DEGREE PROJECT IN MATHEMATICS,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

Deep Learning Approaches for Clustering Source Code by Functionality

MARCUS HÄGGLUND

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

Deep Learning Approaches for Clustering Source Code by Functionality

Marcus Hägglund

Degree Project in Mathematical Statistics (30 ECTS credits)

Master's Programme in Applied and Computational Mathematics

Supervisors at KTH: Amir H. Payberah, Martina Scolamiero

Co-supervisor at KTH: Francisco J. Peña

Examiner at KTH: Martina Scolamiero

External Partner: Research Institutes of Sweden (RISE)

Supervisors at RISE: Sepideh Pashami, Ahmad Al-Shishtawy

STOCKHOLM, SWEDEN 2021

Abstract

With the rise of artificial intelligence, applications for machine learning can be found in nearly every aspect of modern life, from healthcare and transportation to software services like recommendation systems. Consequently, there are now more developers engaged in the field than ever - with the number of implementations rapidly increasing by the day. In order to meet the new demands, it would be useful to provide services that allow for an easy orchestration of a large number of repositories. Enabling users to easily share, access and search for source code would be beneficial for both research and industry alike. A first step towards this is to find methods for clustering source code by functionality.

The problem of clustering source code has previously been studied in the literature. However, the proposed methods have so far not leveraged the capabilities of deep neural networks (DNN). In this work, we investigate the possibility of using DNNs to learn embeddings of source code for the purpose of clustering by functionality. In particular, we evaluate embeddings from Code2Vec and cuBERT models for this specific purpose.

From the results of our work we conclude that both Code2Vec and cuBERT are capable of learning such embeddings. Among the different frameworks that we used to fine-tune cuBERT, we found the best performance for this task when fine-tuning the model under the triplet loss criterion. With this framework, the model was capable of learning embeddings that yielded the most compact and well-separated clusters. We found that a majority of the cluster assignments were semantically coherent with respect to the functionalities implemented by the methods. With these results, we have found evidence indicating that it is possible to learn embeddings of source code that encode the functional similarities among the methods. Future research could therefore aim to further investigate the possible applications of the embeddings learned by the different frameworks.

Sammanfattning

Djupinlärningsmetoder för gruppering av källkod efter funktionalitet

Med den avsevärda ökningen av användandet av artificiell intelligens går det att finna tillämpningar för maskininlärningsalgoritmer i nästan alla aspekter av det moderna livet, från sjukvård och transport till mjukvarutjänster som rekommendationssystem. Till följd av detta så är det fler utvecklare än någonsin engagerade inom området, där antalet nya implementationer ökar för var dag. För att möta de nya kraven skulle det vara användbart att kunna tillhandahålla tjänster som möjliggör en enkel hantering av ett stort antal kodförråd. Att göra det möjligt för användare att enkelt dela, komma åt och söka efter källkod skulle vara till nytta inom både forskning och industri. Ett första steg mot detta är att hitta metoder som gör det möjligt att klustra källkod med avseende på funktionalitet.

Problemet med klustring av källkod är något som har tidigare studerats. De föreslagna metoderna har dock hittils inte utnyttjat kapaciteten hos djupa neurala nätverk (DNN). I detta arbete undersöker vi möjligheten att använda DNN för inlärning av inbäddningar av källkod i syfte att klustra med avseende på funktionalitet. I synnerhet så utvärderar vi inbäddningar från Code2Vec- och cuBERT-modeller för detta specifika ändamål.

Från resultatet av vårt arbete drar vi slutsatsen att både Code2Vec och cuBERT har kapacitet för att lära sig sådana inbäddningar. Bland de olika ramverken som vi undersökte för att finjustera cuBERT, fann vi att modellen som finjusterades under triplet-förlustkriteriet var bäst lämpad för denna uppgift. Med detta ramverk kunde modellen lära sig inbäddningar som resulterade i de mest kompakta och väl separerade klusterna, där en majoritet av klustertilldelningarna var semantiskt sammanhängande med avseende på funktionaliteten som metoderna implementerade. Med dessa resultat har vi funnit belägg som tyder på att det är möjligt att lära sig inbäddning av källkod som bevarar och åtger funktionella likheter mellan metoder. Framtida forskning kan därför syfta till att ytterligare undersöka de olika möjliga användningsområdena för de inbäddningar som lärts in inom de olika ramverken.

Acknowledgements

First I would like to express my deepest gratitude to my supervisor *Amir H. Payberah* for providing me with the opportunity to take part in the project at RISE and for his continuous and invaluable guidance throughout this semester. I want to direct special thanks to *Francisco J. Peña*, for his feedback and our many long and interesting conversations, ranging from concepts of deep learning to simple friendly banter. Furthermore, I would like to express my gratitude to *Sepideh Pashami* and *Ahmad Al-Shishtawy* at RISE for the help and suggestions that has gotten me through this project. I also want to thank my examiner *Martina Scolamiero* for her continuous encouragement and feedback. She has been a much needed voice of reason in this project. Last but certainly not least, I would like to extend a most heartfelt thank you to my friends and family that have given me an unwavering support throughout the years.

Marcus Hägglund
Stockholm, May 2021

Contents

1 Introduction **1**
1.1 Research Question . . . 2
1.2 Goal . . . 2
1.3 Problem . . . 2
1.4 Stakeholders . . . 3
1.5 Scope . . . 3
1.6 Outline . . . 3

2 Background and theory **4**
2.1 Abstract Syntax Tree . . . 4
2.2 Embeddings . . . 6
2.3 Transformer and BERT . . . 7
2.3.1 Attention Mechanism . . . 7
2.3.2 Positional Encoding . . . 9
2.3.3 Encoder and Decoder . . . 10
2.3.4 Transformer Architecture . . . 11
2.3.5 BERT . . . 11
2.4 Clustering Techniques . . . 13
2.4.1 K-means Clustering . . . 14
2.4.2 DBSCAN and OPTICS . . . 14
2.4.3 Hierarchical Agglomerative Clustering . . . 16
2.5 Metrics for Clustering Evaluation . . . 17
2.5.1 Dunn Index . . . 17
2.5.2 Silhouette Coefficient . . . 18
2.5.3 Adjusted Rand Index . . . 19
2.5.4 Cosine Similarity . . . 20
2.6 Visualizing High Dimensional Data . . . 20
2.7 Related Work . . . 22

3 Methods **23**
3.1 Code2Vec . . . 24
3.1.1 Clustering Code2Vec Method Representations . . . 25
3.1.2 Clustering Augmented Code2Vec Method Representations . . . 25
3.2 cuBERT . . . 28
3.3 Fine-tuning cuBERT . . . 28
3.3.1 Triplet Loss . . . 28
3.3.2 Unsupervised Fine-Tuning . . . 30
3.3.3 Deep Robust Clustering . . . 32
3.4 Experimental Design . . . 34
3.4.1 Dataset . . . 34
3.4.2 Process . . . 34
3.4.3 Selecting Cluster Configurations . . . 35
3.4.4 Cluster Analysis . . . 36

4 Results **37**
4.1 Learning and Evaluating Code2Vec Embeddings . . . 37
4.1.1 Baseline . . . 37
4.1.2 Augmented Code2Vec . . . 40

4.2 Learning and Evaluating cuBERT Embeddings . . . 44
4.2.1 Fine-Tuning with Triplet Loss . . . 44
4.2.2 Unsupervised Fine-Tuning of cuBERT . . . 49
4.2.3 Deep Robust Clustering with cuBERT . . . 51

5 Conclusion **54**
5.1 Summary . . . 54
5.1.1 Code2Vec . . . 54
5.1.2 cuBERT . . . 55
5.2 Limitations . . . 56
5.3 Future Work . . . 57

References **58**

Chapter 1

Introduction

The field of artificial intelligence (AI) is experiencing an ongoing boom. It has gone from being a relatively obscure technology into being a part of everyday modern life, basically finding its way into almost every industry. The rate of progress in the field is staggering. It almost seems as if new research is being published on the daily. Consequently, there are now more people involved in the field of AI than ever. Both the number of developers and the different types of implementations are rapidly increasing. To help further the progress made in the field, machine learning code needs to be more accessible and understandable for a larger audience. It is therefore interesting to be able to provide an ecosystem that makes it easy to share, access and search for machine learning code. An important step towards creating this system is to be able to categorize machine learning code by the functionality that they implement.

Software clustering techniques is one of the possible methods that can be used to help manage and understand large repositories of source code. By being able to group source code according to similarities, it is easier to find and manage relevant pieces of source code. Software clustering techniques have been studied previously in the literature ([19], [27]). In general, the idea is to decompose large projects into smaller components. For instance, a large project can be viewed as the set of methods declared within it. Each method then constitutes an important piece of the software puzzle. Once the smaller components have been extracted, we can start to look at the different features in order to sort and group the methods into clusters. Many of the reported works tend to leverage information about the source code, for instance file meta data, documentation and imported packages. Once the set of features have been decided upon, one can start to look for patterns in order to identify meaningful groups of source code. In this work, we aim to to investigate the possibility of using deep neural networks to cluster source code by *functionality.*

In order to capture similarities between code snippets, one can use techniques such as *embeddings.* Embeddings have been used extensively in *Natural Language Processing* (NLP) to teach a model information about useful semantic relationships in the data. More recently, embedding techniques have also been applied to source code. It is therefore interesting to investigate whether embeddings of source code are capable of encoding the functionality of the code snippets. Such embeddings would have many potential uses, ranging from code summarization, code search and code generation to name a few. It is then also possible to apply clustering techniques on the learned embeddings of source code, in order to identify groupings that correspond to the implemented functionality.

There have been a few works presented in the literature that describe methods for learning useful embeddings of source code, which are then evaluated for a variety of downstream tasks ([1], [15]). In addition to this, there have also been a some works that detail different approaches for clustering source code ([19], [27]). Further detail about these works will be presented in section 2.7 after the necessary preliminaries have been given. These approaches have however not attempted to leverage the capabilities of deep neural networks. To that end, we believe that it is worthwhile to investigate the potential of such models for the purpose of clustering source code. To the best of our knowledge, there has not been any previous work that aim to learn embeddings of source code that encode functional similarities between methods for the purpose of clustering.

In this thesis we investigate two models, Code2Vec[1] and cuBERT[15], for the stated problem. The main contributions of this work are as follows:

- We investigate the suitability of Code2Vec for learning representations of source code that reflect functional similarity. Experimentally, we demonstrate in section 4.1.1, that the *code vector* method representation from Code2Vec does not reveal any significant cluster structure. We are however able to identify semantically meaningful clusters, by instead using the code vectors to predict the method name and then using the corresponding embedding for the predicted method name as the method representation.
- We identify three different frameworks for fine-tuning cuBERT for this particular task; a *triplet* framework, an *unsupervised* framework and a *deep clustering* framework. The results in section 4 indicate that the self-supervised approaches, the triplet framework [7] and the Deep Robust Clustering [37] framework, are better suited for the problem statement when compared to the unsupervised framework. We achieved the most compact and well-separated clusters when fine-tuning cuBERT in the triplet setting. Our results suggest that it is possible to learn embeddings of source code that encode functional similarities between methods.

1.1 Research Question

The main research questions of this thesis can be formulated as follows:

Can we learn embeddings of source code that encode functional similarities between methods? Can such embeddings be used to cluster methods by functionality?

1.2 Goal

The purpose of this work is to investigate if it is possible to learn representations of source code for the downstream task of clustering code snippets by their functionality. To achieve this, a suitable source code embedding technique must be identified, which is capable of encoding similarities between code snippets. In particular, we aim to learn and evaluate such embedding techniques on machine learning source code. What follows is an empirical approach where we evaluate different models, frameworks and clustering techniques to quantitatively and qualitatively evaluate the semantic quality of the clusters found within the learned embedding spaces.

1.3 Problem

A feature of machine learning code is that it is structurally very similar across different projects. Consider the following illustration of a high-level view for the typical machine learning pipeline.

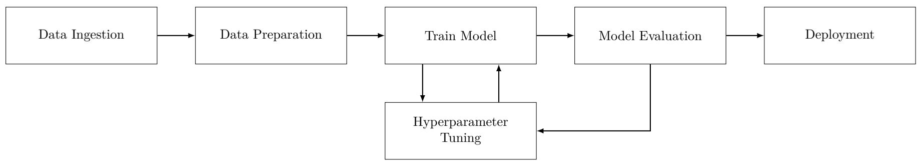


Figure 1.3.1: Machine Learning Pipeline

In each step of this process, we expect to find methods that are related to the particular task involved. For instance in the step of preparing the data for the model there are typically methods responsible for extracting and scaling features. In the model creation step there are methods that define and build the layers of the model etc. One may then consider that each step in this pipeline serves a core functionality and that this is one way of categorizing methods by the functionality that they provide in the project. There are many considerations and difficulties to address with the problem definition. For instance, there are many different ways to write a method that provides the same functionality. To demonstrate this, let us consider the following example.

```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) Method 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) Method B

Figure 1.3.2: An example of two similar code snippets.

The two methods in Figure 1.3.2 are different implementations of methods in Python that provide the same functionality; namely to create generators for the dataset. While this is relatively easy for an experienced developer to see just from the source code, the same cannot be said for a model that faces the same task. A key problem to address is therefore: *how can we model the functionality of source code in such a way that a mathematical model may come to the same conclusion?*

1.4 Stakeholders

This project is conducted at KTH Royal Institute of Technology in collaboration with Research Institutes of Sweden (RISE). None of the stakeholders are affected by the outcome of this project, nor are there any conflicts of interest.

1.5 Scope

First and foremost, the main objective is to find embeddings of source code that reflect the implemented functionality of the code. Embeddings of source code that encode the functional similarities could be useful for a wide variety of different tasks. For instance they could be valuable additions to existing models and frameworks that aim to learn tasks like code search, code summarization and code generation to name a few. However, in this thesis we will evaluate this embedding property through the use of clustering techniques. The purpose is to identify meaningful clusters of methods in the embedding space, with respect to the functionality that is implemented by these methods.

1.6 Outline

This thesis is outlined as follows. In Chapter 2, the theoretical background is presented together with the necessary preliminaries to help the reader to better understand the methods and models used in the thesis. Chapter 3 describes the different methods, models as well as the approaches that were used to obtain the results of this thesis. In Chapter 4 the results from the different models are presented together with an in-depth discussion. In Chapter 5 the thesis is concluded and we provide some suggestions for future research.

Chapter 2

Background and theory

In this section, the necessary background information related to the subject will be presented. First, we will explain one of the most commonly used structures for representing source code. This is then followed by a brief description of what embeddings are and how they are applicable for source code. We then introduce the transformer model and its inner workings, which is necessary for getting a good understanding of BERT and cuBERT models. With the necessary preliminaries given regarding the source code and the embeddings, we then provide a detailed descriptions of the different clustering techniques and evaluation metrics that we will consider in this work. The chapter is then concluded by explaining how we will visualize the high dimensional data.

2.1 Abstract Syntax Tree

Abstract Syntax Tree (AST) is a a tree-like structure that is used to represent the hierarchical syntax of source code. The tree-like structure, simply referred to as a *tree*, is a commonly used data structure in computer science and is a concept that has its origin in graph theory.

Definition 2.1.1 (Tree). *Let $G = (V, E)$ denote a graph that consists of a set of of vertices V and a set of edges E. If G is a connected graph without any cycles, then G is called a tree [35].*

If G is a tree, then the following are equivalent.

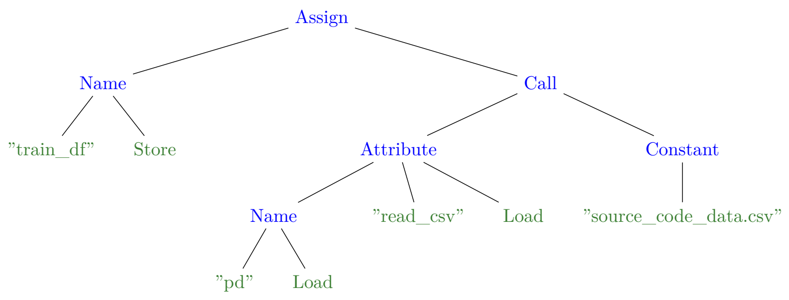
- G has no cycles.
- For every pair of vertices $u, v \in G, u \neq v$, there exists exactly one unique path between u and v.
- Removing *any* edge from G results in a graph that is not connected.
- If G has N number of vertices, then there are exactly $(N - 1)$ edges between the vertices in G.

A vertex, also referred to as *node*, is a data structure that contains a value and a list of references (edges) to other nodes in the graph. The connections represent the hierarchy in the graph and the direction of these connections are therefore very important. A node that has *emerging* connections is said to have *child nodes*. A node that has an *incoming* connection is said to have a *parent* node. For convenience, the notion of the *degree* of a node is introduced and it is defined as the number of parent connections that a node has. The tree has a special type of node, known as the *root*. This node represents the very origin of the tree, meaning that it has no parent connections and therefore a degree of zero [35]. Having understood the tree structure, we may now consider how it applies to source code.

The abstract syntax tree is constructed from the statements and expressions declared in the code. The tree is a collection of nodes which are linked together to represent the hierarchy. All of the nodes in the tree have single parent node, except the root (top node). Each node in the tree represents one of the constructs that occur in the code. Examples of such constructs are: assignments, expressions and if-then-conditions to name a few. A concrete example of an AST is shown in Figure 1.3.2.

