Bidirectional Transformer Language Models for Smart Autocompletion of Source Code

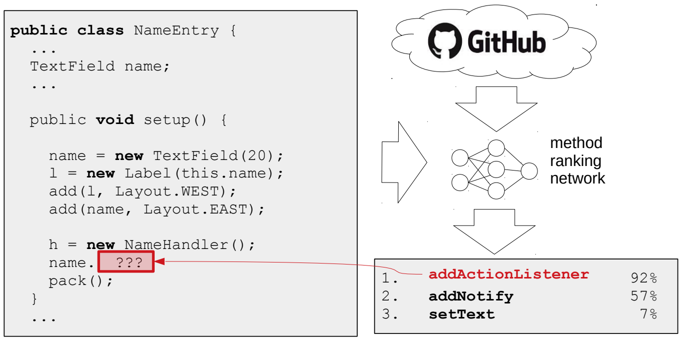
Felix Binder[1], Johannes Villmow[1], Adrian Ulges[1]

Abstract: This paper investigates the use of transformer networks – which have recently become ubiquitous in natural language processing – for smart autocompletion on source code. Our model *JavaBERT* is based on a RoBERTa network, which we pretrain on 250 million lines of code and then adapt for method ranking, i.e. ranking an object's methods based on the code context. We suggest two alternative approaches, namely unsupervised probabilistic reasoning and supervised fine-tuning. The supervised variant proves more accurate, with a top-3 accuracy of up to 98%. We also show that the model – though trained on method calls' full contexts – is quite robust with respect to reducing context.

Keywords: smart autocompletion; deep learning; transformer networks

1 Introduction

AI-based support in software engineering has recently emerged as a research field, and recommenders for software commits [Da16], prediction of code changes [Zh19] or semantic code search [Hu19] have been developed. These are usually trained on vast amounts of source code and documentation from open-source platforms such as GitHub. Another challenge – and the subject of this paper – is *smart autocompletion*: As the developer types source code, a *neural network* suggests names for methods to use next. We refer to this challenge of ranking an object's method names by their plausibility in a given code context as *method ranking*. Figure 1 illustrates this, where a neural network has learned a suggestion from GitHub projects including code passages similar to the target context.



```
public class NameEntry {
  ...
  TextField name;
  ...

  public void setup() {

    name = new TextField(20);
    l = new Label(this.name);
    add(l, Layout.WEST);
    add(name, Layout.EAST);

    h = new NameHandler();
    name. ???
    pack();
  }
  ...
```

```
1.   addActionListener   92%
2.   addNotify           57%
3.   setText              7%
```

Fig. 1: A method ranking network analyzes a position in Java source code (red, left), and infers that – out of the class `TextField`'s methods – `addActionListener()` seems most plausible.

[1] RheinMain University of Applied Sciences, DCSM Department, Wiesbaden/Germany, felix.d.k.binder@gmail.com, [johannes.villmow, adrian.ulges]@hs-rm.de

doi:10.18420/inf2020_83

While previous work on method ranking has used n-grams [Hi12] or recurrent networks [Wh15], we evaluate the transformer-based *masked language model* BERT [De18] (more precisely, its RoBERTa variant [Li19]). This approach has been very successful in natural language processing, but there has been – to the best of our knowledge – only one recent publication on smart autocompletion in source code [Ki20]. We call our model *JavaBERT* (since our focus lies on the Java programming language).
To utilize JavaBERT for method ranking, we propose two alternatives addressing the fact that method names may consist of multiple tokens (e.g., `add/Action/Listen/er`).:

1. *JavaBERT-unsup*: The pretrained (unsupervised) JavaBERT model is applied by masking out varying numbers of tokens. JavaBERT's predictions on token level are then combined in a probabilistic reasoning to predictions on method level.
2. *JavaBERT-sup*: JavaBERT is fine-tuned supervisedly as a binary classifier, estimating whether a certain method call is plausible or not in a given code context.

We evaluate both models in quantitative experiments on random samples from the GitHub Java Corpus [AS13]. Our results indicate that masked language modeling is surprisingly accurate, with a top-3 accuracy of up to 98%. We also study the impact of different contexts, e.g. only the code up to the target method call, or shorter vs. larger pieces of code.

