CoCluBERT: Clustering Machine Learning Source Code

Marcus Hägglund[†], Francisco J. Peña[†], Sepideh Pashami[‡], Ahmad Al-Shishtawy[‡], Amir H. Payberah[†‡]
[†]KTH Royal Institute of Technology, Stockholm, Sweden
[‡] RISE Research Institutes of Sweden, Stockholm, Sweden
[†]{mahaggl,frape,payberah}@kth.se [‡]{sepideh.pashami,ahmad.al-shishtawy}@ri.se

***Abstract*—Nowadays, we can find machine learning (ML) applications in nearly every aspect of modern life, and we see that more developers are engaged in the field than ever. In order to facilitate the development of new ML applications, it would be beneficial to provide services that enable developers to share, access, and search for source code easily. A step towards making such a service is to cluster source code by functionality. In this work, we present CoCluBERT, a BERT-based model for source code embedding based on their functionality and clustering them accordingly. We build CoCluBERT using CuBERT, a variant of BERT pre-trained on source code, and present three ways to fine-tune it for the clustering task. In the experiments, we compare CoCluBERT with a baseline model, where we cluster source code using CuBERT embedding without fine-tuning. We show that CoCluBERT significantly outperforms the baseline model by increasing the Dunn Index metric by a factor of 141, the Silhouette Score metric by a factor of two, and the Adjusted Rand Index metric by a factor of 11.**

***Index Terms*—Source Code Clustering, NLP, BERT, CuBERT**

I. Introduction

We recently witnessed a significant increase of open source code, particularly machine learning (ML) code, in public platforms such as GitHub. Given so many ML code repositories, it would be beneficial to provide a service for orchestrating them to enable users to search for code easily. However, many of the available source code in public repositories are not labeled, meaning that they do not have any metadata or extra information to describe their functionality. So the question is how to categorize unlabeled source code by their *functionality*?

One approach to manage and understand source code is to use *software clustering* technique [1], [2]. Grouping and clustering source code according to their similarities makes it easier to find and manage relevant pieces of code and helps ML engineers to understand code more easily. Many of the existing works leverage source code metadata, documentation, or imported packages to group them [3]. However, we aim to cluster source code based on their functionality. For example, Figure 1 shows two methods in Python with the same functionality (creating generators for a dataset), which are implemented differently. While this similarity is relatively easy to spot for an experienced developer, the same cannot be said for a model that faces the same task.

In this work, we investigate the possibility of learning representations of source code for the downstream task of clustering them by their functionality. To achieve this, we use a BERT-based approach [4] for making source code embedding. Recently, many works use such embedding for various tasks, such as code summarization [5]–[8], code search [9]–[11], and code generation [12], [13], to name a few. However, to the best of our knowledge, there is no work on clustering source code based on their functionality. To this end, we introduce *Code Clustering BERT* (CoCluBERT) based on CuBERT [14] (a variant of BERT pre-trained on source code), and present three different frameworks for fine-tuning it for the clustering task: (i) CoCluBERT-*Triple*, where we use a triplet loss function [15], [16], (ii) CoCluBERT-*Unsupervised*, where we use an unsupervised clustering loss function [17], and (iii) CoCluBERT-*DRC*, where we use the Deep Robust Clustering (DRC) loss function [18].

The experimental results confirm that we can learn to embed source code that encode functional similarities between methods. Through the experiments, we observe that CoCluBERT outperforms a baseline model, where we cluster code using the embedding created by CuBERT without fine-tuning it. Moreover, the results indicate that the contrastive learning

```python
def create_tf_datasets(filepath):
  df = pd.read_csv(filepath)

  train, test = train_test_split(df,
    test_size=0.20, shuffle=True)

  train_ds = tf.data.Dataset.from_tensor_slices(
    (train.features, train.label))

  test_ds = tf.data.Dataset.from_tensor_slices(
      (test.features, test.label))

  return train_ds, test_ds
```

(a) Program A

```python
def create_torch_datasets(path, params):
  dataset = Dataset(path)

  train_set, test_set =\
    torch.utils.data.random_split(dataset, [
      int(dataset.__len__() * (1 - params.test_size)),
      int(dataset.__len__() * params.test_size)
    ])

  train_dataloader = torch.utils.data.DataLoader(
    train_set, batch_size=params.batchsize,
    shuffle=True,
  )

  test_dataloader = torch.utils.data.DataLoader(
    test_set, batch_size=params.batchsize,
    shuffle=True,
  )

  return train_dataloader, test_dataloader
```

(b) Program B

Fig. 1: An example of two source code with similar functionality, but with different implementations.

fine-tuning techniques in CoCluBERT-Triplet and CoCluBERT-DRC are better suited for clustering source code compared to CoCluBERT-Unsupervised. We also achieve the most compact and well-separated clusters when fine-tuning CuBERT in CoCluBERT-Triplet.

II. Background

In this section, first, we present some of the existing models for language modeling, in particular, BERT [4] and CuBERT [14], and then review the clustering techniques we use in this work.

