Analyzing CodeBERT's Performance on Natural Language Code Search

Anonymous ACL submission

Abstract

Large language models such as CodeBERT (Feng et al., 2020) perform very well on tasks such as natural language code search. We show that this is most likely due to the high token overlap and similarity between the queries and the code in datasets obtained from large codebases, rather than any deeper understanding of the syntax or semantics of the query or code.

1 Introduction

Inspired by the success of large pre-trained language models like BERT (Devlin et al., 2019), XLNet (Yang et al., 2019), GPT (Brown et al., 2020) and RoBERTa (Liu et al., 2019) on core NLP tasks, and facilitated by the availability of large datasets that pair natural language with code (e.g. Husain et al., 2019; Lu et al., 2021), there is a growing interest in applying such models to beat the previous state-of-the-art approaches at tasks on natural language and source code, such as code summarization (e.g. Miceli Barone and Sennrich, 2017) or natural language-based code search (e.g. Gu et al., 2018). This work is often motivated by a practical need to help software developers, but can also be seen as an interesting problem to the NLP community (Allamanis et al., 2018).

CodeBERT (Feng et al., 2020), one of the first large language models trained on data consisting of both natural (NL) and programming language (PL) sequences, has been shown to perform particularly well on code search. Code search models assign relevance scores to pairs of source code and natural language queries (descriptions). Early models for code search (e.g. Hill et al., 2011) relied on simplistic metrics based on token overlap, since function names, variable names and comments in high-level programming languages such as Python are often highly descriptive, and very similar (or identical) to the corresponding English descriptions. In this paper, we seek to identify whether CodeBERT exhibits deeper understanding of the structure or underlying semantics of the code, or whether it provides simply a more sophisticated metric of token overlap and similarity, aided by its tokenization and embedding-based token representation. Lu et al. (2021) introduced a test set for code search where function names and variable names are normalized, and CodeBERT performed poorly on this dataset. However, they did not perform any further analysis to explore why CodeBERT fails. Our paper presents a series of studies which investigate how CodeBERT assigns scores to pairs of Python code and English descriptions, and how these scores are related to metrics like token overlap.

Since Feng et al. (2020) trained their code search model as a binary classifier, we first focus on analyzing these classification scores before we examine the impact of these scores on search, where the presence or absence of distractors with similarly high scores matters more than absolute scores. Specifically, we examine the impact of removing or obfuscating certain components of the code, such as function names, comments, and the control structures in the body of the code. In addition to evaluating CodeBERT on Husain et al. (2019)'s CodeSearchNet dataset, we also evaluate it on two versions of CoNaLa (Yin et al., 2018), which allow us to examine performance on shorter code snippets with queries that are either more abstract, or explicitly designed to contain tokens appearing in the code. Our experiments show that CodeBERT does not rely on the structure of the code, but instead largely relies on superficial token similarity, suggesting that it is simply a more sophisticated version of traditional IR models.

2 CodeBERT

Feng et al. (2020) trained RoBERTa-base (Liu et al., 2019), a transformer-based language model with 12 layers, 12 heads and a hidden dimension of 768 (125M parameters in total), on CodeSearchNet

(Husain et al., 2019), a dataset of 2.1M sequences consisting of natural language descriptions paired with code. The resulting model, CodeBERT, induces general-purpose embeddings of the input tokens, and achieves state-of-the-art results on various tasks like natural language code search, code documentation and generation with fine-tuning.

Input to CodeBERT consists of a [CLS] token, the natural language description $\mathbf{w} = w_1 ... w_n$, and the code $\mathbf{c} = c_1 ... c_n$, separated by a [SEP] token, and followed by [EOS]:

$$[\text{CLS}], w_1, w_2, ..w_n, [\text{SEP}], c_1, c_2, ..., c_m, [\text{EOS}].$$

The final embedding of the [CLS] token is taken as the aggregate representation of the entire sequence.

Tokenization Both code and natural language description are tokenized with the same WordPiece model (Wu et al., 2016) that is used by RoBERTa (Liu et al., 2019) for English.

Pre-training: Feng et al. (2020) pre-trained CodeBERT using Masked Language Modeling (MLM) and Replaced Token Detection (RTD) (Clark et al., 2020). In MLM, the model needs to recover input tokens that have been randomly replaced by a [MASK] token, while RTD uses a set of data generators, one one for $\mathbf{w}$ and one for $\mathbf{c}$, to randomly replace tokens with plausible alternatives that the model has to detect.

Natural Language Code Search Feng et al. (2020) train a binary classifier (a softmax layer that receives the final representation of the [CLS] token) to distinguish between correct and incorrect code-description pairs (while fine-tuning CodeBERT itself during the process), and then simply uses this classifier's score to rank a set of 1,000 code snippets (only one of which is correct) for each query.

3 Datasets

We use two datasets in our experiments, CodeSearchNet (Husain et al. (2019), Figure 1a) and CoNaLa (Yin et al. (2018), Figure 1b). Both pair code with natural language descriptions, but CoNaLa has fewer examples, shorter descriptions, and shorter code snippets than CodeSearchNet (Table 1). CodeSearchNet was used to pretrain CodeBERT and contains additional examples in other programming languages.

CodeSearchNet CodeSearchNet (Husain et al., 2019) is a corpus obtained from open-source GitHub repositories for Go, Java, JavaScript, PHP, Python and Ruby. The entire dataset contains 2.1M $(\mathbf{c}_i, \mathbf{d}_i)$ pairs of functions $\mathbf{c}_i$ documented by English descriptions $\mathbf{d}_i$ (Figure 1a). In our experiments, we focus on CodeSearchNet's Python dataset, which contains 503,502 Python functions with corresponding descriptions (their docstrings).[1] Since the examples in this dataset are longer and have a wealth of information, including meaningful function names and comments, we obfuscate or remove certain parts of the code, to see which components of the data have the highest impact on CodeBERT's performance (Section 6.1).

