Are Deep Neural Networks the Best Choice for Modeling Source Code?

Vincent J. Hellendoorn
Computer Science Dept., UC Davis
Davis, CA, USA 95616
vhellendoorn@ucdavis.edu

Premkumar Devanbu
Computer Science Dept., UC Davis
Davis, CA, USA 95616
ptdevanbu@ucdavis.edu

ABSTRACT

Current statistical language modeling techniques, including deep-learning based models, have proven to be quite effective for source code. We argue here that the special properties of source code can be exploited for further improvements. In this work, we enhance established language modeling approaches to handle the special challenges of modeling source code, such as: frequent changes, larger, changing vocabularies, deeply nested scopes, etc. We present a fast, nested language modeling toolkit specifically designed for software, with the ability to add & remove text, and mix & swap out many models. Specifically, we improve upon prior cache-modeling work and present a model with a much more expansive, multi-level notion of locality that we show to be well-suited for modeling software. We present results on varying corpora in comparison with traditional N-gram, as well as RNN, and LSTM deep-learning language models, and release all our source code for public use. Our evaluations suggest that carefully adapting N-gram models for source code can yield performance that surpasses even RNN and LSTM based deep-learning models.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

naturalness, language models, software tools

ACM Reference format:
Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106290

1 INTRODUCTION

There has been much interest in the idea of "naturalness": *viz.,* modeling and exploiting the repetitive nature of software using statistical techniques from natural language processing (NLP) [17, 26, 38]. Statistical models from NLP, estimated over the large volumes of code available in GitHub, have led to a wide range of applications in software engineering. High-performance language models are widely used to improve performance on NLP-related tasks, such as translation, speech-recognition, and query completion; similarly, better language models for source code are known to improve performance in tasks such as code completion [15]. Developing models that can address (and exploit) the special properties of source code is central to this enterprise.

Language models for NLP have been developed over decades, and are highly refined; however, many of the design decisions baked-into modern NLP language models are finely-wrought to exploit properties of natural language corpora. These properties aren't always relevant to source code, so that adapting NLP models to the special features of source code can be helpful. We discuss 3 important issues and their modeling implications in detail below.

Unlimited Vocabulary Code and NL can both have an unbounded vocabulary; however, in NL corpora, the vocabulary usually saturates quickly: when scanning through a large NL corpus, pretty soon, one rarely encounters new words. New proper nouns (people & place names) do pop up–but do so infrequently. Code is different; while each language only has a fixed set of keywords and operators, new identifier names tend to proliferate [4].
Modeling Implications: In NLP, it's *de regeur* to limit vocabulary to the most common *e.g.*, 50,000 words in a pre-processing step, before model estimation. Words outside this vocabulary are treated as an unknown word, or omitted entirely. This artificially limits the space of events over which to distribute probability mass. Similarly, numerals and strings are replaced with generic tokens. This works for NLP, since words outside the dominant vocabulary are *so* rare. Virtually all work in modeling of source code borrows this approach. In source code, given the constant vocabulary innovation, this approach is not appropriate. We demonstrate that a closed vocabulary (even if large) does indeed negatively affect performance (Section 5.4), and introduce methods to address this.

Nested, Scoped, Locality While developers do invent new names for variables, classes and methods, the *repeated use* of these names tends to be localized. In Java, *e.g.*, local variables, parameters and private methods can be introduced & used, repeatedly, in one scope, and never used elsewhere. The package structures in large systems can introduce nesting of such vocabulary scopes, with different identifiers going in and out of use as one traverses the package hierarchy [7, 24, 34, 36]. Researchers have even noted application- and developer-specific vocabularies [35].
Modeling Implications: This type of nested, scoped vocabulary innovation is accompanied by corresponding repetition, where certain

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106290

code structures involving specific local names repeat, locally, within their own nested scopes. This requires a nested modeling approach, which captures the local repetition within a scope s_i, and then makes it available to scopes $s_i, s_{i+1} \ldots$ nested within s_i. Furthermore, if such nested models are used within an interactive tool (such as an IDE) the model would need to be rapidly re-estimated as the programmer's working context changes.

Dynamism Evolution is normal for well-used software systems; bug fixes and new features keep rolling in. NLP corpora evolve much more slowly. Furthermore, during interactive coding, software tools must quickly adjust to new localities and contexts: in a single coding session, a developer may open and close many files. As she explores the code, a language model that works within the IDE (for code completion [15, 33], defect localization [31], *etc.*) must rapidly adapt to the working context.

Modeling Implications: Traditional NLP models cannot handle rapid re-estimation. Deep-learning models, in particular, are not very dynamic, and re-estimation is very slow.

In response to the observations and concerns raised above, we have developed a *dynamic, hierarchically scoped, open vocabulary language model* for source code, that achieves best-in-class performance when using non-parametric (count-based) language modeling. We make the following contributions:

- We introduce *mixed, scoped models* to handle arbitrary nesting and mixing of N-gram models.
- We implement these models using a fast datastructure optimized for *dynamic, scoped counting* of language events.
- We compare several popular smoothing techniques in related work and show that a simple approach (not typically used) works better than others.
- Finally, we evaluate the performance of these models on a large corpus of Java code in comparison and combination with implicit (deep-learning based) models. We find that our model outperforms the RNN and LSTM deep learning models, achieving *unprecedented* levels of entropy & also performance on the code-suggestion task. We also show that our approach adds value, even to LSTM models.

Our runnable API, code and replication details can be found on github.com/SLP-Team/SLP-Core

2 BACKGROUND

We now review language models, including explicit (or non-parametric, count-based) models and implicit (here: deep learning) models.

2.1 Language Models & Performance

Language models assign a probability to (or *"score"*) an utterance (*e.g.*, a phrase, a sentence, or just a word). Models are estimated on large corpora of natural text. A good language model should score an utterance high, if it would sound "natural" to a native speaker, and score low the unnatural (or wrong) sentences. Accurate scoring matters in tasks like machine translation, where the output sentence should "sound normal" and likewise, also in speech recognition, summarization and spell checkers. Similarly, in a source code environment, language migration [20, 27], synthesis of code from natural language and *vice versa* [5, 14, 28, 30] as well as code suggestion engines [15, 33] all need models that score code fragments accurately.

The usual way to score a code fragment s of length $|s|$ is to tokenize it into, *e.g.*, $t_1, t_2, \ldots$ and score each next token t_i given the previous tokens, *i.e.*,:

$$p(s) = \prod_{i=1}^{|s|} p(t_i \mid t_0, \ldots, t_{i-1}) \tag{1}$$

This yields both per-token probabilities and a single probability for the entire phrase. More generally, each token is predicted from its *context c*, including its preceding tokens and perhaps additional information (*e.g.*, from a previous phrase, or the topic of a document). Since the probabilities may vary by orders of magnitude, one often uses the (typically negated) logarithm of the phrase probability, to arrive at the information-theoretic measure of *entropy*:

$$H_p(s) = -\frac{1}{|s|}\log_2 p(s) = -\frac{1}{|s|}\sum_{i=1}^{|s|}\log_2 p(t_i \mid c)$$

Entropy reflects the number of bits needed to encode the phrase (and, analogously, a token) given the language model. An alternative metric, often seen in NLP literature, is *perplexity*, which is simply $2^{H_p(s)}$ and accentuates differences between higher entropy scores.