A. Language Modeling

Embedding is a technique in which an object that is possibly non-numeric and discrete (e.g., words and sentences) is encoded into a real-valued fixed-size vector [9]. These types of representations can encode semantic similarities and analogies between objects [19]. Intuitively, we can say that in this space, the meaning of an object is distributed across the vector components and the goal of the encoding is to learn vector representations such that semantically similar objects have vectors close to each other [20]–[22].

Word2Vec [23] and GloVe [24] are examples of embedding models that are trained on large text corpora based on co-occurrence statistics to generate semantic representations of words. However, these models generate context-free word embedding, meaning that the representation of a word they create is irrespective of the meaning of the word in that sentence. To overcome this limitation, many learning techniques have been proposed to generate contextualized representation [25]–[28], among which Bidirectional Encoder Representations from Transformers (BERT) [4] shows ground-breaking performance in a various Natural Language Processing (NLP) tasks.

BERT [4] is a language embedding model that learns contextual representations of words in a sentence. BERT is pre-trained on a large corpus of text in an unsupervised setting, with two different learning objectives: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). In MLM, some words in the input sentence are masked (hidden) from the model, and the model should predict the original word of the masked token based on the context of the other non-masked words in the sentence. In NSP, BERT is given two sentences, A and B, as input, and the objective is to make BERT predict whether sentence A is followed by sentence B. Once BERT is pre-trained, it can be applied to various downstream tasks, such as question-answer tasks [29] and sentence classification [30]. This is done by adding an extra task-specific output layer to the pre-trained model and fine-tune it.

CuBERT [14] is an adaption of BERT to learn contextual embeddings of source code. CuBERT's architecture and pre-training process are the same as BERT; however, the corpus that the models are pre-trained on and tokenization step are different. CuBERT is pre-trained on a large corpus of 7.4 million Python files extracted from GitHub. Moreover, the BERT tokenization technique is unsuitable for source code since it is mainly designed for natural languages and does not preserve many of the important syntactic elements in programming languages (e.g., newlines and indentations) and special characters (e.g., parentheses, semicolons, and arithmetic operators). By preserving these elements in the training data, CuBERT can gain a deeper understanding of the meaning and context of different elements in the source code.

B. Clustering Methods

There are various clustering techniques in literature, but in this work, we consider *centroid-based*, *density-based*, and *hierarchical* clustering.

Centroid-based clustering methods partition the data points in a dataset by finding the centers for the given number of clusters and assigning each data point to its closest cluster center in such a way that the squared distances between the points and their cluster center are minimized. K-means [31] is a well-known clustering algorithm in this category.

Density-based clustering methods assign clusters based on the density of regions in the data. The underlying assumption is that the data points that lie within high-density regions are more similar and different from points that lie in the lower-density regions of the space. *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN) [32] and *Ordering Points To Identify the Clustering Structure* (OPTICS) [33] are two techniques in this group. Unlike K-means, these algorithms do not require the number of clusters specified before running the algorithm.

Hierarchical clustering methods form clusters following a tree-like structure based on the hierarchy in the data, and create new clusters from the previously formed clusters. *Hierarchical Agglomerative Clustering* (HAC) [34], as an algorithm in this group, initially assigns each data point to a separate cluster, and then the two most similar clusters are merged to form a new cluster. Once the clusters are merged, the new similarity measures are computed again, and the merging continues.

III. Methodology

Here, we present Code Clustering BERT (CoCluBERT), in which we use CuBERT [14] and fine-tune it for source code clustering based on their functionality. We take advantage of three different loss functions for fine-tuning CuBERT: (i) triplet loss function [15], [16], (ii) unsupervised clustering loss function [17], and (iii) Deep Robust Clustering (DRC) loss function [18]. Considering these loss functions, we call our models CoCluBERT-Triplet, CoCluBERT-Unsupervised, and CoCluBERT-DRC, respectively. As input, we give the source code methods to the tokenizer and then give the tokenized data to the model. In CoCluBERT-Triplet and CoCluBERT-Unsupervised, we fine-tune CuBERT according to the respective criterion and then perform the clustering (e.g., using K-means). In contrast, CoCluBERT-DRC follows an end-to-end structure where a single model performs the source code embedding and cluster assignment; thus, it does not need any external clustering technique to obtain clusters. The implementation of these models are available in GitHub[1].

[1] https://github.com/ai-center-kth/cuBERT-source-code-clustering

A. CoCluBERT-*Triplet*

The triplet loss function, presented in [15], is used for training models to learn sentence embedding while preserving the semantic similarities among similar sentences. Inspired by [15], in CoCluBERT-Triplet, we fine-tune CuBERT using the triplet loss function to learn source code methods embedding that encodes their functionalities. Our goal is to make the embedding such that the distance between the embedding of methods with similar functionality is small, while the distance between different methods is large.