Dataset	Total Examples	Avg. No. of Tokens Per Example	
		Code	Description
CodeSearchNet	503,502	117.15	13.56
CoNaLa	102,379	14.15	8.94

Table 1: CodeSearchNet (Python) vs. CoNaLa

Code:
```
def get_vid_from_url(url):
        vid = match1(url,
        'https?://www.mgtv.com/(?:b|l)/\d+/(\d+).html')
        if not vid:
                vid = match1(url,
        'https?://www.mgtv.com/hz/bdpz/\d+/(\d+).html')
        return vid
```

Description:
Extracts video ID from URL.

(a) An example from CodeSearchNet

Code:
```
os.kill(os.getpid(), signal.SIGUSR1)
```

Original Intent:
How can I send a signal from a python program?

Rewritten Intent:
send a signal `signal.SIGUSR1` to the current process

(b) An example from CoNaLa Dataset

Figure 1: CodeSearchNet (a) and CoNaLa (b)

CoNaLa Dataset CoNaLa (Yin et al., 2018) is a corpus of short Python code snippets (mostly one or two lines long) and English descriptions , created by crawling Stack Overflow (Figure 1b). CoNaLa's code snippets do not contain function definitions or comments. Each snippet is accompanied by a description that was mined with it, called the **original intent**. Snippets in CoNaLa's manually curated part (consisting of 2,379 training examples and 500 test examples) are additionally paired with a a **rewritten intent**, where the curators rephrased the intent to make it more similar to the code, e.g. by using variables that occur in the code. CoNaLa

[1] CodeSearchNet has an additional 652,583 Python functions without descriptions that Feng et al. (2020) used to pretrain CodeBERT.

also has an automatically mined dataset of 600k examples, sorted in descending order of confidence of the pair being a match. CoNaLa was designed to test systems for generating program snippets from natural language, but we can re-purpose it to work on code search by generating appropriate negative examples. For training, we used the top 100k mined pairs along with the manually curated set, for a total of 102,379 positive examples. We discuss the process of generating negative examples in Section 4.The performance of CodeBERT on CoNaLa will tell us whether it generalizes well to other datasets in the same language, and how well it works on shorter code with less information resembling natural language.

4 Experimental Setup

In each of our experiments, we start with Feng et al. (2020)'s publicly available CodeBERT model[2] that was trained on all of CodeSearchNet's training data. Like Feng et al. (2020), we then train binary classifiers on top of this original CodeBERT model, and fine-tuning CodeBERT's parameters during this process, using the same hyper-parameters as Feng et al. (2020). Our training data for both CodeSearchNet and CoNaLa has balanced positive and negative examples. Positive examples are the code-description pairs from the training set, and negative examples are generated by randomly replacing the description in a pair with a description from another data point in the training set.

On a cluster of eight Nvidia GeForce GTX 1080 Ti GPUs, each model was trained for a total of eight epochs and took an average of 797 minutes per epoch for CodeSearchNet and 183 minutes for CoNaLa.

We evaluate these classifiers both on classification and on search. For search, we follow Feng et al. (2020), and sample 999 distractor snippets for each code-description pair in the test sets of CodeSearchNet and CoNaLa. For CodeSearchNet, distractors come from the test set, but since CoNaLa's test set has only 500 examples, we use 499 distractors from the test set and sample another 500 examples from the automatically mined set that were used for training.

[2]CodeBERT is implemented in PyTorch (Paszke et al., 2017) and uses HuggingFace's Transformer library (Wolf et al., 2020).

Threshold	Accuracy	F1	Precision	Recall
0.9	0.9909	0.1770	0.0974	0.9710
0.99	0.9919	0.1935	0.1075	0.9700
0.999	0.9951	0.2831	0.1661	0.9550
0.9999	0.9979	0.4702	0.3148	0.9290

Table 2: The Accuracy, F-1, Precision, and Recall scores for different thresholds. While the recall is high for most thresholds, high precision requires a very high threshold

5 Analyzing CodeBERT as a classifier

Before we evaluate our models in search, we analyze the performance of the model trained on CodeSearchNet as a classifier, as measured by accuracy, precision, recall, and F1. Since we are ultimately interested in performance on search, we use the same (unbalanced) test set as in search, which contains 999 distractors (negative examples) for each (positive) gold pair.

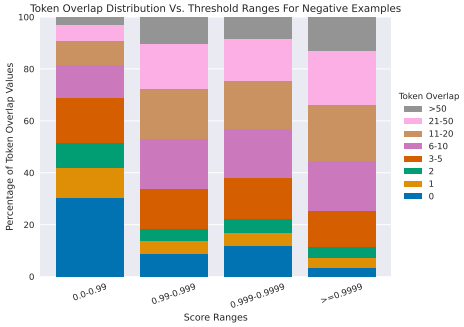
5.1 Classification Results

We first examine how classifier performance is affected by the threshold that we use to translate its real-valued scores into positive and negative labels. Table 2 shows precision, recall, accuracy, and F1 scores for different thresholds. We observe that even a relatively harsh threshold of 0.9 yields a very low F1 score of 0.177: while recall is 0.9710, precision is only 0.0974. Our highest F1 score of 0.4702 is obtained at an even harsher threshold of 0.9999: precision is now 0.3148, but recall has dropped to 0.9290. A full precision-recall curve can be seen in Appendix A.

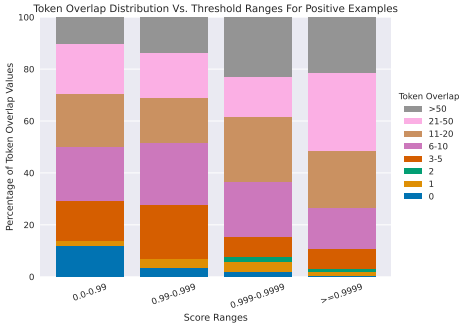
Despite the seemingly low performance on classification, Section 6 will show that search performance of this model is very good. In search, the absolute value of the score is of course less important than whether the score is higher than that of all other code snippets in the search space.

