Symmetric Contrastive Learning On Programming Languages

A THESIS SUBMITTED TO
THE FACULTY OF ENGINEERING
OF THE UNIVERSITY OF SYDNEY
IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF PHILOSOPHY

XIAOHUAN PEI

Supervisor: Dr. Chang Xu
School of Computer Science
Faculty of Engineering
The University of Sydney
Australia

10 July 2023

Authorship Attribution Statement

The work presented in this thesis is published as a conference paper [1] in the 22nd The IEEE International Conference on Data Mining (ICDM), 2022. I designed the study, analysed the data and wrote the drafts of the paper.

Student Name:

Date: 31 March 2023

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Supervisor Name:

Date: 31 March 2023

Symmetric Contrastive Learning On Programming Languages

Xiaohuan Pei (Email: xpei8318@uni.sydney.edu.au)

Supervisor: Dr. Chang Xu

School of Computer Science

Faculty of Engineering

The University of Sydney

Statement of Original Authorship

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes. I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Signature:

To Those Whom I love & Those Who Love Me.

Abstract

Contrastive pre-training has been shown to learn good features by finding the inner difference and similar latent traits among the samples. The pairwise data Programming Languages (PL) and Natural Language (NL) also have strong inner-relationship that can be used on the downstream tasks.

Pre-trained models for Natural Languages (NL) have been recently shown to transfer well to Programming Languages (PL) and primarily benefit different intelligence code-related tasks, such as code search, clone detection, programming translation and code document generation. However, existing pre-trained methods for programming languages are mainly conducted by masked language modelling and next-sentence prediction at token or graph levels. This restricted form limits their performance and transferability since PL and NL have different syntax rules and the downstream tasks require a multi-modal representation. Here we introduce **C3P**, a **C**ontrastive **C**ode-**C**omment **P**re-training approach, to solve various downstream tasks by pre-training the multi-representation features on both programming and natural syntax. The model encodes the code syntax and natural language description (comment) by two encoders and the encoded embeddings are projected into a multi-modal space for learning the latent representation. In the latent space, C3P jointly trains the code and comment encoders by the symmetric loss function, which aims to maximize the cosine similarity of the correct code-comment pairs while minimizing the similarity of unrelated pairs. We verify the empirical performance of the proposed pre-trained models on multiple downstream code-related tasks. The comprehensive experiments demonstrate that C3P outperforms previous work on the understanding tasks of code search and code clone, as well as the generation tasks of programming translation and document generation. Furthermore, we validate the transferability of C3P to the new programming language which is not seen in the pre-training stage. The results show our model surpasses all supervised methods and in some programming language cases even outperforms prior pre-trained approaches.

Keywords

Contrastive Pre-Training, Programming Language Processing, Deep Learning, Code Search, Code Clone, Code Translation, Document Generation.

Acknowledgements

I would like to express my deepest gratitude to a multitude of individuals who have provided invaluable support, guidance, and encouragement throughout the duration of my Mphil journey. Without their unwavering assistance, the completion of this thesis would not have been possible.

First and foremost, I would like to express my sincere appreciation to my supervisor, Professor Chang Xu, for his exceptional mentorship, patience, and guidance. His vast knowledge and enthusiasm for research have been a continuous source of inspiration and motivation for me. I am truly grateful for the countless hours he have devoted to reviewing my work, providing constructive feedback, and helping me navigate the challenges of academic research.

I would also like to extend my heartfelt thanks to my fellow lab mates and colleagues in the AI lab in Sydney for their camaraderie, stimulating discussions, and invaluable insights. Their support and friendship have made the journey enjoyable and fulfilling.

In conclusion, I am deeply grateful to everyone who has contributed to the successful completion of this thesis. This journey would not have been possible without the support, guidance, and encouragement of these wonderful individuals, and I am forever indebted to them.

Table of Contents

CHAPTER 1

Introduction

In the introduction, we discuss the growing interest in Programming Language Processing and its importance in various code-related tasks, the challenges faced in applying deep learning and natural language processing techniques to programming languages, and the contributions made by our novel contrastive pre-training algorithm that effectively addresses these challenges and significantly improves performance on various downstream tasks.

1.0.1 Introduction of Programming Language Processing

Programming language processing for code-related tasks has attracted rising attention from the deep learning community, given its huge potential to build AI-driven coding applications. Such applications include vulnerability detection using deep learning-based systems [2], intelligent code hints for programming learners [3], source code modeling and generation for developers [4], and API modeling for recommendations [5]. The remarkable success of pre-trained models in natural language processing [6, 7] has served as an inspiration for various attempts to advance the development of pre-trained models specifically designed for programming languages. Some notable examples of these models include Code2Seq [8], CuBERT [9], GraphCodeBERT [10], Codex [11], PLBART [12], and CodeT5 [13]. The overarching concept behind these pre-training methodologies for programming languages revolves around predicting the original code from an artificially masked input code sequence or conducting code syntactic prediction by parsing textual and structural information derived from the abstract syntax tree. For instance, both Code2Seq[8] and CuBERT[9] utilized the objectives of mask language modeling (MLM) and next sentence prediction (NSP) in order to obtain a general representation of the special tokens during the pre-training phase.

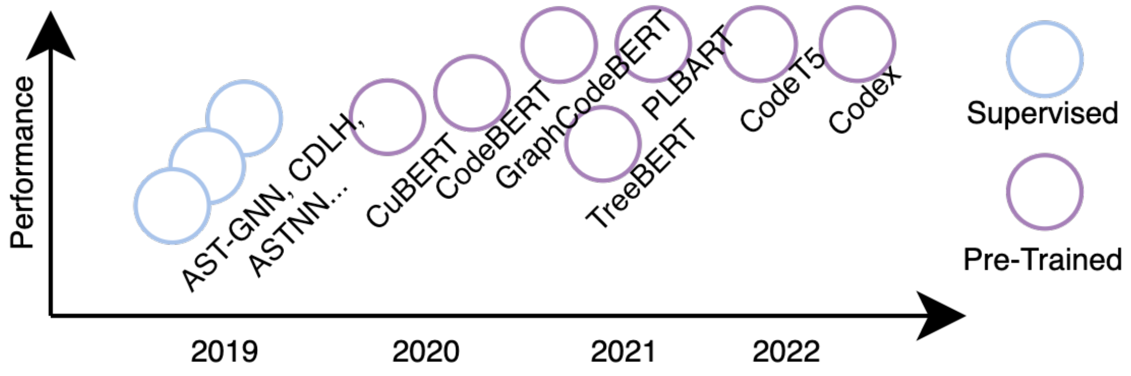


FIGURE 1.1. Progress on Code-Related Tasks.

CodeBERT [14] introduced the objective of replaced token detection (RTD), which leveraged both bimodal and unimodal data for training purposes. In contrast, GraphCodeBERT [10] placed greater emphasis on the structural information of code by training the objectives of Edge Prediction (EP) and Node Alignment (NA). These innovative approaches have contributed to the ongoing development and refinement of pre-trained models in the realm of programming language processing, opening up new possibilities for AI-driven coding applications and further enhancing the capabilities of deep learning techniques within this domain.

1.0.2 Introduction of Pre-Train and Fine-Tuning Paradigm

The pre-training and fine-tuning paradigm has become a widely adopted approach in the field of natural language processing and deep learning, revolutionizing the way researchers tackle various tasks [15]. This paradigm consists of two main phases: the pre-training phase, where models learn generic features from vast amounts of data, and the fine-tuning phase, where models are adapted to specific tasks using smaller, task-specific datasets [16, 6]. The introduction of the Transformer architecture by Vaswani et al. [6] laid the groundwork for the development of large-scale pre-trained models like BERT [17], GPT [15], and RoBERTa [7]. These models have demonstrated state-of-the-art performance on a wide array of natural language understanding and generation tasks, surpassing traditional task-specific architectures [18, 19].

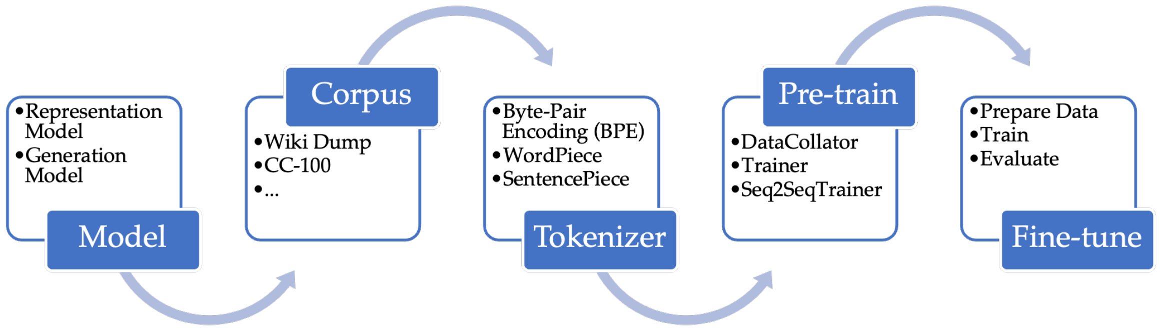


FIGURE 1.2. Overview of Pre-Train and Fine-Tuning Paradigm.

The success of the pre-training and fine-tuning paradigm can be attributed to the ability of models to leverage vast amounts of unsupervised data during pre-training, learning general language representations that can be fine-tuned for specific tasks [17, 15, 19]. This approach has not only led to significant improvements in performance across various tasks but has also reduced the need for large annotated datasets for training, lowering the barrier to entry for tackling new tasks and domains [20].

The pre-training and fine-tuning paradigm has been extended beyond natural language processing to other domains, such as computer vision [21], multimodal learning [22], and even programming languages [14, 10]. This widespread adoption of the pre-training and fine-tuning paradigm showcases its effectiveness in capturing generic features and adapting them to domain-specific tasks, leading to continuous advancements in various fields.

1.0.3 Introduction of Contrastive Pre-Training

Contrastive pre-training has recently emerged as a powerful and influential approach for learning effective and robust representations from a large volume of unlabeled data in various domains. This paradigm is based on the fundamental concept of contrastive learning, which revolves around the idea of learning representations that are closely aligned for semantically related instances, while maintaining a discernable distance for unrelated instances [23, 24]. The contrastive learning process is achieved by optimizing a contrastive loss function that encourages positive pairs (instances with similar semantic content) to be closer in the latent space, while simultaneously pushing negative pairs (instances with dissimilar semantic content) apart [25, 26]. Contrastive pre-training has been successfully applied across a wide range of domains, such as natural language processing [21, 27], computer vision [28, 29], and even programming language processing [14, 10]. The application of contrastive pre-training methods has yielded exceptional performance on numerous downstream tasks, often outperforming traditional supervised learning techniques and setting new state-of-the-art benchmarks. The success of contrastive pre-training is attributed to its ability to capture rich and meaningful representations from vast amounts of unlabeled data, making it highly effective for tasks that involve understanding complex patterns and relationships.

In the realm of natural language processing, contrastive pre-training has been leveraged to learn powerful language representations that generalize well across various tasks and domains [21, 27]. These methods have demonstrated significant improvements in tasks like text classification, sentiment analysis, and natural language understanding, surpassing previous methods that relied solely on supervised learning. In computer vision, contrastive pre-training has been utilized to learn robust visual features from large-scale image datasets, leading to notable advancements in image recognition, object detection, and segmentation tasks [28, 29]. Similarly, in the context of programming language processing, contrastive pre-training has shown promising results in tasks like code understanding, code generation, and natural language-to-code translation [14, 10]. The remarkable success of contrastive pre-training has inspired researchers to explore novel and more efficient contrastive learning strategies, as well as investigate ways to adapt the existing methods to other domains and modalities. As a result, various extensions and modifications to the original contrastive pre-training paradigm have been proposed, which have further advanced the field by introducing innovative approaches to learning powerful and meaningful representations from unlabeled data [30, 31].