In Equation (1), the probability of a token in a phrase is calculated given all previous tokens. In general, this isn't practical, once the corpus gets big enough. There are two ways to approach this problem: using **explicitly defined rules** and using **implicit state**.

2.2 Explicit Language Models

Explicit modeling requires a restriction of the relevant context; this approach is quite mature. Three prior classes of models are based on *N-grams*, on extracted long-distance *Dependencies*, and on the use of *Caches*. These models generally require *smoothing*, to account for rare (or unseen) events in the training data that may behave differently in the test data. We now discuss these ideas.

***N*-gram language models:** are easily constructed (and popular). These models simplify Equation (1) with a Markov-assumption: each token is conditioned on just $N-1$ preceding tokens. We can then score a token using maximum likelihood of its occurrence in some context. For instance, in source code, the score for "`i`" given the context "`for ( int`" is very high, since the former frequently occurs in the latter context. Capturing this information would require a 4-gram model, to count sequences up to for 4 tokens.

Dependency models: can capture long-distance dependencies between tokens, rather than dependencies between tokens that are sequential in the text[1], which are sometimes more effective than Markovian dependencies in left-to-right parses of natural language [9, 10]. Similar models have been proposed in source code, using dependencies extracted by compilers. Researchers have modeled API invocations as a graph prediction problem [26], and code completion by conditioning on identifiers in scope or parent nodes in the AST [24].

Cache models: augment N-gram model with an additional ("cache") N-gram model to track just the local changes. The two models can

[1] *E.g.* the dependency from *walked* to *dog* in *The girl walked the restless yellow dog.*

be mixed using the cache's "confidence" at prediction time (*e.g.*, based on how often it has seen the context, as in [36]). These are distinct from dynamic models, where the trained model is updated with all information at test time, since caches are strictly local: they can "forget" events (*e.g.*, because caches are limited in size), or be swapped out depending on the context (caches are re-initialized when a new file is encountered). We expand on this notion in this work (Section 3).

Smoothing N-gram models come with a trade-off: longer N-gram contexts are more specific, but correspondingly less frequent in the corpus; thus may occur in the test corpus but not in the training corpus. Meanwhile, shorter contexts (especially the empty context) occur more often, but lose information. *Smoothing methods* [12] in language modeling provide a well-founded way to combine information from various context-lengths.

A smoothed N-gram model starts at the longest order N for which the context has been observed in the training corpus. Then, the model assigning both a **probability** p_N to the observed *event* given the context, and a **confidence** λ_N to the *context* per se. The latter quantifies the amount of information that is present in the context: a total of λ_N probability mass is divided among events seen in this context, while a probability mass $1-\lambda_N$ is recursively passed to a shorter context. Generally, the recursion halts after the empty context (which represents the unconditioned token-frequency distribution in the corpus), where the left-over probability mass is divided equally across the complete vocabulary.

The above method merges information present at all context lengths up to any N. The choice of λ_N has been well-studied the NLP field, and many smoothing methods have arisen, each with its own way to compute λ_N. In this work, we consider four methods:

- Jelinek-Mercer (JM) smoothing uses a fixed confidence in all contexts (we use 0.5) and requires the least parameters.
- Witten-Bell (WB) assigns confidence to a context based on the average frequency of events seen in a context (higher is better).
- Absolute Discounting (AD) subtracts a fixed discount from the count of each event and re-distributes this to unseen events, thus penalizing rarely seen events most.
- Kneser-Ney (KN) improves upon AD by considering how likely a token is to appear in a new context (*e.g.*, "Fransisco" virtually always appears in the same context: "San") and represents the state-of-the-art in NLP N-gram models.

Both AD and KN can slightly be improved by using three separate discounts, for events seen once, twice and more than twice; we use these versions and refer to them as MKN and ADM (M for Modified). We refer the reader to [12] for more details on these techniques. As we shall see, the interpretation of λ_N as confidence in a context according to a model will turn out to be a powerful tool in mixing N-gram models that are trained on different corpora.

Explicit Models & Code The issues of unlimited vocabulary, nested locality, and dynamism introduced in Section 1 are not fully addressed by explicit models. N-gram models typically close the vocabulary at test time, which would tend to diminish performance for locally-used terms. They do not consider scope at all, nested or otherwise; all text is treated uniformly. Finally, traditional models assume static estimation & use, since NL text is generally unchanging. *Dependency* based models also limit vocabulary, for reasons similar to N-gram models; the range of dependencies tend to be small, and thus scope outside of the immediate sentence is rarely considered. So far, to date, the dependencies considered in models of code are intra-procedural, and do not capture patterns in a nested, scoped fashion. They are not very good at handling dynamism, since changes would typically require re-doing the dependency analysis. Although incremental static analysis has been explored for code, to our knowledge no one has used it for dependency-based statistical models thereof. Finally, *Cache* models do deal with vocabulary, by accounting for *all* tokens within a single limited (un-nested, typically at a single file level) scope. They do not deal with multiple, nested scopes. There is limited dynamism: a single file's N grams are counted, and stored in the cache, and this cache is flushed when a new file is opened. This approach, however, ignores the nested scoped context of, *e.g.*, the package structure in Java, and cannot quickly handle browsing: if a developer closes a file `a.java` within a package `sales.foo.com` and opens another file `b.java`, we would like to a) flush the counts of sequences in `a.java` to the model of `sales.foo.com`, b) weight that model more than `foo.com`, since it is more appropriate to `b.java`, and c) start tracking the counts in `b.java` —and do all this efficiently, at interactive speeds. No existing model does this; but we do.

2.3 Implicit Language Models

The above models all rely on explicit counts of actual N-gram frequencies. Neural network models, by contrast, use an optimized high-dimensional real-valued parameter space to *implicitly* represent the co-occurrence patterns of tokens in a large corpus. This parameter space is estimated using gradient-descent techniques which propagate loss gradients in entropy-like functions over a training corpus.

We compare our approach with two popular techniques: Recursive Neural Networks and Long Short-Term Memory networks. Both have been used to model software [13, 38]. These models have proven to be quite powerful, but are more computationally expensive to estimate; this limits their dynamism as discussed later. In most cases, it is best to use them with high-performance GPUs.

Recurrent neural networks (RNN): maintain a hidden state vector to capture a digested representation of the current context, as they scan forward, token by token. Learned parameters both read out this vector (*e.g.*, to predict, score a token) and update this vector upon seeing the next token. These models are quite effective when trained with sufficient data (see [25, 38] for more details in a natural language and source code setting respectively).

