Learning Garden-Variety Library Interactions

John Lai and Jean Yang
6.867 Final Project

December 4, 2008

If you have a garden and a library, you have everything you need. –Cicero

1 Introduction

Researchers have been trying to learn the "correct" way of using API libraries. Those working on program verification have tried to do this to separate correct uses from incorrect uses in order to help programmers create more bug-free programs. Ammons et. al. have shown that it is possible to infer specifications about interaction with APIs from looking at program traces [1]. Those working on code synthesis have used the sequences of possible API calls as a search space in which to generate correct code interacting with APIs. Mandelin et. al. have worked on automatically generating call sequences that interacts with APIs to achieve the desired goals [4].

These previous approaches have not specifically addressed the property that the correct way of interacting with a particular API may consist of different distinct modes of interaction. To consider a simple example, in C++ `malloc` is paired with `free`, `new` is paired with `free`, and `new[]` is paired with `free[]`. Trying to call `free[]` on an object allocated with `new` is incorrect. The ideal program specification would take into account different modes of interacting with the data. Similarly, knowing the likelihood of different sequences of API calls would reduce the code synthesizer's search space and likely make code synthesis more effective.

The goal of our project was to determine whether we can apply machine learning techniques to learning modes of interaction with API's and abstract data structures. We wanted to apply machine learning techniques to 1) see how we can map interactions with API libraries to separate clusters and 2) see if we could learn something from what these clusters look like. The main decisions we made in our experimental setup were regarding how to extract meaningful training data from API interactions, how to represent API information as useful feature vectors, and which classification techniques are most appropriate. For this project, we

1. generated data from program traces of Java programs with selected Java standard libraries.

2. investigated classification using a mixture of k-Markov chains.

We discuss our experimental setup in Section 2, our implementation of k-Markov chains in Section 3, and our results for the `ArrayList` library in Section 4.

2 Experimental setup

Though Java and C++ both have sufficiently "interesting" libraries and a large code base, we favored Java over C++ because 1) Java code is more portable, since it compiles to bytecode rather than machine code and is dynamically linked, 2) there are code copying issues involved with C++ templates, and 3) for hisorical reasons, there is more open source Java code that uses Java standard libraries than there is open source C++ code that uses the STL [1]. Here we describe how we implemented our profiler and constructed traces.

2.1 Instrumenting Java code to produce traces

We constructed runtime profiles of the code we run in order to construct the method traces (rather than adding logging code to the standard libraries) because 1) this method is more general, giving us the freedom of being able to construct learning data from any Java `jar` file and 2) this method gave us the freedom to choose our set of libraries to classify after seeing which ones were being used in the code.

We did this by writing a Java profiler that inserts logging instructions in the Java bytecode whenever there is a call to a standard library method of interest. Because Java's standard libraries are loaded in the bootstrap classloader rather than in the system loader, however, this causes problems for directly instrumenting the standard library classes. Because of this, we instrument other classes loaded in the system classloader and, from code in those classes, log calls to standard library objects [2].

We used the Java compiler's `javaagent` support for attaching a profiler to the execution of a program. We build our profiler from the source code from the Java Interactive Profiler [2], a Java profiler written in Java and built on top of the ASM bytecode reengineering library [5]. Our program dynamically inserts bytecode instructions into the profiled code that then call our profiler functions that log calls to standard library functions. We initially tracked sequences of calls to given libraries, but from analyzing preliminary results we realized that this information is not as useful as logging calls to specific instances of classes. To do this, we inserted bytecode instructions that inspect the runtime stack to get at the object. We take the hash value of each object in order to distinguish calls to different instances of the same class.

2.2 Building traces

We initially downloaded an assortment of programs, many of which had interactive interfaces, but we quickly realized that we would not be able to generate enough learning data by interacting with these programs. We chose instead to focus on programs that 1) could take lots of inputs and therefore could provide enough data, 2) used enough common libraries with other programs we found, and 3) did not load libraries that cause our profiler to crash. We were also limited by the fact that the profiler introduced significant overhead.

We generated traces on the following programs:

1. **Java Weather**, a library for parsing METAR fomatted weather data [3]. Our test program polled the San Francisco weather satellite for data at half-second intervals and parsed it.

2. **Lucene**, a library for searching webpages. We acquired a base of 100 websites and ran each resulting page through the Lucene index builder.

3. **Mallet**, a machine learning language toolkit for processing webpages. We ran code to import data from a set of webpages, train with it, and evaluate the results.

We constructed about 100 profiles that contained object-level information for Java Weather and Lucene and took a few traces from Mallet for importing data, training a model, and evaluating the model. We focused on the `ArrayList` library because it was the most extensively used in these programs. While it would have been nicer to have a more systematic way of generating our data, this gave us a good amount of diverse data to work with. Also, while this would cause our model to generalize poorly to new data from other programs, it still allows for the training of clusters and the priors in our mixture model can help account for the non-uniformly selected data.

