Mining Idioms from Source Code

Miltiadis Allamanis, Charles Sutton
School of Informatics, University of Edinburgh
Edinburgh EH8 9AB, UK
{m.allamanis,csutton}@ed.ac.uk

ABSTRACT

We present the first method for automatically mining code idioms from a corpus of previously written, idiomatic software projects. We take the view that a *code idiom* is a syntactic fragment that recurs across projects and has a single semantic role. Idioms may have metavariables, such as the body of a `for` loop. Modern IDEs commonly provide facilities for manually defining idioms and inserting them on demand, but this does not help programmers to write idiomatic code in languages or using libraries with which they are unfamiliar. We present Haggis, a system for mining code idioms that builds on recent advanced techniques from statistical natural language processing, namely, nonparametric Bayesian probabilistic tree substitution grammars. We apply Haggis to several of the most popular open source projects from GitHub. We present a wide range of evidence that the resulting idioms are semantically meaningful, demonstrating that they do indeed recur across software projects and that they occur more frequently in illustrative code examples collected from a Q&A site. Manual examination of the most common idioms indicate that they describe important program concepts, including object creation, exception handling, and resource management.

1. INTRODUCTION

Programming language text is a means of human communication. Programmers write code not simply to be executed by a computer, but also to communicate the precise details of the code's operation to later developers who will adapt, update, test and maintain the code. It is perhaps for this reason that source code is *natural* in the sense described by Hindle et al. [18]. Programmers themselves use the term *idiomatic* to refer to code that is written in a manner that other experienced developers find natural. Programmers believe that it is important to write idiomatic code. This is evidenced simply by the amount of time that programmers spend telling other programmers how to do this. For example, Wikibooks has a book devoted to C++ idioms [51], and similar guides are available for Java [12] and JavaScript [8, 49]. A guide on GitHub for writing idiomatic JavaScript [49] has more 6,644 stars and 877 forks. A search for the keyword "idiomatic" on StackOverflow yields over 49,000 hits; all but one of the first 100 hits are questions about what the idiomatic method is for performing a given task.

The notion of *code idiom* is one that is commonly used but seldom defined. We take the view that an idiom is a syntactic fragment that recurs frequently across software projects and has a single semantic role. Idioms may have metavariables that abstract over identifier names and code blocks. For example, in Java the loop `for(int i=0;i<n;i++) { ... }` is a common idiom for iterating over an array. It is possible to express this operation in many other ways, such as a `do-while` loop or using recursion, but as experienced Java programmers ourselves, we would find this alien and more difficult to understand. Idioms differ significantly from previous notions of textual patterns in software, such as code clones [43] and API patterns [55]. Unlike clones, idioms commonly recur across projects, even ones from different domains, and unlike API patterns, idioms commonly involve syntactic constructs, such as iteration and exception handling. A large number of example idioms, all of which are automatically identified by our system, are shown in Figures 7 and 8.

Major IDEs currently support idioms by including features that allow programmers to define idioms and easily reuse them. Eclipse's SnipMatch [41] and IntelliJ IDEA live templates [22] allow the user to define custom snippets of code that can be inserted on demand. NetBeans includes a similar "Code Templates" feature in its editor. Recently, Microsoft created Bing Code Search [42] that allows users to search and add snippets to their code, by retrieving code from popular coding websites, such as StackOverflow. The fact that all major IDEs include features that allow programmers to manually define and use idioms attests to their importance.

We are unaware, however, of methods for *automatically* identifying code idioms. This is a major gap in current tooling for software development, which causes significant problems. First, software developers cannot use manual IDE tools for idioms without significant effort to organize the idioms of interest and then to manually add them to the tool. This is especially an obstacle for less experienced programmers that do not know which idioms they should be using. Indeed, as we demonstrate later, many idioms are library-specific, so even an experienced programmer will not be familiar with the code idioms for a library that they have just begun to use. Therefore, the ability to automatically identify idioms is needed.

In this paper, we present the first method for automatically mining code idioms from an existing corpus of idiomatic code. At first, this might seem to be a simple proposition: simply search for subtrees that occur often in a syntactically parsed corpus. However, this naive method does not work well, for the simple reason that frequent trees are not necessarily interesting trees. To return to our previous example, `for` loops are much more common than `for` loops that iterate over arrays, but one would be hard pressed to argue that `for(...) {...}` on its own (that is, with no expressions or

body) is an interesting pattern.

Instead, we rely on a different principle: interesting patterns are those that help to explain the code that programmers write. As a measure of "explanation quality", we use a probabilistic model of the source code, and retain those idioms that make the training corpus more likely under the model. These ideas can be formalized in a single, theoretically principled framework using a *nonparametric Bayesian* analysis. Nonparametric Bayesian methods have become enormously popular in statistics, machine learning, and natural language processing because they provide a flexible and principled way of automatically inducing a "sweet spot" of model complexity based on the amount of data that is available [39, 16, 47]. In particular, we employ a *nonparametric Bayesian tree substitution grammar*, which has recently been developed for natural language [9, 40], but which has not been applied to source code.

Because our method is primarily statistical in nature, it is language agnostic, and can be applied to any programming language for which one can collect a corpus of previously-written idiomatic code. Our major contributions are:

- We introduce the idiom mining problem (section 2);
- We present HAGGIS, a method for automatically mining code idioms based on nonparametric Bayesian tree substitution grammars (section 3);
- We demonstrate that HAGGIS successfully identifies cross-project idioms (section 5), for example, 67% of idioms that we identify from one set of open source projects also appear in an independent set of snippets of example code from the popular Q&A site StackOverflow;
- Examining the most common idioms that HAGGIS identifies (Figure 8), we find that they describe important program concepts, including object creation, exception handling, and resource management;
- To further demonstrate that the idioms identified by HAGGIS are semantically meaningful, we examine the relationship between idioms and code libraries (subsection 5.4), finding that many idioms are strongly connected to package imports in a way that can support suggestion.

2. PROBLEM DEFINITION

A *code idiom* is a syntactic fragment that recurs across software projects and serves a single semantic purpose. An example of an idiom is shown in Figure 1(b). This is an idiom which is used for manipulating objects of type `android.database.Cursor`, which ensures that the cursor is closed after use. (This idiom is indeed discovered by our method.) As in this example, typically idioms have parameters, which we will call *metavariables*, such as the name of the `Cursor` variable, and a code block describing what should be done if the `moveToFirst` operation is successful. An Android programmer who is unfamiliar with this idiom might make bad mistakes, like not calling the `close` method or not using a `finally` block, causing subtle memory leaks.

Many idioms, like the `close` example or those in Figure 8, are specific to particular software libraries. Other idioms are general across projects of the same programming language, such as those in Figure 7, including an idiom for looping over an array or an idiom defining a `String` constant. (Again, all of the idioms in these figures are discovered automatically by our method.) Idioms concerning exception handling and resource management are especially important to identify and suggest, because failure to use them correctly can cause the software to violate correctness properties. As these examples show, idioms are usually *parameterized* and the parameters often have syntactic structure, such as expressions and code blocks.

We define idioms formally as fragments of abstract syntax trees, which allows us to naturally represent the syntactic structure of an idiom. More formally, an idiom is a fragment $\mathcal{T} = (V, E)$ of an abstract syntax tree (AST), by which we mean the following. Let G be the context-free grammar of the programming language in question. Then a fragment $\mathcal{T}$ is a tree of terminals and nonterminal from G that is a subgraph of some valid parse tree from G.[1]

An idiom $\mathcal{T}$ can have as leaves both terminals and non-terminals. Non-terminals correspond to metavariables which must be filled in when instantiating the idiom. For example, in Figure 1(c), the shaded lines represent the fragment for an example idiom; notice how the `Block` node of the AST, which is a non-terminal, corresponds to a `$BODY$` metavariable in the pattern.