5.2 Dependence on Lexical Overlap

We hypothesize that the confidence scores assigned by CodeBERT to a code-description pair is influenced greatly by their token overlap and high token similarity. In many cases, we observe that a code snippet and a description contain similar tokens, due to the presence of function names, identifier names and comments in the code. For example, in Figure 1a, the function name is very similar to its description. It is difficult to examine how similar a context-dependent language model like CodeBERT



(a) Token overlap in negative examples



(b) Token overlap in positive examples

Figure 2: Distribution of token overlap versus different score ranges in the test set of CodeSearchNet

Description:
Extracts video ID from URL.
WordPiece Tokens of Description:
'ext', 'ract', 's', 'video', 'id', 'from', 'url'

Code:
```
def get_vid_from_url(url):
        vid = match1(url,
        'https?://www.mgtv.com/(?:b|l)/\d+/(\d+).html')
        if not vid:
                vid = match1(url,
        'https?://www.mgtv.com/hz/bdpz/\d+/(\d+).html')
        return vid
```
WordPiece Tokens of Code:
'def', 'get', '_', 'vid', '_', 'from', '_', 'url', '(', 'url', '):', 'Ċ', '', '', '', 'v', 'id', '=', 'match', '1', '(', 'url', ',', '', '', '', '', '', '', '"', 'https', '?', '://', 'www', '.', 'mg', 'tv', '.', 'com', '/(', '?:', 'b', 'l', 'l', ')/', '\\', 'd', '+', '/(', '\\', 'd', '+', ').', 'html', 'âĢ', 'Ļ', ')', 'Ċ', '', '', '', 'if', 'not', 'v', 'id', ':', 'Ċ', '', '', '', '', '', '', '', 'v', 'id', '=', 'match', '1', '(', 'url', ',', '', '', '"', 'https', '?', '://', 'www', '.', 'mg', 'tv', '.', 'com', '/', 'hz', '/', 'bd', 'p', 'z', '/', '\\', 'd', '+', '/(', '\\', 'd', '+', ').', 'html', 'âĢ', 'Ļ', ')', 'Ċ', '', '', '', 'return', 'v', 'id'

Token Overlap: 9 **Confidence Score:** 0.9999

(a) An example from CodeSearchNet with high token overlap between the code and description

Description:
Returns a Google MLEngine service object
WordPiece Tokens of Description:
'return', 's', 'a', 'google', 'ml', 'engine', 'service', 'object'

Code:
```
def get_conn(self):
        authed_http = self._authorize()
        return build('ml', 'v1', http=authed_http,
cache_discovery=False)
```
WordPiece Tokens of Code:
'def', 'get', '_', 'conn', '(', 'self', '):', 'Ċ', '', '', '', '', '', '', '', 'aut', 'hed', '_', 'http', '=', 'self', '._', 'author', 'ize', '()', 'Ċ', '', '', '', '', '', '', '', 'return', 'build', "('", 'ml', "',", "'", 'v', '1', "',", 'http', '=', 'aut', 'hed', '_', 'http', ',', 'cache', '_', 'd', 'iscovery', '=', 'false', ')'

Token Overlap: 2 **Confidence Score:** 0.0057

(b) An example from CodeSearchNet with low token overlap between the code and description

Figure 3: Examples of token overlap and confidence scores in the test set of CodeSearchNet

treats tokens such as 'vid' and 'video', but we can measure the token overlap between code and descriptions. CodeBERT uses WordPiece (Wu et al., 2016) to tokenize both the code and the description. Figure 3 shows what WordPiece tokens look like, and illustrates our definition of token overlap, which is simply the number of WordPiece tokens in the code that appear in the description, with repetition allowed ('url' only appears once in the description, but several times in the code, and each of these occurrences is counted).

In Figure 2 we see how token overlap is distributed over examples from different score ranges. In general, CodeBERT assigns higher scores to examples that have higher token overlap. This reflects the distribution of positive and negative examples in the dataset, since we see that the negative examples in Figure 2a tend to have a much lower token overlap than the positive examples in Figure 2b. We also performed additional studies computing the correlation coefficient between confidence scores and token overlap in Appendix B

This shows that despite being a coarse metric, token overlap is a strong predictor for classifier scores. In Figure 3, we see some code-description pairs from the test set. In 3a, there is a high token overlap of 9 and also a high confidence score of 0.9999. In 3b, the token overlap is low, only 2, and one of the common tokens *"return"* is a common keyword in Python. As a result, CodeBERT gives the pair a low score of 0.0057 despite the code being relevant to the description.

Next, we examine what influences CodeBERT's performance on natural language code search.

6 Analyzing CodeBERT for code search

Like Feng et al. (2020), we use the classifier trained in Section 5 for natural language code search (we resolve ties as described by McSherry and Najork (2008)). Now, the relative scores among all the candidate code snippets for each query is more important than the absolute values of the scores. To understand CodeBERT's performance, we evaluate it under different scenarios. In each scenario, or variant, we transform the dataset during both training and evaluation of the classifier to get a clearer picture of what effect it has on performance.

