RoBERTvar: Improving code variable name suggestions with fine-tuned language models

G039 (s2449298, s1974565, s2314595)

Abstract

This paper presents a method for transfer learning by fine-tuning two BERT-based pre-trained transformers to perform the downstream task of code variable name suggestion. We compare a code-naive RoBERTa to a code familiar CodeBERT and find that CodeBERT is more capable of the downstream task in all metrics, although RoBERTa also shows some improvement. We also investigate a novel implementation of a cosine similarity loss function. This approach is inconclusive as we see significant performance improvement in RoBERTa over a cross-entropy trained model. However, there is very little improvement in CodeBERT performance using a CS loss function and we find a cross-entropy trained CodeBERT achieves an overall better performance. The largest performance gain was for a CodeBERT model trained with a cross-entropy loss function, resulting in a 13.01% improvement over the baseline.

1. Introduction

A critical task in software engineering is understanding and maintaining existing code. Software developers spend a significant amount of their time reading and modifying code. A crucial part of code comprehension is identifying the purpose and meaning of code variables. A variable name should be meaningful and indicative of its purpose and use. Naming variables can be a time-consuming task, and developers may struggle to come up with appropriate names that are both descriptive and concise. Predicting code variable names is a challenging problem in natural language processing. Accurately predicting variable names can aid in code comprehension, maintenance, and reuse, essential tasks for software developers. Recent advances in language modelling, such as the (in)famous ChatGPT, have shown great potential in various natural language processing tasks, including code generation and program synthesis. Powerful large language models (LLMs) have already been used in a commercial programming product, GitHub co-pilot, which uses a version of GPT-3 (Chen et al., 2021).

In this research, we train two models more capable of code completion for variable names than a pre-determined baseline. Essentially the models make predictions for the masked variable names in code snippets and thus complete the code. The variable names are completed using probabilistic models initially developed for natural language tasks. There has been a proliferation of open-source pre-trained transformer models on platforms such as huggingface[1] and we source two pre-trained transformers from this platorm.

Research in the exact domain of variable name generation by language models is surprisingly limited and only makes use of a single architecture, the LSTM variant of recurrent neural networks (Bavishi et al., 2018). Our approach is novel to this previous work by Bavishi *et al.* as we implement a different architecture, the transformer (Vaswani et al., 2017). The transformer architecture has subsequently become state-of-the-art with the majority of NLP research in recent years utilising this architecture (Svyatkovskiy et al., 2020; Dai et al., 2019; Slovikovskaya, 2019; Tunstall et al., 2022; Wolf et al., 2020; Kalyan et al., 2021; Wolf et al., 2019). We use two transformers that are based on the exact architecture in Vaswani *et al.*, these are two BERT-based (Devlin et al., 2018) pre-trained transformers, RoBERTa[2] (Liu et al., 2019) and CodeBERT[3] (Feng et al., 2020).

The availability of these models allows us to use state-of-the-art architectures to complete our task. This will be done by fine-tuning the pre-trained transformers on a corpus of code snippets focusing on generating variable names. Our research aims are:

1. Fine-tune pre-trained transformers to improve variable name generation over the baseline transformers.
2. Does prior training on code yield a significant advantage over a code naive pre-trained transformer even after our training?
3. What is the impact on variable name suggestion of using cosine similarity as the loss function during training instead of the more standard cross-entropy loss function?

In the rest of this report, we will first in section 2 investigate related works and how our work fits into broader research in NLP for code. In section 3, we will report the details of our dataset and outline how we have pre-processed and managed the dataset. In section 4, we will describe, explain

[1] https://huggingface.co/models
[2] https://huggingface.co/roberta-base
[3] https://huggingface.co/microsoft/codebert-base-mlm

and justify our research methodology. The results, interpretation and discussion will be contained within section 5 and finally, in section 6 our summary, conclusions and possible avenues for future work.

2. Related work

Language models are a statistical approach to natural language that works because natural language uttered by actual people is generally simple and repetitive, designed for communication at speed and in noisy environments. Hindle *et al.* in their 2012 work argues that actual code written by people is very similar to natural language; it is generally quite simple and repetitive to facilitate communication between programmers (Hindle et al., 2012). Before this work, research in understanding code languages was the domain of a formal, grammar-based approach that attempted to deal with code from first principles. This is understandable as code, of course, is an artificial language designed to be mathematically well-formed.

If code possesses similar simplicity and repetition as a natural language does in real utterances. It allows sophisticated, already-developed natural language processing techniques to be applied to code language. This is what has been termed the natualness of code (Hindle et al., 2012). Hindle *et al.*'s claim of code naturalness is strongly supported by the earlier work of Gabel and Su that found a large degree of repetition in a 430 million line code corpus at a 1-7 line granularity (Gabel & Su, 2010). If code is natural, then NLP can be used for code completion, i.e., predicting the parts of the code snippets supplied to the model that are unfinished/masked.

Further support for the naturalness of code came from the success of n-gram models, simple statistical models that assume a Markov property that uses the $n-1$ tokens to predict the nth token (Hindle et al., 2012; Nguyen et al., 2013; Hellendoorn & Devanbu, 2017; Allamanis et al., 2019). The nth token is predicted from simple maximum-likelihood estimation based on the frequency of tokens in the training corpus. Simply increasing the training corpora for n-gram models yielded significant performance gains (Allamanis & Sutton, 2013; Allamanis et al., 2014) with these models even being used to patch open source projects (Allamanis et al., 2014). The success of n-grams showed that NLP techniques could be legitimately applied in the code language case, especially as they are relatively simple architectures only capable of learning local context.

