Pythia: AI-assisted Code Completion System

Alexey Svyatkovskiy
Microsoft
Redmond, WA
alsvyatk@microsoft.com

Ying Zhao
Microsoft
Redmond, WA
zhying@microsoft.com

Shengyu Fu
Microsoft
Redmond, WA
shengyfu@microsoft.com

Neel Sundaresan
Microsoft
Redmond, WA
neels@microsoft.com

ABSTRACT

In this paper, we propose a novel end-to-end approach for AI-assisted code completion called Pythia. It generates ranked lists of method and API recommendations which can be used by software developers at edit time. The system is currently deployed as part of Intellicode extension in Visual Studio Code IDE. Pythia exploits state-of-the-art large-scale deep learning models trained on code contexts extracted from abstract syntax trees. It is designed to work at a high throughput predicting the best matching code completions on the order of 100 *ms*.

We describe the architecture of the system, perform comparisons to frequency-based approach and invocation-based Markov Chain language model, and discuss challenges serving Pythia models on lightweight client devices.

The offline evaluation results obtained on 2700 Python open source software GitHub repositories show a top-5 accuracy of 92%, surpassing the baseline models by 20% averaged over classes, for both intra and cross-project settings.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments**; • **Computing methodologies** → *Neural networks.*

KEYWORDS

Code completion, neural networks, naturalness of software

1 INTRODUCTION

In software development through Integrated Development Environments (IDEs) [13], code completion is one of the most widely used features. Intelligent code completion [14] assists developers by reducing typographic and other common errors, effectively improving developer productivity. These features may include pop-ups when typing, calling methods and APIs on typed classes and objects, querying parameters of functions, variable or function name disambiguation, and program structure completion that use code context to reliably predict following code.

Traditional code completion tools in IDEs typically list out all possible attributes or methods that can be invoked when a user types a "." or an "=" trigger character. However, not ranking the suggested results requires users to scroll through a long alphabetically ordered list, which is often even slower than typing the full name of a method directly. Studies show that developers tend to rely heavily on prefix filtering to reduce the number of choices [10].

In this paper, we introduce Pythia – a neural code completion system, trained on code snippets extracted from a large-scale open source code dataset. The fundamental task of the system is to find the most likely method given a code snippet. In other words, given original code snippet C, the vocabulary V, and the set of all possible methods $M \subset V$, we would like to determine:

$$m^* = argmax(P(m|C)), \forall m \in M. \tag{1}$$

In order to find that method, we construct a model capable of scoring responses and then determining the most likely method.

A large number of intelligent code completion systems for both statically and dynamically typed languages have been proposed in the literature [1, 2, 7, 14, 17, 18]. Best Matching Neighbor (BMN) and statistical language models such as n-grams, as well as recurrent neural network (RNN) based approaches leveraging sequential nature of the source code have been particularly effective at creating such systems. The majority of the past approaches did not leverage the long-range sequential nature of the source code or tried to transfer natural language methods without taking advantage of unique opportunities offered by code's abstract syntax trees (AST).

The nature of the problem of code completion makes long short-term memory (LSTM) networks [4] a promising candidate. Pythia consumes partial ASTs corresponding to code snippets containing member access expressions and module function invocations as an input for model training, aiming to capture semantics carried by distant nodes.

The main contributions of the paper are: (i) we introduce and implement several baselines of code completions systems, including frequency-based and Markov Chain models, (cf. section 2), (ii) we propose and deploy a novel end-to-end code completion system based on LSTM trained on source code snippets (cf. sections 4 and 5), (iii) we evaluate our model on a dataset of 15.8 million method calls extracted from real-world source code, showing that our best model achieves 92% accuracy, beating simpler baselines (cf. section 7), (iv) we discuss and document practical challenges of training deep neural networks and hyperparameter tuning on high-performance computing clusters, and model deployment on lightweight client devices at high throughput (cf. section 8).

2 BASELINE CODE COMPLETION SYSTEMS

The most basic code completion system in an IDE would list out all possible attributes or methods in an alphabetically ordered list. Fig. 1 shows example completions that such a system could serve. As seen, this approach is capable of yielding accurate results for classes with a relatively small number of members (e.g. "socket" Python library), otherwise requiring a developer to scroll through a long list (e.g. in the case of "PyTorch").

2.1 Frequency models

Instead of ranking methods alphabetically, popularity ranking can be adopted to increase recommendation relevance. The simplest frequency-based model one could think of would order methods belonging to each class based on the occurrence count in the training corpus.

Looking for a stronger baseline, we built an improved frequency model incorporating context information about `if`-conditions (referred to as "frequency-if" throughout the paper). In what follows, we subdivide all the invocations into two groups based on whether they are inside an `if`-condition and creating two corresponding popularity lists. As seen in Tab. 1, the most frequent method invocations outside `if`-statements are related to variable or placeholder declarations, setting up a neural network or training routines; inside `if`-conditions, however, a typical call would be to file input-output manipulation wrappers, context managers (e.g. eager execution mode), or specifying tensor properties (sparse or dense). As a result, the "frequency-if" model would provide more accurate code suggestions inside `if`-statements increasing overall accuracy.

