see in MAPO, the authors use a very large C# dataset as input files and compare their results with MAPO ones.

2.3 CLAMS

Go further with the examination of previous, we have also CLAMS tool proposed in [3] that follows a quite similar approach but it differs from the point of view of the results. Infact, CLAMS produce as API recommendations snippet of code that represent a pattern for a certain library. As usual, the preprocessing phase is done by analyze the AST of the source code (in case of CLAMS, projects related to 5 popular Java libraries) with a dept-first search using JDT as previous example. This phase produces snippet of code that bring all information about API implemented in the code. The similarity technique is based on Longest Common Subsequence (LCS) and is more effective rather than an analysis at source code level. By using this technique, CLAMS creates a distance matrix that is used as input by the clustering module of CLAMS, that implements the hierachical version of DBSCAN algorithm, called HDBSCAN, plus a post-clustering processing to eliminate the sequences that are identical to snippets for each cluster obtained. The aim of HDBSCAN is to isolate the less representative methods and, more in general, points into a distribution that are really far from the rest of the dataset. To do this, the algorithm uses a distance called core distance to draw a circle on the points to exlude and then computes the mutual reachability distance that reduce the presence of sparse node in the dataset. Then, by appling Prim's algorithm, HDBSCAN calculates the minimum spanning tree among the closer nodes and finally sort them by a hierarchical clustering (this phse is not present on the original DBSCAN algorithm), as well explained in the related work. There is also a useful Python library used in CLAMS that provide utility functions to make all this step in a very understandable way. The core module of CLAMS is represented by the snippet generator, that performs six main steps in order to obtain the final snippets that represent patterns. First of all, CLAMS replaces all literals present in the code with their abstract type by using srcML, a tool that produce XML file starting from a source code, and removes all comments. At the end of this step, we have code with the same structure of the original source file but more abstract. Then,

CLAMS identifies API call in that code and what are not related to them and creates two lists. In the next step, the authors identify all variable in the scope of the API sequence. To finish the process, the non-API statements are removed and they put on top of snippet variable declaration related to API, plus of course the snippet of code related to those API. So, the snippet of code that is produced in this way is composed by a sequence of variables related to API class and a possible use of them, with considering also the statements present in the original code (CLAMS retained also the structure of the code). The authors put a comment Do Something in the section of code in which the founded variables may be used.
Once CLAMS has these results, the snippet selector module find for each cluster the most representative snippets by giving them a score. To do this, the authors use another algoritm that works on AST, called AP-TED that creates a distance matrix between two clusters and from this, calculate the similarity. Finally, CLAMS ranks all representative snippet using the definition of snippet support define as follow: a snippet is supported if there is a file that contains a supersequence of it. Moreover, for human readability reason, CLAMS beautifies snippet using A-style tool, that removes useless spacing and fixes the indentation of the final snippets. For the evaluation task, the authors use 5 popular Java libraries coming from Github projects and they are Apache Camel, Drools, Restlet framework, Twitter4j, Project Wonder and Apache Wicket. In the evaluation section of their paper, the authors taking into account different clustering algorithm and make experiments with HDBSCAN, already mentioned, k-medoids that is similar to the previous one but achive more coverage but less precision. This algoritm is based on find a central point that has the equal distance from the rest of the point, called medoid. In our case, the medoid is a particular method API call that is the most representative for the library.K-medoid algoritm uses also n-gram based techinique and similarity distance matrix as HDBSCAN. To answer to the research question, the authors of CLAMS preform also a comparison with NaiveSum and NaiveNoSum approaches, that are clustering techinques less precise with respect to the HDBSCAN and K-medoids. Infact, the better results are coming out from the latter algorithms. Furthermore, the evaluation is composed by a user survey about the utility and real support given by CLAMS patterns. All this information are available at the CLAMS official site, that contains also the Github project, the original dataset and the instruction to set up the environment to lauch CLAMS as standalone platform on Linux machine. As claimed by authors, this work is really interesting and flexible because it is not dependend on a specific program language. The only constrain regarding the srcML tool to produce the XML files related to API patterns, but it is not a big issue from the adaptation point of view, as we can see later.

2.4 APIrec

APIrec tool [4], instead, uses statistical techniques in order to keep trace of the context in which a developer writes its own code, by analyzing the co-occurrences and fine-grained code changes. Differently from the previous approaches, the authors keep trace of the context in which the API method call may be useful. As usual, to extract the source code from the 50 Java projects randomly selected from Github, APIrec navigates the AST of the source code using GumTree tool. Before going in deep with the implementation, the authors gives several definition that are useful to understand the behaviour and the contributions to the state of the art given by APIrec.First of all, the term API for the authors indicates both external and internal method calls and APIrec perfoms recommendations only if the context of API is correct for a certain situation. Regarding the AST, we have the atomic change that represent a new element on AST composed by kind of operation, AST node and label. A collection of atomic changes is called transaction and is stored in a bag, a particular data structure that we commonly find in the AST definitions. An important concept that is a key definition for the APIrec implementation