Idiom mining Current IDEs provide tools for manually defining idioms and inserting them when required, but this requires that the developer incur the required setup cost, and that the developer know the idioms in the first place. To eliminate these difficulties, we introduce the *idiom mining problem*, namely, to identify a set of idioms automatically given only a corpus of previously-written idiomatic code. More formally, given a training set of source files with abstract syntax trees $\mathcal{D} = \{T_1, T_2, \ldots T_N\}$, the idiom mining problem is to identify a set of idioms $\mathcal{I} = \{\mathcal{T}_i\}$ that occur in the training set. This is an *unsupervised* learning problem, as we do not assume that we are provided with any example idioms that are explicitly identified. Each fragment $\mathcal{T}_i$ should occur as a subgraph of every tree in some subset $\mathcal{D}(\mathcal{T}_i) \subseteq \mathcal{D}$ of the training corpus.

What Idioms are Not Idioms are not clones. A large amount of work in software engineering considers the problem of clone detection [43, 44], some of which considers *syntactic* clones [5, 23, 28], which find clones based on information from the AST. Clones are contiguous blocks of code that are used verbatim (or nearly so) in different code locations, usually within a project and often created via copy-paste operations. Idioms, on the other hand, typically recur *across* projects, even those from very different domains, and are used independently by many different programmers. Additionally, idioms are typically not contiguous; instead, they have metavariables that bind to expressions or entire code blocks. Finally, idioms have a semantic purpose that developers are consciously aware of. Indeed, we hypothesize that programmers chunk idioms into single mental units, and often type them in to programs directly by hand, although the psychological research necessary to verify this conjecture is beyond the scope of the current paper.

Also, idiom mining is not API mining. API mining [36, 50, 55] is an active research area that focuses on mining groups of library functions from the same API that are commonly used together. These types of patterns that are

[1] As a technicality, programming language grammars typically describe parse trees rather than AST, but as there is a 1:1 mapping between the two, we will assume that we have available a CFG that describes ASTs directly.

Figure 1: An example of how code idioms are extracted from ASTs. (a) A snippet of code from the `PhoneNumberUtils` in the GitHub project `android.telephony`. (b) A commonly occurring idiom when handling `android.database.Cursor` objects. This idiom is successfully discovered by HAGGIS. (c) A partial representation of the AST returned by the Eclipse JDT for the code in (a). The shaded nodes are those that are included in the idiom. (d) An example of a pTSG rule for a simple expression grammar. See text for more details.

inferred are essentially sequences, or sometimes finite state machines, of method invocations. Although API patterns have the potential to be extremely valuable to developers, idiom mining is a markedly different problem because idioms have syntactic structure. For example, current API mining approaches cannot find patterns such as a library with a Tree class that requires special iteration logic, or a Java library that requires the developer to free resources within a `finally` block. These are exactly the type of patterns that HAGGIS identifies.

Simple Methods Do Not Work A natural first approach to this problem is to search for AST fragments that occur frequently, for example, to return the set of all fragments that occur more than a user-specified parameter M times in the training set. This task is called frequent tree mining, and has been the subject of some work in the data mining literature [24, 48, 53, 54]. Unfortunately, our preliminary investigations [30] found that these approaches do not yield good idioms. Instead, the fragments that are returned tend to be small and generic, omitting many details that, to a human eye, are central to the idiom. For example, given the idiom in Figure 1(c), it would be typical for tree mining methods to return a fragment containing the `try`, `if`, and `finally` nodes but not the crucial method call to `Cursor.close()`.

The reason for this is simple: Given a fragment $\mathcal{T}$ that represents a true idiom, it can always be made more frequent by removing one of the leaves, even if it is strongly correlated with the rest of the tree. So tree mining algorithms will tend to return these shorter trees, resulting in incomplete idioms. In other words, *frequent patterns can be boring patterns.* To avoid this problem, we need a way of penalizing the method when it chooses *not* to extend a pattern to include a node that co-occurs frequently. This is exactly what is provided by our probabilistic approach.

3. MINING CODE IDIOMS

In this section, we introduce the technical framework that is required for HAGGIS,[2] our proposed method for the idiom mining problem. At a high level, we approach the problem of mining source code idioms as that of inferring of commonly reoccurring fragments in ASTs. But, as we have seen, simple methods of formalizing this intuition do not work (see section 2), we resort to methods that are not as simple. We apply recent advanced techniques from statistical NLP [9, 40], but we need to explain them in some detail to justify why they are appropriate for this software engineering tasks, and why technically simpler methods would not be effective.

We will build up step by step. First, we will describe two *syntactic* probabilistic models of source code, probabilistic context-free grammars and probabilistic tree substitution grammars (pTSG). We will explain why pTSGs provide a straightforward framework for augmenting a simple CFG to represent idioms. The reason that we employ *probabilistic* models here, rather than a standard deterministic CFG, is that probabilities provide a natural quantitative measure of the quality of a proposed idiom: A proposed idiom is worthwhile only if, when we include it into a pTSG, it increases the probability that the pTSG assigns to the training corpus.

At first, it may seem odd that we apply grammar learning methods here, when of course the grammar of the programming language is already known. We clarify that our aim is *not* to re-learn the known grammar, but rather to learn

[2]Holistic, Automatic Gathering of Grammatical Idioms from Software.

probability distributions over parse trees from the known grammar. These distributions will represent which rules from the grammar are used more often, and, crucially, which rules tend to be used contiguously.

The pTSG provides us with a way to *represent* idioms, but then we still need a way to *discover* them. It is for this purpose that we employ nonparametric Bayesian methods, a powerful general framework that provides methods that automatically infer from data how complex a model should be. After describing nonparametric Bayesian methods, we will finally describe how to apply nonparametric Bayesian methods to pTSGs, which requires a particular approximation known as Markov chain Monte Carlo.

3.1 Probabilistic Grammars

A *probabilistic context free grammar* (PCFG) is a simple way to define a distribution over the strings of a context-free language. A PCFG is defined as $G = (\Sigma, N, S, R, \Pi)$, where Σ is a set of terminal symbols, N a set of nonterminals, $S \in N$ is the root nonterminal symbol and R is a set of productions. Each production in R has the form $X \to Y$, where $X \in N$ and $Y \in (\Sigma \cup N)^*$. The set Π is a set of distributions $P(r|c)$, where $c \in N$ is a non-terminal, and $r \in R$ is a rule with c on its left-hand side. To sample a tree from a PCFG, we recursively expand the tree, beginning at S, and each time we add a non-terminal c to the tree, we expand c using a production r that is sampled from the corresponding distribution $P(r|c)$. The probability of generating a particular tree T from this procedure is simply the product over all rules that are required to generate T. The probability $P(x)$ of a string $x \in \Sigma^*$ is the sum of the probabilities of the trees T that yield x, that is, we simply consider $P(x)$ as a marginal distribution of $P(T)$.

Tree Substitution Grammars A tree substitution grammar (TSG) is a simple extension to a CFG, in which productions expand into tree fragments rather than simply into a list of symbols. Formally, a TSG is also a tuple $G = (\Sigma, N, S, R)$, where Σ, N, S are exactly as in a CFG, but now each production $r \in R$ takes the form $X \to \mathcal{T}_X$, where $\mathcal{T}_X$ is a fragment. To produce a string from a TSG, we begin with a tree containing only S, and recursively expanding the tree in a manner exactly analogous to a CFG — the only difference is that some rules can increase the height of the tree by more than 1. A probabilistic tree substitution grammar (pTSG) G [9, 40] augments a TSG with probabilities, in an analogous way to a PCFG. A pTSG is defined as $G = (\Sigma, N, S, R, \Pi)$ where Σ is a set of terminal symbols, N a set of non terminal symbols, $S \in N$ is the root non-terminal symbol, R is a set of tree fragment productions. Finally, Π is a set of distributions $P_{TSG}(\mathcal{T}_X|X)$, for all $X \in N$, each of which is a distribution over the set of all rules $X \to \mathcal{T}_X$ in R that have left-hand side X.