```
train_df = pd.read_csv("source_code_data.csv")
```

(a) Code Snippet



(b) AST

Figure 2.1.1: Example of an AST for a code snippet.

A more formal definition of the Abstract Syntax Tree is as follows:

Definition 2.1.2 (Abstract Syntax Tree). *An Abstract Syntax Tree (AST) for a code snippet C is a tuple $\langle N, T, X, s, \delta, \phi \rangle$, where N is a set of nonterminal nodes, T is a set of terminal nodes, X is a set of values and $s \in N$ is the root node. Here, $\delta : N \to (N \cup T)^*$ is a function that maps a nonterminal node to a list of its children and $\phi : T \to X$ is a function that maps a terminal node to an associated value [1].*

The abstract syntax tree is read as a depth-first traversal. It starts at the root and then the child nodes are visited recursively in a left-to-right order. To illustrate this, consider Figure 2.1.1b. The traversal starts at `Assign`, then goes down the left branch to find the name of the assignment variable. It then goes back up to the right branch of `Assign` to find the value of the assignment that is obtained through a method call. The traversal proceeds from `Call` down to `pd` and then back up again to the right branch of `Attribute`. This traversal pattern is repeated recursively until all of the nodes have been visited. While the example of an AST in Figure 2.1.1b is a for a very simple expression, it highlights the important features of the AST.

The syntax tree is abstract in the sense that it only represents the context and the syntactic structure of the syntax. Some details that exist in the original piece of source code, like parentheses and brackets are not included in the representation. One can of course use ASTs to express much more complex snippets of code than in the example above, however it becomes increasingly difficult for a human to understand the entirety of the semantics in the tree representation.

ASTs are most commonly used by compilers where a parser produces ASTs to represent the syntactic structure of the source code. The main purpose of this is to serve as an intermediate representation for the compiler and is used during the many steps of compilation, like the intermediate code generator. They have also been used in many other applications that are secondary to its main purpose within compilers. With the increased interest of applying machine learning algorithms to source code, they have been proven to be suitable representations of source code for a large variety of tasks in source code analysis.

2.2 Embeddings

In order to find similarities between code snippets, it is possible to use *embeddings*. It is a data-driven technique in which an object that is possibly non-numeric and discrete, is mapped to a continuous real-valued vector. This vector is referred to as the embedding of the object and it is what makes it possible for a machine learning model to work with data that is essentially non-numerical. The arguably most well known type of embedding techniques are within the realm of natural languages. Continuous space language models have been proven to be very powerful, producing remarkable results for various tasks in NLP. These types of representations have been shown to be able to encode semantic similarities and analogies between words. Intuitively one can consider that, in this space the "meaning" of an object is distributed among the vector components in such a way that semantically similar objects are mapped close to one another [20]. This makes it possible to use simple algebraic operations on the word vectors to perform reasoning. In the most famous example of this property, it was shown that $vec("King") - vec("Man") + vec("Woman")$ results in a vector that is closest to the vector for the word *Queen* [21].

Embeddings for Source Code

Adapting embedding techniques that were originally designed for words and sentences in natural languages, into techniques for variables and statements in programming languages is not too far fetched. Writing code is a form of human communication; it is created by and shared between humans. Both natural languages and programming languages can be very rich and complex, with many different expressions. But humans are lazy; in practice we like the simplicity of things. Therefore, most of our communication is by and large both repetitive and predictable. These statistical properties have been well-studied and exploited for natural languages to build the statistical models that are now the foundation of natural language processing. In 2012, Hindle et al. [10] was able to provide empirical evidence that supports that programming languages share statistical properties that are similar to those of natural language. Programming languages are in fact even more repetitive than natural languages. This can likely be explained by the fact that coding is cognitively demanding task, and humans are once again lazy beings. Code reuse is therefore highly prominent in source code, since it is more efficient to reuse rather than re-invent new methods.

From the success of Word2Vec, a large variety of novel embedding techniques have emerged. In addition to this, the evidence provided by Hindle et al. further encouraged researchers to develop techniques that are particularly well suited for programming languages. Many of the embedding techniques that are presented in the literature tend to leverage the structural information of source code in order to obtain the embeddings. Popular choices for the input of the source code to the machine learning model are AST and Control Flow Graphs (CFG). This is because these representations encode the important syntactical and relational properties of the code. In Code2Vec [1], a set paths in the AST are used to derive an embedding of a code snippet. Xu et al. [36] train a neural network to learn embeddings of source code that are based on the CFG representation of the code. There have also been works that aim to learn embeddings of source code using only the surface text, without constructing any secondary structures like the AST or CFG. One example of this is a model known as cuBERT, presented by Kanade et al. [15]. It is a model that learns contextual embeddings of source code following this approach.

Tokenization

Tokenization is one of the most fundamental methods in natural language processing and it is a necessary step for creating embeddings for textual data. It is a technique for breaking down a piece of text into smaller components called tokens. A token can be a sentence, a word (or part of a word) or even just plain characters. Tokenization is applied to an entire text corpus in order to extract the unique tokens that it contains. A vocabulary can then be formed, by mapping each unique token to its own number. This step effectively transforms textual data into numerical, making it possible for a machine learning model to process the data. A sequence of text can now be given as input to the model, by first tokenizing it and then looking up the unique indices corresponding to the tokens in the sequence. The end result is a numerical representation of the textual input. Numerous learning techniques can then be applied in order to teach the model to understand the context and meaning of words and sentences.

2.3 Transformer and BERT

Transformers is a model introduced by Vaswani et al. [34] that is particularly well suited for language understanding. Transformers use an *attention mechanism* in an *encoder-decoder* structure to learn a mapping between an input sentence and a target sentence. A concrete example of this is the problem of translating a sentence in English into its French counterpart. In this scenario, we can think of both the encoder and decoder as translators. The encoder would first translate the sentence in English into some "abstract language" that both the encoder and decoder understands. Then, the decoder can translate the sentence from this abstract language into French.

The translation problem is not simply a one-to-one mapping between words. Languages are rich and the meaning of words depends on the context in which they appear in. As such, we have to consider all of the words in the sentence to learn the meaning and context. This is what the attention mechanism is used for. It allows the model to consider all words at once and learn important relationships between them. The approach taken with the transformer model is in stark contrast to most of the preceding natural language models. For instance, models like Recurrent Neural Networks need to process the sentence sequentially in order. Such models therefore have restricted access to the words in a sentence. When computing a representation of a word, it only has knowledge of the previous words and is completely oblivious to the succeeding words. Intuitively, we can easily understand why this is detrimental when attempting to gain a deep understanding of the context and meaning of words. Using the attention mechanism, this problem is alleviated, allowing the model to learn long range dependencies between words to build better contextual representations [34].

Another perk of the attention mechanism is that it allows for the model to be parallelized. This means that the model can be trained in a parallel manner, reducing the overall training time. It also makes it more feasible to train on much larger datasets. Summarizing the main features of the transformer it can be said that it is capable of learning better contextual representations of words and that it is more efficient to train. It is therefore easy to see why it a model that is highly suitable for language understanding problems. It has in fact achieved state-of-the-art results on numerous machine translation tasks [34]. The success of the transformer has since led to the development of new models that use the encoder-decoder blocks of the transformer.

2.3.1 Attention Mechanism

To understand exactly how transformers work, we must first understand the *attention mechanism.* Like many concepts in machine learning, the idea is inspired by biology. The attention mechanism mimics the cognitive attention mechanism of the brain and is therefore a very intuitive concept. It can simply be thought of as directing more focus to specific parts of the input when processing the data. There are a few types of attention mechanisms described in the literature, which slightly differ in their workings. However, in this thesis we will only cover two types, *self-attention* and *multi-headed attention.*

Self-Attention

Self-attention is an attention mechanism that relates the different elements of an input sequence *within* itself in order to compute the representations. To make this idea concrete, let us consider the following sentence as an example, as given by Getting Started With Google BERT [25].

A dog ate the food because it was hungry

In the sentence above, the pronoun *it* could refer to either *dog* or *food.* After reading the entire sentence, we can easily understand that *it* refers to the *dog.* With the self-attention mechanism, we mimic the process that we humans just subconsciously performed, in order to reach the same conclusion. For each word, the model computes a representation of it, by attending to all of the other words in the sentence. This is visualized in Figure 2.3.1, where the thickness of the line corresponds to the attention weight, or the relative importance that were assigned between the words. The attention weights will therefore also provide us with some insight to the inner workings of the model. It provides explainability since it is easy to inspect the weights and see which of the other words that were most attended to when computing a contextual representation for a word.

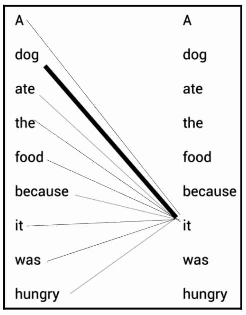


Figure 2.3.1: An example of Self-attention. The thickness of the line corresponds to the relative importance between the words, with a thicker line signifying a greater importance [25]

We have now briefly introduced the concept of the attention mechanism and are now ready to dive in to the mathematics behind it. In a more abstract sense, we can consider the attention mechanism as a mapping between a query vector $\mathbf{q}$ and a set of key-value vector pairs $\mathbf{k}$ and $\mathbf{v}$. The query vector is the embedding of the word that is currently considered. The key is the embedding of another word that the query will be scored in relation to and the value vector is the embedding of the word that corresponds to the key. In most cases, the key and value vectors are identical ($\mathbf{k} = \mathbf{v}$), however in theory each key can have multiple values. In this thesis we will consider the key and value to be identical at all times.

Since we would like to consider all of the words in the sentence at once, we can construct matrices for the set of queries, keys and values. These matrices are obtained from the input sentence. Let X be the matrix representing the input sentence, where each row vector corresponds to the word embedding of each word in the sentence. Furthermore, let W^Q, W^K and W^V denote the three weight matrices of the attention layer. These weight matrices starts of with randomly initialized weights that are then learned during training. Using these weight matrices, the query, key and value matrices are computed by multiplying the input matrix X with the corresponding weight matrices. Let the resulting query, key and value matrices be denoted by $Q, K \in \mathbb{R}^{N \times d_k}$ and $V \in \mathbb{R}^{N \times d_v}$. Here, N is the number of input tokens and d_k and d_v are the dimensionalities of the key and values respectively.

The attention weights are then calculated as follows. First, the dot product between Q and K^T is computed, resulting in a matrix that essentially tells us how similar each word in the sentence are to the other words. The values of this similarity matrix are then divided by the square root of the dimension key d_k in order to obtain more stable gradients. In the next step, the softmax function $\sigma(\mathbf{x})_i = e^{\mathbf{x}_i} / \sum_{j=1} e^{\mathbf{x}_j}$, is applied to the scaled similarity matrix. This function normalizes the similarity scores, bringing them into the range between 0 and 1. Finally, the attention matrix is obtained by multiplying the previous result with the value matrix V. The end result is the attention matrix that contains the self-attention values for all of the words in the sentence [34]. Mathematically, we can express this through Equation 2.1.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$

Multi-Headed Attention

In the previous section, we learned how a single self-attention head works. The transformer expands upon this concept by extending the mechanism to multiple attention heads. This is referred to as *multi-headed attention* and was presented by Vaswani et al. [34]. They found that it improves upon the self-attention in two ways. First, the model expands on its ability to attend to multiple words positions. With single-headed attention, the dominating focus for each word will be placed mainly upon itself, while the importance of other words are toned down in relation. Multiple attention heads allows for more focus to be placed on different word positions. Secondly, it enables the model to attend to multiple "representation subspaces", as described by Vaswani et al. [34]. With single-headed attention, one area of the input sequence is going to be more attended to than the others. By instead using multi-headed attention, the model has the ability to attend to even more areas simultaneously.

Multi-headed attention uses the same mechanism described in the previous section, but the computations are performed by multiple heads simultaneously. Let h be the number of attention heads. Each attention head has its own set of weight matrices W_i^Q, W_i^K and W_i^V where $1 \leq i \leq h$. The queries, keys and values are projected h times using the weight matrices of every head. As a result, each head computes an attention matrix Z_i according to Equation 2.1. The attention matrices Z_i of each attention head are then concatenated into a single attention matrix Z. This matrix is then subsequently multiplied with the weight matrix W^O to represent the final multi-headed attention matrix. Here W^O is a learnable weight matrix of the multi-headed attention layer. Vaswani et al. mathematically define the multi-headed attention mechanism through Equation 2.2:

$$MultiHead(Q, K, V) = Concat(head_1, \ldots, head_h)W^O \tag{2.2}$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$. It should be noted that for computational reasons, the dimensionality of the keys and values in the multi-headed attention layers are reduced by the number of attention heads. Specifically, the dimensions are given by $d_k = d_v = d_{model}/h$ where d_{model} is the maximum input sequence length. The concatenation operation therefore returns an output that has the same dimensionality as the input. The reduced dimensionality of each head also means that the total computational cost of the multi-headed attention is very similar to that of the single-headed attention that uses a full dimensionality representation [34].

2.3.2 Positional Encoding

The attention mechanism allows the transformer to process all words in the input sequence at once, rather than in a sequential order. While there are many advantages to this mechanism, it does also come with its drawbacks. Because the words are not processed in a sequential order, the model has no inherent notion of the ordering of the words that it sees. This property can be seen in Equation 2.1. The dot products between the query and key vectors are computed equally without making any considerations for the ordering in the input sequence. As such, the model effectively only considers the input to be an unordered set of vectors and is therefore completely unaware of the ordering. Naturally, this is detrimental for learning to understand any language since the ordering of the words are of utmost importance. In order to alleviate this problem, Vaswani et al. [34] introduce a *positional encoding* vector that encodes the relative position of the words in the input sequence. This vector has the same dimension as the input embedding and is added to each input word embedding. This vector contains positional information about the words, represented by values between -1 and 1 that are derived from the sinusoidal wave functions in Equation 2.3.

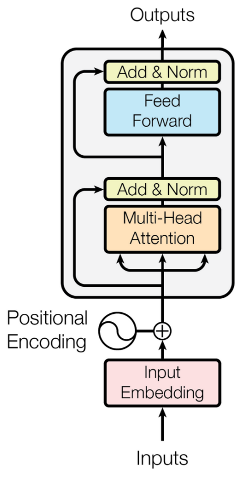
$$\begin{aligned} PE_{(pos,2i)} &= sin(\frac{pos}{10000^{2i/d_{model}}}) \\ PE_{(pos,2i+1)} &= cos(\frac{pos}{10000^{2i/d_{model}}}) \end{aligned} \tag{2.3}$$

where *pos* refers to the position of the word in the sequence and i is the dimension of the word embedding. The intuition here is that the values produced by these wave functions provide meaningful distances between the words that the model can recognize. For any fixed offset k, $PE_{(pos+k)}$ can be represented as a linear function of PE_{pos}, which allows the model to attend to relative positions [34].

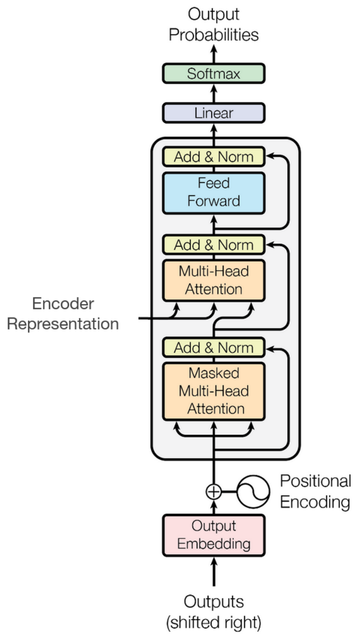
2.3.3 Encoder and Decoder

In this thesis, we will mainly focus on the encoder layer of the transformer. We have therefore omitted some of the intricacies related to the decoder layer. For the interested reader, we suggest reading a detailed explanation of the decoder layer found in Chapter 1 of Getting Started With Google BERT [25].

A single encoder block of the transformer is depicted in Figure 2.3.2a. First, each word in the input sentence is embedded in order to obtain the input matrix X. Next, the positional encoding matrix P is computed and added to the input matrix before feeding it as input to the encoder. The multi-headed attention block then attends to the input, returning an attention matrix. What follows is the add and norm layer, which is a two step process. The original input is added to the attention matrix using an residual connection as shown in Figure 2.3.2a. Layer normalization is then subsequently applied to normalize the resulting matrix. The layer normalization leads to faster convergence when training by preventing the output values in each layer from drifting away too much over the different batches. The resulting matrix from the add and normalize operation is then passed as input to the feed-forward layer. A last add and normalization operation is then applied to obtain the final encoder representation [34].



(a) A single encoder block of the transformer.



(b) A single decoder block of the transformer.

Figure 2.3.2: The encoder and decoder blocks of the Transformer [34].

An illustration of the decoder block is shown in Figure 2.3.2b. The architecture of the decoder block is similar to that of the encoder. While both have a multi-headed attention layer and a feed-forward layer, the decoder also has a secondary multi-headed layer that is masked. The decoder has two inputs. First, the target sentence is embedded and the positional encoding is computed and added to the target sentence embedding. This is then fed as input to the *masked multi-head attention* layer. The second input to the decoder is the representation of the input sentence that the encoder computed. Both of these inputs are then fed to the multi-headed attention layer and subsequently the feed-forward layer to generate a decoder representation. The decoder representation is then finally passed through a fully connected layer, applying the softmax activation function to generate a prediction.

2.3.4 Transformer Architecture

The architecture of the Transformer is as follows. It consists of N encoder and decoder blocks stacked on top of one another as shown in Figure 2.3.3. Each encoder block processes its input data to generate a representation of it, where the encoder tries to only keep the most important features. The representation of each encoder block is passed as input to the next encoder block. The decoders on the other hand, generate an output using the information that is distilled in *all* of the representations produced by the encoder blocks. Each decoder block is also able to access the outputs from the previous decoder. Both the encoder and decoder layers utilize the multi-headed attention mechanism, in which the relevance of the components of each input are weighed. The weights of each sub-layer in the encoder and decoder blocks are updated based on the resulting loss of the objective function during training.

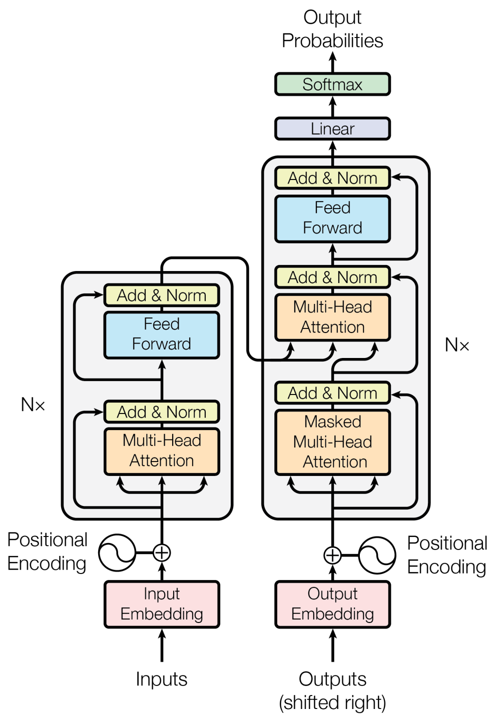


Figure 2.3.3: The transformer model architecture [34]. The leftmost block illustrates the architecture of one encoder layer. The rightmost block illustrates the architecture of one decoder layer.

2.3.5 BERT

Bidirectional Encoder Representations from Transformers (BERT) is a state-of-the-art embedding model, developed by Devlin et al. [6] for various NLP tasks. As the name suggests, it is a model based on the encoder blocks of the transformer. BERT is a deep neural network that is pre-trained in an unsupervised setting to learn representations of text. BERT learns contextual representations of words in a bidirectional manner by considering the context of each word in a sentence. This design choice is in stark contrast to most of the pre-existing NLP algorithms. These models generally read text sequentially in a single direction. Additionally, many of the language models, like Word2Vec, considered words to be context-free entities. The bidirectional property of this model means that for each word, it will not only consider the preceding words in a sentence but also the succeeding words. This behavior is more in line with how humans tend to read text. As you are reading this sentence now you're not only considering the previous words but you're also subconsciously looking at the words ahead. Devlin et al. [6] demonstrated in their paper that a bidirectionally trained network can learn a deeper understanding of the context and structure of a language than their unidirectional counterparts. To further understand why a context-based word embedding model is superior to a context-free embedding model, like Word2Vec, let us consider the following example.

*Sentence A: The dog loves to **bark** at the mailman.*
*Sentence B: The **bark** of the tree is rough to the touch.*

We can understand that there is a different meaning to the word 'bark' in the two sentences above. This is because of the context in which the word appears. In sentence A, 'bark' refers to the sound a dog makes, whereas in sentence B it refers the to the outer covering or skin of the tree. Context-free models like Word2Vec are not able to capture the difference in the meaning of the word since it produces the same exact embedding for the word, regardless of the context in which it appears. Context-based models like BERT will instead relate each word with all of the other words in the sentence to produce the embeddings. It would therefore be able to understand that the word 'bark' in sentence A refers to a sound rather than the skin of the tree, because it appears together with the word 'dog' in the same sentence. Because BERT is able to look ahead in sentence B, it is able to see that the word 'bark' co-occurs with the word 'tree' in same sentence, and builds the word representations accordingly. This illustrates why a bidirectional context-based embedding model is able to more accurately capture the meaning of words.

BERT Architecture

The architecture of BERT is based on the encoder structure of the Transformers model, introduced by Vaswani et al. [34]. In the paper by Devlin et al. [6] they present two model configurations, BERT-base and BERT-large. In this thesis we will use BERT-large and therefore only cover that configuration. The only difference between the two configurations is the number of encoder blocks that are used in the model. BERT-large features 24 encoder blocks with 16 attention heads. The feed-forward layer of each encoder has 1024 units, which is therefore the size of the representations generated by BERT-large. As a result, the total number of trainable parameters in the large BERT configuration is 340 million [6].

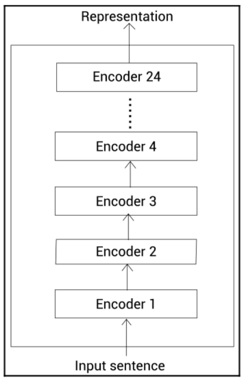


Figure 2.3.4: The architecture of BERT-large [25]

Pretraining
BERT is pre-trained on a large corpus of text in an unsupervised setting. When pre-training, the model has two different objectives to learn, *Masked Language Modeling* (MLM) and *Next Sentence Prediction* (NSP). In MLM, a percentage of the words in the input sentence are masked (hidden) from the model by replacing the word with the token `[MASK]`. The model is then faced with the task of predicting the original word of the masked token, based on the context of the other non-masked words in the sentence. The goal with this task is to train the model to understand the relations between words depending on the context in which they are used within a sentence. The second task when pre-training BERT is NSP. BERT is given two sentences, A and B, as input. The sentences are embedded by the model and the objective is to make BERT predict whether sentence A is followed by sentence B. The sentences are chosen in such a way that sentence B is the next sentence in only 50% of the cases. In the other instances it is randomly sampled from another text source. The goal with this task is to teach the model to understand the relationship between sentences in a longer text [6].

Fine-tuning
Once BERT has been pre-trained it can be applied to various downstream tasks, such as question-answer tasks, sentence classification and sentence similarity to name a few. This is done by adding an extra output layer to the pre-trained model that is task specific. The parameters of the pre-trained BERT can then be fine-tuned for the downstream task to achieve better results [6]. Figure 2.3.5 shows the difference between the BERT architecture when pre-training and fine-tuning the model. The downstream tasks shown in the figure illustrates some of the downstream tasks that the performance of the fine-tuned BERT was evaluated for.

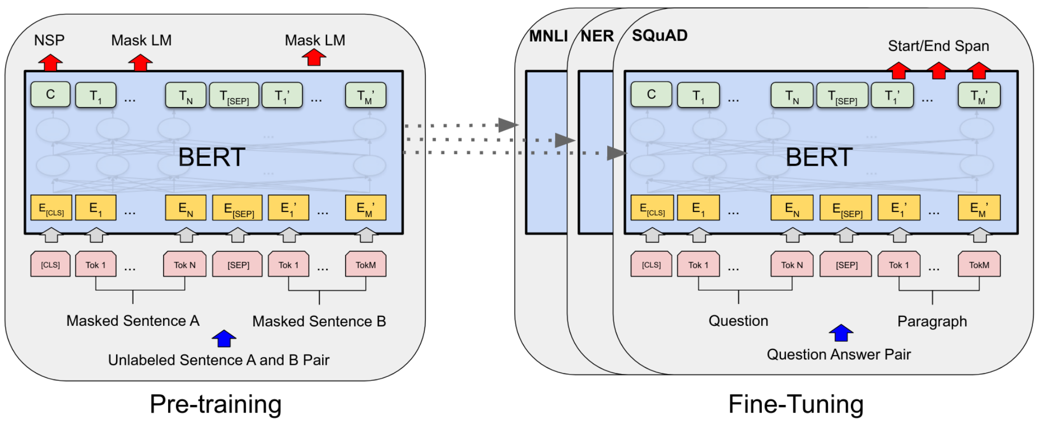


Figure 2.3.5: An overview of the pre-training and fine-tuning of BERT. In both scenarios the models have identical architectures except for the output layer. When the model is fine-tuned, an extra output layer is added to the model that is specific to the downstream task [6].

2.4 Clustering Techniques

Let us now consider the different techniques and metrics that we will use for the cluster analysis of the embeddings learned by the models. Cluster analysis is a class of techniques for exploratory data analysis and is commonly used in a wide variety of different fields, including machine learning, pattern recognition and bioinformatics to name a few. The main problem of clustering is to partition the objects within a dataset into groups such that similar objects are assigned to the same group, while keeping dissimilar objects well-separated. It is considered to be an unsupervised learning task where the goal is to find meaningful groups in the data. As we shall see in section 2.7, several works have demonstrated embeddings that capture similarities between snippets of source code. This suggests that it is possible to use clustering methods to find meaningful groupings of source code using the embeddings.

There are various types of techniques which can be used to find clusters in the data.

- **Centroid-based clustering.** These methods partition the objects in a dataset by finding the centers for the chosen number of clusters and then assigning each object to its closest cluster center in such a way that the squared distances between the points and their cluster are minimized.
- **Density-based clustering.** These methods assign clusters based on the density of regions in the data. The underlying assumption is that 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.
- **Hierarchical clustering.** The clusters formed by these methods follow a tree-type structure based on the hierarchy in the data. New clusters are created from the previously formed clusters. In hierarchical clustering, there are two types to consider; The *agglomerative* (bottom up) approach and the *divisive* (top down) approach.
- **Grid-based clustering.** These methods partition the data space into a finite number of cells, forming a grid-like structure. The clustering operations are then performed on a per-cell basis. The algorithms only consider the density of points in each cell and compares it to neighbouring cells in order to assign clusters. The computational complexity of the operations is therefore low, resulting in fast operations for each cell.

The purpose of this thesis is not to find the best clustering algorithm for clustering source code by functionality, but rather to investigate the possibility of doing so. We will therefore only consider a handful of different types of clustering algorithms that can be used for finding meaningful clusters of source code.

2.4.1 K-means Clustering

K-means is one of the oldest and most used algorithms for clustering data. It is based on the idea of centroids, which is the defined as the mean of a set of data points. Given a set of points $\{x_i \in \mathbb{R}^d\}_{i=1}^n$ and an integer k, one must choose k centers among the data points, with the objective to minimize the total squared distance between each point and its closest centroid. A high-level view of the algorithm is shown in Algorithm 1 below.

Algorithm 1: K-means Algorithm