[1]Many open source software in C++ uses its own version of abstract data structure libraries because 1) the libraries were not standardized across compilers and 2) people have efficiency issues with using the STL.

[2]Because of these unforeseen issues involving Java's idiosyncracies, this process required far more work than initially anticipated. We did, however, emerge victorious: we can now construct a profile of any Java JAR executable file.

3 Classification Model

3.1 k-Markov chains model

Consider the standard Markov model defined by

$$p(x_1, \ldots, x_m) = t(x_1) \prod_{i=2}^{m} t(x_i | x_{i-1})$$

where $t(x_1)$ is the probability that x_1 is the initial state and $t(x_i|x_{i-1})$ are the transition probabilities.

Similar to Gaussian Mixture Models, we can take a mixture of k Markov chains. A simple extension of the basic mixture model to allow variable length sequences gives the final model we used:

$$p(x_1, \ldots, x_m) = \sum_{z=1}^{k} q(z) \left(t_z(x_1) t(END|x_m) \prod_{i=2}^{m} t_z(x_i|x_{i-1}) \right)$$

3.2 Model Choice

We chose this model because it fits our goals of clustering different patterns of sequential interaction. We chose to use k-Markov chains for the following reasons:

1. Mixture models are a natural way to uncover clusters in an unsupervised fashion by examining the component models of the mixture.
2. The temporal relationship between method calls are more interesting and informative than, say, the counts of certain method calls. Markov chains capture this in a simple way by assuming that the system only cares about the previous state. While program interaction is not solely determined by the previous function call (in fact, there may be add/remove sequence that have context-free structures), we reasoned that choosing a simpler model would make learning easier while still yielding insight into API interaction[3].

We considered using hidden Markov models (HMMs) as well, but it was not clear to us what the hidden states would be. Also, the learning problem would have been considerably more difficult if we chose to use a mixture of HMMs, as we would need to estimate both the clusters the points were chosen from as well as the hidden states in EM.

3.3 Maximum Likelihood Estimation

Similar to our derivation of the EM algorithm for other models, we first take a look at maximum likelihood estimation in the case where we observe exactly the underlying Markov Chain that our observations were drawn from. In this case, simlar to GMMs, the maximum likelihood estimates for the parameters in our model are simple. Suppose that we have n training sequence and define $\delta(z,i)$ to be 1 iff sample i was drawn from chain z. The maximum likelihood estimates for our parameters will be:

$$n(z) = \sum_{i=1}^{n} \delta(z,i), \quad q^*(z) = \frac{n(z)}{n}$$

$$t_z^*(x) = \frac{\sum_{i=1}^{n} \delta(z,i)[[x_{i1} == x]]}{\sum_{i=1}^{n} \delta(z,i)}, \quad t_z^*(x_u|x_v) = \frac{\sum_{i=1}^{n} \delta(z,i) \sum_{j=1}^{|\underline{x}_i|-1} [[x_{ij} == x_v \wedge x_{i,j+1} == x_u]]}{\sum_{i=1}^{n} \delta(z,i) \sum_{j=1}^{|\underline{x}_i|} [[x_{ij} == x_v]]}$$

$$t_z^*(END|x_v) = \frac{\sum_{i=1}^{n} \delta(z,i)[[x_{i|\underline{x}_i|} == x_v]]}{\sum_{i=1}^{n} \delta(z,i) \sum_{j=1}^{|\underline{x}_i|} [[x_{ij} == x_v]]}$$

[3]We also considered using other classification methods (k-means clustering, SVMs) and capturing the temporal nature of our data by selecting pairs of function calls as features, but Markov chains capture this in a more straightforward way

k	training log likelihood	test log likelihood	number of parameters	BIC score for training data
2	-12090	-1295	143	-12673
4	-10112	-1078	287	-11284
6	-9268	-991	431	-11028
8	-9214	-989	575	-11562
10	-9207	-988	719	-12143

Table 1: Cross-validation results.

Note that since we have variable length sequences, we sum to $|x_i|$ in the denominator rather than $|x_i| - 1$. Now that we have the estimates for the case where we know which Markov Chain each sequence was generated from, we can extend this to the case where we have a probability distribution over the Markov Chains that could have generated the sequence. We can keep all of our estimates the same, except that we need to change the way the $\delta(z,i)$ are computed. Rather than being strictly 0 or 1, $\delta(z,i)$ now represents a normalized probability that example i was generated from the z^{th} Markov Chain. We compute $\delta(z,i)$ (assume that $\underline{x}_i$ has m observations and the probabilities are drawn from the previous parameters) with

$$\delta(z,i) = \frac{q(z)t_z(x_{i1})t_z(END|x_{im})\prod_{j=2}^{m} t(x_{ij}|x_{i,j-1})}{\sum_z q(z)t_z(x_{i1})t_z(END|x_{im})\prod_{j=2}^{m} t_z(x_{ij}|x_{i,j-1})}.$$

We implemented EM for this model in MATLAB without using any existing libraries or code.

