6.864 Project: Determining API Usage Policies from Method Documentation

Tej Chajed, James Koppel

December 14, 2015

Abstract

The public APIs of various software packages contain constraints on what constitutes a valid use of each API method, often documented in both explicit (e.g., "may not be null") and subtle (e.g., "a duration in miliseconds" implying a non-negative integer) manners. In this work, we attempt to build a system that can learn constraints on method parameters from their documentation. Notably, our approach requires no annotated data, instead learning correct usage automatically by performing static analysis on a corpus of API usages (that is, calls to API methods in other ordinary code). We demonstrate our technique on the problem of detecting when a parameter is expected to be constant. The extracted policies can be used in automated checkers to detect bugs in programs that use the API.

1 Introduction

We focus our study on APIs in programming languages. API methods have various constraints on their correct usage. Some obvious examples include the number of arguments, types of arguments (in languages with type systems), and that a parameter must be positive. Many programming languages can formally express and automatically enforce the first two, while only a handful can handle the last.

Even for constraints that cannot be expressed directly within the language, a class of technologies called *static analysis* can make use of the constraints to provide valuable feedback. Some constraints are quite strict (such as an argument not being null) while others are more suggestions (such as discarding an error value being a likely source of bugs) or even stylistic (such as avoiding the use of `if (false) { }` to disable code).

2 Related Work

Our work is an addition to the large body of work on inferring API properties, surveyed in [7], but part of only a small body of work to use NLP in software-

engineering problems. Out of those, ours is the only approach to ecshew the need for annotated data, and the only approach to combine static analysis with NLP.

The DOC2SPEC system of Zhong et al [12] is the main preexisting approach to inferring API properties with NLP. They focus on inferring state-machine properties, *e.g.*, opening a file transitions it to the open state. They use named-entity recognition to extract action/object pairs (e.g.: `<open, file>`), and then fit these to a predetermined set of state machines.

There are several other systems which infer properties from method documentation, but these focus on the internals of a codebase rather than on public APIs. Tan et al's ICOMMENT system [8] is one of the more sophisticated uses of NLP in software engineering, focused on detecting locking and resource rules (*e.g.*, lock must be acquired before method is called). They first use clustering to identify locking-related comments. Then they train a decision tree on labeled data to map a comment to one of a few rule templates, and find named entities and variables to fill in the template. Its successor, ACOMMENT [9], applies this approach to interrupt behavior. The ACOMMENT system also uses static analysis to interrupt-related functions that the NLP component cannot, although the static analysis component does not directly interact with the NLP component.

The TCOMMENT system [10] focuses on the problem of detecting whether a method's documentation says an argument may or may not be null, and comparing it to the method's actual behavior, as determined by randomly generated tests. They analyze comments entirely using heuristics, such as searching for the word "not" near the word "null."

Finally, there are two other works of note. DOCREF [11] does named-entity recognition to find out-of-date cross-references in documentation. The oldest use of NLP for program analysis comes from [5], which uses whether a function name contains a keyword as a feature for part of a larger machine-learning system.

3 Learning Problem

To set up this information extraction problem as a learning task, we initially thought to use reinforcement learning. The constraints that we wanted to learn would come from pre-determined families (we called these "checkers"): our running example was the following method:

```
/**
 * Adds the specified component to the layout, using the specified
 * constraint object.  For border layouts, the constraint must be
 * one of the following constants:  <code>NORTH</code>,
 * <code>SOUTH</code>, <code>EAST</code>,
 * <code>WEST</code>, or <code>CENTER</code>.
 * <p>
 * Most applications do not call this method directly. This method
```

```
 * is called when a component is added to a container using the
 * <code>Container.add</code> method with the same argument types.
 * @param   comp         the component to be added.
 * @param   constraints  an object that specifies how and where
 *                       the component is added to the layout.
 * @see     java.awt.Container#add(java.awt.Component, java.lang.Object)
 * @exception   IllegalArgumentException  if the constraint object is not
 *                  a string, or if it not one of the five specified
    *              constants.
 * @since   JDK1.1
 */
public void addLayoutComponent(Component comp, Object constraints) {
}
```

Here there is a constraint that the `constraints` argument must be one of the specified constants. We envisioned having a family of checkers for an argument being in a specified set and learning that this checker is applicable to this method. The way we wanted to learn this mapping was through reinforcement learning, with signal provided by usages where the checker passed. This restricted us to constraints where we had checkers, but this was the intention anyway, since checking the constraints automatically is the intended use case.