The key reason that we use pTSGs for idiom mining is that each tree fragment $\mathcal{T}_X$ can be thought of as describing a set of context-free rules that are typically used in sequence. This is exactly what we are trying to discover in the idiom mining problem. In other words, *our goal will be to induce a pTSG in which every tree fragment represents a code idiom* if the fragment has depth greater than 1, or a rule from the language's original grammar if the depth equals 1. As a simple example, consider the PCFG

$E \to E + E$	(prob 0.7)	$T \to F * F$	(prob 0.6)
$E \to T$	(prob 0.3)	$T \to F$	(prob 0.4)
$F \to (E)$	(prob 0.1)	$F \to id$	(prob 0.9),

where x and y are non-terminals, and E the start symbol. Now, suppose that we are presented with a corpus of strings from this language that include many instances of expressions like $id * (id + id)$ and $id * (id + (id + id))$ (perhaps generated by a group of students who are practicing the distributive law). Then, we might choose to add a single pTSG rule to this grammar, displayed in Figure 1(d), adjusting the probabilities for that rule and the $E \to T + T$ and $E \to T$ rules so that the three probabilities sum to 1. Essentially, this allows us to a represent a correlation between the rules $E \to T + T$ and $T \to F * F$.

Finally, note that every CFG can be written as a TSG where all productions expand to trees of depth 1. Conversely, every TSG can be converted into an equivalent CFG by adding extra non-terminals (one for each TSG rule $X \to \mathcal{T}_X$). So TSGs are, in some sense, fancy notation for CFGs. This notation will prove very useful, however, when we describe the learning problem next.

Learning TSGs Now we define the learning problem for TSGs that we will consider. First, we say that a pTSG $G_1 = (\Sigma_1, N_1, S_1, R_1, P_1)$ *extends* a CFG G_0 if every tree with positive probability under G_1 is grammatically valid according to G_0. Given any set $\mathcal{T}$ of tree fragments from G_0, we can define a pTSG G_1 that extends G_0 as follows. First, set $(\Sigma_1, N_1, S_1) = (\Sigma_0, N_0, S_0)$. Then, set $R_1 = R_{CFG} \cup R_{FRAG}$, where R_{CFG} is the set of all rules from R_0, expressed in the TSG form, i.e., with right-hand sides as trees of depth 1, and R_{FRAG} is a set of *fragment rules* $X_i \to \mathcal{T}_i$, for all $\mathcal{T}_i \in \mathcal{T}$ and where X_i is the root of $\mathcal{T}_i$.

The grammar learning problem that we consider can be called the *CFG extension problem*. The input is a set of trees $T_1 \ldots T_N$ from a context-free language with grammar $G_0 = (\Sigma_0, N_0, S_0, R_0)$. The CFG extension problem is simply to learn a pTSG G_1 that extends G_0 and is good at explaining the training set $T_1 \ldots T_N$. The notion of "good" is deliberately vague; formalizing it is part of the problem. It should also be clear that we *are not* trying to learn the CFG for the original programming language — instead, we are trying to identify sequences of rules from the known grammar that commonly co-occur contiguously.

A naïve idea is to use *maximum likelihood*, that is, to find the pTSG G_1 that extends G_0 and maximizes the probability that G_1 assigns to $T_1 \ldots T_N$. This does not work. The reason is that a trivial solution is simply to add a fragment rule $E \to T_i$ for every training tree T_i. This will assign a probability of $1/N$ to each training tree, which in practice will usually be optimal. What is going on here is that the maximum likelihood grammar is overfitting. It is not surprising that this happens: there are an infinite number of potential trees that could be used to extend G_0, so if a model is given such a large amount of flexibility, overfitting becomes inevitable. What we need is a strong method of controlling overfitting, which the next section provides.

3.2 Nonparametric Bayesian Methods

At the heart of any application of machine learning is the need to control the complexity of the model. For example,

in a clustering task, many standard clustering methods, such as K-means, require the user to pre-specify the number of clusters K in advance. If K is too small, then each cluster will be very large and not contain useful information about the data. If K is too large, then each cluster will only contain a few data points, so the again, the cluster centroid will not tell us much about the data set. For the *CFG extension problem*, the key factor that determines model complexity is the number of fragment rules that we allow for each non-terminal. If we allow the model to assign too many fragments to each non-terminal, then it can simply memorize the training set, as described in the previous section. But if we allow too few, then the model will be unable to find useful patterns. Nonparametric Bayesian methods provide a powerful and theoretically principled method for managing this trade-off.

To explain how this works, we must first explain Bayesian statistics. Bayesian statistics [15, 35] is alternative general framework to classical frequentist statistical methods such as confidence intervals and hypothesis testing. The idea behind Bayesian statistics is that whenever one wants to estimate an unknown quantity θ from a data set $x_1, x_2, \ldots x_N$, the analyst should choose a prior distribution $P(\theta)$ that encodes any prior knowledge about θ (if little is known, this distribution can be vague), and then a model $P(x_1 \ldots x_N \,|\, \theta)$. Once we define these two quantities, the laws of probability provide only one choice for how to infer θ, which is to compute the conditional distribution $P(\theta|x_1 \ldots x_N)$ using Bayes' rule. This distribution is called the *posterior distribution* and encapsulates all of the information that we have about θ from the data. Bayesian methods provide powerful general tools to combat overfitting, as the prior $P(\theta)$ can be chosen to encourage simpler models.

If θ is a finite-dimensional set of parameters, such as the mean and the variance of a Gaussian distribution, then it is easy to construct an appropriate prior $P(\theta)$. Constructing a prior becomes more difficult, however, when θ does not have a fixed number of dimensions, which is what occurs when we wish to infer the model complexity automatically. For example, consider a clustering model, where we want to learn the number of clusters. In this case, θ would be a vector containing the centroid for each cluster, but then, because before we see the data the number of clusters could be arbitrarily large, θ has unbounded dimension. As another example, in the case of the CFG extension problem, we do not know in advance how many fragments are associated with each non-terminal, and so want to infer this from data. *Nonparametric Bayesian methods* focus on developing prior distributions over infinite dimensional objects, which are then used within Bayesian statistical inference. Bayesian nonparametrics have been the subject of intense research in statistics and in machine learning, with popular models including the Dirichlet process [19] and the Gaussian process [52].

Applying this discussion to the CFG extension problem, what we are trying to infer is a pTSG, so, to apply Bayesian inference, our prior distribution must be *a probability distribution over probabilistic grammars*. We will bootstrap this from a distribution over context-free fragments, which we define first. Let G_0 be the known context-free grammar for the programming language in question. We will assume that we have available a PCFG for G_0, because this can be easily estimated by maximum likelihood from our training corpus; call this distribution over trees P_{ML}. Now, P_{ML} gives us a distribution over full trees, but what we will require is a distribution over *fragments*. We define this simply as

$$P_0(T) = P_{\text{geom}}\left(|T|, p_\$\right) \prod_{r \in T} P_{\text{ML}}(r), \tag{1}$$

where $|T|$ is the size of the fragment T, P_{geom} is a geometric distribution with parameter $p_\$$, and r ranges over the multiset of productions that are used within T.

