Parameter-Free Probabilistic API Mining at GitHub Scale

Jaroslav Fowkes†  Charles Sutton†

†School of Informatics
University of Edinburgh, Edinburgh, EH8 9AB, UK
{jfowkes, csutton}@ed.ac.uk

ABSTRACT
Existing API mining algorithms are not yet practical to use as they require expensive parameter tuning and the returned set of API calls can be large, highly redundant and difficult to understand. In an attempt to remedy these shortcomings we present PAM (Probabilistic API Miner), a near parameter-free probabilistic algorithm for mining the most informative API call patterns. We show that PAM significantly outperforms both MAPO and UPMiner, achieving 70% test-set precision, at retrieving relevant API call sequences from GitHub. Moreover, we focus on libraries for which the developers have explicitly provided code examples, yielding over 300,000 LOC of hand-written API example code from the 967 client projects in the data set. This evaluation suggests that the hand-written examples actually have limited coverage of real API usages.

1. INTRODUCTION
Learning the application programming interface (API) of an unfamiliar library or software framework can be a significant obstacle for developers [28, 29]. This is only exacerbated by the fact that API documentation can often be incomplete or ambiguous [33]. Fortunately, an opportunity to address this problem has arisen out of the simultaneous growth in the amount of source code that is available online and the growth of large scale data mining and machine learning methods. This confluence has enabled the development of API mining methods [39, 34], which aim to automatically extract a set of API patterns, which are lists of API methods called by client methods, and that together characterize how an API is used.

Despite a number of interesting proposed tools, including well known ones such as MAPO [39] and UPMiner [34], so far API mining tools have not been successful at gaining wide-spread adoption in development environments. We contend that the fundamental reason for this is that the quality of the extracted patterns is not yet high enough: the patterns returned by current methods are numerous, highly redundant, difficult to understand and may miss sequences of interest. For example, Figure 5 shows the top ten API patterns extracted by two state of the art methods. To a large extent, the patterns from both methods are variations on a theme, repeating different variations of ways to repeat the same few API methods.

The fundamental reason for this, we argue, is that current API mining methods are built on statistically shaky ground. Specifically, API mining algorithms largely employ frequent sequence mining, which is a family of techniques from the data mining literature that takes as input a database of sequences and from those attempts to identify a set of patterns that frequently occur as subsequences. In the context of API mining, each sequence in the database is the list of API methods called by some client method, and the subsequence patterns that are mined are candidates for API patterns. Frequent sequence mining methods are very good at their intended purpose, which is to efficiently enumerate subsequences that occur frequently. But they are not suitable for pattern mining all by themselves, for a simple reason: frequent patterns are often uninteresting patterns. Items that are individually frequent but uncorrelated can appear in the list of frequent sequences simply by chance. As an illustration, imagine running a frequent sequence mining algorithm on the sequence of events that occur in the day of a typical academic. The resulting list might include patterns like (SENDEMAIL, DRINKCOFFEE) even though these actions have no direct causal relationship. Such a pattern is not useful because to say that the academic does both actions in sequence provides no additional information over and above just saying that the academic does each of the individual actions constantly.

To address this problem, we must return API patterns that not only occur often, but also that occur in sequence more often than expected by chance. We call these the most informative sequences. In order to quantify what “most informative” means, we employ a powerful set of techniques from statistics and machine learning called probabilistic modelling. That is, we design a probability distribution over the list of API methods called by client methods, and we evaluate a proposed API pattern by whether it improves the quality of the model. The quality of a probabilistic model can be measured very simply by whether it assigns high probability to a training set.

While numerous sequential probabilistic models of source code have appeared in the literature [16, 6, 27] these all rely on n-gram language models. But an n-gram in this context is a contiguous sequence of method calls, whereas API patterns contain gaps, that is, the methods in an API pattern do not always occur contiguously in a client method, but instead can have other method calls interspersed between them. For example, consider the methods in Figure 1 which show real-world usage examples of a four line call sequence to set up a twitter client. The first four lines of the last method shows the basic call sequence but the other two methods have additional calls interspersed between them, e.g. the first method also turns on SSL support. Although frequent sequence mining algorithms handle gaps in sequences automatically, pre-
vrious probabilistic models of code do not, so we need to introduce a new probabilistic model. To address this unsatisfactory state of affairs, we propose PAM (Probabilistic API Miner), a near-parameter-free probabilistic algorithm for mining the most informative API patterns. PAM makes use of a novel probabilistic model of sequences, based on generating a sequence by interleaveing a group of subsequences. The list of component subsequences are then the mined API patterns. This is a fully probabilistic formulation of the frequent sequence mining problem, is able to correctly represent both gaps in sequences, and — unlike API mining approaches based on frequent sequence mining — largely avoids returning sequences of items that are individually frequent but uncorrelated.

Furthermore, another drawback of current methods for API mining is that they have multiple parameters that are very hard to tune in practice, rendering them difficult to use without expensive pre-training. First, frequent sequence mining methods depend on a user-specified dataset-specific minimum frequency threshold [4, 15]. This threshold is extremely difficult to tune in practice as it is prone to exponential blow-up: setting the threshold too low leads to billions of patterns. Conversely, setting it too high leads to no patterns at all. API mining algorithms also tend to employ hierarchical clustering techniques in an attempt to reduce the inherent redundancy of frequent sequence mining (which tends to produce highly repetitive patterns). Choosing the number of clusters/cutoff is considered somewhat of a black art and can only be reliably inferred through computationally expensive training procedures on a validation set [39, 34]. PAM, on the other hand, is near parameter free in the sense that our two user-specified parameters are independent of the dataset, have simple intuitive interpretations and sensible default values.

To showcase PAM, we apply it to mining API patterns for 17 of the most popular Java libraries on the GitHub hosting service. We collect all usages of these APIs on GitHub, yielding a data set of 54,911 client methods from 967 client projects totalling over 4 million lines of code (LOC). We focus on libraries that contain a specific examples/ subdirectory, which allows us to automatically collect API usage examples that the libraries’ developers thought were most salient. This yields a corpus of 3,385 Java files comprising 300,000 LOC solely of hand-written API usage examples. These two sources of information allow us to perform an extensive evaluation of PAM compared to MAPO and UP-Miner, at a scale that has not previously been conducted in the API mining literature.

