Python Code Completion as a "N"LP Prediction Problem

Anthony Lu
6.864 Project

December 15, 2015

1 Introduction and Background

Many techniques in natural language processing rely on statistical methods because natural language is fuzzy and imprecise. In contrast, computer languages follow precise known rules, so tools for working with computer languages are usually strictly rule-based. However, even in the realm of computer languages, some analysis tasks may benefit from statistical methods by being able to capture regularities in how computer code is authored, such as in the usage of common APIs.

One such task that has received recent attention is code completion: given a program with holes, synthesize a list of candidate completions for the holes. Raychev et al. (2014) demonstrated success using an approach of reducing the code completion problem to a sentence completion problem, where the "sentences" are sequences of method calls extracted by running a program analysis, on which N-gram and RNN language models can be used to generate and rank completions. They demonstrate success using this approach on Java programs using Android APIs.

In this project I look at the task of code completion for Python programs. Python presents some challenges for precise and robust program analysis, such as that used in Raychev et al.'s approach, due to its dynamic typing and other properties that make it such that many facts about a program are only known at runtime. For these reasons, I focus on an approach that primarily uses purely syntactic features, with some experiments incorporating heuristics intended to approximately capture some semantic information.

Figure 1 shows an example input program, which is a valid Python program containing holes represented as sequences of underscores. For this project, I focus on completing attribute and method names, so I consider any attribute lookup where the attribute name consists entirely of (three or more) underscores to be a code completion query, to be filled in by a different attribute name.

```
import ast

def example_func(node):
  result = {}
  for field, value in ast.____(node):
    result[field] = value
  return result
```

```
Completions:
walk 0.0
iter_fields -0.76751687
assess -5.2171769
format -5.2171769
get -5.2171769
...
```

Figure 1: Left: An example input program with a hole for completion. Right: An example list of candidate completions.

2 Program Representation

Since a parser for Python code is available, it is unnecessary to model the syntax of Python in terms of a character-level representation of source files. Instead, programs are converted to an abstract syntax tree representation, using Python's `ast` module. The output of `ast.parse` is a tree of nodes whose types represent different syntactic elements in Python and whose fields include any associated data (e.g. identifier names or literal values) as well as child nodes. An example program with its corresponding AST structure is shown in Figure 2.

As a way to explore the suitability of this representation and to provide a baseline for the code completion task, ASTs can be used to train a PCFG language model, which induces a probability distribution over the space of possible syntax trees for Python programs. In a PCFG, each production rule in the grammar is associated with a probability, such that the probability of a syntax tree is the product of the probabilities of each rule used in the tree. The rule probabilities are often estimated by counting the number of occurrences of each rule in a training corpus:

$$P(A \to B|A) = \frac{count(A \to B)}{count(A)}$$

For Python ASTs, a slightly modified version of this scheme can be used to account for the slightly different structure of AST objects compared to syntax trees that are usually considered. Each type of AST node belongs to a category (e.g. statements, expressions) of node types that can appear in the same context. Furthermore, each node type has a predefined set of fields of predefined types, which may either be nodes of a particular category (or lists of nodes, or an optional node), or Python non-node objects (terminals). Exam-

```
> print ast.dump(ast.parse("for thing in things: self.process(thing)"))
Module(body=[For(target=Name(id='thing', ctx=Store()), iter=Name(id='things',
ctx=Load()), body=[Expr(value=Call(func=Attribute(value=Name(id='self', ctx=
Load()), attr='process', ctx=Load()), args=[Name(id='thing', ctx=Load())],
keywords=[], starargs=None, kwargs=None))], orelse=[])])
```

Figure 2: Python AST representation for a small code snippet.

$$
\begin{aligned}
\texttt{stmt} &\rightarrow \texttt{FunctionDef} \mid \texttt{ClassDef} \mid \texttt{Return} \mid \texttt{Delete} \mid \ldots \\
\texttt{expr} &\rightarrow \texttt{BoolOp} \mid \texttt{BinOp} \mid \texttt{UnaryOp} \mid \texttt{Lambda} \mid \ldots \\
&\ldots \\
\texttt{FunctionDef} &\rightarrow \{\texttt{args} : \texttt{arguments}, \\
&\qquad \texttt{body} : \texttt{list}[\texttt{stmt}] \\
&\qquad \texttt{decorator\_list} : \texttt{list}[\texttt{expr}]\}
\end{aligned}
$$