6.1 Variants of CodeSearchNet

In our experiments on CodeSearchNet (Husain et al., 2019), we consider five main transforma-

```
def get_vid_from_url(url):
    return match1(url, r'youtu\\.be/([^?/]+)') or
        match1(url, r'youtube\\.com/embed/([^/?]+)') or
        match1(url, r'youtube\\.com/v/([^/?]+)') or
        match1(url, r'youtube\\.com/watch/([^/?]+)') or
        parse_query_param(url, 'v') or
        parse_query_param(parse_query_param(url, 'u'), 'v')
```

(a) The original example

```
def hfu_wje_gspn_vsm(url):
    return match1(url, r'youtu\\.be/([^?/]+)') or
        match1(url, r'youtube\\.com/embed/([^/?]+)') or
        match1(url, r'youtube\\.com/v/([^/?]+)') or
        match1(url, r'youtube\\.com/watch/([^/?]+)') or
        parse_query_param(url, 'v') or
        parse_query_param(parse_query_param(url, 'u'), 'v')
```

(b) Obfuscating function names

```
def get_vid_from_url(url)
```

(c) Removing function body

```
get_vid_from_url url
match1 url r'youtu\\.be/([^?/]+)'
match1 url r'youtube\\.com/embed/([^/?]+)'
match1 url r'youtube\\.com/v/([^/?]+)'
match1 url r'youtube\\.com/watch/([^/?]+)'
parse_query_param url 'v'
parse_query_param parse_query_param url 'u' 'v'
```

(d) Removing keywords, delimiters and operators

```
def train(url):
    return match1(url, r'youtu\\.be/([^?/]+)') or
        match1(url, r'youtube\\.com/embed/([^/?]+)') or
        match1(url, r'youtube\\.com/v/([^/?]+)') or
        match1(url, r'youtube\\.com/watch/([^/?]+)') or
        parse_query_param(url, 'v') or
        parse_query_param(parse_query_param(url, 'u'), 'v')
```

(e) Replacing function name from different function

Figure 4: Different transformations of a Python function in CodeSearchNet

tions. Figure 4a shows an original code example. Given an original example (Figure 4a), we can obfuscate function names (4b), remove the entire function body (4c), remove keywords, delimiters and operators (4d), replace function names with those of a different function (4e). Additionally, 32.5% of snippets have comments, which we can keep or remove (Figure 5), resulting in two variants of each transformation ((**w/ Comments**) and **(w/o Comments)**).

Original Function Names: This is the unmodified CodeSearchNet dataset.

Obfuscated Function names: Function names often have a high token overlap with the query. We obfuscate function names by replacing each character in the name with the next character in the alphabet ('a' is replaced by 'b', 'b' by 'c' etc.). This forces the model to focus on other cues, like comments, variable names, or the actual structure of the code like for- loops and if-statements.

```
def _set_env_from_extras(self, extras):
    key_path = self._get_field( extras , 'key_path' , False )
    keyfile_json_str = self._get_field( extras , 'keyfile_dict' , False )
    if not key_path and not keyfile_json_str:
        self.log.info( 'Using gcloud with application default credentials.' )
    elif key_path :
        os.environ[ G_APP_CRED ] = key_path
    else : # Write service account JSON to secure file for gcloud to reference
        service_key = tempfile.NamedTemporaryFile( delete = False )
        service_key.write( keyfile_json_str )
        os.environ [ G_APP_CRED ] = service_key.name # Return file object to have a pointer to close after use,
        # thus deleting from file system.
        return service_key
```

⬇

```
def _set_env_from_extras(self, extras):
    key_path = self._get_field( extras , 'key_path' , False )
    keyfile_json_str = self._get_field( extras , 'keyfile_dict' , False )
    if not key_path and not keyfile_json_str:
        self.log.info( 'Using gcloud with application default credentials.' )
    elif key_path :
        os.environ[ G_APP_CRED ] = key_path
    else :
        service_key = tempfile.NamedTemporaryFile( delete = False )
        service_key.write( keyfile_json_str )
        os.environ [ G_APP_CRED ] = service_key.name
        return service_key
```

Figure 5: Removing comments from code

Adversarial Function Names: We replace the original function name with the name of another function. Unlike the previous transformation, by doing this we trick the model into believing that the function name is present, and performance on this transformation will tell us how well the model works when the function name differs from the actual operations performed in the body of the code.

No Function Body: We remove the entire body of the code and leave only the function definition. Here, we will observe how well the model performs when it only has the function name and its arguments available.

No Code Structure: We remove keywords, operators and delimiters from the function. Here, we force the model to only look at function names, identifier names and comments to identify the correct code snippet for a query. The relative performance of a model on this transformation compared to previous transformations will tell us whether the model leverages function names and comments more than the structure of the code.

6.2 Variants of CoNaLa

As discussed in Section 3, the CoNaLa dataset has two description fields, the original *intent* obtained from Stack Overflow, and a manually *rewritten intent* that contains variables in the code, and hence has higher token overlap with the code. To see the impact of high token overlap due to variable names appearing in the English description, we evaluate CodeBERT on both variants: **CoNaLa (Rewritten Intent)** and **CoNaLa (Original Intent)**.

6.3 Experimental Results

In Table 3, we show the Mean Reciprocal Rank (MRR) and Recall at ranks 1, 2, 5 and 10 (R@k) for the different variants of CodeSearchNet and CoNaLa. The MRR is the average of the recipro-

Model	MRR	R@1	R@2	R@5	R@10
Original Function Names (w/ Comments)	**0.8925**	**0.8106**	**0.9072**	**0.9545**	**0.9710**
Original Function Names (w/o Comments)	0.8800	0.7989	0.9008	0.9515	0.9680
Obfuscated Function Names (w/ Comments)	0.8064	0.7042	0.8180	0.9125	0.9430
Obfuscated Function Names (w/o Comments)	0.7722	0.6404	0.7880	0.8967	0.9375
Adversarial Function Names (w/ Comments)	0.3920	0.1479	0.3472	0.7416	0.9160
Adversarial Function Names (w/o Comments)	0.3420	0.1005	0.2825	0.7140	0.9085
No Code Structure (w/ Comments)	0.8833	0.7841	0.8932	0.9524	0.9685
No Code Structure (w/o Comments)	0.8830	0.7622	0.8884	0.9485	0.9680
No Function Body (w/ Comments)	0.6754	0.5319	0.7010	0.8273	0.8858
No Function Body (w/o Comments)	0.6140	0.4547	0.6250	0.7783	0.8510
CoNaLa (Rewritten Intent)	**0.7145**	**0.5770**	**0.7260**	**0.8840**	**0.9400**
CoNaLa (Original Intent)	0.3290	0.1770	0.2930	0.4700	0.6610

