Improving Code Recommendations by Combining Neural and Classical Machine Learning Approaches

Max Eric Henry Schumacher
Potsdam University
Potsdam, Germany
schumacher2@uni-potsdam.de

Kim Tuyen Le
Heidelberg University
Heidelberg, Germany
tuyen.le@informatik.uni-heidelberg.de

Artur Andrzejak
Heidelberg University
Heidelberg, Germany
artur.andrzejak@informatik.uni-heidelberg.de

ABSTRACT

Code recommendation systems for software engineering are designed to accelerate the development of large software projects. A classical example is code completion or next token prediction offered by modern integrated development environments. A particular challenging case for such systems are dynamic languages like Python due to limited type information at editing time. Recently, researchers proposed machine learning approaches to address this challenge. In particular, the Probabilistic Higher Order Grammar technique (Bielik et al., ICML 2016) uses a grammar-based approach with a classical machine learning schema to exploit local context. A method by Li et al., (IJCAI 2018) uses deep learning methods, in detail a Recurrent Neural Network coupled with a Pointer Network. We compare these two approaches quantitatively on a large corpus of Python files from GitHub. We also propose a combination of both approaches, where a neural network decides which schema to use for each prediction. The proposed method achieves a slightly better accuracy than either of the systems alone. This demonstrates the potential of ensemble-like methods for code completion and recommendation tasks in dynamically typed languages.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning approaches**; • **Software and its engineering** → **Development frameworks and environments**.

KEYWORDS

code recommendations, machine learning, neural networks

ACM Reference Format:
Max Eric Henry Schumacher, Kim Tuyen Le, and Artur Andrzejak. 2020. Improving Code Recommendations by Combining Neural and Classical Machine Learning Approaches. In *Proceedings of International Conference on Software Engineering Workshops (ICSEW'20).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3387940.3391489

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3391489

1 INTRODUCTION

Code recommendation systems for software engineering (RSSEs) are designed to accelerate the development of large software projects [17]. A widely used feature of integrated development environments (IDEs) is code completion or next token prediction [1], which is a type of RSSE. Large software projects require developers to work with massive libraries. Therefore, an use-case for next token prediction and automatic code completion could be finding the right function among thousands. An automated next token predictor assists the developer by proposing interesting or probable next tokens, which helps the developer to explore the set of possible functions that can be used at a given position within the program. In this sense, code completion can aid developers in navigating code and learning new libraries by providing information about available functions, methods or attributes.

Simulating what a user might want to write next in a text editor becomes a typical problem of code completion and code recommendation. However, this task has an intrinsic limitation, since by simulating human behavior we have to make a list of simplifying assumptions. Therefore, we can only approximate the next action a programmer might want to take. The number of possible inputs is nearly endless, making it hard to accurately predict code. For instance, a string can represent almost any word used in common language. Furthermore, a machine needs to guess what type of object a user wants to define next. Recently proposed models for code completion tend to struggle with predicting what the value of an object will be, rather than what kind of object will be used next [12] [16] [11] [4]. Here, the syntax of a programming language can help to provide more accurate predictions by limiting the search space in which a valid prediction can be found. Machine learning models can learn this syntax, which helps with predicting next tokens. However, most approaches for intelligent code completion rely on type information, which is only available at compile time [12]. Consequently, it is harder to provide code completion in dynamically typed languages like JavaScript or Python due to the lack of type information when code is written. Since dynamically typed languages are rising in popularity, the demand of intelligent code completion for these languages becomes more urgent. In this paper we introduce a generative model which can be used as a back-end for intelligent code completion or recommendation systems.

Besides, the availability of large code repositories (so-called "Big Code" [1]) such as GitHub, has lead to a rise of deep learning and probabilistic language models for code [1]. Recurrent Neural Networks (RNNs) have been prominent in predicting next tokens in code but face some limitations. Depending on the number of words

in the global vocabulary, the required computational effort to evaluate the softmax function, which is used in RNNs, can get very high. A standard way of dealing with this problem is to restrict the vocabulary. However, this approach leads to another problem: out of vocabulary words (OoV words), as defined in [11], which can not be predicted by a RNN. The recently proposed Pointer-Mixture Network [11] alleviates this problem by utilizing Pointer Networks [19] to predict OoV words. Although the Pointer-Mixture Network can establish OoV word prediction, OoV words which fall outside of the current context can not be predicted by the model.