Believing this would be too hard to train up to interesting uses of language, we went with a simpler learning problem. Instead of using the correct usages as feedback, we decided to directly train a model to learn the analysis results. What we were doing is actually a little subtler than this simple problem gives away. We interpret the analysis results as the *ground truth* for the constraint on the API. This interpretation make the assumption that usages are correct, a reasonable assumption given enough usage examples. Our formulation is also attractive because it requires no labelled data: in effect, static analysis extracts the annotation for us, in terms of actual correct usage rather than just what the documentation implies. This does create a burden for the learning: it must identify when there is actually a constraint in the documentation, or, put another way, "no constraint" always has to be an available output.

At this point our best motivating example of constraint and accompanying static analysis was nullness checking: determining whether or not null was an allowed value for a parameter. A usage can be checked by determining if the object being passed might be null. Here we had another choice for ground truth: analyzing whether the method itself could crash given null as a parameter.

We struggled to come up with a way to translate these facts into a learning task that was reasonable. We ran into several difficulties. Ths first was that the analysis results we had for null arguments were of low quality — the analysis was too conservative and intra-procedural, so it tended to say anything could be null. We also found documentation to be inconsistent with respect to null annotations — some libraries (like the JDK) document where null is not allowed and generally allow it, while in Google's Guava library the default is that nothing can be null. It seems like `null` is a real problem in Java, as reflected in its

handling. Finally, this problem is made less interesting by the availability and use of an `@Nullable` annotation intended to mark an argument as being possibly null (with good hygiene dictating that null is generally not allowed).

In the end we came up with a constraint that was easily checked by static analysis: constantness, explained in the next section.

4 Static analysis properties

The constraint that came up with was whether or not an argument is expected to be a constant. Argument values can either be constant (known at compile time) or dynamically calculated, a property we call *constantness*. Values are often dynamically calculated, at least in many usages, but constant arguments do appear. For example, in Python the `open()` function takes a filename (which is often dynamic) and a mode string, such as `"r"` for read or `"w"` for write, which should typically be constant. This constantness property was attractive because it can be particularly reliably analyzed. It also handled the issue of determining whether or not there was a constraint since we expected to see a class of arguments that could appear as either constant or non-constant (the pathname to `open()` being one such example).

We obtained our ground truth static analysis results using the Soot Framework [6]. We initially decided to focus on extracting three different properties: whether an argument is constant, whether an argument may be null, and whether an argument must be positive. These all have the advantage that they are binary properties (to make the ultimate learning problem easier) and can be computed using off the shelf analyzers. While we implemented passes to collect both nullness and constantness information, we ultimately ended up focusing on constantness because it produced the most reliable data.

Our constantness analysis is based on Soot's reaching definitions analysis. A reaching definitions analysis traces back from each use of a variable to every assignment that may have stored the value currently in that variable. At each method invocation, our constantness analysis reports that an argument is constant if (1) it is a literal or (2) it is a variable with exactly one reaching definition, and that definition is constant (recursively checked). If a definition is a class field, it reports constantness if that field is final, as will commonly be the case if the library exports special constants (*e.g.*, `JDialogBox.OK_CANCEL_OPTION`). This last step requires that the class containing the field be on Soot's classpath; it returns `UNKNOWN` if not. While for each library whose documentation is of interest we attempted to have some version on the classpath, ultimately approximately 0.1% of uses within our corpus had unknown constantness; we elected to discard these.

We also extracted nullness information from our corpus using Soot's nullness analyzer. However, we found the results from this analyzer were too imprecise because it is intraprocedural, meaning it assumes all parameters to a method can be null. As a result, while empirical studies have shown that the overwhelming majority of references in Java cannot be null [4], our static analysis reported

the opposite. As future work, we plan to modify the nullness analyzer to be interprodecural, a straightforward but computationally expensive change that will hopefully giive results precise enough to use in machine learning.

As future work, we are still interested in computing whether method arguments must be positive by using the interval analysis included as part of Soot's array bound analysis. We are also interested in specifically processing "one-of" constraints as in the `addLayoutComponent` example, which can also be done using our constantness analysis.

One interesting direction suggested at our poster presentation was applying our work to return values, for example checking whether users of a method should assume that a return value can be null, or whether a method's return value is an error code that should not be ignored. Indeed, Soot has a "NullnessAssumptionAnalyzer" that we can use for nullness, while the latter could be done using a live variables analysis.