Figure 3: Sample Python grammar rules as obeyed by the AST. Top: Mapping from a node category to the set of concrete node types belonging to that category. Bottom: Mapping from each concrete node type to the categories of its fields. Together these rules define a mutually-recursive CFG.

ples of these rules, which hold for all Python code, are shown in Figure 3. In accordance with this structure, it is convenient to consider two kinds of production probabilities separately:

$$
\begin{aligned}
P(\text{concrete type}|\text{category}), &\text{ distribution for each non-terminal node} \\
P(\text{value}|\text{concrete type}, \text{field}), &\text{ distribution over field values}
\end{aligned}
$$

Note that for fields whose values are themselves nodes (as opposed to terminal values), the "field value" is actually just the type of the node, so the second distribution conveys no information; for fields containing lists of nodes or optional nodes, it is the distribution of the number of nodes in that field (e.g. the number of statements per function definition).

By itself, this simple AST-as-PCFG model is a very naive language model due to its strong context-independence assumption. In particular, in the context of the task of predicting attribute names, it reduces to predicting the attribute names that appeared most frequently overall in the training corpus, since $P(\cdot|\text{concrete type} = \text{Attribute}, \text{field} = \text{attr})$ is considered independent of all surrounding nodes in the AST where the attribute appears.

A simple improvement to the PCFG model is to add parent annotation or "vertical Markovization", where the production probabilities for a node can depend on not only the node's type but also the node's parent's type. This can be generalized to higher-order Markovization by considering longer segments of ancestors of the node, or even infinite-order by considering the entire path from the node to the root. To deal with data sparsity as the Markov order increases, we can use Kneser-Ney smoothing to interpolate between higher and lower order models.

3 Code Completion as Classification

While the PCFG framework with ancestor annotation can capture "vertical" context dependencies, it does not take into account other types of contextual information that may

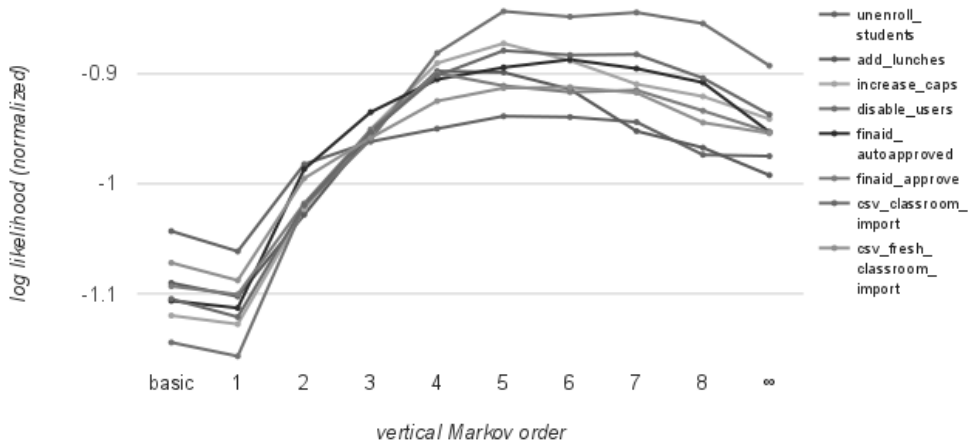

Figure 4: Cross-entropy of the PCFG language model on various test programs in the ESP dataset. The x-axis compares the basic language model with no parent annotation with parent annotation of various orders. The y-axis is the cross-entropy (log-likelihood) of the language model, normalized so that the cross-entropy scores on each test program lie within the same range.

be relevant to the code completion task.

Instead, the code completion problem can be viewed as a multi-class classification problem, where the completion is regarded as a categorical variable and predicted using features extracted from the surrounding AST.

As features, we use the types of the surrounding nodes, including the parent node (as in the parent-annotated PCFG model) but also sibling and child nodes, as well as features extracted from the fields of those nodes, including function arities, variable names, and other attribute names.

However, these features still do not take into account the fact that methods are often called on variables, and variables in a program carry semantic information according to how they are used, beyond the variable name itself (indeed, the programmer can rename all occurrences of the variable and have it still mean the same thing in the context of the program). To capture this, we include variable co-occurrence features based on a heuristic: for each variable in each source file, identify all uses of the same variable name and extract features from the AST nodes surrounding each use, as an attempt to capture how that variable is used.