Another notable approach for code recommendation problem is Probabilistic Higher Order Grammar (PHOG) which was proposed by Bielik et al. in [4]. PHOG utilizes production rules from a context sensitive grammar and has little restrictions in regard to vocabulary size and long-range dependencies.

In this paper, we propose an *Extended Network*, a model for prediction of terminal values of AST nodes based on a combination of the Pointer-Mixture Network and PHOG. Besides learning to predict next tokens by exploiting each of these underlying components, Extended Network also learns when to use PHOG instead of the Pointer-Mixture Network. Furthermore, our Extended Network uses an improved underlying RNN architecture that utilizes multiple long short-term memory (LSTM) layers and dropout. We implemented the Extended Network and evaluated its accuracy with different settings of relevant parameters. We have also compared the performance of each (sub-)component in order to understand the strengths of each method. Overall, our implementation was able to (moderately) surpass the performance of a Pointer-Mixture Network and a PHOG. Based on these results we believe that ensemble-like models can further improve accuracy of code recommender systems.

This paper is organized as follows. Section 2 describes the approach. The following Section 3 shows the evaluation. We outline related work in Section 4 und conclude in Section 5.

2 APPROACH

The idea behind Extended Network is to combine neural and probabilistic language models for code modeling and prediction. Extended Network is a way to build an ensemble of deep-learning and probabilistic models, which intelligently chooses between its components. Hereby, it compensates for the weaknesses of the individual models it consists of. The general methodology behind Extended Network relies on two requirements: (1) a neural network is needed to base the Extended Network upon, and (2) a set of well defined criteria on when to use which model. If the requirements (1) and (2) are met we can build an Extended Network. This is achieved by adding an output-dimension to the output layer of the neural network for each probabilistic language model which we want to utilize in the Extended Network. Therefore, each of these added output-dimensions holds an estimated probability of how likely it is that its corresponding language model produces a good prediction for the current input. The remaining outputs are words from the vocabulary.

To formally define Extended Network consider y to be the output layer of a given neural network:

$$y = f(Wx + b) \tag{1}$$

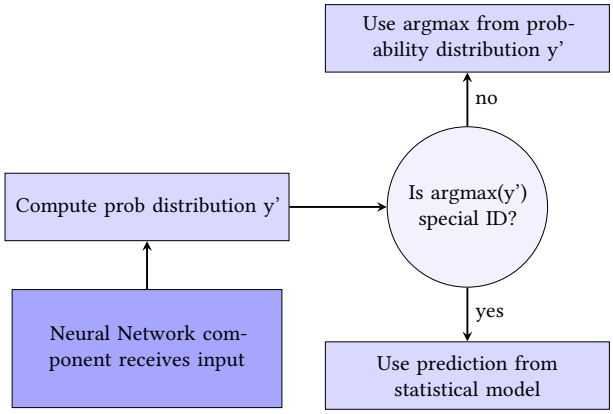

Figure 1: Flowchart describing how Extended Network formulates a prediction.

with f being an arbitrary activation function and $W \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$ being trainable parameters. The output of the neural network $y \in \mathbb{R}^n$ is a probability distribution over n possible outputs. Now let g_i with i in $\{1, .., k\}$ be k probabilistic language models. We build the Extended Network by integrating these models into the neural network by extending the output dimension of the neural network so that $y' \in \mathbb{R}^{n+k}$. This can be achieved by changing the dimensions of W and b. The resulting output of the Extended Network is computed as follows:

$$y' = f(W'x + b') \tag{2}$$

with $W' \in \mathbb{R}^{(n+k) \times m}$ and $b' \in \mathbb{R}^{n+k}$ are the trainable parameters and f is the same activation function as before. Besides giving an estimated probability distribution over a predefined vocabulary, the Extended Network also returns an estimated probability for each model on how likely it is that the model g_i, with i in $\{1, ..., k\}$, produces a correct prediction. From now on we will refer to the outputs $y'_{n+1}, ..., y'_{n+k}$ as *special IDs* which results in each probabilistic language model having its own special ID.

To illustrate how an Extended Network chooses between models, consider an Extended Network which consists of a neural network and a single probabilistic language model. The output of the Extended Network can be a word from the vocabulary or the special ID of the probabilistic model. To formulate a prediction using the Extended Network we either utilize the output of the neural network component or the prediction from the probabilistic language model. Figure 1 shows how the Extended Network chooses a model for formulating the next prediction.