To form a triplet, we choose one sample as an *anchor* method and two other samples as *positive* and *negative* methods. The positive method has the same functionality as the anchor method, while the negative one has different functionality. Given the embedding of anchor, positive, and negative methods (denoted by a, p, and n, respectively), we define the triplet loss $\mathcal{L}(a, p, n)$ as:

$$\mathcal{L}(a, p, n) = max(d(a, p) - d(a, n) + \alpha, 0), \qquad (1)$$

where $d(x, y)$ measures the distance between x and y, and α is the margin that ensures the positive method is closer to the anchor than the negative method. We can consider different distance metrics for d, such as the Euclidean and Cosine distances.

To construct a triplet of methods, we first choose a method randomly from the dataset as an anchor and then select a positive and a negative method for the anchor accordingly. To choose a positive method, we find all methods in the dataset that share one or more of the subwords in the anchor method name. For instance, if the anchor method name is `train_epoch`, then `train_for_epoch` and `get_model_for_training` are its potential positive methods and `get_dataset` and `normalize` are potential negative methods. To do this, we use the BLEU score [35] to compute the similarity between the method names. A higher BLEU score indicates a more similar method name. For example, the BLUE scores between `train_epoch` and `train_for_epoch` and `normalize` are 0.92 and 0, respectively. For each anchor, we select a positive method randomly, such that the methods with higher BLEU scores are more likely to be chosen. A negative method is obtained by randomly sampling a method from the dataset in the complement set of the positive candidates.

As Figure 2 (top-left corner) shows, the CoCluBERT-Triplet consists of three instances of the same CuBERT model, sharing the weights. However, in practice, we use only one single instance of CuBERT with three different input channels to adhere to the triplet structure. To train the model, first, we tokenize each method in the triplet (anchor, positive, and negative) and give them as input to CuBERT to compute the features. We then use a pooling layer to aggregate the set of features that CuBERT computes for each token to obtain a single fixed-length vector representing the given method. We consider the output of the first token (i.e., `[CLS]`) as the representation of the method [15]. In the end, we use the embedding of the anchor, positive, and negative methods (i.e., a, p, and n, respectively) to compute the triplet loss $\mathcal{L}(a, p, n)$.

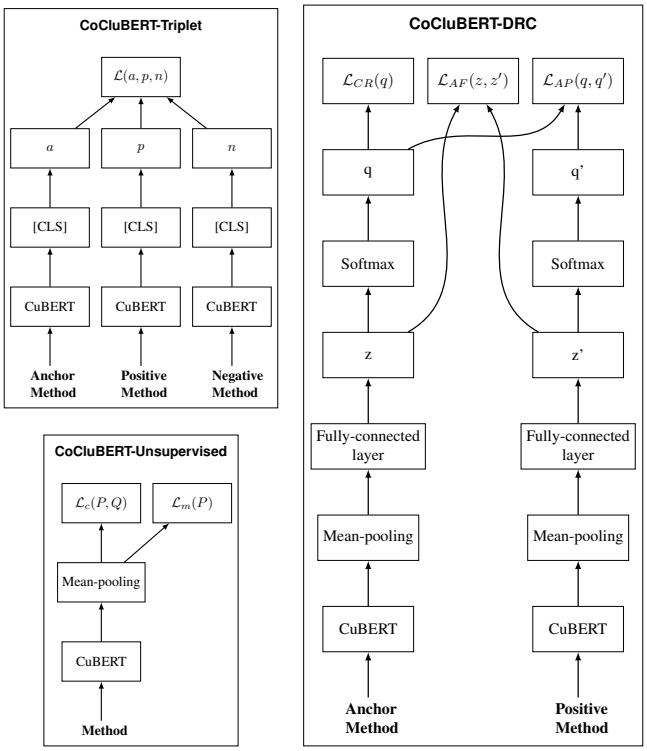


Fig. 2: Top-left: CoCluBERT-Triplet, Bottom-left: CoCluBERT-Unsupervised, Right: CoCluBERT-DRC.

B. CoCluBERT-*Unsupervised*

Huang et al. [17] propose an unsupervised fine-tuning of BERT to learn representations of text for clustering. In CoCluBERT-Unsupervised, we apply the same technique for source code clustering using CuBERT as a pre-trained model. We make an embedding z of a method by giving the tokenized method to CuBERT and applying the mean-pooling on CuBERT output [17]. Then, by computing the loss function as explained below, we update the model. As Figure 2 (bottom-left corner) shows, the loss function $\mathcal{L}$ has two parts: a Masked Language Model (MLM) loss $\mathcal{L}_m$ and a Clustering loss $\mathcal{L}_c$:

$$\mathcal{L} = \mathcal{L}_m + \mathcal{L}_c. \qquad (2)$$

The MLM loss $\mathcal{L}_m$ is the same as the MLM loss in BERT [4]. However, unlike the vanilla BERT that uses the embedding of `[CLS]` token as the representation of input samples, we use the mean-pooling to compute the embedding z of the input source code method, where $z = \frac{1}{M}\sum_i^M h_i$. Here, h_i is the hidden vector of ith token of the input sample and M is the number of tokens.

