On the Localness of Software

Zhaopeng Tu Zhendong Su Premkumar Devanbu
Department of Computer Science, University of California at Davis, CA, USA
{zptu,su,ptdevanbu}@ucdavis.edu

ABSTRACT

The n-gram language model, which has its roots in statistical natural language processing, has been shown to successfully capture the repetitive and predictable regularities ("naturalness") of source code, and help with tasks such as code suggestion, porting, and designing assistive coding devices. However, we show in this paper that this natural-language-based model fails to exploit a special property of source code: *localness*. We find that human-written programs are *localized*: they have useful local regularities that can be captured and exploited. We introduce a novel *cache language model* that consists of both an n-gram and an added "cache" component to exploit localness. We show empirically that the additional cache component greatly improves the n-gram approach by capturing the localness of software, as measured by both cross-entropy and suggestion accuracy. Our model's suggestion accuracy is actually comparable to a state-of-the-art, semantically augmented language model; but it is simpler and easier to implement. Our cache language model requires nothing beyond lexicalization, and thus is applicable to all programming languages.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Documentation, Experimentation, Measurement

Keywords

Localness, Cache Language Model, Code Suggestion

1. INTRODUCTION

The spectacular advances in natural language ($\mathcal{NL}$) processing in recent years, in terms of speech recognition, translation, *etc.*, have in great measure been due to the ability of *language models* to capture the repetition and regularity in commonplace speech and writing. Language models assign a probability to a word sequence using an estimated probability distribution. High-quality language models lie at the heart of most $\mathcal{NL}$ applications, such as speech recognition [22], machine translation [7], spelling correction [24] and handwriting recognition [46]. The most successful class of language models are n-gram models, introduced three decades ago [6].

Recently, Hindle *et al.* [18]—building on the *uniqueness* property of source code reported by Gabel and Su [16]—have shown that software corpora, like $\mathcal{NL}$ corpora, are highly repetitive and thus predictable using language models. Good quality language models show great promise in software engineering applications. Hindle *et al.* show that n-gram language models perform quite well, and leverage this fact for code suggestion. In addition, Nguyen *et al.* [35] have adapted the n-gram models for cross-language porting, and Allamanis *et al.* [3] have applied them to the automatic recognition and checking of coding styles. In this paper, we follow this work, to further improve language models.

We begin our work by remarking on a special property of software: in addition to being even more repetitive and predictable than $\mathcal{NL}$, source code is *also very localized*. Due to module specialization and focus, code tends to take special 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, and neglects local regularities, thus ignoring the *localness of software*. The localness of source code refers to the skewed distribution of repetitive n-gram patterns in the locality, namely the *endemism* and *specificity* of n-gram patterns in the locality. *Endemic* n-grams occur only in one locality (such as a single file). For example, new programming units (files, modules) usually introduce new identifiers (variable names, method names, and such), along with *endemic n-gram patterns* of identifier use (*e.g.* `myinvitees.next()`). *Specificity* is a less degenerate version of endemism, signifying the tendency of some non-endemic n-grams to favor a specific locality, while not being endemic to it. For instance, different programmers have idiosyncratic n-gram frequencies (*e.g.* some programmers favor "`for ( int i`" while others prefer "`for ( int size`"), resulting in the frequency-biased n-gram patterns in the specific files that each programmer is involved in.

Since n-gram models can only provide suggestions occurring in the training data, n-gram patterns specific or endemic to the locality can be overlooked. For example, let us predict the next token after a sequence "`for ( int`". Suppose the token "`i`" followed these three tokens 30% of the time in the training data, while the token "`size`" followed them 5% of the time. The n-gram model will assign "`i`" a probability of 0.3 and "`size`" a probability of 0.05, and thus "`i`" is chosen. For an isolated line, this would be the reasonable choice to make. But now suppose that several previous lines in the same file contained the n-gram "`for ( int size`", while none contained "`for ( int i`". Arguably, the token "`size`" should

FSE'14, November 16–21, 2014, Hong Kong, China
ACM 978-1-4503-3056-5/14/11
http://dx.doi.org/10.1145/2635868.2635875

then be assigned a much higher probability. A token used in the immediate past is *much* more likely to be used again soon than its overall probability that n-gram models would predict.

The key insight underlying our work concerns this critical limitation of traditional n-gram models. Our *central hypothesis* is:

> Source code is *locally repetitive*, *viz.* it has useful local regularities that can be captured in a *locally estimated cache* and leveraged for software engineering tasks.

We believe that this "localness" of software has it roots in the imperative to create modular designs, where modules secrete localized knowledge [37]. To leverage this property, we deploy an additional *cache* component to capture the localized regularities of both *endemic* and *specific* n-gram patterns in the locality. However, we need to overcome several challenges:

- How to combine the global (n-gram) model with the local (cache) model? How to automatically pick the interpolation weights for different n-grams?
- What is the locality to use? For example, how much local data do we need? Which length of n-grams should we use?
- Does the additional cache component impose high computational resource cost?

In Section 4, we solve these problems by introducing a mechanism that does not compromise the robust simplicity of n-gram models.

Our empirical evaluation on projects from five programming languages demonstrates that a cache language model consisting of both n-gram and cache components yields relative improvements of 16%$\sim$45% in suggestion accuracy over the state-of-the-art n-gram approach [18]. Surprisingly, using just the cache component built on 5K tokens, *by itself*, outperforms the n-gram model trained on nearly 2M tokens in suggestion accuracy. It is worth emphasizing that the cache model is quite simple and only requires lexicalization; no parsing, type checking, *etc.* are required. Because of its simplicity and unique focus on localness, it is complementary to most state-of-the-art research on statistical modeling of source code.

Contributions. Our key contributions are:

1. We demonstrate (Section 3) empirically over large software corpora, that source code is localized in the sense that code regularities (n-gram patterns) are *endemic*, *specific*, and *proximally repetitive* in the locality .
2. We introduce (Section 4) a novel cache language model to capture the localness of software that is simple (requiring no additional information other than the tokens) and easy-to-use (automatic selection of dynamic interpolation weights for n-gram and *cache* components).
3. We show (Section 5) that the cache model indeed captures the *localness* of source code and the strength of the cache model can be exploited for code suggestion, which substantially improves the state-of-the-art n-gram approach [18].

4 We provide (Section 5.2.1) an option for code suggestion when there is not enough corpus to train a language model: using only the cache component built on *thousands* of tokens, which achieves comparable suggestion accuracy with the n-gram model trained on *millions* of tokens.

2. BACKGROUND

2.1 Statistical Language Models

Language models are statistical models that assign a probability to every sequence of *words*, which reflects the probability that this sequence is written by a human. Considering a code sequence $S = t_1 t_2 \dots t_N$, it estimates the probability as

$$P(S) = P(t_1) \cdot \prod_{i=2}^{N} P(t_i | t_1, \dots, t_{i-1}) \quad (1)$$

That is, the probability of a code sequence is a product of a series of conditional probabilities. Each probability $P(t_i|t_1,\dots,t_{i-1})$ denotes the chance that the token t_i follows the previous words, the *prefix*, $h = t_1, \dots, t_{i-1}$. However, the probabilities $P(t_i|t_1,\dots,t_{i-1})$ are in practice impossible to estimate, since there are astronomically large numbers of possible prefixes. For a vocabulary of size $20,000$ and sentences with maximal length of 10, there are approximately 10^{43} different probabilities to be estimated, which is impractical.