Table 3: Retrieval Scores of all the variants on the test sets of both CodeSearchNet and CoNaLa

cal of the ranks of the gold item returned for each variant, so it gives us a glimpse at the overall distribution of the ranks through a single metric. R@k shows us the fraction of gold items that appear at rank k or above, so it gives us a more detailed view of the distribution of ranks. R@1 is particularly interesting, since it shows us the fraction of gold items that are ranked at the top for each variant. For all variants, we report results from a single run, except for Adversarial Function Names, where we report the mean scores of 10 runs, since the new function name generated in the variant after replacement changes with every run.[3] We can see that the original CodeSearchNet is a much easier dataset than either version of CoNaLa.

6.4 Analysis of Retrieval Scores on CodeSearchNet

In the top part of Table 3, we see a noticeable drop in performance, especially MRR and R@1 when we obfuscate function names. MRR drops by almost 9 percentage points and R@1 drops by more than 10 percentage points. This suggests that when the entire code is available, CodeBERT places more weight on the function names, and when they are obfuscated, the gold example is no longer ranked at the top in at least 10 percent of additional test cases. Removing comments causes a bigger drop in R@1 when the function names are obfuscated (around 6 percentage points), compared to when function names are present (less than 2 percentage points). This means that CodeBERT relies on comments more when function names are not available. We get the biggest drop in R@1 in Adversarial Function Names (more than 66 percentage points), where the function name is from a different function. This means that when the function name and the body of the code are not in agreement, the model chooses to prioritize the function name for discerning the meaning of the code. Removing comments from Adversarial Function Names causes an additional drop of 4 percentage points in R@1, showing again how important comments are when correct function names are not present.

In the second part of Table 3, we see that No Code Structure shows a small drop in R@1 (2.65 percentage points) compared to Original Function Names. Removing comments from this variant reduces R@1 by an additional 2.19 percentage points. Even though this variant does not contain any syntactically correct code, since all keywords, operators and delimiters have been removed, the impact on performance is small compared to the other variants. In No Function Body w/ comments, the R@1 drops to 0.5319 (w/o comments: R@1 = 0.4547). Since this variant only contains the function definition, this big drop in performance shows that CodeBERT does need the function body to perform well. However, CodeBERT still returns the correct code snippet at rank 1 for around half of queries in these variants.

6.5 Analysis of Retrieval Scores on CoNaLa

The third part of Table 3 shows the performance on both variants of CoNaLa. The overall lower scores (as compared to CodeSearchNet) could be because CodeBERT was pre-trained on CodeSearchNet, or because the examples in CodeSearchNet are longer, giving the model more information. But importantly, we observe that R@1 drops sharply by 40 percentage points when we use the **Original Intent**

[3] Adversarial Function Names (w/ Comments): mean MRR = 0.3920 (std. dev. 0.006); Adversarial Function Names (w/o Comments): mean MRR = 0.3420 (std. dev.= 0.009).

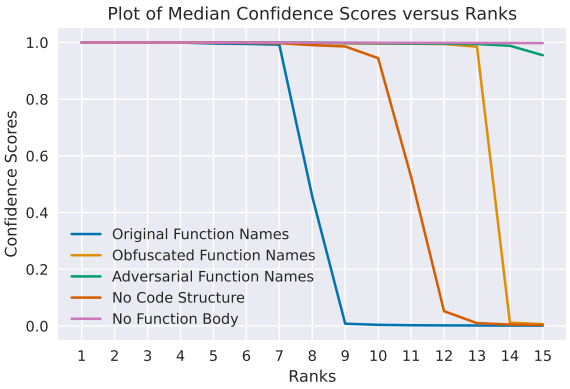


Figure 6: Median confidence scores by rank for each variant of CodeSearchNet (w/comment)

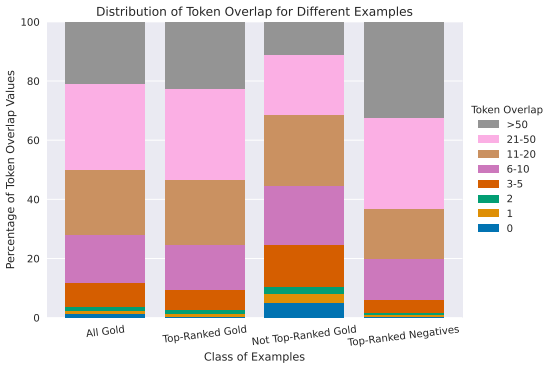


Figure 7: Distribution of token overlap for gold vs top-ranked examples across all queries

instead of the **Rewritten Intent**. This suggests that CodeBERT is dependent on using variable names when they are available, and has an easier time performing retrieval when the gold result has a high token overlap with the user query. However, in real-world scenarios, user queries will typically not contain variables used in the required code snippet, making the performance on the original intent much closer to real-world performance.

6.6 Impact of Variants on Confidence Scores

As we go down the ranked lists returned by CodeBERT, the confidence scores of the code snippets decrease. Figure 6 shows the drop-off of the median confidence scores (across all queries) by ranks in all the variants of CodeSearchNet (with comments). Appendix C gives a more detailed analysis of the distribution of this drop-off. We see that a large drop-off occurs much earlier in better-performing models like Original Function Names (after rank 7) and No Code Structure (after rank 9), whereas the slope of the curve is much gentler for worse-performing models like Adversarial Function Names, where the score starts dropping after rank 14. This suggests that when CodeBERT has access to the original code, it is not only able to identify the gold example as a match, but also assigns lower scores to more examples that are not a match. But obfuscating or removing parts of the data leads to more false positives, since CodeBERT assigns high scores to more negative examples.