In summary, contrastive pre-training has emerged as a highly effective approach for learning robust and meaningful representations from unlabeled data in various domains. Its success in natural language processing, computer vision, and programming language processing has demonstrated its potential to revolutionize the way we approach a wide range of downstream tasks. As research in this area continues to progress, we can expect the development of even more powerful and efficient contrastive pre-training methods that can further advance the state-of-the-art in multiple domains.

1.0.4 Introduction of Challenges in Programming Languages

The advent of deep learning and natural language processing (NLP) techniques has paved the way for numerous advancements in various domains, including programming languages processing. Although these techniques have shown tremendous potential in addressing complex tasks, several challenges remain in the effective application of deep learning and NLP methods to programming languages processing. In this introduction, we discuss some of the key problems faced by researchers and practitioners when utilizing deep learning or NLP techniques for programming languages processing, while citing relevant literature in the field.

One of the primary challenges in applying NLP methods to programming languages is the inherent differences between human languages and programming languages [32]. While natural languages are often ambiguous and context-dependent, programming languages have a more formal and structured nature, which requires specialized techniques for accurate processing and understanding [33]. Consequently, the development of effective deep learning models tailored to the unique characteristics of programming languages remains an ongoing challenge.

Another problem that arises when applying deep learning and NLP techniques to programming languages is the accurate representation and understanding of code semantics [8]. This challenge is rooted in the need to capture the underlying meaning and functionality of code snippets, which often necessitates an understanding of the syntactic structure and relationships between code elements. Graph-based representations, such as the abstract syntax tree (AST), have been employed to address this issue [10], but effectively incorporating such structural information into deep learning models remains a complex task. Code summarization and translation pose additional challenges in programming languages processing [34]. Given the diverse range of programming languages and their unique syntax, creating models capable of automatically generating accurate and concise natural language descriptions or translating code between languages requires a comprehensive understanding of both the source and target languages [35, 14]. Developing models that can perform these tasks effectively and efficiently is an ongoing area of research. The lack of high-quality, large-scale datasets for training and evaluation is another obstacle faced by researchers in this domain [4]. While there has been a surge of interest in developing pre-trained models for programming languages, such as CodeBERT [14] and CodeT5 [13], the availability of extensive and diverse datasets is crucial for achieving state-of-the-art performance and generalization.

In conclusion, while deep learning and NLP techniques have made significant strides in programming languages processing, several challenges persist. Addressing these issues and overcoming the inherent complexities of programming languages will be critical for the continued development and refinement of AI-driven coding applications and the effective utilization of deep learning and NLP methods in this domain.

1.0.5 Introduction of Contributions of Our Paper

In this thesis, we introduce a novel contrastive pre-training algorithm specifically designed to address the encoding challenges associated with programming languages and natural languages. It is a natural assumption to regard code and comments as two distinct views of a single instance. Rather than merely concatenating their intermediate encodings for the purpose of optimizing self-supervised objectives, such as masked language modeling, we aim to ensure that these two views are effectively aligned with each other within the latent space. By maximizing the agreement between the two views through the implementation of a contrastive loss, the resulting embeddings can be optimally fine-tuned to capture the inherent connections that exist between the views.

In addition to the aforementioned views, structural code information serves as another critical perspective that can be employed to describe the semantic information contained within the code. This information can be obtained by parsing data flow information from the source code using a depth-first search (DFS) approach and subsequently labeling the dependency relations of variables. Given the redundancy that exists between the source code and the code structure, we utilize a shared code encoder to map the concatenated textual and structural code to code embeddings. Meanwhile, the comment information, which originates from an entirely different natural language domain, maintains its own dedicated comment encoder.

We thoroughly evaluate our proposed model across four downstream tasks, including natural language code search, clone detection, code translation, and document generation. These tasks encompass the understanding and generation capabilities of the model with respect to code syntax. Our model significantly surpasses previous works in terms of performance on these fundamental downstream tasks. In an effort to further investigate the generalization potential of the learned representations, we establish zero-shot learning tasks by introducing the evaluation of new programming languages that have not been encountered during the training process. The results of this evaluation reveal that the proposed contrastive pre-training approach competes favorably with supervised methods that have been pre-trained on the entire spectrum of programming languages.

CHAPTER 2

Literature review

Our contribution is related to prior works about code pre-training, contrastive learning and fundamental code tasks. We start by reviewing programming language pre-training, after which we continue with contrastive methods for pre-training. Lastly, we introduce the fundamental code-related tasks.

2.0.1 Programming Language Pre-Training

Over the past few years, there has been a growing interest in pre-training methods for programming language models. These methods aim to improve the performance of models on various natural language processing and code-related tasks. Among the various pre-training methods, CodeBERT has gained a lot of attention in recent times. CodeBERT is a pre-trained model that can perform both code and natural language processing. The model is pre-trained using a masked language modeling objective on a large corpus of code and natural language text. One of the advantages of CodeBERT is that it can handle code and natural language jointly, which is particularly useful for tasks such as code summarization, where the model needs to understand both the code and the natural language description.

Another promising approach for pre-training programming language models is GraphCode-BERT. This method represents code using a graph neural network and pre-trains the model using both a masked language modeling objective and a graph reconstruction objective. By using graph representations, GraphCodeBERT can capture the structural information of code, which can be useful for tasks such as code clone detection.

In addition to these methods, Yuan et al. proposed a pre-training method for Transformers that uses an energy-based denoising objective to pre-train the model on a large corpus of images. Although this method is not specifically designed for programming language models, it could potentially be adapted to code by treating code as an image.

Kotikalapudi et al. proposed a pre-training method that uses a masked language modeling objective and a contrastive learning objective to pre-train the model on a large corpus of source code. This method aims to capture the semantics of code and the relationships between code elements. By pre-training the model on a large corpus of source code, it can learn to represent code more effectively and accurately.

Finally, Chang et al. proposed a pre-training method that uses a masked language modeling objective and a task-specific fine-tuning objective. The model is pre-trained on a large corpus of source code and fine-tuned on several downstream tasks. By fine-tuning the model on specific tasks, it can adapt to the specific requirements of those tasks and improve its performance.

Early research in code tasks mainly adopted supervised or self-supervised learning on programming language datasets. Inspired by the success of the 'pre-train+fine-tune' paradigm on natural language processing (NLP), many recent studies attempted to transfer these pre-training methods for p rogramming language tasks, such as Code2Seq [8], CodeBERT [14] and GraphCodeBERT [10]. Most of them directly utilized source code for pre-training [14, 8], and some of them considered exploiting the structural logic of the code [10, 36]. Pre-training objectives in these works were mainly designed by masking some textual or structural information such as mask token modeling, and predicting some elements in the source code such as edge prediction on the code graph parsed by an abstract syntax tree (AST) [10]. Overall, pre-training methods for programming language models have shown promising results in improving the performance of models on various natural language processing and code-related tasks. By using a large corpus of code and natural language text, these methods can effectively capture the semantics of code and natural language and learn to represent them more effectively.

2.0.2 Contrastive Learning for Pre-Training

Recently, contrastive learning methods for pre-training [37, 38, 39] have shown great potential in improving model robustness when fine-tuned for downstream tasks. The model pre-trained with contrastive learning also has a surprising zero-shot transfer ability [39, 38], which aims to learn useful representations for downstream tasks by training the model to distinguish between similar and dissimilar examples. One of the earliest works on contrastive learning is proposed by Krizhevsky et al. [40], who introduced the concept of non-parametric instance discrimination and proposed the contrastive predictive coding method.

Since then, many researchers have proposed various contrastive learning methods for pre-training models in different domains. For example, He et al. [28] proposed momentum contrast for pre-training visual representations, while Chen et al. [21] introduced the SimCLR framework for contrastive learning of visual representations. Joshi et al. [41] proposed the MARGE method, which combines contrastive learning with language and image encodings. Beltagy et al. [42] proposed contrastive learning for prompt tuning, which is a pre-training method for prompt-based language models. Chen et al. [43] propose a hierarchical encoder-decoder architecture that synthesizes visualization programs from natural language utterances and code context, trained on Jupyter notebooks from GitHub.

Contrastive text representation pre-training has been widely studied from both supervised and self-supervised perspectives [37]. For supervised contrastive pre-training [44], the input views can be constructed by manual text augmentation [45], or text summarization [46]. For self-supervised contrastive pre-training, the input views can be constructed via automated text augmentation [47], or next/surrounding sentence prediction [48]. There were also research works on cross-modal contrastive representation pre-training [38, 39], where image and text description information were fused into the same representation space.

Overall, contrastive learning has shown promising results in learning discriminative representations that capture the underlying structure of the data. By using a contrastive loss function, these methods can effectively learn representations that are useful for downstream tasks.

When we are making this submission, we note a contemporaneous work [49] on arXiv that also studies the code and text contrastive pre-training. However, our approach differs in four ways: 1) This work [49] only feeds in source code tokens, while we incorporate code structure information to enrich the representation of the source code; 2) Mask rules are different. Beyond masking source tokens as in [49], we propose to further mask the variable sequence and nodes of data flow; 3) We not only evaluate the proposed contrastive training for code understanding tasks but also investigate its performance on the generation tasks via generation decoders; 4) The comparison methods and experiment settings for transferability are different. As the code semantic rule of each programming language could be different in the real world, we test the transferability of the pre-trained encoders over a new programming language that has not been used during training, rather than taking an ordinary zero-shot setting to predict new labels in classification. These four differences make our contribution a unique one to contrastive learning for code pre-training.

2.0.3 Downstream Code-Related Tasks

Our work mainly focuses on the following fundamental code-related tasks.

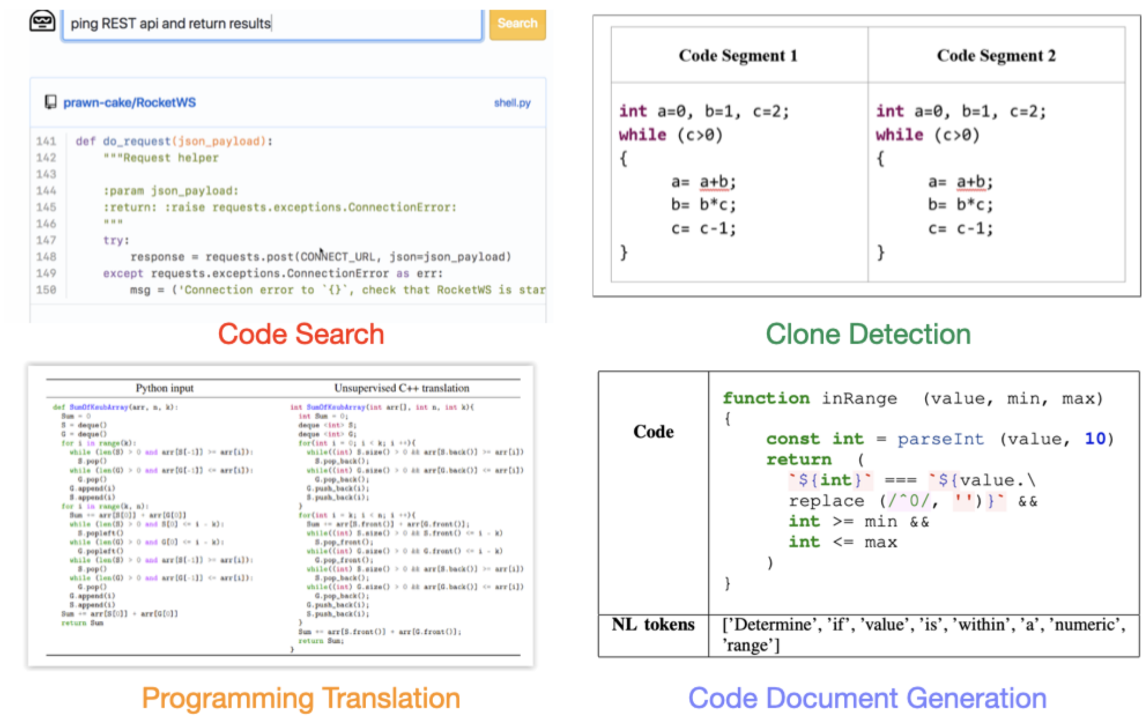
Code Search is the task of answering the natural language query with code snippets, which is also an important task in software engineering that aims to retrieve code snippets that are relevant to a given query. Text-based and code-based methods have been proposed for code search. Text-based methods use natural language queries and treat code as text, while code-based methods use code queries and represent code as structured data.