is the code context, represented by code tokens related to the API that the developer is writing. Taking into account the tokens, the authors consider both the distance and the order of this, as an API method calls have a specific call order and they are really effectiveness only if there are called in the proper way. To remark this feature, APIrec gives also a weight based on how near is the token by considering the distance matrix. Once they define this set of metric and concept, the authors show the inference model based on likehood scores taking into account both change of the context and code.
The entire model is baed on the correlation between a code token and another, expressed as we said in term of atomic changes in the AST and transactions. In particular, APIrec evaluates the problability that certain events, namely the transactions, occurs given another. This score is called association score and it is used to calculate the distance between two changes into the code as well as new methods that will take a part in the final recommendation. At the end of these computations, all possible scores and weights are calculated but it is not enough to perfom the recommendations. Infact, it is necessary to apply machine learning techniques in order to train APIrec with the code changes and context. To do this, the authors use hill-climbing adaptive learning filled with three parameters: numbers of co-occurence founded with fine-grained atomic changes, numbers of co-occurrences related to changes tokens and two weights. With this process, the necessary scores is calculated and APIrec is able to perform the recommendation by considering the most probable method calls for a certain API. The methods are also ranked by highest propability of usage but also distance, scope and dependancy are considered in the ranking phase. To support their tool, the authors conducts several experiment over a very large dataset, including also analysis regarding code change context, user empirical studies, evaluation of accurancy and predicitons.

2.5 Buse-Weimer algorithm

Paper from Buse and Weimer [5] is still about API call but is focused to produce automatically a documentation for the projects in a human readable format. Once they obtain data from mining phase, the proposed algorithm extract API pattern and rearrange them in a more readable and effectivness format. The focus of this work is on Java documentation that provides useful hints and suggestions when a developer is implementing an API functionality. Although in general the Java doc helps in this kind of activity, it often lacks in something, as the examples are too general or it not able to give a concrete hint for the problem that the developer is trying to solve. So, the aim of the authors is to produce an enhanced version of documentation that is really useful for the developers, starting from retrieve the API patterns, defined as the sequence of function call for a certain API class,called target class. Once this first step is done, the algorithm tries to produce an human-written documentation for the API, taking into account some characteristics such as the lines of code, that are 11 on average but 5 for the median. Moreover, the authors consides also the abstract initialization, abstract usage and exception handiling. All this information is extracted using JDK utilities and they are validated by the authors, putting more effort on the human aspect. To validate the results, they involve over 150 developer and collect their answers about the proposed results. By analyzing these statistics, a typical developer wants a multiple uses for a certain class or API method and not only one, that may be not useful for his particular context. Another key point is the cociseness of the suggested snippet of code, as well as the readability and variables names that must be related to the context, also including temporary ones to improve the readability. So from this survey, we can identify four key points to achive the human-written documentation: size of the code, readability, representativeness and concreteness.
To reach these goals, let look to the algorithm implementation, that the authors divide into

four main step, the path enumeration, predicate generation, clutering and finally the output documentation. Starting from the path identification, they scan the code and identify acyclic part of the code that represent a path for the target class. Notice that this approach can led some bias but it happends in practice and the authors choose to stay close with respect to real implementation. The So, they use intensively human users to perform test and to measure the accuracy of their proposed algorithm. As the previous approaches, they parse the code and cluster the significant result with a distance matrix in order to build the related documentation. As the purpose is different from previous works, they are adding some clustering also on predicate and represent the abstract pattern as a graph. Then they computes symbolic execution over these paths on order to produce interprocedural path predicats that are logical formulas used to represent the code and in particular, if a certain statement is reachable or not. From this abstraction, the algorithm computes use seeds that are local instantiations of fields, objects and whatever is related to the target class. Finally, from these it produces concrete uses, the real hints about the targe class, that are stored in graph form in which edges keep trace about what happends before and so, in this way, we have the context as well as the chain of methods necessary to implements the features related to the target class. To have better results, however, it is not enough to produce usage examples, as we seen so far with the other related works. Clustering is necessary to avoid heavy computation, and the proposed algorithm exploits the well-known k-medoids algorithm with some modifications, as the original algorithm is not suitable to detect distance about objects. The distance matrix that is taking into account by the k-medoids algorithmis based on the happends before relation obtained from the graph. So, at the end of this computation, the algorithm obtain the cluster and summrize them into abstract uses, represented once again in graph form. The final step of the proposed algorithm is to produce the documentation. Starting from the abstract uses, it uses a topological approach to avoid the cycle and braches that appear in the code, as the final recommended documentation about the target class must be a flat file. Using this approach, the authors are able to retrieve a java documentation related to the target class and respect also the Java sintax, so they avoid malformed documentation. Moreover, with this approach, the algorithm handle the exception threatment with try catch clauses, that are put always in the correct order. As said before, the focus of this approach is on readability of the recommendation from human's prespective, so the authors set up a very big evaluation framework composed by 47 SDK classes ads dataset, the eXoaDOC tool as concurrent approach and over 150 people as tester. The threats rise up from this evaluation are related to the validity of dataset (it may be not indicative and it doesn't represent all possible situation) and the background of the developers choosen for the evaluation (not expert in the field). However, this approach is useful to understand the concept of readability that is very useful for my approach.