Our main contributions in this paper are as follows:

- We present PAM, a novel and practical algorithm for mining API call sequences that are most informative rather than most frequent.
- We show that PAM significantly outperforms both MAPO and UP-Miner, achieving 70% test-set precision, at retrieving relevant API call sequences of 17 popular Java libraries across 967 open-source Java projects in GitHub.
- To the best of our knowledge, we are the first to mine API calls at GitHub scale. Moreover, PAM exhibits very fast performance, taking an average of 23 minutes per set of library API calls.
- To the best of our knowledge, we are the first to use API call sequences from API library examples directories to evaluate API mining algorithms. Our results suggest that even in projects with extensive examples, the example directories have incomplete coverage.

We show that the set of patterns that PAM returns is dramatically less redundant than those from MAPO or UP-Miner. This is because in our probabilistic framework, a potential list of the patterns is judged holistically by how well the list as whole explains the API usage data.

2. RELATED WORK

The first algorithm for mining API usage patterns from source code was MAPO, proposed by Xie and Pei [37] and extended by Zhong et al. [39]. MAPO [39] mines method call sequences (that call a desired API) from code snippets retrieved by code search engines. Sequences of called API methods are first extracted from the code snippets and then clustered according to a distance metric, computed as an average of the similarity of method names, class names, and the called API methods themselves. For each cluster, MAPO mines the most frequent API calls using SPAM and feeds them to an API usage recommender that ranks them based on their similarity with the developer’s code context.

UP-Miner [34] extends MAPO in an attempt to further reduce the redundancy of the mined API call sequences. This is achieved through three principal approaches: using the BIDE closed frequent sequence miner that returns only the frequent sequences that have no subsequences with the same frequency; using a clustering distance metric based on the set of all API call sequence n-grams and an additional clustering step on the mined call sequences from each cluster. Unlike MAPO, the API call sequences are presented as probabilistic graphs, ranked according to their frequency.

Acharya et al. [3] extract API usage scenarios among multiple APIs as partial orders. Frequent API calls are first mined from inter-procedural program traces, API call ordering rules are then extracted from the frequent calls and presented as partial order graphs.

Buse and Weimer [10] generate API usage documentation by extracting and synthesizing code examples from API method call graphs annotated with control flow information. Their approach first extracts method control flow paths from files that use a desired API. Path predicates are then computed for each statement relevant to a static instantiation of a desired API type and assembled, along with the statements, into API usage graphs. The API usage graphs are then clustered based on their statement ordering and type information, and an abstract usage graph is extracted from each cluster by merging its concrete API usages. Finally, each abstract API usage graph is transformed into a code fragment by extracting the most representative abstract statement ordering from the graph and assigning each abstract statement a concrete name according to heuristics.

MUSE [23] uses static slicing to extract and rank code examples that show how to use a specific method. For each relevant method invocation, MUSE extracts an intra-procedural backward slice, representing a raw code example. The raw code examples are then clustered using type-2 clone detection and the resulting groups of clones ranked according to their popularity. A representative code example from each group of clones is selected based on its readability and ease of reuse, and each example is annotated with inline descriptions of method parameters mined from the Javadoc. Each
ranked list of code examples is presented as an HTML page injected into the official Javadoc documentation.

Other API mining papers include Uddin et al. [32] who detect API usage patterns in terms of their time of introduction into client programs and Li et al. [20] who automatically extract implicit programming rules in large software code. A related approach that has been well-studied in the literature is API code search, where text matching is used find API usage examples from a large, often online, corpus. SNIFF [11] finds abstract code examples relevant to a natural language query expressing a desired task. SNIFF annotates publicly available source code with API documentation and the annotated code is then indexed for searching. In response to a query, matching code examples are retrieved and concise examples are extracted via a syntax-aware intersection, clustered and ranked based on their frequency of occurrence. Similarly, Keivanloo et al. [18] find abstract code examples by combining textual similarity and clone detection techniques, ranking the returned examples according to their similarity to the query and the completeness and popularity of their concrete usage patterns. Bajracharya et al. [9] find concrete API usage examples by combining heuristics based on structural and textual aspects of the code, based on the assumption that code containing similar API usages is also functionally similar.

Strathcona [17] is a code example recommendation tool that automatically generates queries from the developer’s current code context. The code examples relevant to the queries are identified using six heuristics that compare the structural context of the query against the structural context of the classes and methods within an example repository. The resulting code examples are ranked according to their frequency in the final set of top examples returned by each heuristic. Prompter [26] takes a similar approach, matching the generated context-specific queries to Stack Overflow discussions and automatically recommending the discussions most relevant to the developers’ task.

Sequence mining is an extremely well studied problem with a huge number of published papers on the topic. The problem was first introduced by Agrawal et al. [4] along with the GSP algorithm (based on Apriori [5]) which recursively builds larger frequent sequences from frequent items. The PrefixSpan algorithm [25] mines frequent sequences using the pattern-growth approach: growing a prefix pattern to get longer sequential patterns by recursively constructing and mining projected databases. SPAM [8] is tailored towards efficiently mining sequences that are very long and adopts a vertical bitmap representation of the database. The BIDE algorithm [35] uses bidirectional search to mine frequent closed sequences (i.e., those that have no subsequences with the same frequency) without candidate maintenance.

There has also been some existing research on probabilistic models for sequences, especially using Hidden Markov models. In particular, Stolcke et al. [31] developed a structure learning algorithm for HMMs that learns both the number of states and the topology. Landwehr [19] extended HMMs to handle a fixed number of hidden processes whose outputs interleave to form a sequence. Wood et al. developed the sequence memoizer [36], a variable order Markov model with a Pitman-Yor process prior. Also, Nevill-Manning et al. [24] infer a context-free grammar over sequences using the Sequitur algorithm. There have also been sequential probabilistic models of source code proposed in the software engineering literature. Hindle et al. [16] develop an $n$-gram language model for source code and apply it to a code completion task. Allamanis et al. [6] use an $n$-gram language model to learn and suggest variable naming conventions. Raychev et al. [27] develop an $n$-gram language model that synthesizes code completions for programs using APIs.

