Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Canonical Identifier Naming on Code Search Models

Distributed Systems Lab Report

Tobias Stocker

tstocker@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Peter Belcák, Florian Grötschla
Prof. Dr. Roger Wattenhofer

August 29, 2023

Abstract

Focusing on the semantic information in code has been a key challenge for NLP based models when solving code related machine learning tasks such as code search. State of the art models still rely heavily on the syntactic information such as the naming of variables. This does not only make such models weak to variable naming attacks but also worsens the performance greatly when the target data is anonymized. In this work we explore an approach we call Canonical Identifier Naming where the model is trained on transformed data that does not contain any lexical variable information hoping to make it resilient against lexical attacks and achieve good performance on anonymized data. We test this approach on a number of baseline and state of the art models on a set of different programming languages.

Contents

Chapter 1

Introduction

Machine learning has enabled machines to do many tasks that in the past only humans could do. While they tend to be mostly simple tasks it has allowed us humans to become more efficient and focus on the more important tasks. In the software industry a lot of time and effort goes into reading and understanding existing code as well as writing code that has already been written many times before. Machine learning tasks such as code search, completion, explanation etc. aim to either help a developer to do such tasks more efficiently or take them over completely.

1.1 Code Search

In this work we will focus on the task of code search. In code search we input a search query describing a desired functionality and the model then has to pick a suitable option out of a set of code pieces. Solving this task requires the model to get an understanding of the relation between the structure of code as well as the structure of the search query.

With the popularity of natural language processing (NLP) models in the past years researchers have successfully adapted NLP models to do well on code related machine learning tasks. However, programming code has some key differences compared to natural languages. One key difference is source code can be written in many different ways while doing the exact thing. Further, one may choose the naming of variables however they want. As a result, a key to understanding the functionality of some piece is to understand the data flow, i.e. the semantic information, in code. Using NLP models this has always been been a challenge.

1.2 Issues With Current Models

In software it is very common to reuse certain identifiers for certain variables in the source code. Developers often use the variables i, j, k often as indexes to

name an example. The variable name normally suggests the use or role of the variable. So when training a NLP model to understand what a certain piece of source code is doing it is only natural that the model will start to retrieve code information based on the variable names. As a result we get a model that relies less on understanding the structure of the code.

This issue has been analyzed in much more detail in the Challenging the Lexical Focus of Code Search work by Frederik Markus [1]. There he found that removing or obfuscating lexical clues such as the variable identifiers reduced the performance of all tested models regardless of the data sets. This was underlined by the fact that removing the comments alone had a significant effect on the performance of the model.

In the past years researches have tackled this issue mainly by supplying the model with more semantic information such as data flow. One example is the GraphCodeBERT model [2] that makes use of data flow in the pretraining stage of the model.

1.3 Goal Of This Work

In this work we explore an approach where the model does not have any information about the name of a variable. To be more precise we rename all variable names to canonical identifiers. The goal is to force the model into learning the inherent structure of code instead of relying on variable names thus creating a model that is robust against lexical attacks and achieves similar performance on anonymized data. The questions we want to answer are: Does training a model with this approach make it outperform itself when trained with normal data because it better learns to understand the structure of code? Does this approach allow us to train models that work well on anonymized data, i.e. data that has their variable names removed? We will try to answer those questions using a number of baseline as well as state of the art models on a number of different programming languages.

Chapter 2

Canonical Identifier Naming

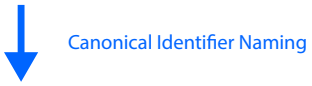
As mentioned before the goal of this work is to try to force the model into learning the inherent structure of code by removing any lexical clues. Our approach is to use normal data and transform it by replacing all variable identifiers with canonical identifiers as well as removing all comments. We further shuffle the identifiers randomly such that the order in which the variables appear is obfuscated as well. We call this approach Canonical Identifier Naming (CIN). A benefit of this approach is that we can also take any anonymized data, i.e. data that has the variable names changed, and turn it into `cin` data.

2.1 Implementation

The implementation is slightly different for every programming language we use but the main idea is the same for all of them.