2.2 Invocation-based Markov Chain model

Markov Chain models have demonstrated great strength at modeling stochastic transitions, from uncovering sequential patterns [12, 21] to modeling decision processes [3]. When analyzing invocation occurrences in the source code, we discovered that some sequences of invocations are always followed by the same methods.

For example, looking at all the function invocations of "os" and "shutil" Python modules, related to file manipulation, which come after an invocation chain `os.path.isfile` → `os.remove`, we have observed that more than 20% of the time it will be "os.rename" or "shutil.move", even though those two modules have over 30 methods related to file manipulation.

Considering following three-state Markov process:

$$os.path.isfile \rightarrow os.remove \rightarrow ?,$$

where the third state can be any function from "os" or "shutil" modules, we compute the corresponding transition probabilities in Tab. 2. Using previous invocations occurrences to predict what invocation will be used next is at the core of our Markov Chain model of code completion:

$$P(m_i|h_i) = P(m_i|m_{i-n}, ..., m_{i-1}), \qquad (2)$$

here, sequence $h_i = m_1, m_2, ..., m_{i-1}$ denotes a history of previous method invocations in the same document scope for a given class. In an n-th order Markov chain model the probability of next invocation will only depend on the previous $n - 1$ invocations, which is estimated by counting the number of invocation sequence occurrences in the training corpus.

The abundance of high-quality source code data available on GitHub and the tools allowing to extract and label code snippets are making it possible to train and deploy advanced machine learning models achieving more accurate predictions (cf. section 5).

3 DATASET

We collected a dataset to train and evaluate code completion models from open source Python repositories on GitHub. A total of 2700 top-starred (non-fork) Python projects have been selected, containing libraries from a diverse set of domains including scientific computing, machine learning, dataflow programming, and web development, with over 15.8 million method calls. Fig. 2 shows Python libraries with the most method call occurrences in the dataset.

We split the dataset into development and test sets in the proportion 70-30 on the repository level. The development set is then split at random into training and validation sets in the proportion 80-20. To serve predictions online, the model is retrained using the entire dataset.

4 REPRESENTING CODE SNIPPETS

Any programming language has an explicit context-free grammar (CFG), which can be used to parse source code into an AST. An AST is a rooted n-ary tree, where each non-leaf node corresponds to a non-terminal in the CFG specifying structural information. Each leaf node corresponds to a syntax token encoding program text.

The Pythia model consumes partial file-level ASTs corresponding to code snippets containing member access expressions and module function invocations as an input for training. Before feeding to LSTM, ASTs are serialized to sequences according to in-order depth-first traversal. We retain up to T lookback tokens preceding each method call when extracting training sequences, where T is a tunable parameter of the model provided in Tab. 3.

Further, in order for an AST to be consumable by LSTM, its nodes and tokens have to be mapped to numeric vectors. Like the weights of the LSTM, the look-up tables mapping discrete values of tokens to real vectors can be learned via backpropagation. The Word2Vec approach [11] has been shown to work well in Natural Language Processing, and is employed in the Pythia system to represent code snippets as dense low-dimensional vectors.

Each input data file is parsed using the PTVS parser [1] to extract its AST. The method invocations and associated metadata, including invocation spans, serialized sequences of syntax nodes and tokens

---
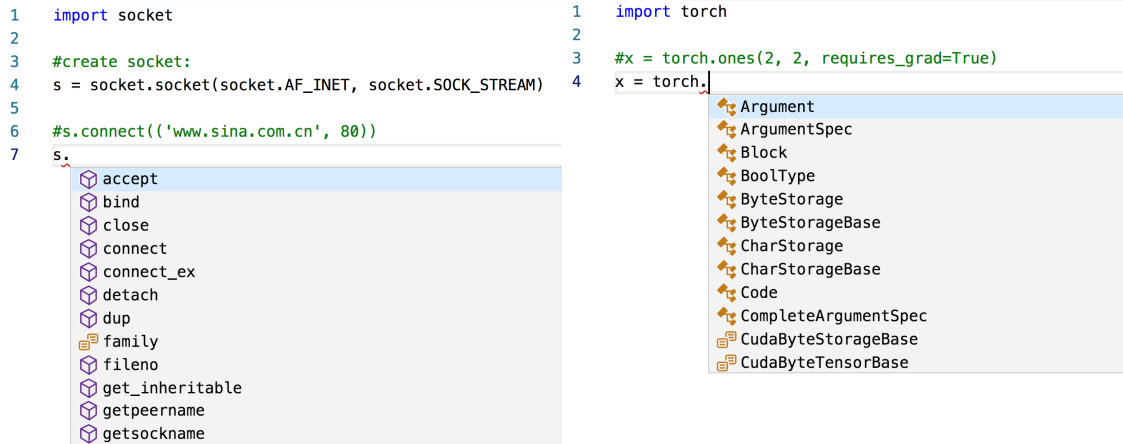
[1]https://github.com/Microsoft/PTVS

```
1   import socket
2
3   #create socket:
4   s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5
6   #s.connect(('www.sina.com.cn', 80))
7   s.
        accept
        bind
        close
        connect
        connect_ex
        detach
        dup
        family
        fileno
        get_inheritable
        getpeername
        getsockname
```

```
1   import torch
2
3   #x = torch.ones(2, 2, requires_grad=True)
4   x = torch.
                Argument
                ArgumentSpec
                Block
                BoolType
                ByteStorage
                ByteStorageBase
                CharStorage
                CharStorageBase
                Code
                CompleteArgumentSpec
                CudaByteStorageBase
                CudaByteTensorBase
```

Figure 1: Code completion system providing suggestions in alphabetical order. Left: "socket" Python library, right: "PyTorch".