Conditioning Extended Network on when to use the neural network component and when to use which of the other components is done through creating custom labels. A label is either a word from the vocabulary or any of the special IDs. An important criterion for creating labels is that they have to be deterministic. Otherwise, the network would have difficulties in learning to predict the correct special IDs or words from the vocabulary. Therefore, we create a hierarchy in which we successively test a set of conditions $C = \{c_0, ..., c_k\}$ where condition c_0 belongs to the neural network component and c_i belongs to probabilistic language models g_i for

i in $\{1, ..., k\}$. As soon as a condition is met, the label is set to the belonging special ID or word. With this kind of hierarchy, we can create deterministic labels and simultaneously prioritize the models according to the order in which we run through the conditions C. The set of conditions C and the order of running through the set has a big impact on the Extended Network's behavior. There is no single best way to formulate the conditions and the set of conditions highly depends on the application of the Extended Network.

Extending Pointer-Mixture Network with PHOG The goal of this Extended Network is to reduce the number of OoV words with long-range dependencies that can not be predicted. Since OoV words can not be predicted by the RNN component of the Pointer-Mixture Network, OoV words would normally be predicted by the pointer component of the Pointer-Mixture Network. Long-range dependencies introduce additional difficulty for the pointer component. The problem with long range dependencies in OoV words lies in the fixed size of the attention window of the pointer component. Since the prediction of the pointer component is a location within the context window, the network and its corresponding pointer distribution can only predict words which appear within the attention window. Therefore, OoV which fall outside the attention window can not be predicted by neither the pointer component nor the RNN component.

In contrast, a PHOG is theoretically not limited by vocabulary size and does not suffer from long range dependencies as much. For PHOG the vocabulary space is only limited by the number of distinct words in the training set. With the TCond language, which is is a domain specific language to move over an AST and accumulate information while traversing the tree, the TGen program of PHOG can be trained to aggregate any context and is not limited by the locations in which words appear in the AST.

To alleviate the problem of predicting OoV words with long-range dependencies, we combine a PHOG and a Pointer-Mixture Network in an Extended Network. This Extended Network will be used to predict next node values in ASTs of Python programs. The experimental setup is explained in Section 3.

Learning to predict PHOG The key idea behind Extended Network is that every time the prediction from a probabilistic language model should be used, a special ID is predicted by the neural network component. For the purpose of this study, the underlying neural network is the Pointer-Mixture Network and the probabilistic language model used to extend it is a PHOG. Whenever the PHOG should be used, the network predicts a special ID, which we refer to as *hog ID*. In all other cases the Pointer-Mixture Network should predict the next node with either the RNN or the pointer component. We achieve this by adding the hog ID to the output layer of the RNN and conditioning the Extended Network to predict it whenever the RNN and pointer component are incapable of predicting the next node. In the following, we discuss the set of conditions C and how to create the labels for the Extended Network.

Conditioning the Extended Network The Extended Network is conditioned in a similar way as the Pointer-Mixture Network is. We organize the labels for next node values in ASTs in a file which we denote as terminal corpus. In other words, the terminal corpus represents the labels for the network for training and development data-set or training and test data-set. For each node in each AST in the training and dev/test- data-set we check which component should be used to predict it. This information is encoded with either an ID from a dictionary that contains words appearing in the vocabulary (terminal dictionary: TDict), location indices from the context window or the hog ID. To further illustrate how the terminal corpus is built consider the following algorithm:

Algorithm 1 Creating the Terminal Corpus

```
1:  procedure Process(file, hogFile, TDict, attn_size)
2:      TCorpus ← [ ]
3:      for AST, hogPred in file, hog_File do
4:          TLine ← [ ]
5:          AttnQue ← deque(attn_size)
6:          for Node, hogNode in AST, hogPred do
7:              if "value" in Node.keys then
8:                  dic_value ← Node["value"]
9:                  if dic_value in TDict then
10:                     TLine.append(TDict[dic_value])
11:                     AttnQue.append("Normal")
12:                 else
13:                     if dic_value in AttnQue then
14:                         attnLoc ←
15:                                 locEq(AttnQue, dic_value)
16:                         TLine.append(attnLoc)
17:                     else if dic_value ==
18:                                 hogNode["value"] then
19:                         TLine.append(hogID)
20:                     else
21:                         TLine.append(unkID)
22:                     AttnQue.append(dic_value)
23:             else
24:                 AttnQue.append("EmptY")
25:                 TLine.append(TDict["EmptY"])
26:         TCorpus.append(TLine)
27:     return TCorpus
```

Figure 2: Algorithm for creating the terminal corpus based on the algorithm used to create the terminal corpus in [11].