```
1. Arbitrarily choose K points as initial centroids
2. Form K clusters by assigning each point with the label of the closest centroid
3. Re-calculate the centroid of each cluster
4. Repeat steps 2 and 3 until there are no changes in centroids
```

K-means is efficient for large datasets containing numerical data, however one of the main drawbacks of K-means is that is is not deterministic. Due to the random assignment of the initial centroids, different clusters will be formed for different executions. It is therefore important to choose the right initial centroids to obtain clusters of good quality. The most straightforward way of choosing the initial centers is to uniformly sample without replacement, k points from the dataset. Arthur et al. propose an augmentation of K-means, called *k-means++*, which is the configuration that we consider in this thesis. By choosing initial centroids in high-density regions of data points, Arthur et al. [3] demonstrated that the speed and accuracy of k-means was improved.

2.4.2 DBSCAN and OPTICS

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a clustering algorithm that clusters data points using the notion of density. The main idea of this algorithm is that a point belongs to a cluster, only if it is close (densely-connected) to many other points from that cluster. DBSCAN was proposed by Ester et al. [9] in order to address a commonly occurring problem with partition-based and hierarchical clustering techniques, namely identifying clusters with arbitrary shape. The algorithm only requires two parameters to be set in order to find clusters. The first parameter being *minPts*, which is the minimum number of neighbouring points for which the point would be considered to make up a basis of a cluster. The second parameter is the radius ϵ, which specifies the area of the neighbourhood. Two points that are at distance of less than or equal to ϵ are considered to be neighbours. It does not require the number of clusters to be specified a priori to running the algorithm, which is a desirable trait. Using these two parameters, DBSCAN classifies data points as either of the following:

- **Core point.** A point is said to be a core point if it as *at least minPts* number of neighbouring points within a surrounding area with radius ϵ.
- **Border point.** A point is regarded as a border point if there are *less* than *minPts* number of neighbour points, but it is reachable within a distance of ϵ by a core point.
- **Outlier.** A point is considered to be an outlier if it is not a core point nor reachable from any of the core points.

To understand how the clusters are formed with DBSCAN, let us consider the example in Figure 2.4.1.

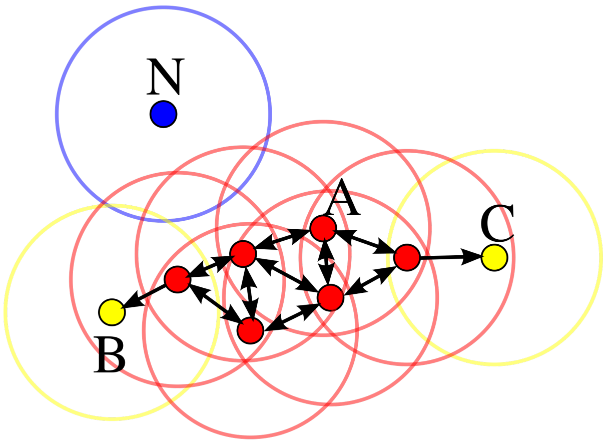


Figure 2.4.1: Example of a cluster as determined by DBSCAN, with the parameter for the minimum number of neighbours ($minPts$) being set to 4 [5].

In this example, the red points are labeled as core points since each point has at least four neighbouring points within a distance of ϵ. The yellow points are considered as border points since they have a neighbouring core point, but no do not have a sufficient number of neighbours to be labeled as a core point. Finally we have the blue point that is not reachable by any other point.

Once DBSCAN has assigned the labels for the points, the algorithm can build the resulting clusters. Two core points are merged into the same cluster if the distance between the two is less than or equal to ϵ. As such, all of the red points in Figure 2.4.1 are part of the same cluster. The bordering points which are reachable by a core point of the resulting cluster are then also added to the newly merged cluster. All of the remaining points that have not been assigned a cluster after this process are considered to be outliers (i.e. noise).

There are two significant disadvantages with DBSCAN. The first drawback is that the algorithm has a time complexity of $\mathcal{O}(n^2)$, where n is the number of points in the dataset [9]. The algorithm will therefore scale very poorly for large datasets. The second problem with DBSCAN is that it has difficulties with identifying natural clusters in the data when these clusters have varying densities [2]. An example of this problem is illustrated in Figure 2.4.2.

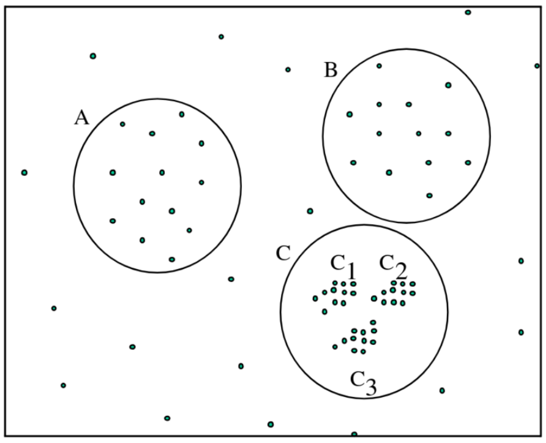


Figure 2.4.2: Clusters with very different densities [2]

All of the clusters in Figure 2.4.2 cannot be distinguished at the same time using a global density parameter. If the density parameter was chosen to fit the clusters A, B and C, then the highly dense clusters C_1, C_2 and C_3 would not be distinguishable by DBSCAN, since each of those clusters would be contained in C. On the other end of the spectrum, if the density parameter was chosen to fit C_1, C_2 and C_3, then the clusters A, B and C could not be detected due to them being sparse in comparison. This problem can be very detrimental in many real world applications, since it is common that datasets feature natural clusters that have highly varying densities. To alleviate the problem with varying densities, Ankers et al. [2] propose an algorithm called *Ordering Points To Identify the Clustering Structure* (OPTICS). It is a new algorithm that is built upon the DBSCAN that aims to solve the problem of varying densities by retaining the distances between core points and allow for the creation of sub-clusters that are based on the density of between the core points within parent cluster.

2.4.3 Hierarchical Agglomerative Clustering

Hierarchical Agglomerative Clustering (HAC) is a member of the Hierarchical Clustering family that follows the "bottom up" approach. Each object in the data is initially assigned its own separate cluster. A similarity metric is used to compute the similarities between every cluster. The two most similar clusters, as determined by a *linkage criterion*, are then subsequently merged together to form a new cluster. A binary tree, referred to as a *dendrogram* is used to represent the cluster configuration after each iteration. Once the clusters have been merged, the new similarity measures are computed once again and the merging continues. This process can then be repeated up until all of the objects have been merged into a single cluster. The final dendrogram can then be used choose a cluster configuration [22].

Linkage Criteria

There are a few linkage criteria and metrics to be considered for determining the similarities between clusters. The most commonly used similarity metric is the Euclidean distance,

$$d(u,v) = \sqrt{\sum_{i=1}^{N}(u_i - v_i)^2} \tag{2.4}$$

however there are other metrics to be considered as well. is is possible to use the Manhattan distance or the cosine distance to name a few other metrics. There are also a few different linkage criterion's that can be used to determine whether two clusters should be merged [22].

The Single Linkage criterion considers the minimum distance between two points u and v in clusters U and V.

$$D(U,V) = \min_{u\in U, v\in V} d(u,v) \tag{2.5}$$

The Complete Linkage criterion uses the maximum distance between a pair of objects u and v from two different clusters. It then chooses to merge the cluster pairs that results in the smallest distance.

$$D(U,V) = \max_{u\in U, v\in V} d(u,v) \tag{2.6}$$

The Average Linkage criterion, as the name implies, calculates the average distance between all of the objects in clusters U and V. The two clusters that yield the smallest distance are then merged.

$$D(U,V) = \frac{1}{\|U\|\|V\|}\sum_{u\in U}\sum_{v\in V} d(u,v) \tag{2.7}$$

The Ward Linkage criterion tries to minimize the total within-cluster variance when merging clusters. For each cluster pair, the total within-cluster variance is computed. The pair that leads to the smallest increase in total within-cluster variance are then chosen to be merged. The criterion is defined in Equation 2.8, where c_u and c_v denotes the centers of clusters U and V respectively.

$$D(U,V) = \frac{\|U\|\|V\|}{\|U\| + \|V\|}\|(c_u - c_v)\|^2 \tag{2.8}$$

2.5 Metrics for Clustering Evaluation

Assessing the quality of the models is one of the most important steps before deploying any machine learning algorithm. In supervised learning tasks the problem of quality assessment is relatively straight forward, given that there are clear labels for each sample. However, the supervised setting does not apply to the problem formulated in this thesis, since the objective is to cluster source code snippets by functionality. There are no absolutely definitive labels regarding the functionality of the code snippets. Clustering techniques are regarded to be unsupervised learning. Because there exists no ground truth label for each sample, there is no clear approach for testing the clustering models. The very idea of testing the performance of a clustering model is a flawed premise.

However, this doesn't imply that the assessment of clustering model is futile. Instead one must consider multiple approaches for evaluating the validity of the produced clusters. The literature presents numerous metrics that can be used to examine the quality of the clusters without requiring ground truth labels. These types of metrics are referred to as *internal measures* since they do not require any external labels to evaluate the results. They generally provide insights regarding the separability and compactness of the clusters which helps us to determine the validity of the assigned clusters. These metrics should therefore be considered as a heuristic, rather than measures for the accuracy of the predictions. The internal evaluation measures will not tell us how meaningful the clusters are, they simply allow us to compare different cluster configurations. We can use these measures to gain some insight about how valid the cluster assignments appear to be, given the similarity of the objects in the clusters.

In order to gain some insight about how meaningful the clusters are, one will simply have to resort to manual inspection of the contents of the clusters. This serves as a form of external evaluation and there are many different ways to approach it. Manual inspection of the cluster contents is a very time consuming process. It is also highly subjective. In this thesis, we will therefore consider the method names to be a sort of "fuzzy" label that we will use for external evaluation. It is not unreasonable to assume that methods that provide similar functionality, also tend to have similar names. As such, if we can find a cluster where the majority of the methods within that cluster have similar method names; then we also have some reason to believe that these methods are functionally similar.

2.5.1 Dunn Index

The *Dunn Index* (DI) is a common metric used for evaluating clusters. It follows an internal evaluation scheme, meaning that the result is based on the data itself instead of any external ground truth labels. As with any internal evaluation, the goal is to identify clusters that are compact and well-separated. A statistical interpretation of this is that the means of the clusters are relatively far apart and that there is minimal variance between the points in each cluster. Before giving the formal definition of the Dunn Index, we will first introduce some notation.

Let C_i denote a cluster of vectors in $\mathbb{R}^n$. Choose any two vectors $\mathbf{u}, \mathbf{v} \in C_i$ and define the following metrics. The maximum *intra-cluster* distance is then given by

$$\Delta_i = \max_{\mathbf{u},\mathbf{v} \in C_i} d(\mathbf{u}, \mathbf{v}) \tag{2.9}$$

Let $\delta(C_i, C_j)$ denote the *inter-cluster* distance between clusters C_i and C_j. It is calculated as the distance between the centroids of the two clusters. Using the notation introduced in this section, the Dunn Index for a set of m clusters is defined as:

$$DI_m = \frac{\min_{1 \leq i < j \leq m} \delta(C_i, C_j)}{\max_{1 \leq k \leq m} \Delta_k} \tag{2.10}$$

The Dunn Index is therefore equal to the smallest *inter-cluster* distance divided by the largest *intra-cluster* distance. By this definition, a greater value is achieved for clusters with better separation (greater inter-cluster distance) and more compact clusters (smaller cluster sizes) [8].

2.5.2 Silhouette Coefficient

Silhouette is a technique used for interpreting and validating the consistency of clusters in the data. A silhouette is a representation of a cluster based on its tightness and separation. It provides information about which of the objects that are located well within their respective clusters and which ones that only lies somewhere in between the clusters. As such, it is a useful measure for gauging the relative quality of the clusters and for choosing a good clustering configuration.

Suppose that a set of clusters $\{C_i\}_{i=1}^{n}$ have been obtained using some clustering technique. In order to get an understanding of how well each data point fits into its assigned cluster, one can define some metric that reflects the dissimilarity between the point and all other points in its assigned cluster. An intuitive choice is to consider the mean distance between the point and all of other points in the cluster as this metric. In mathematical terms, this is expressed as:

$$a(i) = \frac{1}{\|C_i\| - 1} \sum_{j \in C_i, i \neq j} d(i,j) \tag{2.11}$$

where $d(i,j)$ is the distance between data points i and j. This definition implies that the smaller the value of $a(i)$ is, the better the cluster assignment of that point is.

A second dissimilarity measure is also introduced to help understand how similar a data point is to its neighbouring clusters. Once again, the mean distance is considered but now between the point i and all other points belonging to another cluster. Let $b(i)$ be the *smallest* mean distance between i to all points in any other cluster $C_{k \neq i}$:

$$b(i) = \min_{k \neq i} \frac{1}{\|C_k\|} \sum_{j \in C_k} d(i,j) \tag{2.12}$$

The cluster that results in the smallest mean dissimilarity is then to be considered as the "neighbouring cluster" to i, meaning it would be the second best cluster assignment for that point. Now using the introduced definitions, the silhouette score for a single data point i is defined as:

$$s(i) = \begin{cases} 1 - a(i)/b(i) & \text{if } a(i) < b(i) \\ 0 & \text{if } a(i) = b(i) \\ b(i)/a(i) - 1 & \text{if a(i) > b(i)} \end{cases} \tag{2.13}$$

From the definition, it is clear that $-1 \leq s(i) \leq 1$. To further cement our intuition of what the value of $s(i)$ implies, one must look at what the measures $a(i)$ and $b(i)$ implies. Recall that both $a(i)$ and $b(i)$ are measures of dissimilarities between a point i and all of the other points in a cluster. A small value in either case indicates that the point is similar to the other points in the comparison cluster. If $s(i)$ takes on a value that is close to 1, then it is implied that the dissimilarity of a point within its assigned cluster ($a(i)$) is much smaller than the dissimilarity of the same point to all of the other clusters ($b(i)$). The point is therefore considered to be "well-clustered" in the sense that there is nothing apparently questionable about the assignment. The similarity to the assigned cluster is much greater than it would be for the neighbouring cluster. On the other hand, if $s(i)$ is close to zero, then it is not clear whether the assignment could be improved since the distance between the point and the clusters are approximately the same. The most alarming result occurs when $s(i)$ is close to -1. In this case, $a(i) \gg b(i)$ implies that the point is a lot more similar to another cluster than the one that it was assigned. The point has then in a sense been "missclassified" by the clustering algorithm.

By considering the silhouette score for all of the points in a cluster, one obtains a measure of how tightly grouped the points are. Furthermore, by calculating the mean of $s(i)$ over all points in the dataset one gets a measure of how well the data has been clustered. A silhouette plot can be constructed using the computed values of $s(i)$ for each point. For each cluster, the silhouette scores of the points belonging to that cluster is ordered in descending order. For every point in the cluster, a horizontal bar is plotted where the length of the bar corresponds to the silhouette score. A wider silhouette therefore implies large $s(i)$ values and consequently a more pronounced cluster. The height of the silhouette simply corresponds to the number of points in each cluster. For easy comparison, the silhouettes of the clusters are plotted below one another [28]. An example of this is shown in Figure 2.5.1.

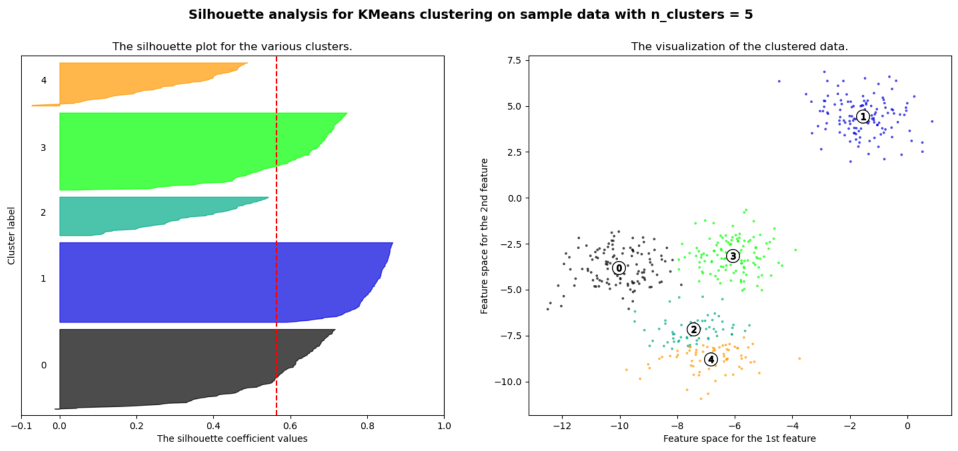


Figure 2.5.1: Example of silhouette analysis [30]

The choice of the number of clusters greatly impacts the score. If there chosen number of clusters is too small or too large then some of the clusters will generally exhibit considerably narrower silhouettes than the others. As an example, consider the case demonstrated in Figure 2.5.1. The silhouette plot indicates that $k = 5$ is a poor choice for the given data since there are clusters with below average silhouette scores. There are also wide fluctuations in the size of the silhouette plots that also raises cause for concern. The method of silhouettes is therefore a valuable metric for diagnosing the results from clustering algorithms by providing a graphical representation that illustrates how well formed the clusters are with respect to the data.

2.5.3 Adjusted Rand Index

Adjusted Rand Index (ARI) is an variation of Rand Index (RI). As such, an introduction to the Random Index is necessary before one can consider ARI. The Random Index is used as a similarity measure between two partitions. It is computed by considering all pairs of samples and counting pairs that are assigned to the same or different clusters in the two partitions. A more formal definition of the Rand Index is as follows:

Definition 2.5.1 (Rand Index)**.** *Let $S = \{o_1, \dots, o_n\}$ be a set containing n objects. Consider two partitions of S, $A = \{A_1, \dots, A_r\}$ and $B = \{B_1, \dots, B_s\}$. A and B partitions S into r and s subsets respectively where $1 \leq r, s \leq n$. Let (o_i, o_j) be the pairs of objects in S, where $1 \leq i \leq r$ and $1 \leq j \leq s$. Finally, the following measures of correspondence between the partitions are defined. Let:*

- ***a**, be the number of object pairs in S that are in **same** subsets in **both** A and B*
- ***b**, be the number of object pairs in S that are in **different** subsets in **both** A and B*
- ***c**, be the number of object pairs in S that are in the **same** subset in A and **different** subsets in B*
- ***d**, be the number of object pairs in S that are in the **different** subsets in A and **same** subset in B*

The Rand Index is then computed by:

$$RI = \frac{a+b}{a+b+c+d} = \frac{a+b}{\binom{n}{2}} \tag{2.14}$$

Intuitively, $a + b$ is interpreted as the number of *agreements* between A and B for the classification of objects in S. Similarly, $c + d$ represent the *disagreements* between the partitions. RI takes on values between 0 and 1, where 0 indicates that the two partitions do not agree on any pair of objects and 1 indicating that the partitions are exactly the same. One can therefore view RI as a measure of the percentage of correct assignments that the algorithm made, given that a ground truth partition exists [24].

With RI defined, one can now consider a variation of it - Adjusted Rand Index. ARI is introduced in order to account for the impact the the randomness has on the resulting RI, which is due to randomly picking pairs of objects while calculating the score. A "corrected-for-chance" score is obtained by instead considering the expected value of the index:

$$\frac{RI - Expected\ RI}{max(RI) - Expected\ RI}$$

More formally, let n_{ij} denote the number of objects that are common between the subsets A_i and B_j. The ARI is then calculated as follows:

$$ARI = \frac{\sum_{i,j}\binom{n_{ij}}{2} - (\sum_i \binom{n_i}{2}\sum_j \binom{n_j}{2})/\binom{n}{2}}{\frac{1}{2}(\sum_i \binom{n_i}{2} + \sum_j \binom{n_j}{2}) - (\sum_i \binom{n_i}{2}\sum_j \binom{n_j}{2})} \tag{2.15}$$

ARI is bounded between -1 and 1, where a value close to 0 indicates random labeling and a value of 1 indicates perfect labeling. ARI can take on negative values since the index may be less than the expected index. A negative ARI means that the agreement is less than what is expected of a random labeling, and the results are therefore complementary to "good" labeling [14].

2.5.4 Cosine Similarity

One of the most commonly used measures for determining the similarity between two vectors $\mathbf{u}$ and $\mathbf{v}$ in the embedded space is *cosine similarity* (CS). It is defined as the cosine angle θ between the two vectors.

$$CS = cos(\theta) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|} \tag{2.16}$$

The geometrical interpretation is that vectors which are similar should be close to one another in the embedded space. Consequently, if the angle θ between the two vectors is relatively small, then the cosine similarity measure will be close to 1. Likewise, a value close to 0 implies that the vectors are dissimilar [31].

2.6 Visualizing High Dimensional Data

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a technique that is commonly used to embed high-dimensional data into lower dimensions, making it suitable for visualization. It is an adaptation of *Stochastic Neighbor Embedding* (SNE) that addresses some of the weaknesses of SNE. As such, we will first define SNE before introducing t-SNE.

Stochastic Neighbor Embedding

Stochastic Neighbor Embedding is an embedding technique based on the idea of probable neighbours. It was proposed by Hinton et al. [11] as a way of mapping high dimensional objects into a space of lower dimension in such a way that the neighbourhood identity is optimally preserved. SNE represents similarities between objects by converting the high-dimensional Euclidean distances between the objects into conditional probabilities. The algorithm is defined as follows,

Consider a set of high dimensional vectors $\{\mathbf{x}_i \in \mathbb{R}\}_{i=1}^{N}$. For each data point $\mathbf{x}_i$ and its potential neighbours $\mathbf{x}_j$ where $i \neq j$, the probability that $\mathbf{x}_i$ would choose $\mathbf{x}_j$ as its neighbour is computed. Specifically, the asymmetric conditional probability $p_{j|i}$ is given by:

$$p_{j|i} = \frac{exp(-d_{ij}^2)}{\sum_{k \neq i} exp(-d_{ik}^2)} \tag{2.17}$$

where d_{ij} is the dissimilarity between i and j. The dissimilarity measure can be specific to the problem at hand, but the most common choice is the scaled squared Euclidean distance between the two high-dimensional vectors $\mathbf{x}_i$ and $\mathbf{x}_j$:

$$d_{ij}^2 = \frac{\|\mathbf{x}_i - \mathbf{x}_j\|}{2\sigma_i^2} \tag{2.18}$$

Here σ_i denotes the variance and it can be chosen directly or indirectly by instead selecting the number of neighbours k to consider. In the latter case a binary search is done for the value of σ_i that causes the entropy of the distribution of neighbours to be equal to $log(k)$.

Similarly for the low-dimensional counterparts $\mathbf{y}_i$ and $\mathbf{y}_j$, the conditional probability is modelled using a Gaussian distribution but with a fixed variance instead. The *induced* probability q_{ij} that point i considers point j to be its neighbour is therefore defined as:

$$q_{j|i} = \frac{exp(-\|\mathbf{y}_i - \mathbf{y}_j\|^2)}{\sum_{k \neq i} exp(-\|\mathbf{y}_i - \mathbf{y}_j\|^2)} \tag{2.19}$$

If the similarity, that exists between the high-dimensional pair $\mathbf{x}_i$ and $\mathbf{x}_j$, is correctly preserved when mapped into the low-dimensional pair $\mathbf{y}_i$,$\mathbf{y}_j$, then the conditional probabilities p_{ij} and q_{ij} will be equal. SNE will therefore try to match the two distributions in Equation 2.17 and Equation 2.19 in an attempt to optimally preserve the neighbourhood identity. Mathematically, this is achieved by minimizing the following loss function using gradient descent:

$$C = \sum_i \sum_j p_{j|i} \log\left(\frac{p_{j|i}}{q_{j|i}}\right) \tag{2.20}$$

t-Distributed Stochastic Neighbor Embedding

While SNE is capable of constructing reasonably good visualization, van der Maaten and Hinton argue that it suffers from two problems. First, the asymmetric loss function as defined in Equation 2.20 is difficult to optimize. The second problem with SNE is that it typically suffers from what is referred to as the *crowding problem.* The crowding problem occurs when a region in the low dimensional space is insufficient for fairly representing the distance between data points that are very far apart from one another in high dimensions. This results in the tendency of crowding points towards the center of the map [33].

The problems of SNE are alleviated by the two following augmentations. First, a symmetric version of the loss function in Equation 2.20 is introduced, which results in faster computation of the gradients.

$$C = \sum_i \sum_j p_{ij} \log\left(\frac{p_{ij}}{q_{ij}}\right) \tag{2.21}$$

Here, p_{ij} and q_{ij} are joint probability distributions where p_{ii} and q_{ii} are set to zero. This version is symmetric because $p_{ij} = p_{ji}$ and $q_{ij} = q_{ji} \quad \forall i, j$. In this symmetric version of SNE, the low-dimensional map q_{ij} is given by Equation 2.19.

However, if one were to define the probability for the high-dimensional map p_{ij} in a similar manner, there would be problems related to outliers. In the case where x_i is an outlier, meaning all pairwise distances to xi are large, then the corresponding values for p_{ij} would be very small. It would therefore have a very small impact when minimizing the loss. This problem is solved by defining the joint probabilities p_{ij} as the symmetric conditional probabilities:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n} \tag{2.22}$$

The second alteration is to replace the Gaussian distribution for calculating the similarity in the *low-dimensional* space with the student t-distribution, resulting in:

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|y_k - y_l\|^2)^{-1}} \tag{2.23}$$

2.7 Related Work

Previous work has been done with the goal of clustering source code using a variety of techniques. However, to the best of our knowledge, no such attempts have been made specifically for the purpose of clustering machine learning source code according to functionality. Kuhn et al. [19] investigated the possibility of using the natural language component of code, namely keywords and identifiers to find semantically similar clusters of code. In their work they present a technique based on *Latent Semantic Indexing* and clustering that was capable of producing groups of source code that use similar vocabulary. Their experiments showed that most topics were related to concepts of application and architecture, rather than domain [19].

Rousidis and Tjortjis [27] proposed a methodology for identifying patterns and similarities in Java source code, with the ultimate objective of clustering the source code to aid with software maintenance and understanding. In their approach they consider file path, packages, classes, methods and parameters associated with the source code as the input to their model. The attributes of these entities were stored in a table, and transformed into a numerical representation. Hierarchical agglomerative clustering was used then used to successfully identify meaningful groupings in the data [27].

Another highly interesting approach that is related to clustering is presented by Zhong et al. [37]. In their work they propose a framework that follows an end-to-end structure, in which the feature extraction, dimensionality reduction and clustering is done using a single model. The aim with this framework is to learn features that result in more robust, well-separated and compact clusters.

Theteen et al. [32] presented Import2Vec, which is an adaption of Mikolovs et al.'s [20] skip-gram model. The model learns embeddings of libraries by training on the co-occurrence patterns of imported libraries in a large corpus of source code. They show that the learned library embeddings capture semantically meaningful similarities among the libraries and were able to cluster these embedding according to specific domains or platform usages [32].

Alon et al. [1] proposed Code2Vec, a method for how to build distributed representations of code snippets that aim to capture deeper semantics of the source code. The key to their implementation was to represent source code as a bag of paths extracted from the Abstract Syntax Tree (AST) of the code snippet. The paths are then embedded and aggregated into code vectors which are then used to predict method names for the code snippet [1]. While they suggest that these code vectors can be used for other tasks such as code retrieval, captioning and classification or as a metric for measuring similarity between snippets, the effectiveness of this has yet to be proven. Kang et al. [16] published a paper where they evaluated the generalizability of the Code2vec embeddings for the downstream tasks of code comment generation, code authorship identification and code clones detection. Their results indicate that the use of Code2Vec embeddings does not provide any improvements over less sophisticated methods. The suitability of Code2Vec source code representations for the downstream task of clustering remains to be seen.

Inspired by the highly successful contextual embedding model for natural languages, known as BERT [6], Kanade et al. [15] propose a tokenization technique for learning contextual embeddings of source code. The BERT model is then pre-trained on a large corpus of source code and evaluated on different downstream tasks, resulting in state-of-the-art performance on a number of benchmark tasks. The BERT embeddings have been proven to be particularly useful when the model is fine-tuned on the specific downstream task. Huang et al. [13] propose an unsupervised framework for fine-tuning the BERT model with the purpose of clustering text. Another highly interesting approach is outlined by Reimers et al. [26]. In their work, they present multiple frameworks for fine-tuning BERT embeddings with the goal of learning embeddings that preserve semantic similarities among sentences.

Chapter 3

Methods

This chapter describes the strategy for answering the research questions raised in section 1, namely: *Can we learn embeddings of source code that encode functional similarities among methods, for the purpose of clustering?*

We provide a detailed explanation of the models, algorithms and evaluation methods used. As previously stated in the introduction, we consider two models for the problem presented in this thesis, Code2Vec and cuBERT. For each model, we device a set of experiments to carry out in order to answer the research questions.

The first model, Code2Vec, follows an AST-based approach for learning source code embeddings. It is trained under the pretext task of predicting the names of methods using only their AST. It learns to compute representations of methods, referred to as *code vectors*, which are used to predict the method names. In addition, Code2Vec also learns an embedding space of the predicted method names during training. The embedding of the predicted method name serves as another type of method representation that is worthwhile to investigate. In our first experiment, we apply the clustering techniques (K-means, OPTICS, HAC) detailed in section 2.4 on the code vectors and predicted method name embeddings separately. Following this experiment, we also investigate using alternative input representations in an attempt to improve the results obtained with the baseline model.

The second model that we evaluate for the problem in this thesis is cuBERT. It is an NLP-based model that learns contextual embeddings of source code. The model takes the raw surface text as input and does not require us to generate any secondary structure, like the AST. cuBERT is pre-trained on a large set of source code and can be fine-tuned to a specific task. In this work, we identify and evaluate three different frameworks for fine-tuning cuBERT for the goal of learning embeddings of source code that can be clustered by functionality. We consider a triplet framework, an unsupervised framework and a deep clustering framework. In the first two frameworks, the model is simply fine-tuned according to the respective criterion's. We then perform the cluster analysis using the cluster techniques and evaluation metrics detailed in section 2.4 and section 2.5. The deep clustering framework stands in contrast to this. It follows an end-to-end structure where a single model is constructed to perform the feature extraction, dimensionality reduction and cluster assignment. Consequently, there is no need for any external clustering technique to obtain the clusters in this framework.

3.1 Code2Vec

Code2Vec as proposed by Alon et al. [1], is a model that aims to learn embeddings of source code such that the semantics of the code is preserved in the embedding. The objective is that semantically similar snippets of source code should be mapped to similar vectors. To achieve this, the model is trained under the pretext task of predicting the label names for snippets of code. This label can correspond to a method, class or even project name that the snippet is associated with. The novelty of this model lies in that it leverages the structural information encoded in the AST of the code snippet. It does this specifically by considering each code snippet as a collection of path contexts. A path context is a tuple consisting of two leaf nodes in the AST and the path that connects the two. As an example, consider the path context for $x = 7$, as given by Alon et al. [1]:

$$\langle x, (\text{NameExpr} \uparrow \text{AssignExpr} \downarrow \text{IntegerLiteralExpr}), 7\rangle$$

A code snippet typically consists of many lines of code and the model therefore has to gauge the relative importance of each path context for making the prediction. This is done by using an *attention mechanism*, which allows the model to assign importance weights to each path context in order to learn the correspondence between paths in the AST and the target label. Code2Vec learns the attention weights simultaneously with the embeddings for the tokens, paths and method names that it sees during training. For each path context, the embeddings for the tokens and the paths are looked up and then subsequently concatenated into a single vector. A fully connected layer is then used to combine the embeddings of each path context with itself to produce a combined context vector. The attention mechanism then allows the model to aggregate the set of path contexts vectors into a *code vector*. The softmax function is then applied to the code vector in order to obtain the probabilities for the target method names. The cross-entropy loss function is used to train Code2Vec. It measures the difference between the predicted distribution of the method names q and the true distribution p. Mathematically, it is expressed as:

$$\mathcal{L} = -\sum_{i=1}^{N} y_{o,n} \log p_{o,n} \tag{3.1}$$

where N is then number of target method names in the model vocabulary and $y_{o,n}$ is a binary indicator taking on the value 1 if the predicted method name n is the correct classification for observation o, otherwise 0. $p_{o,n}$ is the predicted probability that the observation o has the method name n.

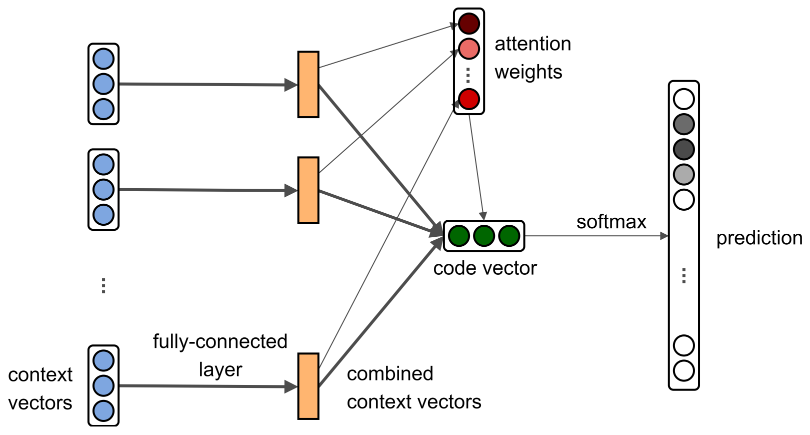


Figure 3.1.1: The architecture of Code2Vec [1].

The attention mechanism makes it possible to observe the different attention scores that each path context was assigned by the model. This increases the transparency of the model's decisions by allowing us to see the path contexts that the model deemed to be the most important when predicting the method name. A concrete example of this is shown in Figure 3.1.2.

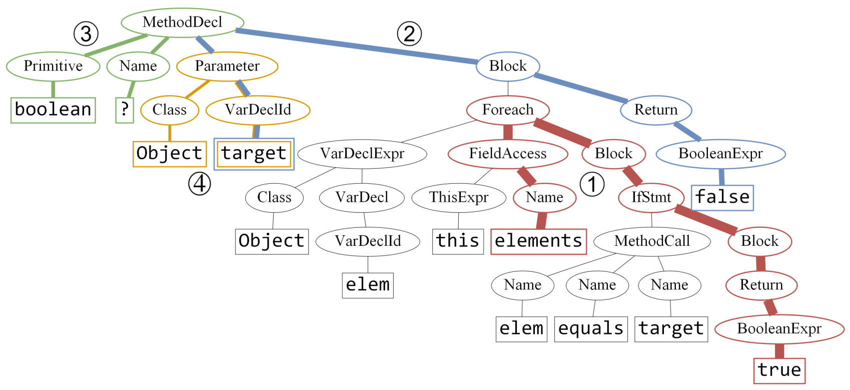


Figure 3.1.2: An example of the top-4 most important paths for classification, as determined by Code2Vec [1]. The colored lines represent the path contexts and the width of the lines are proportional to the attention score that the model assigned. A thicker line implies a higher attention score.

The label name of a code snippet is arguably one of the most semantically meaningful pieces of information that it contains. When programmers follow good coding practices, they choose variable and method names that are explanatory and highly relevant to the functionality and context of the code. Ideally - *if you have a good method name, then you should not need to have to look at the body of the method to determine what it does.* Since Alon et al. demonstrated that the model is capable of producing fairly accurate method name predictions, it is not unreasonable to think that the learned method representations encodes some of the semantics of the methods. In this work, we will therefore consider both the learned embeddings of the predicted method names as well as the code vectors as representations of a code snippet. For each of the two representations, we will apply the set of chosen clustering techniques detailed in section 2.4 in an attempt to find meaningful clusters.

3.1.1 Clustering Code2Vec Method Representations

The process of this experiment follows the flowchart in Figure 3.4.1. For every method, the AST was parsed and traversed in order to obtain the complete set of path contexts that can be found in the AST. We only kept up to 200 path contexts for each method by randomly sampling from the larger set. This was done according to Alon et al. [1] observations that further increasing this number did not improve the predictions of the model. Once the set of path contexts had been gathered for all of the methods in the dataset, the model could be trained to learn the correspondence between path contexts and the method names. From the trained model, we computed the code vectors and extracted the embedding of the predicted method name for each method in the test set. Clustering techniques were then subsequently applied on the code vectors and method name embeddings respectively. A parametric grid search is performed for each clustering technique used, choosing the configuration that yielded the largest silhouette score on the test data. We only compute and report the internal evaluation metrics of the best configurations that were found for the test data. Visualizations of the resulting clusters were obtained by applying t-SNE on the code vectors and method name embeddings, reducing the dimensionality from 384 and 128 to 3 respectively. The results and the following discussion of this experiment can be found in Chapter 4.

3.1.2 Clustering Augmented Code2Vec Method Representations

In the second experiment we investigate how the resulting clusters are affected by changing the input representation that the model is given. It is interesting to see how the cluster results are affected by changing the representation of the input that is given to the model. It could provide us with valuable insights about the properties of the data and allow us to determine the ones that are more or less important for the clustering problem. In the default setting of Code2Vec, the model tries to predict the name of a method by using a set of path contexts that were extracted from its AST. We decide to investigate two alternative approaches for the input representation; the approach of *Keywords and Identifiers* and the approach of *Library references*.

Keywords and Identifiers

In this experiment, the working hypothesis is that the hierarchical information encoded in the abstract syntax tree may not be as important to encode into the input that is given to the model. This hypothesis is based on the observation that the code written for most machine learning projects is at a high-level. A significant portion of the functionalities that are used within the projects are imported from external frameworks and libraries. Everything from the dataloaders used to set up a pipeline of data, to activation functions and the layers of the implemented models are typically imported from modules of frameworks like `TensorFlow` or `Pytorch`. We therefore hypothesize that the names of these functions and the assigned variables in the code are the most important features for learning method embeddings that reflect the functionality of the machine learning code. To test this hypothesis, we follow the same steps as described in section 3.4.2 with the main differences being the preprocessing of the methods and a minor modification to the input layer of the model. The pretext training task was not altered and the hyperparameters of the model were kept constant. The same dataset was used as in the previous experiment to facilitate a fair comparison.

In the preprocessing step, we extract the names of the functions and variables used in the method body as well as code constructs such as `return`, `for` and `while`-statements. We refer to the names of functions and variables as *Identifiers* and the code constructs as *Keywords*. Keywords are the words that are reserved in the programming language for certain functionalities. An example of this is the for-loop statement `for`. We are not allowed to use a keyword as a variable or function name. This is an approach similar to the classical *bag of words* model from NLP, where the words are extracted from the corpus without considering the order of the words in which they appear in the original sentence. In this case the words are the keywords and identifiers that were extracted from the method body. An example of this is demonstrated in Figure 3.1.3. In this example, the keywords are `True` and `return` as highlighted by the green text in Figure 3.1.3a. The remaining words in this list are the identifiers, i.e. the names of the variables and methods that are given by the developer.