Invocation	Frequency	Invocation	Frequency
tf.float32	10508	tf.context.in_graph_mode	905
tf.nn	8954	tf.gfile.Exists	640
tf.constant	7754	tf.gfile	624
tf.train	6869	tf.ops.Tensor	448
tf.variable_scope	6330	tf.context.executing_eagerly	322
tf.placeholder	5622	tf.sparse_tensor.SparseTensor	271

Table 1: Frequencies of the most popular method calls from the TensorFlow library. Left: outside `if`-statements, right: inside `if`-statements.

Method	Transition probability
os.rename, shutil.move	0.22
os.write	0.15
shutil.copy	0.13
os.path.isfile	0.11
os.remove	0.10
All others methods	0.29

Table 2: Transition probabilities to the third state of the three-state Markov chain: `os.path.isfile` → `os.remove` → ?.

from the AST, and the runtime types of method call receiver tokens are extracted. The data processing workflow is summarized in Fig. 3.

Syntax node and token names are mapped to integers from 1 to V to obtain token indices. Infrequent tokens are removed to reduce vocabulary size. During training, the out-of-vocabulary (OoV) tokens in a code snippet are mapped to integers greater than vocabulary size V, in a way that the occurrences of the same OoV token in the training code snippet would be represented by the same integer. Each training sample is an AST serialized into sequence terminated with the "." end-of-sequence character. A name of the method call token serves as a label and is one-hot encoded. We train a word embedding on the corpus of the source code repositories to map token indices to low-dimensional dense vectors preserving semantic relationships in the code.

4.1 Leveraging type information

Python is a dynamically typed language. While PEP 484[2] introduces type annotations for various common types starting Python 3.6, most of the libraries have not yet adopted it. As such, Python interpreter does variable type checking only as code runs. We infer types of the method call receiver tokens and local variables at runtime based on static analysis of the object usage patterns and include this information into training sequences.

In Python programming language, a user has freedom of aliasing imported modules. For instance, a commonly used "import numpy as np" may well be "import numpy as an_unusual_alias". Embedding vectors for "np" and "numpy" tokens are nearby in embedding space. However, an infrequent alias like "an_unusual_alias" may be an
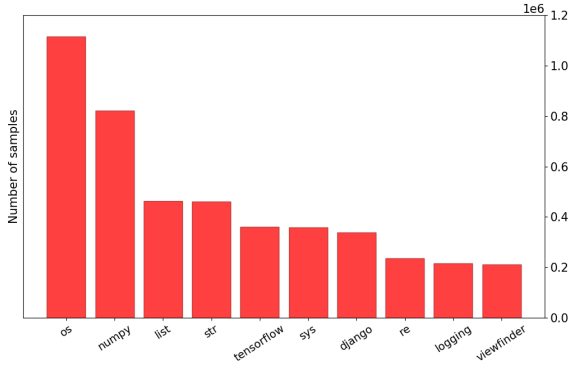
---

[2]https://www.python.org/dev/peps/pep-0484

Figure 2: Number of method calls per library, for top 10 libraries in the training dataset.

out-of-vocabulary token and not represented in the embedding. Including type information during training means an embedding vector corresponding to "numpy" token would be used in all of the cases described above.

Variable names are nouns chosen by a developer, which may differ from one code implementation to another. Keeping them in the original format increases the vocabulary size, making a code completion system dependent on spelling of a variable name. To avoid this, variable names are normalized according to $var :< variabletype >$ convention.

5 NEURAL CODE COMPLETION MODEL

Recurrent Neural Networks are a family of neural networks capable of processing sequential data of arbitrary length. Long short-term memory networks [4], a particularly successful form of RNNs, trained on large datasets can obtain remarkable performance results across a wide variety of domains – from image captioning [9], to sentiment analysis and machine translation [19, 20]. An LSTM unit uses no activation function within its recurrent components, controlling information flow using gates implemented using the logistic function. Thus, the stored values are not iteratively expanded or squeezed over time, and the gradient does not tend to explode or vanish when trained.

The task of method completion is to predict a token m^*, conditional on a sequence of syntax tokens $c_t, t = 0...T$, corresponding to the terminal nodes of the AST of a code snippet C, plus the special end-of-sequence token ".". This problem is a natural fit for sequence learning with LSTM:

$$x_t = Lc_t, \tag{3}$$
$$h_t = f(x_t, h_{t-1}), \tag{4}$$
$$P(m|C) = y_t = softmax(Wh_t + b), \tag{5}$$
$$m^* = argmax(P(m|C)). \tag{6}$$

Here, the matrix $L \in R^{d_x \times |V|}$ is the word embedding matrix, d_x is the word embedding dimension and $|V|$ is the size of the vocabulary. In case of Pythia, function $f(.,.)$ represents a stacked LSTM taking the current input and the previous hidden state and producing the hidden state at the next temporal step. $W \in R^{|V| \times d_h}$ and $b \in R^{|V|}$

are the output projection matrix and the bias. The d_h is the size of the hidden state of LSTM.