2 Related Work

Smart autocompletion The task of code completion has been addressed since 2012 by using n-gram models [Hi12, AS13], cached n-gram models for improved localization [Fr15, TSD14, HD17] and graph-based statistical language models [NN15]. More recently, the availability of large code bases has facilitated the creation of neural network language models, including recurrent neural networks [Wh15, Ra16, Li17], gated recurrent neural network models [KS19] and LSTM models [Da16]. Most recent code completion models for Java use a single layer gated RNN [Ka20] model with Byte-Pair Encoding [SHB16].

Transformer networks in NLP Transformer networks [Va17] use the concept of attention [BCB14] to derive contextualized representations for the single tokens from a sequence (i.e. a sentence or paragraph). Probably the most prominent model is BERT [De18], which applies masked language modeling, i.e. the model is trained to predict random masked tokens in the training text. Other variants use generative transformers (GPTs) trained by left-to-right language modeling [Ra18], optimize hyperparameters such as model depth and learning rate (RoBERTa [Li19]), reduce the amount of parameters (ALBERT [La20]) or perform an adversarial training (ELECTRA [Cl20]). While transformer models have been extremely successful and intensely studied in processing *natural language* recently, we are only aware of one recent publication employing them for code completion [Ki20]. While this work uses an autogenerative model, we employ *masked* language modeling on pieces of source code.

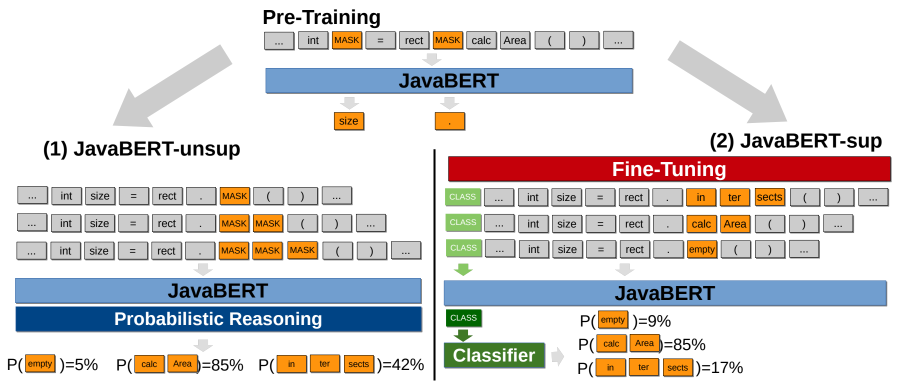

Pre-Training
... int MASK = rect MASK calc Area ( ) ...
JavaBERT
size
.
(1) JavaBERT-unsup
... int size = rect . MASK ( ) ...
... int size = rect . MASK MASK ( ) ...
... int size = rect . MASK MASK MASK ( ) ...
JavaBERT
Probabilistic Reasoning
P(empty)=5%
P(calc Area)=85%
P(in ter sects)=42%
(2) JavaBERT-sup
Fine-Tuning
CLASS ... int size = rect . in ter sects ( ) ...
CLASS ... int size = rect . calc Area ( ) ...
CLASS ... int size = rect . empty ( ) ...
JavaBERT
CLASS
Classifier
P(empty)=9%
P(calc Area)=85%
P(in ter sects)=17%

Fig. 2: Our approach JavaBERT is first pretrained using Masked Language Modeling (top). Afterwards, method ranking can either use masking with probabilistic reasoning (JavaBERT-unsup, left) or fine-tuning with a binary classifier (JavaBERT-sup, right).

3 Approach

As shown in Figure 2, our approach pretrains an encoder (JavaBERT) by masked language modeling. The resulting model can either be applied in an unsupervised fashion (using probabilistic reasoning) or by fine-tuning it into a supervised binary classifier. We discuss these three processing steps in depth in the following subsections.