1. **Remove any comments**: Comments can be identified and deleted easily using regular expressions.
2. **Search the variables**: This is the most difficult step which also differs the most between programming languages. For all languages except PHP the idea is to parse the piece of code into an abstract syntax tree which then shows you all variables. The library used to parse is specific to every language and normally written in this exact language. In PHP every variable starts with a $ character so we use this information to get the variables.
3. **Create the canonical identifiers and shuffle them**: Once we have collected the variable we create a canonical identifier for each different variable name. We choose to use the canonical identifiers `var0`, `var1` etc. The new identifiers are further shuffled randomly to remove any information regarding the order in which the variables appear in the code.
4. **Rename the variables**: Finally we have to replace the old variable names with the new canonical identifiers.

```python
def countByValueAndWindow(self, windowDuration, slideDuration, numPartitions=None):
    keyed = self.map(lambda x: (x, 1))
    counted = keyed.reduceByKeyAndWindow(operator.add, operator.sub,
                                         windowDuration, slideDuration, numPartitions)
    return counted.filter(lambda kv: kv[1] > 0)
```




```python
def var3(var7, var2, var9, var4=None):
    var8 = var7.map(lambda var6: (var6, 1))
    var1 = var8.reduceByKeyAndWindow(var0.add, var0.sub,
                                     var2, var9, var4)
    return var1.filter(lambda var5: var5[1] > 0)
```

Figure 2.1: Example of Canonical Identifier Naming on a Python function.

Figure 2.1 shows in a small example the differences between the original code and the transformed code. We see that both the name of the function as well as all variables are replaced with the canonical identifiers `var0`, `var1`, ..., `var9`. The new identifiers are further shuffled so for example the function identifier is now `var3`.

Chapter 3

Experiment Setup

A lot of this work is build on top of the infrastructure that was established in the work on Challenging the Lexical Focus of Code Search from Frederik Markus [1]. As a result we will use of the same models and data sets as well as the same implementation pipeline.

3.1 Experiment Pipeline

The implementation pipeline, i.e. a collection of various shell scripts, was designed to test different attacks on code search models. Its purpose is to automate the steps required to train and evaluate different combinations of modified data sets and machine learning model. To be more precise this includes the steps of generating a modified data set base using an attack, training a model on a data set, evaluating a model on a data set and recording the scores.

We will use this implementation pipeline as it is build modular by design and all we need to do is to add our transformation in the form of another attack to repurpose the pipeline. More on the implementation of the pipeline as well as information on how to use it can be found in detail in [1].

3.2 Data

We use a cleaned version of the CodeSearchNet corpus from the CodeSearchNet challenge [3]. The corpus is collected from publicly available open-source GitHub repositories sorted by their popularity. Using the pipeline certain elements from the original corpus are removed as they are not suitable for our models. We remove elements that...

- ... cannot be parsed into an abstract syntax tree.
- ... contain functions with very few (< 3) or very many (> 256) tokens.

- ... contain special characters.
- ... have a function description that is not written in English.

This leaves us with the following data sets:

Language	Train	Valid	Test
Python	251,820	13,914	14,918
PHP	241,241	12,982	14,014
Java	164,923	5,183	10,955
Javascript	58,025	3,885	3,291
Go	167,288	7,325	8,122

Table 3.1: The data sets and their sizes.

It is important to mention that by using this corpus containing scraped data sets it is unclear to which extend the documentation string accurately describes the function. Further as the documentation string is normally written by the same person that has also wrote the code it tends to be written using the same vocabulary and not necessary the vocabulary some would use in a code search query.

3.2.1 Modifying The Data

From each data set we create four modified versions that are then used to train the models. For the first version we simply remove all the code comments from the original data set. We call this version the `nocomment` data sets. For the other three versions we remove all the code comments and apply the canonical identifier naming transformation. For the `cin 1` data set we do not augment the code examples. For the `cin 2` and `cin 4` data sets we create two respectively four code examples for every original example each having their canonical identifier names reshuffled. Hence the `cin 2` data set is augmented by a factor of two and the `cin 4` data is augmented by a factor of four.

3.3 Models

We evaluate our approach on different NLP based models. All models were already implemented in the pipeline [1]. Hence, we will only briefly describe each of them. To get more implementation details refer to the Challenging the Lexical Focus of Code Search paper [1] or the model's original papers.

The general approach for all models is to compute an encoding for both the search query as well as the code snippets. Similar code snippets as well as their accompanying documentation should share similar encodings.

3.3.1 CodeSearchNet Models

As a baseline we use four different models from the CodeSearchNet paper [3].

- **Neural Bag of Words**: each (sub)token is simply embedded to a learnable embedding
- **1D Convoluational Neural Network**: the input sequence of tokens is passed through a 1 dimensional neural network
- **Self-Attention**: multi-head attention is used to compute representations of each token in the sequence
- **Convoluational Self-Attention**: a variant of the self-attention model employing convolutions

3.3.2 GraphCodeBERT

GraphCodeBERT [2] is a more sophisticated model that is based on the Transformer technology [4]. It uses data-flow, a sematic-level structure of code, instead of a syntactic structure like the abstract syntax tree in the pre-training stage. This makes the GraphCodeBERT model consider the inherent structure of the code which enhances the code understanding.