One example of a code search system that uses both text-based and code-based methods is CodeHow [50]. CodeBERT [51] is another code search system that uses a pre-trained model for programming and natural language processing. GraphCodeSearch [52] is a code search system that uses graph neural networks to represent code and performs semantic code search. The CodeSearchNet Challenge [53] is a benchmark dataset that evaluates the state of semantic code search, and several state-of-the-art models have been evaluated on the dataset. Allamanis et al. [54] proposed a neural network-based code search system that learns a joint embedding space for code and natural language queries.

Overall, code search is an important task in software engineering that has received significant attention in recent years. These approaches have shown promising results in retrieving relevant code snippets for a given query, using a combination of text-based and code-based methods. With the rapid growth of code repositories, code search is becoming an increasingly important task in software engineering. The fundamental problem of this task is modeling the latent correlation between the high-level inheritance in the natural language queries and the low-level semantic code context [55]. One line of related studies about code search mainly focused on supervised objectives [55], while the other line of studies followed the 'pre-trained+fine-tuning' paradigm [14, 10].

Code Clone is popular software engineering task that involves identifying code fragments that are similar or identical to each other. Many approaches have been proposed for code clone detection, including text-based and code-based methods. Text-based methods use natural language processing techniques to detect similar code fragments based on their textual similarity, while code-based methods use program analysis techniques to detect clones based on their syntactic and semantic similarity.

Several papers have proposed effective techniques for code clone detection. Li and Li [56] presented a comprehensive survey of code clone detection techniques, covering various types of clones and different detection methods. Jiang et al. [57] proposed a scalable code clone detection method that can efficiently detect clones in large codebases. Some recent papers have focused on deep learning-based approaches for code clone detection. White et al. [58] proposed a deep learning-based approach for clone detection that uses a Siamese neural network to learn code embeddings. Zhou et al. [59] proposed a self-supervised learning approach for code clone detection that uses a contrastive loss to learn representations for code fragments. Overall, code clone detection is an important task in software engineering that has received significant attention in recent years. These approaches have shown promising results in detecting code clones and can help developers identify and refactor duplicate code fragments.



```python
def do_request(json_payload):
    """Request helper

    :param json_payload:
    :return: :raise requests.exceptions.ConnectionError:
    """
    try:
        response = requests.post(CONNECT_URL, json=json_payload)
    except requests.exceptions.ConnectionError as err:
        msg = ('Connection error to `{}`, check that RocketWS is star
```

Code Segment 1	Code Segment 2

`int a=0, b=1, c=2;`
`while (c>0)`
`{`
`    a= a+b;`
`    b= b*c;`
`    c= c-1;`
`}`
	`int a=0, b=1, c=2;`
	`while (c>0)`
	`{`
	`    a= a+b;`
	`    b= b*c;`
	`    c= c-1;`
	`}`

Python input

```python
def SumOfKsubArray(arr, n, k):
  Sum = 0
  S = deque()
  G = deque()
  for i in range(k):
    while (len(S) > 0 and arr[S[-1]] >= arr[i]):
      S.pop()
    while (len(G) > 0 and arr[G[-1]] <= arr[i]):
      G.pop()
    G.append(i)
    S.append(i)
  for i in range(k, n):
    Sum += arr[S[0]] + arr[G[0]]
    while (len(S) > 0 and S[0] <= i - k):
      S.popleft()
    while (len(G) > 0 and G[0] <= i - k):
      G.popleft()
    while (len(S) > 0 and arr[S[-1]] >= arr[i]):
      S.pop()
    while (len(G) > 0 and arr[G[-1]] <= arr[i]):
      G.pop()
    G.append(i)
    S.append(i)
  Sum += arr[S[0]] + arr[G[0]]
  return Sum
```

Unsupervised C++ translation

```cpp
int SumOfKsubArray(int arr[], int n, int k){
  int Sum = 0;
  deque <int> S;
  deque <int> G;
  for(int i = 0; i < k; i ++){
    while((int) S.size() > 0 && arr[S.back()] >= arr[i])
      S.pop_back();
    while((int) G.size() > 0 && arr[G.back()] <= arr[i])
      G.pop_back();
    G.push_back(i);
    S.push_back(i);
  }
  for(int i = k; i < n; i ++){
    Sum += arr[S.front()] + arr[G.front()];
    while((int) S.size() > 0 && S.front() <= i - k)
      S.pop_front();
    while((int) G.size() > 0 && G.front() <= i - k)
      G.pop_front();
    while((int) S.size() > 0 && arr[S.back()] >= arr[i])
      S.pop_back();
    while((int) G.size() > 0 && arr[G.back()] <= arr[i])
      G.pop_back();
    G.push_back(i);
    S.push_back(i);
  }
  Sum += arr[S.front()] + arr[G.front()];
  return Sum;
}
```

Code	`function inRange (value, min, max)`
	`{`
	`    const int = parseInt (value, 10)`
	`    return (`
	`      ` `` `${int}` === `${value.\` ``
	`      replace (/^0/, '')}` && `<br>` int >= min &&`
	`      int <= max`
	`    )`
	`}`

NL tokens	['Determine', 'if', 'value', 'is', 'within', 'a', 'numeric', 'range']

FIGURE 2.1. Introduction of Code-Related Tasks.

Code Translation aims to build a code-to-code translator that converts source code from an abstract programming language (such as Java or Python) to another [60]. It involves converting code written in one programming language to another programming language while preserving its functionality. Many approaches have been proposed to tackle this problem, including rule-based and machine learning-based methods. One approach that has been extensively used for code translation is statistical machine translation. Ragavan et al. [61] proposed a statistical machine translation approach for code translation that leverages parallel corpora of code in different languages. Similarly, Srivastava et al. [62] proposed a method for translating code using phrase-based machine translation models.

Deep learning-based approaches have also been proposed for code translation. For example, Gu et al. [63] proposed a neural machine translation approach that uses a sequence-to-sequence model with attention to translate code. Xu et al. [64] proposed a neural machine translation approach for translating code comments, which can help improve code readability and understandability. These approaches have shown promising results in translating code across different programming languages. By enabling developers to work with code in their preferred programming language, these methods can improve their productivity and reduce the time and effort required for software development.

Document Generation is an intelligence task to auto-generate the natural language summary based on the given code snippet. Many approaches have been proposed to tackle this problem, including template-based methods and machine learning-based methods. One approach that has been widely used for document generation is template-based methods. These methods use pre-defined templates to generate documents based on the input data. Zhang et al. [65] proposed a joint template-based approach that uses both content and layout templates to generate documents that have both accurate content and a pleasing visual appearance. Machine learning-based approaches have also been proposed for document generation. Wang et al. [66] proposed a code-to-sequence model that can generate natural language descriptions of code. Similarly, Lebret et al. [67] proposed a neural document model that can generate text documents from structured data.

Recent advancements in deep learning have also led to the development of more sophisticated models for document generation. Zhang et al. [68] proposed a transformer-based approach that can generate coherent and diverse paragraphs from a given topic. Li et al. [69] proposed an unsupervised neural document generation approach that can generate realistic and informative documents without using any training data. And document generation is an active area of research in natural language processing, and these approaches have shown promising results in generating documents from structured data. By automating the document generation process, these methods can help improve productivity and reduce the time and effort required for creating high-quality documents.

Chapter 3

Methods

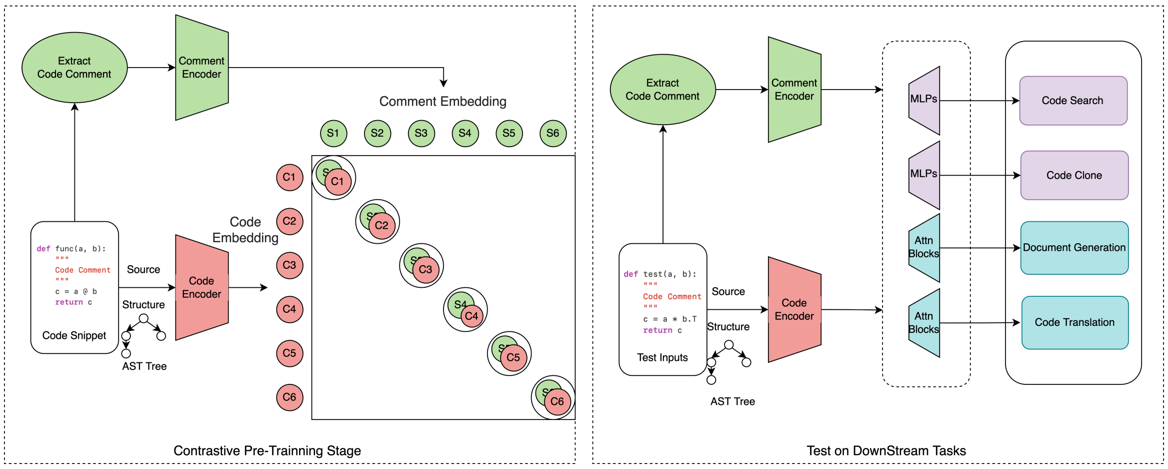
3.1 Contrastive Code and Comment Pre-training

The proposed contrastive code and comment pre-training (C3P) learns the latent representations of code and comment by maximizing their agreement. Fig. 3.1 shows the framework of our proposed pre-training method and relevant downstream tasks. There are two stages in the framework. In the first pre-training stage, we extract the source code (programming language) and comment (natural language) from the given code snippet, which are then fed into the corresponding encoders. The embeddings of code and comment are further aligned in a multi-modal latent space by contrastive learning. In the second fine-tuning stage, we load the pre-trained encoders with additional decoders for different downstream tasks of code search, code clone, code translation and document generation.

3.1.1 Code and Comment Encoders

Consider a source code $C = \{c_1, c_2, \cdots, c_n\}$ with n tokens and its comment $W = \{w_1, w_2, \cdots, w_m\}$ with m tokens. To incorporate the structural information of the source code into the embedding, we parse the code syntax with the standard compiler tool treesitter[1]. Our pre-processing mainly refers to the data flow information based on the variable relation graph $G(V, E)$ [10], where $V = \{v_1, v_2, ..., v_l\}$ represents the variable sequence identified from the leaves of abstract syntax tree (AST) by depth-first search (DFS), and E denotes the set of directed edges of V for modeling the dependency relation between variables.

[1]https://github.com/tree-sitter/tree-sitter



```
def func(a, b):
    """
    Code Comment
    """
    c = a @ b
    return c
```

```
def test(a, b):
    """
    Code Comment
    """
    c = a * b.T
    return c
```

FIGURE 3.1. Overview of Our Proposed C3P Framework. The framework comprises two stages: a code-comment matching pre-training stage and a code-related task fine-tuning stage. In the pre-training stage, we build code embeddings via a Code Encoder learned from textual tokens and structural tokens (nodes and edges) parsed from abstract syntax tree (AST) and we build comment embeddings via a Comment Encoder learned from natural language description. C3P compares the similarity of programming and natural language features for the match. In the fine-tuning stage, we utilize the pre-trained encoders and fine-tune our decoder networks for the downstream tasks. Both in pre-training and fine-tuning stages, the encoders are mapped with linear projection. In the ablation study, we also provide the performance on the Encoder-Only task without fine-tuning.

The structural variable sequence (V) can be then concatenated with the source code tokens to enrich the description of the code. To help C3P understand these two types of segments properly, we add $[CLS]$ as a special token for downstream classification tasks and $[SEP]$ as a symbol to separate different segments between textual and structural information. We denote the $[EOT]$ token as an extra end token in each input sequence. In the outputs of code and comment encoder, the activation on the last layer in the encoder for the $[EOT]$ token is treated as the feature representation of the sequence, which is then layer normalized and linearly projected into the multi-modal latent space for the subsequent contrastive learning. The reconstructed code information can be denoted as $X = \{[CLS], C, [SEP], V, [EOT]\}$, and the corresponding comment sequence can be written as $Y = \{[CLS], W, [EOT]\}$.