3. EXTRACTING API CALLS

For the purposes of this paper we use a best-effort approach to extract API call sequences directly from source code files. Following MAPO [39] we parse each Java source file of interest using the Eclipse JDT parser and extract method calls to API methods from a specified library using a depth-first traversal of the AST. For simplicity, unlike MAPO, we do not attempt to enumerate all possible branches of conditional statements. For example, for the code snippet

```java
if(m1()) { m2(); } else { m3(); }
```

our method returns the call sequence $m1,m2,m3$ whereas MAPO would return the call sequences $m1,m2$ and $m1,m3$. We would argue that returning $m1,m2,m3$ is better in principle as it represents the entire API call construct, especially if the construct is a common API usage pattern.

Furthermore, unlike MAPO, we only consider method invocations and class instance creations, and approximately resolve their fully qualified names from the file import statements. For simplicity, superclass methods (and their return types), super method/constructor invocations and class cast expressions are not considered.

We use an approach similar to the original MAPO paper [37] to approximately resolve fully qualified method names. We keep track of field and local variable declarations so that we can resolve the fully qualified name of a method call on the field-variable. We also keep track of import statements so that we can resolve fully qualified names of classes that are explicitly imported, as well as those imported using a wildcard by scanning for specific imports from the wildcarded package in the corpus in a pre-processing step. Additionally, in the pre-processing step we find the return types of locally declared methods so that we are able to subsequently resolve any calls on them. Finally, we filter out any method names that cannot be fully resolved. Each API call sequence is then the sequence of fully qualified API method names that are called by a method in the source file.

For example, consider the client methods in Figure 1 which all share the common `twitter4j` API call sequence:

```java
conf.ConfigurationBuilder.<init>
conf.ConfigurationBuilder.setOAuthConsumerKey
conf.ConfigurationBuilder.setOAuthConsumerSecret
conf.ConfigurationBuilder.build
```

This is the minimum API call sequence required to set up OAuth authorization for a twitter client. All the methods in Figure 1 have added extra API calls for optional functionality (e.g., SSL encryption) but all contain the minimal API call sequence as a subsequence.

There are of course limitations to this approximation (as duly noted in the original MAPO paper [37]), in particular it is not possible to resolve external nested method calls (i.e., in the call `method1().method2()`, we cannot resolve `method2()` unless `method1()` is declared locally). However, for the purposes of this paper we are primarily interested in assessing the performance of PAM. Moreover, it is important to note that PAM is flexible and supports any API call extractor that returns sequences of calls, making it applicable...
private FinchTwitterFactory(Context context) {
    mContext = context;
    installHttpResponseCache();
    ConfigurationBuilder configurationBuilder = new ConfigurationBuilder();
    configurationBuilder.setOAuthConsumerKey(Keys.consumerKey);
    configurationBuilder.setOAuthConsumerSecret(Keys.consumerSecret);
    configurationBuilder.setOAuthAccessTokenSecret(mSettings.getString("accessSecret", null));
    configurationBuilder.setOAuthAccessToken(mSettings.getString("accessToken", null));
    TwitterFactory tf = new TwitterFactory(configurationBuilder.build()).getInstance();
    mTwitter = new TwitterFactory(configuration).getInstance();
}

public Twitter getTwitterInstance() {
    ConfigurationBuilder cb = new ConfigurationBuilder();
    cb.setOAuthConsumerKey(Keys.consumerKey);
    cb.setOAuthConsumerSecret(Keys.consumerSecret);
    cb.setOAuthAccessToken(mSettings.getString("accessSecret", null));
    cb.setOAuthAccessTokenSecret(mSettings.getString("accessSecret", null));
    TwitterFactory tf = new TwitterFactory(cb.build());
    return tf.getInstance();
}

private void startOAuth() {
    ConfigurationBuilder configurationBuilder = new ConfigurationBuilder();
    configurationBuilder.setOAuthConsumerKey(Con.st.CONSUMER_KEY);
    configurationBuilder.setOAuthConsumerSecret(Con.st.CONSUMER_SECRET);
    twitter = new TwitterFactory(configurationBuilder.build()).getInstance();
    try {
        requestToken = twitter.getOAuthRequestToken(Const.CALLBACK_URL);
        Toast.makeText(this, "Please authorize this app!", Toast.LENGTH_LONG).show();
        this.startActivity(new Intent(Intent.ACTION_VIEW,
            Uri.parse(requestToken.getAuthenticationURL() + "&force_login=true")));
    } catch (TwitterException e) {
        e.printStackTrace();
    }
}

Figure 1: Three real-world usage examples of a twitter4j API pattern that sets up a twitter client with OAuth authorization.

to dynamically inferred call sequences as well as other programming languages. While we mine (possibly incomplete) API calls that are inferred statically from .java files in this paper, one can in principle extract fully resolved static or dynamic API call sequences using the BCEL bytecode library [1, 2]. The reason we did not perform dynamic call sequence extraction is that the idiosyncratic build process of most Java projects would have made compiling all 967 open-source Java projects that used our chosen libraries in our dataset (see Table 1) prohibitive. Finally, note that any API call extractor that only extracts sequences of calls cannot handle conditional statements properly, as it is trying to approximate a graph with a sequence.

4. MINING API CALL PATTERNS

In this section we will describe our novel probabilistic model for API mining. Our model is a joint probability distribution over the API patterns that the programmer actually intended to use (which we never observe) and the list of API calls in a client method (which we do). The model assumes that the set of all true API patterns is known. Then learning the API patterns becomes a search problem. Specifically, we measure the quality of a proposed set of API patterns by supposing those were the true patterns, and measure the probability that the model assigns to all client methods in the database. We search for the set of API patterns that maximizes this probability. We perform this search under the framework of a celebrated algorithm from statistics called the expectation-maximization (EM) algorithm [12], which has seen an enormous number of applications within statistics and machine learning. We use a particular variant of the EM algorithm called structural EM [14], as this deals with the specific problem of learning as search through a large combinatorial space, which in our case is the space of all possible sets of API patterns.

In the next sections, we give a high-level overview of each of the aspects of the model. For ease of exposition, we do not describe some of the more theoretical aspects of the model and the algorithm; for those, see [13].

4.1 Probabilistic Model

In this section, we describe the probabilistic model that PAM is based on. The model is a probability distribution that, based on a set of API patterns, defines a distribution over all possible API patterns present in client code. When a probabilistic model becomes more complex than one of the well-known standard families, then it is often easiest to explain by describing an algorithm to sample from it. This is how we will proceed in this section.