Therefore, we need a method of grouping prefixes into a more reasonable number of equivalence classes. One possible approach to group them is by making a *Markov assumption* that the conditional probability of a token is dependent only on the $n-1$ most recent tokens. The n-gram model is such a model that places all prefixes that have the same $n-1$ tokens in the same equivalence class:

$$P(t_i|h) = P(t_i|t_1,\dots,t_{i-1}) = P(t_i|t_{i-n+1},\dots,t_{i-1}) \quad (2)$$

The latter is estimated from the training corpus as the ratio of the times that the token t_i follows the prefix sequence $t_{i-n+1},\dots,t_{i-1}$:

$$P(t_i|h) = \frac{count(t_{i-n+1},\dots,t_{i-1},t_i)}{count(t_{i-n+1},\dots,t_{i-1})} \quad (3)$$

Thus, if the fragment "`for ( int`" occurs 1000 times, and "`for ( int i`" occurs 300 times, then $p($`i` $|$ `for int (`$) = 0.3$.

2.2 Code Suggestion

The code suggestion task refers to recommending the next token based on the current context. There has been recent work on code suggestion. For instance, one thread of research [10, 20, 25, 39] concerns on the suggestion of method calls and class names. Nguyen *et al.* [34] and Zhong *et al.* [50, 51] aim to predict API calls, and Zhang *et al.* [48] focus on completing the parameter list of API calls. Eclipse (and other development environments like IntelliJ IDEA) makes heavy use of compile-time type information to predict which tokens *may apply* in the current context.

N-gram models have been successfully applied to code suggestion [18]. Hindle *et al.* [18] exploit a trigram model built from the lexed training corpus. At every point in the code, the model gives a list of possible tokens along with their probabilities that are estimated from the training corpus. The probabilities can be used to rank the candidate tokens. Then, the top-k rank-ordered suggestions are presented to the user. The advantages of the n-gram approach are the possibility to suggest all types of tokens (not just method calls), permitting the model to incorporate valuable information (indicating what *most often does apply*) that is not described by the IDE-based approaches (guess what *may apply*). The n-gram model is also very simple, and unlike more recent work [36], does not require anything beyond tokenizing the code. On the other hand, n-gram models will not help much when dealing with tokens and n-grams locally specific to a particular context, and not present in the code used to train the n-gram model.

Table 1: Java and Python projects data and English corpora. Distinct tokens constitute the vocabulary. Thus *Ant* has a total of 919,148 tokens, composed of 27,008 distinct tokens.

			Tokens	
Java Project	**Version**	**Lines**	Total	Distinct
Ant	20110123	254,457	919,148	27,008
Batik	20110118	367,293	1,384,554	30,298
Cassandra	20110122	135,992	697,498	13,002
Log4J	20101119	68,528	247,001	8,056
Lucene	20100319	429,957	2,130,349	32,676
Maven2	20101118	61,622	263,831	7,637
Maven3	20110122	114,527	462,397	10,839
Xalan-J	20091212	349,837	1,085,022	39,383
Xerces	20110111	257,572	992,623	19,542
			Tokens	
Python Project	**Version**	**Lines**	Total	Distinct
Boto	20110714	53,969	393,900	10,164
Bup	20110115	17,287	179,645	5,843
Django-cms	20110824	37,740	567,733	7,526
Django	20120215	205,862	2,137,399	55,486
Gateone	20111013	24,051	230,048	8,212
Play	20111202	231,177	1,893,611	68,928
Reddit	20100721	52,091	527,815	14,216
Sick-beard	20110408	99,586	1,005,823	53,345
Tornado	20120130	16,326	622,715	5,893
			Tokens	
English Corpus	**Version**	**Lines**	Total	Distinct
Brown	20101101	81,851	1,161,192	56,057
Gutenberg	20101101	55,578	2,621,613	51,156

Table 2: Percentage of the *endemic* n-grams (only found in *a single file*). "Freq." denotes the frequency of the n-grams in the file. The denominator is the number of n-grams in the corpus.

Lang.	Freq.	1-gram	3-gram	6-gram	10-gram
Java	$\geqslant$1	4.65%	25.63%	53.98%	70.22%
	$\geqslant$2	3.71%	11.13%	16.17%	14.34%
Python	$\geqslant$1	8.58%	41.82%	73.65%	83.79%
	$\geqslant$2	5.29%	14.44%	17.52%	12.47%
English	$\geqslant$1	3.28%	72.52%	99.70%	100.00%
	$\geqslant$2	1.39%	3.44%	0.65%	0.20%

3. LOCALNESS OF SOURCE CODE

Is source code localized? To answer this question, we analyzed both natural language and code corpora. We found evidence supporting three aspects relating to our hypothesis: the *endemism*, *specificity* and *repetitiveness* of code regularities in the locality.

Data. We collected the same data set of Java projects and English corpora used by Hindle *et al.* [18]. The data set contains nine Java projects with a total of more than 2M LOCs. To investigate whether the commonality holds across different programming languages, we also carried out experiments on another very different language, Python. Table 1 shows the summary statistics on our data.

3.1 Are N-grams Endemic to Localities?

In this experiment, we investigated whether some n-grams are only found in a local context (*endemic* to a locality). Table 2 shows the percentage of the total n-grams in the corpus that were only found in a single file.[1] For example, suppose there are 1M 3-grams extracted from one corpus, 200K of which are only found in a single file, then the percentage is 20%. We can see that 25.63% of 3-grams from the Java corpus (41.82% from the Python corpus) are <u>*found only in one file*</u>. The large proportion of n-grams only seen in a local context denotes that *source code is endemic* to file localities.

[1]In English corpus, each file is an article or a chapter of a book.

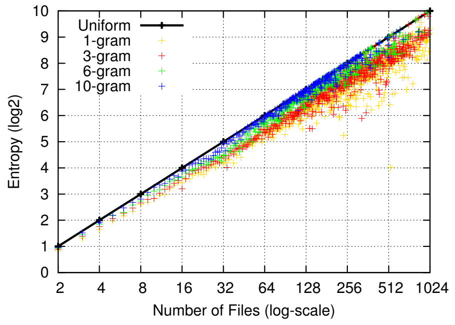

Figure 1: Entropies of the file distributions for non-endemic n-grams (grouped by the number of files) for Java. "Uniform" denotes that the n-grams are distributed uniformly in the files.

Among the endemic n-grams in source code, there are over 10% of the total 3-grams that occur more than one time in a single file; and as the order (i.e. n) of the n-grams increases, a greater proportion repeat in single files. These endemic, but locally repeating n-grams represent *an opportunity to improve upon language models* that do not take locality into account. In contrast, the percentage of the endemic, but locally repeating n-grams in English is much lower than Java and Python, especially for long n-grams (*e.g.* 0.20% for 10-grams). This validates our hypothesis that *the localness is a special feature of source code*, which cannot be found in $\mathcal{NL}$.

3.2 Is Source Code Locally Specific?

In this experiment, we investigated whether some non-endemic n-grams favor a specific locality (*locally specific*). For each non-endemic n-gram that occurs in multiple files, there would be a discrete probability distribution p for the set of files F. Here we use *locality entropy* $H_{\mathcal{L}}$ to measure the skewness of the distribution of locality of an n-gram sequence σ

$$H_{\mathcal{L}}(\sigma) = -\sum_{f \in F} p(f_\sigma) \log_2 p(f_\sigma) \qquad (4)$$

where

$$p(f_\sigma) = \frac{\text{count}(n\text{-gram } \sigma \text{ in } f)}{\text{count}(n\text{-gram } \sigma \text{ in } project)} \qquad (5)$$

Generally speaking, as $\mid F \mid$ increases, $H_{\mathcal{L}}$ increases. However, the more skewed the distribution, the lower the entropy (and the greater the opportunity for language models to exploit localization). One extreme case is the endemic n-grams that occur in only one file, then the entropy will be the lowest, *viz.,* 0. Another example is that n-grams are distributed uniformly in k files, then the entropy will be the highest, *viz.,* $\log_2(k)$. The lower the entropy, the more the n-gram σ tends to favor a smaller subset of files in F.