5 Data

We collected our data from DARPA's MUSE (Mining and Understanding Software Enclaves) corpus [1], a large collection of software and software artifacts intended for use in "big code" projects. From the MUSE corpus we selected and downloaded about 200 projects to get samples of code usage. From the same source we downloaded documentation and pre-compiled bytecode for a few projects we considered libraries: these include the Java standard library, Apache Log4J, and the Android SDK, as well as 5 other projects. The methods in the libraries with documentation formed the APIs we trained and evaluated on.

We chose to get data from the MUSE corpus hoping for the most convenient packaging of code (with compiled JARs available that we could directly run our analysis on) as well as documentation already extracted using Doxygen, a widely-used documentation-generation tool. In retrospect problems we encountered with MUSE probably outweighted the benefits of its curation. Future work on the project could still benefit from the large scale of the corpus, allowing us to try our approach with over an order of magnitude more APIs and usages.

For our analysis, we had two main sources of data: static analysis results on usages and documentation of API methods. The static analysis results were obtained by running analyses written in the Soot Framework, as described above. One unforeseen problem was that the bytecode for projects was not available in a standard location in the project directory, even for projects where MUSE's crawler had managed to build the project itself. Meanwhile, we found that Soot is extremely picky about the classpath and the relative directory of code it is run on, and would often give errors instead of analyzing a project. We thus simply configured the analyzer to run on any JAR it could find in the directory. We managed to generate analysis results for approximately half of the 200 downloaded projects. The documentation was obtained by parsing the output

fraction constant	class
0	never constant
0–0.1	probably non-constant
0.1–0.9	maybe constant
0.9–1.0	constant

Table 1: Binning used for fraction of constant uses

of Doxygen from the MUSE corpus. The documentation was nominally in a structured JSON format, but appeared to have been translated from Doxygen's XML output, making turning it into text a non-trivial task with some loss of textual structure in the original comments.

We put these two pieces of raw information into a database (using sqlite3, a lightweight, serverless database). Care had to be taken to make the fully-qualified method names used in both sources exactly the same — some loss of information in the documentation meant we had to identify methods by package, name, and number of arguments, though technically in Java due to method overloading the types of all the arguments are required to fully disambiguate methods. In practice such type-based overloading is rare so we were not concerned. Then it was a simply matter to join the results by method and argument index and then group by method to aggregate usages, which could be conveniently performed with the corresponding SQL operations.

Rather than directly use the fraction of constant usages and solve a regression problem we decided to bin the fraction into a few classes and learn the classes. The reasoning was that the exact fraction was not very meaningful or related to the documentation, but the very low and very high ends of the spectrum were meaningfully different. The distribution of our data showed most methods fell at one end of the spectrum, with relatively few in the middle. We set our classes according to the bins in Table 1.

Our data consisted of a total of about 36,000 API (method, argument) pairs. These had a total of 11 million uses. We split the APIs into training, development and evaluation sets in a 70%–15%–15% split (ultimately our models did not have hyperparameters that needed tuning so the evaluation set was largely unnecessary). The documentation in our training data amounted to 8859 words, with 678 new unknown words in the development set and 658 unknown words in the evaluation set.

6 NLP models

We initially framed the problem as a two-phase classification problem: First, determining whether a comment encodes a constraint; second, determining what that constraint is. However, we had difficulty determining how to get data for the first problem. With the restriction to focusing on constantness, we could collapse these into a single classification problem: determining whether the documentation encodes a constantness constraint.

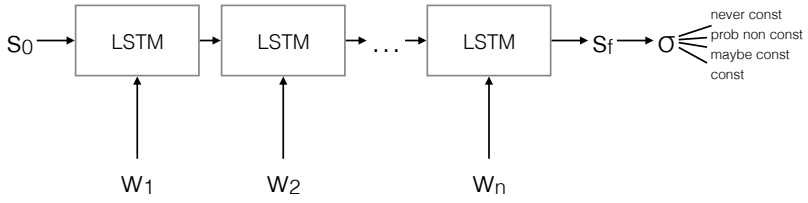

Figure 1: The neural architecture used to classify methods

We found that a majority of arguments (58% in the training corpus) were never constant. As a baseline, we used a prior distribution model that always returns "not constant."

Our next approach was a bag-of-words model: we built feature vectors from word counts, and then trained a logistic regression model. Words were tokenized with NLTK [3]; unknown words during evaluation were simply discarded.