The model has two different types of parameters: a set of API patterns and a set of probabilities. The API patterns are straightforward: each API pattern is a sequence of API patterns and a set of probabilities. The API patterns to occur more than once in the same client method.

Therefore, for each API pattern \( S_a \) in the model, we also includes a probability distribution \( \rho_a \) which is a distribution over the integers \( 0, 1, \ldots, M \) which represents how likely a client method is to include the pattern \( S \) zero times, one time, etc. We define \( I \) to be the set of all API patterns \( S_a \) in the model. We assume that \( I \) also contains singleton sequences (\( m \)) for every method \( m \) in the API — although an API
pattern with only one method call is not very useful, so we never return such patterns to the user, we will see in Section 4.2 that these are a technical device that is necessary for the inference procedure.

Now we present an algorithm that will draw samples from our model, which we call the generative algorithm. The generative algorithm says: hypothetically speaking, if our model were correct (which we know it is not, because all models are incorrect), how would each client method be generated assuming that the API patterns and probabilities are known? We emphasize that the generative algorithm is simply an explanatory tool that helps in understanding our approach, and is never used while performing the API mining. The algorithm has two main phases: First, from the set of all informative API patterns, we sample which ones will appear in the client method that we are about to generate, and how many times they will be used, which yields a multiset that we call $S$. Then we randomly sample a way to interleave the sampled API patterns, and this results in a hypothetical client method. More formally:

1. For each unique API pattern $p$ in the set of informative API patterns $I$, decide independently the number of times $S_p$ should be included in the client API sequence $X$, i.e., draw the count $c_p$ from a suitable probability distribution over the integers, which we shall denote by $p_S$.

2. Set $S$ to be the multiset with counts $c_p$ of all the API patterns $p$ selected for inclusion in $X$, that is, $S := \{S_p : c_p \geq 1\}$. As $S$ is a multiset, a single API pattern can occur more than once in $S$.

3. Set $P$ to be the set of all possible sequences that can be generated by interleave together the API patterns in the multiset $S$, i.e.,

$$P := \{X: \text{S partition of } X, S \subset X \forall S \in S\}$$

(see the discussion below for an illustrative example).

4. Sample $X$ uniformly at random from $P$.

This algorithm defines a probability distribution over client methods, which we can sample from simply by executing it. First, let us clarify the meaning of the set $P$ through an example. To interleave two sequences $S_1$ and $S_2$, we mean the placing of items from $S_1$ into the gaps between items in $S_2$. For example, if $S_1 = (m_1, m_2)$ and $S_2 = (m_3, m_4)$, then the set of all ways to interleave $S_1$ and $S_2$ is

$$P = \{(m_3, m_4, m_1, m_2), (m_3, m_1, m_4, m_2), (m_3, m_1, m_2, m_4), (m_1, m_3, m_4, m_2), (m_1, m_2, m_3, m_4)\}.$$ 

It is possible to uniformly sample from $P$ efficiently by merging in subsequences one at a time, but we omit the details as it is unnecessary in practice.

At this point, the generative algorithm may seem contrived. Certainly we hope that developers do not write code in a manner that is anything like this. To assuage the reader’s conscience, we point out that popular methods such as the n-gram language model and latent Dirichlet allocation, which have been widely applied both to natural language and programming language text, also have generative algorithms, and those algorithms are similarly contrived. The reason that these models are useful anyway is that we are primarily interested not in the forward generative direction, in which we use API patterns to generate client methods, but in the reverse direction, in which we run the generative algorithm backward to use client methods to infer API patterns. As we will see in the moment, the backward version of this model is much more intuitive and natural.

We have defined this probability distribution implicitly using a generative algorithm. Can we now define it explicitly, by giving a formula for the probability of a client method under this algorithm? To do this, we need to introduce notation to handle the fact that our model allows the same API pattern to occur multiple times in a single client method. We will consider each occurrence of an API pattern $S$ in a client API sequence $X$ separately. That is, let $S^{(n)}$ denote the $n$-th occurrence of $S$ in $X$ i.e., by the notation $(m_1, m_2)\[3\]$ we mean “the 3rd time the API pattern $(m_1, m_2)$ occurs in a client sequence”. For example, $X = (m_1, m_2, m_3, m_1, m_2)$ contains the API patterns $(m_1, m_2)\[1\]$ and $(m_1, m_2)\[2\]$ i.e., the first and second occurrences of $(m_1, m_2)$.

In light of this $p(m_1, m_2)\[3\] \in X)$ is naturally defined as the “probability of seeing $(m_1, m_2)$ for the 3rd time given that we’ve seen it for the 2nd time”, i.e., $p(m_1, m_2)\[3\] \in X | (m_1, m_2)\[2\] \in X = p((m_1, m_2)\[3\] \in X | (m_1, m_2, m_1, m_2) \in X)$ (since we are allowing gaps in API patterns and so $(m_1, m_2)\[2\] \in X = (m_1, m_2, m_1, m_2) \in X)$. Formally, the associated probability $\pi_{S}^{(n)}$ for the $n$-th occurrence $S^{(n)}$ is simply the conditional probability of seeing $S^{(n)}$ in a client sequence $X$ given the previous occurrence $S^{(n-1)}$, i.e., $\pi_{S}^{(n)} = p(S^{(n)})/p(S^{(n-1)})$.

We also introduce the binary variable $z_{S}^{(n)} := 1_{(c_{g} \geq n)}$ to indicate whether $S^{(n)}$ is included in $X$ or not. For clarity of exposition we will drop the explicit occurrence superscript in the sequel.