Long short-term Memory networks (LSTM): are extensions of RNNs which can be trained to selectively "forget" information from the hidden state, thus allowing room to take in more important information [13, 18][2].

Implicit Models & Code Like earlier explicit models, deep-learning based models also were not designed for the specific vagaries of code. First, they strongly limit the *vocabulary*: larger vocabularies

[2] For an accessible overview, see also http://colah.github.io/posts/2015-08-Understanding-LSTMs/

substantially increase the number of parameters needed for a model, partly because they require larger hidden state sizes (which quadratically increases training time) and increase the cost of predictions[3]. Character-LSTM models [21, 22] deal with this issue by modeling one character at the time instead and may provide a solution to this problem in the long term, possibly in a hybrid form, since these models at present cannot compete with word-level models [19].

Dealing with nested, scoped vocabulary and dynamism with implicit models is difficult. Counts are not explicitly stored, but are transformed in opaque, non-linear ways into real-valued vectors. It's not clear how to quickly update these to deal with code changes or interactive developer browsing. Dynamically adding observations to these models can be done with some success [38] but there is no option for removing observations, capturing localities or changing contexts. On the other hand, unlike explicit models, these models effectively, seamlessly and smoothly combine modeling of local and non-local patterns. LSTM models are specially capable of capturing relevant long-distance contexts, which N-grams fundamentally cannot do. Thus there is a clear opportunity to combine explicit models together with implicit models.

3 OUR APPROACH

Consider the web-app developer in Figure 1, who is working in a file named “Bill.java" (in package `billing`). Her development context begins with “Bill.java", and then the proximate packages (`shipping`, `customer`) and finally a host of other projects that also include repeating code patterns and variable names (*e.g.*, from Github); all contexts could be captured in models. In Figure 1, the developer queries the model with “`customer . ?`", which might be an API recommendation task. For this, we could start with the longest context (“`customer .`") and ask each model for possible completions, with corresponding (MLE) probabilities. If the global corpus hasn't seen this context, it hasn't any say. Suppose the two local models (`customer+ship` and the cache) have, but that the latter has only seen “`bill`" in this context, and gives it a high probability, while the neighboring packages note that “`ship`" is a possibility as well, without discrediting “`bill`" as an option. These probabilities are mixed; then we consider a shorter context (“ `.` ") and repeat; here, the global model may have a contribution, such as the general prevalence of “get" among API calls.

Algorithm 1 describes how we assign probabilities to possible completions, while considering all relevant localities, assuming simple Jelinek-Mercer interpolation (J-M). Although this example has only three levels of locality, our approach generally improves with increasing project hierarchy depth; deep hierarchies are common in our corpus. When more than two models have seen a context, the assigned probabilities get averaged serially from global to local, strongly favoring the more local model's predictions.

The global model is entirely static, and doesn't have to be an N-gram model; it could be any combination of models (*e.g.*, LSTM or RNN). However, the more local models should be highly responsive; *e.g.*, if the developer suddenly switches to “Shipment.java": we must quickly update the local model hierarchy. These performance challenges are discussed next. Finally, The confidence scores can be derived from various sources, such as the language models themselves or even as parameters learned by an LSTM. Their values are less important than the ordering of models (from global to local), so simple interpolation appears a good enough choice. In fact, in the newer implementations[4] each model just computes one probability and confidence score for the whole sequence, after which we mix everything (rather than mixing at every context length), with no significant loss in accuracy.

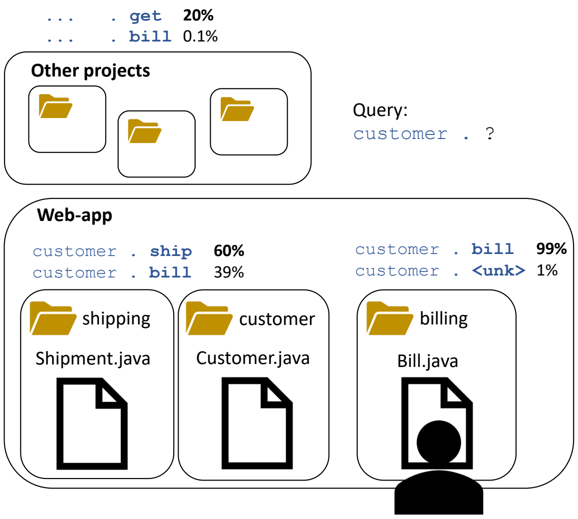

Figure 1: Illustrative example, in which a developer interacts with various localities while working on a file.

3.1 Dynamic, Scoped, Counting

Language modeling packages, originally designed for natural languages, statically estimate N-gram probabilities *a priori* and store these in model files that are read when needed. A probability and interpolation parameter is stored with each sequence, making lookup relatively fast for known sequences. However, updating these models on-line is complex and time-consuming. Suppose, when a new context is entered (or a code-change is made) the observed frequency of “`j`" following “`for ( int`" increases (*e.g.*, because another programmer prefers `j` over `i` as a loop counter in one particular file). Now, changing the probability of one sequence (or adding a novel event) must affect many other sequences with the same context. Yet code bases are perpetually evolving and developers frequently switch files to work on; this can make fine-grained (especially the above hierarchical) models costly to compute, specially in interactive settings.

Ideally, we would be able to rapidly alter models to fit our needs and construct smaller models on the fly, yet not rule out static models (*e.g.*, N-gram or RNN models pre-estimated on large external corpora) when available. To do so, we deviate from conventional approaches that store probabilities and instead store counts. These allow dynamic updating, by incrementing & decrementing language event counts. We use a *trie*-like data-structure, in which each token is a node in a tree with any number of successors (leafs) allowed. At each node, we store a number of values, such as how frequently

[3] This applies particularly when using normal SoftMax output layers, but also when using NCE or CNN SoftMax[19]