We place all non-endemic n-grams that occur in the same number of files in the same group, and report their mean entropy, as plotted in Figure 1. We can see that n-grams with varying orders share the same trend: the entropies of their file distributions are lower than that of a uniform distribution, indicating that the non-endemic n-grams indeed favor a specific locality. For example, the locality entropy of the 3-grams that occur in 326 files is 5.9, which is 2.5 bits lower than that of uniform distribution, *viz*, 8.4. The skewed file distribution of non-endemic n-grams reconfirms our hypothesis that n-gram models can benefit from capturing the locality.

3.3 Is Source Code Proximally Repetitive?

The degree of *proximity* of repeated n-grams to each other is relevant to the design of locality-sensitive language models: exactly

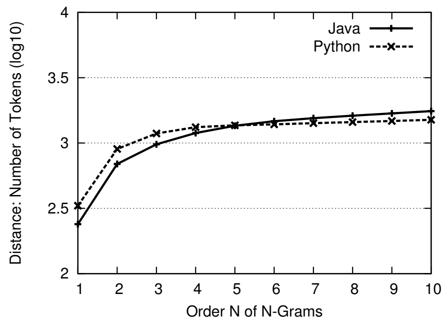

Figure 2: Distance between repetitive n-grams.

what range of proximity should language models consider when estimating a local distribution? We employ *n-gram distance* to measure how far separated the repetitive n-grams in the project are. *Distance* for a specific n-gram is defined as the average span (in term of the number of tokens) between the proximate occurrences of this n-gram. [2] For example, if "`for ( int size`" occurs 50 times in the project and there are 49,000 tokens between the first and last positions where it occurs, we say the distance of "`for ( int size`" is 1000 tokens. In other words, we may expect the n-gram to appear again on average 1000 tokens after the last location where it is previously found.

We plot in Figure 2 the variation of n-gram distance ($log10$) with different n-gram order n. Generally, the n-gram distance goes up with the increase of the order n. As seen, the n-gram distance of 10-grams for Java is 1754 ($10^{3.24}$) and for Python is 1514 ($10^{3.18}$), suggesting that *programming languages are locally repetitive*. The *highly proximate* repetitive property of the n-grams suggests that locally-sensitive language models have modest memory requirements—we do not have to track too much local information.

Source code is <u>*localized*</u> in the sense that code regularities in the locality are *endemic*, *specific*, and *proximally repetitive*.

4. CACHE LANGUAGE MODEL

4.1 Illustration

Our primary goal is to address a critical limitation of the standard traditional n-gram models: *their inability to capture local regularities*. We accomplish this by using a *locally estimated cache* component to capture the *endemic* and *specific* n-gram patterns.

Capturing the structure of the surrounding locality, in an evolving model that changes with the locality, is a good approach to capture the local regularities. First, if we memorize the endemic n-grams in the locality, we can offer the correct suggestion when they occur again, which can never be provided by the n-gram model estimated from the training data. This would be a good supplement to the global n-gram models, which do not perform well, *e.g.*, with n-grams that are endemic and perhaps not seen before.

Second, a language model that captures short-term shifts in n-gram frequencies might perform significantly better than the pure n-gram models described above. If we memorize the n-grams found in the locality, or *cache*, we can update the probability of the n-grams based on their frequency in the cache. Let us revisit the last example. From the n-gram model, we have $p(\texttt{i}|\texttt{for int (}) = 0.3$ and $p(\texttt{size} \mid \texttt{for int (}) = 0.05$. Suppose on the existing lines in the file, "`for int ( size`" occurs 3 times and "`for int ( i`" never appears. Then from the cache we have $p(\texttt{i} \mid \texttt{for int (}) = 0$ and $p(\texttt{size} \mid \texttt{for int (}) = 1.0$. If we use the average of the probabilities from the two components as the final probabilities, we will assign "`i`" a probability of 0.15 and "`size`" a probability of 0.525, and thus "`size`" is chosen.

[2]If the n-gram is found in different files, we skip the other files that the n-gram does not occur in when we calculate the distance.

Our *cache language model* is based on these intuitions; it includes both a standard n-gram and an added "cache" component. Given a source file, the cache will contain all the n-grams found in the local code. Thus, the combined model assigns each token candidate two probabilities, the first based on its frequency in the training corpus (the n-gram component) and its frequency in the cache (the cache component). Linear interpolation of the two probabilities produces an overall probability of each token candidate.

The main idea behind our work is to combine a large global (*static*) language model with a small local (*dynamic*) model estimated from the proximate local context. 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 token patterns change in different localities. The cache component models these local changes, and provides variance around the corpus mean for different local contexts. The strengths of the additional cache component are reflecting the code locality by capturing:

1. *Endemic n-gram patterns*: The cache component captures more different n-grams by memorizing the endemic n-grams in the locality, which do not occur within the n-gram set captured from the rest of the corpus. When the endemic n-grams repeat, the cache component can offer a suggestion.

2. *Specific n-gram patterns*: If a token that followed the same prefix sequence has occurred often in the locality (might span several local files), it will be assigned a higher probability than when its local frequency is low. In this way, the inclusion of a cache component satisfies our goal to dynamically track the frequency-biased patterns of n-gram use in the locality.

In Section 4.2, we describe how the cache model automatically learns to interpolate between these two models. Then, we analyze several local factors that affect the cache model's performance in Section 4.3. By setting the factors appropriately, the cache component will capture the code locality with modest computational resources (as shown in Table 6).

4.2 Mathematical Treatment

The cache model is now introduced mathematically:

$$P(t_i|h, cache) = \lambda \cdot P_{n\text{-gram}}(t_i|h) + (1 - \lambda) \cdot P_{cache}(t_i|h) \quad (6)$$

Here t_i is the token to be predicted, h is the prefix tokens that t_i follows, $cache$ is the list of n-grams that are stored in the cache, and λ is the interpolation weight. The combined model leaves the n-gram component $P_{n\text{-gram}}(t_i|h)$ of the language model unchanged. Note that the traditional n-grams model is the special case of the cache model, when $\lambda = 1$. The n-gram-based probability $P_{n\text{-gram}}(t_i|h)$ can be regarded as a good estimate of the mean around which the value $P_{cache}(t_i|h)$ fluctuates, while the cache-based probability $P_{cache}(t_i|h)$ is the variance around that mean. This allows the estimate of $P_{cache}(t_i|h)$ to deviate from its average value to reflect temporary high or low values.

Probability Estimation in Cache. The cache-based probability is calculated from the frequency of t_i followed the prefix h in the cache. We estimate the cache probability $P_{cache}(t_i|h)$ by

$$P_{cache}(t_i|h) = \frac{\text{count}(< h, t_i > \text{ in } cache)}{\text{count}(h \text{ in } cache)} \tag{7}$$

Here we map the prefix h to the $m-1$ most recent tokens. It is worth noting that the order m is not necessarily the same with the order n for the n-gram component, as discussed in Section 4.3.

Automatic Selection of Dynamic Interpolation Weights. In simple linear interpolation, the weight λ is just a single number that may be set by hand. But we can define a more general and powerful model where the weights are a function of the prefix. We assume that the cache in which more records for the prefix are found is considered more reliable. For example, if more prefix sequences are found in the cache, we want to bias ourselves more towards using the cache. To accomplish that, inspired by Knight [26], we replace $\lambda(h)$ with $\frac{\gamma}{\gamma+H}$, where H is the number of times that the prefix h has been observed in the cache, and γ is a *concentration parameter* between 0 and infinity:

$$P(t_i|h, cache) = \frac{\gamma}{\gamma+H} \cdot P_{n\text{-gram}}(t_i|h) + \frac{H}{\gamma+H} \cdot P_{cache}(t_i|h) \tag{8}$$

We can see that if the prefix occurs few times (H is small), then the n-gram model probability will be preferred. But if the prefix occurs many times (H is large), then the cache component will be preferred. This setting avoids tricky hand-tuned parameters and make the interpolation weight self-adaptive for different n-grams.

4.3 Tuning the Cache

In this section, we will discuss several factors of the locality that would affect the performance of cache language model: *cache context*, *cache scope*, *cache size*, and *cache order*.