Inspired by [8], we are reusing the input word embedding matrix as the output classification matrix as shown in Fig. 4, which allows to remove the large fully connected layer and significantly reduce the number of trainable parameters and the model size on disk. More specifically, we introduce a projection matrix $A = (a)_{ij} \in R^{d_h \times d_x}$ initialized according to a random uniform distribution. Given an LSTM encoded snippet and a hidden state at the last temporal step $h_T \in R^{d_h}$, by multiplying the two together we obtain the predicted embedding vector $L^{pred} = (l^{pred})_j \in R^{d_x}$ as:

$$l_j^{pred} = \sum_i h_{Ti} a_{ij}. \tag{7}$$

Subsequently, the unnormalized predictions of the neural network are obtained as:

$$y_k = \sum_j l_{kj} l_j^{pred} + b_k \tag{8}$$

where $b_k, k = 0...|V| - 1$ is the bias vector initialized to zeros before backpropagation.

6 MODEL TRAINING

Neural networks are trained iteratively, making multiple passes over an entire dataset before converging to a minimum. Backpropagation through time (BPTT) is a gradient-based neural network training algorithm we apply to train the LSTMs statefully.

Training deep neural networks is a computationally intensive problem that requires the engagement of high-performance computing clusters. Pythia uses a data-parallel distributed training algorithm with Adam optimizer, keeping a copy of an entire neural model on each worker, processing different mini-batches of the training dataset in parallel lockstep.

The offline training module of the Pythia system is implemented as a Python library integrating TensorFlow and CUDA-aware MPI for distributed training. The software stack makes use of CUDA 9, GPU accelerated deep learning primitives from CuDNN 7, and TensorFlow 1.10. We use BatchAI [3] – an Azure cloud service providing on-demand GPU clusters with Kubernetes resource manager – for model training and hyperparameter optimization. The online module is implemented in C#, making use of ML.NET library and ONNX data format [4].

6.1 Batching and Learning rate schedule

The number of nodes in a Python file-level abstract syntax tree ranges from $O(10^2)$ to $O(10^4)$. To leverage long-range dependencies in the source code we consider sequence lengths in the range of $100 - 1000$. To overcome the gradient vanishing problem for long sequences, we approximate the computation of the gradient of the loss with respect to the model parameters by truncated backpropagation through time [6].

Efficient batching is important to fully utilize parallelism of modern GPUs. In general, training sequences will have variable lengths, with a majority having the maximum length of T, as shown in Fig. 5. Training samples are presorted and split into three buckets based

---

[3]https://docs.microsoft.com/en-us/azure/batch-ai/overview
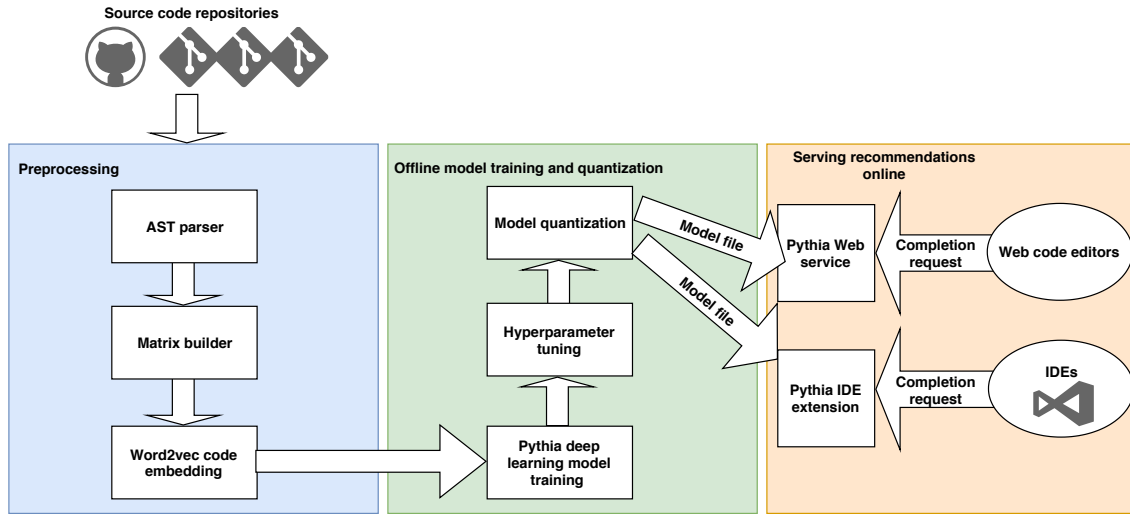[4]https://github.com/onnx/onnx