Given these features, a maximum-entropy (logistic regression) model can be trained, which assigns a probability distribution over the target variable according to the "maximum-entropy" distribution given by

$$P(y|x) \propto \exp \sum_k \theta_k \phi_k(x, y)$$

for parameters θ_k which are fit to the observed data. These probabilities can then be used to rank candidate completions.

4 Experiments

Two experiments were run with different datasets: one on a smaller dataset consisting of source code from just one open-source project (`https://github.com/learning-unlimited/ESP-Website`), and one on a larger dataset consisting of all parseable source files in the 300 most-downloaded packages in PyPI (`https://pypi.python.org`).

The smaller experiment was intended to be an easier, more tractable problem, in that the data size is smaller and there is less variability because the training and test data come from the same project and use the same set of libraries and coding style. The larger experiment was intended to be a more realistic assessment of the code completion task in a more general setting.

For each experiment, test cases were generated by setting aside some of the source files from the training set and introducing holes where attribute accesses or method calls appeared in the program.

On the smaller dataset, the completion accuracy of the maximum-entropy classifier, with and without variable co-occurrence features, was evaluated against results obtained by using the PCFG language models' probability assessments to rank completions, as well as a baseline which uses the object's variable name as the only feature (i.e. it predicts the most commonly-seen attribute accessed on each variable name, or nothing for attributes accessed on non-variables). On the larger dataset, variable co-occurrence features were not evaluated due to making little difference on the smaller experiment and being more expensive to calculate. Additionally, due to the large cardinality of possible completions in the larger dataset, the output space was restricted to the 1000 most commonly-appearing attribute names in the training set, so that only those names would be considered as candidate completions.

5 Results

The results of these evaluations are summarized in Table 1. Performance on the smaller experiment was quite good, ranking the correct answer first in a majority of cases. In the larger experiment, about half of the generated test cases had correct answers that did not appear in the top 1000 most common in the training set, and a sizeable fraction of the correct names did not appear at all in the training set. However, for those where the correct answer was present in the top 1000, it was often ranked highly among the candidates.

The results indicate that an approach of code completion as classification may be effective in settings where the variability in code is relatively small, such as code within a project or code that uses a common framework, but may have some difficulty scaling to a more general setting where the number of distinct possible names is large and the correct answer is not necessarily one of the most commonly seen values.

In this project, it was assumed that it would be difficult to perform program analysis on Python, so a purely syntactic approach would be preferable. However, the experiments suggest that incorporating more program analysis may be necessary in order to be able to complete attribute names that did not appear often in training or to extract more

Model	Base	PCFG 1	PCFG 2	MaxEnt 1	MaxEnt 2
Experiment 1 (ESP, 240 test cases)					
Correct completion in top result	43	25	63	**137**	130
Correct completion in top 3	61	45	103	169	**172**
Correct completion in top 10	91	80	155	**196**	189
Median rank of correct completion	-	38	6	**1**	**1**
Experiment 2 (PyPI, 1151 test cases)					
Correct completion in top result	214	28	92	**317**	
Correct completion in top 3	301	76	165	**395**	
Correct completion in top 10	402	114	311	**431**	
Median rank of correct completion	455	1324	**140**	-	

Table 1: Performance of the classifiers in terms of completion accuracy. Base is the baseline model, which uses the object's variable name as the only feature. PCFG 1 is the basic PCFG model without ancestor annotation; PCFG 2 is the PCFG model with order-3 vertival Markovization; MaxEnt 1 is the maximum-entropy classifier without variable co-occurrence features; and MaxEnt 2 is the maximum-entropy classifier with variable co-occurrence features. Variable co-occurrence features were omitted in experiment 2 after having little effect in experiment 1.

useful information from the training data. Even though this analysis may be more difficult to do reliably than for stricter statically-typed languages, there do exist tools such as PyLint and PyCharm that can extract properties about programs often enough to be useful, and even a noisy analysis could yield a big improvement combined with statistical techniques.

Other interesting directions to explore would be to use different techniques for classification, such as a neural network to automatically learn variable embeddings or program embeddings that would capture the features of a program context that would be relevant for completion, or to incorporate more context dependencies into a generative language model that would be capable of synthesizing a larger class of program fragments as completions.