Cache Context. *Cache context*, by definition, is the lexical context from which n-grams in the cache are *extracted*. The cache is initially designed to capture the local regularities from the preceding tokens in the current file, (the *"prolog"*, in contrast with the succeeding tokens, or *"epilog"*), and is well suited for the initial development (developing new files). However, for software development the maintenance and evolution are critical activities [2], where both prolog and epilog contexts are available. We believe our cache model will benefit from more context, which means that our model would be more useful for software maintenance, as shown in Section 5.2.3.

We hypothesized that the performance of the cache component would depend on the prolog more than the epilog. Recall that given an n-gram, the probability from the cache component along with the weight depends on its frequency in the cache and the total number of n-grams that share the same prefix (*i.e.* H) (Eq. 8). Hence, we expected different performance for the cache component built on the *prolog* and *epilog*, since there maybe different distributions of the n-grams that share the same prefix in the early and later parts of files. This turned out to be false. This aspect of results are discussed in Section 5.4.1.

Cache Scope. *Cache scope* is the part of codebase used to estimate the cache component. The intuitive way is to build the cache on the scope of the current file (*file cache*). However, one problem with the file cache is that those initial sentences in a file may not benefit from the file cache. We use a simple heuristic to alleviate the problem: we build the cache on the previous K tokens which could span files in local grouping (i.e. same subdirectory in the codebase).[3] In other words, the underlying assumption is that *software is also localized in the codebase*, in the sense that the files in the same subdirectory are locally similar. In this way, our cache model incorporates a *latent* scope to capture the code patterns in local files.

[3]The files in the same directory are ordered by the filename.

Cache Size. *Cache size* is defined as the *maximum number* of n-grams stored in the cache. Intuitively, it is more likely to observe a given n-gram if we use a larger cache. However, there has to be a trade-off between *locality* and *coverage* when we decide the size of cache. The smaller the cache is, the more local regularities the cache component can capture, at the cost of missing more possibly useful n-grams. In contrast, a larger cache would cover more n-grams while neglecting the localness of source code. An extreme case is to build the cache on the whole project, which captures only the global regularities. Moreover, maintaining a larger cache is computationally expensive. As a consequence, we need to decide an appropriate cache size to balance the locality and coverage of the cache model.

Cache Order. *Cache order* refers to the *maximum order* m of n-grams stored in the cache. Generally, the longer the prefix, the more accurate the suggestion. A longer prefix is more specific since it has more detailed information. For example, the long prefix "`for (int i=0; i<10; i`" is more likely to predict the correct suggestion "`++`" than the short prefix "`; i`". Given that a cache stores much fewer tokens than the n-gram component, we can employ a higher order m without increasing much complexity. Data sparseness problems arise when we use higher order n-grams in the cache. We employ back-off techniques [23] when a long prefix is not matched.

5. EXPERIMENTS

In this section, we try to answer the following research questions:

R1. Can the cache model capture the code locality?

R2. Can the strength be leveraged for code suggestion?

R3. Why does the cache model help?

R4. What factors affect the performance of the cache model?

In Section 5.1, we demonstrate, using standard cross-entropy over large software corpora, that the cache model is indeed capturing the *localness* of source code, for that the additional cache component estimated from the code locality decreases nearly one bit over the n-gram model. (Note that entropy is log scaled; intuitively, it means the cache model renders code nearly *twice* as predictable.)

In Section 5.2, we evaluate our cache language model for code suggestion, and show that the cache language model greatly improves suggestion accuracy for different programming languages, thus retaining the portability and simplicity of the n-gram approach. Furthermore, our cache model is especially useful for cross-project code suggestion and software maintenance.

In Section 5.3, we point out that identifiers benefit most from the cache model and contribute most to the accuracy improvement. In addition, the improvements are due to that the cache component captures the *endemic* and *specific* n-gram patterns in the locality.

In Section 5.4, we analyze four factors (described in Section 4.3) that will influence the cache model. Our analyses show that (1) there is no difference between prolog and epilog, while combining them achieves a further improvement; (2) the localness of software is also reflected at the localized structure of codebase; (3) the suggestion accuracy increases rapidly with cache size and order, saturating around 2K tokens and 6, respectively.

Setting. We performed the 10-fold cross-validation (in term of files) on each project in Table 1. We tested the statistical significance

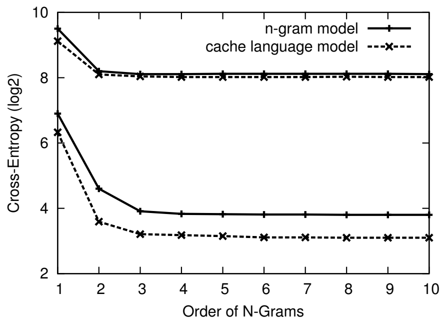

Figure 3: Cross-entropies of Java (below) and English (above). We use the same orders for both n-gram and cache models. The trend for Python is similar to Java.

using *sign-test* [14]. For the cache model, we set concentration parameter $\gamma = 1$, the cache size $K = 5000$ and the cache order $m = 10$.[4] Our baseline is the lexical trigram model (i.e. $n = 3$ for the n-gram model), which is the same with the n-gram component in our cache language model.

5.1 Can the Code Locality be Captured?

We first studied whether the localness of software can be captured by the cache model. We use cross-entropy to measure how good a language model captures the regularities in a specific corpus. Given a corpus $S = t_i \dots t_N$, of length N, with a probability $p_M(S)$ estimated by a language model M. The cross-entropy is calculated as:

$$H_M(S) = -\frac{1}{N}\log_2 p_M(S) = -\frac{1}{N}\sum_{1}^{N}\log_2 P(t_i|h) \quad (9)$$

A good model is expected to predict with high confidence the test data drawn from the same population, thus has a low entropy.

Figure 3 shows the averaged cross-entropies of all projects for both Java and English. The two lines above are the averaged cross-entropies for English corpora, while the lines below are for Java projects. For English corpora, the cache model only outperforms the n-gram model at unigram, while not for higher order n-grams. This is because English is very flexible and n-grams ($n > 1$) are hardly found in a very local cache (5K words). For Java projects, however, the cache language model outperforms n-gram model consistently, showing that the frequency of the n-gram in the recent past, captured by the cache component, is a good indicator to the probability estimation for programming language. This reconfirms our hypothesis that *programming language is quite localized* and *the localness can be captured by the cache model.*

5.2 Cache Model for Code Suggestion

We now show that the cache model can be applied to code suggestion, although *this is by no means the only application.* As mentioned earlier, good language models will be useful for other applications: code porting [35], coding standards checking [3], correcting syntax errors [12]

We emulate code suggestion in two scenarios: coding in new files where only prolog is available (Section 5.2.1 and 5.2.2), and modifying the existing files where both prolog and epilog are available (Section 5.2.3).

Metric. We use the *mean reciprocal rank* (MRR) [41] measure, which is a standard measure of the accuracy of techniques that provide ranked lists of candidate answers. For each token in the test data, the models produce a list of suggestion candidates ordered by the probability. The reciprocal rank of a suggestion list is the multiplicative inverse of the rank $rank_i$ of the real token t_i.[5] MRR averages the reciprocal ranks for all the tokens T in the test data:

$$MRR = \frac{1}{|T|}\sum_{i=1}^{|T|}\frac{1}{rank_i} \quad (10)$$

MRR is better at differentiating between the top few ranks, than the top-n accuracy measure (how often the correct suggestion in the top n suggestions). Mean reciprocal MRR=0.5 means that one may expect the correct suggestion to appear on average at about the second suggestion, and 0.33 indicates correct answer in the top 3.

Table 3: Accuracy with various settings on `Lucene`. Here "*File cache*" denotes the cache built on the previous tokens in the test file, while "*Extended cache*" denotes the cache built on the previous 5K tokens. Although SLAMC has a slight higher accuracy, our model is language independent while theirs is not.