Figure 3: Pythia workflow: from raw source code files to serving recommendations online.
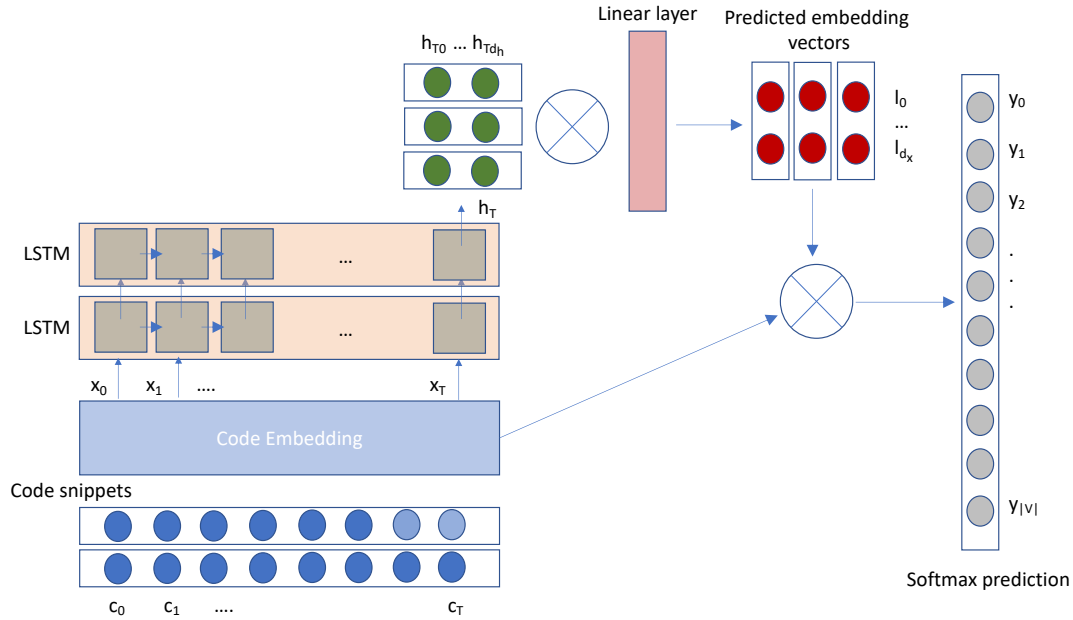

Figure 4: Architecture of the neural network deployed in the Pythia code completion system.

on their lengths. Within each bucket, sequences are padded to the maximum length using the special padding token, which is then excluded from loss calculation by means of masking. A training buffer to maintain sequences belonging to d_b distinct ASTs is allocated. At every training step, the first T_{RNN} timesteps of the buffer are fed as the next batch. The buffer is then shifted by T_{RNN}. Every time a sequence is finished in the buffer, a new set of sequences is loaded, and the LSTM internal states are reset for training.

The learning rate controlling the magnitude of the weight update during gradient optimization is lowered upon completion of

each epoch according to the exponential decay. In a distributed regime, the learning rate is scaled up during the first few epochs to facilitate reliable model convergence. Closely following [5], we linearly scale the learning rate up proportionally to the number of workers N_{worker} during the first 4 epochs ("warm-up" period):

$$\lambda_0(N_{worker}) = \lambda_0 \cdot \gamma^i \cdot \frac{N_{worker}}{\alpha} \tag{9}$$

here, λ_0 is the base learning rate, γ is the learning rate decay constant, i is the epoch number, and parameter α is the scaling fraction
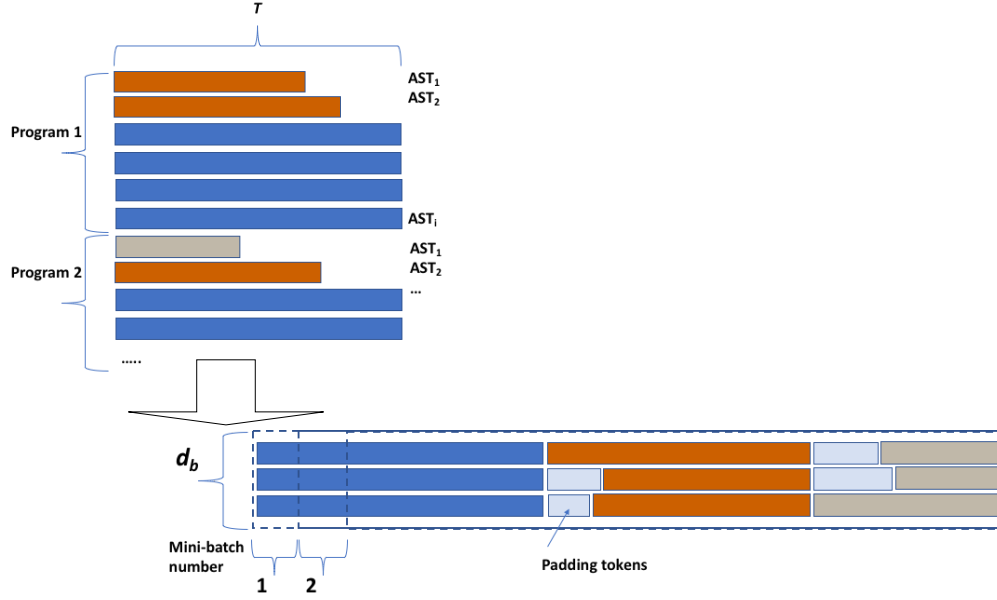
Figure 5: Snapshot of the training buffer.

controlling the learning rate at the end of the warm-up period, $\alpha = 4$ was found to work best via hyperparameter search.