Over the last decade, increasingly advanced language model architectures have been borrowed from NLP and applied to code. Such as recurrent neural networks (RNNs) (Karpathy et al., 2015; White et al., 2015), Long Short-Term Memory (LSTM) networks (Dam et al., 2016; 2018) and of course transformers (Vaswani et al., 2017; Svyatkovskiy et al., 2020; Brown et al., 2020; Xu et al., 2022). The ability of these architectures to understand and capture context increases in this sequence, as does their power for code generation tasks.

The ability of these models to generate code has increased massively since the move away from more classical methods to contemporary machine learning models (Allamanis et al., 2019; White et al., 2015). Vanilla recurrent neural networks (RNNs) outperform the simpler n-gram language models that were already quite successful (Hindle et al., 2012; White et al., 2015; Santos et al., 2018). RNNs gain this performance by being a deep neural network so significantly more expressive than n-gram models and also by processing inputs sequentially whilst incorporating a hidden state vector that copies information back to previous layers (White et al., 2015). The performance gain between LSTM networks and RNNs found in 2016 was a further improvement of 37.9% (Dam et al., 2016) in one common metric. LSTM networks are a variant of RNNs, designed specifically to mitigate the vanishing gradient problem that is highly prevalent and hard to manage in vanilla RNN architectures (Bengio et al., 1994). LSTMs additionally utilise a memory state vector that allows gradients on memory cells to flow back through time unless interrupted by an active forget gate (Karpathy et al., 2015; Choetkiertikul et al., 2018). This allows the LSTM to partially forget information that has lost its relevance.

Previous research into variable name generation using language models as mentioned is quite limited and has notably used the LSTM network architecture (Bavishi et al., 2018). Bavishi *et al.* developed a neural network comprising of three LSTM networks to generate variable names from code. This was done by first summarising the code, then learning the embeddings for the usage summaries with an auto-encoder, and finally generating an output (Bavishi et al., 2018).

The transformer architecture seen in figure 1 is an encoder-decoder architecture in which the encoding block encodes the contextual information in the input of how its different parts relate (Vaswani et al., 2017). This is achieved through an attention block of self-attention layers and a fully-connected feed-forward neural network. The different attention heads in the attention block encode different contextual aspects of the input; this allows the transformer to learn structural dependencies of different scales in the input, the decoder then learns from the encoded input. Transformers process the inputs in their entirety, a fundamentally different method than the sequential LSTM networks that have been previously used in the variable naming task. Both short and long-range context is processed in parallel, as written code generally does not function in a perfect left-to-right manner, e.g. nested loops. This has an intuitive advantage over previous language model architectures. Any metric use to evaluate transformers has found the models to be far more powerful in this domain than any previous architecture (Chen et al., 2021; Xu et al., 2022; Svyatkovskiy et al., 2020). The largest, most powerful and, therefore, extremely expensive transformers are generally trained on an 800GB corpus known as the Pile (Gao et al., 2020), or a corpus of comparative size. The largest of these transformers possess parameters on the order of hundreds of billions

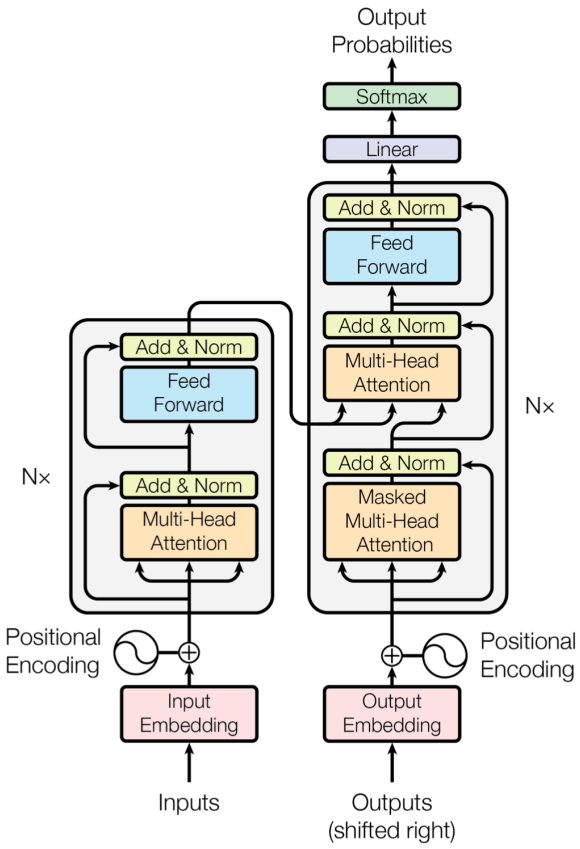

Figure 1. Original transformer architecture that is implemented in RoBERTa and CodeBERT (Vaswani et al., 2017; Liu et al., 2019; Feng et al., 2020). Figure from (Vaswani et al., 2017).

(Brown et al., 2020; Chen et al., 2021; Black et al., 2021; Xu et al., 2022; Poesia et al., 2022; Black et al., 2022).