In addition to $\mathcal{L}_m$, we use the Clustering loss $\mathcal{L}_c$ for better cluster separation and compactness [36], [37]. To define $\mathcal{L}_c$, we consider Q as the distribution of soft assignment of the method embedding z to clusters by Student's t-distribution [38]. To be more precise, q_{zc} shows the similarity between the method embedding z and the clustering centroid μ_c, which is obtained from the chosen clustering technique

(e.g., K-means) [36], [37]:

$$q_{zc} = \frac{(1 + \|z - \mu_c\|^2)^{-1}}{\sum_{c'}(1 + \|z - \mu_{c'}\|^2)^{-1}}. \quad (3)$$

We then use the distribution of soft assignment q_{zc} to assign the cluster label C_z to the method embedding z, such that $C_z = \arg\max_c q_{zc}$. Moreover, from Q we derive an auxiliary target distribution P that puts more emphasis on samples assigned with high confidence [17]:

$$p_{zc} = \frac{q_{zc}^2 / f_c}{\sum_{c'} q_{zc'}^2 / f_{c'}}, \quad (4)$$

where $f_c = \sum_z q_{zc}$ is the soft cluster frequency. Finally, we express $\mathcal{L}_c$ as the Kullback-Leibler (KL) divergence between two distributions Q and P [17]:

$$\mathcal{L}_c = KL(P\|Q) = \sum_z \sum_c p_{zc} \log \frac{p_{zc}}{q_{zc}}. \quad (5)$$

To train CoCluBERT-Unsupervised, we need a set of initial cluster centroids, μ_c. To achieve it, we first compute the embedding of all the methods in the training set and then use K-means to obtain an initial set of cluster centroids.

C. CoCluBERT-*DRC*

In CoCluBERT-DRC, we use Deep Robust Clustering (DRC) technique [18] for fine-tuning CuBERT. DRC considers both the embedding of samples and their clustering assignment to decrease intra-class diversities while increasing inter-class diversities simultaneously. DRC is an end-to-end model that makes the data embedding and clustering together, so it does not use any external clustering algorithm [18]. DRC makes use of contrastive learning [39] and data augmentations for clustering. It uses contrastive learning to separate positive samples from negative ones and uses data augmentation to make more robust embedding by maximizing mutual information between samples and their embeddings [40]. Considering that a sample and its augmentation have similar identity, it is expected that they are assigned to the same cluster.

If the dataset consists of N samples drawn from K semantically different classes, the objective of DRC is to assign the samples to the different clusters, such that semantically similar samples are assigned the same cluster while also maintaining a robust clustering. The DRC loss function $\mathcal{L}$ is defined based on three components: a contrastive loss based on the Assignment Features (AF) $\mathcal{L}_{AF}$, a contrastive loss based on the cluster Assignment Probability (AP) $\mathcal{L}_{AP}$, and a Cluster Regularization (CR) loss $\mathcal{L}_{CR}$ to prevent trivial solutions [18]:

$$\mathcal{L} = \mathcal{L}_{AF} + \mathcal{L}_{AP} + \lambda \mathcal{L}_{CR}, \quad (6)$$

where λ is the regularization term weight.

The AF loss $\mathcal{L}_{AF}$ preserves the similarity in the embedding level, meaning that the embedding of a sample and its augmentation, denoted by z_i and z_i', respectively (Figure 2, the right part), should be similar:

$$\mathcal{L}_{AF} = -\frac{1}{N} \sum_{i=1}^{N} \log \Big(\frac{e^{z_i z_i'/T}}{\sum_{j=1}^{N} e^{z_i z_j'/T}} \Big), \quad (7)$$

where T affects the probability distribution obtained from the softmax, making it more or less certain about its predictions.

The AP loss $\mathcal{L}_{AP}$ maximizes the similarity between the predicted clusters of samples and the augmented samples. So, if q_c and q_c' tell us that the samples and the augmented ones are assigned to cluster c (Figure 2, the right part), then:

$$\mathcal{L}_{AP} = -\frac{1}{K} \sum_{c=1}^{K} \log \Big(\frac{e^{q_c q_c'/T}}{\sum_{c'=1}^{K} e^{q_c q_{c'}'/T}} \Big), \quad (8)$$

The CR loss prevents the model from falling into local optimum.

$$\mathcal{L}_{CR} = -\frac{1}{N} \sum_{c=1}^{K} \Big(\sum_{i=1}^{N} q_c(i) \Big)^2. \quad (9)$$

To train the model, first, we select a method randomly from the dataset. Then, we use the same procedure as in CoCluBERT-Triplet, leveraging the BLEU score between method names, to find a positive sample as an augmented method for the chosen method. Next, we give the method and its augmentation to CuBERT and then apply a mean-pooling aggregation to reduce the set of features into a single vector representing the method. In the next step, we use a fully connected layer to reduce the feature space into K, where K is the number of desired clusters. The output from the fully connected layer is the AF vector for the method (the method embedding, which are denoted by z and z'). Finally, we apply the softmax function to the method embedding to obtain their cluster AP (denoted by q and q').