```
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) Example Method

```
['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']
```

(b) A list of the Keywords and Identifiers extracted from the method body.

Figure 3.1.3: An example of two similar code snippets.

We augmented the input architecture of Code2Vec in order to enable the use of the bag of words as input. In contrast to the original version of the model, we now only need to learn an embedding for the tokens instead of embeddings for both the paths and the tokens. From the methods in the training set, we build a vocabulary that the model will learn the word embeddings for. During training, we look up the embedding of each word extracted from the method body. Code2Vec then uses the attention mechanism to attend to the words, in order to learn the correspondence between the words in the method body and the method name. After training the augmented Code2Vec, we used it to compute the code vectors and then predict the method names of the methods in the withheld test set. For the cluster analysis, we apply the set of chosen techniques on the embeddings of the predicted names and the code vectors separately. We compare the two different method representations with one another, as well as with the results from the baseline model. We considered once again both internal and external evaluation when comparing the clusters obtained from training Code2Vec on the different input types. The results and the following discussion of this experiment can be found in Chapter 4.

Library References

The following experiment is based on the observation that *most of the functionality in machine learning code is imported from external libraries*. It is therefore reasonable to consider using only the imported libraries as the sole input for the augmented Code2Vec model. To further explain this idea, let us consider the following example. In Figure 3.1.4a we have a code snippet as well as the import statements which are declared and referenced. The functionality provided by this snippet is to load data and create a pipeline to feed that data to a model. This is achieved by importing the libraries, `pandas`, `tensorflow` and `sklearn`. Pandas is a data analysis and manipulation tool which is commonly used for preprocessing and postprocessing data. The functionality imported from this library is `read_csv`, which as the name implies allows the user to load and read the contents of a csv-file. From the machine learning framework Tensorflow, the dataset module is imported, specifically the function called `from_tensor_slices`. This function is used to create the input pipeline for the model. Finally we have the method called `train_test_split` that is imported from the Sklearn module that is mainly used for data analysis. This method is used to split a dataset into two parts, which is another highly important step when creating a pipeline for training and validating a machine learning model.

Knowing what these tools are typically used for should be a valuable feature for determining what the method does. Because many of these large libraries are also used in most projects, it should be possible to find patterns across very different projects. The very presence of one or more of these imported functionalities within a method should be highly indicative of the functionality that it implements. The main idea in this experiment is therefore to *only* use the information that we gain from identifying references to libraries and functions like the ones we just looked at, in order to predict the name of the method and to ultimately use the learned representations for clustering. We refer to the extracted features as *Library References*.

```
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import train_test_split

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) Example Method

```
['pandas.read_csv',
'sklearn.model_selection.train_test_split',
'tensorflow.data.Dataset.from_tensor_slices']
```

(b) A list of the extracted Library References

Figure 3.1.4: An example of library references.

The library references are extracted by keeping a record of all of the import statements within a Python file. We then traverse the AST of the method, following the pattern explained in section 2.1. For each node visited during this traversal, we look for the existence of a reference to one of the imported packages in the Python file. If a reference is found, we store the full path of the imported function and consider this to be a feature of the traversed method. As an example, let us consider the code snippet in Figure 3.1.4a. Upon traversing the AST of the declared method, we will eventually visit the node that represents `pd` in the statement `df = pd.read_csv(filepath)`. When we look up the value `pd` among the imported functions, we can see that this is a references to the `pandas` library that was imported as `import pandas as pd`. As such, we make a record for the traversed method, storing `pandas.read_csv` as an extracted feature for the method.

Once this process had been repeated for every method in our dataset, we removed the library references that only occurred within a single project. This was done in order to avoid recording information that is too project specific. A library that is written by a developer only used for a specific project provides no usable information across different projects. Once this preprocessing step was completed, the dataset was partitioned into sets for training, validation and testing. Code2Vec was then trained under the pretext task of predicting the method name, using only the library references as input. All of the hyperparameters were kept constant to keep the comparison fair to the previously done experiments.

3.2 cuBERT

Code Understanding Bidirectional Encoder Representations from Transformers (cuBERT), is an adaptation of BERT that is designed to learn contextual embeddings of source code. cuBERT, as presented by Kanade et al. [15], uses exactly the same model architecture and pre-training process as BERT. The only difference between BERT and cuBERT is the tokenization step and the corpus that the models are pre-trained on. cuBERT is pre-trained on a large corpus of 7.4 million Python files extracted from GitHub [15].

The tokenization technique used in BERT is not suitable for source code. This is in fact true for most of the conventional tokenization techniques since they are designed with natural languages in mind. These techniques do no preserve many of the important syntactical elements that are found in programming languages. Syntactic elements like newlines, indentations and deindentations are typically removed. The tokenization technique presented by Kanade et al. preserves these syntactical elements by assigning them their own unique tokens. Special characters like parentheses, semicolons and arithmetic operators are also preserved in cuBERT. By retaining these elements of the source code, cuBERT is able to gain a deeper understanding of the meaning and context of the elements in the source code [15].

3.3 Fine-tuning cuBERT

There are many frameworks that can be considered for fine-tuning cuBERT. In this thesis we will consider three different ones; a triplet framework, an unsupervised framework and a deep clustering framework.

3.3.1 Triplet Loss

In the work of Reimers et al. they aim to learn embeddings for sentences that preserve the semantic similarities between them [26]. The method for fine-tuning cuBERT proposed in this section is inspired by their work. We aim to fine-tune cuBERT using the triplet loss function to learn embeddings of source code that encodes similarities.

Triplet loss is commonly used for similarity learning problems, such as learning similarities among objects like words or images to give a few examples. It is also used for the task of learning rankings that can be used for information retrieval [12]. The idea behind triplet loss is that it should enforce that *embeddings of similar objects should also be similar.* To achieve this, it considers the identity of objects and the distances between their embeddings. There are a few distance metrics that can be considered in this framework, with the most common ones being the Euclidean and cosine distances. Mathematically, the goal is to learn an embedding function $f : \mathbf{x} \to \mathbb{R}^d$, in which a method $\mathbf{x}$ is mapped into a feature space $\mathbb{R}^d$, such that the distance between all methods of the same identity is small, while the distance between methods with different identities is large. The triplet loss function is therefore designed to enforce the similarity by considering the distances between the embeddings of a triplet of identities. Figure 3.3.1 illustrates the concept.

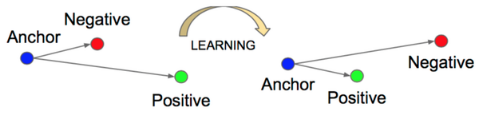


Figure 3.3.1: The objective is to minimize the distance between the anchor and the positive sample, while simultaneously maximizing the distance between the anchor and the negative sample [29].

A triplet is formed by first choosing an anchor sample, which could be of any identity. A positive and a negative sample are then subsequently chosen such that the positive sample has the same identity as the anchor and the negative sample has an identity that is different from the anchor. The embedding for each sample in the triplet are then computed by giving each sample as input to the model. Using the embeddings, the distances are computed between the anchor and the two other samples. The loss function then computes the loss by considering the distances between the samples in the triplet.

Mathematically, the loss is defined as follows. Given an anchor sample a, a positive sample p and a negative sample n, the triplet loss is computed according to Equation 3.2:

$$\mathcal{L}(a,p,n) = max(d(a,p) - d(a,n) + \alpha, 0) \tag{3.2}$$

Here, $d(x,y)$ is a distance function, for instance the Euclidean- or Cosine distance. The margin, denoted by α, is used to ensure that the positive sample is closer to the anchor than the negative sample. It is commonly chosen to be equal to one.

Framework

The architecture for fine-tuning cuBERT with the Triplet loss is illustrated in Figure 3.3.2.

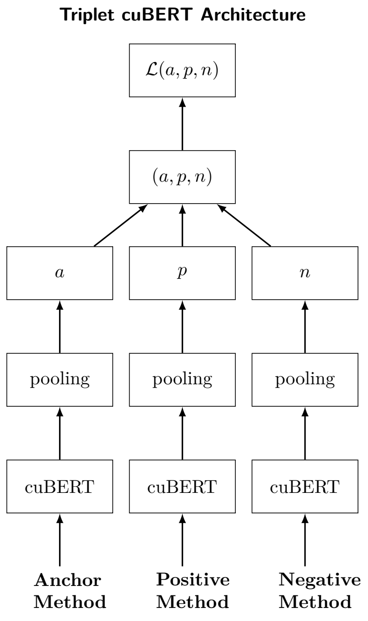


Figure 3.3.2: The architecture used for fine-tuning cuBERT

The *Triplet Network* consists of three instances of the same cuBERT model, sharing the weights. However, in practice only a single instance of cuBERT is used. In order to adhere to the triplet structure, it has three different input channels instead. This is done in order to avoid the wasteful memory allocation of three identical models. Most of the time, it is better to have a single network perform the computations three time, rather than having three identical models perform the computation once. In order to fine-tune the network using a triplet loss, a triplet must first be obtained from the samples in the dataset. Each sample in the triplet is then tokenized and given as input to cuBERT to compute the features. A pooling layer is used to aggregate the set of features that cuBERT computes for each token in the input. This is done in order to obtain a single fixed length vector (embedding) that represents the given method sample. Using the embeddings of the anchor, positive and negative sample, the triplet loss is computed. The weights of the network are then updated accordingly.

Generating Triplets

In order to utilize this loss function, we must first construct triplets from the samples in the dataset. An anchor sample is obtained by randomly choosing a method from the dataset. Positive and negative samples for the anchor are found using the method name. In the case of the positive sample, we find all methods in the dataset that shares one or more of the subwords that occurs in the method name of the anchor sample. The *BLEU score* [23] was then computed as a measure of similarity between the anchor method name and all of the positive candidates. A higher BLEU score indicates a more similar method name. The computed scores are then used to randomly obtain a positive sample from the candidates in a weighted manner; methods that had a higher BLEU score were more likely to be chosen as a positive sample. A negative sample was obtained by randomly sampling a method from the dataset that is in the complement set of the positive candidates. As such, the negative samples do not have any subword in common with the anchor method.

Fine-tuning cuBERT with Triplet Loss

cuBERT is fine-tuned following the schematic shown in Figure 3.3.2. Each sample in the generated triplet is given as input to cuBERT in order to compute the features of the sample. cuBERT then computes the contextual embeddings for each token in the input, which are then passed through a pooling layer to obtain a single embedding for the method. There are a few choices to consider regarding the pooling technique. Reimers et al. [26] outline three approaches. The most straightforward approach is to only consider the output of the first token, which is the special classification token `[CLS]`. The other options are to either compute the mean or the max of the features for each token in the sample, known as *mean-pooling* and *max-pooling* respectively. In this experiment, we will consider the feature of the `[CLS]` token that is extracted from the last encoder layer of cuBERT to be the method representation. After the features have been pooled into a single, fixed length vector (embedding), we compute the triplet loss. The weights of the model are updated accordingly during the backpropagation stage of the training.

3.3.2 Unsupervised Fine-Tuning

Huang et al. [13] propose a framework for unsupervised fine-tuning of a BERT model, with the aim of learning representations of text that are better suited for clustering. The goal with this fine-tuning is to learn an encoder, $f_\theta : \mathbf{x} \rightarrow \mathbf{z}$, which makes the representation $\mathbf{z}$ more suitable for clustering than $\mathbf{x}$. In this thesis, we will evaluate this framework on source code using cuBERT as the pre-trained model. The objective loss function described by Huang et al. [13] has two components; a masked language model loss $\mathcal{L}_m$ and a clustering loss $\mathcal{L}_c$.

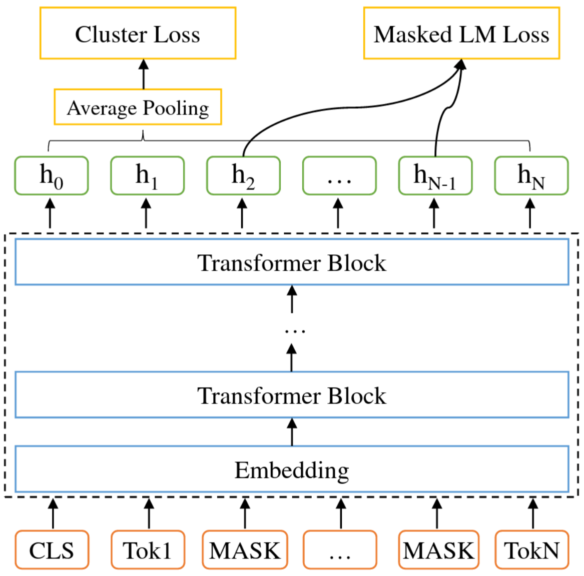


Figure 3.3.3: The architecture for the unsupervised fine-tuning of the pre-trained cuBERT [13].

The masked language model loss used in this framework is the same as presented by Devlin et al. [6]. Some of the input tokens are randomly masked and the model is faced with the task of predicting the masked tokens based on the context in which they appear. It is hypothesized that using the masked language model loss in this framework should help the model to better understand and learn representations of words in a domain-specific dataset.

We optimize this loss by minimizing the negative log-likelihood:

$$\mathcal{L}_m = -\sum_{i=1}^{C} y_{o,c} \log p_{o,c} \tag{3.3}$$

where C is then number of words in vocabulary and $y_{o,c}$ is a binary indicator taking on the value 1 if class label c is the correct classification for observation o, otherwise 0. $p_{o,c}$ is the predicted probability that the observation o is of class c.

The clustering loss is introduced in order to enhance the "natural" clusters that exist in the dataset, by enforcing better cluster separation and compactness. Mathematically, we express this loss as the *Kullback-Leibler* (KL) divergence between two distributions, P and Q, given by Equation 3.6. Here, Q denotes the distribution of the soft assignment by the Student's t-distribution between the method representation z_i and the clustering centroid μ_j:

$$q_{ij} = \frac{(1 + \|z_i - \mu_j\|^2/\alpha)^{-\frac{\alpha+1}{2}}}{\sum_{j'}(1 + \|z_i - \mu_{j'}\|^2/\alpha)^{-\frac{\alpha+1}{2}}} \tag{3.4}$$

where α is chosen to be 1 in this unsupervised setting. The clustering centroids $\mu_j, j = 1$ are obtained from the chosen cluster technique. From Q, we then derive an auxiliary target distribution P, given by:

$$p_{ij} = \frac{q_{ij}^2/f_j}{\sum_{j'} q_{ij'}^2/f_{j'}} \tag{3.5}$$

where $f_j = \sum_i q_{ij}$ are the soft cluster frequencies. The target distribution is similar to Q, but it puts more emphasis on the data points which are assigned with high confidence [13]. Using the two distributions, we compute the clustering loss $\mathcal{L}_c$:

$$\mathcal{L}_c = KL(P\|Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{3.6}$$

The unsupervised loss function can then be expressed as:

$$\mathcal{L} = \mathcal{L}_m + \mathcal{L}_c \tag{3.7}$$

As you may have noticed, this loss requires us to have a set of initial cluster centroids. To satisfy this condition, we first compute the representations of all the methods in the training set. Then, we use K-means to obtain an initial set of K cluster centroids. The target distribution P is only updated at the end of each epoch, rather than after each batch in order to avoid stability issues [13].

Unsupervised Fine-Tuning of cuBERT

Algorithm 2: Unsupervised Fine-Tuning of cuBERT

