Distantly Supervised Contrastive Learning for Low-Resource Scripting Language Summarization

Junzhe Liang, Haifeng Sun, Zirui Zhuang,Qi Qi,Jingyu Wang,Jianxin Liao
Beijing University of Posts and Telecommunications
{ljz,hfsun,zhuangzirui,qiqi8266,wangjingyu}@bupt.edu.cn
jxlbupt@gmail.com

Abstract

Code summarization provides a natural language description for a given piece of code. In this work, we focus on scripting code—programming languages that interact with specific devices through commands. The low-resource nature of scripting languages makes traditional code summarization methods challenging to apply. To address this, we introduce a novel framework: distantly supervised contrastive learning for low-resource scripting language summarization. This framework leverages limited atomic commands and category constraints to enhance code representations. Extensive experiments demonstrate our method's superiority over competitive baselines.

Keywords: code summarization, scripting languages, supervised contrastive learning

1. Introduction

Source code summarization, which generates a readable summary describing a program's functionality, has garnered significant attention in recent years (Zhu and Pan, 2019; Shi et al., 2021). This can help software developers reduce the time needed for software development and maintenance (Xia et al., 2017; Chen et al., 2022). In this study, we mainly focus on scripting languages, which provide a high level of abstraction of system functionalities and implement a direct, sequential command line execution. Common scripting languages include shell, bash, configuration languages, and so on. Figure 1 is an example of a network configuration language, primarily used for configuring and managing network devices such as routers and switches.

Most previous code summarization efforts have primarily focused on traditional programming languages such as Python or Java (McBurney and McMillan, 2015; Wei et al., 2019). Scripting languages, similar to them, are used to define the flow of computer operations and adhere to a set of standard syntactical norms. However, scripting languages still have their distinct features. Firstly, it has command templates. A command template is essentially a predefined sequence or pattern of commands that serve a particular purpose or perform a specific function. Command templates encapsulate complex tasks into simpler, more manageable, and reusable components. For example, 'ls [-R|-c|-d|-a] [file|dir]' is a template in shell, which can parse commands like 'ls -d', 'ls -c -a', 'ls dir', etc., and they serve the function of listing files in a directory. These parsed commands can naturally be categorized into one group, which we refer to as the command category (category constraints) in this paper. The second characteristic of script-

Source Code

```
switch  # interface loopback 2
switch  # ip address 192.168.0.200 255.255.255
switch  # router bgp 64496
switch  # neighbor 10.0.0.100 remote-as 64497
switch  # update-source loopback 1
switch  # disable-connected-check
```

Summarization

This example shows how to source BGP TCP connections for the specified neighbor with the IP address of the loopback interface rather than the best local address.

Figure 1: instances of config code and its summaries

ing languages is that the command set is finite. For any given system or device, there is a limited set of command templates. By fully enumerating these templates, we can parse and generate every valid command for that system. In this paper, we refer to these commands as 'atomic commands'. (To keep the command count manageable, we replace actual parameters with their respective parameter names.) This is a powerful concept that allows us to make precise and comprehensive analysis of the command space. Furthermore, scripting languages tend to be low-resource, since it acts directly on the device, and the set of commands executed by devices varies from one manufacturer to another, so we can't collect a large-scale parallel corpus for each device to train models on.

Most traditional code summarization techniques, such as Ahmad et al. (2020), Feng et al. (2020), rely heavily on extensive annotated corpora for training. However, the lack of training data makes it hard to apply traditional summarization schemes. To apply code summarization to scripting languages, it's crucial to fully leverage the limited atomic commands and category constraints available. In traditional programming languages, there are no command

LREC-COLING 2024, pages 5006–5017
20-25 May, 2024. © 2024 ELRA Language Resource Association: CC BY-NC 4.0