4 Results

We first describe numerical tests that show how well the model fits the data. We then provide some detailed analysis and insights learned from examining the `ArrayList` clusters.

4.1 The number of underlying Markov Chains

To investigate the problem of determining how many underlying mixtures to choose, we split our data into 90% training and 10% test. (Performing n-fold cross-validation would have been better if time had permitted). We then ran our learning algorithm for different values of k, noting the training likelihood and test likelihood. Because the EM algorithm is sensitive to initialization, we chose models with the best training likelihood over 5 random initial starting points. We also included the BIC score to compare it with test set performance. For the BIC score, the dimension of a mixture of k components has $k-1$ parameters for the prior and if n is the number of possible states, $n-1$ parameters for the initial probabilities and n^2 parameters for the transition probabilities (since we have an end state, this is n^2 rather than $n(n-1)$.) We show our results in Table 1.

The data shows that $k = 6$ is the first k value to yield a close to maximal test log likelihood. The BIC score agrees that this is the best choice for number of underlying clusters. One observation of note is that we do not seem to have an over-training problem in terms of the classical increase in test error. One possible explanation for this phenomena is that our training set was much larger than our test set, though it is possible that we would start seeing more over-training for even larger values of k.

4.2 Relationship between programs and clusters

If clusters map traces back to their original programs, then this suggests different programs have different patterns of interactions; if not, then this suggests there may be some common patterns of interaction.

To examine this, we determined the best cluster for each training sample (by taking the cluster with the highest $p(z|x)$) and grouped these mappings by program. The data (Table 2) shows that that there was a single dominant cluster composed of samples from every program. It appears that the identity of the

program	best cluster	number in best cluster	% in best cluster	prior of best cluster
mallet-import	2	29	0.66	0.43
mallet-train	2	23	0.79	0.43
jweather	2	2203	0.80	0.43
mallet-evaluate	2	23	0.79	0.43
lucene-index	2	232	0.34	0.43

program	best cluster	number in best cluster	% in best cluster	prior of best cluster
mallet-import	5	28	0.64	0.47
mallet-train	5	23	0.79	0.47
jweather	5	2212	0.81	0.47
mallet-evaluate	5	23	0.79	0.47
lucene-index	5	342	0.50	0.47

Table 2: Best Clusters for $k = 4, 6$, respectively.

Cluster 1 (291)	Cluster 2 (3231)
`get->get`(77.0%)	`size->get`(27.4%)
`size->toArray`(6.6%)	`get->get`(14.6%)
`toArray->size`(5.6%)	`get->size`(14.5%)
`add->add`(3.5%)	`get->remove`(8.5%)
`size->size`(1.1%)	`size->add`(7.7%)

Cluster 1 (685)	Cluster 2 (208)	Cluster 3 (2629)
`size->get`(30.4%)	`size->toArray`(40.0%)	`get->get`(91.5%)
`get->size`(18.1%)	`toArray->size`(34.1%)	`get->clear`(1.1%)
`size->size`(14.1%)	`add->add`(17.3%)	`size->add`(1.4%)
`get->remove`(10.7%)	`toArray->add`(5.8%)	`clear->get`(1.1%)
`size->add`(9.2%)	`add->size`(2.9%)	`size->get`(1.1%)

Figure 1: Most frequently occuring call sequences for $k = 2, 3$.

program was not a strong determinant of the type of trace or interaction, and rather, there was a single dominant way of interacting with the API that was consistent across programs. This held true across different choices of k, hinting that adding more mixture components would simply refine our performance on the rarer edge cases.

4.3 Call sequence analysis

We profiled how the code interacted with `ArrayList` library methods `add`, `get`, `remove`, `size`, `clear`, `toArray`, `iterator`, and `clone`. There were the functions that appeared in the libraries we profiled; there are other functions such as `addAll` that did not appear. We clustered a total of 3522 sequences. The most frequently occurring sequences are very short sequences ([`add`], [`size`, `add`], [`size`, `get`, `remove`]).