3.1 Pretraining

Our approach starts with training a RoBERTa model on the Github Java Corpus [AS13] using the *fairseq* library [Ot19]. RoBERTa replicates and improves key hyperparameters of the well-known BERT model [De18] for a more robust training.
The training set of the Github Java Corpus consists of around 1.1 billion tokens of Java source code, from which we separate the last 5% into a validation set. We tokenize all Java code with a language parser[2] to separate natural language identifiers from syntax symbols. Thereby, we also remove multiple whitespace and replace string, character, float, and integer literals with a respective constant (e.g. `<INT>`). As source code may contain unicode identifiers, we use the *unidecode* library to transcribe any non-ASCII letters into ASCII.
Afterwards, we train and apply a Byte-Pair Encoding [SHB16] of V=10,000 sub-words on the tokenized source code. We perform neither lower-casing nor camel-case splitting as it is often done in machine learning on source code [ALY18]. A vocabulary of 10K tokens is used, which we assume is sufficient for source code (most identifiers are rather short or combined with camel-casing) while improving training speed. For example, our preprocessing tokenizes `public float myFloat = 10.0;` into `public/float/my/Float/=/<FLOAT>/;`.

[2] We use the *javalang* library for tokenization.

Model and Training We use a configuration similar to the RoBERTa$_{BASE}$ model (12 layers, 768-dimensional embeddings, 12 attention heads, 110M params total). Like ToBERTa, we use GELU activation functions [HG16], learned positional embeddings and a dropout of 0.1 throughout all layers and activations. The model is optimized with the Adam optimizer [KB14] (β_1=0.9, β_2=0.98, ϵ=10^{-6}, weight decay 0.01) using a linear warm-up of the learning rate for $6K$ steps up to $6{\times}10^{-4}$ followed by a cosine decay over $30K$ steps.
For efficiency reasons, we first train $15K$ steps on shorter code blocks of 128 tokens each (batch size $8K$) and then increase the block length to 512 (batch size 1,500). The sampled blocks do not cross document (i.e. source file) boundaries. We use *gradient accumulation* to mimic larger batch sizes on our limited hardware (6 NVIDIA GeForce GTX 1080 Ti).
After a total training time of about two weeks, the JavaBERT model reached a validation perplexity of 1.16 for predicting masked out tokens, which is significantly better than common results for *natural* language (which is more ambiguous and unstructured).

3.2 Unsupervised Method Ranking (JavaBERT-unsup)

We assume an incomplete piece of code to be given, which is a sequence of n tokens $T{:=}(t_1, \dots, t_s, \dots, t_n)$ containing a slot t_s for the missing method call, and a set of candidate method names $\{C^1, \dots, C^l\}$ which contains the correct method call C^* as well as other method names from the same class as C^*. The task is to maximize the probability $P(C^*|T)$. Note that – after tokenization – each candidate method name may consist of multiple tokens, i.e. $C{=}(c_1, \dots, c_L)$ with $L{>}1$.
Note that the JavaBERT model's original training is *similar* to method ranking: The original source sequence is transformed into a sequence T', in which – similar to our input sequence – random tokens t_i have been masked out by replacing them with a `<mask>` token. During training the probability $P(T_i{=}t_i|T')$ of predicting the masked token is maximized. The key difference with method ranking is that *multi-token* method names have to be predicted. To do so, we calculate the probability of a candidate method (e.g., $C{=}(c_1, .., c_L)$) by replacing the slot token t_s with L `<mask>` tokens, obtaining a sequence T^L. Then, the overall probability of candidate C is defined as

$$P(C|T) = P(L) \cdot \prod_{j=1}^{L} P(T_{s+j-1}{=}c_j|T^L) \tag{1}$$

$P(L)$ acts as a prior on method name length, exploiting the fact that shorter names are more likely (an estimate on the training set is given in Table 1).

L	1	2	3	4	5	6	7	8	9	10+
$P(L)$	26.6%	23.9%	21.0%	12.4%	6.9%	3.9%	2.2%	1.2%	0.7%	1.2%

Tab. 1: The prior $P(L)$ indicates the probability of different method name lengths L.