6.7 Impact of Token Overlap on Ranking

To understand how ranking is affected by token overlap, we plot the distribution of token overlaps for all gold examples, top-ranked gold examples, gold examples that were not top-ranked, and top-ranked negative examples returned for each query in the test set of the Original Function Names (w/ comments) variant (Figure 7). We see that top-ranked negatives have the highest overall token overlap. This makes sense, since these examples fool the classifier into thinking they are most relevant to the query, so having a high token overlap is likely helpful. Top-ranked gold examples have a slightly lower overlap. We hypothesize this is either because there are no negatives with lower token overlap for these queries, or because these examples were ranked highly not just due to high token overlap, but also due to the nature of the tokens overlapping, like the function names. Unsurprisingly, gold examples that were not ranked at the top have the highest fractions of low or zero overlaps values. We do not know if the top-ranked negatives with zero overlap occur in cases where the non-top-ranked gold item has also zero overlap.

6.8 Summary of Findings

Our experiments (Table 3) show that CodeBERT does not need syntactically correct code (or even code with any control structure) in order to perform well on search. Function names seem to be important though: obfuscating them causes a noticeable drop in performance, and replacing them with a name from a different function causes a much more drastic drop. Comments seem to be less important when the correct function name is present. We also see in Figure 7 that CodeBERT performs poorly in ranking when there are other distractor code snippets present that have a higher token overlap than the gold example, meaning it places more importance on the text similarity than the actual semantics of the code when computing

its relevance to an English description.

7 Discussion and Related Work

Earlier models for code search (Hill et al., 2011; McMillan et al., 2011; Lv et al., 2015) used classical IR approaches that are based on simple word overlap, but were outperformed by simple neural models (Gu et al., 2018), including approaches that incorporate Abstract Syntax Trees (Zhang et al., 2019; Wan et al., 2019; Haldar et al., 2020), Graph Neural Networks (Sieper et al., 2020; Ling et al., 2021; Liu et al., 2021), and reinforcement learning (Yao et al., 2019). However, CodeBERT (Feng et al., 2020) set a new benchmark for code search and other applications by employing a transformer-based large language model. Unlike some earlier neural models, CodeBERT uses only the tokenized code and the English description to perform better at benchmark tasks for program understanding and generation. However, our results indicate the CodeBERT may in fact simply be better at modeling overlap, both because it does not treat tokens as atomic symbols, and because its tokenization greatly increases overlap between queries and gold snippets. While other transformer-based models in this domain have been proposed, like PLBART (Ahmad et al., 2021) and CoTexT (Phan et al., 2021) and CodeGPT (Lu et al., 2021), they still treat code and text as a series of tokens, and are expected to have the same issues as CodeBERT. Moreover, bigger is not always better. Cambronero et al. (2019) presented an improved version of NCS (Sachdev et al., 2018) and showed that a simple bag-of-words-based network outperformed the larger sequence-of-words-based CODEnn (Gu et al., 2018) and SCS (Husain, 2018; Husain and Wu, 2018). This indicates that we need to find novel ways of preprocessing source code instead of treating it like a regular document. In this paper, we did not evaluate if models like GraphCodeBERT (Guo et al., 2020), which incorporate data flow in addition to tokenized codes, address these challenges effectively, but leave this analysis to future work.

8 Conclusion

In this paper, we presented a series of experiments to gain a deeper insight into what makes CodeBERT effective at natural language code search. We saw that token overlap between code and descriptions plays a big role, while the structure of the code has little to no importance to CodeBERT, since it performs essentially equally well when all structure, i.e. keywords, operators and delimiters, are removed from the code. There are some limitations to our study: we did not specifically study the impact of variable names, and how much they contribute to token overlap. We also only looked at function names in the definition, but not at function calls in the body of the function. We also only analyzed the confidence score CodeBERT assigns to examples for code search, and did not look at the attention weights of the model or what type of information each layer produces. Despite that, our experiments establish a clear trend between the predictions of CodeBERT and the token overlap between the function and the query. Future work should also address to what extent a model that fine-tunes CodeBERT with an explicit ranking loss could overcome these shortcomings. And while our work highlighted a big difference in performance between both versions of CodeSearchNet and CoNaLa, we did not attempt to analyze (or overcome) the reasons for this discrepancy, which are likely due to the different sizes of the datasets and the code snippets they contain, and to CodeBERT's pre-training on CodeSearchNet. CoNaLa may also simply be harder because it is concerned with generic algorithmic questions (e.g. "how do I zip lists in Python?") that are answered on StackOverflow, and does not contain code and docstrings taken from large code-bases, where the intelligibility of variable and function names is crucial from a software engineering perspective. Conversely, this might imply that models like CodeBERT that do not attempt to "understand" code, but simply capture surface similarities of code and natural language, might be sufficient for practical applications, since such similarities are likely abundant in well-written and well-documented code. However, this then implies further that code search might not be a useful test case for questions about language understanding. While the large size of real-world datasets makes this task attractive, it may in fact not require any models that try to go deeper, and, as argued e.g. by Bender and Koller (2020), it may be impossible to induce semantic models of code from such data alone. Another open question is how effective models like CodeBERT would be on natural languages other than English, although recent progress on unsupervised machine translation suggests this may not be a significant hurdle either.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4).

Emily M. Bender and Alexander Koller. 2020. Climbing towards NLU: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5185–5198, Online. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 964–974, New York, NY, USA. Association for Computing Machinery.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training text encoders as discriminators rather than generators. In *ICLR*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. Graphcodebert: Pre-training code representations with data flow. *CoRR*, abs/2009.08366.

Rajarshi Haldar, Lingfei Wu, JinJun Xiong, and Julia Hockenmaier. 2020. A multi-perspective architecture for semantic code search. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8563–8568, Online. Association for Computational Linguistics.

Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 524–527.