The development and availability of transformers that have been pre-trained on large text corpora has spurred research in transfer learning. This technique involves first training a model on a large and rich dataset before being fine-tuned for a downstream task (Slovikovskaya, 2019; Raffel et al., 2020; Li et al., 2021; Xue et al., 2021). Specifically, fine-tuning transformer models only on a subset of their hidden layers, the final n layers, has proved to be a particularly robust method (Raffel et al., 2020). This is especially the case when data for the downstream task or computational power is limited. This allows the representations learnt from the pre-training to be leveraged on a new task by freezing the earlier network layers (Raffel et al., 2020). The pre-training corpora of the two transformers used in this research is therefore relevant to this research. RoBERTa was trained using 161GB of different text data (Liu et al., 2019; Zhu et al., 2015; Gokaslan & Cohen, 2019; Radford et al., 2019; Trinh & Le, 2018). CodeBERT is trained using the full 20GB CodeSearchNet Challenge dataset (Husain et al., 2019). RoBERTa is simply a version of BERT trained systematically to produce a robust model with carefully evaluated hyperparameter values (Liu et al., 2019). RoBERTa additionally builds upon the original BERT by training the model with a greater variety of and more data (Li et al., 2021). CodeBERT is another BERT-based transformer trained using a multilingual training approach (Pires et al., 2019) on both natural and code language. Both of these models use what is effectively the original transformer architecture (Vaswani et al., 2017; Liu et al., 2019; Feng et al., 2020).

3. Data set and task

Our training corpus required a sufficient volume of high-quality python code snippets. To assure the quality, we selected a well-known data set, this is the codesearchnet[4] corpus (Husain et al., 2019). This data set has been developed and collected by researchers from Microsoft and GitHub, is has been used widely and cited on Google Scholar alone the research has been cited 400 times. This is by a range of other papers for many code and NLP applications (Feng et al., 2020; Hu et al., 2020; Koh et al., 2021; Wang et al., 2021; Guo et al., 2020; Neelakantan et al., 2022).

The corpus contains over 6 million code snippets in 6 languages; we only required python code snippets for our fine-tuning task, so we have removed the additional 5 languages. The corpus originally has 12 columns and comes in a json format; we select only the code and repository information columns, we split this data set into training, validation and test sets and save these using pickle. The pickle files can then be loaded into python scripts as pandas data frames for use with our training and evaluation scripts.

The python corpus is then used to develop a fine-tuned masked language model to suggest variable names in code. By doing this, we aim to help novice programmers create intuitive names to aid code readability and development. We also recommend that such an approach could be used to aid programmers in new teams to comply with institution code conventions to help teams integrate faster. An auxiliary aspect of this task is to test the constraints of the necessary resources to accomplish it and to find to what degree the pre-trained language models are few-shot learners (Brown et al., 2020). This is important as we will develop our models with significantly less computing power than other significant developments such as CodeBERT by Microsoft (Feng et al., 2020).

Our models' success in their downstream task of variable name suggestion will be evaluated in three ways. Firstly, we shall assess qualitatively several suggestions made by both the original and fine-tuned models. We will make subjective judgements on the performance of the baseline and the fine-tuned models; this will include whether or not the suggestions make sense, even if they are somewhat dissimilar to the actual variable name in strict token matching, to see if the semantic meaning is captured.

We will also use perplexity as a quantitative evaluation metric, as this is a prevalent metric in NLP and has been widely

[4] https://github.com/github/CodeSearchNet

used in papers relating to language models for code (Hindle et al., 2012; Nguyen et al., 2013; Hellendoorn & Devanbu, 2017; White et al., 2015; Dam et al., 2016). Perplexity is 2^H where H is the entropy of the prediction made by the model (Ranjan et al., 2016). As this model is derived from entropy, using cross-entropy as the loss function in training will also train to reduce perplexity.

Additionally, we will use cosine similarity to evaluate our models; this method finds the cosine of the angle between two vectors. Our cosine similarity is bounded between 0 and 1, with the vectors being identical at 1 and completely different at 0 (Gomaa et al., 2013; Lahitani et al., 2016). To create the vectors on which to calculate the cosine similarity, we use a sentence transformer[5] to encode the true variable name and the suggested variable name (Wang et al., 2020b). We selected all-MiniLM-v2 to encode our variables for cosine similarity calculation as it is a widely used (over 2 million downloads in February 2023 alone) sentence transformer and has been specifically developed considering BERT-based models, as these models very closely follow the original transformer architecture (Wang et al., 2020b; Vaswani et al., 2017).

4. Methodology

This section will outline our methodology and our decisions in determining it[6]. We chose to use BERT-based models for several reasons; the BERT architecture is widely used and researched and nearly identical to the original transformer architecture (Liu et al., 2019; Vaswani et al., 2017). Whilst BERT models are large, these models are not too large and are still trainable on the computing resources available to us. Additionally, both models contain $\sim$ 125M parameters; they are easily comparable. A vital aspect of these models that made them especially attractive is that they are already capable of making predictions of mask tokens due to the beneficial huggingface pipeline for inference. This allows our models to be easily tested on their variable suggestion capabilities. Additionally, the tokenizer used by the RoBERTa and CodeBERT models is a BPE (byte pair encoding) tokenizer. The tokenizer was also trained during the development of the base RoBERTa model; it works by iteratively merging the most frequent pairs of adjacent characters or character sequences in the input text until a desired vocabulary size is reached (Sennrich et al., 2015; Wang et al., 2020a). RoBERTa's tokenizer is based on the tokenizer used by GPT-2 (Radford et al., 2019; Liu et al., 2019) and is also used by CodeBERT (Feng et al., 2020). The fact that the BPE tokenizer merges frequent pairs means it is somewhat more flexible than word tokenizers, whilst being more capable of learning structure than character-level tokenizers (Sennrich et al., 2015). Flexibility and capability are what is needed for code generation. Evidence for this comes from the same tokenizer already being used by CodeBERT for code synthesis (Feng et al., 2020). Indicating that it is a reasonable choice of tokenizer for code and natural language.

