CACHECA: A Cache Language Model Based Code Suggestion Tool

Christine Franks*, Zhaopeng Tu*†, Premkumar Devanbu* and Vincent Hellendoorn‡
*Department of Computer Science, University of California at Davis
{clfranks, zptu, ptdevanbu}@ucdavis.edu
† Huawei Noah's Ark Lab, Hong Kong tuzhaopeng@gmail.com
‡ Delft University of Technology v.j.hellendoorn@student.tudelft.nl

***Abstract*—Nearly every Integrated Development Environment includes a form of code completion. The suggested completions ("suggestions") are typically based on information available at compile time, such as type signatures and variables in scope. A statistical approach, based on estimated models of code patterns in large code corpora, has been demonstrated to be effective at predicting tokens given a context. In this demo, we present CACHECA, an Eclipse plugin that combines the native suggestions with a statistical suggestion regime. We demonstrate that a combination of the two approaches more than doubles Eclipse's suggestion accuracy. A video demonstration is available at https://www.youtube.com/watch?v=3INk0N3JNtc.**

I. Introduction

The task of code completion is concerned with suggesting appropriate code tokens to a user during editing. Many Integrated Development Environments provide code suggestion, which can improve the programmers' productivity. Generally, such engines deduce what tokens "might" apply in the current syntactic context based on pre-defined syntactic and semantic rules. In this way, the suggestions produced by these engines are typically *valid* (in terms of syntax and semantics), but not necessarily *likely* (in terms of real-world usage).

N-gram models have been shown to successfully enhance the built-in suggestion engine by using corpus statistics (*i.e.* by suggesting what *most often* applies) [3]. However, standard n-gram models fail to deal with a special property of software: source code is *also very localized* [8]. Due to module specialization and focus, code tends to take especially repetitive forms in local contexts. The n-gram approach, rooted as it is in $\mathcal{NL}$, focuses on capturing the global regularities over the whole corpus, but fails to capture local regularities. To overcome this weakness, Tu *et al.* introduced a novel *cache language model* to capture the localness of source code [8].

The *cache language model* (denoted as *\$-gram*) extends the traditional n-gram models by deploying an additional *cache* to capture regularities in the locality. The n-gram and cache components capture different regularities: the n-gram component captures the corpus linguistic structure, and offers a good estimate of the mean probability of a specific linguistic event in the corpus; around this mean, the local probability fluctuates, as code patterns change in different localities. The cache component models these local changes, and provides variance around the corpus mean for different local contexts. Technical details can be found in [8].

This paper presents the novel features and core architecture of CACHECA, an Eclipse plugin for code suggestion based on the cache language model. CACHECA (a portmanteau of Cache and Content Assist, Eclipse's term for code suggestions) combines suggestions from two different sources: (1) the Eclipse built-in suggestion engine offers *syntactic* and *semantic* suggestions based on type information available in context; (2) a *\$-gram* model suggestion engine offers suggestions that commonly occur in either the whole corpus (from the *global* n-gram model) or the local context (from the *local* cache model). Our experiments show that the *\$-gram* based suggestion tool greatly enhances Eclipse's built-in engine by incorporating both the corpus and locality statistics, especially when no type information is available.

CACHECA is available for download as a plugin from macbeth.cs.ucdavis.edu/CACHECA/ and the source code is available at https://github.com/christinef/CACHECA.

II. Features

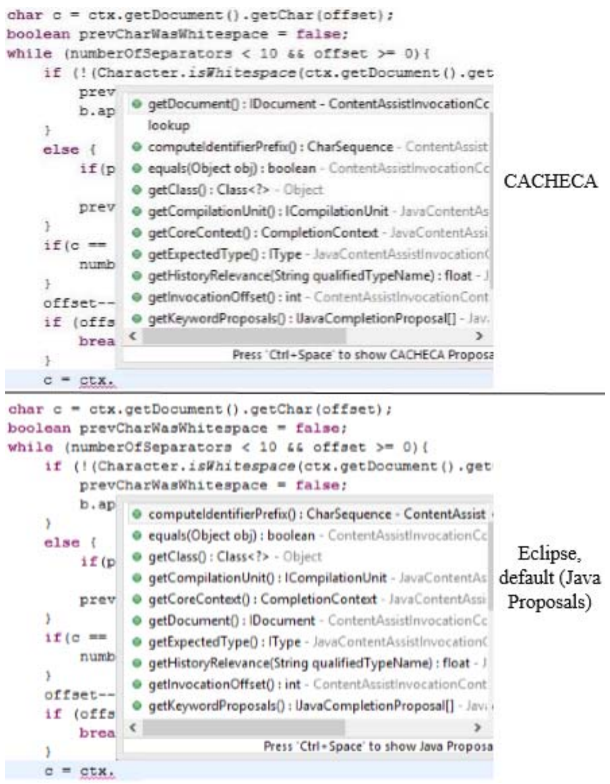
CACHECA operates within Eclipse's editor and can be invoked either automatically (wherever the user has set Content Assist to normally be invoked) or manually (using Ctrl+Space). By leveraging the cache language model, it has three appealing features to enhance Eclipse's built-in suggestion engine:

1. It is still able to offer suggestions even when no type information is available, making it applicable to both dynamic and static typed programming languages.
2. The suggestions and corresponding probabilities from the cache language model help to rerank and complement the original suggestions from the built-in plugin. In this way, CACHECA enforces a *natural* ordering on the suggestions.
3. Based on the observation that a token used in the immediate past is much more likely to be used again soon, CACHECA is able to capture the short-term shifts in token frequencies in the locality (*e.g.* a single file).

A. Natural Code Suggestion

Different projects have idiosyncratic token frequencies [3], [8]. Therefore, even given the same context, suggestions in different orders should be provided for different projects. This is overlooked by the built-in suggestion engine, and therefore by incorporating a language model engine, we can sort the suggestions for different projects using corpus statistics.

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.228



```
char c = ctx.getDocument().getChar(offset);
boolean prevCharWasWhitespace = false;
while (numberOfSeparators < 10 && offset >= 0){
    if (!(Character.isWhitespace(ctx.getDocument().get
        prev
        b.ap
    }
    else {
        if(p

        prev
    }
    if(c ==
        numb
    }
    offset--
    if (offs
        brea
    }
    c = ctx.
```

```
getDocument() : IDocument - ContentAssistInvocationCc
lookup
computeIdentifierPrefix() : CharSequence - ContentAssist
equals(Object obj) : boolean - ContentAssistInvocationCc
getClass() : Class<?> - Object
getCompilationUnit() : ICompilationUnit - JavaContentAs
getCoreContext() : CompletionContext - JavaContentAssi
getExpectedType() : IType - JavaContentAssistInvocation(
getHistoryRelevance(String qualifiedTypeName) : float - J
getInvocationOffset() : int - ContentAssistInvocationCont
getKeywordProposals() : IJavaCompletionProposal[] - Jav.
Press 'Ctrl+Space' to show CACHECA Proposa
```

```
char c = ctx.getDocument().getChar(offset);
boolean prevCharWasWhitespace = false;
while (numberOfSeparators < 10 && offset >= 0){
    if (!(Character.isWhitespace(ctx.getDocument().get
        prevCharWasWhitespace = false;
        b.ap
    }
    else {
        if(p

        prev
    }
    if(c ==
        numb
    }
    offset--
    if (offs
        brea
    }
    c = ctx.
```

```
computeIdentifierPrefix() : CharSequence - ContentAssist
equals(Object obj) : boolean - ContentAssistInvocationCc
getClass() : Class<?> - Object
getCompilationUnit() : ICompilationUnit - JavaContentAs
getCoreContext() : CompletionContext - JavaContentAssi
getDocument() : IDocument - ContentAssistInvocationCc
getExpectedType() : IType - JavaContentAssistInvocation(
getHistoryRelevance(String qualifiedTypeName) : float - J
getInvocationOffset() : int - ContentAssistInvocationCont
getKeywordProposals() : IJavaCompletionProposal[] - Jav
Press 'Ctrl+Space' to show Java Proposa
```

Fig. 1. CACHECA produces suggestions intelligently, based on localness.

B. *Intelligent Code Suggestion*

If a programmer uses a certain token when (s)he is coding, there is an increased likelihood of this token being used again in the near future. For example, the declaration of an identifier in a method scope makes usage of this identifier highly likely in the subsequent code. The cache component of CACHECA is designed to capture such localized regularity by estimating the probability of a suggestion from its recent frequency of use. Indeed, Tu *et al.* [8] showed that, by adding a cache component, the accuracy of predicting identifiers is more than two times the accuracy using only the n-gram model.