Fig. 6 shows the reduction in training time due to distributed training and the effect of learning rate schedule.

6.2 Hyperparameters

Overall, the model architecture, training procedure, and data normalization produce a large number of hyperparameters that must be tuned to maximize predictive performance. These hyperparameters include numerical values such as the learning rate and number of LSTM layers, dimension of embedding space, but also abstract categorical variables such as the precise model architecture or the normalization algorithm. These parameters are summarized in Tab. 3.

Throughout this work, the "best" model is determined by hyperparameter tuning. This is done via random search in the respective hyperparameter space of each method, i.e. by training a number of models with random hyperparameters on the training set and choosing the one with the highest performance on validation set.

6.3 Tuning neural network architecture

Architecture of the neural network is a hyperparameter. Besides LSTM, we consider gated recurrent units (GRU), LSTM with attention mechanism for temporal data [16], and variations of classification layer following the LSTM.

Selecting the best model for serving online is a trade-off between accuracy and model size. Tab. 4 shows validation level accuracy for various neural model architectures and associated sizes on disk. As seen, removing a large fully connected classification layer in favor of predicted embedding reduces the number of trainable parameters and the model size on disk by 25%. The best top-5 accuracy is achieved when the attention mechanism is employed, however,

this model is 8% larger and has slower inference speeds. Consequently, we have chosen the LSTM with predicted embedding for deployment.

7 EVALUATION

7.1 Evaluation Metrics

Top-k accuracy and the mean reciprocal rank (MRR) [15] are used to measure the quality of recommendations, defined as:

$$Acc(k) \quad = \quad \frac{N_{top-k}}{Q}, \tag{10}$$

$$MRR \quad = \quad \frac{1}{Q} \cdot \sum_{i=1}^{Q} \frac{1}{rank_i}, \tag{11}$$

where N_{top-k} denotes the number of relevant recommendation in top k suggestions, Q represents the total number of test data samples and $rank_i$ is the prediction rank of a recommendation.

Accuracy in top-1 tells how often the top recommendation is correct, while top-5 accuracy provides an idea of how often the top five recommendation list contains the suggestion a user is looking for. The MRR captures the overall rank of the result, thus providing information of how far outside of the list of top suggestions the model prediction was. MRR values closer to one indicate overall smaller recommendation ranks, corresponding to a better performing model.

7.2 Evaluation results

Tab. 5 shows the performance comparison of the Pythia neural model and various simpler baselines, including basic alphabetic ordering, frequency models, and invocation-based Markov Chain model. As seen, our model significantly outperforms all the base-
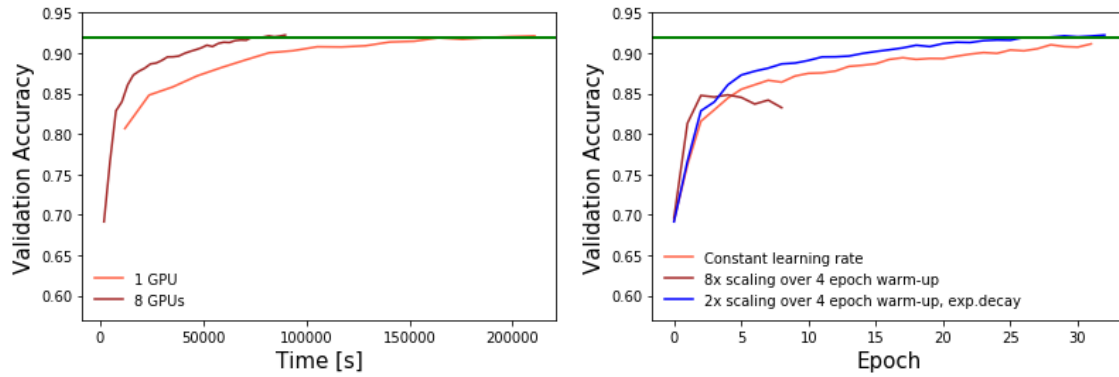
Figure 6: Accuracy on the validation set for various training regimes. Left: serial and distributed training with 8 worker GPUs; right: distributed training with 8 GPUs for various learning rate schedules. The horizontal green line indicates the target model accuracy of 92%.

Hyperparameter	Explanation	Best value
λ_0	Base learning rate	0.002
γ	Learning rate decay per epoch	0.97
N	Number of recurrent neural network layers	2
d_h	Number of hidden units in LSTM, per layer	100
T	Number of lookback tokens, timesteps through which backpropagation is run	1000
T_{RNN}	Number of timesteps through which backpropagation is run	100
RNN type	Type of RNN	LSTM
d_b	Batch size	256
Loss function	Type of loss function	Categorical cross-entropy
d_x	Embedded vector dimension	150
Optimizer	Stochastic optimization scheme	Adam
Dropout	Dropout keep probability	0.8
L2 Regularization	Weight regularization of all layers	10
Clip norm	Maximum norm of gradients	10
Token frequency threshold	Minimum frequency of syntax token in the corpus for inclusion in vocabulary	500

Table 3: Hyperparameters to be optimized together with explanations and well-performing values.