Metric	Value
Precision	0.90
Recall	0.82
F-measure	0.86

Table 1. Three metrics to evaluate the masking script used to mask variable names probabilistically.

We perform the fine-tuning through the transfer learning process (Raffel et al., 2020); we freeze the gradients in all but the final layer of the model so that only the weights in the final layer undergo training. This will leverage the pre-training on the more general but far larger natural language corpora and pass that through the final fine-tuned layer for the downstream task. This makes our models realistic to train with our computing resources. As they are both widely used language models that have been studied in depth, the hyperparameters should need very little, if any, fine-tuning as they have been robustly determined for these BERT-based models. This allows us to use the recommended hyperparameters for training with cross-entropy loss. We do investigate the best learning rate for the models, but we shall start the investigation at the recommended learning rate of 5×10^{-5}. We train the models using the Adam optimiser (Kingma & Ba, 2014).

To be used as inputs for training our models, the training, valid, and test corpora are first sampled; we use different proportions of this data due to computing resource limitations; for example, we perform a greedy hyperparameter tuning using only 1% of the training corpus so that it is feasible to try several different hyperparameters. We also used a subset of the data to determine our early stopping epoch. The data is randomly sampled, and then a percentage of the variable names are masked using a masking function. The masking function uses regular expression patterns to match variable assignments using common variable identifiers[7]. We tested the accuracy of this function by manually masking variable names in 20 code snippets and comparing this to the result of our masking function on these 20 snippets. The masking function's precision, recall and F-measure are given in table 1. A fraction of variable names are left unmasked so that there are examples of variable names to learn from and infer what the tokens under the masks are. For each masked variable, a window of the code snippet is created; this leaves a fixed number of characters on either side of the variable. This is to provide the model with the contextual information of the code snippet of which the variable is part; the model should learn this context to predict the variable name under the mask. Throughout our experiments, we keep the window to

[5] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

[6] The code used in this research is available at: https://github.com/s1974565/MLP_Final

[7] See the mask_variable_names() function in testing.py to find the exact patterns used for matching.

Model	Top K	Cosine Similarity	Perplexity
RoBERTa	1	0.413	10.802
	2	0.370	-
	3	0.337	-
CodeBERT	1	0.472	10.780
	2	0.429	-
	3	0.399	-

Table 2. Baseline cosine similarity scores for Top K = {1,2,3} predictions made using 10% of the validation data. As cosine similarity is always largest for the top K = 1 prediction we only train for the top suggestion going forward.

100 characters on either side of the mask tokens; however, ultimately, this is a hyperparameter in and of itself that, with sufficient time and computing resources, would also be optimised. We keep the window at 100 characters as this seems the smallest reasonably sized window and we must be careful to not make memory requirements too high. After the variables are masked and the corpus is broken down into windows, the code is tokenized and converted into PyTorch tensors for optimized training with CUDA on a GPU. It should be noted that we always train on a GPU but due to the computational cost of our training we perform the greedy hyperparameter searching on a personal GPU, an Nvidia GeForce GTX 1650. However, when we train on the largest proportion of the training corpus of 20% (approximately 86,000 code snippets and 220,000 masked variable windows), we use a Google cloud virtual machine with an Nvidia Tesla V100; this allows one epoch to take approximately 1.5 hours. When sampling the training, validation and test corpora we set the seed number to 42 for reproducible results.

As the pipeline for inference is capable of making multiple 'top k' predictions, we tested the optimal number of predictions to optimise the values given by the two evaluation metrics when tested on RoBERTa and CodeBERT before fine-tuning. Table 2 shows that evaluating only the top 1 prediction produces the baseline models' highest cosine similarity value and lowest perplexity value. In our model training, we only optimise and evaluate for the top prediction by the model.

To assess our research aims, we use a combination of qualitative and quantitative evaluation. Perplexity and cosine similarity will be used as quantitative and intrinsic metrics to measure the performance of the models. Perplexity is a standard metric in NLP and NLP for code specifically and is cross-entropy raised to the power of two (Allamanis et al., 2019). As the loss function will be cross-entropy, we will effectively train our models to minimise perplexity, making it a natural metric. The cosine similarity will measure the similarity of the model's variable name suggestion and the true name, and this is done by calculating the angle between the two token vectors of the suggestion and the true variable name (Reimers & Gurevych, 2019). We will also manually evaluate several predictions made by the models for variable names and compare them to the true variable names. The code's function will also be considered to see if the suggestions are sensible. Human judgement will then be used to see whether the fine-tuned models perform better than the baseline.

We perform two types of training with two different loss functions, which are also our aforementioned evaluation metrics; cross-entropy and cosine similarity. Cross-entropy is a common loss function and is directly related to the perplexity metric widely used in NLP. As the model produces results in logits, bounded $(-\infty, +\infty)$, whereas the one-hot true labels are binary, the softmax function is applied to the logits before the cross-entropy is calculated. We also use a novel loss function, the cosine similarity, this approach is novel due to the need for creating word embeddings for which a dot product can be taken to calculate cosine similarity. This is done with the popular and BERT-optimized all-MiniLM-L6-v2 sentence transformer. When determining the word embeddings, we perform mean pooling to take the attention mask in the transformer into account for correct averaging.