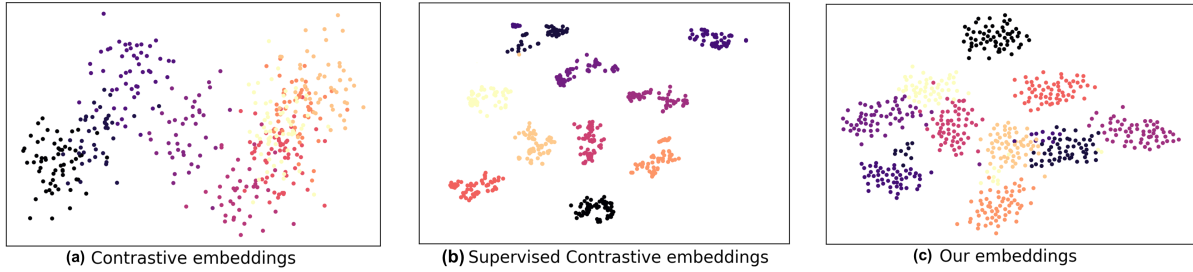


Figure 2: A t-SNE Visualization of command embeddings from 10 random categories. The same color means from the same category. Using contrastive learning alone cannot make representations of the same category spatially adjacent, as shown in Figure(a).

templates. Every symbol, variable, and operator must be precisely placed in the correct position (Xie et al., 2021). As a result, we need to model them at the token level, typically represented by an AST (Hu et al., 2018). For scripting languages, the presence of atomic commands allows us to model them at the sentence level. Furthermore, the limited number of commands in scripting languages facilitates the learning of individual command representations through pre-training. This alleviates challenges posed by the lack of a parallel corpus, which might otherwise hinder effective source code representation. Additionally, the categories of scripting languages offer an alternative perspective on commands, i.e., abstracting a single command into the category it belongs to. This reduces the variance between data and can be viewed as a means of data augmentation when the data set is small.

In order to exploit the aforementioned two properties, this paper proposes a distantly supervised contrastive learning approach. Contrastive learning is to map the representations to the unit hypersphere (Wang and Isola, 2020), which can be conveniently trained in sentence units (Gao et al., 2021) and does not require the same attention to token level details as BERT (Devlin et al., 2019), RoBERTa (Liu et al., 2019b). Furthermore, the differences emphasized by contrastive learning are more easily captured in a limited command space. To harness the benefits of category constraints, we employ supervised contrastive learning (Khosla et al., 2020). This approach aims to bring command representations from the same category closer together on the unit hypersphere. However, this method can diminish feature quality. Specifically, supervised contrastive learning struggles to differentiate between various positive examples, as evidenced by the vector overlap in Figure 2(b). To prevent excessive information loss, we adopted a balanced strategy by implementing the Maximizes the Minimum Angle (MMA) technique (Wang et al., 2020). This approach ensures that command representations are situated adjacently in the hypersphere space, yet distinct enough to be discernible, as illustrated in Figure 2(c). The term 'distantly' in our method signifies that we consider the category as a form of 'weak label' for the command. This perspective enables us to view scripting language either as an aggregation of specific commands or a collective of categories.

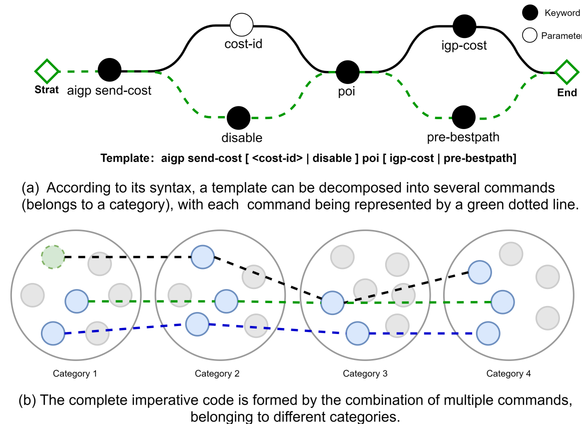
To assess our method's effectiveness, we benchmarked it against four baseline models using three evaluation metrics. The results showed our model's superior performance. The main contributions of our research are as follows:

- We introduce a distantly supervised contrastive learning approach, enabling effective summarization of scripting language in low-resource scenarios.
- Our method distinguishes inter-class features while ensuring clear differentiation within intra-class features.
- Through comprehensive experiments, we show that our approach consistently outperforms traditional baseline models.