We plan to keep up to K_X valid tokens in each sequence X in a batch during the training, and thus a mask rule needs to be defined to handle the input sequences of different lengths:

$$[X\_MASK]_j = \begin{cases} 1, & j \leq \min(|X|, K_X), \\ 0, & else, \end{cases} \quad \forall 1 \leq j \leq |X| \tag{3.1}$$

where $[X\_MASK]_j$ denotes the binary mask value of the j-th token in the sequence X and $|X|$ is the length of the sequence. Similarly, we define the following mask rule for the sequence Y:

$$[Y\_MASK]_j = \begin{cases} 1, & j \leq \min(|Y|, K_Y), \\ 0, & else, \end{cases} \quad \forall 1 \leq j \leq |Y| \tag{3.2}$$

where $[Y\_MASK]_j$ denotes the binary mask value of the j-th token in the sequence Y, $|Y|$ is the length of the sequence, and K_Y is the comment sequence length to be processed by the encoder.

C3P aims to learn two BERT-based encoders (code encoder $\mathcal{F}_{\theta_X}$ and comment encoder $\mathcal{F}_{\theta_Y}$) parameterized by θ_X and θ_Y, to extract the latent feature representations of the code and comment sequences X and Y respectively. Following the design in [10], the network backbone of the encoders $\mathcal{F}(\cdot)$ is set as a 12-layer 512-max wide model with eight self-attention heads of BERT [17]. The C3P feeds the wrapped batch pairs (X, Y) to the encoders and derive the embeddings of the code and comment in the following process:

$$r_X \leftarrow \mathcal{F}(X \odot [X\_MASK]; \theta_X) \tag{3.3}$$

$$r_Y \leftarrow \mathcal{F}(Y \odot [Y\_MASK]; \theta_Y), \tag{3.4}$$

where $\odot$ stands for the mask operation between the binary mask and the corresponding sequence.

3.1.2 Contrastive Learning

The contrastive pre-training is designed to discover discriminative code-comment pairwise information in the projection space. We map each encoder with linear projection.

Assume X, Y represent the reconstructed code tokens and source comment tokens. Given a batch of size N of unlabeled samples $\{(X_i, Y_j)\}_{i,j=1}^{N}$, the C3P is responsible for predicting the correct pairs $\{(X_i, Y_j)\}_{i=j}$ and pulling unmatched pairs $\{X_i, Y_j)\}_{i\neq j}$ away. We adopt cosine similarity to measure similarity between the code latent features r_X and comment latent features r_Y as formulated by:

$$S(r_X, r_Y) = \frac{r_X^T r_Y}{\|r_X\|\|r_Y\|}, \tag{3.5}$$

where $\|\cdot\|$ is L_2 norm and $S(r_X, r_Y)$ represents the similarity between the code embedding and the comment embedding. For contrastive learning, we follow the symmetric loss settings of [38] and [70]. Let $\mathcal{L}_{X2Y}, \mathcal{L}_{Y2X}$ represent the code-to-comment loss and comment-to-code loss that employ logits softmax function on the latent representation. Given a batch of reconstructed codes and comments, the loss $\mathcal{L}_{X2Y}$ is defined to match each code to its corresponding comment:

$$\mathcal{L}_{X2Y} = -\sum_{i=1}^{N} \log \frac{exp(S(r_X^i, r_Y^i)/\tau)}{\sum_{j=1}^{N} exp(S(r_X^i, r_Y^j)/\tau)}, \tag{3.6}$$

where τ is the temperature parameter. Similarly, we have $\mathcal{L}_{Y2X}$ to match each comment to its corresponding code:

$$\mathcal{L}_{Y2X} = -\sum_{i=1}^{N} \log \frac{exp(S(r_X^i, r_Y^i)/\tau)}{\sum_{j=1}^{N} exp(S(r_X^j, r_Y^i)/\tau)}. \tag{3.7}$$

The resulting objective function for batch B is as follows:

$$\mathcal{L}_{(X,Y)\in B} = \frac{1}{2N}\left(\mathcal{L}_{X2Y} + \mathcal{L}_{Y2X}\right). \tag{3.8}$$

The optimization goal in the pre-training stage is to jointly maximize the similarity between paired code-comment while minimizing the cosine similarity of the projected latent features of the incorrect pairs.

The contrastive pre-training process can be summarized as Algorithm 1. With the parsing process in Algorithm 1 from line 5 to line 6, we reconstruct code information with textual tokens and structural graph variables by parsing abstract syntax tree. As the line 11 to line 14 of the pre-training process demonstrated, we employ the symmetric objective in the latent

Algorithm 1: Contrastive Pre-Training with Code (C) and Comment (W).

```
Data: Code-comment pairs datasets $\mathcal{D}$ comprised of batches
Result: Code and comment encoders $\mathcal{F}_{\theta_X}$, $\mathcal{F}_{\theta_Y}$ transferable to downstream tasks
1  Initialization of $\mathcal{F}_{\theta_Y}$ with words of BERT-case;
2  Initialization of $\mathcal{F}_{\theta_X}$ with code corpus;
3  for Each batch $B \in \mathcal{D}$ do
4      for Each $C, W \in B$ do
5          Reconstruct code and comment by parsing abstract syntax tree:
6          $X \leftarrow \{[CLS], C, [SEP], V, [EOT]\}$;
7          $Y \leftarrow \{[CLS], W, [EOT]\}$;
8          Encode reconstructed pairs with $[X\_MASK], [Y\_MASK]$ :
9          $r_X \leftarrow \mathcal{F}(X \odot [X\_MASK]; \theta_X)$;
10         $r_Y \leftarrow \mathcal{F}(Y \odot [Y\_MASK]; \theta_Y)$;
11     Get all reconstructed codes and comments: $\{(X_i, Y_j)\}_{i,j=1}^N$;
12     Compute similarities between encodings of codes and comments in the batch:
         $S(r_X^i, r_Y^j)$;
13     Compute symmetric loss with $\mathcal{L}_{X2Y}, \mathcal{L}_{Y2X}$: $\mathcal{L}_{(X,Y)\in B} = \frac{1}{2N}\left(\mathcal{L}_{X2Y} + \mathcal{L}_{Y2X}\right)$;
14     Joint optimization of the symmetric objective:
         $\operatorname{argmin}_{\{\theta_X,\theta_Y\}} \mathcal{L}_{(X,Y)\in B}\left(\{(X_i, Y_j)\}_{i,j=1}^N \middle| \mathcal{F}_{\theta_X}, \mathcal{F}_{\theta_Y}\right)$.
```

space for the two encoders to jointly learn the multi-modal representation. The dual-encoder structure utilized in our model has a time complexity of O($n^2 * d$) for n tokens of d dimensions each. Furthermore, during the contrastive learning phase, the complexity is O(n^2), given that each positive sample is compared with one matched code-comment pair and n-1 unmatched code-comment pairs.

3.1.3 Fine-Tuning

Fine-tuning is a vital process in which a pre-trained model is further trained on a smaller, task-specific dataset, enabling it to efficiently adapt its parameters to better accommodate the target domain. This method typically leads to enhanced performance, as the model is able to leverage the knowledge acquired during the pre-training phase and apply it to the new domain.

In the context of natural language processing tasks, fine-tuning plays a crucial role in adjusting the model to tackle the complexities and subtleties of human language. By training the model on datasets tailored to specific languages or domains, the model becomes increasingly proficient at comprehending and generating text that aligns with the desired language or domain characteristics.

Similarly, for programming language tasks, fine-tuning fulfills a comparable function. Pre-trained models undergo further training on datasets containing programming language code, allowing them to develop a more refined understanding, generation, and processing of code snippets. Consequently, the model becomes well-equipped to handle the syntax, structure, and semantics of programming languages, ultimately resulting in superior performance in tasks such as code completion, code search, and automated debugging. By fine-tuning the model for programming language tasks, it evolves into a potent tool that can significantly support developers in their daily work, boosting productivity and minimizing the risk of errors.

In the fine-tuning phase of the C3P model, specialized decoders are introduced for distinct downstream tasks, following the pre-trained encoders. Each encoder-decoder neural network is optimized using the AdamW optimizer in conjunction with a scheduler. This approach ensures that the model can adapt to the unique requirements of each task, resulting in a highly versatile and effective tool for addressing a wide range of programming language challenges. Through the fine-tuning process, the model's capabilities are honed to provide optimal performance across various tasks, domains, and programming languages.

Code Search. The backbone decoder for code search is of two fully connected layers. The last layer outputs a normalized probability over all set-id of candidate code snippets. We just fine-tuned one epoch for the task-specific decoder.

Code Clone. The backbone decoder for code clone is a three-layered MLP with the GELU activations that define what is to be fired to the next neural layer. The output of the decoder is a two-element vector that represents the probability distribution of positive and negative. We set only one epoch for fine-tuning following the same settings as [14, 10].

Code Translation. The decoder for translation is of a multi-head attention structure with 6 layers and 12 heads. The decoder is fed with both outputs of encoders on [EOT] tokens and the inputs $\{[CLS], V, [SEP], E, [SEP]\}$, $\{[X\_MASK]\}$ as shown in the sub-section 3.1.1.

We apply the auto-frozen strategy for fine-tuning on this generation task. If the model detects the loss has no sign of dropping on the validation dataset, the model will freeze all layers but the last layer of the encoder and fine-tune the parameters of the decoder.

Document Generation. The architecture of the decoder is of the same number of layers and hidden size as pre-trained models. The output of the decoder is a sequence of tokens that represents the natural language description. We fine-tuned 50 epochs, which follow the experimental pipeline in [14].

Model	**Encoder**			**Decoder of Understanding**			**Decoder of Generation**			
	#H	#D	#Bl	Task	#L	#W	Task	#H	#D	#Bl
C3P	12	768	6	S	2	{512, 128}	T	12	768	6
	12	768	6	C	3	{1024, 512, 64}	G	12	768	6

TABLE 3.1. The Backbone of Decoders for Downstream Understanding and Generation Tasks: Code Search (S), Clone Detection (C), Code Translation (T) and Document Generation (G). #L: number of fully-connected hidden layers. #W: width of each fully-connected hidden layer. #H: number of attention heads. #D: dimensions of each embedding layer. #Bl: number of multi-head attention blocks. We also provide small size on the settings of 3/6 on #H/#Bl and medium size on the settings of 6/6 on #H/#Bl for the ablation study 4.2.

Implementation Details. Table 3.2 shows the training recipe in the pre-training and fine-tuning stage. In the pre-training stage, we apply the same training recipe of [10] for C3P. All models are trained for 500 epochs with a batch size of 1024 optimized by AdamW [71]. The learning rate for the optimizer is $1e-3$ with cosine decay strategy. We transfer the pre-trained encoders for the understanding task by concatenating with MLP-based decoders and apply the pre-trained encoders fine-tuning on the generation tasks with attention-based decoders, as demonstrated in Table 3.1. Specifically, we freeze the pre-trained embeddings and fine-tune one epoch on each task for the understanding task, while we don't freeze the embeddings in the fine-tuning stage for generation tasks.

	EPs	BS	OP	LR	LD	WU	FE
Pre-Training Task	500	1024	AdamW	2e-4	cosine	10	-
Fine-Tuning on DoU	1	1024	AdamW	1e-6	-	-	✓
Fine-Tuning on DoG	100	1024	AdamW	2e-5	cosine	5	✗

TABLE 3.2. Pre-Training and Fine-Tuning Recipes of the C3P. CS: Whether adding code structure representation; DoU: Decoders of understanding tasks; DoG: Decoders of generation tasks; EPs: Epochs; BS: Batch size; OP: Optimizer; LR: Learning rate; LD: Learning decay; WU: Warm-up; FE: Freeze pre-trained embeddings.