3.3.3 SynCoBERT

SynCoBERT [5] employs a combination of different ideas not found in other models to achieve state-of-the-art performance on the CodeSearchNet benchmark. What stands out is the usage of Cross Momentum Contrastive Learning (xMoCo) [6] which allows the separation of encoders for the documentation and code as well as providing some mechanisms to stabilize training. It further makes use of a Barlow loss.

3.4 Experiment Details

All experiments are conducted on the TIK Arton cluster at ETH Zurich. The models are all written in Python based primarily on the PyTorch [7] and TensorFlow [8] frameworks.

There is a policy employed on the TIK Arton cluster which forces jobs to have a maximum runtime of two days Having this restriction we were unable to collect the experiment results for the GraphCodeBert and SynCoBert models using the `cin 4` data sets as the training time exceeded the maximum runtime by a large

margin. We also have an issue with the GraphCodeBERT model in combination with the cin modified Go data sets that we were unable to resolve in time.

Chapter 4

Results

In this chapter we will glance over some of the key observations in the experiment results. All results can be found under appendix A.

4.1 CIN versus No Comment

In this section we want to analyze the performance difference between using data that contains the original variable identifiers and data that has this information removed, i.e. is using canonical identifiers. To be more precise we compare the results of the models trained and tested on the `no comment` data set, i.e. the original data with only the comments removed, to the models trained and tested on the `cin` data sets.

Generally we cannot see a clear pattern that emerges for each model or dataset. In fact we notice that the results are very different for each of the five programming languages we analyze as well as the models. As a result we will analyze patterns for both the models as well as the languages. We evaluate the languages in the three groups suggested in the prior work on Challenging the Lexical Focus of Code Search [1] which are:

1. Languages that are (nearly) averse to any modification (Go and PHP)
2. Languages that are not impacted by mild modifications but are affected by severe modifications (Java and Python)
3. Languages that are impacted even by small modifications (JavaScript)

Further we group the models according to the patterns that they exhibit.

Neural Bag of Words (NBow) Model

The NBoW model performs well on the Python and PHP data sets but fails dramatically for the other data sets. This applies for both the `no comment` and