Model	MRR	Top1	Top5
1. N-gram	51.88%	42.64%	63.97%
2. File cache	43.30%	41.65%	45.36%
3. Extended cache	54.44%	52.04%	57.45%
4. N-gram + 2	65.57%	58.30%	75.01%
5. N-gram + 3 [**Cache LM**]	68.13%	61.45%	76.72%
N-gram model [36]	N/A	53.60%	66.10%
SLAMC [36]	N/A	64.00%	78.20%

5.2.1 Impact of Components

In this experiment, we evaluated the impact of different components on code suggestion accuracy. Table 3 shows accuracy of different configurations on `Lucene`, our largest Java project. The first row is the lexical n-gram model [18]. The second and third rows show the accuracies of using only the cache and extended cache components, which are built on the file and previous 5000 tokens respectively. The fourth and fifth rows are the combinations of the n-gram and cache components.

Surprisingly, using only cache component built on the previous 5K tokens ("Extended cache") outperforms the n-gram model built on about 2M tokens ("N-gram"). The improvement is mainly due to the increase of top1 accuracy (+9.4%), indicating that local context is quite specific and accurate.[6] Using only cache component built on the current file produces worse performance, confirming our hypothesis that the localness of software also reflects at the localized scope of the codebase (Section 4.3). Combining both n-gram and cache components achieves the best performance. The absolute improvements are 16.25%, 18.81%, and 12.75% for MRR, top1 and top5 accuracies, respectively. This suggests that n-gram and cache components capture different regularities in the source code.

We also list the accuracies reported in Nguyen *et al.* [36]. We do not claim that those results are directly comparable to ours, because of potential subtle differences in tokenization, cross-validation setting, suggestion tool implementations etc, but we present them here for reference. Even though we have a lower baseline, our cache model achieves comparable performance with the semantic language model [36]. Unlike the Nguyen *et al.* model, our cache model is quite simple, language-independent and requires no extra information besides the tokens; so the results are quite encouraging.

[4]We will discuss the influence of the cache size and order in Section5.4, and γ in the future work.

[5]If the t_i is not in the suggestion list, the reciprocal rank is 0.

[6]An increase of +3.2% is still found for top 1 accuracy when we set the same order (i.e. 3) for both n-gram and cache components.

Table 4: Accuracy (MRR) of code suggestion. "Improv." denotes the absolute improvement of cache language model ("Cache LM") over the n-gram model ("N-gram"). "Sig." denotes the statistical significance tests against "N-gram" using *sign-test*.

Java Proj.	N-gram	Cache LM	Improv.	Sig.	Python Proj.	N-gram	Cache LM	Improv.	Sig.
Ant	53.78%	65.07%	11.29%	< .001	Boto	45.76%	59.50%	13.74%	< .001
Batik	50.56%	68.34%	17.78%	< .001	Bup	39.43%	50.07%	10.64%	< .001
Cassandra	53.06%	66.73%	13.67%	< .001	Django-cms	63.70%	74.34%	10.64%	< .001
Log4j	51.74%	66.47%	14.73%	< .001	Django	44.06%	61.24%	17.18%	< .001
Lucene	51.88%	68.13%	16.25%	< .001	Gateone	38.94%	56.58%	17.64%	< .001
Maven2	53.96%	68.74%	14.78%	< .001	Play	42.41%	56.28%	13.87%	< .001
Maven3	56.79%	68.99%	12.20%	< .001	Reddit	40.08%	50.53%	10.45%	< .001
Xalan	50.86%	65.44%	14.58%	< .001	Sick-beard	37.89%	50.40%	12.51%	< .001
Xerces	52.61%	70.38%	17.77%	< .001	Tornado	49.33%	63.43%	14.10%	< .001
Average	52.80%	67.59%	14.79%		Average	44.62%	58.04%	13.42%	

Table 5: Accuracy of suggestion for other languages.

Lang.	N-gram	Cache LM	Improv.
C	48.44%	62.14%	13.70%
PHP	56.98%	68.71%	11.73%
Javascript	49.28%	61.72%	12.44%

Table 6: Speed and Memory Comparison. "μs" denotes microsecond (10^{-6} second).

	Speed (μs/token)		**Memory (Mb)**	
Lang.	N-gram	Cache LM	N-gram	Cache LM
Java	6.8	23.4	15.4	16.6
Python	6.9	21.2	12.6	13.8

Cache Model Portability. In this experiment, we compared our cache model with the lexical n-gram model in two data sets of Java and Python. Table 4 lists the comparison results. For Java projects, the cache model achieves an averaged improvement of 14.79% in MRR from 52.80% to 67.59% over the lexical n-gram model. For Python projects, the improvement is 14.01% from 40.26% to 54.27%. All absolute improvements are statistically significant at $p < 0.001$ using *sign-test* [14]. Table 5 shows the result on three other languages: C, PHP, and Javascript. As seen, our cache model significantly ($p < 0.001$) outperforms the n-gram approach consistently, indicating that *the improvement of our cache model is language-independent*.

Cache Model: Resource Usage. We measured the computational resources used by the cache model. See Table 6; the memory usage increases slightly, and performance is 3X slower than the n-gram model. We contribute the increase to the additional maintenance of cache and the real-time calculation of cache probabilities along with their weights. Note that the probabilities from the n-gram model are calculated offline and only one inquiry is needed for each token. In contrast, the cache is dynamically updating during the whole suggestion procedure, and thus the cache probabilities depend on the dynamic counts of n-grams and are calculated online. Even so, the code suggestion based on the cache model is fast enough to be imperceptible for interactive use in an IDE. It should be emphasized that the additional memory of the cache only depend on the size of cache (not shown) while is independent of the training corpus.

Case Study. Here are some interesting cases to show why our cache model improves the performance. The variable `heap` is only found in the file `CBZip2OutputStream.java` in `Ant`. It occurs 70 times, following 12 different prefixes, of which 11 occur more than 1 time. The n-gram model does not suggest correctly, whereas the cache model does when the n-grams are found in the cache.

In `Lucene`, the identifier `search` works as various roles (*e.g.* method, variable) and occurs 3430 times. The top 3 prefixes that it follows are "`lucene .`" (1611), "`searcher .`" (414), "`solr .`" (366). Given a specific file `FrenchStemmer.java`, it occurs 35 times and works as a variable (`String[] search`). The top3 prefixes are "`[ ]`" (8), "`i <`" (5), "`endsWith (`" (5). The cache model recommends correctly when the n-grams occur again, while the n-gram model does not.

Table 7: Cross-project code suggestion (Java).

Proj.	N-gram	Cache LM	Improv.	Sig.
Ant	47.13%	62.41%	15.28%	< .001
Batik	41.83%	65.65%	23.82%	< .001
Cassandra	41.54%	63.55%	22.01%	< .001
Log4j	46.13%	64.63%	18.50%	< .001
Lucene	40.82%	64.72%	23.90%	< .001
Maven2	58.48%	70.61%	12.13%	< .001
Maven3	53.08%	68.15%	15.07%	< .001
Xalan	41.28%	61.98%	20.70%	< .001
Xerces	45.32%	68.29%	22.97%	< .001
Average	46.18%	65.55%	19.37%	

5.2.2 Cross-Project Code Suggestion

We performed another experiment to simulate a new, "greenfield" project setting, where training data can only be obtained from other projects. Most data-driven approaches, including language models, are subject to the well-known problem of lack of portability to new domains/projects. Usually there is a substantial drop in performance when testing on data from a project different from the training data. Our data suggests that the cache model can alleviate the cross-project problem. Table 7 shows the results of cross-project code suggestion. For each project we performed 10-fold cross-validation as in the previous experiment. The difference is that we used the other eight Java projects rather than the other nine folds for training. As seen, the lexical n-gram model has a accuracy of 46.18%, which is 6.62% lower than that of the in-project setting reported in Table 4, which empirically reconfirms the cross-project problem of language models. Comparing the accuracies in Tables 4 and 7, we can see that the gap between the cross-project and in-project settings decreases from 6.62% (46.18% versus 52.80%) to 2.04% (65.55% versus 67.59%). We attribute this to that the cache component, built on a few n-grams in a quite local context, makes up for the loss of the project-specific code regularities to some extent. This also indicates that programming language is quite localized, since project-specific code patterns are grouped locally.