[4] github.com/SLP-team/SLP-Core

Algorithm 1 Probability calculation in nested model

```
Require: global ← counts on global corpus (may be empty)
Require: root ← root of project
  ▷ Returns counters (data-structures holding the frequency of all
  sequences) nested around a file, from global to local
  function GET_COUNTERS(file)
      path ← directories from root (inclusive) to file
      counters = [count(dir) for dir in path]
      for $i \in [0, |counters|)$ do
          counters[$i$] -= counters[$i + 1$] ▷ remove more local counts
      end for
  return global ∪ counters ∪ cache(file) ▷ add global & file-cache
  end function

  ▷ Returns the (J-M) probability of a sequence of tokens in a file
  function GET_JM_PROBABILITY(file, tokens)
      counters ← get_counters(file)        ▷ retrieve nested counters
      probability ← 0
      for $i$ from |tokens| - 1 to 0 do
          seq = tokens[$i$ : |tokens|]  ▷ start with shortest sequence
          context = tokens[$i$ : |tokens| − 1]          ▷ may be empty
          p ← 0
          for all c ∈ counters do
              if context ∉ c then continue
              end if
              p = (p + c.freq(seq)/c.freq(context))/2 †
          end for
          probability = (probability + p)/2 †
      end for
      probability += 1/$|V|$                  ▷ add vocabulary base-rate
  return probability
  end function
  †Simple J-M smoothing amounts to averaging with other coun-
  ters and context lengths. Technically no averaging is used for
  the first model that returns an observation; omitted for brevity
```

it occurs as a context to other tokens, as well as its successors in a sorted array. All tokens are pre-translated using a vocabulary that is either dynamically constructed or pre-computed. As an optimization, any sequence of tokens is initially stored as a simple array with an occurrence; only if multiple successors are observed for the sequence is it promoted to a full-fledged node. This greatly reduces the memory footprint of our models (which generally require more memory than conventional probability-based models) since many long sequences of tokens are singular appearances in source code (similar to how many tokens are rarely seen, *i.e.*, Zipf's law).

With nested scopes, each scope has it's own set of counts, stored in a trie. This allows quick switching of scopes, or creation of new scopes. In most cases, we have between one and 5 nested tries. It's quite efficient: Typically, to close one file, and go to another, it takes only a few milliseconds. These nested tries allow probabilities and confidences to be computed by cumulating the various counts stored. Although this makes modeling slightly slower than using pre-computed probabilities in the static case, our model is still able to answer queries in microseconds and delivers all the benefits of dynamicity. Furthermore, we memoize the top-2n successors at the most frequently seen nodes for prediction tasks: these are re-scored to get n ranked suggestions in micro-seconds at prediction time and the memoization is flushed if a change has been detected to the trie node (*e.g.*, a context-switch). Finally, this choice of model achieves two additional benefits:

- Count-based models allow any choice of models after-the-fact, which means we can switch out smoothing methods and many other models that can work on counts of data (*e.g.*, skip-gram models, perhaps even neural networks)
- This model can represent any change granularity. Recent bug-detection work has built models on snapshots at one month apart and tested on the ensuing changes until the next snapshot [31] in order to make the computation tractable; the models in this work can be updated with every commit from a project's inception and run in the order of minutes across thousands of commits, which may also be beneficial for modeling code changes [32] in code reviews [16].

4 EVALUATION METHODOLOGY

We use Allamanis *et al.*'s Giga-token corpus [4] which collects over 14 thousand popular Java projects from Github[5] (sans forks). We also use the original partition of this data into train (75% of projects) and test (25% of projects) sets.

For base-lining, we focus on a 1% subset of the corpus, since some prior models (specially deep learning) don't scale well. We took 1% of train and test projects from the original split, and further took 1/3% of the projects (from the train-split, non-overlapping) to a validation corpus for the neural networks. The corpus statistics are shown in table Table 1.

Table 1: Corpus statistics as reproduced from [4], full corpus and 1% train/test/validation splits.

	Full corpus	Train	Test	Valid
Projects	14,317	107	38	36
Files	2,230,075	13,751	8,269	7,227
Tokens	1,602M	15.98M	5.3M	3.8M

To our knowledge, our 16M token training corpus is the largest yet used to train *deep learning models* of source code. To make training computationally feasible, we fix the vocabulary for the experiments involving deep learning models (but not in our other experiments) by removing tokens seen less than 5 times in the training data and replacing these (and any novel tokens in the validation and test data) with a generic unknown token. This limits the vocabulary to 74,046 tokens, comparable to prior work using deep learning [13, 38]. This vocabulary limit is NOT needed for experiments not involving deep learning models; our nested dynamic counting models can easily handle much larger vocabularies. Later, on evaluation, we relax the vocabulary limits to show that this limit falsely inflates modeling performance of all models.

[5] Retrieved corpus statistics deviate slightly from those reported in the original work.

4.1 Metrics

We evaluate both intrinsic (using entropy, Section 2.1) and extrinsic (using code suggestion) performance of each of our models. In the suggestion task, the model provides an ordered list of suggestions for each token in the file given the context; we collect the rank of the true next token in the file from this list. We also collect the top 10 predictions from each model, and compute the top-k accuracy (the fraction of times the correct suggestions appears in the top k suggestions) for $k \in 1, 5$. We mainly report Mean Reciprocal Ranking (MRR), a summary metric which calculates the average, over all predication points, of the reciprocal of the correct suggestion's rank (or 0 if there are no correct suggestions). This metric balances prediction accuracy across various k, and can intuitively be interpreted as the inverse of the average expected position in the rank list. For instance, a MRR value of 0.5 suggests the correct suggestion can be expected at position 2 in the prediction list on average. Evaluation on other extrinsic tasks, like bug prediction [31] or variable renamings [2] is left for future work.

Partially due to the large number of samples, even minor improvements (*e.g.*, 0.01 bits) in entropy can be statistically significant in language modeling. For all of our comparisons we used a paired, two-tailed t-test; we report when this was not the case. The same applies to prediction accuracy scores, which are strongly inversely correlated with entropy. Even if small, improvements in terms of entropy can help many applications, and shed more light on the repetitive nature of source code. Improving prediction accuracy is practically important, and in our case also comes with faster implementations. Thus we expect that our mixed-scope dynamic models would be the preferred choice when token-level language models are needed. We report effect sizes using Cohen's D.

4.2 Model Configurations

Tu *et al.*'s cache model is compared with our cache implementation, as our approach to mixing differs both in terms of choice of interpolation parameters and in terms of choice of the models to be mixed. For consistency with prior work, we use Modified Kneser-Ney 3-gram smoothing for the global model and a 3-gram back-off cache model, mixed with a dynamically computed concentration parameters as in [36]. We further vary the N-gram order of both components from 1 through 6.

White *et al.* found that Recurrent Neural Network works quite well for modeling source code [38]. We replicate their experiments using the Recurrent Neural Network Language Model toolkit (RNNLM[6]) [25], which includes a hidden layer size of 300 and 1,000 direct connections. More recent work has demonstrated that LSTM networks achieve superior results for source code [13], similar to natural language. We adapt Tensorflow's [1] LSTM code for our purposes, testing two configurations corresponding to the "Small"[7] and "Medium" configurations reported in both Tensorflow's implementation[8] and various other work.

Most notably, the LSTMs *embed* their vocabulary into a state-based high-dimensional Euclidean space, whereas RNNLM simply one-hot encodes its vocabulary. Embedding allows faster and potentially more accurate training, as words are encoded in lower-dimensional, potentially semantically more meaningful vectors. For this reason, we slightly alter the Small configuration to closely match White *et al.*'s RNN configuration, reducing it to a single tier and increasing its hidden layer size to 300 neurons. This allows us to interpret the differences in performance between the RNNLM and the LSTM-300 models as the gain achieved by embedding and using LSTM units. The Medium configuration is left as is, and uses two tiers of 650 hidden-layer neurons as well as drop-out regularization during training. This model was the largest that could be trained on our corpus in reasonable time (approximately three days), requiring 39 passes over the 16M tokens of training data at an average of *ca.*, 2,500 words per second on a Tesla K40c GPU.