Consider the example in Figure 1. Unlike Eclipse's default engine, CACHECA suggests the correct `getDocument()` method first, because it was invoked twice in the lines above it. The re-ranking of the suggestions (through the Mix model; see section IV-D for more detail) is one of Cacheca's strong points; because Eclipse's default relies on a low-granularity ranking system, mostly based on types, often the anticipated completion is ranked very low on the list (6^{th} here). CACHECA, however, anticipates reappearances and ranks them higher.

C. *Type Information*

A drawback of Eclipse's built-in suggestion engine is that it relies heavily on type information in the context. In dynamically typed languages, where type information is inherently unavailable, code suggestion based on semantic and syntactic information is particularly difficult. While CACHECA

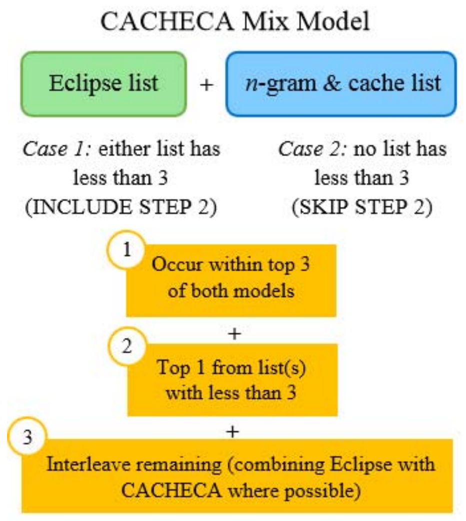

CACHECA Mix Model
Eclipse list
+
n-gram & cache list
Case 1: either list has less than 3 (INCLUDE STEP 2)
Case 2: no list has less than 3 (SKIP STEP 2)
1
Occur within top 3 of both models
+
2
Top 1 from list(s) with less than 3
+
3
Interleave remaining (combining Eclipse with CACHECA where possible)

Fig. 2. The architecture of CACHECA's Mix model.

yields a substantial improvement for Java, there is also great potential for languages that are dynamically typed.

III. Architecture

CACHECA combines suggestions from two sources: the suggestions that Eclipse provides (ordered by a custom 'relevance score'), and the suggestions from the *$-gram* model (ranked by their probabilities).

A. *Methodology*

In order to build the *$-gram* model, the tool detects (by means of a listener) when a new file is visible (*i.e.* a new file opened, reopened, or brought to the front). At this point, the n-gram model is read in from a file with word counts and a new, empty cache is generated. For statistics on time and memory usage, see [8].

When CACHECA is invoked, either by pressing a key character (such as the dot operator) or using Ctrl+Space, CACHECA starts by retrieving Eclipse's suggestions, as described in section III-B. Subsequently, the *$-gram* suggestions are computed (as described in [8]) and the lists are mixed to obtain the proposal list. The mixing process is described in section IV-D and Figure 2 provides a high level illustration. To preserve the ordering imposed on the suggestions by the mixing procedure, CACHECA includes a custom sorter ("CACHECA Sort") which extends the JavaCompletionProposalSorters extension point. The final generated list is then displayed to the user by Eclipse.

Since the *$-gram* model is estimated using only lexical code tokens, it cannot produce Javadoc material, nor can it provide parameter information or distinguish between data members and methods. However, utilizing the fact that Eclipse presents a laundry list of suggestions (albeit typically in a poor order), we search through Eclipse's default list of suggestions to find matches to CACHECA's suggestions. By using CACHECA's ranking preferences and Eclipse's additional information, CACHECA provides a robust code completion system.

B. Implementation Details

CACHECA is written entirely in Java. Initially, the aim was to combine statistical modeling code (written in C++) and the plugin code (necessarily written in Java, due to the Eclipse plugin framework) using sockets and the Java Native Interface (JNI). This approach was unsuccessful due to cross-platform limitations and latency issues, after which the decision was made to convert the existing modeling code to Java. Due to restrictions placed on plugins by the Eclipse extension framework, CACHECA must 'collect' Eclipse's unordered list of completion suggestions from the compilation unit and then sort the suggestions exactly as Eclipse does internally (*i.e.* first by the internal 'relevance' value and then alphabetically). This results in an ordered list exactly like that which Eclipse would display if unaided, which we can then incorporate into the Mix model. Ideally, a future iteration of CACHECA would place this tool inside Eclipse itself, eliminating overhead.

IV. Results

The goal of CACHECA is to combine the advantages of the two core suggestors: Eclipse's Java Proposals and the *$-gram* model. We seek to make use of the different information that is captured by these approaches to yield superior suggestion performance over Eclipse alone.