5.2.3 Code Suggestion in Software Maintenance

In this experiment, we emulate code suggestion in software maintenance, where we change existing files rather than writing new files. When predicting a token, we build the cache on the rest tokens

Table 8: Maintaining code suggestion (Java). The cache is built only on the current file rather than the previous 5K tokens.

Proj.	N-gram	Cache LM*	Improv.	Sig.
Ant	53.78%	67.07%	13.29%	< .001
Batik	50.56%	70.05%	19.49%	< .001
Cassandra	53.06%	69.10%	16.04%	< .001
Log4j	51.74%	68.31%	16.57%	< .001
Lucene	51.88%	70.28%	18.40%	< .001
Maven2	53.96%	70.06%	16.10%	< .001
Maven3	56.79%	70.81%	14.02%	< .001
Xalan	50.86%	65.98%	15.12%	< .001
Xerces	52.61%	72.24%	19.63%	< .001
Average	52.80%	69.32%	16.52%	

Table 9: Descriptions of token abstractions.

Abstraction	Description	Examples
ID_MCALL	method calls	getMessage(), call()
ID_TYPE	types	QueryNode, ScoreMode
ID_VAR	variables	i, size, node
LIT	literals	"String", 'a', 270
KW	keywords	int, for, while
OP	operators	>, <, +, =, -
SEP	separators	{, }, (,), [,], ;

in the current file. It is different from previous settings at that we incorporate both *prolog* and *epilog* contexts here rather than only exploiting the prolog context in the above experiments. Comparing Tables 4 and 8, we found that our cache model is even more useful for software maintenance, by achieving significant improvement in suggestion accuracies (69.32% versus 67.59%) with less tokens (0.9K versus 5K). This result reconfirms our hypothesis that *source code is localized at the file level*.

5.3 Why Does the Cache Model Help?

In this section, we investigated the reasons why the cache model works. We first checked which tokens benefit most from our cache model, then investigated whether our cache model is indeed capturing the localized regularities of both *endemic n-gram patterns* and *frequency-biased patterns of non-endemic n-grams*.

5.3.1 Token Abstraction Analysis

In this experiment we investigated which tokens benefit most from our cache model. We divided the tokens into 7 classes of token abstractions: method calls (ID_MCALL), type identifiers (ID_TYPE), variables (ID_VAR), literals (LIT), keywords (KW), operators (OP), and separators (SEP). The first four classes belong to open-vocabulary (unlimited number of vocabulary) while the latter three classes are close-vocabulary (limited number of vocabulary). Table 10 lists the results for different token abstractions.

As one would expect, the identifiers have a low suggestion accuracy (ranging from 19.83% to 28.25%), compared with the overall accuracy of 52.80%. A recent study of code cross-entropy [4] shows that identifiers contribute most to the uncertainty of the source code. Our results reconfirm these findings. We have found that the use of local caches drastically improves the suggestion accuracy of identifiers (+21.91% to +27.38%).

In contrast, the programming-specific abstractions (i.e., *keywords*, *operators*, and *separators*), have relatively higher accuracy. For example, the separators, which take a large proportion of the total tokens, have a suggestion accuracy of 75.20%. Consequently, their improvements from the cache model are relatively small (8.41% to 13.10%). This is because the regularities of these abstractions are more specific to the programming language syntax, which can be captured by the lexical n-gram models from the training data.

Table 11: Improvements from different patterns captured by the cache component. "Endemic" and "NonEndemic" denote the *endemic* n-gram patterns and the frequency-biased patterns of *non-endemic* n-grams, respectively.

	Java		Python	
Abstraction	Endemic	NonEndemic	Endemic	NonEndemic
ID_MCALL	+21.81%	+5.57%	+12.92%	+0.92%
ID_TYPE	+16.74%	+5.61%	+13.67%	+2.81%
ID_VAR	+17.73%	+4.18%	+16.33%	+2.89%
LIT	+15.24%	+2.95%	+11.43%	+1.78%
KW	+5.06%	+5.61%	+6.15%	+4.35%
OP	+4.73%	+8.37%	+6.05%	+5.24%
SEP	+3.26%	+5.15%	+5.39%	+4.91%
ALL	+9.30%	+5.42%	+9.57%	+3.85%

Case Study. In `Lucene`, the most frequent method `length` denotes the function of *getting the length of a object*, and therefore it usually follows "ID_VAR .". The method obtains improvements of 30.33%, 39.72%, and 5.28% in MRR, Top1, and Top10 accuracies respectively, indicating that most of the improvement is contributed by reranking the candidates.

The keyword (KW) `new` occurs 25459 times in `Lucene`, and achieves an improvement of 7.59% in MRR by using the cache model. 75% of the prefixes that it follows contain at least one identifier or literal. For example, nearly half of the prefixes are the instances of the pattern of *constructing a new object and assigning it to a variable*. This suggests that the keywords also benefit from the locality of identifiers and literals.

5.3.2 N-gram Patterns Analysis

In this experiment, we investigated whether our cache model is indeed capturing the localized regularities of *endemic n-gram patterns* and of *non-endemic but specific n-grams*. We use the terms "Endemic" and "NonEndemic" to denote the improvements from the two kinds of patterns above, respectively. We distinguished the accuracy improvement for a given token as follows. If the correct suggestion only occurs in the cache component, we attribute the improvement to the capture of *endemic n-gram patterns* from the cache component; otherwise, it is from the capture of *non-endemic but specific n-grams*, since the correct suggestion is found in both n-gram and cache components; however, it is assigned a higher rank because the cache component correctly captures the more local usage patterns.

Table 11 lists the improvements (endemic and non-endemic) for different token abstractions on Java projects. Several observations can be made. First, the cache component indeed captures both the endemic n-grams patterns and the non-endemic (but locally specific) patterns of n-grams. They achieve the absolute accuracy improvements of +9.30% and +5.42% respectively on Java, and +9.57% and +3.85 on Python. Second, different abstractions benefit differently from the two kinds of patterns. For identifiers (ID_*) and literals (LIT), the coverage of endemic n-grams improves the most. This jives with intuition, because a large portion of new identifiers introduced by new functions and files can only be captured by the cache component. In contrast, the language-specific tokens (KW, OP, and SEP) benefit similarly from both patterns.

5.4 What Factors Affect the Cache Model?

Next, we investigate several factors' influence on the cache model.

5.4.1 Cache Context

To investigate the influence of cache context on the cache model, we build the cache on different kinds of contexts *in the current file*: *prolog*, *epilog*, and *both*. It came as a surprise to us that, in general,

Table 10: Improvements for different abstractions.