```
Data: A set of training samples, the number of epochs $N_e$
and the number of desired clusters $K$
Result: A fine-tuned cuBERT model with parameters $\Theta$
for each epoch do
    Compute the features for all samples in the training set
    Compute the cluster centroids $\mu_j$, $j = 1 \dots K$ using K-means
    for each batch do
        Step 1: Randomly sample a mini-batch of methods
        Step 2: Compute the MLM loss for each sample according to Equation 3.3
        Step 3: Compute assignment probability loss according to Equation 3.6
        Step 4: Update $\Theta$ with Adam optimizer by minimizing the unsupervised loss given by
          Equation 3.7
    end
end
```

3.3.3 Deep Robust Clustering

Deep Robust Clustering (DRC) is a framework presented by Zhong et al. that is used to train a neural network to learn invariant features and obtain robust clusters [37]. *Deep clustering* frameworks follow an end-to-end structure, in which the feature extraction, dimensionality reduction and clustering is done by a single model. The aim with these types of frameworks is to train a deep neural network to learn representations of the input data that are suitable for the clustering objective. Many of the previously proposed methods for deep clustering make use of *data augmentations* and *contrastive learning* to achieve this.

Contrastive learning is a framework where the task is formulated as learning to distinguish between objects. We simply want to learn what makes two objects similar or dissimilar. To achieve this, it is necessary to know the similarity relationship between the objects in the dataset. In many applications, we do not have labels for the similarity between two objects. In order to overcome this, researchers have resorted to the use of data augmentation. Data augmentation is a commonly used technique in data analysis where one applies transformations to the samples in the dataset to synthetically increase the diversity and number of samples. It acts as a form of regularization and helps to reduce overfitting. With the application of data augmentations, we also have an easy way of generating a new sample that has the same identity as the input sample. The previously proposed methods for deep clustering build on this assumption. Since a sample and its augmentation should have the same identity, they should also be assigned to the same cluster. However, as noted by Zhong et al. this assumption is often not sufficient in order to achieve robust clusters [37]. To understand why that is, let us consider Figure 3.3.4 to gain some further intuition about this problem. The assignment feature (AF) is the vector representation of the sample that is computed by the model. It is used to obtain the assignment probability (AP), i.e. the cluster assignment for the sample.

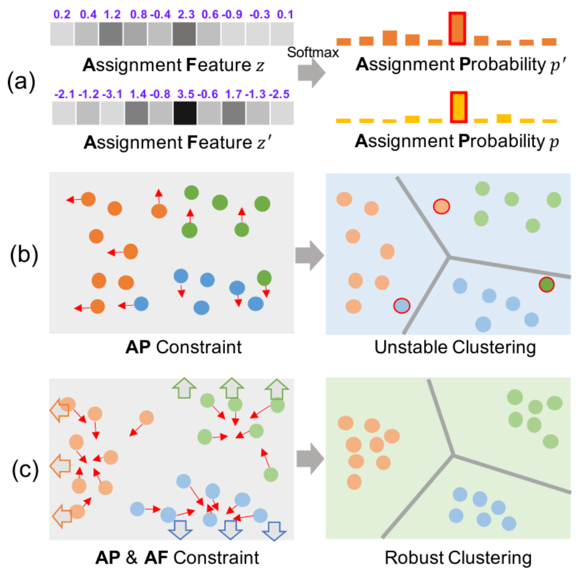


Figure 3.3.4: The intuition behind DRC [37]

In Figure 3.3.4a, z and $z^{'}$ denotes the features for a sample and its corresponding augmentation. Here we can see that even while the assignment probability of the two samples can be similar, their respective assignment features can at the same time be comparatively different. This is attributed to the fact that the assignment probability is most sensitive to the largest values in the assignment features. It is therefore possible that the assignment features from semantically different samples lead to same cluster assignment, resulting in unstable clustering as illustrated in Figure 3.3.4b. In order to attain a more robust clustering, the model must learn invariant features between the samples and their corresponding augmentations. As such, the DRC framework will consider the clustering problem from two viewpoints; the assignment probability and the assignment features. The assignment probability constraint encourages better cluster separation, i.e., larger *inter-class* variance. In order to also encourage more compact clusters, i.e., reduce the *intra-class* variance, we need introduce a constraint on the assignment features [37].

Loss Function

Consider a set of N samples, drawn from K semantically different classes. The objective of DRC is to assign the samples to the K different clusters, such that semantically similar samples are assigned the same cluster while also maintaining a robust clustering. The proposed loss function is based on three components; a contrastive loss based on the assignment probability, a contrastive loss based on the assignment features and a cluster regularization loss to prevent trivial solutions [37].

The constraint on the assignment probabilities is built on the notion of similarity between a sample and its augmentation. Since the samples should be similar, the clustering assignment should be consistent. The assignment probability loss is designed to reflect this desired behavior. Mathematically it is defined as:

$$\mathcal{L}_{AP} = -\frac{1}{K}\sum_{i=1}^{K}\log\Big(\frac{e^{q_i^t q_i^{'}/T}}{\sum_{j=1}^{K} e^{q_i^t q_j^{'}/T}}\Big) \tag{3.8}$$

where q_i and $q_i^{'}$ tells us which of the samples and their augmentations that are assigned to cluster i. The temperature denoted by T is a hyper-parameter that is used to affect the probability distribution obtained from the softmax. It is used to tune the model, making it more or less certain about its predictions. The constraint on the assignment features is based on the assumption that the features of a sample and its augmentation should be similar. Similar to the previously defined loss, the assignment feature loss is given by:

$$\mathcal{L}_{AF} = -\frac{1}{N}\sum_{i=1}^{N}\log\Big(\frac{e^{z_i^t z_i^{'}/T}}{\sum_{j=1}^{N} e^{z_i^t z_j^{'}/T}}\Big) \tag{3.9}$$

where z_i and $z_i^{'}$ are the assignment features for the sample and its augmentation, as computed by the network.

A common problem in deep clustering is that the model may assign most of the samples to the same cluster. To counteract this, we also introduce a *Cluster Regularization* (CR) term to prevent the model from falling into this local optimum.

$$\mathcal{L}_{CR} = -\frac{1}{N}\sum_{i=1}^{K}\Big(\sum_{j=1}^{N} q_i(j)\Big)^2 \tag{3.10}$$

The objective loss function of DRC is then expressed as:

$$\mathcal{L} = \mathcal{L}_{AF} + \mathcal{L}_{AP} + \lambda\mathcal{L}_{CR} \tag{3.11}$$

where λ is a weight parameter that is used to control the regularization term.

Fine-Tuning cuBERT with DRC

The framework for fine-tuning cuBERT with DRC is shown in Figure 3.3.5. A method is randomly sampled from the dataset. We use the same procedure detailed in section 3.3.1, leveraging the BLEU score between method names in order to find a positive sample for the chosen method. The method and its positive counterpart are then given as input to cuBERT. The output for each method from the last encoder layer is a feature vector of size 1024 for each token in the input. Mean-pooling is then applied to reduce the set of features into a single vector that will represent the method. In order to comply with the DRC framework, a fully-connected layer is used to reduce the feature space from $\mathbb{R}^{1024}$ into $\mathbb{R}^{K}$, where K is the number of desired clusters. The output from the fully-connected layer is the assignment feature vector for the method. The softmax function is then finally applied to the assignment features in order to obtain the assignment probabilities. Algorithm 3 details the steps taken during the fine-tuning process.

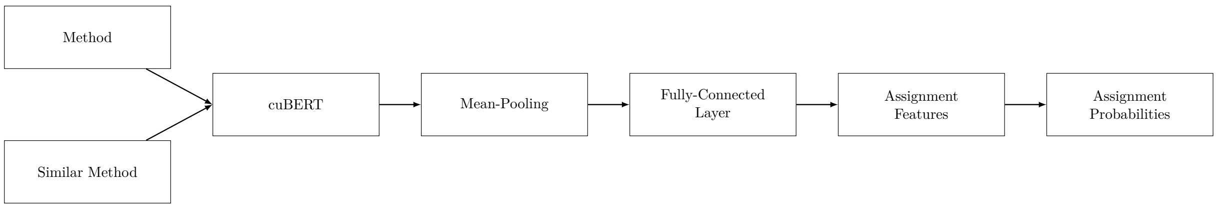


Figure 3.3.5: Framework for fine-tuning cuBERT with DRC.

Algorithm 3: Fine-Tuning cuBERT with DRC