Now we can give an explicit formula for the probability of a client method $X$ under our model. Given a multiset of informative API patterns $I$, let $z$, $\pi$ denote the vectors of $z$, $\pi$ for all API patterns $p \in I$. Assuming $z$, $\pi$ are fully determined, the generative model implies that the probability of generating a client API sequence $X$ is

$$p(X, z, \pi) = \left\{ \begin{array}{ll}
\prod_{p \in I} \pi_{S}^{(n)} (1 - \pi_{S})^{1 - z_{S}} & \text{if } X \in P, \\
0 & \text{otherwise.}
\end{array} \right.$$ 

Calculating $\frac{1}{|P|}$ may seem problematic, however it turns out to be rather straightforward. Pick an arbitrary ordering $s = 1, \ldots, |S|$ for the selected API pattern $S \in S$ and observe that in our iterative sampling algorithm, when merging $S_a$ into $S_b$, we have $|S_a| + 1$ points to splice $|S_a|$ elements into, that is $|S_a| + 1$ multichoose $|S_a|$, denoted $\binom{|S_a| + 1}{|S_a|} = \binom{|S_a| + |S_a|}{|S_a|}$. To see how to compute this, consider a sequence of $k = |S_a|$ stars (⋆) and $n = |S_b|$ bars (|). The number of ways that these two sequences can be spliced together is equal to the number of ways to place the $k$ stars into $n + 1$ bins delimited by the bars (this is by definition $n + 1$ multichoose $k$). This can be equivalently viewed as the number of ways to arrange the $n$ bars amongst the $k$ stars and $n$ bars (which is clearly $n + k$ choose $n$). Applying this formula iteratively we obtain

$$|P| = \prod_{s=1}^{|S|} \left(1 + \sum_{t=1}^{s} |S_s| \right) = \prod_{s=1}^{|S|} \frac{(\sum_{t=1}^{s} |S_s|)!}{|S_s|! (\sum_{t=1}^{s-1} |S_t|)!}.$$ 

(4.1)
4.2 Inference

Inference in a probabilistic model is the task of running a generative procedure backwards. In our model, this is the task of: given a client method X, infer the vector z, which indicates which API patterns were used in the method. While the generative algorithm provides a way to sample from the joint distribution \( p(X, z|\pi) \), inference is focused on the conditional distribution \( p(z|X, \pi) \). Essentially the inference procedure amounts to computing a partition of the API calls in \( X \) according to the mined API patterns.

At this point the reader may well be wondering why we need to do inference at all? For a specific client method \( X \), can’t we just look up the API patterns that \( X \) subsumes? The answer is that the mined API patterns overlap, and this is what we want, because we want to be able to learn more general and more specific versions of the same pattern. Consider the case where we have to choose between returning a more general pattern such as

- builder.<init>
- builder.setCommonProperty
- builder.build

and a more specific one such as

- builder.<init>
- builder.setCommonProperty
- builder.setRareValue
- builder.build

We want these two patterns to compete with each other to explain each client method, and the fact that we use a partitioning approach means that each API call in a client can be explained by at most one API pattern. At the end of the day, the effect is that the more specific pattern will only survive into the final list of mined patterns if it manages to be used to explain enough client methods. In other words, because we have a probability associated with each of the two API patterns, we are able to choose the more informative of the two in a well-defined and rigorous statistical way.

More formally, the inference procedure assumes that the probabilities \( \pi \) are known (we learn them in the next section). To infer the best \( z \) for a client API sequence \( X \), the natural probabilistic way of asking this question is using the conditional distribution \( p(z|X, \pi) \). Specifically, we search for the binary vector \( z \) that maximizes \( \log p(z|X, \pi) \). Sadly it can be shown that this problem is NP-hard in general. Happily this problem can be approximately solved using a simple greedy algorithm (see [13] for an explanation of why this is possible), and we find that the greedy algorithm works well in practice. The greedy algorithm repeatedly chooses an API pattern \( S \) that maximizes the improvement in \( \log \) probability divided by the number of methods in \( S \) that have not yet been explained. In order to minimize CPU time spent solving the problem, we cache the API patterns and coverings for each API client sequence as needed.

Now we can see why we have insisted on including singleton sequences for API patterns into \( I \) even though we would never return them to a user. Including the singleton sequences, and allowing them to be repeated arbitrarily many times, ensures that every client method has at least one valid partitioning.

4.3 Learning

Given a multiset of informative API patterns \( I \), consider now the case where both variables \( z, \pi \) in the model are unknown. There is an obvious chicken and egg problem here: the most informative API patterns \( S \) are determined by maximizing \( \log p(z|X, \pi) \) for \( z \), which means we need to know \( \pi \). But the probability \( \pi \) of an API pattern \( S \) depends on how often that pattern is used in a client API sequence \( X \). To get round this we can use the expectation maximization (EM) algorithm [12] which is an algorithm for estimating parameters in a model that has unobserved variables. The EM algorithm is a clever way to get around the chicken and egg problem, by iteratively solving for the best \( z \) given the current value of \( \pi \), the solving for the best value of \( \pi \). Of course, EM needs an initial guess for the unobserved variables \( \pi \) and a good first guess is simply the relative support (i.e., relative frequency of occurrence) of each API pattern \( I \) (in fact this guess is correct if all the API patterns in \( I \) are independent of each other). For the mathematical details, we refer the interested reader to Algorithm 1 in [13].

4.4 Inferring new API patterns

Now that we have shown how to learn the parameters of our probabilistic model, the astute reader may well note that we still haven’t got anywhere as we have no way of inferring which sequences to include in the set of informative API patterns \( I \). However, we can once again turn to the statistical community for a solution, in the form of the structural Expectation Maximization algorithm [14]. As the name suggests, this is a variant of the EM algorithm that lets us grow the set of informative API patterns \( I \). In particular, we can add a candidate API pattern \( S’ \) to the set of informative API patterns \( I \) if doing so improves the value of \( \log p(z|X, \pi) \) averaged across all client API sequences \( X \).

To get an estimate of maximum benefit to including candidate \( S’ \), we must carefully choose an initial value of \( \pi_{S’} \) that is not too low, to avoid getting stuck in a local optimum. To infer a good \( \pi_{S’} \), we force the candidate \( S’ \) to explain all API client sequences it supports by initializing \( \pi_{S’} = 1 \) and update \( \pi_{S’} \) with the probability corresponding to its actual usage once we have inferred all the coverings. As before, structural EM also needs an initial guess for the set of informative API calls \( I \) and associated probabilities \( \pi \) and we can simply initialize \( I \) with all the API methods in the dataset and \( \pi \) with their relative supports. Once again, we omit the mathematical details of the structural EM algorithm, but refer the interested reader to Algorithm 2 in [13].