2.6 APIMiner

In [6], the authors extend APIMiner tool with API pattern considering Android projects from Github as dataset. In particular, they develop a module that perfom the recommendation based on mining. They perfom the API extraction by considering FP-Growth association as main index of similarity and run it using Weka tool that consider also the relation between two API call, defined as sequence of methods that implements a specific functions. This approach relies completely on Weka implementation. During the mining, the tool discharge the call with single call because is not relevant. The result are evaluated by define two main metrics: support, defined as the number of patterns that include the method, and confidence, the probability of the method in the antecedent transaction. The final recommendatin is displayed in the JavaDoc window in Eclipse and Android Studio, althought the results is related to Android API functions.

2.7 JSS approach

Then, we have a look on [7], that perform clustering by using graph format. The authors, starting from a graph representation of the so called object usage, build a social network based on the co-existing relation among nodes. The aim of this work is to cover the less frequent API pattern in Android context. Before going in deep to the proposed implementation, the authors point out some definitions that are used in the approach. First of all, take a general code snippet, an object usage is a list of methods that belongs to an API class that are used in the part of code that we analyze; in general, a fragment of code can contain more objects usage and this feature is represeted by a co-existence relation among objects usage. So, one object usage became a node in the graph and, if there is a co-existence relation, we put an edge between them and the weight are the number of co-occurrencies. An usage pattern, instead, is the sequence of object usages that belong to different API class. Once they define this to key concept, they underline the challenges undelying the API mining and propose an apporach quite different that we have seen so far. Infacts, the represent the object usage, and more in general usage patterns, with a graph as we said; then, they define the co-existence relation and method call similarity, that are the baseline to define a similarity score among API call. Regarding the co-existence relation, it is represented as a weight in the graph and represent the number of occurences of the object usage in a API class; basically, objects usage that are included in a particular API class are connected by a co-existence relation. As the authors want to reach quite good coverage of API call and avoid the redundacy, they need some clustering technique, as we know from the other related works.
To do this, the exploit the previous definitions and propose two level of clustering, one related to co-existence relation and the other one based on method calls similarity. For the first level, they propose a modularity index applyed to community structures, defined as subnets of node densely connected. So, exploiting the graph format, they apply a greedy algorithm that calculates time by time the modularity, with the function goal that try to achive the maximum one. By running this algorithm, they get the optimum cluster and perfom the first level of clustering. For the second level, they focus on the method call similarity and propose the Gamma index, based on consistent and inconsistent comparison. A comparison is consistent if the distance of two object usage that belongs to different clusters is smaller than an other pair that belong to the same cluster. By applaying these two techiniques, they obtain a good coverage of usage patterns and avoid the redundancy by using abundace metric that describes how many times an object usage appears in the corpus. Through this metric, the authors retrieve the most popular object usage but it is not enough to perfom the recommendations. The last step, infacts, is to map the objects usage into usage patterns in form of real code snippet that support the developer during the API implementation. For the testing phase, the authors provide a lage corpus of 11,520 Android projects coming from Github, focusing on the Android Application Package (APK) because it contains the compiled code and it is not possible to analyze the Google Play original source code. The next step is the setup of a golden set that is a set of queries suitable to the author's purpose. This set is extracted in order to reach the high coverage as possible. The tool is fed with a single query and following the described process, the authors retrieve the expected pattern for that query. To validate the overall process, they also set up an user evaluation with questions about the readability and understandability of the suggested code.

2.8 Jira extension

In [8], the authors propose a tool for mining API and make recommendation winthin the JIRA platform, that is an issue management project based on summary,description and component related to a particular issue. The main idea of this work is analyze the pre-change and post