3.3 Supervised Method Ranking (JavaBERT-sup)

Our second approach *fine-tunes* the pretrained JavaBERT model using a supervised training. The idea is to insert candidate method names into code blocks and estimate their plausibility with a binary classifier. To do so, we use a classifier token t_{CLASS} (as is common practice), replace the slot token t_s with the candidate C, and add markers t_{START} and t_{END} before the bound object t_{OBJ} and after the method call, resulting in the input sequence

$$T^C := (t_{CLASS}, \ldots, t_{START}, t_{OBJ}, t_{dot}, \overbrace{c_1, \ldots, c_L}^{=C}, t_{END}, \ldots, t_n).$$

We encode this sequence with the JavaBERT model and feed the resulting contextualized classifier token into a binary classifier, which is trained to predict whether T^C contains the correct method name. The classifier first projects the encoded representation with a linear layer into another embedding space of the same dimensionality as the JavaBERT model, followed by layer normalization and a second projection to our binary output space.
A training set of $3.3M$ labeled code blocks is constructed of $2,649$ repositories from the GitHub Java Corpus' test split. Each positive sample (containing the true method call) is complemented with six negative samples, three of which feature another method name from the same class and the other three containing another random method name from the corpus. The model was fine-tuned for 5 days on 4 GPUs.

4 Experiments

This section compares the models JavaBERT-sup and JavaBERT-unsup in quantitative experiments on held-out test data from the Github Java Corpus. We also analyze how different amounts of context information affect the model's accuracy.
For this, we use another part of the original test split, which consists of 969 repositories that are not overlapping with the repositories used for pretraining or fine-tuning from Section 3. From these test projects, we sample $14K$ random code blocks of up to 504 tokens each. In each block, a randomly selected method call is chosen as slot t_s, and a list of candidate method names to be ranked is extracted from the method call's bound object's class. The median length of those candidate lists is 39.

Comparison Unsupervised vs. Supervised We compare the supervised and unsupervised model by measuring the hits@1, hits@3 and hits@5 rates, and the mean reciprocal rank (MRR). For example, a hits@3 of 98% indicates that for 98% of our $14K$ test blocks, the model ranks the correct method name among the top 3. Table 2 illustrates the results. Both approaches surpass a baseline that ranks the target methods randomly and the supervised approach outperforms the unsupervised model by a significant margin, especially for the top 1 predictions (difference $\approx$ 15%) and mean reciprocal rank (difference $\approx$ 10%).
Figure 3 illustrates an example for a random piece of code. Here, both models (supervised and unsupervised) rank the 8 candidate methods from Class `Scanner`, and the method `nextInt()` is ranked highest correctly. Overall, we found the model to prefer methods

	JavaBERT-unsup	JavaBERT-sup	Random guessing
hits@1	77.5	**92.3**	7.4
hits@3	92.5	**98.0**	20.8
hits@5	94.6	**98.9**	29.9
MRR	85.4	**95.2**	18.3

Tab. 2: Results of method ranking: The supervised approach significantly outperforms the unsupervised one and shows remarkable accuracy (hits@3 is 98%). We report all values as a percentage.

```java
import java.util.Scanner;

public class EvenOdd {
  public static void main(String[] args) {
    Scanner reader = new Scanner(System.in);
    System.out.print("Enter a number: ");
    int num = reader . SLOT  ();
    String evenOdd = (num % 2 == 0) ? "even":"odd";
    System.out.println(num + "is" + evenOdd);
  }
}
```

JavaBERT-unsup	JavaBERT-sup
1: nextInt	1: nextInt
2: nextLong	2: nextShort
3: nextShort	3: close
4: nextByte	4: hasNext
5: skip	5: nextByte
6: close	6: skip
7: hasNext	7: locale
8: locale	8: nextLong

Fig. 3: Given this example code (left) with a left out target method (`SLOT`), both JavaBERT variants rank the correct method (`nextInt()`) out of 8 candidate methods highest.

with the correct parameters and return types (e.g., boolean methods are ranked high in if-statements). Note that this is not inferred from a static code analysis but only from the method name (e.g., `isEmpty`, `hasConnection`). Also, we found methods that have already been defined or used in the context to be ranked higher.