We compare the two training methods using the cosine similarity evaluation on our held-out test corpus. Alongside this, we manually compare the predictions made by the differently trained models for a number of different variable suggestions to human evaluate the difference in suggestions by the different fine-tuned models.

In the next section, we present the results of our experiments, interpret these results and discuss the limitations of our research.

5. Results & Interpretation

In this section, we present the results of our experiments; we begin with our greedy hyperparameter tuning. A range of hyperparameters were systematically tested for both RoBERTa and CodeBERT and with both cross-entropy and cosine similarity as the loss functions, the final values that we used are contained in table 3.

Figure 2 shows the training curves for a RoBERTa model being fine-tuned on 2.5% of the training corpus. We observe that a large degree of overfitting as the model is visibly learning to fit to the training data with a classicly shaped training curve. Due to decreasing improvement in validation loss we enact early stopping at the fifth epoch.

Our models were then trained on a larger proportion of the data, the corpora size was increased to 10% for both models and due to the high computing cost only RoBERTa was trained on a 20% portion of data. Figure 3 displays the training curves for the 10% RoBERTa and figure 4 shows the curves for the 20% RoBERTa. We observe that in both figures 3 and 4, the cross-entropy starts significantly smaller in the first epoch than when 2.5% of the data is used in figure 2. This indicates that the more data that is provided the more easily the model can train to minimise the loss function. However, this does not neatly translate

Model	Loss Func	LR	ES Epoch
RoBERTa	CE	5e-5	5
RoBERTa	CS	1e-5	3
CodeBERT	CE	1e-5	5
CodeBERT	CS	1e-5	3

Table 3. Determined learning rates and early stopping epochs for both transformers and loss functions. Batch size remains constant at 32, the L1 weight decay coefficient at the recommended 0.01 (Liu et al., 2019) and the mask probability at 0.5.

Model	CS (base)	CS (f-t)	ΔPP (%)
RoBERTa CE (10%)	0.427	0.430	-9.56
RoBERTa CS (10%)	0.427	0.438	-10.56
RoBERTa CE (20%)	0.423	0.434	-10.50
CodeBERT CE (10%)	0.495	0.511	-13.01
CodeBERT CS (10%)	0.495	0.511	-1.47

Table 4. Average cosine similarity scores produced by the baseline (CS (BASE)), cross-entropy (CE) and cosine similarity (CS) fine-tuned (CS (F-T)) models. Additionally, we have the percentage change in average perplexity (ΔPP) of the fine-tuned model compared to the baseline perplexity. These values are calculated on 10% of the held-out test data set for the 10% models and 20% for the 20% model.

to the validation loss, which, especially for 10% of the data, remains at similar values to the first five epochs for the 2.5% training data RoBERTa training. For the 20% data training, we observe that training and validation curves start much lower than both previous models. The training loss curve is also much steeper than the in figures 2 and 3; the model is learning much faster. Again the validation loss curve is quite shallow, showing that the training is not generalising well. However, we observe the sharpest decrease in validation loss of the three RoBERTa cross-entropy figures between the fourth and fifth epochs, it is possible that this is an aberration; it is also possible that as significantly more training data is used, the model may need more training. This research's critical limitation is that powerful and expensive computing resources are required to train these models, limiting our ability to train the models for many epochs with more data.

We also present the numerical comparison between the baseline and fine-tuned models on the same proportion of held-out test data in table 4. All the trained models show some improvement over the baseline cosine similarities, indicating that the approach is fundamentally working. We observe the greatest RoBERTa improvement in the model fine-tuned with a cosine similarity loss function on 10% of the data and the RoBERTa-based model trained using cross-entropy on 20% of the data. The RoBERTa model that uses the CS loss functions also performs better in the percentage reduction in perplexity; this supports our use of the novel loss function as it successfully indirectly minimises a standard NLP metric. However, it is important to contrast the numerical result with what is shown in figure 5. Here, we have the training curves for CodeBERT using the CS loss function. As cosine similarity is a value that is maximum at 1, the training cosine similarity should increase during training, we do not see that, and it remains flat. This results in an extremely modest reduction in perplexity and the same cosine similarity increase as for the CodeBERT model trained using the cross-entropy loss function. Ultimately we observe the greatest increase in cosine similarity by both of the CodeBERT models, this is sensible as CodeBERT is not code naive like RoBERTa and has more quickly optimised to the downstream code task.

As suspected, CodeBERT-based models outperform the originally code-naive RoBERTa-based models as the baseline CodeBERT outperforms even our best fine-tuned RoBERTa model. CodeBERT is evidently more capable of leveraging its specialist code training for more successful transfer learning to our specific task.

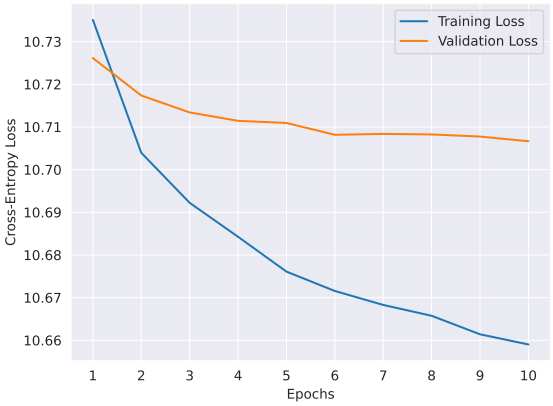

Figure 2. Cross-entropy loss plot for RoBERTa trained on 2.5% of the python codesearchnet corpus for 10 epochs. We can observe a large gap between the training and validation loss. This increases up to the 10th epoch indicating overfitting to the training sample.