Model architecture	Top-5 accuracy	Model size (quantized size), MB
LSTM+fully connected	0.91	202 (51)
GRU+predicted embedding	0.91	152 (38)
LSTM+predicted embedding	0.92	152 (38)
LSTM+attention	0.93	164 (41)

Table 4: Accuracy on the validation set for various neural model architectures and associated model sizes.

lines, especially for the accuracy of top-1 suggestions. Tab. 6 summarizes the performance of the Pythia system for 10 most popular completion classes. Fig. 7 illustrates a performance improvement achieved by the Pythia neural model as compared to the Markov Chain baseline on the test set. As seen from the histogram, Pythia yields over 50% accuracy improvement for nearly 6000 completion classes. Markov Chain model relies on the type inference, which results in a lower coverage than the Pythia neural model. The classes not covered by the Markov Chain model are included in the overflow bin.

Method	Top-1 accuracy	Top-5 accuracy	MRR
Alphabetic	0.36	0.47	0.372
Frequency	0.38	0.64	0.495
Frequency-if	0.40	0.67	0.521
Markov Chain	0.58	0.83	0.704
Pythia	0.71	0.92	0.814

Table 5: Accuracy and mean reciprocal ranks on the test set for the Pythia neural model and various baselines.

Class name	Top-5 accuracy, MC	Top-5 accuracy, Pythia	$\frac{\Delta Acc}{Acc}$ [%]	$N_{recommend}$
os	0.863	0.950	10.0	125673
numpy	0.575	0.697	21.2	113227
list	0.978	0.989	1.1	79406
str	0.974	0.988	1.4	77507
os.path	0.895	0.957	6.8	75270
sys	0.821	0.959	16.8	51231
wx	0.272	0.533	95.9	47821
logging	0.846	0.914	8.01	40871
time	0.951	0.980	3.1	27874
tensorflow	0.511	0.754	47.6	16620

Table 6: Performance summary on the test set for the Pythia neural model and the Markov Chain baseline. Relative percentage accuracy change is defined as: $\frac{\Delta Acc}{Acc} = 100 \cdot (Acc_{Pythia} - Acc_{MC})/Acc_{MC}$ [%]
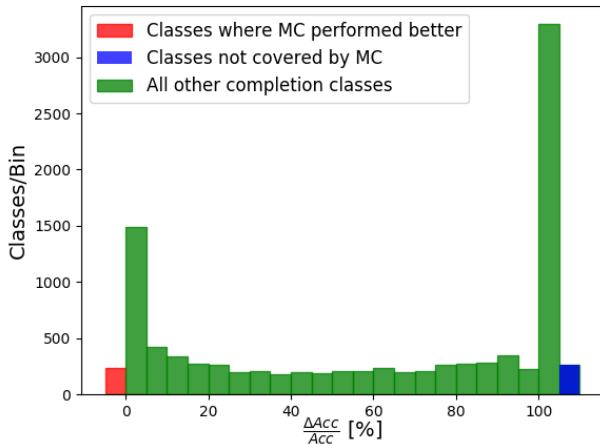

Figure 7: Number of completion classes with a given relative percentage accuracy difference for the Pythia neural model as compared to the invocation-based Markov Chain baseline. $\frac{\Delta Acc}{Acc} = 100 \cdot (Acc_{Pythia} - Acc_{MC})/Acc_{MC}$ [%]

8 MODEL DEPLOYMENT

The Pythia code completion system is currently deployed as a part of Intellicode extension [5] in Visual Studio Code IDE. The main challenges in designing and implementing an online system capable of serving models on lightweight client devices are the prediction latency and memory footprint. We apply neural network quantization into an 8-bit unsigned integer numeric format to reduce memory footprint and model size on disk.

8.1 Neural network quantization

Neural network quantization is a process of reducing the number of bits to store weights. The numerical format used during training of the Pythia neural model is IEEE 754 32-bit float. Quantization is performed layer-by-layer, extracting minimum and maximum values of weights and activations in the floating point format, zero shifting, and scaling. Given a weight matrix $W = w_{ij}$ for a layer, the quantized weight matrix W^q is obtained as:

$$\beta = \frac{max(W) - min(W)}{2^8}, \quad w_{ij}^q = \frac{w_{ij} - min(W)}{\beta}. \quad (12)$$

In the case of Pythia, post-training quantization into 8-bit integer results in model size reduction to quarter the size – from 152 MB to 38 MB – reducing top-5 accuracy from 92% to an acceptable 89%.

An example online recommendation served by the Pythia neural model is shown in Fig. 8. As seen, Pythia accurately suggests the most likely method call should involve an optimizer, with two out of the top five recommendations being Adam and SGD optimizers. Other suggestions include "Saver" – a method adding ops to save and restore variables to and from checkpoints, "exponential_decay" – controlling learning rate decay, and "Feature" – a data wrapper specific to TensorFlow, all of which are likely to be called to before creating a session and running the training ops.