Figure 1 shows the top transitions for different sequences. Looking at the $k = 2$ case, we end up with clusters that have the following methods with high frequency, from most frequent to least frequent:

1. `get`, `size`, `toArray`, `add`, `clone`, `clear`
2. `add`, `get`, `size`, `remove`, `clone`, `clear`, `iterator`

For $k = 3$ we get the following most frequent calls (from most to least):

1. `size`, `get`, `add`, `remove`, `iterator`, `clone`
2. `toArray`, `size`, `add`, `get`
3. `get`, `add`, `size`, `clear`, `clone`, `toArray` (relatively much fewer)

From these clusters we can infer that `get`, `add`, and `size` are a part of all modes of interacting with the library. The clusters for $k = 3$ suggest that if we were to categorize uses into three modes, we would have 1) creating `ArrayList` objects for the purpose of turning them into an array, 2) building an `ArrayList` and iterating over the elements, and 3) using only random access methods like `get` and `add`. These usage patterns also suggest copying (`clone`) and emptying (`clear`) are not typically used when creating an `ArrayList` for the purpose of turning it into an array.

For $k = 6$ we get similar results, with `iterator` and `toArray` contuiting to be separated, one cluster of just `size` and `get`, and other clusters combining the usual functionalities. We may also be able to separate read-only cases from read-write cases. Though we performed this inference by examining our data, we could potentially automate this process by looking at calls not common across clusters and how they are used, perhaps with more processed data (e.g., with multiple equivalent calls compressed into one).

In observing the initial and transition probabilities we noted that Markov mixture chains will not cluster sequences separately if they use disjoint states. Unlike GMMs which have a single mean, for Markov chains, a single chain can actually contain multiple (semantic) clusters if the clusters have disjoint states. For instance, for one chain we learned for $k = 3$, the initial probabilities are mostly spread between `get` (0.50 initial probability) and `add` (0.41 initial probability). Examining the transition probabilities, we see that (`get -> end`: 0.98), (`size -> toArray`: 1.0), (`toArray -> size`: 0.85). This suggests that the generated sequence will look very different depending on the initial state. If `get` is chosen, then we are very likely to terminate immediately. If `size` is chosen, then we will likely bounce between `size` and `toArray`. Because the initial state probabilities are more or less a coin flip between `get` and `size`, this one cluster actually captures two distinct interactions. Because of this, it is possible that our chains cluster together things that should be apart, and we need some sort of post-processing to really extract the exact modes of interaction. (For instance, we could break constituent chains into separate chains with deterministic start states if initial probabilities are split equally.)

Though we had hoped it would not matter, the length of the sequence seems to correspond to its classification. For $k = 2, 4, 5$, the largest cluster is associated with the shortest average sequence length and the smallest cluster is associated with the longest average length. (We could have tried to account for this in our learning algorithm, but we had not thought it would have a significant effect.)

5 Conclusion

We have explored applying machine learning techniques to the problem of clustering interactions with APIs and discovered that it may be possible to cluster library interactions.

For classification, we developed the k-Markov chain model based on the GMM mixture model and variable length HMM model developed in class. We implemented 1) a Java profiler for extracting object-level Java method traces from any Java JAR executable and 2) MATLAB routines for learning and classifying with a k Markov chains mixture model. We constructed method traces for learning from Java programs we found online. In learning, we explored the problem of determining the number of underlying mixtures by running cross-validation for different values of k and examining the BIC score. We used cross-validation and the BIC score to confirm our expectations that, after a certain point, increasing k does not yield better results.

Through examining cluster contents and parameters of the Markov chains, we determined that one could automatically learn a set of common ways of interacting with libraries. Individual programs did not appear to have idiosyncratic ways of interacting with the libraries we examined; this is consistent with our expectation that there is some fixed number of modes of interaction. Our model assigned most traces to a single cluster, indicating that there may be a dominant mode of interaction. It seems that for the `ArrayList` library, using Markov chains provides sufficiently strong assumptions to yield interesting results. We noticed, however, the deficiency that a single Markov chain can comprise multiple modes of interaction, but it seems that one could mitigate this problem with post-processing. We would also like to note that we were fortunate that the Markov model structure could capture the `ArrayList` interactions in a sufficiently interesting way. If we had a library with more complex interactions that did not fit the Markov assumption well the learning results may not have been quite as nice.

The results of this work has many applications. Given that we know that there are common use cases of an API, we could generate a useful set of example use cases for the programmer. We could also use this information in code synthesis: knowing about different usage modes, and perhaps about the common cases, could greatly optimize the search space. This information is also useful in test generation: we might be able to predict interactions with the code we write and make sure our code is robust to those cases.

If there had been no time constraints, we would have loved to look closely at other libraries, construct traces from more programs, and use variations in classification such as collapsing repeated states into a single states or a set of states and handling variable length clusters in a more clever way.

References

[1] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. pages 4–16, 2002.

[2] Jip - the java interactive profiler. `http://jiprof.sourceforge.net/`.

[3] jweather. `http://sourceforge.net/projects/jweather/`.

[4] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. *SIGPLAN Not.*, 40(6):48–61, 2005.

[5] ObjectWeb. Asm. `http://asm.objectweb.org/`.