First of all, a trained PHOG is used to create a set of predictions for the training and testing data. The set is stored in a *json* file similarly to the train and test data-set. Each line corresponds to an AST and has its own json object. Each object contains a value and type prediction for each node in the AST. The file *hogFile* stores the predictions from a pre-trained PHOG and is constructed so that each line in the hogFile corresponds to the AST in the training (or testing) set with the same line number. The *PROCESS* procedure (Figure 2 line 1) accepts a file which stores the ASTs, the corresponding hogFile, a dictionary with the n most appearing words (n is the vocabulary size) and the size of the attention window (attn_size). Thereafter, the terminal corpus is created. For each AST we create one line whereby each terminal node value is translated to either an ID from the terminal vocabulary, an index denoting the location in the context window, or the hogID. The resulting terminal corpus is

a list of lists with each line representing one AST's terminal values. Hereby, each value is encoded via an integer-ID.

The encoding follows a hierarchy where we condition the network on when to use which component. If possible, the ID stored in the terminal dictionary is appended to the current line in the terminal corpus. This is equivalent to telling the network to use the RNN component for prediction. If the word can not be found in the terminal dictionary and is present in the attention window, the location in the current context window is appended to the current line in the terminal corpus. If the latter does not work, the prediction of the PHOG can be used. If its prediction is correct, the hogID is appended to the current line in the terminal corpus. If not, all of the components have failed to produce a correct prediction and the unkID is appended to the current line in the terminal corpus. Figure 3 shows this procedure for the example terminal T with value "foo". Since for non-terminals there is no "value" component, we encode the absence of a value as "EmptY".

In practice, the Extended Network is conditioned to always give a prediction by denying the unkID as a prediction. For this, all unkID predictions are counted as wrong predictions. In consequence, the network will avoid predicting unkID. It would have been possible to utilize the PHOG prediction whenever we would have predicted the unkID keyword before – disregarding whether the prediction is correct or not. This would probably have conditioned the Extended Network to predict the hogID more often than practical. In short, we condition the network to utilize the PHOG prediction whenever it is correct, instead of whenever it is unsure what to predict. Although the RNN component is not able to give the correct prediction in these cases, it can produce a prediction that is close to the correct prediction in the embedding space. Whenever the PHOG fails to produce a correct prediction, the output of the RNN is preferred, because there is no similarity metric for PHOG predictions. This makes it hard to evaluate if the wrong prediction of the PHOG is close to the actual label or not.

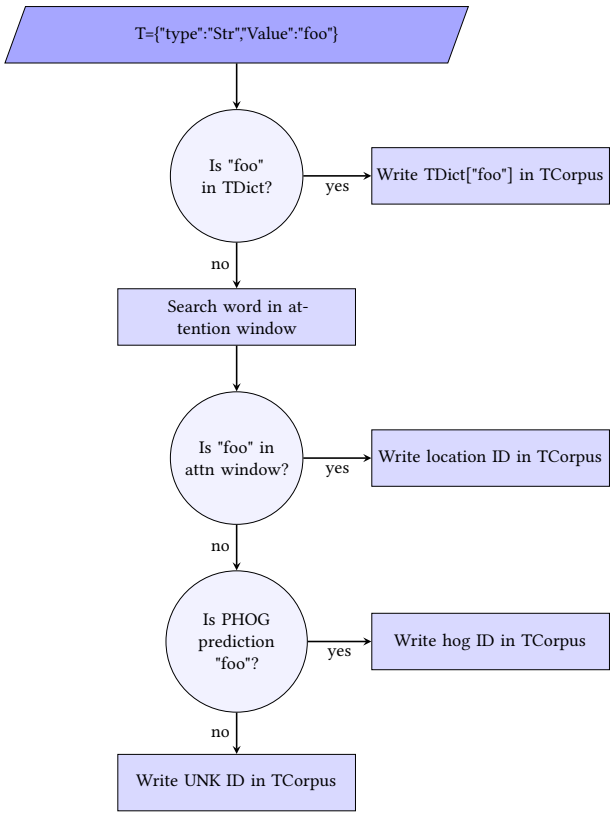

Figure 3: Process of generating a label from a terminal and writing it to the terminal corpus.