2. Preliminaries

2.1. Scripting Language Summarization

Scripting languages are typically designed to simplify routine programming tasks, especially for automation or integrating various software components. While languages like Python and JavaScript were initially used to some extent as scripting languages, they have evolved into more complex, full-fledged programming languages. Freed from the constraints of command templates, they are now more suitable for large-scale application development, and thus, fall outside the scope of this study. In contrast, network configuration languages (or config code) are a subset of scripting languages, which are notably characterized by being constrained by command templates. These config code are specifically crafted for configuring



(a) According to its syntax, a template can be decomposed into several commands (belongs to a category), with each command being represented by a green dotted line.

(b) The complete imperative code is formed by the combination of multiple commands, belonging to different categories.

Figure 3: Command template

and managing network devices. With the rapid expansion of network scales, providing accurate and efficient summarization for network configuration languages has become an urgent task, which is the primary focus of this paper. Figure 3.a illustrates the parsing structure of a command template in network configuration language (for specific syntax, please refer to Appendix). A complete code sequence is composed of sub-commands from various templates.

2.2. Problem Statement

Traditional methods for code summarization rely on end-to-end training with vast parallel corpora. However, the low-resource nature of scripting languages hampers the efficacy of such end-to-end approaches in learning adequate source code representations. To address this, we exploit the atomic command and category constraints. Our approach begins by pre-training to learn individual command representations. Subsequently, we fine-tune using a smaller parallel corpus. Experimental results attest to the superiority of our method.

3. Related Work

3.1. Code Summarization

Many code summarization methods employ the encoder-decoder architecture, as evidenced by Liang and Zhu (2018) and Lin et al. (2018). Taking a novel approach, Hu et al. (2018) was the first to represent code as an Abstract Syntax Tree (AST) and utilized random path selections from the tree as network input. Building upon this, Ahmad et al. (2020) leveraged the Transformer architecture, addressing issues related to long-range dependencies and the omission of code structure information. Wei et al. (2019) posited code summarization and code generation as dual tasks, arguing that model parameters for generation and summarization ought to be akin since both encapsulate the mapping relationship between distinct domains. Lastly, CodeBERT (Feng et al., 2020) harnesses massively parallel datasets for training and functions as a feature extractor in downstream tasks.

3.2. Contrastive Learning on the Unit Hypersphere

The unit hypersphere feature space offers favorable properties. It's been observed that contrastive learning produces commendable results when features are projected onto this space (Wang et al., 2017). Diving into the underlying rationale, Bachman et al. (2019) presents an explanation rooted in the InfoMax principle (Linsker, 1988). They emphasize the significance of maximizing mutual information from varied perspectives within the hypersphere space. Intuitively, achieving maximum information entropy happens when features are evenly spread across the hypersphere, resulting in more representative learned features. Supporting this notion, Bojanowski and Joulin (2017) managed to derive quality representations by directly mapping uniformly sampled points onto the unit hypersphere.

The challenge lies in ensuring vectors are uniformly distributed on the hypersphere. While this is an ideal scenario, in practice, efforts often revolve around decreasing vector correlations. For instance, Liu et al. (2018) achieved this by minimizing potential energy, while Wang et al. (2020) opted to maximize minimum angles, thereby cutting down on neural weight connections.

Most preceding works have aimed to evenly distribute inter-class vectors on the hypersphere but tend to overlook the distinction of intra-class vectors. Contrarily, our proposed method not only differentiates between inter-class features but also enables a pronounced distinction of intra-class features, proving advantageous for subsequent generation tasks.