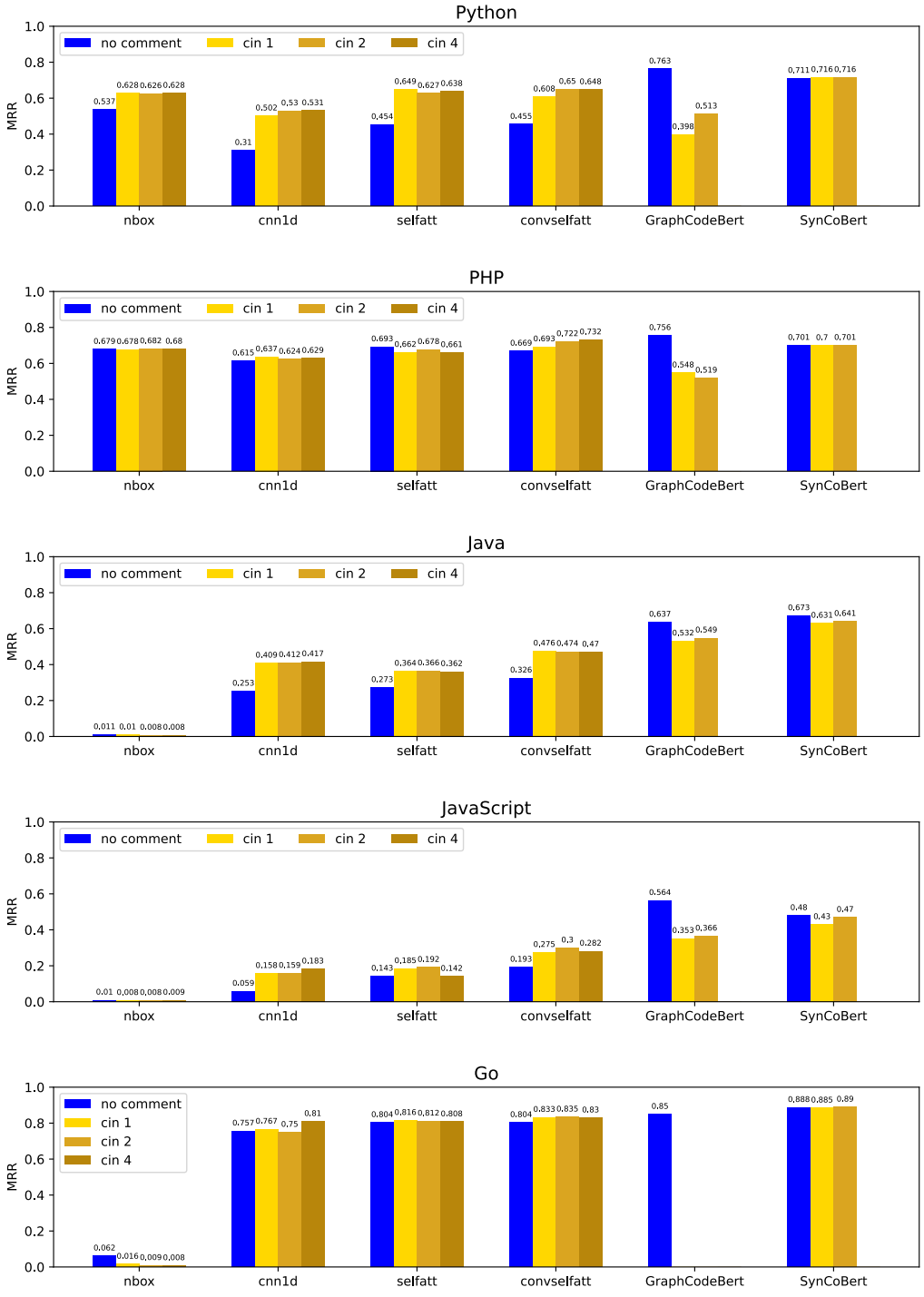


Figure 4.1: The Mean Reciprocal Rank (MRR) for all models trained and evaluated on `no comment` data (*blue*) and `cin` data with an augmentation factor of 1 (*gold*), 2 (*goldenrod*) and 4 (*darkgoldenrod*). Values for the GraphCodeBert and SynCoBert models on `cin 4` are missing.

the `cin` data sets. Regarding the effect of cin modified data compared to the `no comment` data we cannot spot any pattern that stands out. For Python the results improve while for Go the results deteriorate and all other languages do not seem to show any change of performance.

1dCNN, SelfAtt and ConvSelfAtt Models

The 1D convolutional network (1dCNN), self-attention based (SelfAtt) and convolution self-attention based (ConvSelfAtt) models are by far the models that benefit most from the cin modification. The results generally improve for all languages with the exception of the PHP data set in combination with the SelfAtt model. The improvements are only marginal for the PHP and Go languages which for both types of data sets generally show good performance. For the other languages the improvements are rather significant.

GraphCodeBert and SynCoBert Models

The two far more sophisticated models benefit the least from the cin modifications. The results for the GraphCodeBert model are significantly worse for each language we test. The same applies to the SynCoBert model though the performance loss is far less drastic yet still significant with the exception of the Python data set with even shows a marginal improvement.

Go and PHP Languages

The Go and PHP data sets show only minor changes in performance for all models when comparing the `no comment` and `cin` data set versions. The sole exception is the GraphCodeBert model which has deteriorating scores for the `cin` data sets. The two languages also achieve the best overall scores which the exception of the NBoW model which completely fails for the Go data set.

Java and Python Languages

Generally the Java and Python languages show good improvements for the `cin` data sets for all the baseline models. The sole exception is once again the NBoW model which completely fails for the Java data set. For the two more sophisticated models, i.e. GraphCodeBert and SynCoBert, the `cin` data sets tend to worsen the performance of the models.

JavaScript Language

The JavaScript data sets generally show the worst performance throughout all models. While the NBoW model fails for all data set variants, the other baseline models achieve some performance gain with the cin modifications. However, the more sophisticated models again loose performance.

4.2 No Comment models evaluated on CIN data

In this section we analyze the performance of models that have been trained on data that contains the original variable identifiers (`no comment` data) when they are evaluated on data that has this information removed, i.e. `cin` data. We will further analyze the performance difference of those models compared to models that are trained on `cin` data.

As compared to the previous section here we can identify some clearer trends though again not one that is true for every model and language. We will again focus on groups of languages and models instead.

Baseline Models

For the baseline models we notice that the scores for the model trained on `no comment` data but evaluated on anonymized data, i.e. `cin` data, is either around equal or really bad where the model fails completely. For the Java, JavaScript and Go languages no baseline model trained on the `no comment` data works for `cin` data.

GraphCodeBert and SynCoBert Models

For the GraphCodeBert model we first notice that the models evaluated on anonymized data, i.e. `cin` data, always perform worse than evaluated on `no comment` data. Second, we notice that the model being trained on `cin` data performs better than the model being trained on the `no comment` data. The one exception being the Python programming language where the performance is marginally better for the model trained on the `no comment` data.

For the SynCoBert model the trend is the same. The scores for the models evaluated on `cin` data are generally worse or equally good compared to the models evaluated on `no comment` data. The difference to the GraphCodeBert model is that the scores for all three types of training and evaluation data are equal for the Python and PHP languages.