3 EVALUATION

3.1 Experimental Setup

The data-set used for all training and evaluation tasks is the 150k Python data-set from Eth-Zürich[1]. The data-set contains 150,000 Python programs which were collected from GitHub repositories. Duplicates such as forks were removed and only repositories with common license agreements like the MIT, Apache or BSD-license were used. The programs were parsed into AST format using the Python parser included in Python 2.7. The Python 150k data-set has proven its quality and has been widely used in the field of code completion [4] [11] [12] [16]. Originally, the data-set was split into 100,000 ASTs for training and 50,000 ASTs for testing.

For parameter tuning and developing the models, we split the training data using a training-development split. From the original 100,000 training ASTs we used 90,000 ASTs for the training and 10,000 ASTs for the development data-set, leaving the evaluation set untouched. By using a dedicated training-development split for parameter tuning, we keep validity of our testing data-set and make our model comparable to other models, which were also trained using the 150k Python data-set. Furthermore, this setup allows us to measure accuracy on unseen data which is helpful to detect overfitting. No parameters were tuned using the testing data-set. Only the final evaluation of our model is done on the testing data-set.

Data pre-processing. For sequence to sequence models, e.g. the Pointer-Mixture Network or Extended Network, the input sequence (flattened AST) is shifted for one time step to form labels. At time step t an input-sequence $w_0 ... w_t$ has the target/label sequence $w_1 ... w_{t+1}$. As described in section 2, for the Extended Network we need to generate a terminal corpus (modified labels) to condition the networks on using the right component for a given prediction task. We built terminal corpora for the train-dev split and train-test split for vocabulary sizes of 1,000 and 10,000 words.

Experiment configuration. The basis of all our experiments is an Extended Network consisting of a Pointer-Mixture Network and a PHOG. For all our experiments, we used a pre-trained PHOG, trained on the training data (either from the train-dev or train-test split). As a base-line for our Extended Network we used the hyper-parameters presented in [11]. Accordingly, the LSTM of the underlying Pointer-Mixture Network of our Extended Network has an unrolling length of 50, an attention window of size 50 and a hidden unit size of 1,500. For training the Extended Network, we used the cross entropy loss function with stochastic gradient descent

[1] https://www.sri.inf.ethz.ch/py150

and Adam optimizer [10] for optimization. Here, the gradient is clipped to 5 to prevent gradients from exploding [11]. We trained the models for a maximum of 8 epochs with an initial learning rate of 0.001 and a decay of 0.6 after each epoch.

Thereafter we carried out three sets of experiments: 1) we trained a single layer Extended Network with varying amounts of dropout, 2) we increased the number of layers to two layers and again tuned the amount of dropout, and 3) we used the train-test split and trained the most promising model from 1) and 2) on 1k and 10k vocabulary sizes. Performance metrics were the same among all experiments. We measured accuracy on the train- and dev-set and observed the difference between both sets.

3.2 Experimental Results

For our first experiment we did not introduce dropout or multi-layers in order to isolate the effect the Extended Network has on the accuracy. This is important to establish a fair comparison to the Pointer-Mixture Network. Multi-layers and dropout are likely to be effective on the Pointer-Mixture network as well. Therefore, the performance gains from implementing an Extended Network would be less obvious. We were able to reproduce the results presented in [11] after training the Pointer-Mixture Network for 7 epochs on the train-test data-set. After 7 epochs of training we could see an improvement of 0.6% in accuracy on the test-set and 1.9% on the train-set. Table 1 shows the results of the Extended Network on next node value predictions and compares the accuracy to the accuracy of an equivalent Pointer-Mixture Network and a PHOG.

	accuracy on test data-set
PHOG	63.8%
Pointer-Mixture Network	66.4%
Extended Network	**67.0%**

Table 1: Results after training a single layer Extended Network without dropout for 7 epochs on train-test split.