changed files and, in this way, find recommendations starting from a textual description of the input. The first step is the preprocessing of the input, necessary to clean the code and to give it a proper representation for the algorithm used in next steps. This texual preprocessing is done taking into account two issue: tokenization and stemming. The first one involves the process of break into smaller piece of code the entire document using delimiters as frontier and put it in a word token structures (also called bag of words). The stemming, instead, is related to the root of the word and transform it in stem word: in this way, the authors summarize multiple words to avoid bias during the analysis. Once this preprocessin is done, the algorithm use a term frequency indicator to count the number of times that a word appear in the docuement and so, obtain the most popular token. A similar measure is calculate also for the document and after a formula showed in the paper, the authors retrieve weights and put them in a vector; in this way, each bag of word is associated to a weight that measure its relevance for the recommendation.
The framework is composed by three main part: history based recommender, descriptor based recommender and the integrator that put all together. For the history recommender, the algorithm compares the indicator on the Jira platform using similarity distance matrix, starting from Jira fields, named This module consider summary and description as key value of comparison and store them in its knownledge base called Historical Feature Request Database. Once the similarity scores are obtained, the algorithm perform aggregation of this scores and perfom the final comparison between the historical scores (calculated in this step) and the new feature request that is coming. To do this, they create a top-k request looking at the history and choose the recommendation with the highest value among them. Description based component, instead, compares the new feature request with the Javadoc of the method, to have a more detailed recommendation. Here there is a preprocessi phase regarding the API doc which consists in extract method call taking into account the @param and @return annotation plus the discharge of HTML tags and Java comments. As similarity measure, in this phase they use cosine similarity between the corrent feature and the preprocessed API. The last component is the integration, that merge the historical part with the description part, apply Gibbs sampling and try to calculate the best results at each iteration. In paticular, they pick first the no-zero historical recommendations and then compare them with the results of descriptor module; from this results, tha algorithm creates a top rank recommendation realed to the developer's features. Regarding the evaluation of these results, the authors select 5 most famous Apache projects (Hadoop, CXF, AxisJava, Hbase and Struts 2) and loooking for Github projects that implements this libraries. They filter these projects considering the presence or not of the pom.xml file and, after this preprocessing, they retrieve 207 projects as corpus of the tool. To select the golden set, the authors also considering the status of the file that belongs to Jira platform: in particular, they taking into account is the new files are added or are changed with respect to original while they not include the deleted files to the golden set.

2.9 CodeBroker

Finally, we analyze CodeBroker tool, proposed in [9] that use infomation retrieval techniques in order to make recommendations. The authors consider a lot of techinques for theri tool, such as information delivery, retrieval by reformulation, knowledge augmentation and finding task with similarity metric. All these definitions compose the conceptual framework that is a baseline for the implementatio and evaluation of the CodeBroker tool. It is a interface agent with back-end utilities that takes a query as input and return the component related to it. Notice that this recommendation is based on Javadoc generate from Java source files. The tool is based on two communication channel with the developer that is insterested in API recommendation: one is an implicit where the system autonomously retreive methods information and details from a

given query. In this case it shows information organized in three layers, namely task relevance, signature details and full JavaDoc. Regarding the second channel, it is explicit because the developer can refine the query based on its current needs and the system can adapt it self to this new situation and this technique is called retrieval by reformulation. Furthermore, CodeBroker creates a discourse model to represent the projects and an user one to represent in some way the software knoledge and personal infomation regarding used method in the project. It uses LSA as similarity techniques to do comparison and Java core libraries as dataset to test the tool. About the implicit communication, this term describe a set of information that can be inferred by the system without taking in account the user's hints; for the explicit channel communication, instead, the tool consider the user's need by looking its model plus the discourse one that I mentioned before.
As first step, CodeBroker arrange the query taking into account the context of the developer, called constrain part, the program that is the concept of functionality and the code that is the embodiment of it. So for the similarity analysis, the authors consider the context and so the conceptual similarity and also the constrain similary, that involved betwee two different signature of the methods. About this last concept, they reuse it to apply the Latent similarity analisys (LSA) as main technique to perform comparison between two different API methods. Going in deep, the tool perform the so called signature matching that outlines the similarity of two componet based on their signature structure. Although this comparison should be not representative enough, the authors claim that is suitable for thei purpose: so, the value of the comparison is in the range from 0.0 to 1.0, that represent the exactly match between two different method. However, this first analysis must be enhanced by retreival by reformulation technique as the LSA cannot analysis in deep fragment with comments or task relevant information from the code. So, as mentioned before, the authors use explicit communication channel to allow the developer to formulate once again the inital query: the typical use case is that a user want to improve the initial query with other components and so he change the query in order to retrieve more information or very different components with respect to the initial one. It is true especially for the Github repository, that have very complex structure inside them and may a non expert developer want to know all this details. Now it is time to introduce the concept of module, namely the part of the code that implements the developer's main feature. This kind of activity is perfomed by building the discourse model of the developer, that represent the sequence of tasks necessary to implement the all features and it is used to improve the final components recommendation. At the beginning, this model is empty as the developer is starting to develop and he doesn't know a priori what are the components that are useful for his task. During the query phase, this model is fillied with respect to the developer's choice and all this information are retreive on the RCI console used for the final recommendation. There is another model that CodeBroker takes into account during its analysis: the user model, that represent the developer's knowledge in abstract form. Based on this defintion, it is very different with respect to the discourse model and it is partially filled based on the developer skills. This model is used to remove possible componets that the user already know and so to avoid the redundacy problem. The authors define the knowledge as the number of implemented class by the developer and, in general, it is different from user to another. The final recommendations is perfomed through RCI-display already mentioned, with three layers: the first one shows the components related to the query, the second is linked to mouse movement (it displays signature infomation about the retrieved components) and, finally, a completed description in HTML external page, included the JavaDoc, affers to the third layer recommendation. For the evaluation task, the authors use Java 1.1.8 core libraries and JGL library, with 663 classes and about 7000 methods to analyze.