```
Data: A set of training samples, the number of epochs $N_e$
and the number of desired clusters $K$
Result: A deep clustering model with parameters $\Theta$
for each epoch do
    for each batch do
        Step 1: Randomly sample a mini-batch of methods
        Step 2: Compute the assignment features and assignment probabilities for the method
          and its positive counterpart as shown in Figure 3.3.5
        Step 3: Compute assignment probability loss according to Equation 3.8
        Step 4: Compute assignment feature loss according to Equation 3.9
        Step 5: Compute cluster regularization loss according to Equation 3.8
        Step 6: Update $\Theta$ with Adam optimizer by minimizing the DRC loss given by Equation
          3.11
    end
end
```

3.4 Experimental Design

This section provides a general high-level overview of how the different models and clustering algorithms were used to find clusters in the source code embeddings.

3.4.1 Dataset

To evaluate the models, a large set of machine learning source code was downloaded from publicly available repositories on Github. The projects were filtered out by the tag “machine-learning” in an attempt to only retain relevant source code. The repositories were downloaded and subsequently pruned to only keep python files. Any Jupyter Notebooks that were present in the project were converted into their python script counterpart. This was done in order to be able to parse the AST, due to limitations regarding the compatibility of the AST parser. The granularity for the code snippets were chosen to be on the method level. As such, we extracted only the declared methods from each project to form our final dataset. The extracted methods from all the projects were shuffled and split into sets for training, validation and testing. All of the models were trained and validated using the training and validation sets respectively. The test set was withheld and exclusively used for the cluster analysis of the learned method representations.

3.4.2 Process

The methods declared within every Python file of the dataset were first extracted. Then, the raw source code of the methods were transformed into the source code representation that the model required as input.

For Code2Vec, this entails constructing the AST of the methods and then extracting the path contexts. It can then be trained to learn how to predict the name of a method, given a set of path contexts from the method body. With the trained model, the code vectors and the embeddings of the predicted method names of the test set can be extracted. We then apply the set of clustering techniques described in section 2.4 on the code vectors and method name embeddings separately, in order to identify meaningful clusters.

With cuBERT, we first parse the raw surface text of the methods. The text is then given to the tokenizer, allowing cuBERT to compute contextual embeddings for every token in the text. In each experiment, we fine-tune cuBERT according the the respective frameworks. In the triplet framework, we consider the `[CLS]`-token from the last encoder layer as the embeddings for the entire method. We therefore extract these method embeddings and use the set of clustering techniques to find clusters in this embedding space. The unsupervised framework follows a similar approach, however it requires that the clustering method is chosen when fine-tuning the model. As such, we chose to evaluate this framework using only K-means. In the third and final framework, Deep Robust Clustering, the model performs the cluster assignments and we therefore do not have to rely on techniques like K-means and HAC.

The cluster evaluation is conducted in the same way for all experiments. The quantitative evaluation consists of computing the internal evaluation metrics presented in section 2.5, which provides us with insights regarding cluster separability and compactness. The qualitative evaluation was done by manually inspecting the contents of the clusters to subjectively evaluate the semantic meaning of the clusters. Additionally, a low dimensional representation of the clusters were computed and visualized using t-SNE to further assist us with the qualitative assessment. The internal and external evaluation of the clusters provides us with different viewpoints for assessing the quality of the clusters as a whole. A high-level view of the experimental process is illustrated in Figure 3.4.1.

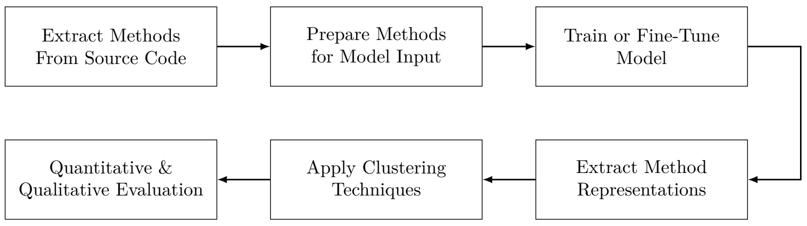

Extract Methods From Source Code
Prepare Methods for Model Input
Train or Fine-Tune Model
Extract Method Representations
Apply Clustering Techniques
Quantitative & Qualitative Evaluation

Figure 3.4.1: A high-level view of the general steps taken for learning and evaluating clusters embeddings of source code for clustering.

As described in section 1.3, the objective is to investigate if natural clusters can be identified in the learned method embedding space, that correspond to the functionality that they implement. The working assumption is that the method names should relate to the functionality that the methods implement. In order to qualitatively assess if the contents of the clusters correspond to the semantics that we are looking for, we need to effectively inspect the contents.

Our approach is as follows. We limit the clustering problem by only considering a subset of the extracted methods for cluster analysis. A subset of extracted methods is created by filtering out methods using the method name. We chose to only keep methods that had one of the words, `train`, `predict`, `process`, `save` and `forward` as a substring in the method name. The reasoning for choosing these particular words is that they are highly relevant and semantically meaningful words in the domain of machine learning code. They are frequently referenced when writing the source code for the different steps of the machine learning process. These words will therefore be considered as the label indicating the functionality of a method. The objective is therefore to investigate whether the clusters found in the learned embedding space are meaningful with respect to these labels.

3.4.3 Selecting Cluster Configurations

In section 2.4 we discussed a wide variety of different clustering algorithms that can be considered for this problem. Due to the time constraints and the problem formulation of this thesis, we are not trying to find the best algorithm. The objective is to investigate the potential of applying clustering techniques to embeddings of source code to find meaningful groupings. As such, we settled on choosing the clustering methods described in section 2.4.

3.4.4 Cluster Analysis

It is difficult to evaluate the semantic quality of the source code clusters without any access to real ground truth labels for each sample. In cases like this one should consider using both internal and external evaluation in order to draw conclusions about the methods. The metrics described in section 2.5 follow the internal evaluation scheme. These metrics will therefore only reflect the mathematical properties found within the embeddings rather than the sought after cluster property, e.g. the functional similarity. One should therefore not consider these metrics to be more reliable than what the subjective evaluation tells us when the contents of the clusters are inspected.

Chapter 4

Results

In this chapter, the Dunn index, silhouette score and adjusted rand index will be presented for the clusters produced by the different models. t-SNE visualizations of the clusters will be provided and the cluster labels will be compared to the categorical labels of the method names.

4.1 Learning and Evaluating Code2Vec Embeddings

In this section, we present the results that we obtained for the baseline Code2Vec model, as well as the results for the augmented version.

4.1.1 Baseline

Code2Vec was trained using a batch size of 64. The Adam optimizer [17] was used to minimize the objective loss function during training, using a learning rate of 1×10^{-3}. We used the ASTminer from JetBrains [18] to extract path contexts for the methods in the dataset that we curated and described in 3.4.1. After the extraction, we shuffled and partitions the methods into training, validation and test sets containing 621365, 34520 and 34521 methods respectively. The model was evaluated on the validation set at the end of every epoch. The metrics used to monitor the model performance for the pretext task of prediction method names were accuracy, precision, recall and F1. Code2Vec was trained until the F1 score on the validation set stopped improving.

Dataset	Precision	Recall	F1
Validation	0.461	0.310	0.370
Test	0.458	0.310	0.370

Table 4.1.1: The evaluation scores on the validation and test set for Code2Vec after the final training epoch.

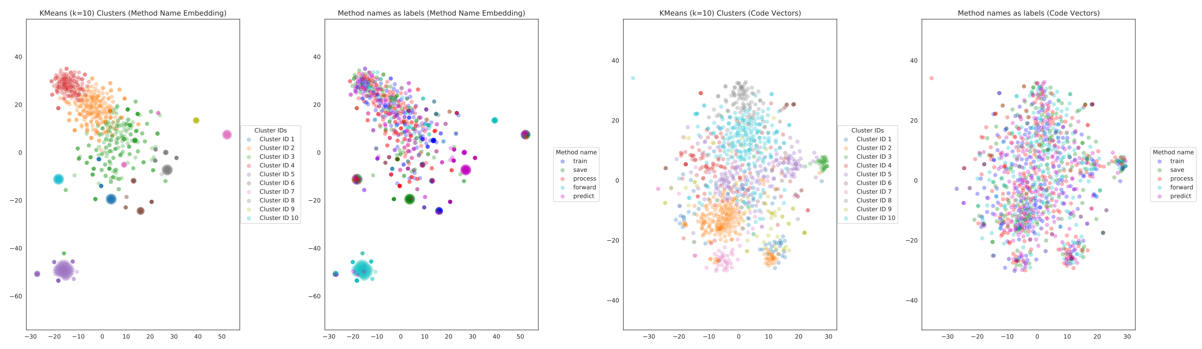
The achieved scores for the sets in Table 4.1.1 imply that Code2Vec is capable of predicting reasonably accurate method names using the bag of path contexts extracted from the method body. These scores were achieved on the entire test set, but for the cluster analysis we will only consider a subset of methods from the test set. Specifically, we choose to only consider methods where the true method name contains one of the words `train`, `predict`, `process`, `save` and `forward` as a subtoken.

Subtoken	Train	Save	Process	Forward	Predict	Total
Number of instances	469	223	255	644	359	1950

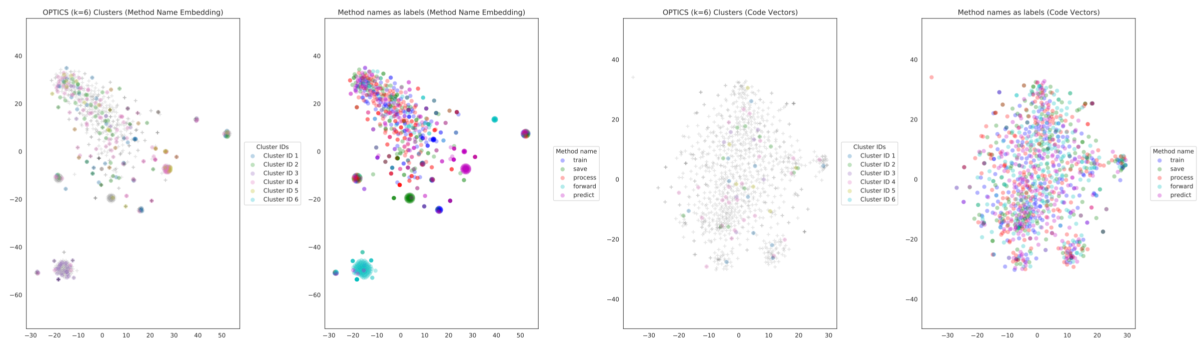
Table 4.1.2: The number of samples in the test set after filtering out methods by their name.

Using this set, we first calculate the number of instances where Code2Vec correctly predicted that one of these subtokens occured in the true method name. For instance, if a method has the true name `train_model` and Code2Vec predicted the method name to be `train_step`, we consider it to be a semantically accurate prediction since it correctly predicted that `train` was included in the original method name. Code2Vec was able to correctly predict that one of the subtokens were included in the true method name in 953 out of 1950 instances in the subset, resulting in an subtoken prediction accuracy of (48.8%). As such, Code2Vec appears to be fairly competent in its pretext task of predicting method names.

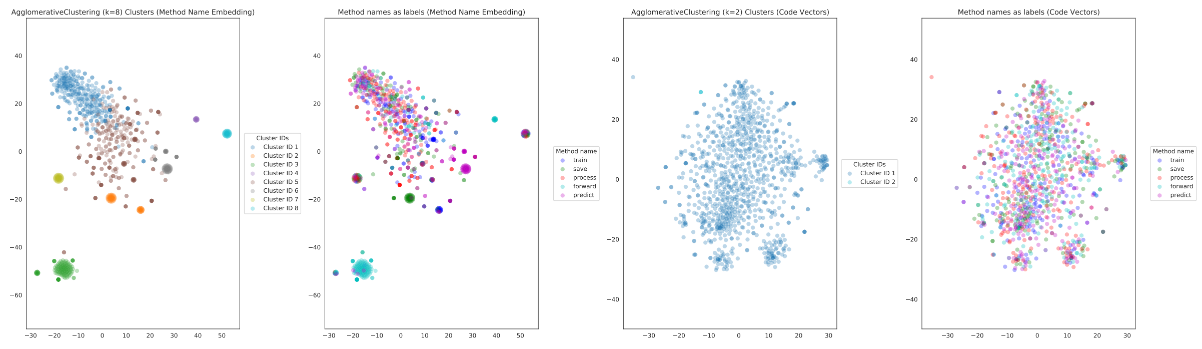
We may now focus our attention to the representations that Code2Vec learned during training. The embeddings of the predicted names as well as the code vectors were extracted for each method in the test set, shown in Table 4.1.2. The resulting clusters are depicted in Figure 4.1.1.



(a) K-means



(b) OPTICS



(c) Agglomerative Clustering

Figure 4.1.1: Clustering method name embeddings and code vectors for the chosen subset of methods in the test set. Visualized with t-SNE.

Method Representation	Estimator	Dunn Index	Silhouette Score	Adjusted Rand Index
Method Name Embedding	KMeans	0.489	0.306	0.128
Code Vectors	KMeans	0.144	0.074	0.003
Method Name Embedding	OPTICS	0.000	-0.102	0.101
Code Vectors	OPTICS	0.022	-0.104	0.008
Method Name Embedding	AgglomerativeClustering	0.508	0.317	0.116
Code Vectors	AgglomerativeClustering	0.549	0.494	-0.000

Table 4.1.3: Results from the cluster analysis of the method name embeddings and code vectors extracted from Code2Vec.

It can be seen that the clusters found among the code vector representations did not reveal any noteworthy structure. The cluster separation is quite poor and there is clearly a lot of noise. There does not seem to be any meaningful groupings that correspond to the expected method names. While the code vectors proved to be useful representations for predicting the method name, they do not appear to be suitable for clustering by functionality.

When we instead applied the clustering techniques to the embedding space for the predicted method names, we started to notice some interesting results. Suddenly there appears to be some notable clusters that are relatively compact and considerably better separated than their code vector counterparts. The embeddings of the predicted method names result in a few natural clusters that appear to contain methods that are semantically similar. Overall, the results imply that Code2Vec was able to a significant degree, learn the correspondence between the contents of a method body and the method name. Because of this, one can conclude that methods that have similar contents in the body also tend to have similar method names. However, we also note that there is a significant number of samples that do not have the cluster assignments that we expected. This is because when Code2Vec fails to provide a semantically correct prediction of the method name, it has also failed to learn an accurate representation of the method. Consequently, the noisy samples are likely the samples that did not obtain a semantically accurate prediction of the method name.

Considering the performance of the different clustering techniques, it can be concluded that K-means and Agglomerative clustering offer rather similar performance. The Dunn Index, silhouette score and adjusted rand index are fairly close for both methods when evaluated on the code vectors and the method name embeddings. K-means resulted in a slightly higher ARI for this particular dataset. OPTICS is by far the worst performing clustering technique for this dataset. In Figure 4.1.1b, we can see that the algorithm is struggling to find clusters that are relatively compact and well-separated. The poor performance of OPTICS can be likely be attributed to the fact that the algorithm forms clusters based on the notion of density. However, in high-dimensional space it becomes exponentially less likely to find a close neighbour. This phenomenon is known as *the curse of dimensionality* and it is likely the main contributing factor to the poor result. In contrast, both K-means and HAC are capable of finding clusters that seem to better correspond with the expected labels. In Figure 4.1.1b we can see that OPTICS had difficulties in finding coherent clusters. K-means and HAC produced similar clusters. From our results, it appears as if both K-means and HAC are preferable for clustering these representations.

To summarize these findings it can be concluded that the intermediate representations of methods in Code2Vec, known as code vectors, does not appear to be well suited for the task of clustering by methods which implement semantically similar functionality. While proven as useful representation for predicting method names, they do not appear to be provide good results for the downstream task of clustering on their own. Our results suggest that the code vectors are highly task dependent and may only be well suited for the pretext task which the model was trained for. This appears to be in line with what Kang et al. [16] reported, when evaluating Code2Vec embeddings for the downstream tasks of code comment generation, code authorship identification and code clone detection. Code2Vec is competent in its pretext task, however the learned embeddings do not appear to generalize that well to other tasks. Instead, we found that the embeddings of the predicted method names proved to be better suited for the problem of clustering by functionality. We believe that there is some room for further improvement on the results that we present. By training Code2Vec on an even larger dataset, it should be able to better learn the correspondence between path contexts and method names.

4.1.2 Augmented Code2Vec

In the second experiment we trained the augmented Code2Vec model on the same dataset, but using different features to represent the source code to be used as input. We compare the standard model which uses Path Contexts, with using Identifiers & Keywords extracted from the method body as input.

Method Input Features	Precision	Recall	F1
Path Contexts (Baseline)	0.458	0.310	0.370
Identifiers & Keywords	0.394	0.242	0.300
Library References	0.314	0.273	0.292

Table 4.1.4: The evaluation scores for Code2Vec on the entire test set.

Identifiers and Keywords

The obtained clusters for using only the identifiers and keywords in the method body as model input are shown in Figure 4.1.2. Once again we computed the number of instances where Code2Vec correctly predicted the existence of one of the chosen subtokens in the original method names. The number of instances of methods for each subtoken are shown in Table 4.1.5. The slight discrepancy between the number of samples when compared with Table 4.1.2 is due to the usage of different feature extractors. ASTminer [18] developed by JetBrains was used to extract the set of path contexts used to train Code2Vec in the previous experiment. When extracting the Keywords and Identifiers in this experiment, we used our own script that is based on the built-in Python library AST [4], combined with the usage of regular expressions. Using this dataset and the augmented Code2Vec model, it was able to accurately predict one of the subwords in the true method name in 864 out of 1878 instances, resulting an accuracy of 46.0%. Compared to the previous experiment, we note that this is a slight decline the prediction accuracy. The performance is however still fairly good.

Subtoken	Train	Save	Process	Forward	Predict	Total
Number of instances	455	232	237	590	364	1878

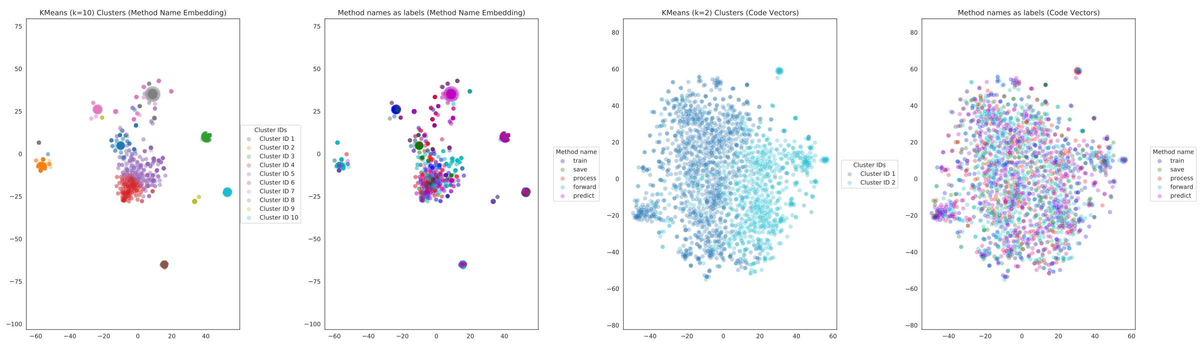
Table 4.1.5: The number of samples in the test set after filtering out methods by their name.

Table 4.1.6 shows the results from the internal evaluation of the clusters. Interestingly, the clustering results do not appear to be negatively impacted by this. OPTICS is once again the worst performing clustering technique, however it was able to assign more points to a cluster in this experiment when compared to the previous one. It therefore seems as if the vectors for the predicted method names are more densely packed in this learned embedding space. Comparing the ARI scores that were achieved by K-means and HAC in this experiment with those obtained in the previous experiment, we can see an improvement. When computing ARI, we compare the obtained cluster labels with our inferred ground truth labels. This measure therefore acts as a sort of accuracy measure. As such, the improved scores obtained in this experiment therefore imply that the clusters identified in this experiment have better correspondence to the expected labels, than what the clusters did in the previous experiment.

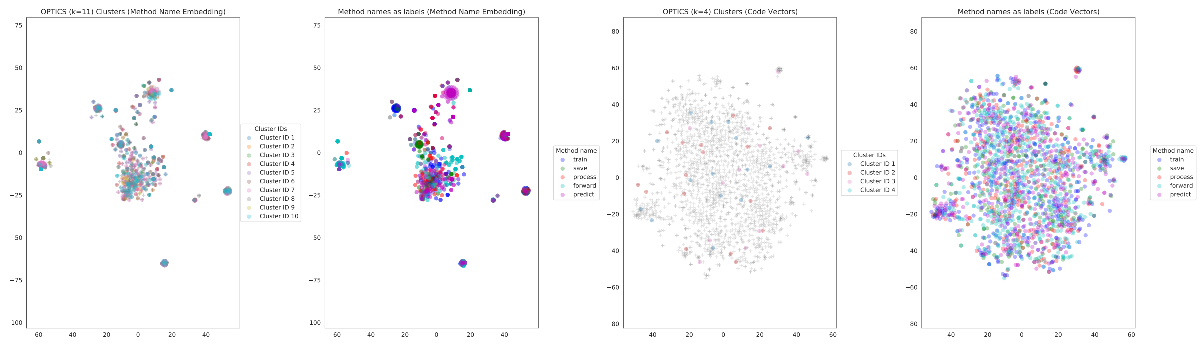
Method Representation	Estimator	Dunn Index	Silhouette Score	Adjusted Rand Index
Method Name Embedding	KMeans	0.398	0.529	0.196
Code Vectors	KMeans	0.098	0.125	0.000
Method Name Embedding	OPTICS	0.000	-0.064	0.003
Code Vectors	OPTICS	0.000	-0.058	0.001
Method Name Embedding	AgglomerativeClustering	0.386	0.537	0.152
Code Vectors	AgglomerativeClustering	0.222	0.563	0.000

Table 4.1.6: Results from the cluster analysis of the method representation learned by Code2Vec when given only tokens as input.

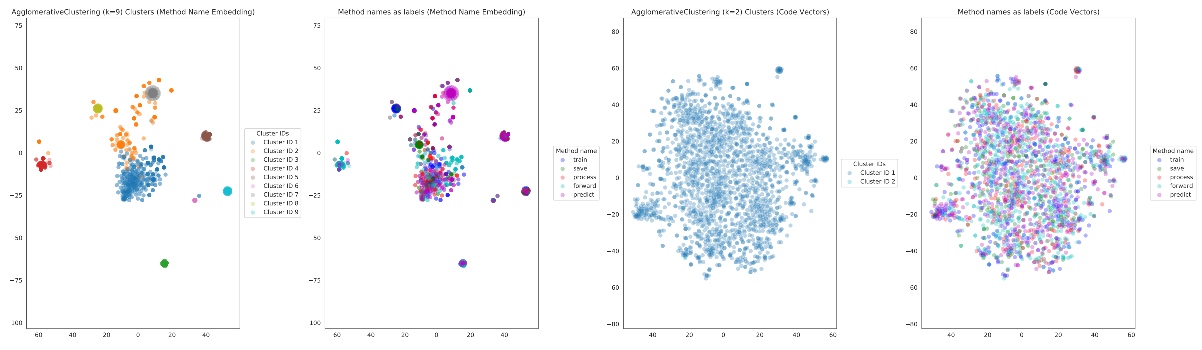
Figure 4.1.2 depicts the clusters that we obtained from the three different clustering techniques.



(a) K-means



(b) OPTICS



(c) Agglomerative Clustering

Figure 4.1.2: Clusters of the method representations learned by Code2Vec when given only tokens as input. Visualized with t-SNE.

The results from the this experiment are quite good, however it is important to note that comparing these results to the previous experiment may not be entirely fair. This model was trained on a smaller number of training samples than the baseline model, which is due to the difference in the method extractors that we previously mentioned. In particular, the model in this experiment was trained on 539,286 samples, instead of the 621,365 samples that the baseline was trained on. However, since Code2Vec is a data-hungry model that requires many training samples in order to perform well, we believe that the results of this experiment would be further improved if it had access to more samples during training. While we expected that the model in this experiment should perform slightly worse than the baseline model due to the number of training samples, it actually resulted in a more accurate cluster assignment. This could therefore suggest that the augmented model is better suited for the clustering task than the baseline, although a more fair comparison has to be done in order to provide conclusive evidence for this.

Library References

The number of instances of methods which we evaluate the clustering on are shown in Table 4.1.7. Again there is a notable discrepancy between the number of instances in this set when compared to the set shown in 4.1.2, which is due to two reasons. Firstly, we once again used our own script to extract these references by traversing the AST using the built-in Python module. The second reason, which accounts for most of the difference, is that we discarded methods where there were no library references to be found within the method body since it does not make sense to provide the model with a method where there is no input representation.

Subtoken	Train	Save	Process	Forward	Predict	Total
Number of instances	201	111	74	285	184	855

Table 4.1.7: The number of samples in the test set after filtering out methods by their name.

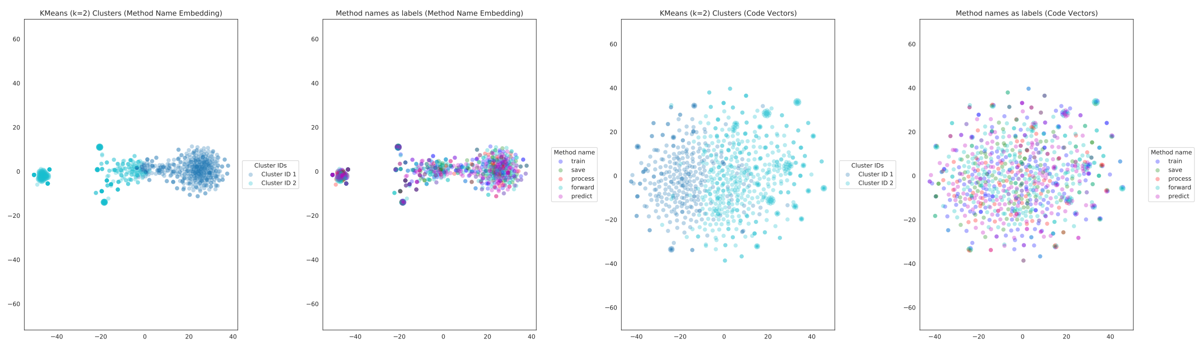
From the methods that we extracted, we computed once again the number of instances in the test set where Code2vec made a semantically correct prediction of the original method name. In this experiment, the model was only able to accurately predict one of these subwords in 13 out of 855 instances, resulting in a very poor accuracy of 1.5%. This was due to the significantly reduced number of samples that Code2Vec was able to see during training. Code2Vec is a very data-hungry model, which is restricted by the number of samples seen during training. Given more samples for training, the results could have been improved slightly. However, we noted that the average number of library references in the training set was only 2.3 library references per method. Since there are so few features for the model to consider during training, it does not come as a surprise that the task of predicting the method name became more difficult.