A. Data

We evaluated our models on a corpus of token sequences in seven Java projects of varying sizes, which was used previously for evaluation of code completion performance: `Ant,Batik,Cassandra,Log4J,Maven2,Maven3` and `Xalan` [3], [6], [8]. The included projects were cloned from public Git repositories between December 2009 and January 2011 and contain between 60KLOC and 367KLOC for a total of 5 million tokens. On each project, we used 10-fold validation by training our model on 90% of the lines, counting the token sequences, and testing on the token sequences in the remaining 10% of the lines. Each test-case can be seen as a scenario, where a certain history of tokens is known and the model is tasked with predicting the next token.

We specifically excluded a number of tokens from the task of suggesting, namely all parentheses and the symbols: `;` `.` `=` as these symbols are used to trigger a completion in Eclipse. Tu *et al.* [8] showed that separators and operators such as these are relatively easy to predict and make up a large part of source code. Therefore, we expect the completion performance of the cache model to be lower than the overall scores reported in their work and more in line with the performance reported on identifier and keyword completion.

B. Metrics

In each testing scenario (a context of tokens from which the suggestors must predict the next token), each suggestor returns a list of suggestion in ranked order, which may be empty and is cut off at ten suggestions. The rank of the correct suggestion in this list is used as a measure of performance. Following the common settings in the code suggestion task [3], [6], [8], we measure the performance of the suggestors in terms of both MRR score and Top$K$ accuracies. The Top$K$ accuracies are computed as the percentage of contexts in which the suggester ranks the correct suggestion at index K or less. We report accuracies for $K = 1$, 5 and 10. The Mean Reciprocal Rank (MRR) score is computed by averaging the reciprocal of the rank of the correct suggestion over all suggestion tasks in the test data. If the correct suggestion is not returned, a score of 0 is assigned. Hence, an MRR score of 0.5 implies that the correct suggestion is on average found at index 2 of the suggestion list.

C. Models

We have two baseline models: the Eclipse content assist and the *$-gram* suggestor. The Eclipse suggestions are retrieved as described in section III-B. The *$-gram* suggestor consists of the n-gram and cache components as described in [8].

Both models produce an ordered list of suggestions based on a score system: a 'relevance score' for the Eclipse model and a probability for the *$-gram* model. The former is computed based on a large number of hard-coded weights, linked to possible properties that a suggestion can have. Besides the lack of justification for these (rather diverse) weights, the relevance scores for different suggestions were often identical (*e.g.* the same weight for all possible method invocations on an object). Beyond the ordering, we were unable to gain much insight from the relevance scores.

The output of CACHECA is a new model, named Mix, which mixes the suggestions returned by the baseline models. The aim is to leverage information that is captured by the individual models and use this to improve suggestion accuracy. We constructed the Mix model based on a case study on the Ant project (section IV-D) and verified the results across the entire dataset.

D. Mixing

As shown in Table I, the *$-gram* model (model 2) performs approximately twice as well on MRR score and even better in Top1 accuracy as the Eclipse model (model 1). Hence, we compare the performance of the Mix model with the stronger baseline, the *$-gram* model, in the following experiments. We construct the Mix model based on a collection of *heuristics*. Each heuristic is applied in order and will take zero or more items from both of the suggestors and add these to the mix. The scores for the Mix model with one or more heuristics (in order) are listed in table I as Mix (Hi, $\cdots$,Hj).

Heuristic 1: interleave suggestions of the baseline models.
At the core of the Mix model is a simple interleaving of the results, which starts with the best suggestion of the *$-gram* model (the alternative performs worse than 2.). As can be

TABLE I
ACCURACY WITH VARIOUS SETTINGS ON *Ant*.

Model	*MRR*	*Top1*	*Top5*	*Top10*
Baseline				
1. Eclipse	24.35%	18.23%	33.30%	39.77%
2. *$-gram*	48.11%	38.95%	60.40%	67.10%
Mix				
3. Mix (H1)	49.76%	38.95%	65.04%	73.58%
4. Mix (H2, H1)	51.19%	41.24%	65.44%	73.98%
5. Mix (H3, H1)	51.44%	40.76%	66.41%	75.05%
6. Mix (H3, H2, H1)	52.29%	42.34%	66.40%	75.05%

seen from the results of model 3 in table I, interleaving yields superior MRR scores, primarily by improving Top5 accuracy. To improve the mix model further, we add two heuristics based on properties of the models under consideration. Due to its general nature, interleaving is always executed last.