Now we can define a prior distribution over pTSGs. Recall that we can define a pTSG G_1 that extends G_0 by specifying a set of tree fragments $\mathcal{F}_X$ for each non-terminal X. So, to define a distribution over pTSGs, we will define a distribution $P(\mathcal{F}_X)$ over the set of tree fragments rooted at X. We need $P(\mathcal{F}_X)$ to have several important properties. First, we need $P(\mathcal{F}_X)$ to have infinite support, that is, it must assign positive probability to *all possible fragments*. This is because if we do not assign a fragment positive probability in the prior distribution, we will never be able to infer it as an idiom, no matter how often it appears. Second, we want $P(\mathcal{F}_X)$ to exhibit a "rich-get-richer" effect, namely, once we have observed that a fragment $\mathcal{T}_X$ occurs many times, we want to be able predict that it will occur more often in the future.

The simplest distribution that has these properties is the Dirichlet process (DP). The Dirichlet process has two parameters: a *base measure*,[3] which in our case will be the fragment distribution P_0, and a concentration parameter $\alpha \in \mathbb{R}^+$, which controls how strong the rich-get-richer effect is. One simple way to characterize the Dirichlet process is the *stick-breaking* representation [45]. Using this representation, a Dirichlet process defines a distribution over $\mathcal{F}_X$ as

$$\Pr[\mathcal{T} \in \mathcal{F}_X] = \sum_{k=1}^{\infty} \pi_k \delta_{\{\mathcal{T}=\mathcal{T}_k\}} \qquad \mathcal{T}_k \sim P_0$$
$$u_k \sim \text{Beta}(1, \alpha) \qquad \pi_k = (1-u_k) \prod_{j=1}^{k-1} u_j.$$

To interpret this, recall that the symbol $\sim$ is read "is distributed as," and the Beta distribution is a standard distribution over the set $[0, 1]$; as α becomes large, the mean of a $\text{Beta}(1, \alpha)$ distribution will approach 1. Intuitively, what is going on here is that a sample from the DP is a distribution over a countably infinite number of fragments $\mathcal{T}_1, \mathcal{T}_2, \ldots$. Each one of these fragments is sampled independently from the fragment distribution P_0. To assign a probability to each fragment, we recursively split the interval $[0, 1]$ into a countable number of sticks $\pi_1, \pi_2, \ldots$. The value $(1 - u_k)$ defines what proportion of the remaining stick is assigned to the current sample $\mathcal{T}_k$, and the remainder is assigned to the infinite number of remaining trees $\mathcal{T}_{k+1}, \mathcal{T}_{k+2}, \ldots$. This process defines a distribution over fragments $\mathcal{F}_X$ for each non-terminal X, and hence a distribution $P(G_1)$ over the set of all pTSGs that extend G_0. We will refer to this distribution as a *Dirichlet process probabilistic tree substitution grammar* (DPpTSG) [40, 9].

This process may seem odd for two reasons: (a) each sample from $P(G_1)$ is infinitely large, so we cannot store it exactly on a computer, (b) the fragments from G_1 are sampled randomly from a PCFG, so there is no reason to think that they should match real idioms. Fortunately, the

[3]The base measure will be a probability measure, so for our purposes here, we can think of this as a fancy word for "base distribution".

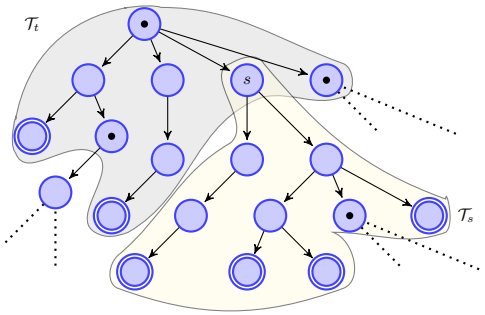


Figure 2: Sampling an AST. Nodes with dots show the points where the tree is split (*i.e.* $z_t = 1$). Nodes with double border represent terminal nodes.

answer to both these concerns is simple. We are *not* interested in the fragments that exist in the prior distribution, but rather of those in the posterior distribution. More formally, the DP provides us with a prior distribution G_1 over pTSGs. But G_1 itself, like any pTSG, defines a distribution $P(T_1, T_2, \ldots T_N|G_1)$ over the training set. So, just as in the parametric case, we can apply Bayes's rule to obtain a posterior distribution $P(G_1|T_1, T_2, \ldots T_N)$. It can be shown that this distribution is also a DPpTSG, and, amazingly, that this posterior DPpTSG can be characterized by a *finite* set of fragments $\mathcal{F}'_X$ for each non-terminal. It is these fragments that we will identify as code idioms (section 4).

3.3 Inference

What we have just discussed is how to define a posterior distribution over grammars that will infer code idioms. But we still need to describe how to *compute* this distribution. Unfortunately, the posterior distribution cannot be computed exactly, so we resort to approximations. The most commonly used approximations in the literature are based on Markov chain Monte Carlo (MCMC), which we explain below. But first, we make one more observation about pTSGs. All of the pTSGs that we consider are extensions of an unambiguous base CFG G_0. This means that given a source file F, we can separate the pTSG parsing task into two steps: first, parse F using G_0, resulting in a CFG tree T; second, group the nodes in T according to which fragment rule in the pTSG was used to generated them. We can represent this second task as a tree of binary variables z_s for each node s. These variables indicate whether the node s is the root of a new fragment ($z_s = 1$), or if node s is part of the same fragment as its parent ($z_s = 0$). Essentially, the variables z_s show the boundaries of the inferred tree patterns; see Figure 2 for an example. Conversely, even if we don't know what fragments are in the grammar, given a training corpus that has been parsed in this way, we can use the z_s variables to read off what fragments must have been in the pTSG.

With this representation in hand, we are now ready to present an MCMC method for sampling from the posterior distribution over grammars, using a particular method called Gibbs sampling. Gibbs sampling is an iterative method, which starts with an initial value for all of the z variables, and then updates them one at a time. At each iteration, the sampler visits every tree node t of every tree in the training corpus, and samples a new value for z_t. Let s be the parent of t. If we choose $z_t = 1$, we can examine the current values of the z variables to determine the tree fragment $\mathcal{T}_t$ that contains t and the fragment $\mathcal{T}_s$ for s, which must be disjoint. On the other hand, if we set $z_t = 0$, then s and t will belong to the same fragment, which will be exactly $\mathcal{T}_{\text{join}} = \mathcal{T}_s \cup \mathcal{T}_t$. Now, we set z_t to 0 with probability

$$P(z_t = 0) = \frac{P_{\text{post}}(\mathcal{T}_{\text{join}})}{P_{\text{post}}(\mathcal{T}_{\text{join}}) + P_{\text{post}}(\mathcal{T}_s)P_{\text{post}}(\mathcal{T}_t)}. \quad (2)$$

where

$$P_{\text{post}}(\mathcal{T}) = \frac{\mathsf{count}(\mathcal{T}) + \alpha P_0(\mathcal{T})}{\mathsf{count}(h(\mathcal{T})) + \alpha}, \quad (3)$$

h returns the root of the fragment, and count returns the number of times that a tree occurs as a fragment in the corpus, as determined by the current values of z. Intuitively, what is happening here is that if the fragments $\mathcal{T}_s$ and $\mathcal{T}_t$ occur very often together in the corpus, relative to the number of times that they occur independently, then we are more likely to join them into a single fragment.

It can be shown that if we repeat this process for a large number of iterations, eventually the resulting distribution over fragments will converge to the posterior distribution over fragments defined by the DPpTSG. It is these fragments that we return as idioms.

We present the Gibbs sampler because it is a useful illustration of MCMC, but in practice we find that it converges too slowly to scale to large codebases. Instead we use the type-based MCMC sampler of Liang *et al.* [32] (details omitted).

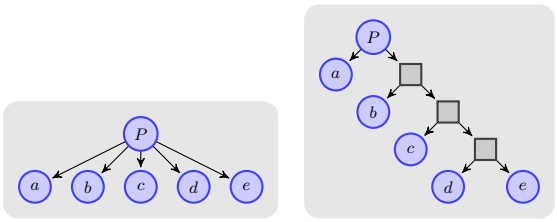
4. SAMPLING A TSG FOR CODE

Hindle *et al.* [18] have shown that source code presents some of the characteristics of natural language. HAGGIS exploits this fact by using pTSGs — originally devised for natural language — to infer code idioms. Here, we describe a set of necessary transformations to ASTs and pTSG to adapt these general methods specifically to the task of inferring code idioms.