Because the model was not able to learn to predict method names sufficiently well during training, it comes as no surprise that the achieved clusters do not show any useful structure. We note that the Dunn index and silhouette scores are large for both K-means and HAC. These two techniques were able to identify two large clusters, which explains the large scores. The ARI score is extremely low for all techniques. As such, we have no reason to believe that the cluster assignments are semantically meaningful with respect to our expected labels.

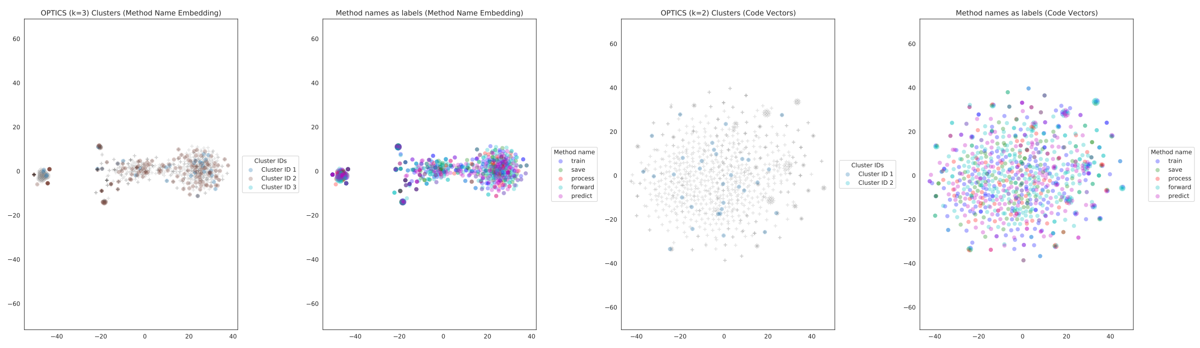
Method Representation	Estimator	Dunn Index	Silhouette Score	Adjusted Rand Index
Method Name Embedding	KMeans	0.485	0.582	-0.000
Code Vectors	KMeans	0.407	0.098	0.006
Method Name Embedding	OPTICS	0.002	-0.053	0.000
Code Vectors	OPTICS	0.000	0.015	0.002
Method Name Embedding	AgglomerativeClustering	0.451	0.569	0.000
Code Vectors	AgglomerativeClustering	0.614	0.110	0.000

Table 4.1.8: Results from the cluster analysis of the method representation learned by Code2Vec when given only library references as input.

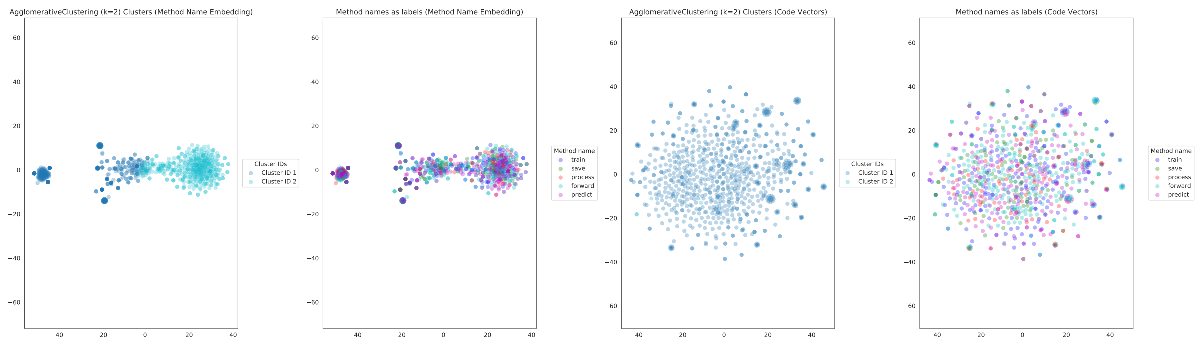
Qualitatively, this is confirmed by comparing the cluster assignments with the categorical method name labels shown in Figure 4.1.3.



(a) K-means



(b) OPTICS



(c) Agglomerative Clustering

Figure 4.1.3: Clusters of the method representations learned by Code2Vec when given only library references as input. Visualized with t-SNE.

While the results in this experiment could possibly be improved upon by training on a larger dataset, we have little reason to believe that the improvement will be significant. Our working hypothesis was that the library references are highly important. While being a reasonable assumption, it did not appear to be a suitable representation for the method functionality. By only considering it as the sole representation of the input source code, much of the 'noise' removed. However, because the number of features were very low for each method in our dataset, it made it more difficult for the model to predict the method names.

4.2 Learning and Evaluating cuBERT Embeddings

We implemented and evaluated three different frameworks for fine-tuning cuBERT.

4.2.1 Fine-Tuning with Triplet Loss

We fine-tuned a cuBERT model using the triplet loss. The model configuration was BERT-large, meaning it has 24 encoder layers, 16 attention heads and a hidden layer size of 1024. We set the maximum input sequence length to 256 and used a batch size of 4 for training. We used the cosine distance as the distance metric for the triplet loss. We also tried using the Euclidean distance but observed the best results when the cosine distance was used. The learning rate was initially set to 3×10^{-5} and we used a scheduler that reduced the learning rate with a factor of 0.1 if the validation loss stopped improving after 3 consecutive evaluations on the validation set. We only saved the model weights that were used for the evaluation on the test set when the validation loss improved. As such, the presented results are for the cuBERT model when it achieved the lowest validation loss.

We evaluate this framework on two datasets. First we fine-tune cuBERT on a small subset of the dataset that we have been using so far. The reasoning for first evaluating cuBERT on this small dataset is that we would like to first determine how well the model works on an easier set, before proceeding with a larger and more difficult one.

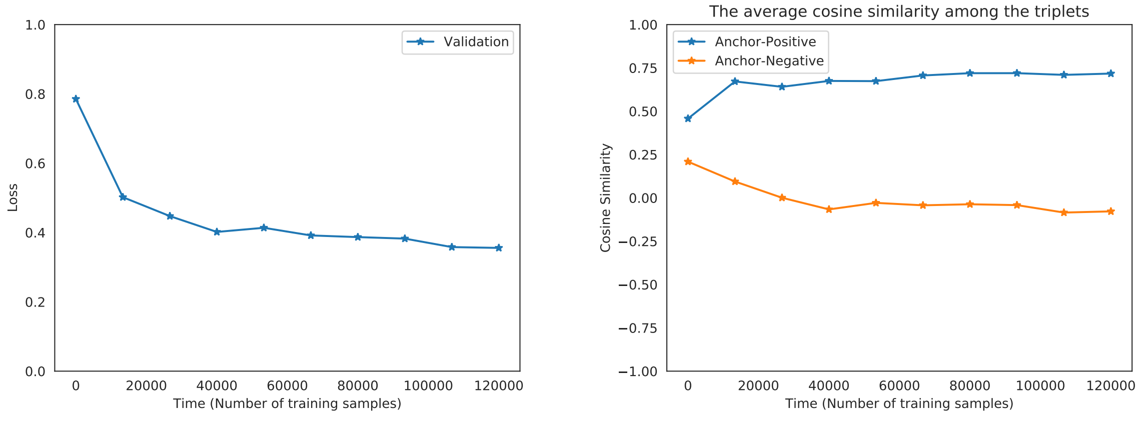
Fine-Tuned on the Small Dataset

The small dataset was constructed by only keeping methods where the method name contained one of the subtokens, `train`, `save`, `process`, `forward` and `predict`. The number of instances of each type are shown for the training, validation and test set in table 4.2.1.

Dataset	Train	Save	Process	Forward	Predict	Total
Train	2827	1371	1714	257	1126	7295
Validation	145	78	93	15	74	405
Test	159	64	101	10	72	406

Table 4.2.1: The number of instances of each method type in the datasets that cuBERT was fine-tuned and evaluated on.

In Figure 4.2.1a, we can see that the triplet loss for the validation set steadily declines during training. The early and relatively steep decline of the loss suggest that cuBERT was able to fairly quickly learn embeddings that reflect the similarities between the triplets. This may not be all that surprising seeing as this dataset only contains five different types of methods.

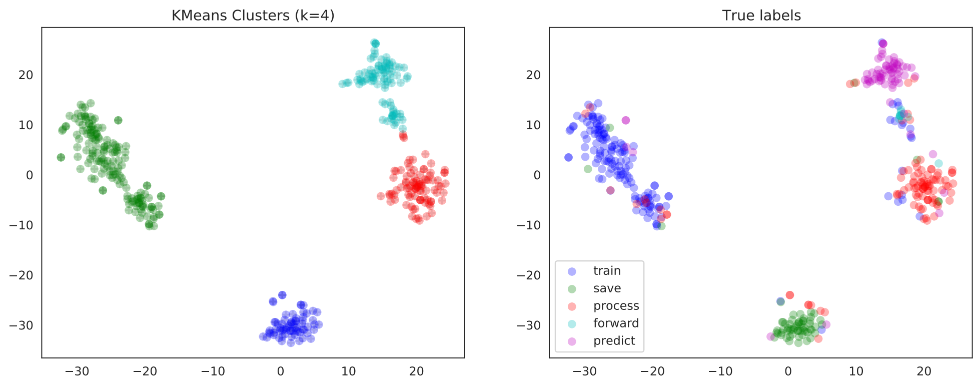


(a) The triplet loss on the validation set when fine-tuning cuBERT on the small dataset.

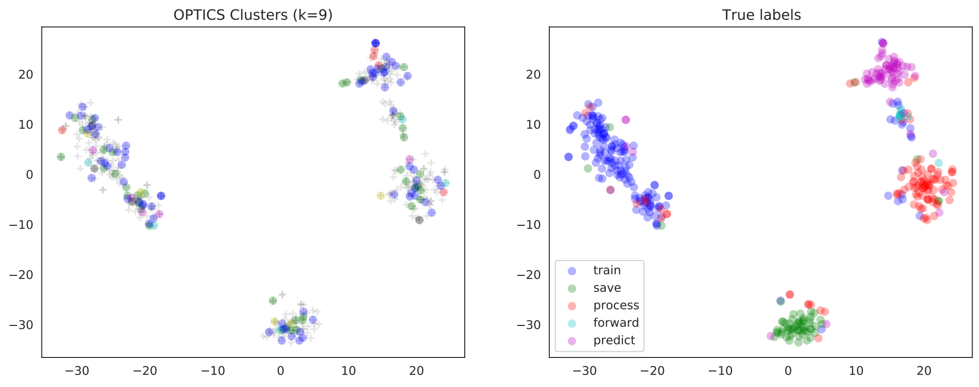
(b) The average cosine similarity between the samples in the triplets when fine-tuning the model on the small dataset.

Figure 4.2.1: The metrics tracked during the fine-tuning of cuBERT on the small dataset.

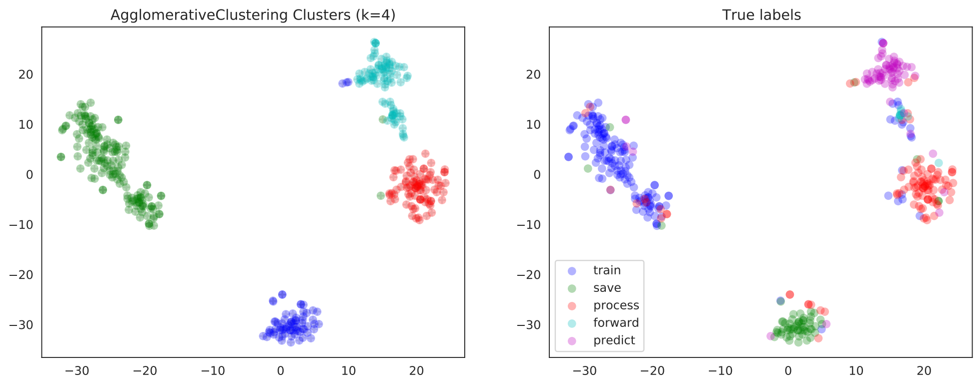
Figure 4.2.2 depicts the clusters obtained from the different clustering techniques. The results suggest that cuBERT was able to learn embeddings that encode the similarities among the methods. Qualitatively, we can see that the clusters are compact and well-separated. The clustering labels assigned by both K-means and HAC are very similar to the labels that we expect.



(a) K-means



(b) OPTICS



(c) Agglomerative Clustering

Figure 4.2.2: Clusters of the method embeddings extracted from cuBERT that was fine-tuned on the small dataset.

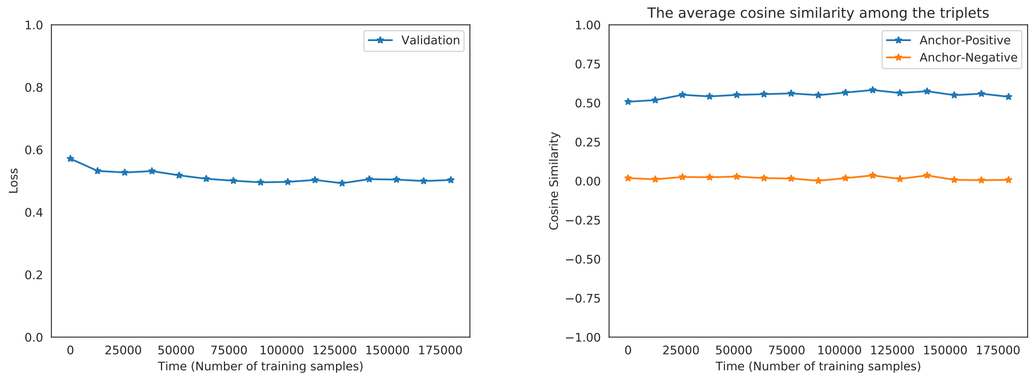
Looking at the quantitative evaluation metrics in Table 4.2.2, we can see further evidence supporting that the clusters are meaningful. The high ARI scores that were achieved by K-means and HAC indicate a high level of accuracy with the respect to the method name labels. The silhouette scores provide further reason to believe that cluster assignments are reasonable given the distances between the data points. OPTICS is once again the worst performing technique, likely still suffering from the curse of dimensionality. The initial results from the triplet loss fine-tuned cuBERT embeddings seem very promising. In the next part of this experiment, we fine-tune and evaluate cuBERT on the larger dataset to see how well it performs when the difficulty of the problem increases.

Estimator	Dunn Index	Silhouette Score	Adjusted Rand Index
KMeans	0.318	0.777	0.624
OPTICS	0.000	-0.317	-0.000
AgglomerativeClustering	0.303	0.767	0.646

Table 4.2.2: Results from the cluster analysis of the method embeddings extracted from cuBERT that was fine-tuned on the large dataset.

Fine-Tuned on the Large Dataset

Encouraged by the results from the small dataset, we fine-tuned cuBERT on a larger dataset instead. The larger dataset consisted of 204609 methods. The methods were shuffled and partitioned into sets for training, validation and testing, containing 184148, 10230 and 10231 methods respectively. All of the hyper-parameters were kept the same as when fine-tuning on the small dataset. Figure 4.2.3 shows the triplet loss and the cosine similarities between the triplets during training.

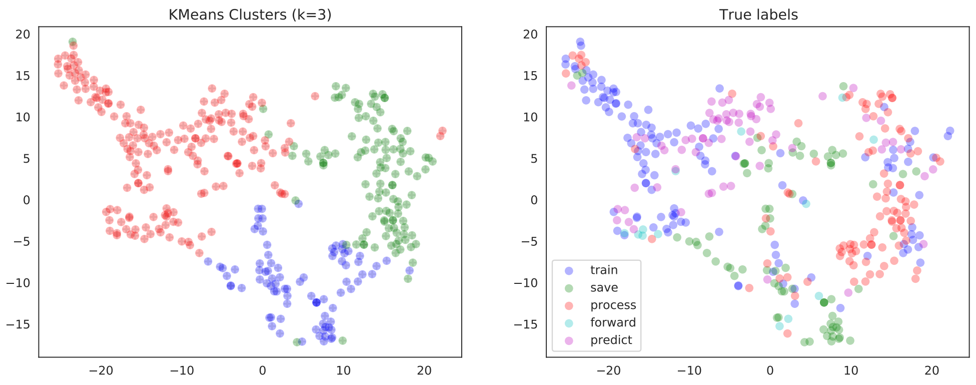


(a) The triplet loss on the validation set when fine-tuning cuBERT on the large dataset.

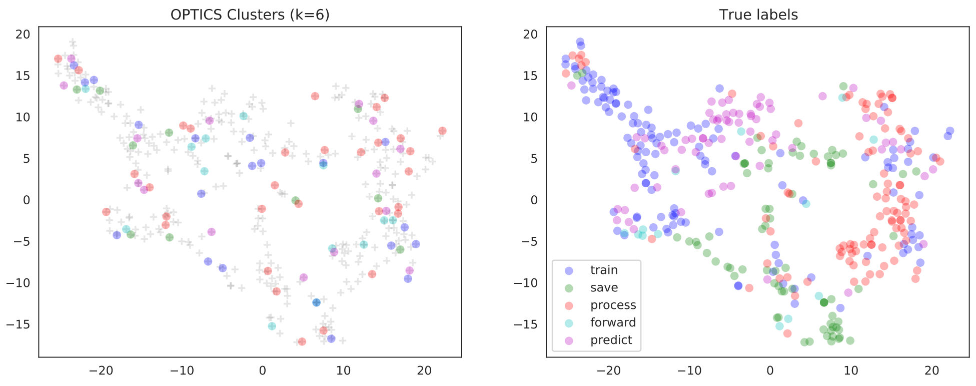
(b) The average cosine similarity between the samples in the triplets when fine-tuning the model on the small dataset.

Figure 4.2.3: The metrics tracked during the fine-tuning of cuBERT on the large dataset.

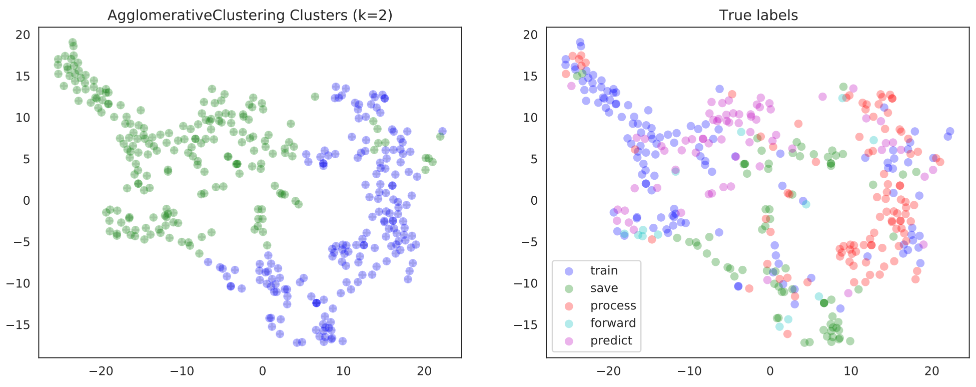
After training cuBERT on the large dataset, we extract the features of the methods in the test set. We will once again only consider a few methods for the clustering analysis. In order to easily compare the results in this experiment with the results from the previous experiments, we perform the clustering on the methods that has the previously mentioned subtokens the method names. The resulting clusters are shown in Figure 4.2.4.



(a) K-means



(b) OPTICS



(c) Agglomerative Clustering

Figure 4.2.4: Clusters of the method embeddings extracted from cuBERT that was fine-tuned on the large dataset.

In Figure 4.2.4 it can be seen that the clusters are no longer quite as compact and well-separated as they were when cuBERT was fine-tuned on the smaller dataset. This is to be expected since the difficulty of the task increased significantly due to the considerably larger number of semantically different method types in the large set. However, there still appears to be some weakly defined clusters for the method types investigated in Figure 4.2.4. The clustering techniques had troubles identifying the clusters that we were seeking in this case. K-means assigned cluster labels that most closely resembles the one labels that we were expecting. HAC had an even harder time finding these “natural” clusters. In terms of the silhouette score, it achieved the best cluster configuration when there were two clusters as seen in Figure 4.2.4c. We note that the ARI scores are low as a consequence of this. Needless to say, OPTICS was once again the worst performing technique.

Estimator	Dunn Index	Silhouette Score	Adjusted Rand Index
KMeans	0.157	0.402	0.155
OPTICS	0.000	-0.087	-0.004
AgglomerativeClustering	0.183	0.376	0.139

Table 4.2.3: Results from the cluster analysis of the method embeddings extracted from cuBERT that was fine-tuned on the large dataset.

While fine-tuning cuBERT we noticed that it is quite sensitive to the triplet pairs that are chosen during training. Initially, we generated positive samples via functionality preserving augmentations of the source code. The augmentations that we used were variable renaming, for loop to while loop conversion, moving variables to the top of the function scope and code formatting. The advantage of using augmentations in this case is we can be completely certain that the anchor and the positive sample are functionally similar. However, when fine-tuning cuBERT using this method of generating positive samples, we noticed that the problem became trivial for the model. The augmentations did not produce code that was different enough for it to be a difficult problem for cuBERT. As such, we got very high cosine similarities between the anchor and positive samples.

The only way for the model to improve was then to maximize the dissimilarity between the anchor and all of the negative samples. The end result was that cuBERT in this case became very sensitive. It would practically only consider the augmented sample of a method to be similar because it heavily focused on making most of the other sample as dissimilar as possible. Because of the poor results when generating positive samples via augmentations, we resorted to finding similar methods through the method name. The reasoning behind this decision was once again that the method name is arguably one of the most semantically meaningful pieces of information in the source code. It is by far the easiest to use since a developer has already provided us with a label. We therefore decided to obtain a positive sample for the anchor by comparing method names using the BLEU score as the similarity measure, as described in section 3.3.1.

The results of this experiment can likely be further improved upon by designing an even more sophisticated method for finding good positive samples, other than the BLEU score approach that we used in our work. Additionally, one could also use a technique known as *online triplet mining* [29], where the idea is to generate the triplets that are the most difficult for the model to distinguish. The most difficult negative samples are then the samples that have a different label than that of the anchor, but are nonetheless the ones that are closest to the anchor. In this method, a hard triplet is generated by first computing the embeddings of all samples in the mini-batch. For each anchor sample, the distance in the embedding space is then computed between the anchor and all other samples that have a different label. The sample that is closest to the anchor is then chosen as the negative sample for the triplet. The resulting triplet loss for such a pair would be relatively large, which is something that is desirable during training. This is because the loss for the simple triplet is low, resulting in practically non-existent changes to the model weights during backpropagation. When the triplets are hard, the loss is much greater which ultimately requires cuBERT to make larger changes to its weights during training. Online triplet mining would therefore not only be likely to improve the performance of the model, but also reduce the total training time.

4.2.2 Unsupervised Fine-Tuning of cuBERT

We fine-tuned a cuBERT model using the unsupervised loss function detailed in section 3.3.2. The small dataset, presented in Table 4.2.1, was used to make an initial evaluation of the model performance when fine-tuned with this framework. We chose the number of desired clusters to be $K = 5$, since that is the number of semantically different classes that the methods were chosen from. Once again, we used the BERT-large configuration, with the maximum input sequence length set to 256. We trained the model for 10 epochs, using a batch size of 8 with the learning rate set to 3×10^{-5}. The unsupervised loss achieved on the training and validation set during training is shown in Figure 4.2.5.

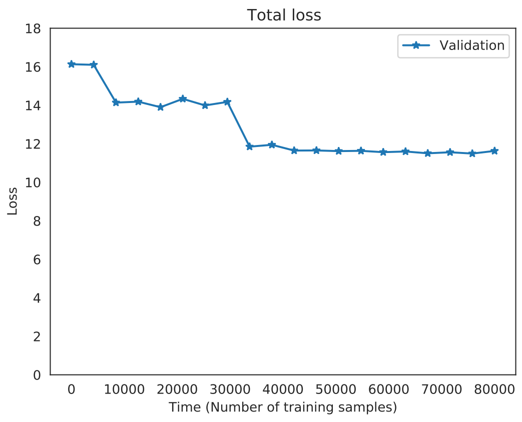
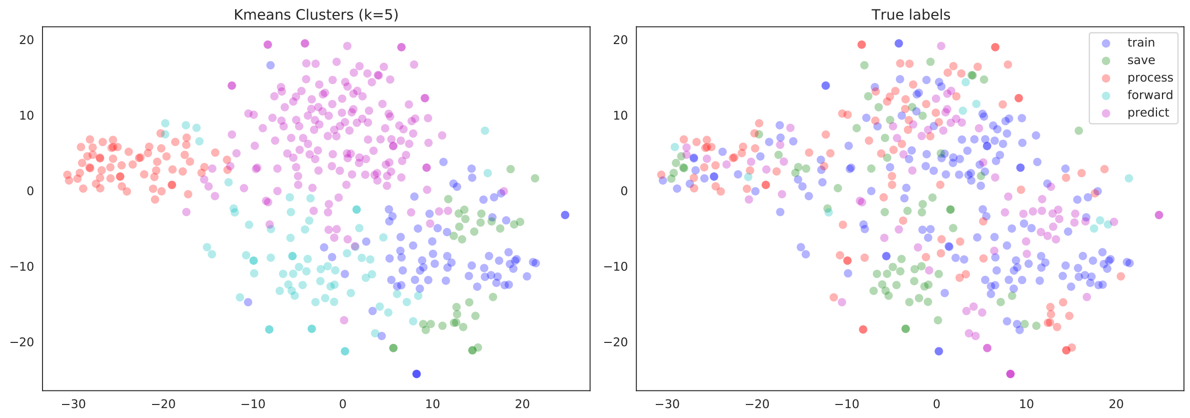
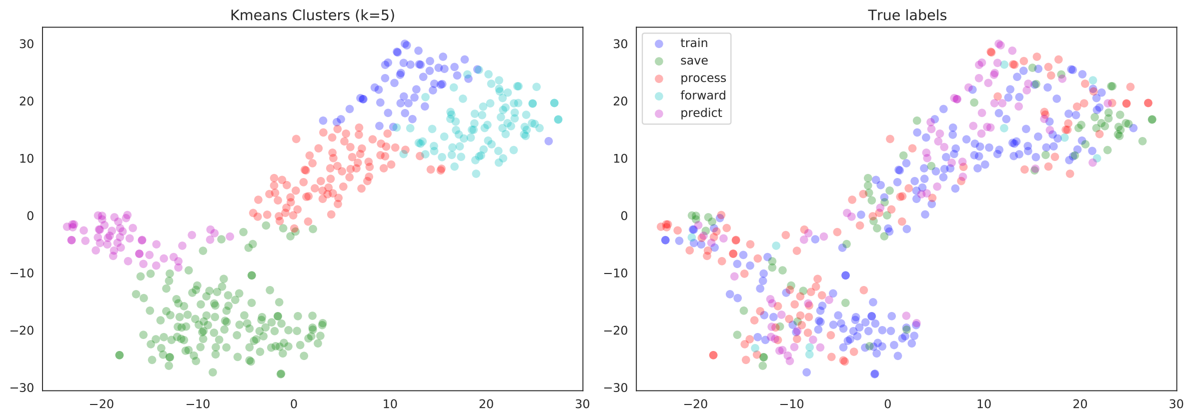


Figure 4.2.5: The loss on the training and validation sets when fine-tuning cuBERT using the unsupervised loss function on the small dataset.

As seen in Figure 4.2.5, there was a relatively steep reduction in the loss early on. We noticed that during training, the MLM loss reduced dramatically, going from the initial value of approximately 10 down to values below 1 within the first 2000 seen samples. This suggests that the model was able to very quickly learn the context and meaning of the words in the domain specific vocabulary. On the other hand, the KL component of the loss had a much slower convergence in comparison. During each epoch, the KL loss remained mostly the same across batches. We only noted some notable changes to this after the end of each epoch, when the new cluster centroids had been computed. The difficult part of the problem formulated in the unsupervised loss function therefore seems to be one that is related to the cluster assignments, rather than the component for understanding the language. To gain some further insight to the implications we should consider the cluster assignments before and after fine-tuning the model, shown in Figure 4.2.6.



(a) Pre-trained



(b) Fine-tuned

Figure 4.2.6: Visualization of the K-means clusters found in the cuBERT embeddings before and after fine-tuning.

Model	Dunn Index	Silhouette Score	Adjusted Rand Index
Pre-trained	0.002	0.339	0.057
Fine-tuned	0.006	0.608	0.033

Table 4.2.4: Results from the cluster analysis of the method embeddings extracted from cuBERT before and after fine-tuning.

The initial cluster assignments from K-means on the features from the pre-trained cuBERT model, seen in 4.2.6a, appear to be reasonable given the distances between the data points. If we instead consider the same data points but use the method names as the label, the structure does not appear to be quite as clear. We can see some weakly defined regions where methods with similar method names also have similar features. There is however a fair bit of noise and overlap among these regions. The pre-trained cuBERT features therefore do not appear to be well suited for clustering, highlighting the requirement for fine-tuning the model for this particular task.

The working hypothesis for fine-tuning cuBERT with the unsupervised loss function was that the weakly defined regions that we see would become more distinct by considering the initial clusters and the language use within these methods. 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 4.2.6b we can see that the end result is not exactly as we had first hoped.

For the particular dataset that we used, the MLM loss does not appear to have had any significant impact on the fine-tuning process. As we noted earlier, the loss converged very quickly indicating that the model already had a fairly good contextual understanding of the words used in the methods. This can likely be attributed to the very narrow vocabulary used within machine learning code. The unsupervised loss function evaluated here was first designed for natural language datasets by Huang et al. [13]. The language used in the datasets that they evaluated this framework on were likely not as narrow and specific as the machine learning source code dataset that we used for evaluation.

We note that the KL component of the loss function appears to have been fairly effective. We can see that the resulting clusters are more compact and slightly more separable than what they were for the pre-trained model. However, these clusters do not appear to be meaningful with respect to the method names. The weakly defined regions of similar methods that we saw earlier appears to be no more. This is likely attributed to the KL component of the loss. The KL loss is designed to pull data points towards the closest centroid. But it does so without considering the identities of the two data points. Again, the base assumption is that similar methods should have similar features. While that is true for some of the methods shown in Figure 4.2.6a, we can see that it is clearly not the case for most of the methods.

The end result is that many methods were pulled towards a cluster centroid where the majority of the methods did not have the same identity as the method being pulled. This is in contrast to the triplet loss that we evaluated earlier. The triplet loss resulted in well defined clusters because it considers the identities of the samples to determine whether they should be closer or further apart. With the KL loss, we simply assume that samples close to one another should already be similar, and we then enhance that property. As a result, many methods in this case were pulled towards a centroid where the majority of the points did not have the same identity. We would therefore have seen better results if the pre-trained features were better at reflecting the functional similarities between the methods.

4.2.3 Deep Robust Clustering with cuBERT

We fine-tuned a cuBERT model using the DRC loss function detailed in section 3.3.3. Similar to the previous experiments, we used the small dataset described in Table 4.2.1 to evaluate this framework on. We used the BERT-large configuration, with the maximum input sequence length set to 256. We trained the model for 5 epochs, using a batch size of 4 with the learning rate set to 3×10^{-5}. For the hyper-parameters of the DRC loss, we chose the regularization term to be $\lambda = 0.5$. We set the temperature $T = 0.5$ for the assignment features and $T = 1.0$ for the assignment probabilities.

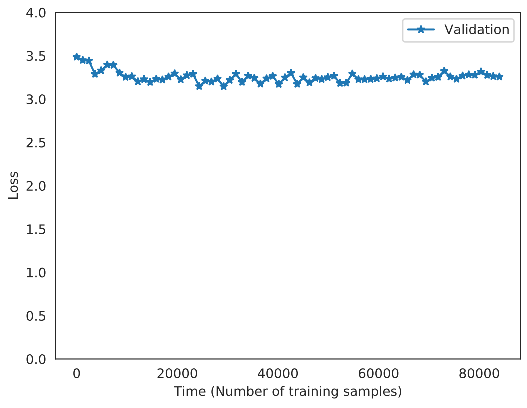
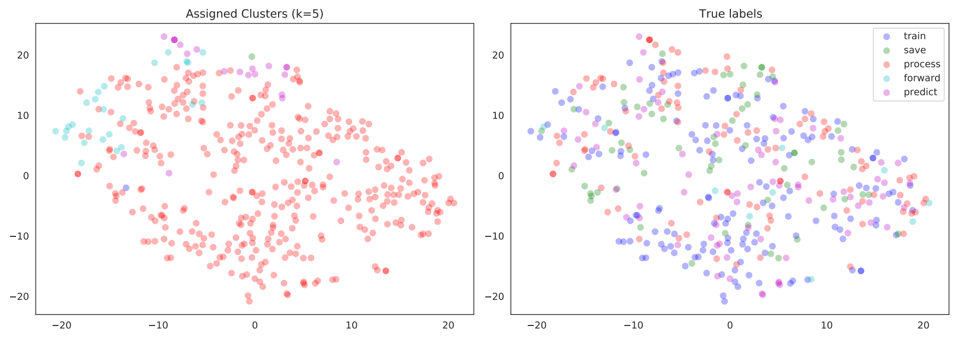
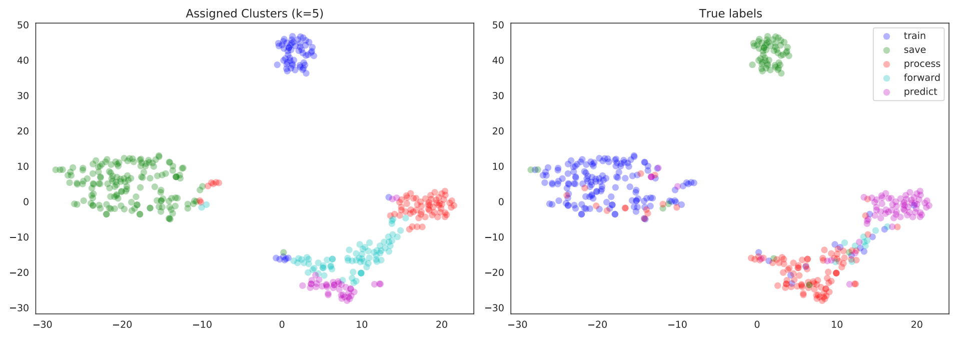


Figure 4.2.7: The loss on the training and validation sets when fine-tuning cuBERT using the deep robust clustering framework on the small dataset.

In Figure 4.2.7, we can see that the validation loss declines early on during training but then appears to stabilize after approximately 15,000 samples. This seems to suggest that cuBERT able to find a cluster configuration that was very advantageous with respect to the cluster criteria fairly early on. Now, let us consider the resulting clusters in the assignment feature and probability spaces of this model.

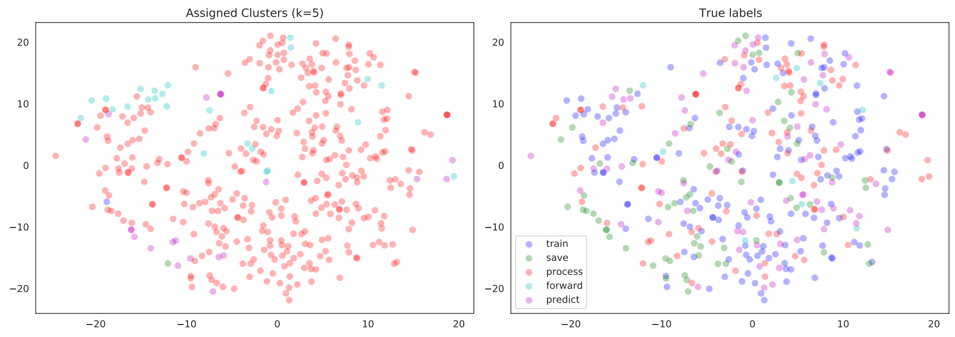


(a) Pre-trained

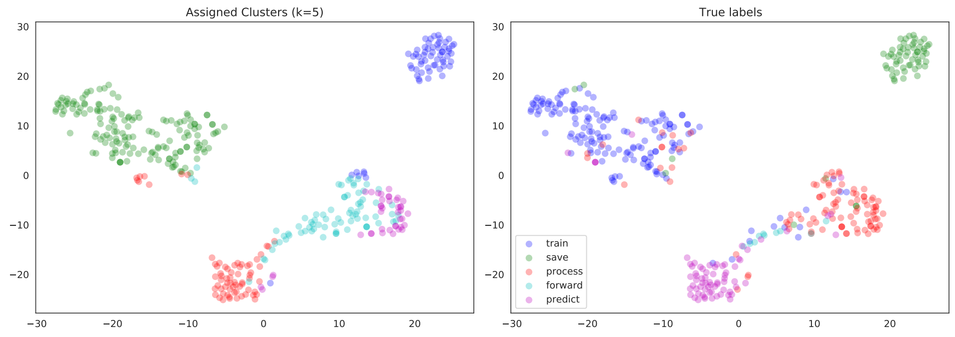


(b) Fine-tuned

Figure 4.2.8: Visualization of the assignment probabilities before and after fine-tuning.



(a) Pre-trained



(b) Fine-tuned

Figure 4.2.9: Visualization of the assignment features before and after fine-tuning.

Model	Dunn Index	Silhouette Score	Adjusted Rand Index
Pre-trained	0.052	0.367	0.015
Fine-tuned	0.066	0.662	0.563

Table 4.2.5: Results from the cluster analysis of the method embeddings extracted from cuBERT before and after fine-tuning.

Initially, we can see that the cluster assignments for the pre-trained model fall towards the trivial solution of assigning most samples to the same cluster. This problem is solved when the model gets fine-tuned, due to the regularization term that was introduced. The internal evaluation metrics presented in Table 4.2.5 demonstrates that the cuBERT model that is fine-tuned with the DRC framework achieved significantly better clustering than the pre-trained cuBERT model. We note that the Dunn Index is surprisingly low, given the good clustering. The ARI scores are relatively high, indicating that there is a significant overlap between the cluster assignments and the expected cluster labels.

Although the ARI score is not quite as high as it was when fine-tuning the model with triplet loss on the same dataset, it is still a good result. Similarly, we note that the silhouette scores are fairly high as well, which provide further reason to believe that cluster assignments are reasonable given the distances between the data points. The quantitative results are further validated by the qualitative results, shown in Figure 4.2.8 and Figure 4.2.9. Here it can be seen that the clusters from the fine-tuned model are considerably more compact and well-separated, compared to the pre-trained model. It remains to be seen how well this framework performs on the large dataset.

Chapter 5

Conclusion

In this chapter we summarize the key results of this thesis. We discuss the limitations of the results in this thesis and provide suggestions for future research.