Abstraction	Percent	In-project N-gram	In-project Cache LM	In-project Improv.	Cross-project N-gram	Cross-project Cache LM	Cross-project Improv.
Java							
ID_MCALL	6.20%	19.83%	47.21%	27.38%	12.91%	45.43%	32.52%
ID_TYPE	7.15%	23.98%	46.33%	22.35%	16.84%	44.62%	27.78%
ID_VAR	17.77%	28.25%	50.16%	21.91%	19.08%	48.47%	29.39%
LIT	7.20%	26.21%	44.41%	18.20%	22.45%	42.41%	19.96%
KW	10.35%	54.34%	65.00%	10.66%	51.07%	63.90%	12.83%
OP	14.78%	68.20%	81.30%	13.10%	59.67%	78.83%	19.16%
SEP	36.54%	75.20%	83.61%	8.41%	69.44%	81.28%	11.84%
ALL	100%	52.80%	67.59%	*14.79%*	46.18%	65.55%	*19.37%*
Python							
ID_MCALL	2.38%	13.44%	27.28%	13.84%	6.81%	25.74%	18.93%
ID_TYPE	3.62%	21.92%	38.40%	16.48%	8.83%	33.04%	24.21%
ID_VAR	25.03%	22.35%	41.57%	19.22%	15.18%	39.75%	24.57%
LIT	13.54%	20.15%	33.36%	13.21%	10.26%	30.23%	19.97%
KW	8.15%	39.15%	49.66%	10.51%	33.15%	47.09%	13.94%
OP	14.24%	66.84%	78.14%	11.30%	60.32%	76.10%	15.78%
SEP	33.03%	67.41%	77.71%	10.30%	59.24%	74.89%	15.65%
ALL	100%	44.63%	58.04%	*13.41%*	36.27%	55.62%	*19.35%*

Table 12: The influence of *cache context* on the code suggestion accuracy. We build the cache on the current file.

Lang.	Model	Cache Context Prolog	Epilog	Both
Java	cache	41.47%	41.53%	52.98%
	+ n-gram	64.96%	65.00%	69.32%
Python	cache	36.53%	36.50%	47.73%
	+ n-gram	56.55%	57.17%	62.43%

Table 13: The influence of *cache scope* on the code suggestion accuracy. We build the cache on the previous 5K tokens.

Lang.	Model	File Order Ordinal	Reverse	Random
Java	cache	53.42%	53.44%	47.51%
	+ n-gram	67.59%	67.59%	66.56%
Python	cache	42.81%	42.81%	42.01%
	+ n-gram	57.23%	57.22%	56.86%

there is no difference between the prolog and epilog, as shown in Table 12. We had naively assumed that the cache built on prolog would reflect the "burstiness" of the n-grams, and hence outperforms that on epilog. It turned out to be wrong. The results suggest that code locality holds its trend in the file. Intuitively, the cache model benefits from more context, reaffirming our finding that the cache model is especially useful for code suggestion.

5.4.2 Cache Scope

It should be emphasized that the cache component is built on the previous $5K$ tokens, which may across different files. Therefore, the performance of our model can be influenced by the order of files. Table 13 lists the suggestion results with different file orders: "Ordinal" (the default setting in above experiments) and "Reverse" denote that all localized files in the same subdirectory are grouped together and are sorted in *ascending* and *descending* order respectively, while "Random" denotes that all files are in randomized order and the localized files are not grouped together. First, using only the cache component built on the localized files, either from ordinal or reverse order, outperforms that on the random files consistently for both languages, indicating that *the localness of code is also reflected at the localized structure of codebase*. The improvement for Python

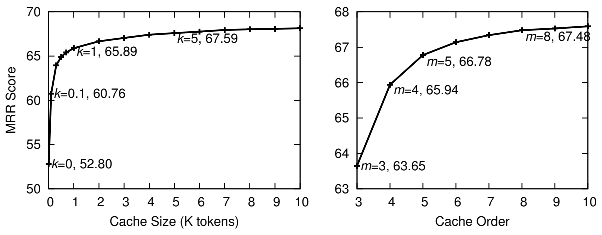

Figure 4: The influence of *cache size* and *order* on the suggestion accuracy (Java). The trend for Python is similar to Java.

is not so obvious as Java. One possible reasons is that in Python projects a file consists of multiple related classes. Second, working together with n-gram component (*i.e.* + n-gram), we achieve similar performances for different settings. We conjecture that all settings exploit the previous tokens in the current file, which captures most of the localized regularities. This suggests that our cache model is robust by taking advantages of both n-gram and cache components.

5.4.3 Cache Size and Order

The cache size and order have similar trends of the variation of MRR scores with the increase of the value, as plotted in Figure 4. The MRR score increased rapidly with cache size, saturating around 2K tokens. It is worth mentioning that we have already achieved an absolute improvement of 12.1% when the cache size $K = 500$, which confirms our hypothesis that the local knowledge of source code is assembled through a specific contextual piece. When we employed an unlimited cache (we stored in the cache all the previous tokens in the test data), the suggestion accuracy is 68.86%, which is only 1.27% higher than when $K = 5000$. This suggests that *localized regularities cannot be captured beyond the specific context.*

Similarly, the accuracy went up sharply when the order m increased from 3 to 6, while was improved slowly when the order continue increased. One possible reason is that prefixes of 5 tokens are precise enough for the suggestion task and longer n-grams (*e.g.* 10-grams) that are locally repetitive are usually extended from these shorter n-grams (*e.g.* 6-grams).

Threats to Validity and Limitations. The most likely threat to the validity of our results is the corpus we used. Although we chose many projects with large numbers of LOCs, we cannot say for sure how representative our corpus is in practice. Nevertheless, the commonality we have seen across different programming languages gives us confidence that our results hold generally.

The threat to internal validity includes the influence of the settings for the cache model (e.g. γ and cache scope). For instance, the cache model would be affected by the different orders of the the localized files under the same subdirectory (Section 5.4.2). However, the cache model built on the current file regardless of the localized files, still achieves over 12% improvement in MRR score, which is only 2.6% lower than that built on the localized files (Table 3). For code suggestion in software maintenance where both prolog and epilog contexts are available, the cache model built on the current file achieves comparable performance with that built on the localized files (Section 5.2.3).

6. RELATED WORK

Applying NLP to Software Engineering. As more repositories of open source software have become publicly available (*e.g.* on GitHub and BitBucket), software engineering researchers have turned to empirical methods to study the process of developing and maintaining software [11, 27, 38, 40, 42–45, 47, 49]. Statistical language modeling has also emerged as an approach to exploit this abundance of data [5, 21, 32, 33]. Gabel and Su [16] reported on the non-uniqueness of even large code fragments; Hindle *et al.* [18] showed that n-gram models could capture the predictable properties of source code and thus support code suggestion. Along the same direction, to alleviate the data sparseness problem that n-gram models face, Allamanis and Sutton [4] learned language models over more data from different domains, while Nguyen *et al.* [36] exploited a more general unit–semantic tokens.

Our approach is complementary to theirs: it captures different regularities of source code, that is, the *localness of software*. The simplicity of our cache model makes it broadly applicable, and easy to incorporate into other approaches: just use an interpolation of the cache model estimated probability with that from an existing model. Notably, Allamanis and Sutton [4] showed that the identifiers contribute most to the uncertainty of source code, even on a giga-token corpus. Our approach works especially well for the identifiers.

Analysis of Identifiers. Identifiers take up the majority of source code tokens [9], hence play an important role in software engineering in both code cognition [31] and understanding [28]. It has been shown that identifiers are most difficult to predict [4]. Our results confirm this finding: the n-gram approach only achieved suggestion accuracies of 22.11% to 26.05% on identifiers, which is half of other types (it is even worse in cross-project code suggestion). With the help of the cache component, it greatly improves the suggestion accuracies of identifiers.

There has been a line of research on predicting the identifiers based on the contextual information [10, 19, 25, 34, 39, 48]. For instance, Holmes *et al.* [19] and Kersten and Murphy [25] incorporated contextual information to produce better recommendations of relevant API examples. Bruch *et al.* [10] and Robbes and Lanza [39] concerned predicting the most likely method calls from prior knowledge in similar scenarios. Nguyen *et al.* [34] used graph algorithms to predict API which is similar to the current code. Our approach is generally complementary: the prediction can be improved using local statistics of corpus.

Local regularities have been exploited in the past to help splitting source code identifiers [1, 13, 15, 17, 30]. They focused on splitting compound identifiers based on the observation that terms composing identifiers usually appear in a local context (e.g. method or file).