CHAPTER 4

Results

4.1 Performance of C3P vs. Other Methods

	Model	Ruby	Javascript	Go	Python	Java	Php	Overall
Supervised	NBow [14]	0.162	0.157	0.330	0.161	0.171	0.152	0.189
	CNN [14]	0.267	0.224	0.680	0.242	0.263	0.260	0.324
	BiRNN [14]	0.213	0.193	0.688	0.290	0.304	0.338	0.338
	SelfAtt [14]	0.275	0.287	0.723	0.398	0.404	0.426	0.419
Pre-trained	RoBERTa [14]	0.587	0.517	0.850	0.587	0.599	0.560	0.617
	RoBERTa (code) [14]	0.628	0.562	0.859	0.610	0.620	0.579	0.643
	CodeBERT [14]	0.679	0.620	0.882	0.672	0.676	0.628	0.693
	GraphCodeBERT [10]	0.703	0.644	0.897	0.692	0.691	0.649	0.713
	PLBART [12]	0.675	0.616	0.887	0.663	0.663	0.611	0.685
	CodeT5 [13]	0.719	0.655	0.888	0.698	0.696	0.645	0.716
Ours	**C3P**	**0.756**	**0.677**	**0.906**	**0.704**	**0.759**	**0.684**	**0.748**

TABLE 4.1. Downstream Task 1: Code Search Performance Measured by Mean Reciprocal Rank (MRR).

We evaluate C3P on four fundamental tasks: code search, clone detection, code translation and document generations. All of them are compared with previous well-documented models on the same experimental datasets.

We include the following important baselines for pre-training in the comparison experiments:

- RoBERTa[7]: RoBERTa (Robustly optimized BERT pre-training approach) is an optimized version of BERT, introduced by Liu et al. (2019). The main improvements in RoBERTa over the original BERT model are the removal of the next sentence prediction (NSP) task, larger batch sizes, longer training, and the use of dynamic masking for the masked language model (MLM) objective. RoBERTa has shown state-of-the-art performance on a variety of natural language processing benchmarks like GLUE, RACE, and SQuAD.

- CodeBERT[14]: CodeBERT proposed by Feng et al. (2020), is a transformer-based model specifically designed for programming language tasks. CodeBERT is pre-trained on a large-scale dataset consisting of natural language text and code from various programming languages. The model is fine-tuned for various tasks, such as code summarization, code search, and code translation, achieving state-of-the-art results in several benchmarks.
- GraphCodeBERT[10]: GraphCodeBERT, introduced by Guo et al. (2021), is an extension of CodeBERT that incorporates graph-based representations to model code structure. The model integrates a graph neural network (GNN) with the transformer architecture, allowing it to capture both syntactic and semantic information from code. GraphCodeBERT has demonstrated superior performance in tasks like code summarization and code search compared to the original CodeBERT.
- PLBART[12]: It is proposed by Keskar et al. (2021), is a transformer-based model specifically designed for programming language understanding and generation tasks. It is pre-trained on a large multilingual source code dataset and leverages denoising autoencoding and sequence-to-sequence modeling to capture syntactic and semantic information in code. PLBART has achieved state-of-the-art performance on various code-related tasks, including code summarization, bug detection, and code translation.
- CodeT5[13], introduced by Hashimoto et al. (2021), is an extension of Google's T5 model, specifically designed for code-related tasks. The model is pre-trained on a large dataset containing a mix of natural language text and code from various programming languages. By leveraging the T5's text-to-text transfer approach, CodeT5 has demonstrated state-of-the-art performance on several programming language tasks, such as code summarization, code search, and code completion.

Besides these pre-trained methods, we also consider the representative supervised methods that have been particularly developed for each downstream task in the experiments.

4.1.1 Code Search

Code search task targets finding correct code snippets from a candidate set by giving a query of natural language description. We evaluate our model on the CodeSearchNet code corpus [72] and the filtering rules follow the settings of GraphCodeBERT [10] on this task. The dataset is composed of both uni-modal and bi-modal data. The statistics of dataset are shown in Table 4.2.

Since the decoder outputs a list of possible responses to a sample of queries ordered by the probability of matching, we select a widely used evaluation method, i.e., mean reciprocal rank (MRR), to measure the score. For each query q, the score is computed by:

$$MRR = \frac{1}{|C|} \sum_{q \in C} \frac{1}{rank_q} \tag{4.1}$$

where C is the candidate set of code snippets and $rank_q$ stands for rank list of query q.

The baselines for code search include supervised and unsupervised pre-trained models that are public and well documented. The first group of Table 4.1 shows the previous supervised experimental records on this dataset. NBow is the bag of words model that is particularly good at keyword matching on code. CNN and Bi-RNN were implemented on this task by [72]. The results of Self-Attention [6] obtain the best performance among the supervised models. And then, pre-trained models RoBERTa [7], CodeBERT [14], GraphCodeBERT [10], PLBART [12] and CodeT5 [13] recorded in the second group of Table 4.1 boost the performance on this task.

PL	**Train**	**Valid**	**Test**	**Candidates**
Python	251,820	13,914	14,918	43,827
Php	241,241	12,982	14,014	52,660
Go	167,288	7,325	8,122	28,120
Java	164,923	5,183	10,955	40,347
Javascript	58,025	3,885	3,291	13,981
Ruby	24,927	1,400	1,261	4,360

TABLE 4.2. Dataset of Code Search

However, as the last line of Table 4.1 shows, our proposed approach achieved better results as compared to other public recorded methods, with the improvement of 3.7%, 2.2%, 0.9%, 0.6%, 6.3%, 3.5% on the programming language Ruby, Javascript, Go, Python, Java, Php.

4.1.2 Clone Detection

Code clone detection, a crucial aspect of software development, involves assessing the extent to which two code snippets share the same functionality. In our study, we conducted a series of experiments utilizing the BigCloneBench dataset [73], a comprehensive collection of known true and false positive clones found within a large-scale inter-project Java repository. Our experimental approach involved comparing a diverse range of baseline methods, which included both supervised learning models and pre-trained models that underwent fine-tuning.

	Train	**Valid**	**Test**
Clone Pairs	$901,028$	$415,416$	$415,416$

TABLE 4.3. Dataset of Code Clone

In the realm of supervised baselines, we examined Deckard [74], RtvNN [75], CDLH [76], and ASTNN [77]. Deckard [74] is an algorithm that calculates the similarity degree between code snippets by analyzing the structure of the abstract syntax tree and employing locality sensitive hashing. RtvNN [75] makes use of an auto-encoder to reconstruct the abstract syntax tree, while CDLH [76] converts the abstract syntax tree into an LSTM network [78]. On the other hand, ASTNN [77] saves a portion of AST information using an RNN network and takes advantage of the naturalness of statements to generate a vector representation. FA-AST-GNN[79] employs two distinct types of graph neural networks (GNN) on the flow-augmented abstract syntax tree (FA-AST).

As for the pre-trained models, we considered RoBERTa [17], CodeBERT [14], GraphCode-BERT [10], PLBART [12], and CodeT5 [13]. Each model was fine-tuned using a multi-layer perceptron (MLP) decoder. We adopted Precision, Recall, and F1 as our evaluation metrics to gauge the performance of these models.

Model	Precision	Recall	F1
Deckard [14]	0.93	0.02	0.03
RtvNN [14]	0.95	0.01	0.01
CDLH [14]	0.92	0.74	0.82
ASTNN [14]	0.92	0.94	0.93
FA-AST-GNN [79]	0.96	0.94	0.95
RoBERTa (code) [14]	0.960	0.955	0.957
CodeBERT [14]	0.964	0.966	0.965
GraphCodeBERT [14]	0.973	0.968	0.971
PLBART [12]	-	-	0.972
CodeT5 [13]	-	-	0.972
C3P	**0.979**	**0.978**	**0.979**

TABLE 4.4. Downstream Task 2: Code Clone Performance Measured by Precision, Recall and F1.

Our findings, as illustrated in Table 4.4, demonstrate that our proposed C3P model outperforms both supervised and pre-trained baseline methods. In comparison to the top-performing supervised model, FA-AST-GNN [79], our model achieves a 1.9%, 2.8%, and 2.9% improvement in Precision, Recall, and F1, respectively. Moreover, as shown in the second group of Table 4.4, C3P emerges as the best-performing model among pre-trained approaches for this task, displaying enhancements of 0.6%, 1%, and 0.8% in the three evaluation metrics.

4.1.3 Code Translation

Code translation serves the essential purpose of migrating code functionality between different programming languages, thereby easing the burden of transitioning existing projects to a new programming language. To evaluate the efficacy of various methods, we use an experimental dataset and settings that adhere to the standards set forth in mppSMT [35], attention-based tree [80], CodeBERT [14], and GraphCodeBERT [10], as depicted in Table 4.5. We assess the performance of our C3P model in comparison to other previous models by reporting the standard generation metrics, BLEU-4 and accuracy, as shown in Table 4.6.

The first group of results presented in Table 4.6 pertains to traditional supervised methods. A simplistic approach involves directly copying the original code fragments without making any alterations. Moreover, we cite the results of the Transformer model [14] on this dataset, given the remarkable success of the attention mechanism in text processing. As demonstrated by the results in the first group, our model significantly surpasses the state-of-the-art supervised models, boasting a 23.07% and 23.34% improvement in the task of translating Java code to C-sharp, and a 26.73% and 26.73% improvement in the task of translating C-sharp code to Java. decoders [14, 10] to ensure fair comparisons between the models. The results reveal that the C3P model performs competitively when compared to prior work on the task of translating Java code to C-sharp. Furthermore, the C3P model notably outperforms all previous methods in the task of translating C-sharp code to Java, establishing its superior performance in this particular translation challenge.

	Train	**Valid**	**Test**
Translation Pairs	$10,300$	500	1000

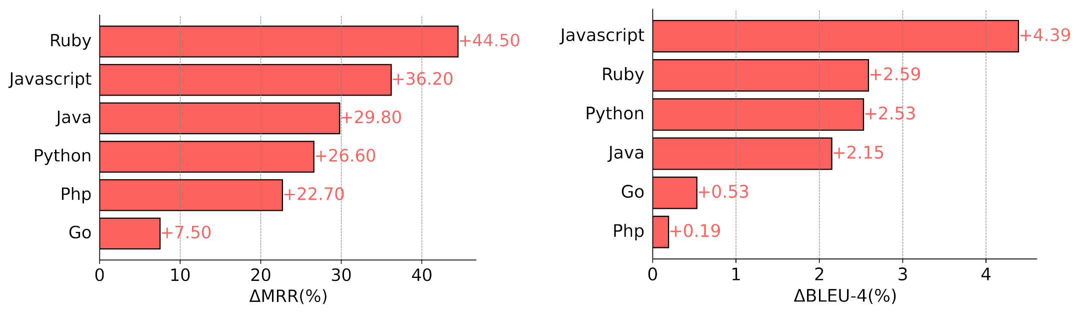
TABLE 4.5. Dataset of Code Translation

	Java → C#		**C# → Java**	
Model	**BLEU**	**Acc**	**BLEU**	**Acc**
Naive [14]	18.54	0.00	18.6	0.00
PBSMT [14]	43.53	12.50	40.06	16.10
Transformer [14]	55.84	33.00	50.47	37.90
RoBERTa (code) [14]	77.46	56.10	71.99	57.90
CodeBERT [14]	79.92	59.00	72.14	58.00
GraphCodeBERT [10]	**80.58**	59.40	72.64	58.80
C3P	78.91	**59.73**	**73.81**	**59.42**

TABLE 4.6. Downstream Task 3: Code Translation Performance Measured by BLEU-4 and Accuracy.