4. Methodology

4.1. Model Architecture

In the pre-training stage, our model is composed of two main components: an encoder network, denoted as $Enc(\cdot)$, and a projection network, $Proj(\cdot)$, as described in Chen et al. (2020). The encoder utilizes the BERT structure in conjunction with average pooling to map the input into a representation vector. The projection network then processes this encoder output for further dimensionality reduction. Specifically, we employ a multilayer perceptron with a 768-dimensional input that produces a 128-dimensional output. It's crucial to note that this projection is active only during training and is omitted in the inference phase.

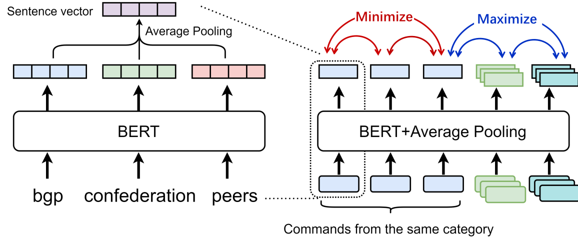


Figure 4: Pre-training framework, we pre-train an encoder by minimizing the distance between projected codes from the same category and maximizing those from different categories.

For differentiation of inter-category vectors, we integrate supervised contrastive learning (Khosla et al., 2020). In contrast, the MMA method (Wang et al., 2020) is used for intra-category vector differentiation. Both techniques are applied to our model based on specific learning stages. For the downstream code summarization task, we enhance each command with positional encoding, which reflects their execution sequence in the source code.

4.2. Supervised Contrastive Loss

Consider a random batch of data. Let's denote its index by $i \in I \equiv \{1 \ldots P \ldots N\}$. In this notation, $i \in \{1 \ldots P\}$ signifies the index of all samples that belong to the same category. The supervised loss function can be represented as:

$$\mathcal{L}_c = \frac{-1}{|P(i)|} \sum_{p \in P(i)} \log \frac{\exp\left(sim(\boldsymbol{z}_i, \boldsymbol{z}_p)/\tau\right)}{\sum_{a \in A(i)} \exp\left(sim(\boldsymbol{z}_i, \boldsymbol{z}_a)/\tau\right)}$$

Here, $A(i) \equiv I \backslash \{i\}$, index i represents the anchor, $P(i) \equiv \{p \in A(i) : y_p = y_i\}$ represents the positive cases, the remaining $N - P$ samples represent the negative cases. $|P(i)|$ is the number of all positive samples in a batch, τ is a scalar temperature parameter. Where $z_l = Proj(Enc(x_l))$, sim is the dot product operation of two normalized vectors, i.e. $\operatorname{sim}(z_i, z_j) = z_i^T z_j / \|z_i\|_2 \|z_j\|_2$.

4.3. Maximizes the Minimum Angle Loss

In our effort to enhance the distinction of intra-class features, we draw inspiration from the "Thomson problem" (Thomson, 1904). This problem seeks to determine the optimal arrangement of n points on a unit sphere such that the minimum distance between any two points is maximized. There exist both numerical and analytical solutions to this problem. However, the analytical solution is only applicable for specific combinations of n (number of vectors) and d (vector dimension). While the analytical solution may not be suitable for our context, the advancements in optimizers and automatic derivative libraries (Paszke et al., 2017) make it feasible to derive approximate solutions through numerical approaches. Thus, we primarily employ a numerical solution, which can be expressed as:

$$\mathcal{L}_{MMA} = -\frac{1}{n} \sum_{i=1}^{n} \min_{j, j \neq i} \boldsymbol{\theta}_{ij}, \quad \boldsymbol{\theta} = \arccos\left(\hat{\boldsymbol{Z}} \hat{\boldsymbol{Z}}^T\right)$$

In this formulation, $\hat{\boldsymbol{Z}} \in R^{n \times d}$ denotes a set of vectors distributed on the unit hypersphere. Meanwhile, $\boldsymbol{\theta} \in R^{n \times n}$ represents the pairwise angle matrix. It's noteworthy that optimizing solely based on the global minimum angle in each batch iteration can be inefficient. Therefore, we consider the average of the minimal angle for each vector. Here, n indicates the total count of positive samples in a batch.The numerical approach, focusing exclusively on the optimization of within-class features, proves more effective and robust, especially when contrasted with loss functions of the form cos (Wang et al., 2020), which it surpasses in terms of faster convergence, particularly evident when vectors are in close proximity.