Context Analysis So far, we have trained and tested our model on *full* code contexts, including the code before and after the target method as well as their arguments. In practice, e.g. when typing code from left to right, only the code before the target may be available. Also, it is interesting how much context is required for a stable inference. Therefore, we evaluate JavaBERT-sup (trained with full contexts) on various forms of reduced context:

- **Original**: uses the complete context.
- **PC** (**P**receding **C**ontext): all tokens after the candidate method are removed, the context only consists of preceding tokens.
- **FC** (**F**ollowing **C**ontext): all tokens before the candidate method call (more precisely, before the bound object) are removed.
- **PC+ParaC** (**P**receding **C**ontext plus **Para**meter **C**ontext): Most tokens after the candidate method token are removed. Only eight complete words or symbols following the candidate method are kept, which can contain up to four parameters.
- **FAM** (**F**ew words **A**round **M**ethod): Only eight complete word or symbols preceding and following the method call are kept.
- **MAM** (**M**ore words **A**round **M**ethod): Only 40 complete word or symbols preceding and following the method call are kept.

	Original	**PC**	**FC**	**PC+ParaC**	**FAM**	**MAM**
hits@1	**92.3**	70.5	73.2	86.0	61.9	77.7
hits@3	**98.0**	86.1	85.5	94.3	73.8	87.3
hits@5	**98.9**	90.5	88.8	96.1	77.0	89.8
MRR	**95.2**	79.0	80.0	90.0	69.0	83.0

Tab. 3: Comparing JavaBERT's ranking accuracy with different context windows.

These experiments are based on the same test set as before. Since the location of the target method call in a code block is chosen randomly, the amount of text for different context forms varies accordingly. Table 3 shows the results of this experiment. As expected, using the full context performs best. The follow-up run is PC+ParaC, indicating that the parameters of a method call form an important source of information for the ranking model.
Comparing the results from FAM to PC, FC and MAM and from original to MAM showcases the influence of input size on the method ranking. An observation on the three best runs (Original, PC+ParaC and MAM) is that those are a combination of preceding and following content and are ordered descendingly by the size of their input. The results from MAM show, however, that combining the preceding and following content has a larger influence on the method ranking than the input size when compared to PC and FC. In conclusion, using surrounding content rather than only the preceding content like left-to-right models does have an impact on the ranking of candidate methods.

5 Conclusions

In this paper, we have shown that transformer networks pretrained by masked language modeling are a promising approach towards method ranking. Particularly, we have demonstrated the benefits of supervised fine-tuning and studied different context windows, whereas surprisingly small context windows combining a bit of preceding and following code suffice for an accurate inference. Our future work will focus on enhancing JavaBERT (which is a token-only model) with syntax trees to obtain richer code representations, as well as tackling other challenges in automated source code understanding, such as code search and summarization.

Bibliography

[ALY18] Alon, Uri; Levy, Omer; Yahav, Eran: code2seq: Generating Sequences from Structured Representations of Code. CoRR, abs/1808.01400, 2018.

[AS13] Allamanis, Miltiadis; Sutton, Charles: Mining Source Code Repositories at Massive Scale using Language Modeling. In: Proc. MSR. pp. 207–216, 2013.

[BCB14] Bahdanau, Dzmitry; Cho, Kyunghyun; Bengio, Yoshua: Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.

[Cl20] Clark, Kevin; Luong, Minh-Thang; Le, Quoc V.; Manning, Christopher D.: ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In: Proc. ICLR. 2020.

[Da16] Dam, Hoa Khanh; Tran, Truyen; Grundy, John; Ghose, Aditya: DeepSoft: A Vision for a Deep Model of Software. In: Proc. 24th ACM SIGSOFT FSE. p. 944–947, 2016.

[De18] Devlin, Jacob; Chang, Ming-Wei; Lee, Kenton; Toutanova, Kristina: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2018.

[Fr15] Franks, C.; Tu, Z.; Devanbu, P.; Hellendoorn, V.: CACHECA: A Cache Language Model Based Code Suggestion Tool. In: Proc. ICSE. pp. 705–708, 2015.