IV. Evaluation

In this section, first, we present the dataset and explain the evaluation metrics, and then we study the performance of our proposed models, i.e., CoCluBERT-Triplet, CoCluBERT-Unsupervised, and CoCluBERT-DRC.

A. Datasets

We make the dataset by downloading a large set of ML source code from publicly available repositories on GitHub. We only download Python projects with two or more stars containing the "machine-learning" tag. We then discard none Python files and keep Jupyter notebooks and convert them into equivalent Python scripts. Finally, we set the code granularity at the method level and take their Abstract Syntax Trees (AST) as their identity.

Among all the methods, we keep only those with the method names containing one of the sub-strings `train`, `save`, `process`, `forward` and `predict`, and consider them as labels indicating the functionality of the methods. For example, `train_epoch`, `train_for_epoch` and `get_model_for_training` are all methods with `train` in their method name. The reasoning for choosing these particular words is that they are highly relevant and semantically meaningful words in ML codes. They are frequently referenced when writing the source code for the different steps of the ML process. Therefore, the objective is to investigate whether the clusters found in the embedding space are meaningful with

respect to these labels. In the experiments, we consider five clusters ($K = 5$), one for each of these labels. However, it is also possible that the function names do not represent the functionally of the functions, e.g., if developers choose improper names for the functions. Therefore, we formulate the problem as an unsupervised setting to investigate how to group code snippets based on their functionalities rather than their names.

Our dataset consists of 8106 methods in total, and we shuffle and split them into training, validation, and test sets using a 90-5-5 ratio. Table I shows the number of samples of each type (i.e., with the specific word in the method name). All the details of our implementation to make the dataset are available on the following link[2].

TABLE I: The number of methods containing specific words in their method names.

Train	Save	Process	Forward	Predict	Total
3131	1513	1908	282	1272	8106

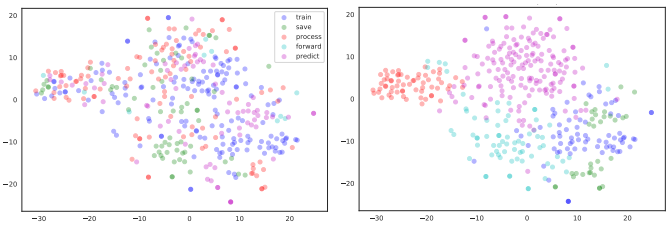
B. Clustering Methods and Evaluation Metrics

For clustering, we conduct the experiments using K-means [31], OPTICS [33], and HAC [34] techniques. The choice of these algorithms is to choose one from the different families of clustering techniques presented in Section II-B. To evaluate the quality of the clusters created by these techniques, we consider two types of metrics: (i) *Dunn Index* (DI) and *Silhouette Score* (SS) that do not require labels of data to evaluate the results, and (ii) *Adjusted Rand Index* (ARI) that need labels of data.

The metrics, such as DI and SS that examine the quality of clustering without requiring labels, generally provide insights regarding the separability and compactness of the clusters and determine the validity of the assigned clusters. DI identifies if the clusters are compact and well-separated. The bigger DI indicates that the clusters are separated better and are more compact. SS is another technique for interpreting and validating the consistency of the clusters that provides information about which samples are located well within their respective clusters and which only lie somewhere between them. In other words, SS indicates if the cluster assignments are reasonable given the similarity (distances) between the samples. Values closer to 1 suggest that the samples are well-clustered. In contrast, values tending towards -1 imply that many samples are assigned to wrong clusters given the distances/similarity between them and the other samples in their assigned clusters.

DI and SS, however, do not show how meaningful the clusters are. For this purpose and the qualitative evaluation of the clusters, we use ARI to measure the similarity between the assigned cluster labels and the inferred true labels of the samples. It is computed by considering all pairs of samples and counting pairs assigned to the same or different clusters in the two partitions. In this work, we consider the method names to be a sort of fuzzy label for clusters. It is not unreasonable to assume that methods that provide similar functionality also tend to have similar names. Suppose we can find a cluster where most of the methods within that cluster have similar method names. In that case, we also have some reason to believe that these methods are functionally similar. ARI values closer to 1 signifies a very high similarity between the two cluster configurations. Values near 0 (or slightly below) show that the cluster configurations are very dissimilar.

[2]https://github.com/ai-center-kth/ml-code-dataset



(a) True labels (b) K-means

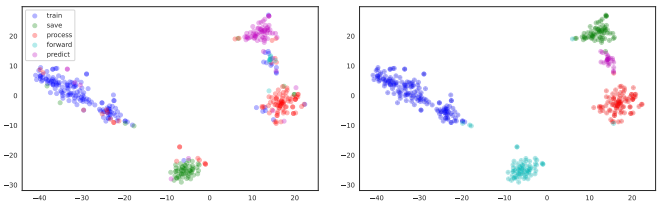
Fig. 3: Methods clusters using the baseline model.