Cache LM in $\mathcal{NL}$ Community. It is worthy emphasizing that our cache language model is different from that is used in $\mathcal{NL}$ community. As noted in Section 3.1, natural language is not as *locally repetitive* as programming languages at the levels of n-grams. Therefore, the cache component in NLP is usually a unigram model and working with class language models [29]. A typical class language model [8] first predicts the classes of the words (*e.g.* grammatical part-of-speech), then transforms the classes to the words on the basis of their frequency within the classes. Nguyen *et al.* [36] implemented a similar cache LM as in NLP, where semantic tokens worked as word classes and the cache for variables stored all the variables that belong to the same or containing scope in the search path for each semantic token.

While their work does not explicitly use a cache language model to capture the general "localness" of software over a large corpus, the differences lie in: (1) we directly build the cache model at the granularity of n-grams for programming languages; (2) we cache the n-grams of all types of tokens, not only variables; and (3) we don't require any additional information (*e.g.* type information, code path) which is difficult and expensive to produce. Our work is based on the observation that programming languages are far more regular than natural language [18] and code fragments of surprisingly large size tend to reoccur [16]. The main advantage of our cache language model is that the cache component exploits more contextual information and makes more precise suggestions. Therefore, the probabilities of the cache component is directly interpolated with those from the n-gram component. Our additional caching component achieves better improvement in top1 accuracy than that used in Nguyen's work [36] on the same data: +9.4% versus +0.7%. We contribute the improvement to the broader coverage of tokens and more expressive prefixes exploited in the cache component.

7. CONCLUSION

The cache language model introduced here captures the *localness of software*. It augments n-gram models with a *cache* component to capture the *endemic* and *specific* n-gram patterns in the locality. Experimental results show that the cache language model captures the localized regularities in the source code. Consequently, the suggestion engine based on the cache model improves the suggestion accuracy over the n-gram approach, especially in the cross-project setting and during software maintenance. It should be emphasized that our cache language model is quite simple and requires no additional information other than the tokens, thus is applicable to all programming languages. Furthermore, our approach is complementary to most state-of-the-art works since they capture different regularities of the source code.

Besides the simple n-gram models, our method is also applicable to other paradigms such as the semantic language model [36]. Another interesting direction is to further explore the potential of the cache, such as building the cache on more semantic scopes (e.g. import the same header files or packages, or files that are changed together with the current files), or introducing a decaying factor to account for recency (i.e. n-gram distance) in the cache.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1247280. We thank the three anonymous reviewers for their insightful comments on an earlier draft of the paper. We gratefully acknowledge the help of Hao Zhong, who provided us with invaluable suggestions, of Bogdan Vasilescu, who gave a careful check of typos.

9. REFERENCES

[1] S. L. Abebe and P. Tonella. Automated identifier completion and replacement. In *CSMR*, pages 263–272. IEEE, 2013.

[2] G. Alkhatib. The maintenance problem of application software: an empirical analysis. *Journal of Software Maintenance: Research and Practice*, 4(2):83–104, 1992.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. *ACM SIGSOFT FSE*, 2014.

[4] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modelling. In *MSR*, pages 207–216, 2013.

[5] S. C. Arnold, L. Mark, and J. Goldthwaite. Programming by voice, VocalProgramming. In *ASSETS*, pages 149–155. ACM, 2000.

[6] L. R. Bahl, F. Jelinek, and R. L. Mercer. A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(2):179–190, 1983.

[7] P. F. Brown, J. Cocke, S. A. D. Pietra, V. J. D. Pietra, F. Jelinek, J. D. Lafferty, R. L. Mercer, and P. S. Roossin. A statistical approach to machine translation. *Computational Linguistics*, 16(2):79–85, 1990.

[8] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479, 1992.

[9] M. Broy, F. Deißenböck, and M. Pizka. A holistic approach to software quality at work. In *3WCSQ*, 2005.

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

[11] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *ASE*, pages 33–42. ACM, 2010.

[12] J. C. Campbell, A. Hindle, and J. N. Amaral. Syntax errors just aren't natural: improving error reporting with language models. In *WCRE*, pages 252–261. ACM, 2014.

[13] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *WCRE*, pages 112–122. IEEE, 1999.

[14] W. J. Dixon and A. M. Mood. The statistical sign test. *Journal of the American Statistical Association*, 41(236):557–566, 1946.

[15] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *MSR*, pages 71–80. IEEE, 2009.

[16] M. Gabel and Z. Su. A study of the uniqueness of source code. In *FSE*, pages 147–156. ACM, 2010.

[17] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc. Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software: Evolution and Process*, 25(6):575–599, 2013.

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

[19] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[20] D. Hou and D. M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *ICSM*, pages 233–242. IEEE, 2011.

[21] F. Jacob and R. Tairas. Code template inference using language models. In *ACMSE*, pages 104:1 – 104:6. ACM, 2010.

[22] F. Jelinek, L. Bahl, and R. Mercer. Design of a linguistic statistical decoder for the recognition of continuous speech. *IEEE Transactions on Information Theory*, 21(3):250–256, 1975.

[23] S. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, volume 35, pages 400–401. IEEE, 1987.

[24] M. D. Kernighan, K. W. Church, and W. A. Gale. A spelling correction program based on a noisy channel model. In *COLING*, pages 205–210. Association for Computational Linguistics, 1990.

[25] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *FSE*, pages 1–11. ACM, 2006.

[26] K. Knight. Bayesian inference with tears. *Tutorial Workbook*, 2009.

[27] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE*, pages 372–381. ACM, 2005.

[28] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[29] R. Kuhn and R. D. Mori. A cache-based natural language model for speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(6):570–583, 1990.

[30] D. Lawrie, H. Feild, and D. Binkley. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12(4):359–388, 2007.

[31] B. Liblit, A. Begel, and E. Sweetser. Cognitive perspectives on the role of naming in computer programs. In *PPIG*. Psychology of Programming Interest Group, 2006.

[32] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61. ACM, 2005.

[33] D. Movshovitz-Attias and W. W. Cohen. Natural language models for predicting programming comments. In *ACL*, pages 35–40. Association for Computational Linguistics, 2013.

[34] 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.

[35] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Lexical statistical machine translation for language migration. In *FSE*, pages 651–654. ACM, 2013.

[36] 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.

[37] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. In *ICSE*, pages 408–417. IEEE, 1984.

[38] S. Rastkar, G. C. Murphy, and A. W. Bradley. Generating natural language summaries for cross-cutting source code concerns. In *ICSM*, pages 103–112. IEEE, 2011.

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

[40] C. Rolland and C. Proix. A natural language approach for requirements engineering. In *Advanced Information Systems Engineering*, pages 257–277. Springer, 1992.

[41] C. Shah and W. B. Croft. Evaluating high accuracy retrieval techniques. In *SIGIR*, pages 2–9. ACM, 2004.

[42] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and

K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *AOSD*, pages 212–224. ACM, 2007.

[43] D. Shepherd, L. Pollock, and T. Tourwé. Using language clues to discover crosscutting concerns. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–6, 2005.

[44] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *ASE*, pages 43–52. ACM, 2010.

[45] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *ICSE*, pages 101–110. ACM, 2011.

[46] R. Srihari and C. Baltus. Combining statistical and syntactic methods in recognizing handwritten sentences. In *AAAI Symposium: Probabilistic Approaches to Natural Language*, pages 121–127, 1992.

[47] W. F. Tichy and S. J. Koerner. Text to software: developing tools to close the gaps in software engineering. In *FoSER*, pages 379–384. ACM, 2010.

[48] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical API usage. In *ICSE*, pages 826–836. IEEE, 2012.

[49] H. Zhong and Z. Su. Detecting API documentation errors. In *OOPSLA*, pages 803–816. ACM, 2013.

[50] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: mining and recommending API usage patterns. In *ECOOP*, pages 318–343. Springer, 2009.

[51] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring specifications for resources from natural language API documentation. *Automated Software Engineering*, 18(3-4):227–261, 2011.