Structured Statistical Syntax Tree Prediction

Cyrus Omar
Carnegie Mellon University
comar@cs.cmu.edu

Abstract

Statistical models of source code can be used to improve code completion systems, assistive interfaces, and code compression engines. We are developing a statistical model where programs are represented as syntax trees, rather than simply a stream of tokens. Our model, initially for the Java language, combines corpus data with information about syntax, types and the program context. We tested this model using open source code corpuses and find that our model is significantly more accurate than the current state of the art, providing initial evidence for our claim that combining structural and statistical information is a fruitful strategy.

Categories and Subject Descriptors D.3.m [*Programming Languages*]: Miscellaneous

Keywords statistical models; prediction

1. Introduction

Programming languages are both formal systems with rich syntactic and semantic structure and human systems, in that they are used by people in patterned ways to express their intent. Many tools are designed to help people write code more efficiently by predicting the source code that a developer intends. For example, code completion systems for editors like Eclipse for Java display pop-up menus containing relevant class members and other snippets.

These code completion systems make use of the semantic structure of the language and API information extracted from imported libraries, but do not incorporate data about how developers have written programs in the past. Several pieces of recent work have shown, however, that incorporating statistical information is useful in particular settings (e.g. [1]). Indeed, Hindle et al. [2] have demonstrated that the next token that a user will enter can be predicted with reasonable accuracy using a purely statistical model that considers only a few previous tokens, neglecting all language, API and context-specific knowledge.

Our work aims to combine the structured and statistical approaches to source code prediction. Rather than using a tokenized representation of source code, we perform statistical prediction on a more natural representation of source code: the typed syntax tree. We can then condition our predictions on structural information, specifically:

- the *type*, denoted τ, of the expression being predicted (e.g. `int` or `Color`)
- the *syntactic context*, denoted σ, in which the expression occurs (e.g. whether the expression is an argument of a function call, the guard of an `if` statement, etc.)
- the *program context*, denoted Γ, in which the expression occurs (e.g. the set of variables paired with their types that are in scope at the location of the expression.)

For example, if a user enters the Java code `Planet destination =` where `Planet` is an enumeration type containing `Mercury`, `Venus`, `Earth`, etc. (but not `Pluto`, of course), then we have that the type of the expression being entered at the cursor is `Planet`, the syntactic context is *assignment*, and given a program context, our prediction space need only assign non-zero probabilities to:

1. literal members of the `Planet` enumeration
2. variables and fields of type `Planet` available from the program context
3. calls to methods available from the program context that have return type `Planet`[1].

The particular distribution of probability across expressions within these three categories is influenced in part by data derived from code corpus analyses.

In addition to the applications to code completion systems in code editors like Eclipse, more accurate source code prediction techniques could be useful for other programming tools. For example, programmers with severe physical impairments may benefit from predictive programming interfaces that allow them convey source code using devices more

[1] We can consider operators like + and [] as methods of the built-in types in Java.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2514876

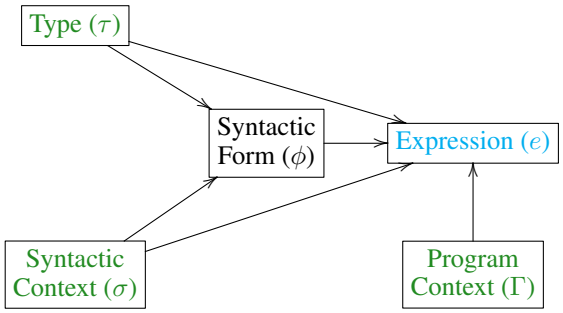

Figure 1. A graphical model representing our approach. Green variables are always observed (we do not assign marginal distributions to them). The syntactic form, ϕ, is a latent variable, and the expression, e, is unknown. The form is a function of the expression.

limited than a keyboard [3]. In addition, source code compression algorithms may benefit from more accurate probability models, which is an important consideration as applications are increasingly being sent over the network each time they are executed.

2. Methodology

To assign a probability to an expression, denoted e, we first determine how likely it is that the expression is of each of the three syntactic forms mentioned before, where syntactic forms are denoted ϕ. For each form, we can then assign probabilities to particular expressions of that form according to some form-specific conditional distribution. The conditional distributions for both the syntactic form and expression are learned using data gathered from analyses of prior code corpuses (smoothed using some suitable method in cases where enough information is not available).