C. Experiments and Results

In the rest of this section, we evaluate the clustering of the three CoCluBERT models using the metrics introduced in Section IV-B. For all the CoCluBERT models, we use the CuBERT large with 24 encoded layers, 16 attention heads, and a hidden layer size of 1024. We also set the maximum input sequence to 256. To visualize the resulting clusters, we apply t-Distributed Stochastic Neighbor Embedding (t-SNE) to reduce the dimensionality [41]. We also set the number of the clusters to be $K = 5$, the number of semantically different classes that the methods are chosen from (Table I).

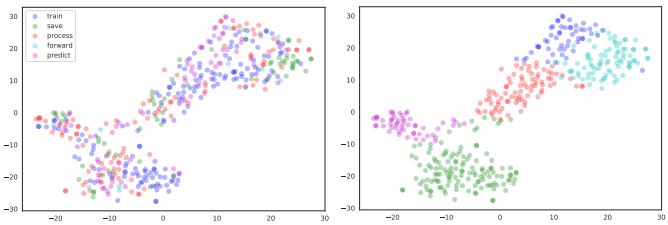
Baseline: As the baseline, we use CuBERT without fine-tuning for source code embedding, and then we give the embedding of [CLS] token to the clustering algorithm (K-means in our experiments). Table II shows the DI, SS, and ARI of the baseline model are 0.002, 0.339, and 0.057, respectively, and Figure 3a and 3b show the t-SNE of the distribution of method names and the result of clustering using the baseline model, respectively. As both Table II and Figure 3 show, the baseline model does not provide a clear separation among the clusters. We can see some weakly defined regions where methods with similar method names are located in the sample cluster, however, there is a fair bit of noise and overlap among these regions. So, CuBERT without fine-tuning does not appear to be well suited for clustering, highlighting the requirement for fine-tuning the model for this particular task. Below, we show how to fine-tune the CuBERT for clustering.

CoCluBERT-*Triplet:* In the first set of experiments, we evaluate CoCluBERT-Triplet by considering the Cosine similarity as the distance metric d in the triplet loss function (Equation 1). We trained the model using a batch size of four and with an initial learning rate 3×10^{-5}, which is reduced with a factor of 0.1 if the validation loss stopped improving



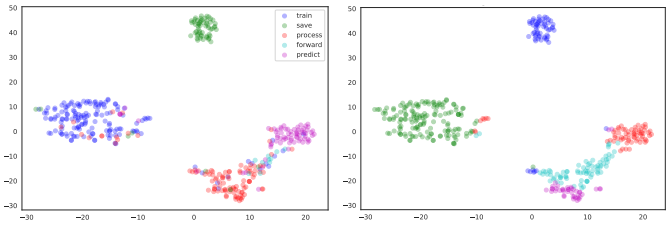
(a) True labels (b) K-means

Fig. 4: Methods clustering using CoCluBERT-Triplet.



(a) True labels (b) K-means

Fig. 5: Method clustering using CoCluBERT-Unsupervised.



(a) True labels (b) K-means

Fig. 6: Method clustering using CoCluBERT-DRC.

after three consecutive steps. Figures 4a and 4b depict the distribution of the method names and the obtained clusters. We use three different clustering techniques, K-means, OPTICS, and HAC, but in Figure 4b we show only the result of K-means. However, Table II shows DI, SS, and ARI for the three clustering techniques.

As Figure 3 shows, CuBERT without fine-tuning cannot learn an embedding to encode similarities among methods' functionality. However, in Figure 4 we see that fine-tuning CuBERT using the triplet loss function creates clusters more compact and well-separated. The clustering labels assigned by both K-means and HAC are very similar to the labels that we expect. However, OPTICS does not perform well, likely due to suffering from the curse of dimensionality. Looking at the quantitative evaluation metrics in Table II, we can see further evidence supporting that the clusters are meaningful. The high ARI scores that K-means and HAC achieved indicate a high level of accuracy concerning the method name labels. The SS provides further reason to believe that cluster assignments are reasonable given the distances between the data points (methods).

CoCluBERT-*Unsupervised:* Here, we fine-tuned CuBERT using the unsupervised loss function explained in Section III-B. We train the model for ten epochs, using a batch size of eight with the learning rate set to 3×10^{-5}. The hypothesis for fine-tuning CuBERT using the unsupervised loss function is that the weakly defined regions that we see in Figure 3 become more distinct by considering the initial clusters and the methods codes. Thus, better cluster separability and compactness could be achieved through the KL component of the loss function, while the MLM component would encourage CuBERT to learn better contextual representations of the words. However, in Figure 5 we see that the result is not exactly as we expect. We observe that the loss converges very quickly, indicating that the model already has a fairly good contextual understanding of the words used in the methods. This can likely be attributed to the narrow vocabulary used within machine learning code. Therefore, the MLM loss does not appear to impact the fine-tuning process significantly for our dataset.