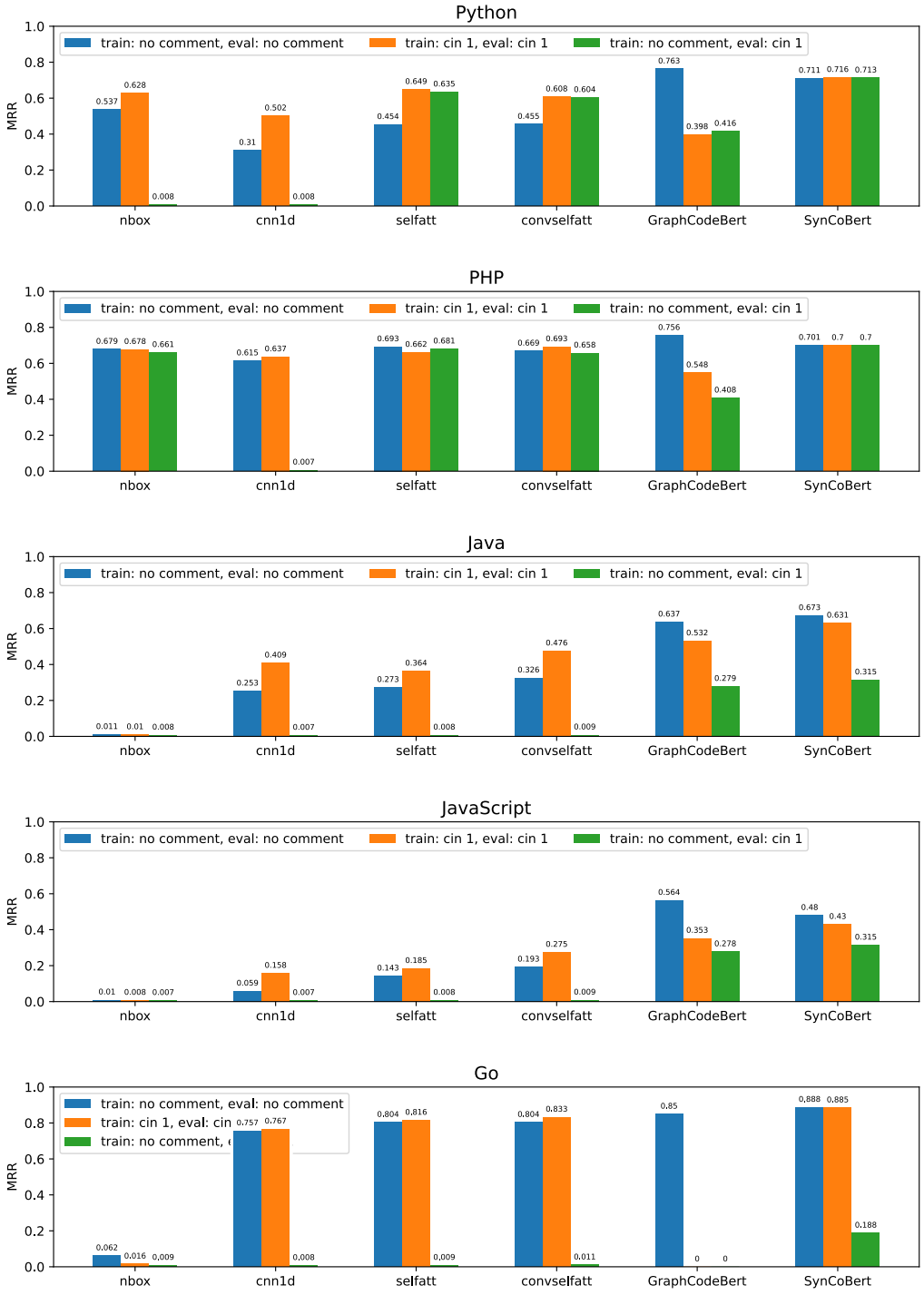


Figure 4.2: Plots for each language showing the Mean Reciprocal Rank (MRR) for each model being trained and evaluated on `no comment` data (*blue*), trained and evaluated on `cin 1` data (*orange*) and trained on `no comment` data and evaluated on `cin 1` data (*green*).

Python and PHP Languages

The Python and PHP data sets are the only two languages for which some of the baseline models achieve meaningful results when being trained on normal (i.e. `no comment`) data and evaluated on anonymized (i.e. `cin`) data. Meanwhile when we train those models on `cin` data as well they all achieve results similar to the results achieved using `no comment` data for training and testing or even surpass them. For the same two languages we also notice the results for the SynCoBert model is almost identical for all three approaches.

Java, JavaScript and Go Languages

The results for the Java, JavaScript and Go languages are poor for all models trained on normal (i.e. `no comment`) data and evaluated on anonymized (i.e. `cin` data). The more sophisticated models are a lot worse compared to when they are trained on `cin` data while the base line models fail completely.