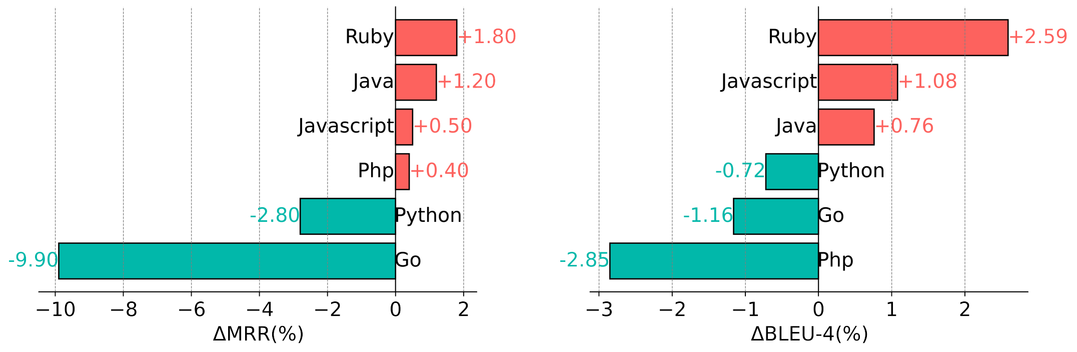
We report the standard generation metrics BLEU-4 and accuracy of the C3P against other previous models in Table 4.6. The first group is the results of traditional supervised methods. The naive method is to directly copy original code fragments without modifications. We also cite the Transformer model results [14] on this dataset since the success of the attention mechanism on text processing. The results listed in the first group demonstrate our model significantly outperforms supervised SOTA models with 23.07%, 23.34% improvement on the task of Java transferring to C-sharp, and 26.73%, 26.73% improvement on the task of C-sharp transferring to Java.

For the pre-training approaches, we employ the same decoders of multi-attention backbone as the previous generation decoders [14, 10] to make sure the results can be fairly compared. The results indicate that the C3P is competitive to previous works on the task of Java transferring to C-sharp while outperforming all previous methods evidently on the task of C-sharp transferring to Java.



(a) Code Understanding Task. (b) Code Generation Task

FIGURE 4.1. Transferability: Improvement of C3P (w/o PL) over supervised SOTA trained on full datasets including PL.



(a) Code Understanding Task. (b) Code Generation Task.

FIGURE 4.2. Transferability: Improvement of C3P (w/o PL) over pre-trained SOTA using full datasets including PL.

4.1.4 Code Document Generation

Model	Ruby	Javascript	Go	Python	Java	Php	Overall
Seq2Seq [14]	9.64	10.21	13.98	15.93	15.09	21.08	14.32
Transformer	11.18	11.59	16.38	15.81	16.26	22.12	15.56
RoBERTa [14]	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT (RTD) [14]	11.42	13.27	17.53	18.29	17.35	24.10	17.00
CodeBERT (MLM) [14]	11.57	14.41	17.78	18.77	17.38	24.85	17.46
CodeBERT (MLM+RTD) [14]	12.16	14.90	18.07	19.06	17.65	25.16	17.83
C3P	**14.67**	**14.98**	**18.43**	**19.12**	**18.09**	**25.88**	**18.52**

TABLE 4.7. Downstream Task 4: Code Document Generation Performance Measured by Smoothing BLEU-4.

The objective of document generation is to automatically generate concise natural language descriptions of code snippets, effectively summarizing their functionality. Given the robust and well-documented foundation provided by CodeBERT [14], we have chosen to adopt their experimental datasets and settings for our analysis. Table 4.7 presents a comparative assessment of our C3P model's performance in relation to earlier pre-training efforts.

The Seq2Seq model [81] makes use of a multilayered recurrent neural network to map input sequences to fixed-length vectors and subsequently decodes these latent vectors into target sequences. This model has been successfully adapted to the code2seq approach [8], which harnesses the syntactic structure of programming languages to translate them into natural language sequences.

As illustrated in Table 4.7, the performance of document generation is evaluated using the BLEU-4 metric. Our C3P model demonstrates a significant improvement in performance when compared to the best results achieved by previous works across six distinct programming languages. This clearly showcases the potential of our model in effectively summarizing code snippets and generating accurate natural language descriptions, thereby outperforming its predecessors in the realm of document generation.

4.1.5 Transferability

To study the transferability of our model to new programming languages, we try to train the model C3P (w/o PL) by removing one of the programming languages (PLs) from the

	Model	Ruby	Javascript	Go	Python	Java	Php	Overall
Understanding (MRR)	Supervised SOTA [14]	0.275	0.287	0.723	0.398	0.404	0.426	0.419
	Pre-trained SOTA [10]	0.703	0.644	**0.897**	**0.692**	0.691	0.649	0.713
	C3P (w/o Ruby)	0.721	0.662	0.873	0.679	**0.737**	0.659	0.721
	C3P (w/o Javascript)	**0.739**	0.649	0.885	0.681	0.705	**0.672**	**0.722**
	C3P (w/o Go)	0.727	0.637	0.798	0.675	0.694	0.661	0.698
	C3P (w/o Python)	0.716	0.648	0.865	0.664	0.693	0.655	0.706
	C3P (w/o Java)	0.732	0.625	0.874	0.680	0.703	**0.672**	0.714
	C3P (w/o Php)	0.719	**0.652**	0.884	0.690	0.677	0.653	0.712
Generation (BLEU-4)	Supervised SOTA [14]	11.18	11.59	16.38	15.81	16.26	22.12	15.56
	Pre-trained SOTA [10]	12.16	14.90	18.07	19.06	17.65	25.16	17.83
	C3P (w/o Ruby)	**13.77**	15.31	17.41	17.81	16.32	**25.73**	17.72
	C3P (w/o Javascript)	12.08	15.98	18.27	**19.35**	**18.51**	25.49	**18.28**
	C3P (w/o Go)	13.36	**16.14**	16.91	16.13	17.92	23.52	17.33
	C3P (w/o Python)	12.79	14.13	18.47	18.34	17.24	24.85	17.63
	C3P (w/o Java)	12.84	14.67	**18.94**	17.44	18.41	23.10	17.56
	C3P (w/o Php)	13.02	13.39	17.91	18.29	16.74	22.31	16.94

TABLE 4.8. Transferability. C3P (w/o PL) is compared with the best available supervised SOTA and pre-trained SOTA.

pre-training and fine-tuning set. For example, C3P (w/o Python) means the model pre-trained and fine-tuned on the datasets without Python codes, which is then evaluated on Python datasets in downstream tasks. Since C3P (w/o Python) has not seen the code style of Python before the test stage, the transferability of the model (to Python) can be evaluated through the test performance. As shown in Table 4.8, we test the transferability of six programming languages by removing one language at a time in the training stage. We show the performance on two downstream tasks: code search (code understanding) and document generation (code generation), and compare the results with fully supervised / pre-trained SOTA models. For code understanding task, we adopt the same evaluation scripts of MRR as [10]. For code generation task, we adopt the same evaluation function BLEU-4 as [14].

Fig. 4.1 and Fig. 4.2 illustrate the improvement brought by our models. We first compare C3P (w/o PL) with fully supervised SOTA models trained on the whole datasets including the specific PL language. As shown in Fig. 4.1 (a), in the code understanding task, the scores of our models without prior PL information achieve considerable enhancement, respectively increased by 44.50%, 36.20%, 29.80%, 26.60%, 22.70%, 7.50%. Also, Fig. 4.1 (b) demonstrates that in the code generation task, C3P (w/o PL) surpasses all supervised specific-task models with different degrees of improvement, with margins from 4.39% to 0.19% on the six programming languages.

We then compare the transfer performance of C3P (w/o PL) with SOTA fine-tuned models pre-trained on the whole datasets including PLs. Fig. 4.2 (a) shows the comparison of MRR scores on the six programming languages in the code understanding task. We observe that our models achieve enhancement of 1.8%, 1.2%, 0.5%, 0.40% on Ruby, Java, Javascript, Php tasks. However, without prior information on Python and Go, the performance of the model falls behind the model pre-trained and fine-tuned on the whole datasets that include these two languages. Fig. 4.2 (b) summarises the comparison in the code generation task. Without adding PLs in the pre-training stage, our models are still competitive with fully pre-trained models on Ruby, Javascript and Java tasks, while on Python, Go and Php tasks, our models achieve slightly worse results than fully pre-trained CodeBERT [14].

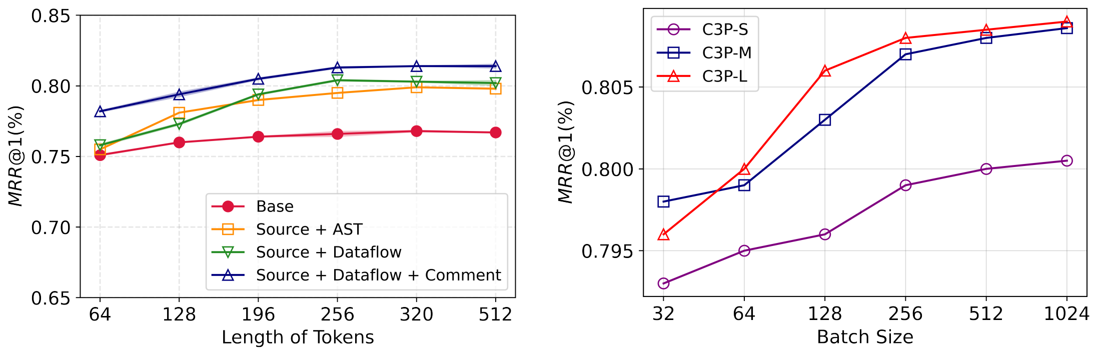
4.2 Ablation Study

We present the performance of C3P vs. other methods on a variety of downstream tasks and then analyze the transferability of the models pre-trained without one of programming languages in section 4.1. In this section, we provide more ablation study results by removing certain components and compare with other training strategies on the same experimental settings as [10].

Impact of Token Length and Batch Size. The efficacy of contrastive learning tends to be significantly affected by factors such as the length of the input data and the batch size utilized during training, as demonstrated by Conde et al. [38]. In order to thoroughly investigate the optimal hyperparameters for various learning strategies, we conducted a comprehensive series of experiments. To eliminate the potential impact of specific decoders, we assessed our approach without any fine-tuning by focusing on the Encoder-Only Task (Code Search).

Figure 4.3(a) illustrates the influence of token length on performance when employing different representation techniques. Through our analysis, we found that incorporating code structure information into the learning process results in noticeable enhancements in the code search task's outcomes. Additionally, when a comment encoder is integrated into the process, the performance experiences a further boost, emphasizing the value of this approach.

In Figure 4.3(b), we explore the effects of varying batch sizes on the performance of small, medium, and large variants of the C3P model. It is important to note that the large-size backbone configuration remains consistent with the experimental settings outlined in section 4.1. Our findings reveal that increasing the batch size during the pre-training phase has a positive impact on the performance of differently sized pre-trained models. This suggests that the choice of batch size is a crucial factor to consider when aiming to optimize the effectiveness of contrastive learning in various contexts.



(a) Impact of Token Length with Multi-Modalities Representation. (b) Impact of Batch Size on Different Size of Encoders

FIGURE 4.3. Ablation Study on Input Length and Batch Size. Tested on the code search task by pre-trained encoders WITHOUT fine-tuning. The score is measured by MRR(%). C3P-S/C3P-M/C3P-L represent the small/medium/large size of backbone presented in Table 3.1 respectively.

Fusion Strategy and Impact of Fine-Tuning. In order to thoroughly examine the fusion of heterogeneous features of code and text, we conduct a comprehensive validation of our proposed method in comparison with the approach of directly concatenating inputs for a single encoder without distinguishing between code and comments, as presented in Guo et al.'s GraphCodeBERT [10]. This comparative analysis aims to investigate the benefits of effectively combining code and natural language features within our model's architecture.