Heuristic 2: If either model returns no more than 3 suggestions, add the first element to the mix.
In the case of the *$-gram* model, a shorter list indicates that the context was only seen in the cache and is therefore *local* to the code. Similarly, when encountering an unusual context, Eclipse will only suggest results that match the needed type of a variable. We found that top suggestions in shorter lists were significantly more likely to be correct for both models, and that a large part of the improvement in performance of H2 over H1 is due to Eclipse suggestions, since simple interleaving already prioritizes the top suggestion from the *$-gram* model. As can be seen from table I (model 4), this heuristic substantially improves Top1 suggestions and the MRR score.

Heuristic 3: If a suggestion occurs among the top 3 suggestions of both baseline models, add it to the mix.
This heuristic gives extra weight to an element that occurs near the top of both baseline models. We use the 'top 3' criterion to ensure that the overlap is meaningful, *i.e.* that the suggestion is deemed likely by both models (particularly for *$-gram*) and is not simply part of a group of suggestions that are all deemed equally accurate (particularly for Eclipse). From table I, we see that H3 yields slightly higher MRR performance and slightly lower Top1 performance than H2. Applying first H3 and then H2 (model 6) yields superior performance in both regards.

The results were consistent across the tested projects. CACHECA achieved an average improvement of 29.83 percent points over Eclipse's Content Assist in terms of MRR. Top10 accuracy improved by an average of 34.81 percent points and Top1 accuracy more than doubled from 25.36% to 55.19%.

Finally, although the prediction performance of the Mix model is dominated by the *$-gram* model, the Eclipse suggestions play another important role. As was observed in III-A, the *$-gram* model can only produce *lexical* completions, whereas the Eclipse suggestor typically returns *template* completions. Hence, for each suggestion returned by the *$-gram* model we first attempt to find a matching template from an extended list of Eclipse suggestions (the correct completion is often contained in the top 30) and return this template instead. This strategy proves successful in approximately half of the cases.

V. Related Work

Another Eclipse plugin, dubbed Calcite [4], operates on the same assumption as CACHECA (that more commonly-used suggestions should rank higher) but attempts to accomplish this by capturing the personalized usage across groups of programmers through crowdsourcing, rather than capturing the local regularities. In particular, Calcite presents suggestions above and below Eclipse's, whereas CACHECA generates a blended list of suggestions. Bruch *et al* [2] describe a plugin that suggests method calls, based on co-occurrence statistics; CACHECA is not restricted to method calls. A follow-on work [1] also uses a "social suggestion" feature where actual uses by real users are monitored and used to improve performance; this approach is potentially complementary to CACHECA. GraPacc [5], another code Suggestion plugin, focuses on APIs and incorporating the popularity of certain Suggestions. There are several other suggestion engines; we refer the reader to longer papers [2], [3], [7] for a more complete literature survey.

VI. Conclusions

We created a plugin for Eclipse, CACHECA, that combines the default suggestions with suggestions returned by a cache language model [8]. We found that the cache language model yields substantially superior performance to the Eclipse suggestion engine. Furthermore, by combining the suggestions returned by both models, CACHECA improves the Top1 accuracy of suggestions by 26.43 percent points and the Top10 accuracy by 34.81 percent points. We demonstrate that the cache language model introduced in [8] has practical application and that there is substantial potential for CACHECA to improve code suggestion. Building on the results, we suggest extension to dynamically-typed languages. This material is based on work supported by NSF under Grants 1247280 and 1414172.

References

[1] M. Bruch, E. Bodden, M. Monperrus, and M. Mezini. Ide 2.0: collective intelligence in software development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 53–58. ACM, 2010.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *FSE*, pages 213–222. ACM, 2009.

[3] A. Hindle, E. Barr, M. Gabel, Z. Su, and P. Devanbu. On the naturalness of software. In *ICSE*, pages 837–847. IEEE, 2012.

[4] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers. Calcite: Completing code completion for constructors using crowds. In *VLHCC*, pages 15–22. IEEE, 2010.

[5] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *ICSE*, pages 69–79. IEEE, 2012.

[6] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *FSE*, pages 532–542. ACM, 2013.

[7] R. Robbes and M. Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010.

[8] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *FSE*, pages 269–280. ACM, 2014.