Finally, we trained an LSTM neural network to perform the classification. The network first converts each token into a word vector embedding, initialized randomly and trained with the network. We then feed these through a sequence of LSTM nodes, and finally feed the final state into a softmax layer, which produces the final classification results. To handle variable-length documentation, we pad sentences out to a multiple of 10 tokens by prepending a special PAD token, and then run it through an LSTM net of that length. We discard examples with more than 100 tokens; this discards fewer than 5% of examples in the training set. Figure 6 outlines this neural architecture. We implemented this architecture using TensorFlow [2]. For the evaluation and dev sets, as in the bag-of-words model we first discard any words not seen in the training corpus. To classify methods with documentation longer than than 100 tokens, we revert to the baseline model.

7 Evaluation results

Our basic accuracy numbers are presented below. These are when the fraction of constant usages is binned according to Table 1 above.

classifier	dev	eval
baseline (prior distribution)	57%	61%
MaxEnt with Bag-of-words	62%	67%
Neural Net	61%	61%

Especially in response to feedback from the poster session we tried adjusting the binning strategy. Reducing the number of bins by either combining the $(0, 0.1)$ and $[0.1, 0.9)$ bins or combining the $\{0\}$ and $(0, 0.1]$ bins improved

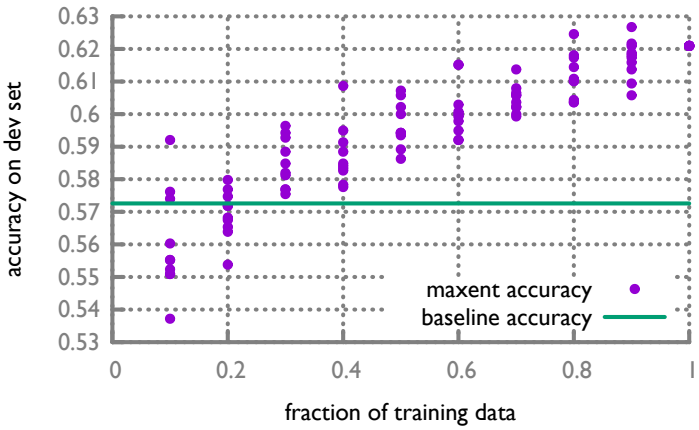

Figure 2: Learning curve for the maxent, bag-of-words classifier. Multiple points at the same fraction are repeated random samples, with ten trials in total for each fraction.

performance very slightly. In particular, combining the middle bins into a single $(0, 0.9)$ bin improved the model's accuracy to 64% without affecting the baseline. Eliminating the API methods with in-between fractions (defined as either $(0.1, 0.9)$ or even more extremely as $(0, 0.9)$) improved performance greatly, but affected the baseline just as much, to the point where the maxent model's performance was comparable to that of the baseline.

Further analysis of the learned model didn't reveal any great insights into its performance. The words with the highest positive and negative coefficients aren't particularly insightful and may even indicate some overfitting, such as "Content-Disposition", which clearly indicates an HTTP-related method. Abstracting words such as these would help avoid the possibility of overfitting in our domain-specific text.

The learning curve for the maxent classifier (Figure 2) was encouraging: performance did indeed improve with more examples, and does not appear to plateau. Thus we believe that more data may actually improve performance.

8 Division of Labor

Jimmy was responsible for:

- writing the nullness and constantness analyses in Soot,
- downloading projects from MUSE (and obtaining access),
- running Soot on projects,
- and developing the neural network model in TensorFlow.

Tej was responsible for:

- munging documentation from MUSE corpus,
- exploratory analysis,
- combining documentation and analysis data in sqlite3,
- developing the baseline and bag-of-words models,
- and creating the poster.

Overall contributions are close to equal.

References

[1] MUSE corpus. `http://corpus.museprogram.org/`, 2015.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python*. O'Reilly Media, 2009.

[4] P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP 2007–Object-Oriented Programming*, pages 227–247. Springer, 2007.

[5] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 161–176. USENIX Association, 2006.

[6] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[7] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.

[8] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158. ACM, 2007.

[9] L. Tan, Y. Zhou, and Y. Padioleau. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd international conference on software engineering*, pages 11–20. ACM, 2011.

[10] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 260–269. IEEE, 2012.

[11] H. Zhong and Z. Su. Detecting API documentation errors. In *ACM SIGPLAN Notices*, volume 48, pages 803–816. ACM, 2013.

[12] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318. IEEE Computer Society, 2009.