4.4. Pre-training Overall Objective

A basic approach to incorporate both loss functions is illustrated in equation:

$$\mathcal{L}' = \mathcal{L}_C + \lambda \cdot \mathcal{L}_{MMA}$$

However, in practical settings, $\mathcal{L}_C$ and $\mathcal{L}_{MMA}$, having distinct optimization objectives, often counterbalance each other. As a result, achieving simultaneous optimization for both is challenging. We anticipate that these functions might operate more efficiently if allowed to act independently during different training periods. While manually balancing the two can be cumbersome, an automated learning approach, inspired by dynamic weight average (Liu et al., 2019a), becomes more appealing. Instead of resorting to time-intensive access to the network's internal gradients, we assess different training phases by tracking the rate of loss changes. Here, λ is designated for $\mathcal{L}_{MMA}$, and $1 - \lambda$ is for $\mathcal{L}_c$. Our primary optimization goal is then described by equation:

$$\mathcal{L} = (1 - \lambda_t) \cdot \mathcal{L}_c + \lambda_t \cdot \beta \cdot \mathcal{L}_{MMA}$$
$$\lambda_t = \tanh\left(\frac{\mathcal{L}_c(t-1)}{\mathcal{L}_c(t-2)}\right)$$

In this context, λ_t gauges the decreasing rate of $\mathcal{L}_c$ and lies within the range of (0,1). It steers the significance between the two optimization objectives. A value of λ_t close to 1 (which tends to stabilize around 0.75 when $\mathcal{L}_c$ levels off) implies $\mathcal{L}_c$ nearing convergence, prioritizing optimization towards $\mathcal{L}_{MMA}$. Moreover, β remains the sole hyperparameter and dictates the dominance between the two, independent of the training phase.

In our methodology, when t is 1 or 2, we initialize λ_t to 0. The loss values for $\mathcal{L}_c$ and $\mathcal{L}_{MMA}$ are calculated as an average over multiple iterations within each epoch, mitigating potential variances from stochastic data choices and gradient descent nuances.

4.5. Transfer Learning

Our model is built upon the standard encode-decode architecture. In the encoding phase, after the pre-training, we transfer the encoder with its retained pre-trained weights to the downstream task. This encoder processes a configuration language code, consisting of n commands $s_1, s_2, \ldots, s_n$, and outputs an $n \times d$ feature matrix. Given the parallel processing of input features and the absence of inherent positional information, we incorporate Sinusoidal encoding (Vaswani et al., 2017) to capture each command's sequential position within the code fragment.

$$\begin{cases} \boldsymbol{p}_{k,2i} = \sin\left(k/10000^{2i/d}\right) \\ \boldsymbol{p}_{k,2i+1} = \cos\left(k/10000^{2i/d}\right) \end{cases}$$

Specifically, $\boldsymbol{p}_{k,2i}$,$\boldsymbol{p}_{k,2i+1}$ denote the $2i$-th and $2i+1$-th components of the k-th command's encoding vector. After calculating this positional encoding, it merges with the original feature matrix to yield the final output features. For the decoding phase in the summarization task, we employ a transformer decoder, which is trained from scratch, to bridge the gap between the target space and the encoder's output. The model is then trained on a parallel dataset using a cross-entropy loss function.

5. Experiment

5.1. Datasets

In the realm of configuration languages, Huawei and Cisco stand out as the largest network equipment providers. We sourced configuration commands for network equipment by crawling openly accessible data from the companies' websites[1]. The dataset encompasses both command templates and pairs of commands with their natural language descriptions, denoted as <code, nl> pairs. Specifically, the Cisco dataset encompasses 325 templates, totaling around 12,000 commands, which encapsulates all necessary commands for device configuration. Within this dataset, there are 21,462 <code, nl> pairs. On the other hand, the Huawei dataset boasts 364 templates, approximately 14,000 commands, and 26,400 <code, nl> pairs. We've partitioned the final dataset into 80% for training, 10% for validation, and 10% for testing.