AST Transformation For each `.java` file we use the Eclipse JDT [11] to extract its AST — a tree structure of `ASTNode` objects. Each `ASTNode` object contains two sets of properties: *simple properties* — such as the type of the operator, if `ASTNode` is an infix expression — and *structural properties* that contain zero or more child `ASTNode` objects. First, we construct the grammar symbols by mapping each `ASTNode`'s type and simple properties into a (terminal or non-terminal) symbol. The transformed tree is then constructed by mapping the original AST into a tree whose nodes are annotated with the symbols. Each node's children are grouped by property.

The transformed trees may contain nodes that have more than two children for a single property (*e.g.* `Block`). This induces unnecessary sparsity in the CFG and TSG rules. To reduce this sparsity, we perform *tree binarization*. This process — common in NLP — transforms the original tree into binary by adding dummy nodes, making the data less sparse. It will also help us capture idioms in sequential statements. Note that binarization is performed *only* on structural properties that have two or more children, while an arbitrary node may have more than two children among its properties.

One final hurdle for learning meaningful code idioms are variable names. Since variable names are mostly project or



(a) No binarization (b) Binarized

Figure 3: Tree Binarization for nodes with multiple children. Square nodes represent the dummy nodes added.

class specific we abstract them introducing an intermediate `MetaVariable` node between the `SimpleName` node containing the string representation of the variable name and its parent node. `MetaVariable` nodes are also annotated with the type of the variable they are abstracting. This provides the pTSG with the flexibility to either exclude or include variable names as appropriate. For example, in the snippet of Figure 1(a) by using metavariables, we are able to learn the idiom in Figure 1(b) without specifying the name of the `Cursor` object by excluding the `SimpleName` nodes from the fragment. Alternatively, if a specific variable name is common and idiomatic, such as the `i` in a `for` loop, the pTSG can choose to include `SimpleName` in the extracted idiom, by merging it with its parent `MetaVariable` node.

Training TSGs and Extracting Code Idioms Training a pTSG happens offline, during a separate training phase. After training the pTSG, we then extract the mined code idioms which then can be used for any later visualization. In other words, a user of a HAGGIS IDE tool would never need to wait for a MCMC method to finish. The output of a MCMC method is a series of (approximate) samples from the posterior distribution, each of which in our case, is a single pTSG. These sampled pTSGs need to be post-processed to extract a single, meaningful set of code idioms. First, we aggregate the MCMC samples after removing the first few samples as *burn-in*, which is standard methodology for applying MCMC. Then, to extract idioms from the remaining samples, we merge all samples' tree fragments into a single multiset. We then prune the multiset by removing all tree fragments that have been seen less than c_{min} times to ensure that the mined tree fragments are frequent enough. We also prune fragments that have fewer that n_{min} nodes to get a set of non-trivial (*i.e.* sufficiently large) code idioms. Finally, we reconvert the fragments back to Java code. The leaf nodes of the fragments that contain non-terminal symbols represent metavariables and are converted to the appropriate symbol that is denoted by a $ prefix.

Additionally, to assist the sampler in inducing meaningful idioms, we prune any `import` statements from the corpus, so that they cannot be mined as idioms. We also exclude some nodes from sampling, fixing $z_i = 0$ and thus forcing some nodes to be un-splittable. Such nodes include method invocation arguments, qualified and parametrized type node children, non-block children of `while`, `for` and `if` statement nodes, parenthesized, postfix and infix expressions and variable declaration statements.

5. CODE SNIPPET EVALUATION

We take advantage of the omnipresence of idioms in source code to evaluate HAGGIS on popular open source projects. We restrict ourselves to the Java programming language, due to the high availability of tools and source code. We emphasize, however, that HAGGIS is language agnostic. Before we get started, an interesting way to get an intuitive feel for any probabilistic model is simply to draw samples from it. Figure 4 shows a code snippet that we synthetically generated by sampling from the posterior distribution over code defined by the pTSG. One can observe that the pTSG is learning to produce idiomatic and syntactically correct code, although — as expected — the code is semantically inconsistent.

```
try {
  regions=computeProjections(owner);
} catch (RuntimeException e) {
  e.printStackTrace();
  throw e;
}
if (elem instanceof IParent) {
  IJavaElement[] children=((IParent)owner).getChildren();
  for (int fromPosition=0; i < children.length; i++) {
    IJavaElement aChild=children[i];
    Set childRegions=findAnnotations(aChild,result);
    removeCollisions(regions,childRegions);
  }
}
constructAnnotations(elem,result,regions);
```

Figure 4: Synthetic code randomly generated from a posterior pTSG. One can see that the pTSG produces code that is syntactically correct and locally consistent. This effect allows us to infer code idioms. It can be seen that, as expected, the pTSG cannot capture higher level information, such as variable binding.

Methodology We use two evaluation datasets comprised of Java open-source code available on GitHub. The PROJECTS dataset (Figure 5) contains the top 13 Java GitHub projects whose repository is at least 100MB in size according to the GitHub Archive [17]. To determine popularity, we computed the z-score of forks and watchers for each project. The normalized scores were then averaged to retrieve each project's popularity ranking. The second evaluation dataset, LIBRARY (Figure 6), consists of Java classes that import (*i.e.* use) 15 popular Java libraries. For each selected library, we retrieved from the Java GitHub Corpus [2] all files that import that library but do not implement it. We split both datasets into a train and a test set, splitting each project in PROJECTS and each library fileset in LIBRARY into a train (70%) and a test (30%) set. The PROJECTS will be used to mine project-specific idioms, while the LIBRARY will be used to mine idioms that occur across libraries.

To extract idioms we run MCMC for 100 iterations for each of the projects in PROJECTS and each of library filesets in the LIBRARY allowing sufficient burn-in time of 75 iterations. For the last 25 iterations, we aggregate a sample posterior pTSG and extract idioms as detailed in section 4. A threat to the validity of the evaluation using the aforementioned datasets is the possibility that the datasets are not representative of Java development practices, containing solely open-source projects from GitHub. However, the selected datasets span a wide variety of domains, including databases, messaging systems and code parsers, diminishing any such possibility. Furthermore, we perform an extrinsic evaluation on source code found on a popular online Q&A website, StackOverflow.

Name	Forks	Stars	Files	Commit	Description
arduino	2633	1533	180	2757691	Electronics Prototyping
atmosphere	1606	370	328	a0262bf	WebSocket Framework
bigbluebutton	1018	1761	760	e3b6172	Web Conferencing
elasticsearch	5972	1534	3525	ad547eb	REST Search Engine
grails-core	936	492	831	15f9114	Web App Framework
hadoop	756	742	4985	f68ca74	Map-Reduce Framework
hibernate	870	643	6273	d28447e	ORM Framework
libgdx	2903	2342	1985	0c6a387	Game Dev Framework
netty	2639	1090	1031	3f53ba2	Net App Framework
storm	1534	7928	448	cdb116e	Distributed Computation
vert.x	2739	527	383	9f79416	Application platform
voldemort	347	1230	936	9ea2e95	NoSQL Database
wildfly	1060	1040	8157	043d7d5	Application Server

Figure 5: PROJECTS dataset used for in-project idiom evaluation. Projects in alphabetical order.