3 Problem definition

So far, we have seen several approaches to perform recommedations related to API context. The proposed tool from the literature are very different from the point of view of dataset, similarity measure and clustering techinques but they share a conceptual model that starts from the abstract syntax tree of the code to the final recommendations. This procedure is depicted in Figure 4, that describe at high level the main steps to reach the final output. Many of these are optionally but, as we can see from the related works section, are strongly recommended to perfom a bettern analysis and to have good enough results. Although this approach is very common, there are differences in definitions, methodologies and techiniques that make an approach very different from another, starting from the basic concepts of recommendation, API and pattern. These key concepts, that compose the core of my work, are often ambiguous in the literature and change meaning based on the cosidered context. For this reason, before to describe my approach to face the issue, I define the building block of it, looking at the common meaning for each of them as well as the defintion taking into account my developement context

3.1 Concept of recommedation

3.2 Concept of API

3.3 Concept of Pattern

4 Code clone with Simian tool

4.1 About code cloning

4.2 Simian overview

As we see from the releated works, we can perform recommendation at differerent levels of abstraction (pattern, methods, code snippets) in order to give a complete and useful hints to a developer. If we look at level of code, there are many tools that perform code clone analysis by retrieving some information about lines of code in the same project or file by file. I analyze Simian tool, a project developed in Java that performs this kind of analysis for many languages as Java, C, C#, Ruby, JavaScript, COBOL, Lisp, SQL, Visual Basic. Once I download the .tar file, I simply run the simian.jar file using command line interface in the directory that I'm interested to analyze. We can enhance the Simian analysis by using a lot of options such as fix a threshold for the number of duplicated lines, ignore literals or string case, numbers, parenthesis and so on plus specifiing singolar files. For my analysis, I focus the attention on Java projects and I run the tool using options only related to Java code, so I discarted spefiic options for other tool (for example C++ or Ruby) or options that are too restrictive such as ignoreSubtypesNames or ignoreLiterals. To make a better analysis, I consider both projects that we can suppose are similar in some way and very different projects that implements different functionalities. Table 3 and Table4 decribe respectvely the options that I have used to perfom the comparison among projects and projects characteristics and similarities while Figure 2 represent the related output, both in case of strong similarity and no common lines of code. Once we are in the diectory that contains the projects to analyze, the command is java -jar simian.jar path and the options that we want to use.

Table 2: Simian options used in the experiment

Option name	Default value	Description
-threshold	6	This option fix an lower bound on the number of duplicated lines of code (if present)
-formatter	none, possible values: plain, xml, emacs, vs (visual studio), yaml, null	This option is used to obtain results in a specified format
-reportDuplicateText	disable , type + to add	With this option, the duplicated lines of code present in all projects are printed on the console
-language	disable , type + to add	This option specify the language of the input files to compare
-defaultLanguage	disable , type + to add	If not file type is not specified, Simian inferred the type and set it as default
-failOnDuplication	able , type - to remove	If this option is able, it causes an exception when the checker finds duplicate code
-reportDuplicateText	disable , type + to add	With this option, the duplicated lines of code present in all projects are printed on the console
-ignoreRegions	disable , type + to add	It ignores block in regions structures (only for C# programming language
-ignoreBlocks	disable , type + to add	It excludes specified blocks from the comparison (start/end line must be specified
-ignoreCurlyBraces	disable , type + to add	The curly braces are ignored so it should be match as duplicate line
-ignoreIndentifier	disable , type + to add	With this option, the variable with different (identifiers match as equal
-ignoreIdentifierCase	able, type - to disable	This option not consider the case of identifiers present in the code: so Name and name are considered equal)
-ignoreStrings	disable , type + to able	This option consider all strings in the comparison and doesn't take care about the form in which are write
-ignoreStringCase	able, type - to disable	Same as previous option but consider the upper and lower case as the same
-ignoreNumbers	disable, type + to add	This option considers different numbers as equal
-ignoreCharacter	disable, type + to add	With this option, all character type match as equal
-ignoreCharacterCase	able, type - to disable	Same as ignoreStringCase but consider char by char. Useful for more precise analysis
-ignoreLiterals	disable, type + to add	All literals should be seen as equal for Simian
-ignoreVariableNames	disable, type + to add	This option allow to Simian to see different variable names as equal
-ignoreModifiers	able, type - to disable	This option doesn't consider modifiers of methods (public, private, protected as element of diversity in the code

Table 3: Projects considered in the comparison

Projects name	Main features	Similarity level (duplicated LOC)
ADTPlugin, ModiscoPlugin	Plugin projects created with same wizard	39 lines of code in common
CyberGea, NeoEMFExample	Cybergea: Plugin, Servlets and JDBC NeoEMF: Metamodels, Neo4J facilities, EMF framework	No lines in common
CyberGea, Scuna project	Cybergea: Plugin, Servlets and JDBC SCuna: Swing GUI and JDBC	12 lines on common (basic JDBC statement)
Simple Servlet, ServletSsession	Web projects with servelts	35 lines of code in common