___
[5]https://marketplace.visualstudio.com/items?itemName=VisualStudioExptTeam.vscodeintellicode

```
1    import tensorflow as tf
2    # Model parameters
3    W = tf.Variable([.3], tf.float32)
4    b = tf.Variable([-.3], tf.float32)
5    # Set up linear model
6    x = tf.placeholder(tf.float32)
7    linear_model = W * x + b
8    # target labels
9    y = tf.placeholder(tf.float32)
10   # loss calculation
11   loss = tf.reduce_sum(tf.square(linear_model - y))
12   # optimizer
13   optimizer = tf.train.
                      ★ AdamOptimizer
                      ★ Saver
                      ★ GradientDescentOptimizer
                      ★ exponential_decay
                      ★ Feature
                        AdadeltaOptimizer
                        AdagradDAOptimizer
                        AdagradOptimizer
                        BytesList
                        Checkpoint
                        CheckpointSaverHook
                        CheckpointSaverListener
```

Figure 8: Example code completion served by Pythia.

9 CONCLUSIONS

We have introduced a novel end-to-end approach for AI-assisted code completion called Pythia, generating ranked lists of method and API recommendations that can be used by software developers. We have deployed the system as part of Intellicode extension in Visual Studio Code IDE. Pythia makes use of long short-term memory networks trained on long-range code contexts extracted from abstract syntax trees, capturing semantics carried by distant nodes. Evaluation results on a large dataset of 15.8 million method calls extracted from real-world source code showed that our best model achieves 92% top-5 accuracy, beating simpler baselines.

We have overcome and documented several practical challenges of training deep neural networks and hyperparameter tuning on high-performance computing clusters, and model deployment on lightweight client devices to predict the best matching code completions at edit time.

Besides Python, Intellicode extension is providing AI-assisted code completions for a variety of programming languages including C#, Java, C++, and XAML based on our Markov Chain model. In the future, advanced deep learning approaches will be explored in application to programming languages other than Python, and for more sophisticated code completion scenarios.

ACKNOWLEDGMENTS

REFERENCES

[1] A. Alnusair, T. Zhao, and E. Bodden. 2010. Effective API navigation and reuse. *IRI* (2010), 7–12.

[2] 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.

[3] R. I. Brafman G. Shani and D. Heckerman. 2002. An mdp-based recommender system. In *Proceedings of 8th Conference on Uncertainty in Artificial Intelligence*. UAI.

[4] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: Continual prediction with LSTM. (1999).

[5] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR* abs/1706.02677 (2017). arXiv:1706.02677 http://arxiv.org/abs/1706.02677

[6] Alex Graves. 2013. Generating Sequences With Recurrent Neural Networks. *CoRR* abs/1308.0850 (2013). arXiv:1308.0850 http://arxiv.org/abs/1308.0850

[7] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. 2013. Complete completion using types and weights. In *PLDI'13*. ACM.

[8] Hakan Inan, Khashayar Khosravi, and Richard Socher. 2016. Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling. *CoRR* abs/1611.01462 (2016). arXiv:1611.01462 http://arxiv.org/abs/1611.01462

[9] Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, Zhiheng Huang, and Alan Yuille. 2015. Deep Captioning with Multimodal Recurrent Neural Networks (m-RNN). *ICLR* (2015).

[10] M. Maraoiu, L. Church, and A. Blackwell. 2015. An empirical investigation of code completion usage by professional software developers. (2015).

[11] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3111–3119. http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[12] B. Mobasher, H. Dai, T. Luo, and M. Nakagawa. 2002. Using sequential and non-sequential patterns in predictive web usage mining tasks. In *International Conference on Data Mining*. IEEE.

[13] G. C. Murphy, M. Kersten, and L. Findlater. 2006. How are Java software developers using the eclipse IDE? *IEEE* 23, 4 (2006), 76–83.

[14] Sebastian Proksch, Johannes Lerch, and Mira Mezini. 2015. Intelligent code completion with Bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 3.

[15] Dragomir R. Radev, Hong Qi, Harris Wu, and Weiguo Fan. 2002. Evaluating Web-based Question Answering Systems. *Language Resources and Evaluation* (2002).

[16] Colin Raffel and Daniel P. W. Ellis. 2015. Feed-Forward Networks with Attention Can Solve Some Long-Term Memory Problems. *CoRR* abs/1512.08756 (2015). arXiv:1512.08756 http://arxiv.org/abs/1512.08756

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

[18] D'Souza Andrea Renika, Di Yang, and Cristina V Lopes. 2016. Collective Intelligence for Smarter API Recommendations in Python. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*. IEEE, 51–60.

[19] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Stroudsburg, PA, 1631–1642.

[20] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3104–3112. http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

[21] A. Zimdars, D. M. Chickering, and C. Meek. 2001. Using temporal data for making recommendations. In *Proceedings of 7th Conference on Uncertainty in Artificial Intelligence*. UAI, 580–588.