Put more formally, our goal is to learn a model that allows us to produce $\mathbf{P}(e|\tau,\sigma,\Gamma)$. We determine this probability by marginalizing over a latent variable that represents the *syntactic form* of the expression, denoted ϕ:

$$P(e|\tau,\sigma,\Gamma) = \sum_{\phi\in\Phi} P(e|\phi,\tau,\sigma,\Gamma)P(\phi|\tau,\sigma)$$

As diagrammed in Figure 1, τ is the type of the expression, $\sigma \in \Sigma$ is the syntactic context, Γ is the program context, and $\phi \in \Phi$ is the syntactic form. We consider syntactic contexts $\Sigma = \{\textbf{statement}, \textbf{assignment}, \textbf{arg}, \textbf{other}\}$ representing plain statements, assignments to variables, method arguments and a catch-all for other syntactic contexts (e.g. conditional and loop guards, return statements, etc.). We consider syntactic forms $\Phi = \{\textbf{lit}, \textbf{var}, \textbf{meth}\}$, representing literals (of built-in and enumeration types), variables and method calls, respectively.

In the equations below, $\#\{e,\tau,\phi,\sigma\}$ represents the number of expressions in the training set constrained by the provided expression, type, syntactic form and program context (summing over any omitted categories.)

The conditional distribution for the syntactic forms is simply categorical, with the probability for each $\phi \in \Phi$ learned as:

$$P(\phi|\tau,\sigma) = \begin{cases} \frac{\#\{\phi,\tau,\sigma\}}{\#\{\tau,\sigma\}} & \#\{\tau,\sigma\} \neq 0 \\ \frac{\#\{\phi,\sigma\}}{\#\{\sigma\}} & \text{o/w} \end{cases}$$

That is, we use the empirical probability that that the expression has syntactic form ϕ given that the expression has type τ and is in syntactic context σ. If we have no data for type τ in that syntactic context, we marginalize over τ.

The conditional distribution for an expression e given the syntactic form, if the actual syntactic form of e is ϕ_e, is broken down according to the syntactic form as follows:

$$P(e|\phi,\tau,\sigma,\Gamma) = \begin{cases} P_{\textbf{lit}}(e|\tau,\sigma,\Gamma) & \text{if } \phi = \phi_e = \textbf{lit} \\ P_{\textbf{var}}(e|\tau,\sigma,\Gamma) & \text{if } \phi = \phi_e = \textbf{var} \\ P_{\textbf{meth}}(e|\tau,\sigma,\Gamma) & \text{if } \phi = \phi_e = \textbf{meth} \\ 0 & \text{if } \phi \neq \phi_e \end{cases}$$

The distributions for literals are determined by corpus data counting their frequencies. For variables, there is generally no relevant corpus data (because the lifespan of a variable is scoped), so we assign uniform probability to all variables in scope of the correct type. The distribution for methods benefits considerably from corpus data, however. To calculate the probability of a method itself, we consider two cases. It may be a method that we have seen in the training set, in which case we use the empirical probability of that method in the provided syntactic context. It may also be a method that hasn't been seen in the training set. In this case, we give a uniform distribution over all such methods callable via a variable or type in the program context.

2.1 Implementation

We used the Java language to implement our model and perform experiments to analyze the effectiveness of our methodology. Our prediction library is named **Syzygy** and is available at `http://github.com/cyrus-/syzygy`.

2.2 Results

We compared the probabilities generated in a cross-validated scenario to show that our method (SSCP) performed significantly better than the n-gram model of Hindle et al. across all six open source projects that we considered.

References

[1] M. Bruch, M. Monperrus, and M. Mezini. In *ESEC/FSE '09*, pages 213–222, New York, NY, USA. ACM.

[2] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[3] C. Omar, A. Akce, M. Johnson, T. Bretl, R. Ma, E. Maclin, M. McCormick, and T. P. Coleman. A feedback information-theoretic approach to the design of brain–computer interfaces. *Intl. Journal of Human–Computer Interaction*, 27(1):5–23, 2010.