Finally, for extrinsic evaluation purposes we compare and combine our models with the two LSTM configurations (the most powerful benchmarks). Here, we restrict ourselves to the first one million tokens in the test data, storing the top ten suggestions for each model in a file and merging the suggestion lists after the fact. We encountered slight irregularities in Tensorflow's output that led it to occasionally skip predicting a token and accommodated our code accordingly; these events accounted for less than 1% of test samples and are unlikely to distort the results.

4.3 Test Settings

We evaluate the models in 3 different settings: *Static*, similar to prior work; *Dynamic*, which favours deep learning, and *maintenance*, which arguably most resembles developers' normal mix of browsing & coding activities.

Static tests: By *static* we mean training on a fixed training corpus, and then evaluating the performance on a separate test dataset. Prior work evaluated models in intra-project and cross-project settings [4, 17, 26, 36, 37]. We begin in a setting similar to Allamanis *et al.*, on our 1% sub-sets, training on one set of projects and testing on a disjoint part of the corpus. Here we demonstrate the performance of our cache model in comparison to Tu *et al.*, and the baseline performance of the recurrent neural networks. We later refer back to this setting when we discuss the impact of vocabulary constraints that were frequently used in prior work.

Dynamic tests: While our static tests are strictly cross-project, in practice, source code files are rarely modified in isolation. Our study subject projects comprise many nested directories. We use a cache to exploit the nested context, at test time, in a rather limited way: if the cache grows too large, performance can actually decrease [36], as the cached information becomes less locally relevant. *Dynamic models* deal with this problem differently: they update with all information available at test time, without any cache-model-type "forgetting". This provides considerable gains for neural network models [38]. This approach does not suffer from the too-large cache problem discussed above, because they do not mix a local and global model – information is added to the global model directly; however, these models don't truly leverage the full potential of scoped locality, as we shall see.

Our second test setting models projects in a dynamic fashion, allowing the models to observe the testing corpus left-to-right, and absorb information to their training data as encountered. The

[6] www.fit.vutbr.cz/~imikolov/rnnlm/

[7] with two changes, see next paragraph

[8] https://goo.gl/Urgrpy

RNN/LSTM models are dynamically updated by training once on each new sample with a learning rate fixed to 0.1 for the RNN and 0.05 for the LSTM models (this yielded slightly better results than 0.1). The neural network models do not at present have the ability to cache.

Software maintenance tests: Finally, we evaluate the model performance in a software maintenance setting, where a developer explores code as she does her work. In this "maintenance" setting, the full test project is available for training minus only the file to be modeled. The neural networks cannot take full advantage of this situation without constructing many models, neither can traditional language modeling tool-kits. Our dynamic models can however, quickly update nested counts on-the-fly. Thus, our nested models recursively accumulate the local repetitions from all nested levels of the project, down to the target file. As we shall see, performance in this final setting is best-in-class, and has ramifications for code synthesis [14, 30], fault localization [8, 31], code naturalization [2] and many real-world code suggestions settings [15, 29].

5 RESULTS

Our first experiments clarify our choice of base N-gram model design, including a replication of the Tu *et al.* [36] cache. Then, we compare the Tu *et al.* cache with our nested cache models in both an intrinsic and extrinsic setting. We next compare our static and dynamic models with RNN and LSTM models, first in a closed-vocabulary setting and then, following an analysis of the impact of such a constraint, in an unbounded vocabulary setting. We furthermore show mixture results between these classes of models, in which we demonstrate the mutual performance and complementarity of LSTM and N-gram models, particularly in terms of entropy. Our LSTM models were trained using Tensorflow[1] on a Tesla K40c GPU with 12GB graphics memory; the RNN was trained using a single core of an Intel Xeon CPU and all other evaluations were run on an Intel i7-4710MQ processor with up to 24GB of RAM.

5.1 Initial Experiments

We implemented the three smoothing methods described in Section 2.2, JM, WB and ADM, also using Tu *et al.*'s code to evaluate MKN. Of these, MKN is best suited for natural language, followed by ADM which we found performed nearly as well as MKN on a one-billion token corpus from [11], as well as on several small corpora. MKN (and sometimes WB) have been used in most prior work. We tried various lengths of N-grams for our models and the JM model at $n = 6$ yielded a durable optimum for both the plain and cache models, as shown in Figure 2, with higher orders yielding little to no gain past this point. Thus, we use JM-6 models in our experiments. Unfortunately, the SRILM toolkit used by Tu *et al.* [36] did not work for orders 7 or higher, due to problems in the discounting. Nonetheless, the trend from 4 through 6-grams suggests that we should expect little to no improvement beyond the 6-gram level.

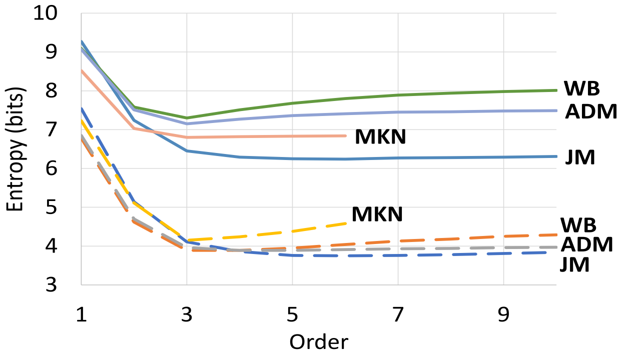

Figure 2: Modeling performance (entropy) obtained with various smoothers for N-grams plotted against order of N. Models without cache (top) and with cache (bottom) largely follow the same pattern, with Jelinek-Mercer smoothing outperforming more refined alternatives.

There are some notable patterns. For one, with natural language, cache-less N-gram models beyond the 3 or 4-gram level achieve minimal gains, and only with highly aggressive smoothing to lower orders; MKN is typically capped at 5-grams where it can achieve minor improvements over 3-grams with large training corpora. For source code MKN behaves likewise, with both unbounded (as used here), or restricted vocabularies (as in White *et al.* [38]). Cache performance degrades notably with higher orders.

This pattern does *not* obtain with less aggressive smoothers, especially JM. Perhaps MKN is too aggressive for source code, where longer contexts seem to contain more useful information. Prior work has used 3-gram Kneser-Ney smoothed models for source code: but this model scores about 0.56 bits of entropy worse than JM-6 in our experiments (0.40 bits for the cache model). Interestingly, JM is very simple smoothing approach; perhaps further improvements could arise from more software-appropriate smoothing methods.