4.3 Augmentation Factor

In this section we analyze the performance difference of the augmentation factors we used.

We see the smallest difference for the NBoW model where the differences are only marginal throughout all languages. The 1dCNN model works best with an augmentation factor of four throughout all languages. The differences between a factor of one or two appears to be rather random though. The SelfAtt and ConvSelfAtt models tend to improve slightly with a factor greater than one though there are once again multiple exceptions. Using the GraphCodeBert we get the biggest differences but they are both negative and positive. Last but not least, SynCoBert shows small improvements when using a factor of two compared to a factor of one.

To summarize, we notice that there is no general performance increase nor decrease that is true for all models or languages when using different augmentation factors. The differences are rather small too.

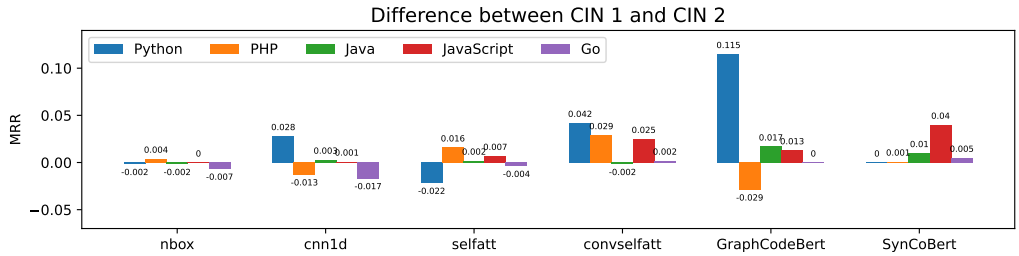


Figure 4.3: Plot showing the difference in Mean Reciprocal Rank (MRR) for each model and language trained on `cin 1` data compared to being trained on `cin 2` data. A positive score means the `cin 2` score is higher, i.e. the values are computed as `cin_2_mrr` - `cin_1_mrr`.

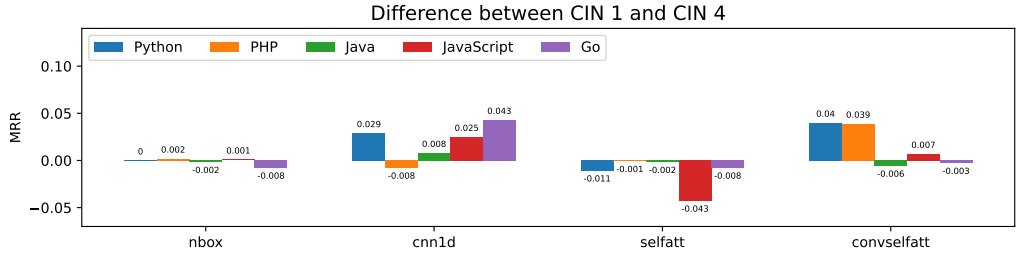


Figure 4.4: Plot showing the difference in Mean Reciprocal Rank (MRR) for each model and language trained on `cin 1` data compared to being trained on `cin 4` data. A positive score means the `cin 4` score is higher, i.e. the values are computed as `cin_4_mrr` - `cin_1_mrr`.

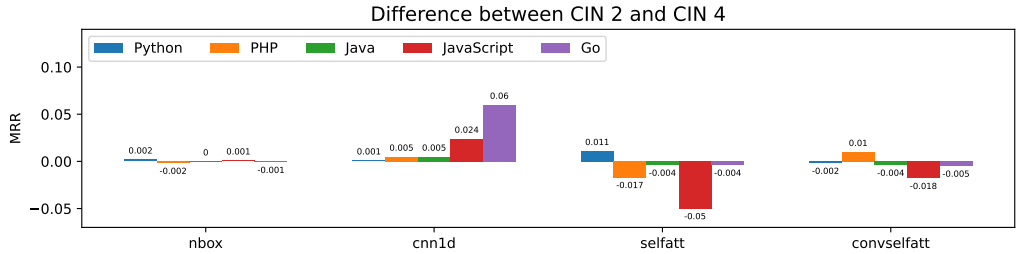


Figure 4.5: Plot showing the difference in Mean Reciprocal Rank (MRR) for each model and language trained on `cin 2` data compared to being trained on `cin 4` data. A positive score means the `cin 4` score is higher, i.e. the values are computed as `cin_4_mrr` - `cin_2_mrr`.

Chapter 5

Conclusion

The questions to which we wanted to find answers in this work where: First, does training an existing model with our cin approach make it outperform itself as compared to training it with normal data? And second, does this approach allow us to train models that work well on anonymized data, i.e. data that has its variable names removed?