[1] https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus5500/sw/command/reference/unicast/n5500-ucast-cr/n5500-bgp_cmds_n.html

Data Parsing On public websites, we can access templates for each configuration command, such as "router bgp <as-number>", the segment within angle brackets represents the command parameter. In our context, the parameter typically identifies the network device and has minimal impact on code summarization performance. To mitigate the influence of these parameters, we identify their positions within the command line and substitute them with the corresponding parameter names. Thus, "route bgp 100" becomes "route bgp as-number". This substitution occurs during both the pre-training and fine-tuning stages to maintain consistency and enhance the generalizability of the pre-trained feature representation.

5.2. Baseline Methods

We compare our approach with four baseline models: the Pointer Network, highlights the utilization of a shared vocabulary between the source and target domains. The remaining models are focused on traditional methodologies within the spheres of text and code summarization.

Pointer Network (Vinyals et al., 2015): A pointer structure is added to the seq2seq model to determine whether the current prediction is copied straight from the source text or generated from the vocabulary, addressing summary inaccuracies. This seq2seq model is a Transformer-based architecture, pre-trained on our <code, nl> corpus.

BART (Lewis et al., 2019): Using the standard transformer structure, it is possible to fine-tune the text summary task directly on the <code,nl> corpus. In our work, we treat the source code as plain text, using separators to denote distinct command lines.

Dual Model (Wei et al., 2019): Treating code summarizing and code generation as a dual task, with the connection between the two tasks as a training constraint, it increases the performance of both the code summary task and the generation task.

CodeBERT (Feng et al., 2020): A model capable of managing bimodal data. In the pre-training phase, replacement token detection (RTD) and masked language modeling (MLM) techniques are applied to unimodal command input. During the fine-tuning phase, the bimodal paired data is utilized. The downstream decoder is consistent with our model and is trained on the <code,nl> corpus.

Approach	Cisco						Huawei					
	B-1	B-2	B-3	B-4	METEOR	ROUGE-L	B-1	B-2	B-3	B-4	METEOR	ROUGE-L
Pointer Network	31.71	18.62	10.34	7.12	15.64	45.28	32.85	20.11	11.62	8.24	14.37	47.39
BART	35.76	21.32	13.94	11.62	18.26	47.19	37.74	22.96	15.14	11.93	20.61	48.27
Dual Model	42.36	27.41	21.54	15.33	21.42	51.27	41.26	28.91	22.63	16.27	22.51	50.65
CodeBERT	51.78	30.61	23.87	17.21	24.82	52.63	52.61	32.17	24.66	18.92	25.74	53.82
Our Model	**58.23**	**36.79**	**27.14**	**20.32**	**26.72**	**54.93**	**57.21**	**37.83**	**29.67**	**22.34**	**27.98**	**55.71**

Table 1: BLEU 1-4, METEOR and ROUGE-L comparision of our model with other counterparts. The best results are in bold font.

Approach	Cisco						Huawei					
	B-1	B-2	B-3	B-4	METEOR	ROUGE-L	B-1	B-2	B-3	B-4	METEOR	ROUGE-L
w/o Category Constraint	49.81	31.27	23.64	16.31	24.23	51.07	50.41	32.47	23.82	17.53	24.96	50.44
w/o MMA	50.67	29.87	22.93	17.61	24.97	52.76	51.84	30.65	22.04	18.15	25.62	53.81
w/o Relative Positon	56.41	35.92	26.87	19.31	26.31	54.37	57.52	35.43	27.16	21.40	27.55	54.83
Our Model	**58.23**	**36.79**	**27.14**	**20.32**	**26.72**	**54.93**	**57.21**	**37.83**	**29.67**	**22.34**	**27.98**	**55.71**

Table 2: The effects of different components.