To ensure that the observed improvements in performance are attributable to the pre-trained models themselves and not to specific decoders on the downstream task, we also carefully assess the impact of our proposed models without employing fine-tuning by task-specific

Strategy	CS	WU	FT	FE	Valid		Test	
					MRR@1	MRR@5	MRR@1	MRR@5
Single Encoder	✓				72.47	78.19	69.26	75.47
		✓			70.28	81.27	68.05	73.29
	✓		✓		75.83	84.18	71.02	80.36
	✓		✓	✓	75.12	87.42	73.06	85.39
	✓	✓	✓	✓	80.17	88.42	73.53	82.95
Two Encoders	✓				74.15	82.09	69.46	76.54
		✓			73.92	78.44	70.25	78.23
	✓		✓		78.37	88.02	71.98	86.32
	✓		✓	✓	76.26	88.06	74.24	85.73
	✓	✓	✓	✓	82.04	90.07	74.80	87.85

TABLE 4.9. Ablation Study. Performance comparisons with encoding strategy on the validation set and test set. CS: Whether adding code structure representation; WU: Warp-up; FT: Fine-tuning; FE: Freeze pre-trained embeddings. Each model with ✓ in the FT option is fine-tuned for just one epoch.

decoders. This evaluation helps to eliminate potential confounding factors and provides a clearer understanding of the inherent strengths of our pre-training approach.

The experimental results, as documented in Table 4.9, consistently demonstrate that our pre-training method outperforms the single encoder approach across both the validation set and the test set. This finding suggests that our model is capable of effectively leveraging the distinct features of code and natural language, resulting in superior performance on various tasks.

Moreover, we observe that the C3P model exhibits a more pronounced increase in accuracy scores when provided with prior knowledge of natural language descriptions, particularly when considering the acceptance rate from top 1 to top 5 (MRR@1 vs. MRR@5). This result indicates that our model is not only proficient in combining code and text features but also adept at capitalizing on the additional information provided by natural language descriptions, thereby leading to a more robust and versatile model capable of handling a wide range of tasks.

4.3 Case Study

In this section, we showcase four distinct cases to exemplify the performance of our proposed C3P model on various downstream tasks. In the code search task, as demonstrated in Table 4.10, C3P effectively identifies the correct code fragment corresponding to the query, which aims to extract the video ID from a given URL. With regards to code clone detection, illustrated in Table 4.11, C3P accurately discerns the subtle differences in the keywords **a'** and **b'** present in the two code snippets, resulting in a relatively low similarity score that reflects their dissimilarity.

In the context of the code translation task, we present a case study in Table 4.12 where the C3P model translates a Java code fragment to its C-sharp equivalent. The model effectively captures the nuances by incorporating the definition of the type variable in the translated code snippet. Finally, for the document generation task, as illustrated in Table 4.13, C3P produces an accurate and concise natural language description that effectively summarizes the functionality of the given code, which is to determine if a specific value falls within a particular range. These four cases demonstrate the robust performance of C3P across a diverse set of tasks, highlighting its versatility and effectiveness in handling different programming language processing challenges.

Query	Extracts video ID from URL.

Code

```python
import utils.match as match
def get_vid_from_url (url):
    path1='youtu\.be/ ([^?/]+)'
    path2='youtube\.com/emb/([^/?]+)'
    path3='youtube\.com/v/ ([^/?]+)'
    return match (url, path1) or\
           match (url, path2) or\
           match (url, path3)
```

TABLE 4.10. A Case of Code Search Output by C3P.

Code1	Code2

```
import numpy as np
def f (array):
    a=np.sum (array)
    b=np.mean (array)
    return b
```
	```
	import numpy as np
	def f (array):
	    a=np.sum (array)
	    b=np.mean (array)
	    return a
	```

similarity=0.1	

TABLE 4.11. A Case of Code Clone Output by C3P.

Java	
	```
	public void removePresentation\
	    Format ()
	    {
	    remove1stProperty (PropertyIDMap\
	    .PID_PRESFORMAT);
	}
	```

C-sharp	
	```
	public void RemovePresentation\
	    Format ()
	    {
	    MutableSection s =  (Mutable\
	    Section)FirstSection;
	    s.RemoveProperty (PropertyIDMap.\
	    PID_PRESFORMAT);
	}
	```

TABLE 4.12. A Case of Code Translation from Java to C-sharp Output by C3P.

Code	`function inRange (value, min, max)` `{` `const int = parseInt (value, 10)` `return (` `` `${int}` === `${value.\ `` `` replace (/^0/, '')}` && `` `int >= min &&` `int <= max` `)` `}`
NL tokens	['Determine', 'if', 'value', 'is', 'within', 'a', 'numeric', 'range']

TABLE 4.13. A Case of Document Generation Output by C3P.

CHAPTER 5

Conclusions

5.1 Discussion

Beyond this work, there are some technologies that may be introduced in the future to explore related issues. One potential direction in contrastive learning for programming-natural language tasks is to explore multitask learning, where a model learns to simultaneously optimize multiple objectives such as code summarization, code search, and code translation. This approach could lead to more generalized representations and improve overall performance and transferability. Alongside this, integrating curriculum learning into contrastive pre-training could provide an ordered sequence of tasks for the model to learn, starting with simpler tasks and gradually progressing to more complex ones, enabling the model to build a stronger foundation in both programming and natural language understanding.

Moreover, incorporating meta-learning techniques could allow models to adapt more quickly to new programming languages or domains, while also facilitating more effective fine-tuning on smaller amounts of data. This adaptability could be further enhanced by developing contrastive learning techniques that enable zero-shot or few-shot learning, which would involve training a model capable of generalizing to new programming languages or tasks without requiring extensive labeled data.

Another intriguing avenue for future research involves incorporating knowledge graphs into contrastive learning models, as this would provide rich semantic information about programming languages, libraries, and frameworks, potentially improving the model's understanding of complex relationships between programming concepts and natural language. Furthermore, exploring self-supervised pre-training techniques involving code generation tasks could lead to a more comprehensive understanding of both programming and natural language syntax, as models would learn to generate syntactically and semantically correct code.

Finally, integrating active learning into the contrastive learning process could help models adapt more effectively to new programming languages or domains by selectively acquiring the most informative samples, reducing the amount of labeled data needed for fine-tuning. By combining these innovative ideas and paradigms, contrastive learning techniques for programming-natural language tasks could revolutionize the way we work with code and natural language understanding.

5.2 Conclusion

In this work, we present a novel contrastive pre-training approach designed to effectively capture and comprehend multi-modal representational features of both programming and natural language syntax. By doing so, we aim to establish a robust model capable of tackling various tasks across these domains. To ensure the efficacy and reliability of our methodology, we conduct a comprehensive series of experiments following the same pre-training and fine-tuning protocol as established benchmarks in the field.

The results of our experiments reveal that our proposed model consistently and significantly surpasses the performance of previous works in a wide range of code-related downstream tasks. These tasks include, but are not limited to, code search, clone detection, code translation, and document generation. Furthermore, our model demonstrates exceptional generalization capabilities and transferability across different programming languages. Notably, its performance on previously unseen programming languages during the training phase outperforms all supervised methods, showcasing its competitiveness with pre-training methods that have been trained on a diverse range of programming languages.

In order to provide a thorough understanding of the various factors that contribute to the success of our proposed method, we conduct an in-depth ablation study. This study examines the impact of different aspects of our methodology, including the fusion strategy employed for combining programming and natural language representations, the code representation techniques used, the token length of input data, and the batch size chosen for training. By systematically analyzing these components, we aim to offer valuable insights into the effectiveness of our contrastive pre-training approach and its potential applications in a variety of multi-modal tasks.

Bibliography

[1] X. Pei, D. Liu, L. Qian and C. Xu, 'Contrastive code-comment pre-training,' in *2022 IEEE International Conference on Data Mining (ICDM)*, IEEE, 2022, pp. 398–407.

[2] Z. Li *et al.*, 'Vuldeepecker: A deep learning-based system for vulnerability detection,' *arXiv preprint arXiv:1801.01681*, 2018.

[3] J. McBroom, B. Paassen, B. Jeffries, I. Koprinska and K. Yacef, 'Progress networks as a tool for analysing student programming difficulties,' in *Australasian Computing Education Conference*, 2021, pp. 158–167.

[4] T. H. Le, H. Chen and M. A. Babar, 'Deep learning for source code modeling and generation: Models, applications, and challenges,' *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–38, 2020.

[5] X. He, L. Xu, X. Zhang, R. Hao, Y. Feng and B. Xu, 'Pyart: Python api recommendation in real-time,' in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1634–1645.

[6] A. Vaswani *et al.*, 'Attention is all you need,' *Advances in neural information processing systems*, vol. 30, 2017.

[7] Y. Liu *et al.*, 'Roberta: A robustly optimized bert pretraining approach,' *arXiv preprint arXiv:1907.11692*, 2019.

[8] U. Alon, S. Brody, O. Levy and E. Yahav, 'Code2seq: Generating sequences from structured representations of code,' *arXiv preprint arXiv:1808.01400*, 2018.

[9] A. Kanade, P. Maniatis, G. Balakrishnan and K. Shi, 'Pre-trained contextual embedding of source code,' 2019.

[10] D. Guo *et al.*, 'Graphcodebert: Pre-training code representations with data flow,' *arXiv preprint arXiv:2009.08366*, 2020.

[11] M. Chen *et al.*, ‘Evaluating large language models trained on code,’ *arXiv preprint arXiv:2107.03374*, 2021.

[12] W. U. Ahmad, S. Chakraborty, B. Ray and K.-W. Chang, ‘Unified pre-training for program understanding and generation,’ *arXiv preprint arXiv:2103.06333*, 2021.

[13] Y. Wang, W. Wang, S. Joty and S. C. Hoi, ‘Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,’ *arXiv preprint arXiv:2109.00859*, 2021.

[14] Z. Feng *et al.*, ‘Codebert: A pre-trained model for programming and natural languages,’ *arXiv preprint arXiv:2002.08155*, 2020.

[15] A. Radford, K. Narasimhan, T. Salimans and I. Sutskever, ‘Improving language understanding by generative pre-training,’ *OpenAI Blog*, vol. 1, no. 8, 2018.

[16] J. Howard and S. Ruder, ‘Universal language model fine-tuning for text classification,’ in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 328–339.

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

[18] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy and S. R. Bowman, ‘Glue: A multi-task benchmark and analysis platform for natural language understanding,’ *arXiv preprint arXiv:1804.07461*, 2018.

[19] P. Rajpurkar, J. Zhang, K. Lopyrev and P. Liang, ‘Squad: 100,000+ questions for machine comprehension of text,’ in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016, pp. 2383–2392.

[20] S. Ruder, ‘A survey of cross-lingual word embedding models,’ *Journal of Artificial Intelligence Research*, vol. 65, pp. 569–630, 2019.

[21] T. Chen, S. Kornblith, M. Norouzi and G. Hinton, ‘Simclr: A simple framework for contrastive learning of visual representations,’ in *International Conference on Machine Learning*, PMLR, 2020, pp. 1597–1607.

[22] J. Lu, D. Batra, D. Parikh and S. Lee, 'Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks,' in *Advances in Neural Information Processing Systems*, 2019, pp. 13–23.

[23] R. Hadsell, S. Chopra and Y. LeCun, 'Dimensionality reduction by learning an invariant mapping,' *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2, pp. 1735–1742, 2006.

[24] A. v. d. Oord, Y. Li and O. Vinyals, 'Representation learning with contrastive predictive coding,' *arXiv preprint arXiv:1807.03748*, 2018.

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

[26] Z. Wu, Y. Xiong, S. X. Yu and D. Lin, 'Unsupervised feature learning via non-parametric instance discrimination,' *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3733–3742, 2018.

[27] M. Lewis *et al.*, 'Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,' *arXiv preprint arXiv:1910.13461*, 2020.

[28] K. He, H. Fan, Y. Wu, S. Xie and R. Girshick, 'Momentum contrast for unsupervised visual representation learning,' in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 9729–9738.

[29] T. Chen, S. Kornblith, K. Swersky, M. Norouzi and G. Hinton, 'Big self-supervised models are strong semi-supervised learners,' *arXiv preprint arXiv:2006.10029*, 2020.

[30] Y. Tian, D. Krishnan and P. Isola, 'What makes for good views for contrastive learning?' *arXiv preprint arXiv:2005.10243*, 2020.