After carrying out our first experiment we noticed a variance in accuracy between the training- and testing data-set. To mitigate the variance and boost testing accuracy, we used a version of dropout, as presented in [6], which is suitable for LSTM networks. The goal was to regularize the Extended Network, to improve its ability to generalize and thus boost the accuracy on the test data-set. To find the optimal amount of dropout, we used the train-dev split and the configurations displayed in Table 2. Additionally, we increased the depth of the network by adding a second layer to the LSTM. We could measure a decreased delta between training- and developing accuracy (from 2.3% to 1.5%) when adding 20% dropout and a second layer to the network. Furthermore, the training accuracy did not change for these setups, suggesting that we did not overfit more by adding a second layer to the network.

Our final results were obtained using a 2 layer Extended Network with 20 % dropout trained and evaluated on the original train-test split of the 150k Python data-set. Table 3 shows the results for the Extended Network on a vocabulary size of 10,000 and 50,000 words (the evaluation test) and compares it to competing probabilistic language models [16][4] as well as the Pointer-Mixture network

	Dropout Percentage	dev acc	train acc
Single Layer	0 %	66.6 %	69.9%
	20 %	66.7 %	68.5%
	40 %	66.3 %	67.6%
Two Layers	15 %	66.7 %	68.1%
	20 %	**67.0 %**	68.5%
	30 %	66.5 %	67.6%

Table 2: Accuracies for single layer Extended Networks and two layer Extended Networks for varying amounts of dropout.

[11]. The Extended Network was trained on a Nvidia Tesla V100 GPU and took approximately 1.5 hours per epoch on a 1k vocabulary and 8 hours per epoch on a 10k vocabulary.

	Accuracy for value prediction	
PHOG	63.8%	
Decision Trees	69.2%	
	1k vocab	10k vocab
Pointer-Mixture Network	66.4%	68.9%
Extended Network (ours)	67.5%	**69.3%**

Table 3: Accuracy of value predictions for Extended Network compared to state-of-the art probabilistic models and the original Pointer-Mixture network.

3.3 Detailed Method Comparison

We have performed an additional analysis in order to compare in detail the performance of each of the components of the Extended Network. We use the same model configuration as in Section 3.2 with a vocabulary size of 10,000, and perform experiments on an evaluation dataset with 50,000 Python source files. We focus here on the value prediction only (i.e. prediction of the terminal values in AST nodes).

Table 4 shows the results. Column "RNN" describes the pure LSTM-model, "Attn" the pure attention mechanism, and "PHOG" the approach from [4]. The rows "Able to predict" show the percentage of cases where a component was in principle able to predict. For example, if the target token is not in the terminal dictionary TDict, the RNN is not able to predict at all. Note that PHOG does not have such constraints and is always able to predict.

The rows "Used as predictor" indicate how many times a particular prediction component was really used. Essentially, it shows the distribution of the model component selection mechanism illustrated in Figure 1. Overall, the most used component is the RNN. Other components are used by the selection mechanism roughly according to their accuracy. The preference for RNN can be attributed to the fact that this model has seeming specialized on predicting the "EmptY" token (as noted below, it appears frequently in the dataset).

The rows "Used and correct" show the percentage of correct predictions (among all predictions) of a component if it was selected.

Overall, RNN contributes most correct prediction results, while the impact of the pure attention mechanism and of the PHOG are marginal.

The rows "Correct own predictions" indicate the precision of each component by itself. The numbers show the percentage of correct predictions of a component among all own predictions. It shows that RNN is correct for almost 60% of own predictions (but see discussion in Section 3.4 on predicting the "EmptY" token why this figure is rather misleading). The least precise component is PHOG with roughly 5% correct predictions.

	RNN	Attn	PHOG
Able to predict	84.2%	5.4%	100%
Used as a predictor	91.5%	1.1%	7.4%
Used and correct	54.1%	0.2%	0.4%
Correct own predictions	59.1%	20.2%	5.4%

Table 4: Accuracy of the tree components of the Extended Network including predictions of nodes without value, i.e. "EmptY" (for a dictionary size of 10,000 on an evaluation dataset with 50,000 ASTs).

3.4 AST Nodes without Values

A large part for AST nodes in our dataset (or in general, in a representative Python source code) are lacking a value. Such AST nodes are typically non-terminal nodes or terminal nodes manifesting in sources code with a fixed keyword (e.g. "as"). Following [11], we consider *all* nodes with this property as a special value in the dictionary TDict, i.e. as a special case of a node value (denoted as "EmptY" in Section 2, or EMPTY in [11, Fig. 4]).