Many previous approaches report reductions in entropy by using some measure of local state (hidden, explicit or both) into their models, reporting improvements over simple N-gram models. Tu *et al.*'s cache model [36] provides an elegant way of incorporating local state, but has unfortunately not usually been explicitly compared. We hope that the availability of our tool, with implementations, can help set the standard for N-gram baselines in work on modeling of source code. Finally, we briefly note that all these results did replicate in the restricted vocabulary setting that we explore in section Section 5.3, which was used in most related work.

5.2 Cache Models

We now expand the idea of caching, extending it to *nested scopes*, mixed as described in Section 3. Nested models only apply when the models have a view of the context of the file to be modeled, either incremental or full. Thus, we demonstrate results in two settings: dynamically updated models, which add each file to their training corpus after modeling it, and software maintenance models, which are allowed to see each file in the project excluding the one to be modeled, as well as the nested, scoped directory hierarchy in which the test file occurs.

In Figure 3 we show the results. The "Flat" models (without and with cache) treat all seen files as a single ("flat") corpus. They show higher entropies of 5.3-5.5 bits and 3.3-3.4 respectively; MRR performance is around 0.6 without cache, and rises to 0.7 with cache. However, hierarchy-aware nested models substantially boost model & prediction performance: Entropy decreases by about 2 bits for the cache-*less* model, and about 1 bit otherwise, with concomitant

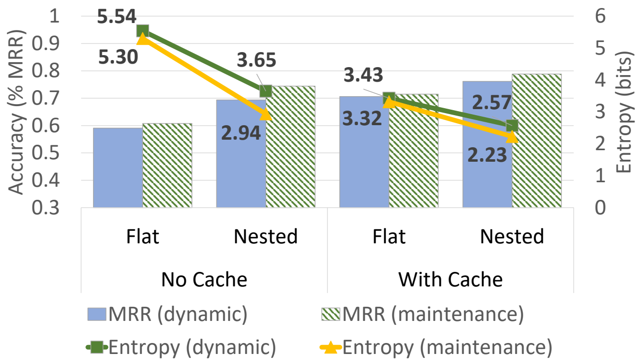

Figure 3: Entropy and MRR prediction performance in a dynamic and software maintenance setting for both non-nested and nested models. Each model is shown with and without a cache component.

increases in MRR. Remarkably, the nested model without a local file cache outperforms the non-nested model with cache in the maintenance setting. This suggests that information in nearby files is almost as useful as information from the same file. Prediction accuracy also increases with nested models, boosting maintenance setting MRR to a best-in-classs 78.8% (and 76.2% in a dynamic setting); again, with very good (timing) performance. Again, we emphasize that improvements are statistically significant ($p \ll$ 1e−10, small effect sizes in general, *e.g.*, 0.18 between cache and nested cache and 0.29 between plain and nested plain).

5.3 Implicit Models

Baseline We compare our baseline JM-6 smoothed N-gram models (with and without cache) with the RNNLM, and LSTM implementations in table Table 2. Please note, in this section only, we restrict the vocabulary to 74,064 words as specified in Section 4. Consequently, the overall entropy scores are *deceptively* lower than in previous sections, even for the N-gram model; we shall revisit this result later. The RNN/LSTM models outperform the Plain N-gram model in the static setting by upto a bit (and, (See section Section 5.1), the KN-3 baseline used in most prior work by well over a bit). Furthermore, the LSTM/300 model outperforms the RNN baseline, despite their similar configuration, which can be attributed to the use of LSTM units and word embeddings in this class of model. This is in line with prior work [13, 38]. Finally, the largest LSTM model, which was stacked and trained with drop-out, achieves best results in this cross-project setting. Nonetheless, the (non-nested) cache model outperforms all of these by a margin by only incorporating a small amount of local information ($p \ll$ 1e−10, Cohen's D: 0.21 compared with LSTM/650).

The comparison between the LSTMs and cache model is complicated, since both model local information at test time; the cache model does so explicitly while the LSTM represents local state implicitly in its hidden state vector. In the second setting ("Dynamic" column), both models are allowed to dynamically update with any content seen after modeling, making a more fair comparison. In this setting, the plain cache model is somewhat worse than the best LSTM (300[9]). The nested cache model, however, decisively outperforms all other models ($p \ll$ 1e−10, Cohen's D: 0.05 compared LSTM/300).

Finally, in the *maintenance* setting our best model (nested cache), scores 1.41 bits ($p \ll$ 0.001, Cohen's D: 0.14 compared with dynamic LSTM/300, the best dynamic LSTM). The deep learning models are not applicable in the maintenance setting; testing on each file given all other files in the nested directories of tested project would be intractable even on this small corpus.

Table 2: Modeling results for our various baselines. The nested models are not applicable in a cross-project setting, and the deep models cannot at present accommodate software maintenance settings in any tractable manner.

Setting / Model	Static	Dynamic	Maintenance
Plain	3.94	2.64	2.32
Cache	2.45	1.89	1.73
Nested plain	–	2.28	1.85
Nested cache	–	1.70	1.41
RNN/300	3.66	1.93	–
LSTM/300	3.22	1.84	–
LSTM/650	3.03	1.91	–

Mixing results We now report *prediction accuracy* for our (nested) N-gram models and LSTMs in isolation and, importantly, when mixed together. First, as in prior work, we credit models for predicting out-of-vocabulary words as unknown, since we use a restricted vocabulary, and then, in the next section, report an alternative. The mixing procedure is a simple interpolation of probability scores, since the LSTMs do not report a confidence score with their predictions (future work may investigate methods to do so). We compare with the LSTMs, as these performed best in our previous experiment, focusing specifically on the first one million tokens in the test set for timing purposes. Each LSTM model required on average 30 hours of run-time on a Tesla K40c to generate top-10 predictions for these test sets; our models are much, much faster.

As expected from the entropy results, (see Table 3) in the static setting the two LSTM models beat the plain N-gram model but are beaten by the cache model. Mixing these models yields small but significant gain for the cache model in both intrinsic and extrinsic terms. Our best performing models, JM-6 cache and LSTM/650 mix best, yielding the best entropy score in a static setting (2.20 bits) and boosting the cache's MRR from 75% to 77% ($p \ll$ 1e−10 either way, Cohen's D: 0.04 *vs.*, cache, 0.24 vs LSTM/650). Mixing LSTMs and plain models all yielded significant improvements in terms of entropy, but not in terms of prediction accuracy, suggesting the plain model is too weak to complement the LSTMs.

In the dynamic setting, mixing the best performing models (dynamic LSTM/300 @ 1.73 bits and nested cache N-gram @ 1.31 bits), gives the *best combined score* of 1.17 bits of entropy ($p \ll$ 1e−10 either way, Cohen's D: 0.06 *vs.*, nested cache, 0.19 vs dynamic LSTM/300), yielding a best-in-class, token-level model for source

[9] The LSTM-300 model proved more amenable to dynamic updating, possibly because the larger model would need several passes over each new observation to update its larger number of parameters.