4.5 Candidate generation

However, we are not quite done yet. The structural EM algorithm requires a method to generate new candidate sequences \( S’ \) that are to be considered for inclusion in the set of informative API patterns \( I \). One possibility would be to use the GSP frequent sequence mining algorithm to recursively suggest larger API patterns starting from all the API methods in the dataset, however preliminary experiments found this was not the most efficient method. For this reason we take a somewhat different approach and recursively combine the informative API patterns in \( I \) with the highest support first. In this way our candidate generation algorithm is more likely to propose viable candidate API patterns earlier and we find that this heuristic works well in practice. Although the algorithm is straightforward, it adds little to the exposition and we omit the details here and refer the interested reader to Algorithm 3 in [13].
Algorithm 1 Probabilistic API Miner (PAM)

Input: Client method API call sequences \(X^{(1)}, \ldots, X^{(m)}\)
Initialize \(\mathcal{I}\) with singleton sequences and \(\pi\) with their supports

while not converged do
    Add API patterns to \(\mathcal{I}, \pi\) using structural EM
    (Algorithm 2 in [13])
    Optimize parameters for \(\mathcal{I}, \pi\) using EM
    (Algorithm 1 in [13])
end while
Remove all singleton API patterns from \(\mathcal{I}\)
return \(\mathcal{I}, \pi\)

4.6 Mining Informative API Patterns

Finally, we can present our complete Probabilistic API Mining (PAM) algorithm in Algorithm 1. Note that the parameter optimization step (EM algorithm) need not be performed at every iteration, in fact it is more efficient to suggest several candidate API patterns before optimizing the parameters. As all operations on API client sequences in our algorithm are independent, and so trivially parallelizable, we perform the \(E\) and \(M\)-steps in both the EM and structural EM algorithms in parallel.

We can rank the retrieved API patterns according to their informativeness, that is how likely they are under the probabilistic model, and therefore we rank the API patterns \(S \in \mathcal{I}\) according to their probabilities \(\pi_S\) under the model.

An important property of formulating PAM as a pattern covering problem on each API client sequence is that it strongly favours adding only API patterns of associated methods, i.e., methods that predominantly co-occur in the code. Note that unlike frequent sequence mining based approaches (such as MAPO and UPMiner), our model favours co-occurrence per API client sequence and there is no assumption on frequency of co-occurrence.

5. EXPERIMENTS

In this section we perform a comprehensive evaluation of PAM at GitHub scale, comparing and contrasting it against MAPO and UPMiner. In particular, we aim to answer the following three research questions.

RQ1: Are the API patterns mined by PAM more prevalent? This research question evaluates the quality of the API call sequences mined by PAM as we would expect a set of more representative API call sequences to be more prevalent in a held-out corpus of code that was not used for mining. By performing a random 50/50 split of a suitable API call dataset we can see if sequences mined from one half of the dataset are prevalent on the other half and thus if they are representative.

RQ2: Are the API patterns mined by PAM more diverse? We would also expect a more representative set of API patterns to have lower redundancy, as a list in which every pattern uses the same few methods will be both redundant and non-diverse. A redundancy of zero is not necessarily desirable: As mentioned previously (Section 4.2), a good list of patterns may contain both more general and more specific versions of the same pattern. That said, a highly redundant list is clearly problematic.

RQ3: Are the API patterns mined by PAM useful to developers? Finally, we investigate if the mined API patterns are actually useful in practice. To do so, we look at libraries and frameworks that explicitly contain directories of API examples provided by the library’s developers. This allows us to measure whether API call sequences present in API example directories are returned by PAM, and also vice versa, that is, whether the hand-built example directories can be improved by including API patterns that are mined from client code by PAM.

Evaluation Metrics A good measure of the quality API call mining is to see what proportion of mined call sequences are contained in a set of relevant gold standard sequences, a measure we term sequence precision. This allows us to measure the degree to which the mined patterns represent relevant API call sequences. Similarly, we also define sequence recall as the proportion of relevant gold standard sequences that contain a mined call sequence. This allows us to measure the degree to which the API miner is able to retrieve relevant API call sequences. In other words, sequence precision measures the percentage of mined sequences that are somewhere used, and sequence recall measures the degree to which the mined sequences cover the usages in the gold standard data set. We present these metrics in a precision/recall curve, as is standard practice in the information retrieval literature [22]. Each point on the precision/recall curve corresponds to a different point in the ranked list of API patterns returned by each method, and indicates what the sequence precision and sequence recall would be if we forced the method to stop returning patterns at that point in the ranked list. In a precision/recall curve, being up and to the right is better, as it means that the system returns more accurate results for any fixed recall value. For MAPO and UPMiner, we rank the list of API patterns by support, whereas for PAM we rank the patterns by the probability parameter \(\pi_S\), which represents PAM’s best estimate of the probability of the pattern. As for redundancy, we measure how redundant the set of mined sequences is by calculating the average over API patterns of the number of larger API patterns that contain it (we call this no. containing sequences).

Dataset In order to assess the performance of PAM and perform a thorough comparison with MAPO and UPMiner we assemble a data set of target libraries and frameworks from the GitHub Java corpus [7]. We focus on those projects that contain an examples/ directory of code examples, so that we can compare mined patterns to those written by the library’s developers. We include in our data set all Java projects on Github that are (1) sufficiently popular, (2) imported by a sufficient number of other projects, and (3) that contain a sufficiently large examples/ directory.

More specifically, we first find all Java projects in the corpus that have an example directory (i.e., a directory matching `<example>/Example`) containing more than 10K LOC. From these projects we then select those that are in the top 1,000 projects in the corpus, ranked according to popularity. Popularity in the GitHub corpus is calculated as the sum of the number of project forks and watchers, where each is separately normalized into a z-score. From these top projects, we determine which of these are called from 50 or more methods belonging to other projects in the corpus, leaving us with the 17 projects in Table 1. We call this set of projects and associated client code the EXAMPLE dataset, to emphasize the fact that we focus on libraries and frameworks that...
include examples.

Each of these 17 projects is a library or framework, which we will call a target project, for which we wish to extract API patterns. For each target project, we perform API mining separately, and all results are reported as an average over the 17 target projects. To extract a set of client methods for each target project, we search the GitHub Java corpus for all source files which import a class belonging to the respective package or one of its subpackages. Each Example file set contains all source files that are present in the project’s example directory.