We do note that these results show only minimal increases in the similarity between the predicted variable names and the true variable names.

We will now compare some of the actual differences in variable name suggestions by the models and see whether their ability to determine clear and concise variable names correlates well with optimising their loss functions. Table 5 contains some examples where the fine-tuned models perform better than the baseline models. In these examples, the fine-tuned models are better at getting close to the true label than the baseline. They are often only a character out contrasting with the baseline, which suggests very different tokens. Some suggestions by the fine-tuned model also clearly capture the semantic information better than the baseline models. For example, the CS-trained CodeBERT

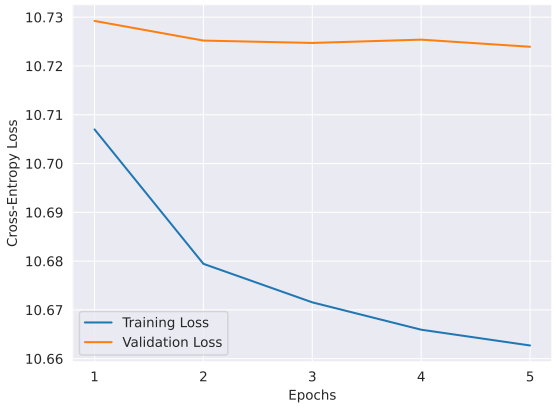

Figure 3. Cross-entropy training curve for 10% corpus trained RoBERTa. The model is clearly fitting to the data but the validation loss improves only minimally.

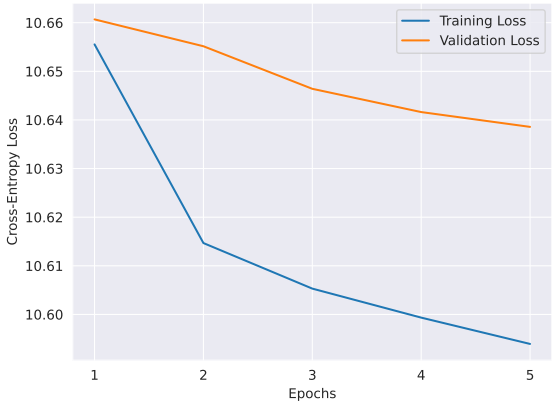

Figure 4. Cross-entropy training curve for CodeBERT trained on 10% of the python corpus. The gradients of the curves are steeper than for previous plots.

suggests 'client' in the space of 'http' which is a variable for a string that is a URL. The variable name 'client' is a much better suggestion than a hashtag which is an actively problematic choice as it is the commenting character.

One limitation of our fine-tuning method is that no model at any time correctly suggests model names with underscores. This is a significant issue as variable names frequently contain underscores. This is likely due to both RoBERTa and CodeBERT using the RoBERTa pre-trained tokenizer. To improve the results of any future model, we may need to train a specialist BPE tokenizer. A new tokenizer would need to be capable of utilising underscores to improve the model's variable name suggestions.

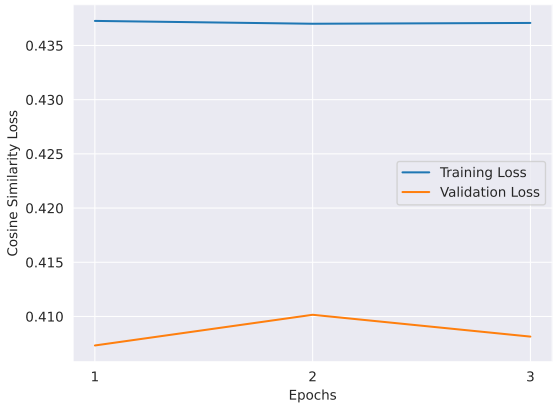

Figure 5. CodeBERT training curves using the cosine similarity loss function. The training curve is effectively flat, indicating limited-to-no training occurring.

MODEL	TRUE	BASE VAR.	F-T VAR.
RoBERTa CE (10%)	DIC	V	DICT
RoBERTa CS (10%)	ETAG	KEY	TAG
RoBERTa CE (20%)	X_POS	SF	AXIS
CodeBERT CE (10%)	POST_ARGS	BODY	PARAMS
CodeBERT CS (10%)	HTTP	#	CLIENT

Table 5. Example variable name suggestions by the models compared to the true variable names. Improvement is evaluated by manually looking at the code snippet.

6. Summary & Conclusions

In this research, we investigated transfer learning with two pre-trained transformer models for the downstream task of variable name suggestion.We had three primary aims: 1) to improve variable name suggestion, 2) to investigate the effect of prior training on code, and 3) to determine if cosine similarity is an effective loss function. We were successfully able to show that by freezing the gradients in all but the final layer of the transformers, we were able to cause some optimisation of perplexity and cosine similarity. We also manually showed that the fine-tuned models produce more sensible variable names than the baseline models. Secondly, we did observe that CodeBERT was a more successful transfer learner for the variable name suggestion. This was unsurprising due to its prior training on the same natural language data as RoBERTa and 20GB of code snippets. Finally, we tested and compared our novel loss function implementation and compared it to the more common perplexity approach. Compared to training to optimise perplexity, we found that the effect of cosine similarity loss is somewhat inconclusive. RoBERTa trained with CS loss and 10% of the data performs essentially on par with RoBERTa trained on 20% of the corpus using cross-entropy loss. This result contrasts with the modest decrease in per-

plexity experienced by CodeBERT when trained using CS loss.