Package Name	Files	Description
android.location	1262	Android location API
android.net.wifi	373	Android WiFi API
com.rabbitmq	242	Messaging system
com.spatial4j	65	Geospatial library
io.netty	65	Network app framework
opennlp	202	NLP tools
org.apache.hadoop	8467	Map-Reduce framework
org.apache.lucene	4595	Search Server
org.elasticsearch	338	REST Search Engine
org.eclipse.jgit	1350	Git implementation
org.hibernate	7822	Persistence framework
org.jsoup	335	HTML parser
org.mozilla.javascript	1002	JavaScript implementation
org.neo4j	1294	Graph database
twitter4j	454	Twitter API

Figure 6: LIBRARY dataset for cross-project idiom evaluation. Each API fileset contains all class files that `import` a class belonging to the respective package or one of its subpackages.

Evaluation Metrics We compute two metrics on the test corpora. These two metrics resemble precision and recall in information retrieval but are adjusted to the code idiom domain. We define *idiom coverage* as the percent of source code AST nodes that can be matched to the mined idioms. Coverage is thus a number between 0 and 1 indicating the extent to which the mined idioms exist in a piece of code. We define *idiom set precision* as the percentage of the mined idioms found in the test corpus. This metric shows the precision of mined set of idioms. Using these two metrics, we also tune the concentration parameter of the DPpTSG model by using `android.net.wifi` as a validation set, yielding $\alpha = 1$.

5.1 Top Idioms

Figure 8 shows the top idioms mined in the LIBRARY dataset, ranked by the number of files in the test sets where each idiom has appeared in. The reader will observe their immediate usefulness. Some idioms capture how to retrieve or instantiate an object. For example, in Figure 8, the idiom 8a captures the instantiation of a message channel in RabbitMQ, 8r retrieves a handle for the Hadoop file system, 8e builds a `SearchSourceBuilder` in Elasticsearch and 8l retrieves a URL using JSoup. Other idioms capture important transactional properties of code: idiom 8h uses properly the memory-hungry `RevWalk` object in JGit and 8i is a transaction idiom in Neo4J. Other idioms capture common error handling, such as 8d for Neo4J and 8p for a Hibernate transaction. Finally, some idioms capture common operations, such as closing a connection in Netty (8m), traversing through the database nodes (8n), visiting all AST nodes in a JavaScript file in Rhino (8k) and computing the distance between two locations (8g) in Android. The reader may observe that these idioms provide a meaningful set of coding patterns for each library, capturing semantically consistent actions that a developer is likely to need when using these libraries.

In Figure 7 we present a small set of Java-related idioms mined across all datasets. These idioms represent frequently used code patterns that would be included by default in tools such as Eclipse's SnipMatch [41] and IntelliJ's live templates [22]. Defining constants (Figure 7c), creating loggers (Figure 7b) and iterating through an iterable (Figure 7a) are some of the most common language-specific idioms in Java. All of these idioms have been automatically identified by HAGGIS.

```
for (Iterator iter=$methodInvoc;
    iter.hasNext(); )
  {$BODY$}
```

(a) Iterate through the elements of an Iterator.

```
private final static Log $name=
  LogFactory.getLog($type.class);
```

(b) Creating a logger for a class.

```
public static final
  String $name = $StringLit;
```

(c) Defining a constant String.

```
while (($(String) = $(BufferedReader).
    readLine()) != null) {
  $BODY$
}
```

(d) Looping through lines from a `BufferedReader`.

Figure 7: Sample Java-language idioms. `$stringLit` denotes a user-defined string literal, `$name` a freely defined (variable) name, `$methodInvoc` a single method invocation statement, `$ifstatement` a single `if` statement and `$BODY$` denotes a user-defined code block of one or more statements.

	Name	Precision (%)		Coverage (%)		Avg Size (#Nodes)	
LIBRARY	HAGGIS $n_{min} = 5$, $c_{min} = 2$	8.5	±3.2	23.5	±13.2	15.0	±2.1
	HAGGIS $n_{min} = 20$, $c_{min} = 25$	16.9	±10.1	2.8	±3.0	27.9	±8.63
	DECKARD minToks=10, stride=2, sim=1	0.9	±1.3	4.1	±5.24	24.6	±15.0
PROJECTS	HAGGIS $n_{min} = 5$, $c_{min} = 2$	14.4	±9.4	30.29	±12.5	15.46	±3.1
	HAGGIS $n_{min} = 20$, $c_{min} = 25$	29.9	±19.4	3.1	±2.6	25.3	±3.5

Figure 9: Average and standard deviation of performance in LIBRARY test set. Standard deviation across projects.

```
channel=connection.
  createChannel();
```

(a)

```
Elements $name=$(Element).
  select($StringLit);
```

(b)

```
Transaction tx=ConnectionFactory.
    getDatabase().beginTx();
```

(c)

```
catch (Exception e) {
  $(Transaction).failure();
}
```

(d)

```
SearchSourceBuilder builder=
  getQueryTranslator().build(
    $(ContentIndexQuery));
```

(e)

```
LocationManager $name =
  (LocationManager)getSystemService(
    Context.LOCATION_SERVICE);
```

(f)

```
Location.distanceBetween(
    $(Location).getLatitude(),
    $(Location).getLongitude(),
    $...);
```

(g)

```
try {
  $BODY$
} finally {
  $(RevWalk).release();
}
```

(h)

```
try {
  Node $name=$methodInvoc();
  $BODY$
} finally {
  $(Transaction).finish();
}
```

(i)

```
ConnectionFactory factory =
  new ConnectionFactory();
$methodInvoc();
Connection connection =
  factory.newConnection();
```

(j)

```
while ($(ModelNode) != null) {
  if ($(ModelNode) == limit)
    break;
  $ifstatement
  $(ModelNode)=$(ModelNode)
    .getParentModelNode();
}
```

(k)

```
Document doc=Jsoup.connect(URL).
  userAgent("Mozilla").
  header("Accept","text/html").
  get();
```

(l)

```
if ($(Connection) != null) {
  try {
    $(Connection).close();
  } catch (Exception ignore) { }
}
```

(m)

```
Traverser traverser
  =$(Node).traverse();
for (Node $name : traverser) {
  $BODY$
}
```

(n)

```
Toast.makeText(this,
  $stringLit,Toast.LENGTH_SHORT)
    .show()
```

(o)

```
try {
  Session session
    =HibernateUtil
      .currentSession();
  $BODY$
} catch (HibernateException e) {
  throw new DaoException(e);
}
```

(p)

```
catch (HibernateException e) {
  if ($(Transaction) != null) {
    $(Transaction).rollback();
  }
  e.printStackTrace();
}
```

(q)

```
FileSystem $name
  =FileSystem.get(
    $(Path).toUri(),conf);
```

(r)

```
(token=$(XContentParser).nextToken())
  != XContentParser.Token.END_OBJECT
```

(s)

Figure 8: Top cross-project idioms for LIBRARY projects (Figure 5). Here we include idioms that appear in the test set files. We rank them by the number of distinct files they appear in and restrict into presenting idioms that contain at least one library-specific (*i.e.* API-specific) identifier. The special notation `$(TypeName)` denotes the presence of a variable whose name is undefined. `$BODY$` denotes a user-defined code block of one or more statements, `$name` a freely defined (variable) name, `$methodInvoc` a single method invocation statement and `$ifstatement` a single `if` statement. All the idioms have been automatically identifies by HAGGIS

We now quantitatively evaluate the mined idiom sets. Figure 9 shows idiom coverage, idiom set precision and the average size of the matched idioms in the test sets of each dataset. We observe that HAGGIS achieves better precision and coverage in PROJECTS. This is expected since code idioms recur more often in a similar project rather than across disparate projects. This effect may be partially attributed to the small number of people working in a project and partially to project-specific idioms. Figure 9 also gives an indication of the trade-offs we can achieve for different c_{min} and n_{min}.