Table 3: Isolated (first row and column) and mixture (intersection of models) MRR prediction results for various cross-combinations of models from two categories: deep learning networks and *n*-gram networks.

Static setting

n-gram **LSTM**	–	Plain	Cache
–		58.0%	75.7%
LSTM/300	66.1%	65.9%	76.1%
LSTM/650	67.9%	67.3%	**77.3%**

Dynamic setting

n-gram **LSTM**	–	Plain	Cache
–		81.8%	**86.2%**
LSTM/300	82.0%	84.6%	85.7%
LSTM/650	80.0%	83.3%	84.8%

code, suppressing the information content of source code to around 1 bit per token! Notably, unlike [6, 24, 29, 33] our approach is language-agnostic. However, the prediction scores don't improve as expected; the cache model appears to dominate the LSTM so that simple mixing does not help. In the next section, we will see how a more realistic scenario actually shows beneficial mixing.

5.4 Vocabulary Use for Source Code

The excellent model combinations above (with single-bit entropy scores, and unprecedented prediction accuracy) come with a big caveat (also applicable to prior work): this is due to *closed, limited vocabularies.* This arises from a convention in natural language to train using a finite vocabulary on a training corpus and close this vocabulary at test time. Oddly enough, the term "open vocabulary" refers to the practice of closing the vocabulary at test time, but treating new tokens as unknown tokens rather than skipping them entirely. In source code, limiting vocabulary on training data is arguably inappropriate. Developers introduce new vocabulary in new scopes and use them in contained scopes. Code models must capture this.

Consider Figure 4, showing the (misleading) lift in performance (y-axis) with artificially limited vocabularies, which "cuts off" events seen just a given number of times (x-axis) in the training data (solid lines) and are closed at test time (replacing all unseen events in the test data with the same generic unknown token), compared to the best estimate of the "true" performance (flat dashed lines), in which the vocabulary is never closed. For many settings, including code suggestion, only the latter curve matters: predicting tokens as "unknown" at test time is unhelpful.

Several patterns stand out from Figure 4: the plain model jumps substantially in performance when the vocabulary size decreases even just a little. A similar effect can be observed for the cache model from just closing a vocabulary at test time. Crucially, however, the plain model cannot attain the performance of the (dotted) cache baseline, even with a closed vocabulary and a count cut-off that reduces the vocabulary size by two-thirds (*i.e.*, cutoff ≤ 5). We thus conclude two things: 1.) reducing vocabulary size and/or closing it at test time causes substantial, but misleading inflation in modeling and prediction performance, and 2.) a cache component elegantly deal with the vocabulary innovation problem while requiring no artificial limits.

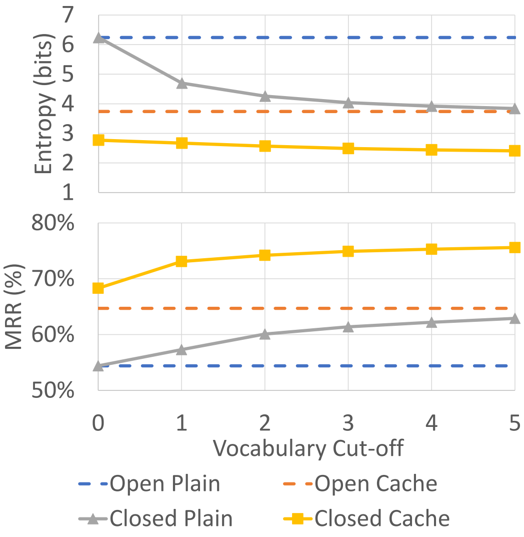

Figure 4: Vocabulary cut-off (minimum number of times an event must be seen in training data to be included in the vocabulary) *vs.*, performance on two mainstream metrics (solid lines), as well as truly open vocabulary at test time (dashed lines).

Ramifications for Deep Learning: the LSTM and RNN models struggle with vocabularies larger than those we used. A recent investigation of natural language which heavily optimized performance of LSTM models required over one Tesla K40c GPU year to achieve optimal performance with a vocabulary of less than 800K; although our 1% subset has a vocabulary of 200K, the full dataset's vocabulary exceeds 12M. Training our biggest LSTM models with just 76K vocabulary took many days (compared to *ca.*, 15 seconds for the explicit models). Furthermore, without using character-level models (which so far don't work as well as word-level models in NLP), opening the vocabulary remains impractical.

Next, we repeat the experiments in Section 5.3, but without giving credit to the LSTM models for predicting an unknown token. For entropy, the LSTM models are assigned the vocabulary base-rate score (corresponding to 17.63 bits of entropy in this setting) for each encountered unknown token, exactly as the plain N-gram model (with our smoothing approach). The N-gram models here are trained with no vocabulary constraints and tested similarly, thus never predicting an unknown token. This setting can be interpreted as one in which the LSTM models are trained as an aide to N-gram models; where the former have the upper-hand in terms of static modeling capacity, the latter contribute dynamic insights that can be rapidly obtained and integrated at test time, as well as greater vocabulary range at training time to account for less common events. The mixture works as before, only defaulting to the N-gram model when the LSTM predicts the unknown token.

The prediction results are shown in Table 4. As can be seen, the LSTM model's numbers are much worse, as are the plain *n*-gram model's. This does *not* apply to the cache-based *n*-gram model (in the static setting) or to the nested model (in the dynamic setting): the nested cache model loses only 4.4% points, still achieving an MRR of 81.8% (and top-1 accuracy of 75.9%)! Interestingly, the mixtures no longer improve each others performance in all but the static, plain model. In terms of entropy performance, however, we still find substantial gain: the best models prove remarkably complementary, decreasing each other's entropy scores from 1.92 (nested cache) and 3.93 (dynamic LSTM/300) to **1.25 bits per token** ($p \ll 1e{-}10$ either way, Cohen's D: 0.21 *vs.*, nested cache, 0.62 vs dynamic LSTM/300). This is not only the best ever-reported entropy score without vocabulary limit, but also astonishingly close to the mixture of these same models *with* vocabulary limits! This due to a high degree of complimentarity: while the LSTM predicts the unknown token (costing it 17.63 bits), the nested cache has an average entropy of about 0.72 bits, well below its own average. Contrariwise, on the other tokens the LSTM outperforms the nested cache by $\sim$0.23 bits (and by over 1 bit on tokens that the nested cache assigns 4 bits or less). These models excel in different contexts and contribute mutual robustness. This complementary potential is reflected in the standard deviations of entropy scores for the LSTM (6.28 bits), nested cache (s.d.: 3.78 bits) and the mixture (2.37 bits) which has the most narrow range of all. Thus, we see substantial complementarity in terms of intrinsic, modeling performance.