Overall we have successfully investigated our research aims, the effects of cosine similarity loss do, however warrant further investigation.

6.1. Future Research

The first and most frequently mentioned route of future research is to increase the training corpus size. Training on more code snippets will allow the models greater opportunity to learn the complex structural dependencies and more frequent formulations of variable names in code, which possesses more flexibility in its tokens than natural language. Doing this in conjunction with the use of the cosine similarity loss function is the most natural progression of this research.

Finally, we would like to investigate transfer learning further using different fine-tuning methods. We only fine-tuned the final layer of the two transformers, which is only one transfer learning approach. Another standard method is to add an extra layer on top of the pre-trained transformer and train this layer (Raffel et al., 2020). This is likely a better approach than the higher computational resources required to train more layers of the transformers.

References

Allamanis, Miltiadis and Sutton, Charles. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*, pp. 207–216. IEEE, 2013.

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

Allamanis, Miltiadis, Barr, Earl T., Devanbu, Premkumar, and Sutton, Charles. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51 (4):1–37, 2019. ISSN 0360-0300. doi: 10.1145/3212695. URL https://dx.doi.org/10.1145/3212695.

Bavishi, Rohan, Pradel, Michael, and Sen, Koushik. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*, 2018.

Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2): 157–166, 1994.

Black, Sid, Gao, Leo, Wang, Phil, Leahy, Connor, and Biderman, Stella. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. URL https://doi.org/10.5281/zenodo.5297715.

Black, Sid, Biderman, Stella, Hallahan, Eric, Anthony, Quentin, Gao, Leo, Golding, Laurence, He, Horace, Leahy, Connor, McDonell, Kyle, Phang, Jason, Pieler, Michael, Prashanth, USVSN Sai, Purohit, Shivanshu, Reynolds, Laria, Tow, Jonathan, Wang, Ben, and Weinbach, Samuel. GPT-NeoX-20B: An open-source autoregressive language model. In *Proceedings of the ACL Workshop on Challenges & Perspectives in Creating Large Language Models*, 2022. URL https://arxiv.org/abs/2204.06745.

Brown, Tom, Mann, Benjamin, Ryder, Nick, Subbiah, Melanie, Kaplan, Jared D, Dhariwal, Prafulla, Neelakantan, Arvind, Shyam, Pranav, Sastry, Girish, Askell, Amanda, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chen, Mark, Tworek, Jerry, Jun, Heewoo, Yuan, Qiming, Pinto, Henrique Ponde de Oliveira, Kaplan, Jared, Edwards, Harri, Burda, Yuri, Joseph, Nicholas, Brockman, Greg, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Choetkiertikul, Morakot, Dam, Hoa Khanh, Tran, Truyen, Pham, Trang, Ghose, Aditya, and Menzies, Tim. A deep learning model for estimating story points. *IEEE Transactions on Software Engineering*, 45(7):637–656, 2018.

Dai, Zihang, Yang, Zhilin, Yang, Yiming, Carbonell, Jaime, Le, Quoc, and Salakhutdinov, Ruslan. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 2978–2988, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1285. URL https://aclanthology.org/P19-1285.

Dam, Hoa Khanh, Tran, Truyen, Grundy, John, and Ghose, Aditya. Deepsoft: A vision for a deep model of software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 944–947, 2016.

Dam, Hoa Khanh, Tran, Truyen, Pham, Trang, Ng, Shien Wee, Grundy, John, and Ghose, Aditya. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 47(1):67–85, 2018.

Devlin, Jacob, Chang, Ming-Wei, Lee, Kenton, and Toutanova, Kristina. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Feng, Zhangyin, Guo, Daya, Tang, Duyu, Duan, Nan, Feng, Xiaocheng, Gong, Ming, Shou, Linjun, Qin, Bing, Liu, Ting, Jiang, Daxin, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Gabel, Mark and Su, Zhendong. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 147–156, 2010.

Gao, Leo, Biderman, Stella, Black, Sid, Golding, Laurence, Hoppe, Travis, Foster, Charles, Phang, Jason, He, Horace, Thite, Anish, Nabeshima, Noa, Presser, Shawn, and Leahy, Connor. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Gokaslan, Aaron and Cohen, Vanya. Openwebtext corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

Gomaa, Wael H, Fahmy, Aly A, et al. A survey of text similarity approaches. *international journal of Computer Applications*, 68(13):13–18, 2013.

Guo, Daya, Ren, Shuo, Lu, Shuai, Feng, Zhangyin, Tang, Duyu, Liu, Shujie, Zhou, Long, Duan, Nan, Svyatkovskiy, Alexey, Fu, Shengyu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Hellendoorn, Vincent J and Devanbu, Premkumar. Are deep neural networks the best choice for modelling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773, 2017.

Hindle, Abram, Barr, Earl T., Su, Zhendong, Gabel, Mark, and Devanbu, Premkumar. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 837–847. IEEE Press, 2012. ISBN 9781467310673.

Hu, Weihua, Fey, Matthias, Zitnik, Marinka, Dong, Yuxiao, Ren, Hongyu, Liu, Bowen, Catasta, Michele, and Leskovec, Jure. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.