In section 4.1 we can see that many baseline models generally get a small boost in performance when training them with `cin` data sets. However, we also notice that the more sophisticated models, i.e. GraphCodeBert and SynCoBert, do not improve with our approach. Especially the GraphCodeBert model shows rather significant performance drops instead. Because those more sophisticated models are the ones that achieve the best overall scores we cannot say that our approach achieves better overall results.

When there is only anonymized data available our approach becomes more interesting. First, as seen in section 4.2, we can see that in most cases the model trained on `cin` data outperforms the same model trained on `no comment` data. As we can transform any type of data that has its variables anonymized into `cin` data we can use this approach to train models that work better on such data. Second, as seen in figure 4.1 using our approach as enough data, i.e. the bigger Python and PHP data sets, we can get some of the baseline models to reach similar scores as the more sophisticated GraphCodeBert and SynCoBert models.

Regarding the use of an augmentation factor, we see only small differences without a real tendency to improve or decrease the performance of all models. We think that the randomized order of the canonical identifiers itself already does enough to remove any left over patterns in the order of the identifiers. Having multiple versions of the same code snippets with different orders of canonical identifiers does not seem to really change anything for the models. More interesting are the significant differences for the GraphCodeBert model for which we do not have an explanation.

Bibliography

[1] F. Markus, "Challenging the lexical focus of code search," confidental, 2023.

[2] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," 2020. [Online]. Available: https://arxiv.org/abs/2009.08366

[3] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: http://arxiv.org/abs/1909.09436

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[5] J. Studer, "Contrastive learning for programming languages," 2021. [Online]. Available: https://pub.tik.ee.ethz.ch/students/2021-HS/BA-2021-25.pdf

[6] N. Yang, F. Wei, B. Jiao, D. Jiang, and L. Yang, "xMoCo: Cross momentum contrastive learning for open-domain question answering," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Online: Association for Computational Linguistics, Aug. 2021, pp. 6120–6129. [Online]. Available: https://aclanthology.org/2021.acl-long.477

[7] "Pytorch website," https://pytorch.org/, accessed: 2023-08-29.

[8] "Tensorflow website," https://www.tensorflow.org/, accessed: 2023-08-29.

Appendix A

Experiment Results

MRR, Trained and Evaluated on `no comment`

Models	Python	PHP	Java	Js	Go
nbow	0.537	0.679	0.011	0.010	0.062
1dcnn	0.310	0.615	0.253	0.059	0.757
selfatt	0.454	0.693	0.273	0.143	0.804
convselfatt	0.455	0.669	0.326	0.193	0.804
GraphCodeBert	**0.763**	**0.756**	0.637	**0.564**	0.850
SynCoBert	0.711	0.701	**0.673**	0.480	**0.888**

Table A.1: The results of all models trained and evaluated on the `no comment` data sets. The results are taken from [1].

Experiment Results: Trained and Evaluated on `CIN 1`

Models	Python				PHP				Java			
Metric	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
nbow	0.628	0.527	0.749	0.81	0.678	0.578	0.801	0.859	0.01	0.002	0.008	0.014
1dcnn	0.502	0.392	0.629	0.714	0.637	0.54	0.751	0.815	0.409	0.328	0.497	0.566
selfatt	0.649	0.555	0.763	0.823	0.662	0.566	0.777	0.835	0.364	0.28	0.45	0.524
convselfatt	0.608	0.498	0.742	0.815	0.693	0.598	0.807	0.86	0.476	0.387	0.572	0.647
GraphCodeBert	0.398	0.294	0.515	0.596	0.548	0.432	0.687	0.769	0.532	0.41	0.674	0.761
SynCoBert	**0.716**	**0.623**	**0.832**	**0.88**	**0.7**	**0.602**	**0.82**	**0.873**	**0.631**	**0.542**	**0.737**	**0.789**

Models	JavaScript				Go			
Metric	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
nbow	0.008	0.001	0.005	0.01	0.016	0.004	0.014	0.026
1dcnn	0.158	0.103	0.21	0.26	0.767	0.699	0.853	0.878
selfatt	0.185	0.13	0.238	0.289	0.816	0.756	0.888	0.913
convselfatt	0.275	0.207	0.343	0.402	0.833	0.783	0.895	0.919
GraphCodeBert	0.353	0.245	0.473	0.566	-	-	-	-
SynCoBert	**0.43**	**0.347**	**0.521**	**0.589**	**0.885**	**0.842**	**0.935**	**0.952**

Table A.2: Results of all models trained and evaluated on the `CIN 1` data sets.