5.2 Code Cloning vs Code Idioms

Previously, we discussed that code idioms differ significantly from code clones. We now show this by using a cutting-edge code clone detection tool: DECKARD [23] is a state-of-the-art tree-based clone-detection tool that uses an intermediate vector representation for detecting similarities. To extract code idioms from the code clone clusters that DECKARD computes, we retrieve the maximal common subtree of each cluster, ignoring patterns that are less that 50% of the original size of the tree.

We run DECKARD with multiple parameters (stride $\in \{0, 2\}$, similarity $\in \{0.95, 1.0\}$, minToks $\in \{10, 20\}$) on the validation set and picked the parameters that achieve the best combination of precision and coverage. Figure 9 shows precision, coverage and average idiom size (in number of nodes) of the patterns found through DECKARD and HAGGIS. HAGGIS found larger and higher coverage idioms, since clones seldom recur across projects. The differences in precision and coverage are statistically significant (paired t-test; $p < 0.001$). We also note that the overlap in the patterns extracted by DECKARD and HAGGIS is small (less than 0.5%).

It is important to note these results are not a criticism of DECKARD—which *is* a high-quality, state-of-the-art code clone detection tool—but rather, these results show that *the task of code clone detection is different from code idiom mining*: Code clone detection is concerned with finding pieces of code that are not necessarily frequent but are maximally identical. In contrast, idiom mining is not concerned with finding maximally identical pieces of code, but mining common tree fragments that trade-off between size and frequency.

5.3 Extrinsic Evaluation of Mined Idioms

Now, we evaluate HAGGIS extrinsically by computing coverage and precision in the test sets of each dataset and the StackOverflow question dataset [4], an extrinsic set of highly idiomatic code snippets. StackOverflow is a popular Q&A site containing programming-related questions and answers. When developers deem that their question or answer needs to be clarified with code, they include a code snippet. These snippets are representative of general development practice and are usually short, concise and idiomatic, containing only essential pieces of code. We first extract all code fragments in questions and answers tagged as java or android, filtering only those that can be parsed by Eclipse JDT [11]. We further remove snippets that contain less than 5 tokens. After this process, we have 108,407 partial Java snippets. Then, we create a single set of idioms, merging all those found in LIBRARY and removing any idioms that have been seen in less than five files at the test portions of LIBRARY. We end up with small but high precision set of idioms across all APIs in LIBRARY.

Figure 10 shows precision and coverage of HAGGIS's idioms comparing StackOverflow, LIBRARY and PROJECTS. Using the LIBRARY idioms, we achieve a coverage of 31% and a precision of 67% on StackOverflow, compared to a much smaller precision and coverage in PROJECTS. This shows that the mined idioms are more frequent in StackOverflow than in a "random" set of projects. Since we expect that StackOverflow snippets are more highly idiomatic than average projects' source code, this provides strong indication that HAGGIS has mined a set of meaningful idioms. We note that precision depends highly on the popularity of LIBRARY's libraries. For example, because Android is one of the most popular topics in StackOverflow, when we limit the mined idioms to those found in the two Android libraries, HAGGIS achieves a precision of 96.6% at a coverage of 21% in StackOverflow. This evaluation provides a strong indication that HAGGIS idioms are widely used in development practice.

Test Corpus	Coverage	Precision
StackOverflow	31%	67%
PROJECTS	22%	50%

Figure 10: Extrinsic evaluation of mined idioms. All idioms were mined from LIBRARY.

5.4 Idioms and Code Libraries

Previously, we found code idioms across projects and libraries. As a final evaluation of the mined code idioms' semantic consistency, we now show that code idioms are highly correlated with the packages that are imported by a Java file. We merge the idioms across our LIBRARY projects and visualize the *lift* among code idioms and `import` statements. Lift, commonly used in association rule mining, measures how dependent the co-appearance of two elements is. For each imported package p, we compute the lift score l of the code idiom t as $l(p,t) = P(p,t)/\left(P(p)P(t)\right)$ where $P(p)$ is the probability of importing package p, $P(t)$ is the probability of the appearance of code idiom t and $P(p,t)$ is the probability that package p and idiom t appear together. It can be seen that $l(p,t)$ is higher as package p and idiom t are more correlated, *i.e.*, their appearance is not independent.

Figure 11 shows a covariance-like matrix of the lift of the top idioms and packages. Here, we visualize the top 300 most frequent train set packages and their highest correlating code idioms, along with the top 100 most frequent idioms in LIBRARY. Each row represents a single code idiom and each column a single package. On the top of Figure 11 one can see idioms that do not depend strongly on the package imports. These are language-generic idioms (such as the exception handling idiom in Figure 7c) and do not correlate significantly with any package. We can also observe dark blocks of packages and idioms. Those represent library or project-specific idioms that co-appear frequently. This provides additional evidence that HAGGIS finds meaningful idioms since, as expected, some idioms are common throughout Java, while others are API or project-specific.

Suggesting idioms To further demonstrate the semantic consistency of the HAGGIS idioms, we present a preliminary approach to suggesting idioms based on package imports. We caution that our goal here is to develop an initial proof of concept, not the best possible suggestion method. First, we score each idiom $\mathcal{T}_i$ by computing $s(\mathcal{T}_i|\mathbb{I}) = \max_{p\in\mathbb{I}} l(p, \mathcal{T}_i)$ where $\mathbb{I}$ is the set of all imported packages. We then return a ranked list $\mathbb{T}_\mathbb{I} = \{\mathcal{T}_1, \mathcal{T}_2, \dots\}$ such that for all $i < j$,

Figure 11: Lift between package imports and code idioms. A darker color signifies higher lift, *i.e.* more common co-occurrence. Each row shows the "spectrum" of an idiom. Darker blue color shows higher correlation between a package and an idiom. One can find idioms generic-language idioms (top) and others that are package-specific (dark blocks on the right). Idioms and packages are only shown for the `android.location`, `android.net.wifi` and `org.hibernate` APIs for brevity.

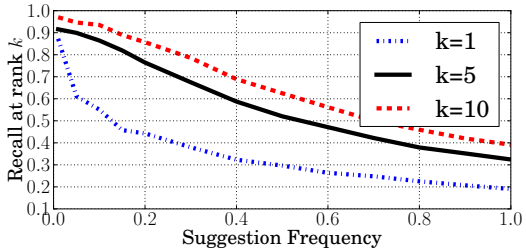


Figure 12: The recall at rank k for code idiom suggestion.

$s(\mathcal{T}_i, \mathbb{I}) > s(\mathcal{T}_j, \mathbb{I})$. Additionally, we use a threshold s_{th} to control the precision of the returned suggestions, showing only those idioms t_i that have $s(\mathcal{T}_i, \mathbb{I}) > s_{th}$. Thus, we are only suggesting idioms where the level of confidence is higher than s_{th}. It follows, that this parameter controls suggestion frequency *i.e.* the percent of the times where we present at least one code idiom.

To evaluate HAGGIS's idiom suggestions, we use the LIBRARY idioms mined from the train set and compute the recall-at-rank-k on the LIBRARY's test set. Recall-at-rank-k evaluates HAGGIS's ability to return at least one code idiom for each test file. Figure 12 shows that for suggestion frequency of 20% we can achieve a recall of 76% at rank $k = 5$, meaning that in the top 5 results we return at least one relevant code idiom 76% of the time. This results shows the quality of the mined idioms, suggesting that HAGGIS can provide a set of meaningful suggestions to a developer by solely using the code's imports. Further improvements in suggestion performance can be achieved by using more advanced classification methods, which we leave to future work, which could eventually enable an IDE side-pane that presents a list of suggested code idioms.

6. RELATED WORK

Source code has been shown to be highly repetitive and non-unique [14] rendering NLP methods attractive for the analysis of source code. N-gram language models have been used [2, 18, 37] to improve code autocompletion performance, learn coding conventions [3] and find syntax errors [7]. Models of the tree structure of the code have also been studied with the aim of generating programs by example [34] and modeling source code [33]. However, none of this work has tried to extract non-sequential patterns in code or mine tree fragments. The only work that we are aware of that uses language models for detecting textual patterns in code is Jacob and Tairas [21] that use n-grams to autocomplete code templates.