We also note that the KL component of the loss function appears to have been fairly effective. We see that the resulting clusters are more compact and slightly more separable than those in the baseline model. However, these clusters do not appear to be meaningful concerning the method names, which is likely because the KL component of the loss pulls data points towards the closest centroid. However, it does so without considering the identities of the two samples. Again, the base assumption is that similar methods should have similar features. Although that is true for some of the methods shown in Figure 5, it is not the case for most of the methods, as many methods are pulled towards a cluster centroid where the majority of the methods do not have the same identity as the method being pulled.

TABLE II: The performance of each clustering method.

Embedding Model	**Clustering Algorithm**	**DI**	**SS**	**ARI**
CuBERT	K-means	0.002	0.339	0.057
CoCluBERT-Triplet	K-means	0.282	0.772	0.657
CoCluBERT-Triplet	HAC	0.277	0.753	0.643
CoCluBERT-Triplet	OPTICS	0.000	-0.085	0.012
CoCluBERT-Unsupervised	K-means	0.006	0.608	0.033
CoCluBERT-DRC	– –	0.003	0.527	0.492

CoCluBERT-*DRC:* Finally, we fine-tune CuBERT model using the DRC loss function as explained in Section III-C. We train the model for five epochs, using a batch size of four with the learning rate set to 3×10^{-5}. We also set $\lambda = 0.5$ in DRC loss function, and considert $T = 0.5$ for AF (Equation 7) and $T = 1.0$ for AP (Equation 8).

Table II demonstrates that CoCluBERT-DRC achieves significantly better clustering than the baseline model. We note that the DI is surprisingly low, given the good clustering. The ARI is also relatively high, indicating a significant overlap between the cluster assignments and the expected cluster labels. Similarly, we note that the SS is fairly

high, providing further reason to confirm that the cluster assignments are reasonable given the distances between the samples (i.e., methods functionality). Figure 6 shows the quantitative results, and we can see that the clusters from the CoCluBERT-DRC are considerably more compact and well-separated than that of the baseline model.

Discussion: Intuitively, the difference between the three fine-tuning frameworks is that both the CoCluBERT-Triplet and CoCluBERT-DRC use contrastive learning and enforce similarity preserving embeddings by considering the method names between samples in the dataset. This contrasts with CoCluBERT-Unsupervised that attempts to enhance the natural clusters, which already exist in the data without finding similar samples. From our results in this experiment, we conclude that CuBERT, when fine-tuned with the triplet loss function, is most suitable for clustering by functional similarities. We obtain compact and well-separated clusters with cluster assignments that are accurate concerning the inferred labels of functionality. The experiment results confirm that it is possible to learn source code embedding that encodes the functional similarities among source code snippets. As we observe, compared to the baseline model, clustering source code using CoCluBERT-Triplet and K-means has increased the DI metric by a factor of 141, the SS metric by a factor of two, and the ARI metric by a factor of 11. As we explained in Section IV-B, the bigger DI and SS values indicate that the clusters are better separated, and the samples are assigned more appropriately, respectively. Moreover, the bigger ARI scores indicate a higher level of accuracy concerning the method name labels.

V. Related work

Traditional source code embedding models have used AST. For example, Code2Vec [42] decomposes the source code as a bag of paths extracted from the code's AST and makes the representation by embedding and aggregating the paths into code vectors. Based on Code2Vec, Alon et al. present Code2Seq [5] that generates natural language representations from source code. TravTrans/PathTrans [13], ASTNN [43], and MTN [44] are others models that make use of AST for source code embedding. NCS [11] uses Word2Vec [23] to produce embeddings for both code and natural language. NCS is a model for searching code using queries. UNIF [10] is another work that learns the embedding of the codes and queries for searching codes. It uses the Cosine similarity between each query-code pair in the loss function.

BERT-based models are another family of models for source code embedding. CuBERT [14] is a model that uses BERT to learn contextual embedding from source code, and CodeBERT [45] is a bimodal pre-trained model for both natural language and programming language embedding. Fret [8] is another BERT-based model that generates documentation from source code. Unlike CuBERT and CodeBERT, Fret uses a reinforcer-transformer architecture to emphasize the function names that have a high probability of performing the task that their name represents.

Although there are several works on source code embedding, there is little work on clustering source code. Rousidis and Tjortjis [46] present a model that extracts entities and attributes from source code and uses HAC [34] to cluster them. Kuhn et al. introduce a model for source code comprehension [47]. Their model converts the source code into a term-document matrix and then applies Latent Semantic Indexing [48] to create document embeddings and finally clusters the documents. Theeten et al. propose Lib2Vec [3] to cluster source code by the libraries imported in the code. They adapt the skip-gram model from Word2Vec [23] to generate library embedding. To the best of our knowledge, our model is the first BERT-based model for source code clustering.