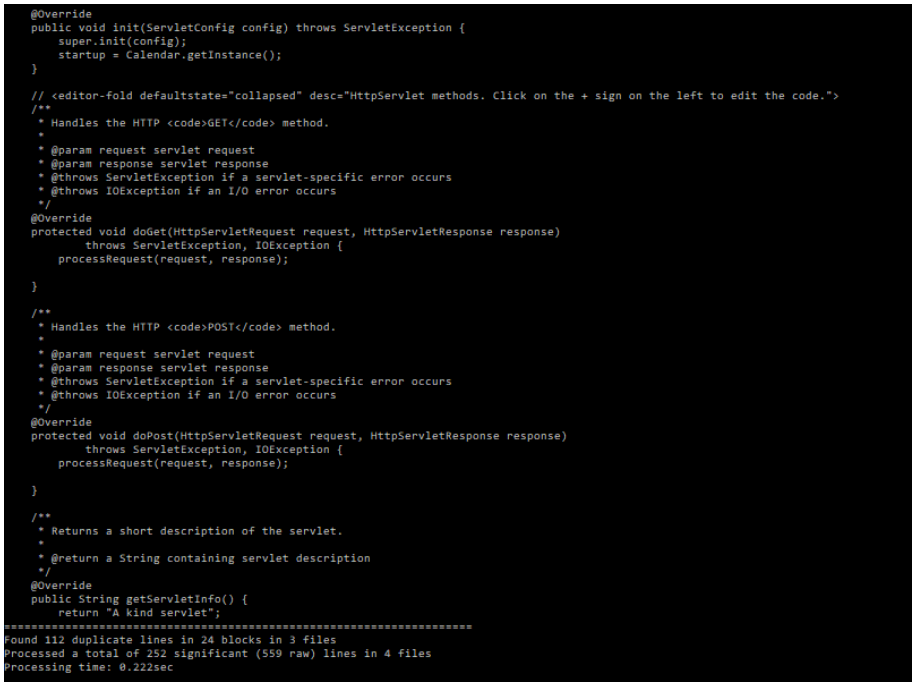

Figure 1: An example of simia tool output

In particular, I showed that the projects that follow the same structure such as Plugin projects using the same wizard for the creation or the same framework shared same lines of code while in case of very different projects, as in the comparison between CyberGea and NeoEmf, there are no commons lines of code.

5 Proposed approach

5.1 Overview

After the problem statement and an brief description of Simian tool, let me present a possible approach to solve the problem of API function call recommendations. This approach, described in Figure 2, exploits the CLAMS work, in particular the patterns extracted as output, plus the code cloning features provided by Simian. So, at the beginning of the process, we have the patterns files and the devoper's file, represented by a single string. Notice that with this method we keep trace on the context in which the user is developing. Then, all these files is used to extract the recommendations in form of patterns by using Simian integrated in Eclipse platform, following options that I specify in next sections. Basically, at the of this phase, Simian retreives the cloned clone between the developer's file and the CLAMS patterns related to the library that the user is implemented. By using these file, I perfom recommendations by remove the cloned part and suggest to the user the new lines of code that represent the missing pattern for the user. In next sections, I going in deep to describe the entire system and how the integration of Simian and CLAMS works in practise.

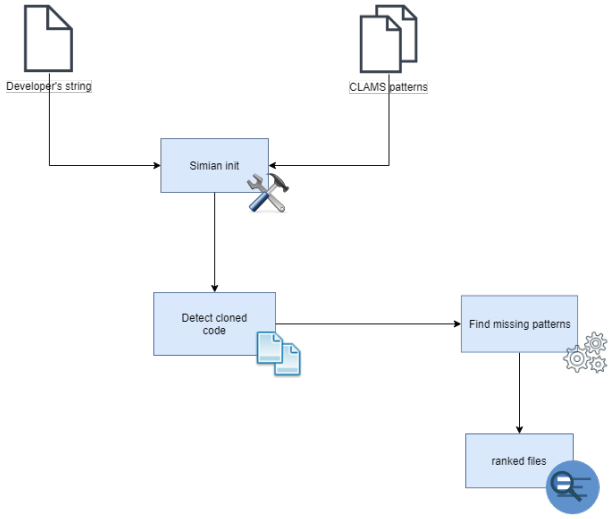

Figure 2: Recommendation for a single file using Simian and CLAMS

5.2 Input files

About input files that are necessary to initialise the tool, we have from one side the developer's file with the real snippet of code that he is implementing and may want support for this. On other hand, there are patterns mined by CLAMS, in form of ranked Java files sorted by rational specified in CLAMS paper. These file contains patterns, defined as sequence of API method calls that define, instantiate and using class belonging to the APIs contained in the developer'string. The number of these file and also their dimensions in term lines of code depend on the considered libraries. As Simian is a tool based on file comparison, I need to use temporary files of Java to do the comparison; after the process, the files are destroyed.