Hamel Husain. 2018. How to create natural language semantic search for arbitrary objects with deep learning.

Hamel Husain and Ho-Hsiang Wu. 2018. Towards natural language semantic code search.

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

Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2021. Deep graph matching and searching for semantic code retrieval. *ACM Trans. Knowl. Discov. Data*, 15(5).

Shangqing Liu, Xiaofei Xie, Lei Ma, Jingkai Siow, and Yang Liu. 2021. Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search. *arXiv preprint arXiv:2111.02671*.

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

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie

Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270.

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 111–120.

Frank McSherry and Marc Najork. 2008. Computing information retrieval performance measures efficiently in the presence of tied scores. In *Proceedings of the IR Research, 30th European Conference on Advances in Information Retrieval*, ECIR'08, page 414–421, Berlin, Heidelberg. Springer-Verlag.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 314–319, Taipei, Taiwan. Asian Federation of Natural Language Processing.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch.

Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task learning with code-text transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 40–47, Online. Association for Computational Linguistics.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: A neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, page 31–41, New York, NY, USA. Association for Computing Machinery.

Anna Abad Sieper, Omar Amarkhel, Savina Diez, and Dominic Petrak. 2020. Semantic code search with neural bag-of-words and graph convolutional networks. In *SKILL 2020 - Studierendenkonferenz Informatik*, pages 103–115, Bonn. Gesellschaft für Informatik e.V.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 13–25. IEEE Press.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, ukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 2203–2214. ACM.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794.

A Precision-Recall Curve of Classifier

We plot the Precision-Recall curve of the classifier from Section5 in Figure 8. We observe a sharp drop in precision after the recall goes beyond 0.8, suggesting that it is impossible to get this model to perform well on classification if we require a recall higher than 0.8.

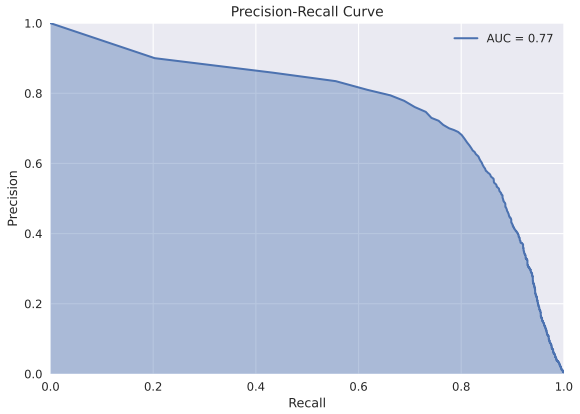


Figure 8: Precision-Recall Curve of the CodeBERT binary classifier on the test set of CodeSearchNet

B Correlation Between Token Overlap and Confidence Score

To further explore how token overlap and confidence scores are correlated over different score ranges, we used three different correlation metrics in this study - the **Pearson**, the **Spearman's Rank** and the **Kendall Rank Correlation Coefficient**.

While Pearson Coefficient looks at the correlation between the absolute values of the respective scores, the other two metrics look at the rank correlation or how well items are ranked using these scores, which is more relevant since in the ranking task we are more concerned with the rank assigned to a candidate code-description pair instead of the absolute value of its relevance score.

Threshold	Pearson	Spearman	Kendall
0.9	0.1007	0.3047	0.2155
0.99	0.1368	0.2799	0.196
0.999	0.2282	0.3793	0.267
0.9999	0.192	0.3954	0.2735

Table 4: The Pearson, Spearman and Kendall rank coefficients for code-description pairs at different threshold cutoffs

In Table 4 we see the correlation coefficients for all code-description pairs in the test set of the CodeSearchNet dataset above a certain threshold. Higher thresholds give us the correlation between the pairs that were scored high by CodeBERT. We see that there is a high rank correlation coefficient (both Spearman and Kendall) for high scoring examples, whereas the correlation between them is low when we consider all pairs. For instance, code-description pairs with scores above 0.9999 have a Spearman correlation coefficient of 0.3954 between their scores and their token overlap, whereas for examples scoring above 0.9, the correlation falls to 0.3047. This shows that during the ranking task, high token overlap in negative examples makes it difficult to find the gold example.

C Additional Analysis of Confidence Scores in Code Search

C.1 Median of Drop-off in Confidence Scores on CoNaLa

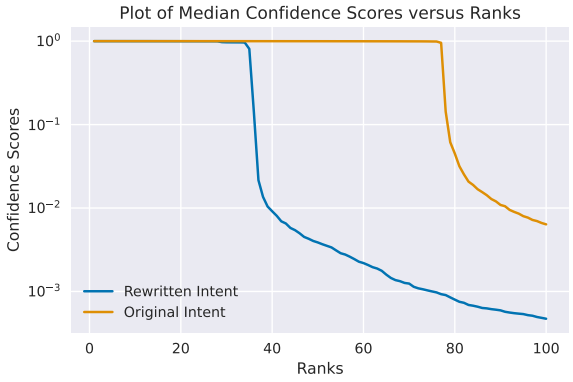


Figure 9: Median confidence scores of the top 50 ranks for all test examples for the CoNaLa dataset over each variant

Similar to Section 6.6, we computed the median drop-off in confidence scores on the test set of the CoNaLa dataset in Figure 9. We see that the drop-off in confidence scores is sharper when using the **Rewritten Intent** compared to the **Original Intent**. This is because here too for higher performing variants the model has an easier time identifying negative examples and scoring them low to make ranking easier. However, the performance here is much worse than on CodeSearchNet. In **Rewritten Intent**, there is a large drop in confidence scores only after rank 35, whereas in **Original Intent**, we do not see a large drop even after rank 50. This means given a query, the model will return a large number of candidate code snip-

pets with high confidence scores that it believes to be correct. This shows that even after being fine-tuned on CoNaLa, CodeBERT does not perform well on this dataset, suggesting that it cannot be easily be used on new datasets, even with the same programming language.