5.1 Summary

In this project we evaluated two different models, Code2Vec and cuBERT for the purpose of learning and evaluating embeddings of source code with respect to the implemented functionality. The working hypothesis is that *methods that implement similar functionality should also have similar values in the embeddings*. For the cluster analysis, we used a set of clustering techniques in order to identify clusters in the learned embedding spaces. For the cluster analysis, we used both internal and external evaluation metrics. The Dunn Index and Silhouette Scores were used to determine how well the clustering criteria had been achieved. Adjusted Rand Index was used to compare the resulting cluster labels with the labels that we had inferred from the data. We complement this with a qualitative evaluation by visually plotting and comparing the clusters using both the predicted labels and the inferred labels to determine semantic coherence of the clusters.

5.1.1 Code2Vec

The first model, Code2Vec, was trained on a set of machine learning source code under the pretext task of predicting the method name. We found that the intermediate representation in Code2Vec, known as code vectors, were not suitable for cluster analysis. Our results did not demonstrate any meaningful structure among the code vectors, with respect to the method names. This seems to be inline with the results reported by Kang et al. where they evaluated the generalizability of the embeddings learned by Code2Vec for other downstream tasks. In their experiments, they were not able to demonstrate any improvement over the baseline models for the tasks of code comments generation, code authorship identification and code clone detection [16].

Since the code vectors did not provide a meaningful representation for the purpose of this thesis, we resorted to investigating the embedding space of the method names that Code2Vec learns during training. We used the code vectors to first predict the method name and then extracted the corresponding embedding for the predicted method name. Because the embedding space of the method name is learned from the contents in the bodies of methods, it provides a good reflection of the functionality. Upon applying clustering techniques to the learned embedding space of the predicted method names, we were able to find a few semantically meaningful clusters.

However, while we were able to identify some meaningful, we also noticed that a significant portion of the methods did not appear to belong to a semantically meaningful cluster. This is caused by Code2Vec failing to learn a representation for these that accurately reflected the functionality. We believe that our results could be improved upon by increasing the number of samples used for training. Code2Vec is a data-hungry model that needs to see many different path contexts and method names during training in order to become more competent in its task of predicting the method name. This should alleviate the problem that we encountered, but it may not be sufficient for making this model suitable for the task of clustering source code by functionality.

To further explore the capabilities of this model, we augmented the baseline model and trained it on different types of input representations. This was done in an attempt to gain some further insight into the relative importance of the different components in the method body. In our experiments, we compared the default bag of path contexts representation of the method, with two constructs of our own design; a bag of keywords and identifiers and a bag of library references. The keywords and identifiers are the names of the variables and the code constructs that are declared within the method body. The library references are the set of methods imported from external libraries, which we hypothesized to be very significant for the functionality that is implemented in machine learning code.

When comparing the results from the baseline Code2Vec with the bag of keywords and identifiers version of the model, we noticed a slight decline in the accuracy in the predictions of the method names. Interestingly enough, the following clustering results were still very similar in terms of both quantitative and qualitative evaluation. In fact, we noticed an improvement with respect to the ARI score, which suggest that the cluster assignments were more accurate in this experiment. However, it is important to note that the comparison may not be entirely fair. Due to the different method extraction process that was used in this experiment, we obtained a smaller number of samples to train the augmented Code2Vec model on. While we expected that this model should perform slightly worse, due having seen fewer number of samples during training, we actually obtained better results with respect to the achieved clusters. This result is interesting but it needs to be investigated further, using a fairer comparison in order to be able to definitely say that the augmented Code2Vec model works better for clustering.

In the last experiment with Code2Vec, we compared the bag of library references to the baseline. We found that the library references were not suitable for this application since we obtained very poor results. We attribute this mainly to the fact that the number of features in each method were relatively low. On average there were 2.3 library references per method in our dataset, which naturally makes it very difficult for a model to learn a correspondence with so few features to consider.

5.1.2 cuBERT

The second model that we explored in this thesis was cuBERT. For this model, we identified three different frameworks that can be used for fine-tuning cuBERT with the ultimate goal of learning method embeddings that encode the implemented functionality.

In the first framework, a triplet loss is defined where the objective is to learn embeddings of methods in such a way that the similarities are preserved. Given a method, a similar method and a dissimilar method, the model has to learn embeddings such that the distance in the embedding space between the similar methods is minimal, while simultaneously maximizing the distance between dissimilar methods. We fine-tuned cuBERT using the triplet loss on both a small and large dataset and were able to demonstrate clusters that were both compact, well-separated and semantically meaningful. In addition to this, a considerable advantage of this framework, in contrast to the following two frameworks, is that it does not requires us to specify the number of desired clusters a priori. With this framework, we can learn source code embeddings that encode similarities among methods, which may also be useful for numerous downstream tasks other than just clustering.

In the second framework, we explore using a completely unsupervised approach to fine-tune cuBERT. The unsupervised loss function we used has two components; a MLM loss and a KL loss. The idea behind the two components is that the MLM loss should encourage cuBERT to learn better representations of words that are domain specific, while the KL loss should enforce more compact and well-separated clusters. After fine-tuning cuBERT with this loss function, we were able to conclude that the resulting clusters from the fine-tuned model were more compact and well-separated than the clusters from the pre-trained model. However, upon inspecting the cluster contents we were unable to find semantically meaningful clusters with respect to the method names. We believe that this is mainly due to the weak cluster properties that the pre-trained cuBERT features exhibited.

In the third and final framework, we fine-tuned cuBERT following a deep robust clustering framework. This framework consists of an end-to-end structure, in which the feature extraction, dimensionality reduction and clustering is performed by a single model. The idea behind this framework is that the features and the cluster assignments should be similar for inputs that have the same identity. As such, the loss function consists of three components, formulating a loss based on the assignment features, assignment probabilities and a cluster regularization loss. By considering not only the assignment probabilities but also the assignment features, the loss function encourages more robust, well-separated and compact clusters. In our experiments we found that this framework resulted in relatively well defined clusters. The resulting clusters were considerably better than those for the unsupervised model. They were however not as impressive as they were for the model that was fine-tuned with the triplet loss.

Intuitively, the difference between the three frameworks evaluated in this thesis is that both the triplet and deep robust clustering frameworks enforce similarity preserving embeddings by considering the identities between samples in the dataset. This is in contrast to the unsupervised framework, which merely attempts to enhance the natural clusters that already exist in the data. From our results in this experiment, we conclude that cuBERT when fine-tuned with the triplet loss framework resulted in the source code embeddings that were most suitable for clustering by functional similarities. We obtained clusters that were both compact and well-separated with cluster assignments that were accurate with respect to the inferred labels of functionality. With the results of this thesis, we have found evidence indicating that it is possible to learn embeddings of source code that encode the functional similarities among snippets of source code.

5.2 Limitations

There are some limitations of this project that should be considered. First and foremost, the main limitation of this thesis is the assumption for which we based the qualitative evaluation on; namely that *the method names reflect the implemented functionality of the source code.* It is difficult to qualitatively evaluate the functionality that a piece of source code implements since this is partly subjective. Doing so for a large set of methods makes this task even more daunting. Manual inspection of the source code for each sample in the dataset is simply not feasible. As such, we chose to rely on the method names to inform us about the implemented functionality, since the author of the code has then in some sense already done the labeling for us. However, if this assumption does not hold a large majority of the samples, then the results from the clustering will be largely inconclusive.

To address this concern, we took some measures in an attempt to validate that this assumption is fairly sound. First of all, we trained our models on methods from top-starred open-source machine learning projects available on Github. These projects generally have good method and variable naming. In addition to this we also manually inspected a randomly sampled subset of the extracted methods. This was done in order to subjectively determine how accurate the names of the methods were in relation to the implemented functionality. While there were of course a few outliers, for the most part we found that the method names were quite explanatory and representative of the implemented functionality. It is however very important to take this into consideration since the assumption is plays a central part in the investigated frameworks and it forms the basis for the qualitative evaluation of the clusters.

Another limitation is that we only evaluated the clusters for a chosen set of methods. While the we obtained some encouraging results, it is possible that the results could be very different when evaluating the clusters for another set of methods. We must emphasize that it is difficult to efficiently investigate how well these frameworks works for all of the different in a large and diverse source code dataset. In order to do so, one would have to define a sound methodology to perform such evaluations. This task is something that is difficult enough that it would deserve its own work. Due to the time constraints of this project, we had to settle for the evaluation approach that we have described in this thesis. Nonetheless, it is something that should be considered when interpreting the generalizability of these results.

Expanding upon this, the models and frameworks evaluated in this thesis were applied to Python source code that were written for machine learning applications. It is unclear how the performance would differ for a different dataset of source code. We believe that the frameworks would yield similar performance. In fact, some may even work better for such datasets due to the increased variance between the samples, making it easier to learn more the distinctive features between methods that are written for different applications. It remains to be seen how well these methods generalize on even larger datasets. Additionally, the unsupervised and deep robust clustering frameworks require us to specify the number of desired clusters a priori. This is something that is notoriously difficult to determine - even more so for a dataset of source code. While it is possible to do hyper-parameter tuning in order to find a good configuration for the particular dataset that these frameworks are applied to, there is still no guarantee that the resulting clusters are semantically meaningful. It is therefore a noteworthy drawback of the two aforementioned frameworks that one should consider.

5.3 Future Work

There has been a very limited number of previous works to compare the results of this thesis with. There have certainly been works presented in the literature, that aim to cluster source code. However to the best of our knowledge, these works have not leveraged deep neural networks for the purpose of clustering source code by functionality. Even less so for Python code in the very specific domain of machine learning. Considering the results and the limitations of the work in thesis, it could serve as a basis for future research to expand upon. There are many things left to be explored and as such we will provide a few suggestions for future work to look into.

In this thesis, we restricted ourselves to only investigate these methods for machine learning source code. However, the different methods and models that we investigated are very much applicable to other datasets. The language used within the methods in the machine learning source code dataset is fairly specific, which makes the problem of learning distinguishing features more difficult. We believe that the methods evaluated in this thesis would perform quite well on a dataset with more diversity. A dataset containing source code from many different domains is likely to have a greater variance in the vocabulary used. Methods that are written for different types of applications tend to use different kinds of identifiers. We therefore believe that it would be easier for the models to learn more distinguishing features in such datasets. As such, it would be interesting to investigate the potential of this.

It would also be interesting to further investigate different frameworks for fine-tuning cuBERT. The family of BERT models have achieved state-of-the-art results on numerous tasks. It is a very powerful model and in this thesis we were able to demonstrate some promising results. We believe it would also be beneficial to further investigate and empirically determine the impact that the different components of the loss functions have on the results. In addition to this, it would be fruitful to evaluate the embeddings learned through the different frameworks that we used for tasks other than clustering. This could provide us with valuable information, leading to a greater understanding of relative importance of the different properties in source code for the machine learning models. Ultimately this could be a valuable aid in the development of new models and frameworks that could achieve new state-of-the-art results.

References

[1] Alon, Uri, Zilberstein, Meital, Levy, Omer, and Yahav, Eran. "code2vec: Learning Distributed Representations of Code". In: *CoRR* abs/1803.09473 (2018). arXiv: `1803.09473`. URL: `http://arxiv.org/abs/1803.09473`.

[2] Ankerst, Mihael, Breunig, Markus M, Kriegel, Hans-Peter, and Sander, Jörg. "OPTICS: Ordering points to identify the clustering structure". In: *ACM Sigmod record* 28.2 (1999), pp. 49–60.

[3] Arthur, David and Vassilvitskii, Sergei. *k-means++: The advantages of careful seeding.* Tech. rep. Stanford, 2006.

[4] *ast - Abstract Syntax Trees.* Accessed on 21.04.2021. URL: `https://docs.python.org/3/library/ast.html`.

[5] *DBSCAN Example.* Accessed on 21.04.2021. Feb. 2021. URL: `https://en.wikipedia.org/wiki/DBSCAN#/media/File:DBSCAN-Illustration.svg`.

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

[7] Dong, Xingping and Shen, Jianbing. "Triplet loss in siamese network for object tracking". In: *Proceedings of the European conference on computer vision (ECCV).* 2018, pp. 459–474.

[8] Dunn, Joseph C. "Well-separated clusters and optimal fuzzy partitions". In: *Journal of cybernetics* 4.1 (1974), pp. 95–104.

[9] Ester, Martin, Kriegel, Hans-Peter, Sander, Jörg, Xu, Xiaowei, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *Kdd.* Vol. 96. 34. 1996, pp. 226–231.

[10] Hindle, Abram, Barr, Earl T, Gabel, Mark, Su, Zhendong, and Devanbu, Premkumar. "On the naturalness of software". In: *Communications of the ACM* 59.5 (2016), pp. 122–131.

[11] Hinton, Geoffrey and Roweis, Sam T. "Stochastic neighbor embedding". In: *NIPS.* Vol. 15. Citeseer. 2002, pp. 833–840.

[12] Hoffer, Elad and Ailon, Nir. "Deep metric learning using triplet network". In: *International workshop on similarity-based pattern recognition.* Springer. 2015, pp. 84–92.

[13] Huang, Shaohan, Wei, Furu, Cui, Lei, Zhang, Xingxing, and Zhou, Ming. "Unsupervised Fine-tuning for Text Clustering". In: *Proceedings of the 28th International Conference on Computational Linguistics.* 2020, pp. 5530–5534.

[14] Hubert, Lawrence and Arabie, Phipps. "Comparing partitions". In: *Journal of classification* 2.1 (1985), pp. 193–218.

[15] Kanade, Aditya, Maniatis, Petros, Balakrishnan, Gogul, and Shi, Kensen. "Learning and Evaluating Contextual Embedding of Source Code". In: *International Conference on Machine Learning.* PMLR. 2020, pp. 5110–5121.

[16] Kang, Hong Jin, Bissyandé, Tegawendé F, and Lo, David. "Assessing the generalizability of code2vec token embeddings". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE. 2019, pp. 1–12.

[17] Kingma, Diederik P and Ba, Jimmy. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[18] Kovalenko, Vladimir, Bogomolov, Egor, Bryksin, Timofey, and Bacchelli, Alberto. “PathMiner: a library for mining of path-based representations of code”. In: *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press. 2019, pp. 13–17.

[19] Kuhn, Adrian, Ducasse, Stéphane, and Gı̀rba, Tudor. “Semantic clustering: Identifying topics in source code”. In: *Information and software technology* 49.3 (2007), pp. 230–243.

[20] Mikolov, Tomas, Chen, Kai, Corrado, Greg, and Dean, Jeffrey. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[21] Mikolov, Tomáš, Yih, Wen-tau, and Zweig, Geoffrey. “Linguistic regularities in continuous space word representations”. In: *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*. 2013, pp. 746–751.

[22] Nielsen, Frank. “Hierarchical Clustering”. In: Feb. 2016, pp. 195–211. ISBN: 978-3-319-21902-8. DOI: `10.1007/978-3-319-21903-5_8`.

[23] Papineni, Kishore, Roukos, Salim, Ward, Todd, and Zhu, Wei-Jing. “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.

[24] Rand, William M. “Objective criteria for the evaluation of clustering methods”. In: *Journal of the American Statistical association* 66.336 (1971), pp. 846–850.

[25] Ravichandiran, S. *Getting Started with Google BERT: Build and train state-of-the-art natural language processing models using BERT*. Packt Publishing, 2021. ISBN: 9781838826239. URL: `https://books.google.se/books?id=CvsWEAAAQBAJ`.

[26] Reimers, Nils and Gurevych, Iryna. “Sentence-bert: Sentence embeddings using siamese bert-networks”. In: *arXiv preprint arXiv:1908.10084* (2019).

[27] Rousidis, Dimitris and Tjortjis, Christos. “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.

[28] Rousseeuw, Peter J. “Silhouettes: a graphical aid to the interpretation and validation of cluster analysis”. In: *Journal of computational and applied mathematics* 20 (1987), pp. 53–65.

[29] Schroff, Florian, Kalenichenko, Dmitry, and Philbin, James. “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.

[30] Scikit-learn. *Selecting the number of clusters with silhouette analysis on KMeans clustering*. Accessed on 21.04.2021. URL: `https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html`.

[31] Singhal, Amit et al. “Modern information retrieval: A brief overview”. In: *IEEE Data Eng. Bull.* 24.4 (2001), pp. 35–43.

[32] Theeten, Bart, Vandeputte, Frederik, and Cutsem, Tom Van. “Import2vec - Learning Embeddings for Software Libraries”. In: *CoRR* abs/1904.03990 (2019). arXiv: `1904.03990`. URL: `http://arxiv.org/abs/1904.03990`.

[33] Van der Maaten, Laurens and Hinton, Geoffrey. “Visualizing data using t-SNE.” In: *Journal of machine learning research* 9.11 (2008).

[34] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N, Kaiser, Lukasz, and Polosukhin, Illia. “Attention is all you need”. In: *arXiv preprint arXiv:1706.03762* (2017).

[35] Williamson, E.A.B.S.G. *Lists, Decisions and Graphs*. S. Gill Williamson. URL: `https://books.google.se/books?id=vaXv%5C_yhefG8C`.

[36] Xu, Xiaojun, Liu, Chang, Feng, Qian, Yin, Heng, Song, Le, and Song, Dawn. “Neural network-based graph embedding for cross-platform binary code similarity detection”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 363–376.

[37] Zhong, Huasong, Chen, Chong, Jin, Zhongming, and Hua, Xian-Sheng. “Deep robust clustering by contrastive learning”. In: *arXiv preprint arXiv:2008.03030* (2020).