5.3 Simian within Eclipse platform

Once we define the imput, there is another step in order to use Simian to do API recommendations: the integration with Eclipse platform to have a more flexible and usable version of the tool. As I said in previous section, the basic version of Simian is a jar file that I launched from the terminal console with different options (see the Table 1). Although it is very easy to use, in this version is not integrable with the other parts of the system and it is necessary to integrate directly the Simian jar file, available on this site https://www.harukizaemon.com/simian/javadoc/index.html. To keep the integration with a Maven project, we create a repository that contain the update versionof this jar and put it as reference in the POM file of the project.

Following this documentation, we have the following main classes:

Table 4: Overview about Simian classes

Simian class	Description
Auditstener	This class is necessary to initialise Simian tool and collect all notification from events that occur
Block	This class represents the duplicated block of code as an object and we can interact using method utilities
FileLoader	It is used to load all files for the comparison, with the method load
Checker	This class is used to perform the real comparison by calling the method check() on preloaded files
StreamLoader	Once we load files and create the Checker, this class load them into the Checker
Options	A data structure that encapsulates all options enabled for the comparison
Option	This class represents a single option and we can specify it by accessing to a static field
Language	This class contains static fields to set all supported languages as type of input files
CheckSummary	It contains all statistical data such as cloned code, number of total files, requested time and duplicated files

5.4 CLAMS adaptation

Regarding the CLAMS ouput, I describe the original structure as well as the necessary modification to integrate it in our approach. The authors, differently from other works, provide the complete source code and commands to set up the entire environment to have CLAMS working on Linux . CLAMS is written in Python and uses srcXML and Astyle to produce xml files and to formatter in a human-readable way the code respectively, but as claimed by the authors, there is no really constrain about the technologies to use in case of a new implementation. As input, CLAMS takes two kind of files: client files, that represent the real project on Github related to the dataset that authors use for evaluation phase while example files are used to find the method call and for the ranking phase. All these files are collected in a folder, that CLAMS loads by getting the path. Moreover, there is a namespace file that idetify the name of classes used in the clients and example files by using their complete namespaces such as org.codehaus.jackson. The last input used by the main.py file, that is used to initialise the platform, is the methods cals and caller are represented by an .arff file, using in the machine learning domain.
There is a phase of preprocessing in which CLAMS extracts API call and their AST using JDT utilities and represent them in xml using srcXML. The core of the project is the snippet generator module (represented by summarise.py file) that takes as input a source code file (java in this case) and using srcXML they first replace literals with xml types and delete comment. Then , they separate the API code from code that doesn't contain API call and highlights the variable in local scope of API. Finally, the code without API call is removed and Clams add some comments near the API statement and needed variables. Notice that their approach considers also the classical statement like if-else structure as a part of API statement. For clustering, they use both HDBSCAN and k-medoids algorithm that are quite similar and differs only in the preci-

sion of the returned snipped (HDBSCAN is more accurate but k-medoids covers more methods). For both of them, the authors import Python libraries that implement these algorithm quite well and we can switch the algorithm by change the parameter in the main.py. Moreover, they have the file ranking.py to order the generated snippet. The rank is based on the example files that contains a sequence of API call; if the sequence within the file is a super-sequence of the sequence of snippet that we considered, so this snippet is supported, and its rank is increased. In the result folder, CLAMS put the library that we want to analyze, the methods, the source file (both in .java and xml format), some json file that represent all information about a method (class, package, rank, id) and the arff file related to the library.
For the integration step in our platform, it is necessary to slightly modify the original approach to have better results. In particular, if we use the pattern of CLAMS as they are, there are some bias because, through srcML, CLAMS substitutes the literals with its own type and Simian is not able to detect them as cloned code, even using all available options regarding the code. So, to avoid this situation, we must modify the function that substitute literals, putting some default value instead of srcML types. This modification doesn't affect the validity and accurancy of extracted path because is just a matter of modify literals with another. Furthermore, we have to modify the clients files with our dataset that is composed by projects that use MQTT and some Json libraries.