[31] M. Caron, I. Misra, J. Mairal, P. Goyal, P. Bojanowski and A. Joulin, 'Unsupervised learning of visual features by contrasting cluster assignments,' *arXiv preprint arXiv:2006.09882*, 2020.

[32] M. Allamanis, E. T. Barr, C. Bird and C. Sutton, 'A survey of machine learning for big code and naturalness,' *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[33] P. Yin, B. Deng, E. Chen, B. Vasilescu and G. Neubig, 'Learning to mine aligned code and natural language pairs from stack overflow,' *arXiv preprint arXiv:1906.07155*, 2019.

[34] L. Hu, X. Lu and S. Zhang, 'Deep code search: Natural language processing powered software reuse,' *arXiv preprint arXiv:1804.00699*, 2018.

[35] A. T. Nguyen, T. T. Nguyen and T. N. Nguyen, 'Divide-and-conquer approach for multi-phase statistical migration for source code (t),' in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2015, pp. 585–596.

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

[37] N. Rethmeier and I. Augenstein, 'A primer on contrastive pretraining in language processing: Methods, lessons learned and perspectives,' *arXiv preprint arXiv:2102.12982*, 2021.

[38] M. V. Conde and K. Turgutlu, 'Clip-art: Contrastive pre-training for fine-grained art classification,' in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 3956–3960.

[39] A. Radford *et al.*, 'Learning transferable visual models from natural language supervision,' in *International Conference on Machine Learning*, PMLR, 2021, pp. 8748–8763.

[40] A. Krizhevsky, I. Sutskever and G. E. Hinton, 'Unsupervised feature learning via non-parametric instance discrimination,' in *Proceedings of the 2011 IEEE International Conference on Computer Vision*, IEEE, 2011, pp. 481–488.

[41] C. K. Joshi, D. Chen, A. Shah, Z. Parekh and M. Bendersky, 'Marge: Pre-training with contrastive language and image encodings,' *arXiv preprint arXiv:2104.07033*, 2021.

[42] I. Beltagy, M. E. Peters and A. Cohan, 'The power of scale for parameter-efficient prompt tuning,' *arXiv preprint arXiv:2104.08691*, 2021.

[43] X. Chen, L. Gong, A. Cheung and D. Song, 'Plotcoder: Hierarchical decoding for synthesizing visualization code in programmatic context,' in *Proceedings of the 59th*

Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 2021, pp. 2169–2181.

[44] N. Rethmeier and I. Augenstein, *Data-efficient pretraining via contrastive self-supervision*, 2021. arXiv: 2010.01061 [cs.CL].

[45] T. Klein and M. Nabi, 'Contrastive self-supervised learning for commonsense reasoning,' in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 7517–7523.

[46] X. Duan, H. Yu, M. Yin, M. Zhang, W. Luo and Y. Zhang, 'Contrastive attention mechanism for abstractive sentence summarization,' in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 3044–3053.

[47] Y. Qu, D. Shen, Y. Shen, S. Sajeev, J. Han and W. Chen, 'Coda: Contrast-enhanced and diversity-promoting data augmentation for natural language understanding,' *arXiv preprint arXiv:2010.08670*, 2020.

[48] D. Iter, K. Guu, L. Lansing and D. Jurafsky, 'Pretraining with contrastive sentence objectives improves discourse performance of language models,' in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4859–4870.

[49] A. Neelakantan *et al.*, *Text and code embeddings by contrastive pre-training*, 2022. arXiv: 2201.10005 [cs.CL].

[50] X. Ge, Y. Yu and L. Song, 'Codehow: Code search across multiple repositories with natural language queries,' in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 1018–1029.

[51] K. Wang, D. Liu and X. Huang, 'Codebert: A pre-trained model for programming and natural language processing,' in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1059–1069.

[52] G. Singh, R. Kumar, S. Kumar and S. Sanyal, 'Graphcodesearch: Semantic code search using graphs,' *arXiv preprint arXiv:2006.15210*, 2020.

[53] H. Husain, K. Wu, T. Gazit, M. Allamanis and M. Brockschmidt, 'Codesearchnet challenge: Evaluating the state of semantic code search,' *arXiv preprint arXiv:2009.09464*, 2020.

[54] M. Allamanis, H. Peng and C. Sutton, 'A neural framework for code search with natural language queries,' in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, IEEE, 2018, pp. 610–620.

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

[56] M. Li and Y. Li, 'A survey on code clone detection research,' in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2018, pp. 1–12.

[57] L. Jiang, X. Zhang and Z. Su, 'Scalable code clone detection: Practices and trends,' *Journal of Systems and Software*, vol. 131, pp. 408–429, 2017.

[58] M. White, M. Tufano, C. Vendome and D. Poshyvanyk, 'Deep learning code fragments for code clone detection,' in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2016, pp. 87–98.

[59] S. Zhou, Y. Chen, Y. Liu, Q. Hu and S. Wang, 'Self-supervised learning for code clone detection,' in *Proceedings of the 43rd International Conference on Software Engineering*, 2021, pp. 1255–1266.

[60] B. Roziere, M.-A. Lachaux, L. Chanussot and G. Lample, 'Unsupervised translation of programming languages,' *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 601–20 611, 2020.

[61] H. Ragavan, V. Srikumar and D. Roth, 'Statistical machine translation for code migration,' in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 2049–2059.

[62] S. Srivastava, M. Malik and C. Jawahar, 'Toward statistical machine translation for programming languages,' *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 36, no. 2, pp. 1–28, 2014.

[63] J. Gu, Z. Lu, H. Li, V. O. Li and X. Xie, ‘Deep api learning,’ in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 2016, pp. 631–647.

[64] K. Xu, K. Liu, Z. Chen and Y. Zhao, ‘Neural machine translation for code comments: Learning from large-scale parallel corpus,’ *Journal of Systems and Software*, vol. 162, p. 110 448, 2020.

[65] Y. Zhang, J. Xu, Z. Wang, Y. Zhang and Y. Lu, ‘Joint template-based generation for accurate and diverse chinese table layout description,’ in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 7251–7258.

[66] K. Wang, C. Yao and H. Wang, ‘Code2seq: Generating sequences from structured representations of code,’ in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 3611–3621.

[67] R. Lebret, D. Grangier and M. Auli, ‘Neural text generation from structured data with application to the biography domain,’ in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016, pp. 1203–1213.

[68] Z. Zhang, Y. Zhao, X. Zhang and J. Su, ‘Toward diverse and coherent paragraph generation from a given topic,’ *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 2, pp. 795–807, 2021.

[69] Z. Li, X. Wang, H. Zhang and T. Zhao, ‘Unsupervised neural document generation with domain-specific knowledge,’ *ACM Transactions on Information Systems (TOIS)*, vol. 38, no. 1, pp. 1–28, 2020.

[70] F. Yu *et al.*, ‘Ernie-vil: Knowledge enhanced vision-language representations through scene graph,’ *arXiv preprint arXiv:2006.16934*, 2020.

[71] I. Loshchilov and F. Hutter, ‘Decoupled weight decay regularization,’ *arXiv preprint arXiv:1711.05101*, 2017.

[72] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis and M. Brockschmidt, ‘Codesearchnet challenge: Evaluating the state of semantic code search,’ *arXiv preprint arXiv:1909.09436*, 2019.

[73] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy and M. M. Mia, 'Towards a big data curated benchmark of inter-project code clones,' in *2014 IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2014, pp. 476–480.

[74] L. Jiang, G. Misherghi, Z. Su and S. Glondu, 'Deckard: Scalable and accurate tree-based detection of code clones,' in *29th International Conference on Software Engineering (ICSE'07)*, IEEE, 2007, pp. 96–105.

[75] M. White, M. Tufano, C. Vendome and D. Poshyvanyk, 'Deep learning code fragments for code clone detection,' in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2016, pp. 87–98.

[76] H. Wei and M. Li, 'Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code.,' in *IJCAI*, 2017, pp. 3034–3040.

[77] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang and X. Liu, 'A novel neural source code representation based on abstract syntax tree,' in *ICSE*, 2019, pp. 783–794.

[78] S. Hochreiter and J. Schmidhuber, 'Long short-term memory,' *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[79] W. Wang, G. Li, B. Ma, X. Xia and Z. Jin, 'Detecting code clones with graph neural network and flow-augmented abstract syntax tree,' in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2020, pp. 261–271.

[80] X.-P. Nguyen, S. Joty, S. C. Hoi and R. Socher, 'Tree-structured attention with hierarchical accumulation,' *arXiv preprint arXiv:2002.08046*, 2020.

[81] I. Sutskever, O. Vinyals and Q. V. Le, 'Sequence to sequence learning with neural networks,' *Advances in neural information processing systems*, vol. 27, 2014.

5.3 Appendix A

What the contrastive pre-training learned in the process.

Contrastive pre-training has emerged as a powerful approach to learn high-quality features and representations from data by exploiting the intrinsic differences and similarities among samples. In this paragraph, we delve into the pre-training process, examining the learning dynamics from various perspectives, including mathematical formula analysis, sample extraction, and objective functions.

Mathematically, contrastive pre-training employs a loss function that aims to maximize the similarity between semantically related samples, while minimizing the similarity between unrelated samples. This is typically achieved by computing the cosine similarity between the learned embeddings of the samples in the latent space. The objective function is carefully designed to ensure that the model captures the underlying relationships among the samples and adequately adapts to the given task. In the symmetric contrastive learning process, the goal is to learn a representation that maximizes the similarity between related samples (positive pairs) while minimizing the similarity between unrelated samples (negative pairs). The symmetric loss function used for this purpose can be represented using mathematical formulas. Let's consider a pair of related samples x_i and x_j, where x_i is the code and x_j is the corresponding comment. Their embeddings in the latent space are represented by $v_i = f(x_i)$ and $v_j = g(x_j)$, where f and g are the code and comment encoders, respectively.

The symmetric contrastive loss function can be expressed as:

$$L(x_i, x_j) = -\log \frac{\exp(\text{sim}(v_i, v_j)/\tau)}{\sum_{k=1}^{K} \exp(\text{sim}(v_i, v_{j_k})/\tau)}$$

Here, $\text{sim}(v_i, v_j)$ denotes the similarity between the embeddings v_i and v_j, which is typically calculated using the cosine similarity. τ is a temperature parameter that controls the concentration of the probability distribution, and K is the total number of negative pairs. The term v_{j_k} represents the k-th negative sample's embedding for the code sample x_i.

This loss function encourages the model to maximize the similarity between related samples (i.e., code and corresponding comment) and minimize the similarity between unrelated samples (i.e., code and other comments). By optimizing this objective, the model learns a representation in the latent space that is effective in capturing the intrinsic connections between code and comments, which subsequently leads to improved performance on downstream tasks.

In essence, the symmetric contrastive learning process, as represented by the mathematical formula above, allows the model to learn meaningful relationships between related samples by maximizing their similarity while minimizing the similarity between unrelated samples. This is achieved by optimizing the symmetric loss function, which ensures that the learned embeddings in the latent space capture the essential information required for successful transfer learning and improved performance on various downstream tasks.

From a sample extraction perspective, the pre-training process involves carefully selecting and preparing data that captures the essential characteristics of both related and unrelated samples. For instance, when working with code-comment pairs, the model should be trained on a diverse set of examples that represent various programming languages, coding styles, and natural language descriptions. This allows the model to learn a comprehensive set of relationships that can be effectively applied to the downstream tasks.

Regarding objective functions, contrastive pre-training seeks to optimize a symmetric loss function that balances the need to maximize the similarity of related pairs while minimizing the similarity of unrelated pairs. This is often achieved by employing a temperature-scaled cross-entropy loss that encourages the model to focus on the most informative and discriminative features. As a result, the learned embeddings in the latent space exhibit rich semantic information, which is crucial for successful transfer to downstream tasks.

In summary, contrastive pre-training learns valuable features and representations by carefully balancing mathematical formulas, sample extraction, and objective functions. This approach enables the model to effectively capture the underlying relationships among the samples, which is essential for successful transfer learning and ultimately leads to improved performance on various downstream tasks.