The component RNN is solely responsible for predicting such "EmptY" nodes. Essentially, a value prediction of type "EmptY" means that we should subsequently invoke a (separate) prediction model for predicting the *type* of this AST node, and - if applicable - suggest as a completion node's unique representation in the source code (see [11]).

Training and evaluating models with this special value "EmptY" creates some problems. First, a large share of AST nodes (in our case, 47.6% out of 29,903,343 prediction instances) are such "EmptY" nodes. This might bias the RNN model to focus on this particular special value during training. Furthermore, predicting this special value might be easier than others, since the context of AST nodes without value might give hints not present for other node values. E.g., it is quite likely to encounter keyword "as" shortly after keyword "with".

In general, including the predictions of this special value in the evaluation significantly boosts the accuracy values, which we also observed in our experiments. In our main evaluation, we follow a schema used in the previous works [4, 11] and include predictions of the "EmptY" nodes in order to provide comparable results. However, we do not consider such a evaluation a realistic one since it might not correspond to a perceived user experience. In fact, other state-of-the-art works [2, 5] report much lower accuracy values for code completion (e.g. accuracy@5 of 24.83% for Java [2]).

Consequently, we supply complementary results of an evaluation with all AST nodes with the special value "EmptY" being ignored. Note that such nodes are still used in the training, and so the presented results might be worse than in a setting where they are not considered at all.

Table 5 shows the results (the meaning of entries is analogous to those in Section 3.3). The accuracy of the RNN component drops significantly (from 54.1% to 18.8%). This confirms that the much better accuracy of this component reported in Table 4 can be attributed to (correct) predictions of the "EmptY" values. As expected, the individual performances (fraction of correct own predictions) of the PHOG model and the attention mechanism remain nearly the same. With 84.3% of invocations the RNN model is still preferably used, which could be attributed to the training with the "EmptY" values. Due to this fact it still dominates the overall performance of the Extended Network, which drops from 54.7% (in Section 3.3)) to now only 20.0%.

	RNN	Attn	PHOG
Able to predict	70.0%	10.3%	100%
Used as a predictor	84.3%	2.0%	13.7%
Used and correct	18.8%	0.4%	0.7%
% of own correct	22.3%	20.7%	5.6%

Table 5: Accuracy of the tree components of the Extended Network with ignoring AST nodes with value "EmptY". Overall model accuracy drops to 20.0%.

4 RELATED WORK

Natural Language Processing (NLP) can be defined as an intersection of artificial intelligence and linguistics [15]. The hypothesis that code recommendation can be framed as a natural language problem is the basis for machine learning models which originate from NLP.

The natural hypothesis As presented in [1], states that software is a form of human interaction and software corpora can be used similarly to natural language corpora. This analogy of code to natural language has been studied as well as the repetitiveness and predictability of code have been shown. For instance, n-grams [8][18] have been used to demonstrate the naturalness and repetitiveness of code; many machine-learning and deep-learning models that originate from NLP have proven they work well for code [12][16] [11][14].

Machine Learning Models for Code Completion To apply machine learning to intelligent code completion we treat code as natural language and represent program code with abstract syntax trees (ASTs). The code completion task is defined accordingly as: given an AST, predict the next node in the tree. Some typical techniques of neural networks used to solve this problem are Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) and Word embedding [14][9][13]. Besides, the performance of a neural network drastically improves with the amount of data that can be utilized for training and testing the network. In 2017 Liu et al [12] proposed a LSTM that learns code completions from a large corpus of dynamically typed JavaScript code. The LSTM they introduced

competes with state-of-the-art probabilistic models for code, e.g. decision trees [16]. However, these standard neural models suffer from long-range dependencies and out-of-vocabulary (OoV) words [11].

Neural Attention and Pointer Networks Recently, further networks have been proposed for code completion which are based on a LSTM architecture and neural attention/pointer networks [11]. The Pointer-Mixture Network introduced by Li et. al in 2018 [11] combines an attentional-LSTM and a Pointer Network to tackle the problem of OoV words and long range dependencies. Since storing the information in a single vector of fixed length in RNNs leads to a *hidden state bottleneck* [11] in capturing long-range dependencies, more than one hidden state vector is needed to propagate information through the RNNs time steps over extended time intervals. The attention mechanism is proposed in [3] to address this issue, which utilizes a weighted sum of past and current hidden state vectors. For an attentional LSTM, there are two considered types of attention: *Context Attention* and *Parent Attention* as defined in [11]. While the first one is an attention mechanism with a fixed-sized context window, the latter is a special form of Context Attention used for ASTs which adds the hidden state from the parent of the currently processed node to the attentional layer.