5.5 API recommendations

At the end of these preparatory phases, I can start with decribe the core of this project, the API recommendations. Once the Simian is lauched, it performs the detection of code cloning activity on the CLAMS patterns files and the developer's code snippet. Notice that the notion of cloned code depends on the options that we have selected and turn on as the Table 4 decribes. The mandatory options to enable is the threshold, that set the minimum line of code in commons, reportDuplicateText, otherwise we couldn't show and manipulate the result and language that is Java because we analyze projects related to it. Other options, such as the strings, identifiers or modfiers that should be introduced in the comparison, can be enable with respect to the level of cloning that we want to reach. To find useful results, it is necessary to set at least ignoreIdentifiers, ignoreIdentifierCase, ignoreLiterals, ignoreVariableName, ignoreNumbers and ignoreModifiers because Simian goes beyond the developer personal implementations and looking only for the structure of the code, in order to use the concept of pattern in a more effective way. Furthermore, I compare the pair developer' snippet - pattern because some CLAMS pattern includes some duplicated lines of code and this can bring some bias. Once we load the files, the check is performed and the results that include lines of code, name of pattern file and time to perfom the comparison. At the end of this step, we have the patterns (a complete one or only partial) that the developer is start to implement and we can dischard this from the comparison, as the developer is not interested to see what he have done so far. Moverover, new pattern can introduce something new or suggest an alternative implementation for the developer. The last step is remove the duplicated line of code from the suggested pattern and show to the developer only the novel part, in form of some that integrate his code or completely new pattern, related of course to the APIs that he is implementing. About the ranking, we order the pattern by considering the number of cloned lines, so the first pattern is the contains more duplicated lines rather than second and so on. The rank phase is simply perfomed on the APIRecommendation object that we produce as output.

```
Current file
 Twitter mTwitter;
    final String CONSUMER_KEY;
    final String CONSUMER_SECRET;

    final Action action;
    final User first_source;
    final int sources_length;
```

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

```
New pattern found
    mTwitter = new
        TwitterFactory().getInstance();
    mTwitter.setOAuthConsumer(CONSUMER_KEY,
        CONSUMER_SECRET);
```

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

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

```
package com.nookdevs.twook.services;

import java.util.ArrayList;
import twitter4j.DirectMessage;
import twitter4j.ResponseList;
import twitter4j.Twitter;
import twitter4j.TwitterException;
import android.util.Log;

import com.nookdevs.twook.activities.Settings;
import com.nookdevs.twook.activities.Tweet;
import com.nookdevs.twook.utilities.Utilities;

public class DirectMessagesDownloaderService extends MessagesDownloaderService {
    private static final String TAG = DirectMessagesDownloaderService.class
       .getName();
```

```
    @Override
    protected ArrayList<Tweet> getTweets() {
      ArrayList<Tweet> tweets = new ArrayList<Tweet>();
      Settings settings = Settings.getSettings(this);
      // The factory instance is re-useable and thread safe.
      final Twitter receiver = settings.getConnection();
      ResponseList<DirectMessage> messages;
      try {
          messages = receiver.getDirectMessages();
          for (DirectMessage message : messages) {
         final Tweet tweet = new Tweet();
         tweet.setMessage(message.getText());
         tweet.setUsername(message.getSender().getName());
         tweet.setImage(Utilities.downloadFile(message.getSender()
            .getProfileImageURL()));
         tweets.add(tweet);
          }
          return tweets;

      } catch (TwitterException e) {
          Log.e(TAG, e.getMessage());
          return new ArrayList<Tweet>();
      }
    }
}
```

```
    final String TAG;
    final Twitter twitter;
    try {
        Query query = new Query(((SearchActivity)getMainActivity()).getSearchTerm());
        QueryResult result = twitter.search(query);
        return Utilities.tweetsToTweets(result.getTweets());
    } catch (TwitterException e) {
        Log.e(TAG, e.getMessage());
    }
```

6 Validation

6.1 Evaluation framework

6.2 Comparing results

6.3 Perfomances

7 Conclusion

References

[1] Lu Zhang JianPei Hao Zhong, TaoXie and Hong Me. Mapo: Mining and recommending api usage patterns. pages pp 318–343.

8 References

Here there are the related works that I consulted until now:

[1] Hao Zhong,TaoXie, Lu Zhang,JianPei and Hong Me, MAPO: Mining and Recommending API Usage Patterns, In ECOOP pp 318-343, 2009
[2] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, Dongmei Zhang, Mining succinct and high-coverage API usage patterns from source code, MSR pp 319-328, 2013
[3] Summarizing Software API Usage Examples using Clustering Techniques, Nikolaos Katirtzis, Themistoklis Diamantopoulos and Charles Sutton
[4] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, Danny Dig, API code recommendation using statistical learning from fine-grained changes. SIGSOFT FSE pp 511-522,2016
[5] Raymond P. L. Buse, Westley Weimer, Synthesizing API usage examples. ICSE pp 782-792, 2012
[6] Hudson S. Borges, Marco Tulio Valente, Mining usage patterns for the Android API. PeerJ Computer Science 1: e12 2015
[7] Haoran Niu, Iman Keivanloo, Ying Zou, API usage pattern recommendation for software development. Journal of Systems and Software 129: 127-139, 2016
[8] Ferdian Thung, Shaowei Wang, David Lo and Julia Lawall, Automatic Recommendation of API Methods from Feature Requests, ASE 2013, Palo Alto, USA
[9]Gerhad Fischer and Yunwen Yu, Reuse-Conducive Development Environments, Automated Software Engineering, 12, 199–235, 2005