Table 4: Isolated (first row and column) and mixture (intersection of models) MRR prediction results for various cross-combinations of models from two categories: deep learning networks and *n*-gram networks.

Static setting

***n*-gram** **LSTM**	–	Plain	Cache
–		51.1%	**69.3%**
LSTM/300	56.0%	64.3%	67.5%
LSTM/650	57.4%	65.4%	68.2%

Dynamic setting

***n*-gram** **LSTM**	–	Plain	Cache
–		77.1%	**81.8%**
LSTM/300	63.8%	78.7%	78.7%
LSTM/650	62.1%	77.0%	77.2%

6 DISCUSSION

Deep Learning vs. Count Models DL models are now used everywhere, including for modeling of source code [3, 37]. However, DL models use a great many parameters, require extensive configuration and are also often heavily tuned on the task at hand.[10]

In addition, they are often compared with rather simple baslines, which casts doubt upon any (often minor) improvements observed. In the NLP community, Omer *et al.* conducted various investigations into *word-embeddings* (semantic representations of words) and found that state-of-the-art neural-network models perform similarly to simple matrix-factorization models provided the latter were enriched with just a few hyper-parameters [23]. Our work paints a similar picture: RNN/LSTM fail to beat a well-calibrated cache model (even *sans* tuning on validation data).

We do find, however, that RNN & LSTM complement our dynamic mixed-scope model! While high training costs and the difficulty of incorporating local information make DL insufficient on its own, it can provide semantic insights beyond the reach of simpler models. We encourage future work to focus on fair comparisons between deep and simpler models and optimize both in conjunction.

LSTM probabilities: We found LSTM model predictions to be often quite polarized: the correct next tokens often scored very high or very low. This might arise from the SoftMax output layer, which can accentuate differences. This partially explains the positive mixture results; the *n*-gram models are more conservative in their estimates, rarely either ruling out events entirely or having high confidence in an observation.

We studied the *ca.*, 5.5% of cases in which the LSTM-300 assigned 100% probability to its prediction and found that 99.93% of these were correct. This may help when applying neural networks to study program properties: in this line of work, soundness is often a strong requirement and neural networks cannot as yet guarantee this. However, the LSTM's inclination to assign high probabilities only in cases of great certainty can prove a very helpful property in generating samples for search-based (SBSE) methods.

Training corpus size: Current LSTM models can't manage the largest corpus [4] (Size exceeding 1B tokens), but our N-gram models can do so handily. We trained a 6-gram model on the full 75% training data of the corpus from Table 1, requiring *ca.*, 3 hours and 12GB of RAM, and tested it on the same test data. The nested cache model on this corpus shows a small but significant gain of 0.06 bits and 0.4% MRR.

7 CONCLUSION

We have made the following contributions.

- We introduce a *dynamically updatable*, *nested scope*, *unlimited vocabulary* count-based N-gram model that significantly outperforms all existing token-level models, including very powerful ones based on deep learning. Our model is far faster than the deep learning models. Our nested cache model achieves an MRR performance of 0.818, with unlimited vocabulary (0.85 with limited vocabulary) which is **best-in-class.**. Our work illustrates that traditional approaches, with some careful engineering, *can beat deep learning models.*
- We show that our count-based approach "plays well" with LSTM models, and yields even better performance in combination, particularly in terms of entropy scores where the best mixture achieving 1.25 bits of entropy per token *without constraining the vocabulary.*
- Our detailed evaluations reveal some new observations:
  1. Jelinek-Mercer smoothing outperforms smoothing approaches used in prior work.
  2. Limiting vocabularies artificially and misleadingly boosts intrinsic performance, without boosting actual performance on the suggestion task.

[10] See also http://blog.dennybritz.com/2017/01/17/engineering-is-the-bottleneck-in-deep-learning-research/ for a discussion of this phenomenon.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293.

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of The 33rd International Conference on Machine Learning*. 2091–2100.

[4] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 207–216.

[5] Miltiadis Allamanis, Daniel Tarlow, Andrew D Gordon, and Yi Wei. 2015. Bimodal Modelling of Source Code and Natural Language.. In *ICML*, Vol. 37. 2123–2132.

[6] Pavol Bielik, Veselin Raychev, and Martin T Vechev. 2016. PHOG: probabilistic model for code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML*. 19–24.

[7] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 213–222.

[8] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 252–261.

[9] Ciprian Chelba, David Engle, Frederick Jelinek, Victor Jimenez, Sanjeev Khudanpur, Lidia Mangu, Harry Printz, Eric Ristad, Ronald Rosenfeld, Andreas Stolcke, and others. 1997. Structure and performance of a dependency language model.. In *EUROSPEECH*.

[10] Ciprian Chelba and Frederick Jelinek. 1998. Exploiting syntactic structure for language modeling. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 1*. Association for Computational Linguistics, 225–231.

[11] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2013. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005* (2013).

[12] Stanley F. Chen and Joshua Goodman. 1996. An Empirical Study of Smoothing Techniques for Language Modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics (ACL '96)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 310–318. DOI:http://dx.doi.org/10.3115/981863.981904

[13] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. *arXiv preprint arXiv:1608.02715* (2016).

[14] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, and others. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 345–356.

[15] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. Cacheca: A cache language model based code suggestion tool. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 705–708.

[16] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. 2015. Will they like this?: evaluating code contributions with language models. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 157–167.

[17] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[19] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410* (2016).

[20] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 173–184.

[21] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. 2015. Visualizing and Understanding Recurrent Networks. *arXiv preprint arXiv:1506.02078* (2015).

[22] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2015. Character-aware neural language models. *arXiv preprint arXiv:1508.06615* (2015).

[23] Omer Levy and Yoav Goldberg. 2014. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*. 2177–2185.

[24] Chris J Maddison and Daniel Tarlow. 2014. Structured Generative Models of Natural Source Code.. In *ICML*. 649–657.

[25] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model.. In *Interspeech*, Vol. 2. 3.

[26] Anh Tuan Nguyen and Tien N Nguyen. 2015. Graph-based statistical language model for code. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 858–868.

[27] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 651–654.

[28] Thanh Nguyen, Peter C Rigby, Anh Tuan Nguyen, Mark Karanfil, and Tien N Nguyen. 2016. T2API: synthesizing API code usage templates from English texts with statistical translation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 1013–1017.

[29] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 532–542.

[30] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: synthesizing what I mean: code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 357–367.

[31] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 428–439.

[32] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. 2014. *The Uniqueness of Changes: Characteristics and Applications*. Technical Report. Microsoft Research Technical Report.

[33] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 419–428.

[34] Romain Robbes and Michele Lanza. 2008. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 317–326.

[35] Juliana Saraiva, Christian Bird, and Thomas Zimmermann. 2015. Products, developers, and milestones: how should I build my N-Gram language model. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 998–1001.

[36] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 269–280.

[37] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.

[38] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, 334–345.