Inspired by the *localness of software* hypothesis, which states that source code is locally repeated [18], Pointer Networks is proposed to reduce the problem of static output vocabularies by predicting tokens from the input sequence [19]. More precisely, the Pointer Network predicts next tokens by *pointing* to a token from the input sequence.

Last but not least, **Probabilistic Higher Order Grammar** (PHOG) was first introduced in [4] by Bielik et. al. Prior to their model, probabilistic context free grammars (PCFG) [7] and n-gram models [18] have been used as generative models for code. PHOG generalizes upon PCFG, by not only conditioning on the parent non-terminal node, but also dynamically accumulating a context through navigating over the AST. In contrast to n-gram models and PCFG, PHOG dynamically generates a program representation through the generated context. PCFGs and n-gram models, on the other hand, are often engineered to fit a domain specific problem and therefore do not perform well in other domains. By the flexible program representation, PHOG becomes widely applicable and can be applied to any programming language that features a representation through ASTs.

5 CONCLUSION

This paper demonstrates the potential of the Extended Network as an ensemble method for predicting next tokens for code completion in dynamically typed languages.

The Extended Network based on a Pointer-Mixture Network and a PHOG showed improvements in accuracy over each of the stand-alone models. Iterating on the RNN architecture illustrated the importance of a well performing RNN for an Extended Network.

We understand as Extended Network an ensemble-like architecture designed to fit arbitrary models and able to include future approaches as components of the ensemble model. Possible extensions could be n-gram models, variations of n-gram models, e.g. cached n-gram models and nested n-gram models, as well as decision trees. In our opinion, the most promising approaches for future research are: creating bigger ensembles, finding the best hierarchy of models, and tuning the underlying RNN architectures.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (July 2018), 37 pages. https://doi.org/10.1145/3212695

[2] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural Language Models of Code. *arXiv:1910.00577* (Feb. 2020). http://arxiv.org/abs/1910.00577

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1409.0473

[4] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 48)*. PMLR, New York, New York, USA, 2933–2942. http://proceedings.mlr.press/v48/bielik16.html

[5] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *International Conference on Learning Representations (ICLR 2019)*. http://arxiv.org/abs/1805.08490

[6] Yarin Gal and Zoubin Ghahramani. 2016. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. In *Advances in Neural Information Processing Systems 29*. Curran Associates, Inc., 1019–1027. http://papers.nips.cc/paper/6241-a-theoretically-grounded-application-of-dropout-in-recurrent-neural-networks.pdf

[7] T. Gvero and V. Kuncak. 2015. Interactive Synthesis Using Free-Form Queries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 689–692. https://doi.org/10.1109/ICSE.2015.224

[8] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[9] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[10] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.6980

[11] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. ijcai.org, 4159–4165. https://doi.org/10.24963/ijcai.2018/578

[12] Chang Liu, Xin Wang, Richard Shin, Jose Enrique Gonzalez, and Dawn Song. 2017. Neural Code Completion.

[13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. http://arxiv.org/abs/1301.3781

[14] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. 2013. Linguistic Regularities in Continuous Space Word Representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Atlanta, Georgia, 746–751. https://www.aclweb.org/anthology/N13-1090

[15] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. 2011. Natural language processing: an introduction. *Journal of the American Medical Informatics Association* 18, 5 (09 2011), 544–551. https://doi.org/10.1136/amiajnl-2011-000464 arXiv:http://oup.prod.sis.lan/jamia/article-pdf/18/5/544/5962687/18-5-544.pdf

[16] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA, 731–747. https://doi.org/10.1145/2983990.2984041

[17] M. Robillard, R. Walker, and T. Zimmermann. 2010. Recommendation Systems for Software Engineering. *IEEE Software* 27, 4 (July 2010), 80–86. https://doi.org/10.1109/MS.2009.161

[18] 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* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 269–280. https://doi.org/10.1145/2635868.2635875

[19] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. In *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc., 2692–2700. http://papers.nips.cc/paper/5866-pointer-networks.pdf