Experiment Results: Trained on `CIN 2`, Evaluated on `CIN 1`

Models	Python				PHP				Java			
Metric	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
nbow	0.626	0.523	0.75	0.81	0.682	0.585	0.801	0.859	0.008	0.001	0.005	0.01
1dcnn	0.53	0.426	0.653	0.727	0.624	0.527	0.738	0.801	0.412	0.329	0.497	0.56
selfatt	0.627	0.529	0.741	0.806	0.678	0.585	0.791	0.847	0.366	0.279	0.458	0.532
convselfatt	0.65	0.547	0.775	0.836	**0.722**	**0.62**	**0.846**	**0.899**	0.474	0.382	0.577	0.65
GraphCodeBert	0.513	0.44	0.623	0.694	0.519	0.446	0.627	0.701	0.549	0.426	0.697	0.779
SynCoBert	**0.716**	**0.623**	**0.832**	**0.88**	0.701	0.603	0.824	0.877	**0.641**	**0.546**	**0.754**	**0.806**

Models	JavaScript				Go			
Metric	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
nbow	0.008	0.001	0.006	0.011	0.009	0.002	0.007	0.013
1dcnn	0.159	0.107	0.205	0.251	0.75	0.676	0.842	0.869
selfatt	0.192	0.138	0.239	0.289	0.812	0.75	0.889	0.916
convselfatt	0.3	0.228	0.377	0.438	0.835	0.783	0.9	0.924
GraphCodeBert	0.366	0.255	0.488	0.583	-	-	-	-
SynCoBert	**0.47**	**0.383**	**0.568**	**0.634**	**0.89**	**0.846**	**0.938**	**0.955**

Table A.3: Results of all models trained on `CIN 2` data sets and evaluated on `CIN 1` data sets.

Experiment Results: Trained on `CIN 4`, Evaluated on `CIN 1`

Models	Python				PHP				Java			
Metric	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
nbow	0.628	0.526	0.75	0.81	0.68	0.582	0.8	0.856	0.008	0.001	0.005	0.01
1dcnn	0.531	0.425	0.654	0.731	0.629	0.531	0.744	0.806	0.417	0.332	0.509	0.571
selfatt	0.638	0.541	0.753	0.814	0.661	0.567	0.774	0.83	0.362	0.279	0.447	0.52
convselfatt	**0.648**	**0.544**	**0.774**	**0.839**	**0.732**	**0.634**	**0.851**	**0.899**	**0.47**	**0.38**	**0.569**	**0.638**
GraphCodeBert	-	-	-	-	-	-	-	-	-	-	-	-
SynCoBert	-	-	-	-	-	-	-	-	-	-	-	-

Models	JavaScript				Go			
Metric	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
nbow	0.009	0.002	0.006	0.011	0.008	0.001	0.005	0.01
1dcnn	0.183	0.128	0.235	0.283	0.81	0.761	0.871	0.894
selfatt	0.142	0.089	0.188	0.24	0.808	0.745	0.887	**0.913**
convselfatt	**0.282**	**0.209**	**0.355**	**0.425**	**0.83**	**0.779**	**0.891**	**0.913**
GraphCodeBert	-	-	-	-	-	-	-	-
SynCoBert	-	-	-	-	-	-	-	-

Table A.4: Results of all models trained on `CIN 4` data sets and evaluated on `CIN 1` data sets.

Experiment Results: Trained on `no comment`, Evaluated on `CIN 1`

Models	Python				PHP				Java			
Metric	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
nbow	0.008	0.001	0.006	0.011	0.661	0.559	0.787	0.847	0.008	0.001	0.005	0.01
1dcnn	0.008	0.001	0.006	0.01	0.007	0.001	0.005	0.01	0.007	0.001	0.005	0.01
selfatt	0.635	0.534	0.753	0.819	0.681	0.589	0.791	0.844	0.008	0.001	0.006	0.012
convselfatt	0.604	0.499	0.733	0.805	0.658	0.563	0.77	0.829	0.009	0.001	0.007	0.013
GraphCodeBert	0.416	0.309	0.542	0.622	0.408	0.3	0.526	0.617	0.279	0.187	0.376	0.462
SynCoBert	**0.713**	**0.618**	**0.83**	**0.879**	**0.7**	**0.602**	**0.82**	**0.872**	**0.315**	**0.228**	**0.406**	**0.487**

Models	JavaScript				Go			
Metric	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
nbow	0.007	0.001	0.007	0.01	0.009	0.001	0.006	0.012
1dcnn	0.007	0.001	0.004	0.01	0.008	0.001	0.006	0.01
selfatt	0.008	0.001	0.007	0.012	0.009	0.002	0.007	0.013
convselfatt	0.009	0.002	0.007	0.013	0.011	0.002	0.009	0.017
GraphCodeBert	0.278	0.182	0.38	0.469	-	-	-	-
SynCoBert	**0.315**	**0.228**	**0.406**	**0.487**	**0.188**	**0.124**	**0.249**	**0.313**

Table A.5: Results of all models trained on `no comment` data sets and evaluated on `CIN 1` data sets.