[HD17] Hellendoorn, Vincent J.; Devanbu, Premkumar: Are Deep Neural Networks the Best Choice for Modeling Source Code? In: Proc. ESEC/FSE 2017. pp. 763–773, 2017.

[HG16] Hendrycks, Dan; Gimpel, Kevin: Gaussian error linear units (gelus). arXiv preprint arXiv:1606.08415, 2016.

[Hi12] Hindle, Abram; Barr, Earl T; Su, Zhendong; Gabel, Mark; Devanbu, Premkumar: On the Naturalness of Software. In: 2012 34th ICSE. IEEE, 2012.

[Hu19] Husain, Hamel; Wu, Ho-Hsiang; Gazit, Tiferet; Allamanis, Miltiadis; Brockschmidt, Marc: CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv preprint arXiv:1909.09436, 2019.

[Ka20] Karampatsis, Rafael-Michael; Babii, Hlib; Robbes, Romain; Sutton, Charles; Janes, Andrea: Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. arXiv preprint arXiv:2003.07914, 2020.

[KB14] Kingma, Diederik P; Ba, Jimmy: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[Ki20] Kim, Seohyun; Zhao, Jinman; Tian, Yuchi; Chandra, Satish: Code Prediction by Feeding Trees to Transformers. arXiv preprint arXiv:2003.13848, 2020.

[KS19] Karampatsis, Rafael-Michael; Sutton, Charles A.: Maybe Deep Neural Networks are the Best Choice for Modeling Source Code. CoRR, abs/1903.05734, 2019.

[La20] Lan, Zhenzhong; Chen, Mingda; Goodman, Sebastian; Gimpel, Kevin; Sharma, Piyush; Soricut, Radu: ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In: Proc. ICLR. 2020.

[Li17] Li, Jian; Wang, Yue; King, Irwin; Lyu, Michael R.: Code Completion with Neural Attention and Pointer Networks. CoRR, abs/1711.09573, 2017.

[Li19] Liu, Yinhan; Ott, Myle; Goyal, Naman; Du, Jingfei; Joshi, Mandar; Chen, Danqi; Levy, Omer; Lewis, Mike; Zettlemoyer, Luke; Stoyanov, Veselin: RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv preprint arXiv:1907.11692, 2019.

[NN15] Nguyen, A. T.; Nguyen, T. N.: Graph-Based Statistical Language Model for Code. In: Proc. ICSE. pp. 858–868, 2015.

[Ot19] Ott, Myle; Edunov, Sergey; Baevski, Alexei; Fan, Angela; Gross, Sam; Ng, Nathan; Grangier, David; Auli, Michael: fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In: Proc. NAACL-HLT. pp. 48–53, 2019.

[Ra16] Raychev, Veselin: Learning from large codebases. PhD thesis, ETH Zurich, 2016.

[Ra18] Radford, Alec; Narasimhan, Karthik; Salimans, Tim; Sutskever, Ilya: Improving language understanding by generative pre-training. OpenAI Blog, 2018.

[SHB16] Sennrich, Rico; Haddow, Barry; Birch, Alexandra: Neural Machine Translation of Rare Words with Subword Units. In: Proc. ACL. Berlin, Germany, pp. 1715–1725, August 2016.

[TSD14] Tu, Zhaopeng; Su, Zhendong; Devanbu, Premkumar: On the Localness of Software. In: Proc. 22nd ACM SIGSOFT FSE. ACM, pp. 269–280, 2014.

[Va17] Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N; Kaiser, Łukasz Kaiser; Polosukhin, Illia: Attention is All you Need. In: NIPS 2017, pp. 5998–6008. 2017.

[Wh15] White, Martin; Vendome, Christopher; Linares-Vásquez, Mario; Poshyvanyk, Denys: Toward Deep Learning Software Repositories. In: Proc. MSR. pp. 334–345, 2015.

[Zh19] Zhao, Rui; Bieber, David; Swersky, Kevin; Tarlow, Daniel: Neural Networks for Modeling Source Code Edits. arXiv preprint arXiv:1904.02818, 2019.