Each of these 17 projects is a library or framework, which we will call a target project, for which we wish to extract API patterns. For each target project, we perform API mining separately, and all results are reported as an average over the 17 target projects. To extract a set of client methods for each target project, we search the GitHub Java corpus for all source files which import a class belonging to the respective package or one of its subpackages and this set of files (excluding duplicates) formed the Client file set. Extracting, for each project, all source files in the aforementioned example directory formed the Example file set. Statistics on the API and Example file sets are given in Table 1.

**Experimental Setup** We implemented MAPO [39] and UP-Miner [34] based on the algorithmic descriptions provided in their respective papers. We used a clustering threshold of 40% for MAPO as this gave consistent performance and 20% for UP-Miner as this matched the natural cutoff in the dendrogram. The minimum support thresholds for both algorithms were set as low as was practically feasible for each run. We ran PAM for 10,000 iterations with a priority queue size limit of 100,000. We can therefore say with certainty that the API call sequence mined by PAM are more prevalent. Note that while the best recall that PAM achieves is 43%, this is actually close to the theoretical maximum recall on the test set. This can be approximated by the proportion of test set sequences that contain training set sequences, which is around 45%.

**RQ1: Are the API call sequences mined by PAM more prevalent?** As previously mentioned, in an attempt to answer this question we divide our dataset of API calls in half and see if sequences mined from one half of the dataset are prevalent in the other half. Specifically, we randomly divide the Client file set (cf. Table 1) into two (roughly) equal train and test sets. This enables us to mine API call subsequences from the training set and evaluate them using the sequence precision and recall metrics against the API call sequences in the test set. Figure 2 shows the sequence precision against recall, averaged across all projects in the dataset. It is evident that PAM has significantly higher precision and recall than both MAPO and UP-Miner, reaching a precision of 70%. MAPO performs especially poorly, as its precision degrades significantly as the recall increases. We can therefore say with certainty that the API call sequence mined by PAM are more prevalent. Note that while the best recall that PAM achieves is 43%, this is actually close to the theoretical maximum recall on the test set. This can be approximated by the proportion of test set sequences that contain training set sequences, which is around 45%.

**RQ2: Are the API call sequences mined by PAM more diverse?** We now turn our attention to the complete dataset and mine call sequences from the entire Client file set for each project (cf. Table 1). We can then use the number of containing sequences metric to determine how redundant the set of mined call sequences is. Figure 3 shows the average number of sequences containing other sequences in the set of top-k mined train-set sequences (ranked by support/probability) as a threshold. Each plotted curve is the interpolated precision.

![Average test-set precision against recall for PAM, MAPO and UP-Miner on the Example dataset, using the top-k mined train-set sequences (ranked by support/probability) as a threshold. Each plotted curve is the interpolated precision.](image)

Figure 2: Average test-set precision against recall for PAM, MAPO and UP-Miner on the Example dataset, using the top-k mined train-set sequences (ranked by support/probability) as a threshold. Each plotted curve is the interpolated precision.

**RQ3: Are the API patterns mined by PAM useful to...**

<table>
<thead>
<tr>
<th>Project</th>
<th>Package Name</th>
<th>Client LOC</th>
<th>Example LOC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AndEngine</td>
<td>org.andengine</td>
<td>18,274</td>
<td>19,529</td>
<td>Android 2D OpenGL game engine</td>
</tr>
<tr>
<td>Apache Camel</td>
<td>org.apache.camel</td>
<td>141,454</td>
<td>15,256</td>
<td>Enterprise application integration framework</td>
</tr>
<tr>
<td>Cloud9</td>
<td>edu.umd.cloud9</td>
<td>35,523</td>
<td>10,466</td>
<td>Cloud-based IDE</td>
</tr>
<tr>
<td>Drools</td>
<td>org.drools</td>
<td>187,809</td>
<td>15,390</td>
<td>Business rules management system</td>
</tr>
<tr>
<td>Apache Hadoop</td>
<td>org.apache.hadoop</td>
<td>1,951,653</td>
<td>26,162</td>
<td>Map-reduce framework</td>
</tr>
<tr>
<td>HornetQ</td>
<td>org.hornetq</td>
<td>30,564</td>
<td>22,541</td>
<td>Embeddable asynchronous messaging system</td>
</tr>
<tr>
<td>Apache Mahout</td>
<td>org.apache.mahout</td>
<td>48,206</td>
<td>11,772</td>
<td>Scalable machine learning environment</td>
</tr>
<tr>
<td>Neo4j</td>
<td>org.neo4j</td>
<td>239,825</td>
<td>7,710</td>
<td>Graph Database</td>
</tr>
<tr>
<td>Netty</td>
<td>io.netty</td>
<td>8,196</td>
<td>9,725</td>
<td>Network application framework</td>
</tr>
<tr>
<td>RESTEasy</td>
<td>org.jboss.resteasy</td>
<td>131,447</td>
<td>16,055</td>
<td>RESTful application framework</td>
</tr>
<tr>
<td>Restlet Framework</td>
<td>org.restlet</td>
<td>208,395</td>
<td>41,078</td>
<td>RESTful web API framework</td>
</tr>
<tr>
<td>Spring Data MongoDB</td>
<td>org.springframework</td>
<td>16,567</td>
<td>18,786</td>
<td>Spring framework MongoDB integration</td>
</tr>
<tr>
<td>Spring Data Neo4J</td>
<td>org.springframework</td>
<td>6,600</td>
<td>9,915</td>
<td>Spring framework Neo4j integration</td>
</tr>
<tr>
<td>Twitter4J</td>
<td>twitter4j</td>
<td>96,010</td>
<td>6,560</td>
<td>Twitter API</td>
</tr>
<tr>
<td>Project Wonder</td>
<td>com.webobjects</td>
<td>375,064</td>
<td>37,181</td>
<td>WebObjects frameworks</td>
</tr>
<tr>
<td>Weld</td>
<td>org.jboss.weld</td>
<td>23,298</td>
<td>9,489</td>
<td>Contexts and Dependency Injection API</td>
</tr>
<tr>
<td>Apache Wicket</td>
<td>org.apache.wicket</td>
<td>564,418</td>
<td>33,025</td>
<td>Web application framework</td>
</tr>
</tbody>
</table>