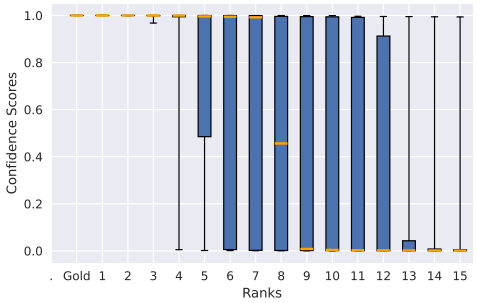
C.2 Distribution of Drop-off in Confidence Scores on CodeSearchNet

We plot boxplots showing the distribution of confidence scores on the test set of CodeSearchNet for each of the top 15 ranks. Similar to plotting the median, the box plots in Figure 10 show that higher performing variants show a large drop-off in scores much earlier than worse-performing variants. The first column, shows the distribution of confidence scores for the gold examples, and the subsequent columns show the distribution of confidence scores at each rank. The orange lines in Figure 10 denote the median, and the blue bars show the first and third quartiles. The whiskers show the highest and lowest scores at each rank.

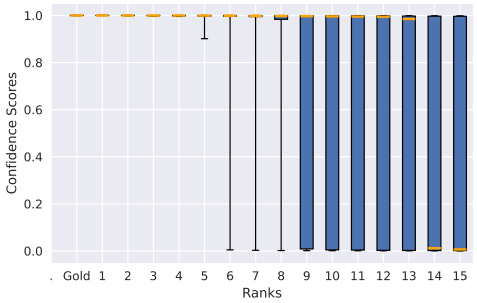
We see in Figure 10a that in Original Function Names, which is the best-performing variant, all the examples in the top 3 ranks are assigned very high confidence scores. In the second-best variant, No Code Structures in Figure 10d, we see a large drop in confidence scores after rank 4. In Obfuscated Function Names in Figure 10b, most queries see at least eight candidate code snippets with very high confidence scores. Both Adversarial Function Names (10c) and No Function Body (10e), the model returns more candidate code snippets for each query. This implies that worse performing variants have far more false positives, and given a query there is a lot more ambiguity as to what the correct code snippet is. We observe that out of all the variants where we modified the original data, the best performing variant is No Code Structures (10d) which does not even contain syntactically correct code.

C.3 Distribution of Drop-off in Confidence Scores on CoNaLa

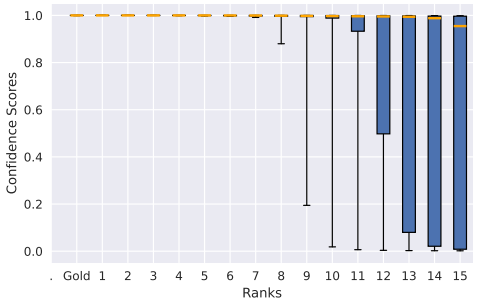
We also computed the boxplots showing the distribution of confidence scores of the top 25 results in the test set of the CoNaLa dataset and showed them in Figure 11. Overall, we see that the confidence scores in this dataset are much more ambiguous compared to the scores in CodeSearchNet, meaning that all examples in the top 25 ranks have very confidence scores (above 0.998). This shows that given a query in CoNaLa, CodeBERT is not very confident what the correct code snippet is out of a pool of candidates.



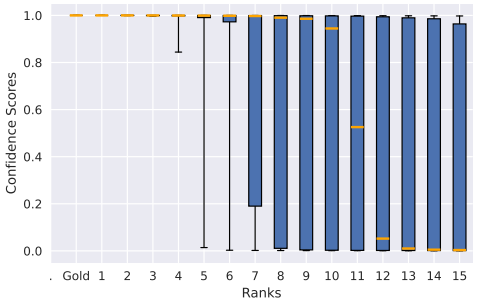
(a) Original Function Names



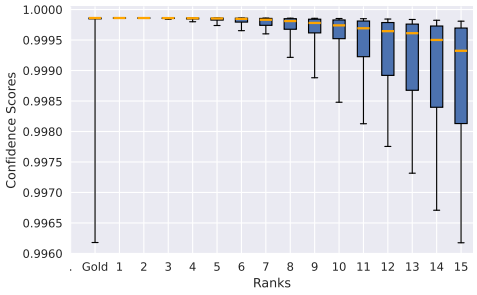
(b) Obfuscated Function Names



(c) Adversarial Function Names

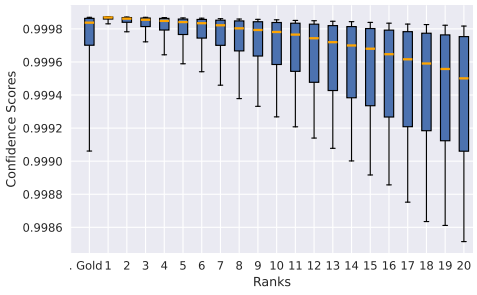


(d) No Code Structure

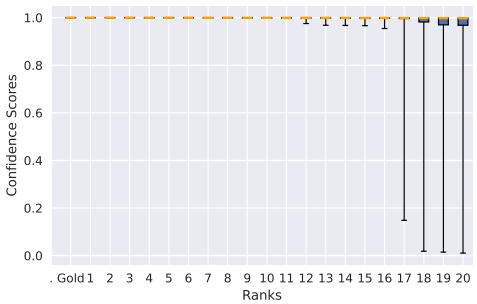


(e) No Function Body

Figure 10: Boxplots of Confidence Scores versus the top 15 Ranks for each variant. The scores drop off earliest for (a) and the general trend is for lower performing models the scores drop off more gently.



(a) CoNaLa (Rewritten Intent)



(b) CoNaLa (Original Intent)

Figure 11: Boxplots of confidence scores assigned to the top-20 ranks and the gold examples for when we use the rewritten intent vs the intent.