VI. Conclusions

In this work, we have explored the question *is CuBERT applicable to cluster source code by functionality?* We found out that using an out-of-the-box pre-trained CuBERT model does not produce good clusters, therefore we introduced CoCluBERT, based on CuBERT, and presented three learning models, CoCluBERT-Triplet, CoCluBERT-Unsupervised, and CoCluBERT-DRC that fine-tune CuBERT for creating clusters. After conducting experiments, we realized that CoCluBERT-Triplet and CoCluBERT-DRC could successfully group source code by functionality, while CoCluBERT-Unsupervised failed. We also observed that CoCluBERT-Triplet showed superior performance by showing that it can produce more compact, well-separated clusters, which correlated better with the true labels of the source code (the method names in our experiments). In summary, we have shown how to group source codes by functionality in an unsupervised way. We hope this work opens the door to more approaches that can help users understand and interpret undocumented source code.

Acknowledgment

This work was partly supported by the EC H2020 DataCloud[3] project under Grant Agreement No. 101016835.

References

[1] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and software technology*, vol. 49, no. 3, pp. 230–243, 2007.

[2] D. Rousidis and C. Tjortjis, "Clustering data retrieved from java source code to support software maintenance: A case study," in *Ninth European Conference on Software Maintenance and Reengineering*. IEEE, 2005, pp. 276–279.

[3] B. Theeten, F. Vandeputte, and T. Van Cutsem, "Import2vec: Learning embeddings for software libraries," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 18–28.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[5] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[3] http://datacloudproject.eu

[6] X. Chen, L. Ma, W. Jiang, J. Yao, and W. Liu, "Regularizing rnns for caption generation by reconstructing the past with the present," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7995–8003.

[7] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[8] R. Wang, H. Zhang, G. Lu, L. Lyu, and C. Lyu, "Fret: Functional reinforced transformer with bert for code summarization," *IEEE Access*, vol. 8, pp. 135 591–135 604, 2020.

[9] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[10] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "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*, 2019, pp. 964–974.

[11] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, pp. 31–41.

[12] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *arXiv preprint arXiv:1910.05923*, 2019.

[13] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[14] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.

[15] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[16] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[17] S. Huang, F. Wei, L. Cui, X. Zhang, and M. Zhou, "Unsupervised fine-tuning for text clustering," in *Proceedings of the 28th International Conference on Computational Linguistics*, 2020, pp. 5530–5534.

[18] H. Zhong, C. Chen, Z. Jin, and X.-S. Hua, "Deep robust clustering by contrastive learning," *arXiv preprint arXiv:2008.03030*, 2020.

[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[20] A. M. Dai, C. Olah, and Q. V. Le, "Document embedding with paragraph vectors," *arXiv preprint arXiv:1507.07998*, 2015.

[21] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[22] T. Hofmann, "Probabilistic latent semantic analysis," *arXiv preprint arXiv:1301.6705*, 2013.

[23] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*, 2013.

[24] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[25] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[26] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *arXiv preprint arXiv:1802.05365*, 2018.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[28] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *Science China Technological Sciences*, pp. 1–26, 2020.

[29] W. Yang, Y. Xie, A. Lin, X. Li, L. Tan, K. Xiong, M. Li, and J. Lin, "End-to-end open-domain question answering with bertserini," *arXiv preprint arXiv:1902.01718*, 2019.

[30] A. Cohan, I. Beltagy, D. King, B. Dalvi, and D. S. Weld, "Pretrained language models for sequential sentence classification," *arXiv preprint arXiv:1909.04054*, 2019.

[31] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA, 1967, pp. 281–297.

[32] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[33] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," *ACM Sigmod record*, vol. 28, no. 2, pp. 49–60, 1999.

[34] A. Lukasová, "Hierarchical agglomerative clustering procedure," *Pattern Recognition*, vol. 11, no. 5-6, pp. 365–381, 1979.

[35] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[36] J. Xie, R. Girshick, and A. Farhadi, "Unsupervised deep embedding for clustering analysis," in *International conference on machine learning*. PMLR, 2016, pp. 478–487.

[37] X. Guo, L. Gao, X. Liu, and J. Yin, "Improved deep embedded clustering with local structure preservation." in *IJCAI*, 2017, pp. 1753–1759.

[38] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[39] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1. IEEE, 2005, pp. 539–546.

[40] W. Hu, T. Miyato, S. Tokui, E. Matsumoto, and M. Sugiyama, "Learning discrete representations via information maximizing self-augmented training," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1558–1567.

[41] G. Hinton and S. T. Roweis, "Stochastic neighbor embedding," in *NIPS*, vol. 15. Citeseer, 2002, pp. 833–840.

[42] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[43] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[44] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, "Modular tree network for source code representation learning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–23, 2020.

[45] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[46] D. Rousidis and C. Tjortjis, "Clustering data retrieved from java source code to support software maintenance: A case study," in *Ninth European Conference on Software Maintenance and Reengineering*. IEEE, 2005, pp. 276–279.

[47] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and software technology*, vol. 49, no. 3, pp. 230–243, 2007.

[48] T. Hofmann, "Probabilistic latent semantic indexing," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, 1999, pp. 50–57.