Code clones [10, 25, 26, 27, 31, 43, 44] are related to code idiom mining, since they aim to find highly similar code, but not necessarily identical pieces of code. Code clone detection using ASTs has also been studied extensively [6, 13, 23, 29]. For a survey of clone detection methods, see Roy *et al.* [43, 44]. In contrast, as we noted in section 5, code idiom mining searches for frequent, rather than maximally identical subtrees. It is worth noting that code clones have been found to have a positive effect on maintenance [26, 27]. Another related area is API mining [1, 20, 55, 50]. However, this area is also significantly different from code idiom mining because it tries to mine sequences or graphs [36] of API method calls, usually ignoring most features of the language. This difference should be evident from the sample code idioms in Figure 8.

Within the data mining literature, there has been a series of work on *frequent tree mining* algorithms [24, 48, 53, 54], which focuses on finding subtrees that occur often in a database of trees. However, as described in section 2, these have the difficulty that frequent trees are not always interesting trees, a difficulty which our probabilistic approach addresses in a principled way. Finally, as described previously, Bayesian nonparametric methods are a widely researched area in statistics and machine learning [19, 16, 47, 39], which have also found many applications in NLP [46, 9, 40].

7. DISCUSSION & CONCLUSIONS

In this paper, we presented HAGGIS, a system for automatically mining high-quality code idioms. We found that code idioms appear in multiple settings: some are project-specific, some are API-specific and some are language-specific. An interesting direction for future work is to study the reasons that code idioms arise in programming languages, APIs or projects and the effects idioms have in the software engineering process. It could be that there are "good" and "bad" idioms. "Good" idioms may arise as an additional abstraction layer over a programming language that helps developers communicate more clearly the intention of their code. "Bad" idioms may compensate for deficiencies of a programming language or an API. For example, one common Java idiom mined by HAGGIS is a sequence of multiple catch statements. This idiom is indeed due to Java's language design, that led Java language designers to introduce a new "multi-catch" statement in Java 7 [38]. However, other idioms, such as the ubiquitous `for(int i=0;i<n;i++)` cannot be considered a

language limitation, but rather a useful and widely understandable code idiom. A more formal study of the difference between these two types of idioms could be of significant interest.

Acknowledgments

The authors would like to thank Jaroslav Fowkes, Sharon Goldwater and Mirella Lapata for their insightful comments and suggestions. This work was supported by Microsoft Research through its PhD Scholarship Programme and by the Engineering and Physical Sciences Research Council [grant number EP/K024043/1].

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34. ACM, 2007.

[2] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *MSR*, 2013.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. *arXiv preprint arXiv:1402.4182*, 2014.

[4] A. Bacchelli. Mining challenge 2013: StackOverflow. In *The 10th Working Conference on Mining Software Repositories*, 2013.

[5] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49, 1993.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.

[7] J. Campbell, A. Hindle, and J. N. Amaral. Syntax errors just aren't natural: Improving error reporting with language models.

[8] S. Chuan. JavaScript Patterns Collection. `http://shichuan.github.io/javascript-patterns/`, 2014. Visited Feb 2014.

[9] T. Cohn, P. Blunsom, and S. Goldwater. Inducing tree-substitution grammars. *The Journal of Machine Learning Research*, 9999:3053–3096, 2010.

[10] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 214–225. ACM, 2008.

[11] Eclipse-Contributors. Eclipse JDT. `eclipse.org/jdt`, 2014. Visited Mar 2014.

[12] J. I. Editors. Java Idioms. `http://c2.com/ppr/wiki/JavaIdioms/JavaIdioms.html`, 2014. Visited Feb 2014.

[13] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13(6):601–643, 2008.

[14] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[15] A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin. *Bayesian data analysis*. CRC Press, 2013.

[16] S. J. Gershman and D. M. Blei. A tutorial on Bayesian nonparametric models. *Journal of Mathematical Psychology*, 56(1):1–12, 2012.

[17] I. Grigorik. GitHub Archive. `www.githubarchive.org`, 2014. Visited Mar 2014.

[18] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *ICSE*, 2012.

[19] N. L. Hjort. *Bayesian nonparametrics*. Number 28. Cambridge University Press, 2010.

[20] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *Software Engineering, IEEE Transactions on*, 32(12):952–970, 2006.

[21] F. Jacob and R. Tairas. Code template inference using language models. In *Proceedings of the 48th Annual Southeast Regional Conference*, page 104. ACM, 2010.

[22] JetBrains. High-speed coding with Custom Live Templates. `bit.ly/1o8R8Do`, 2014. Visited Mar 2014.

[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[24] A. Jiménez, F. Berzal, and J.-C. Cubero. Frequent tree pattern mining: A survey. *Intelligent Data Analysis*, 14 (6):603–622, 01 2010.

[25] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[26] C. J. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[27] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.

[28] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. In *Reverse engineering*, pages 77–108. Springer, 1996.

[29] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 253–262. IEEE, 2006.

[30] I. Kuzborskij. Large-scale pattern mining of computer program source code. Master's thesis, University of Edinburgh, 2011.

[31] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176–192, 2006.

[32] P. Liang, M. I. Jordan, and D. Klein. Type-based MCMC. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 573–581. Association for Computational Linguistics, 2010.

[33] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. *arXiv preprint arXiv:1401.0514*, 2014.

[34] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *Proceedings of The 30th International Conference on Machine Learning*, pages 187–195, 2013.

[35] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT Press, 2012.

[36] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392. ACM, 2009.

[37] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *FSE*, 2013.

[38] Oracle. Java SE Documentation: Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking. `http://docs.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html`, 2014. Visited Feb 2014.

[39] P. Orbanz and Y. W. Teh. Bayesian nonparametric models. In *Encyclopedia of Machine Learning*. Springer, 2010.

[40] M. Post and D. Gildea. Bayesian learning of a tree substitution grammar. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, pages 45–48. Association for Computational Linguistics, 2009.

[41] E. Recommenders-Contributors. Eclipse SnipMatch. `wiki.eclipse.org/Recommenders/Snipmatch`, 2014. Visited Mar 2014.

[42] M. Research. High-speed coding with Custom Live Templates. `research.microsoft.com/apps/video/dl.aspx?id=208961`, 2014. Visited Mar 2014.

[43] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queen's University at Kingston, Ontario, 2007.

[44] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[45] J. Sethuraman. A constructive definition of Dirichlet priors. Technical report, DTIC Document, 1991.

[46] Y. W. Teh. A hierarchical Bayesian language model based on Pitman-Yor processes. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 985–992, 2006.

[47] Y. W. Teh and M. I. Jordan. Hierarchical Bayesian nonparametric models with applications. In N. Hjort, C. Holmes, P. Müller, and S. Walker, editors, *Bayesian Nonparametrics: Principles and Practice*. Cambridge University Press, 2010.

[48] A. Termier, M.-C. Rousset, and M. Sebag. Treefinder: a first step towards XML data mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 450–457. IEEE, 2002.

[49] R. Waldron. Principles of Writing Consistent, Idiomatic JavaScript. `https://github.com/rwaldron/idiomatic.js/`, 2014. Visited Feb 2014.

[50] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 319–328. IEEE, 2013.

[51] Wikibooks. More C++ Idioms. `http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms`, 2013. Visited Feb 2014.

[52] C. K. Williams and C. E. Rasmussen. Gaussian Processes for Machine Learning, 2006.

[53] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM, 2002.

[54] M. J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *Knowledge and Data Engineering, IEEE Transactions on*, 17(8):1021–1035, 2005.

[55] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.