Husain, Hamel, Wu, Ho-Hsiang, Gazit, Tiferet, Allamanis, Miltiadis, and Brockschmidt, Marc. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Kalyan, Katikapalli Subramanyam, Rajasekharan, Ajit, and Sangeetha, Sivanesan. Ammus: A survey of transformer-based pretrained models in natural language processing. *arXiv preprint arXiv:2108.05542*, 2021.

Karpathy, Andrej, Johnson, Justin, and Fei-Fei, Li. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

Kingma, Diederik P and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Koh, Pang Wei, Sagawa, Shiori, Marklund, Henrik, Xie, Sang Michael, Zhang, Marvin, Balsubramani, Akshay, Hu, Weihua, Yasunaga, Michihiro, Phillips, Richard Lanas, Gao, Irena, et al. Wilds: A benchmark of in-the-wild distribution shifts. In *International Conference on Machine Learning*, pp. 5637–5664. PMLR, 2021.

Lahitani, Alfirna Rizqi, Permanasari, Adhistya Erna, and Setiawan, Noor Akhmad. Cosine similarity to determine similarity measure: Study case in online essay assessment. In *2016 4th International Conference on Cyber and IT Service Management*, pp. 1–6. IEEE, 2016.

Li, Yanghao, Xie, Saining, Chen, Xinlei, Dollar, Piotr, He, Kaiming, and Girshick, Ross. Benchmarking detection transfer learning with vision transformers. *arXiv preprint arXiv:2111.11429*, 2021.

Liu, Yinhan, Ott, Myle, Goyal, Naman, Du, Jingfei, Joshi, Mandar, Chen, Danqi, Levy, Omer, Lewis, Mike, Zettlemoyer, Luke, and Stoyanov, Veselin. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Neelakantan, Arvind, Xu, Tao, Puri, Raul, Radford, Alec, Han, Jesse Michael, Tworek, Jerry, Yuan, Qiming, Tezak, Nikolas, Kim, Jong Wook, Hallacy, Chris, et al. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*, 2022.

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

Pires, Telmo, Schlinger, Eva, and Garrette, Dan. How multilingual is multilingual bert? *arXiv preprint arXiv:1906.01502*, 2019.

Poesia, Gabriel, Polozov, Oleksandr, Le, Vu, Tiwari, Ashish, Soares, Gustavo, Meek, Christopher, and Gulwani, Sumit. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*, 2022.

Radford, Alec, Wu, Jeffrey, Child, Rewon, Luan, David, Amodei, Dario, Sutskever, Ilya, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8): 9, 2019.

Raffel, Colin, Shazeer, Noam, Roberts, Adam, Lee, Katherine, Narang, Sharan, Matena, Michael, Zhou, Yanqi, Li, Wei, and Liu, Peter J. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

Ranjan, Nihar, Mundada, Kaushal, Phaltane, Kunal, and Ahmad, Saim. A survey on techniques in nlp. *International Journal of Computer Applications*, 134(8):6–9, 2016.

Reimers, Nils and Gurevych, Iryna. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL http://arxiv.org/abs/1908.10084.

Santos, Eddie Antonio, Campbell, Joshua Charles, Patel, Dhvani, Hindle, Abram, and Amaral, José Nelson. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 311–322. IEEE, 2018.

Sennrich, Rico, Haddow, Barry, and Birch, Alexandra. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Slovikovskaya, Valeriya. Transfer learning from transformers to fake news challenge stance detection (fnc-1) task. *arXiv preprint arXiv:1910.14353*, 2019.

Svyatkovskiy, Alexey, Deng, Shao Kun, Fu, Shengyu, and Sundaresan, Neel. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443, 2020.

Trinh, Trieu H and Le, Quoc V. A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847*, 2018.

Tunstall, Lewis, Von Werra, Leandro, and Wolf, Thomas. *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N, Kaiser, Łukasz, and Polosukhin, Illia. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wang, Changhan, Cho, Kyunghyun, and Gu, Jiatao. Neural machine translation with byte-level subwords. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 9154–9160, 2020a.

Wang, Wenhui, Wei, Furu, Dong, Li, Bao, Hangbo, Yang, Nan, and Zhou, Ming. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in Neural Information Processing Systems*, 33:5776–5788, 2020b.

Wang, Yue, Wang, Weishi, Joty, Shafiq, and Hoi, Steven CH. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

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

Wolf, Thomas, Debut, Lysandre, Sanh, Victor, Chaumond, Julien, Delangue, Clement, Moi, Anthony, Cistac, Pierric, Rault, Tim, Louf, Rémi, Funtowicz, Morgan, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Wolf, Thomas, Debut, Lysandre, Sanh, Victor, Chaumond, Julien, Delangue, Clement, Moi, Anthony, Cistac, Pierric, Rault, Tim, Louf, Rémi, Funtowicz, Morgan, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.

Xu, Frank F, Alon, Uri, Neubig, Graham, and Hellendoorn, Vincent Josua. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

Xue, Fanglei, Wang, Qiangchang, and Guo, Guodong. Transfer: Learning relation-aware facial expression representations with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 3601–3610, 2021.

Zhu, Yukun, Kiros, Ryan, Zemel, Rich, Salakhutdinov, Ruslan, Urtasun, Raquel, Torralba, Antonio, and Fidler, Sanja. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pp. 19–27, 2015.