TOTAL 4,083,303 310,640

Table 1: Example dataset extracted from the GitHub Java corpus. Each row is a separate library or framework for which we mine a set of API patterns. Each Client file set contains all source files that import a class belonging the respective package or one of its subpackages. Each Example file set contains all source files that are present in the project’s example directory.
developers? First, we measure whether the mined API patterns correspond to the hand-written examples in the data set. We therefore mine, for each project, call sequences from the Client file set and evaluate them against the call sequences in the Example file set. Figure 4 shows the sequence precision against recall, averaged across all projects. Once again PAM has evidently better precision and recall than MAPO and UPMiner. The best recall achieved by PAM is 27%, significantly better than other mining methods, and for any fixed recall value, PAM has higher precision than the other methods. This supports the claim that the API patterns returned by PAM are more useful to developers than those returned by MAPO or UPMiner, because the fact that a library developer made the significant time investment necessary to provide a handwritten example is clear evidence that the he thought the example would be useful.

In an absolute sense, the level of agreement between PAM and the hand-written examples, although substantial, might not seem especially high. This raises an interesting question: Does this level of disagreement occur because the PAM patterns are not representative of the client code they were mined from, or does it occur the hand-written examples themselves are not fully representative of the client code? Although previous work has explored what it means for a single API example to be useful [38, 30], there seems to be much less work about what it means for a set of examples to be useful, and how well example directories in popular projects reflect all of the actual common uses of an API.

We can however make an initial assessment of this question by going back to the held-out test set of client code that we used in RQ1. We can measure how well the client test set and the handwritten examples agree by measuring sequence precision and recall if we take the handwritten examples as if they were API patterns and the client test set as the gold standard. When we do this, we find that the handwritten examples have a recall of 27%, meaning that three-quarters of the API sequences from the example code are not used by any client method (where “used by” means “fully contained by”). Turning to precision, the handwritten examples have a precision of 36%, meaning that two-thirds of client API method sequences are not contained within any of the handwritten examples. What makes this result perhaps surprising is that we specifically chose our data set to consist only of popular libraries with extensive handwritten examples — indeed, our data set averages 18,000 lines of example code per target API. This is significantly lower than the precision between the training set of client methods and the test set, suggesting that the training set is more representative of the test set than the handwritten examples are. Although this might be seen as a suggestive result, we caution that this result has an important threat to validity: handwritten examples may include scaffolding code that is unnecessary in client methods. For this reason, we advise caution about drawing strong conclusions from the precision of handwritten examples, but we note that this threat does not apply to the recall.

These results suggest an intriguing hypothesis, that even in extremely well documented projects with extensive sets of examples, the API usage examples written by developers are still incomplete. To get an initial indication if this intuition is likely to be correct we randomly selected three projects from our dataset and looked at the top five API patterns returned by PAM that were not present in any example call sequence. We found that the 15 selected API patterns fell into the following three categories: 4 referred to an API method not in any of the examples, 5 referred to an API class not in any of the examples and 3 referred to a more specific API pattern, i.e., an API pattern that contained additional method calls interleaved within an API example call. For the remaining 3 API patterns, our static API call extractor was unable to resolve the correct parametrized type in an API method in the pattern and thus unable to match the pattern to an example call. This provides some support for the hypothesis that the API patterns document part of the API that are used in client code but for which the original developers have not chosen to write specific examples.

Much more empirical work would be needed to fully confirm this hypothesis, so we leave this exploration to future work. If further research does indeed confirm that even ex-

---

2This figure excludes hadoop as we had problems with our implementation of MAPO and UPMiner running out of memory (hadoop has around 2 million client LOC). While PAM took 80 minutes, we excluded it for a fair comparison.
Figure 5: Top twitter4j.* API patterns mined by MAPO [39] (left), UPMiner [34] (middle), and our method PAM (right).

tensive hand-written API usage examples are incomplete, this would raise both interesting questions about why this occurs — e.g., because developers only document the most commonly used examples or because they cannot foresee the common API patterns that evolve once the API is adopted by the community — and it also would raise an interesting opportunity, that API mining tools such as PAM could be used to helping detect novel and undocumented API usages and feed them back to library and framework maintainers.

Qualitative Evaluation To provide further support to RQ3, whether the mined patterns from PAM are useful, we qualitatively compare and contrast the top sequences returned by PAM, MAPO, and UPMiner on an example target API. Figure 5 shows the top ten mined API patterns from twitter4j (ranked by probability/support) returned by PAM, MAPO and UPMiner on the EXAMPLE dataset. One can clearly see that the API calls found by MAPO are extremely repetitive, in fact most of the top ten calls are just combinations of subsequences of the following pattern:

```java
ConfigurationBuilder.<init>
ConfigurationBuilder.setOAuthConsumer
ConfigurationBuilder.build
```

which (according to our manual inspection) occurs commonly in client code but does not appear anywhere in the top ten patterns returned by MAPO. Similarly, the majority of the top ten UPMiner patterns are combinations of subsequences of the pattern:

```java
TwitterFactory.getInstance
TwitterFactory.<init>
```

but the full version of this sequence does not appear in the top ten patterns return by UPMiner. PAM on the other hand retrieves both of these full sequences within in the top ten patterns without including any redundant subsequences of this pattern. The sequences returned by PAM clearly display a more diverse selection of the API methods: The top ten PAM sequences use 19 unique API methods, compared to 10 for MAPO, and only 8 for UPMiner.

On this example, PAM tends to return longer API patterns than the other methods, which is notable because frequent sequence mining is inherently biased toward returning short patterns, because a subsequence always has higher support than a longer sequence that subsumes it. This makes it especially notable that we saw quantitatively in the previous section that on a held out test set, API patterns from PAM were more prevalent than those from MAPO or UPMiner — in general, shorter sequences should be naturally more prevalent. This further demonstrates the strength of our underlying probabilistic model of informative sequences.

6. CONCLUSIONS

We presented a parameter-free probabilistic API mining algorithm that makes use of a novel probabilistic model to infers the most informative API call patterns and demonstrated the efficacy of our approach on dataset of several hundred thousand API client files from GitHub. Through our experiments we found suggestions that API calls are not well documented in example code and in future we would like to verify this through a large-scale empirical study.
References