5.3. Evaluation Metrics

Accuracy metrics We evaluate the config code summarization performance using the following metrics, BLEU-1, BLEU-2, BLEU-3, BLEU-4 (Papineni et al., 2002), ROUGE-L[2] (Lin, 2004) and METEOR (Banerjee and Lavie, 2005). The basic idea of BLEU and ROUGE is calculate the proportion of repeated words between the reference and generated sentences, where the BLEU-1 measures word-level accuracy and the higher-order BLEU measures sentence fluency. METEOR[3] introduced synonym matching using an additional knowledge source (such as WordNet).

Uniformity metric To further validate our approach, we employed the quantifiable metrics from Wang and Isola (2020) to assess the uniformity of vector distribution in space. As illustrated below, this metric adheres to the Gaussian potential kernel, yielding values between 0 and 1 that diminish with increasing distance.

$$G(u, v) \triangleq e^{\|u-v\|_2^2} = e^{2(u^\top v-1)}$$

We use the expectation $\mathbb{E}\left[G_t(u, v)\right]$ to examine the overall data distribution characteristics. It's important to note that this metric is minimized only when the distribution is uniform. When using the Eulerian distance, it can achieve its lowest value for any distribution with a zero mean. For a comprehensive discussion, please see Wang and Isola (2020) and Borodachov et al. (2019).

[2] https://github.com/google-research/google-research/tree/master/rouge

[3] https://www.cs.cmu.edu/~alavie/METEOR/README.html

5.4. Experimental Settings

We utilize an embedding and hidden vector dimension of 512, with both the Encoder and Decoder consisting of 6 layers. Multi-head attention is employed with 8 heads. For training, we use 5 positive samples per batch and a batch size of 2048. Our temperature coefficient is set to 0.1, which determines how much attention the contrast loss pays to difficult negative samples, and a hyperparameter β set at 0.3. During inference, a beam size of 4 is applied. The Adam optimizer (Kingma and Ba, 2014) is employed throughout the training. Detailed hyperparameter configurations are elaborated in Section 8.

6. Results and Discussion

6.1. Code Summarization Results

Table 1 presents the performance of our proposed method alongside baseline methodologies. Text summarization models like Pointer Network and BART offer notable improvements over preceding methods, they still lag our technique by a 7.7% margin in ROUGE-L. This gap underscores that despite some similarities between config code and natural language, treating it purely as a text summarization task may not be adequate. Both CodeBERT and our method, benefitting from training on a bimodal corpus (command templates and <*code,nl*> paired dataset), excel in capturing inter-code semantic nuances. However, our model's capacity to integrate both high-level category and intricate low-level details accounts for its superior performance across metrics.

	Supervised Contrastive	Contrastive Learning	Our Model
Uniformity Score	0.64	0.27	0.31

Table 3: Measurement of distribution uniformity among different models

6.2. Ablation Study

We analyzed the impact of different model components as outlined in Table 2. Our observations suggest that while most components play a pivotal role in model performance, the omission of some might not significantly degrade the performance metrics. To elaborate: (1) Category Constraints: When removed, the model essentially reverts to a basic contrastive learning framework. In this context, each dataset sample and its augmentation (achieved via methods such as dropout (Gao et al., 2021), commonly used in contrastive learning) serve as mutual positive examples. This change results in a decline of the BLEU-1 score to 49.81 and the METEOR to 24.23, underscoring the importance of the category constraints in our design. (2) MMA Component: Without MMA, our model mirrors supervised contrastive learning. Consequently, representations of samples from the same category lose their distinctiveness. This is evident from the METEOR score, which drops to 24.97. This highlights the significance of discriminative representation in enhancing model performance. (3) Position Embedding: Its omission led to a slight reduction in the METEOR score to 26.31. The impact on performance is relatively minimal. A plausible rationale is that while certain functions demand specific commands, altering their sequence might make them unable to work on actual devices, but their intended functionality remains unchanged.

In Table 3, we present the uniformity metric results for the three discussed models. This metric evaluates how uniformly the learned feature vectors are distributed over the unit hypersphere. A smaller value denotes a more uniform distribution. From the results, our model ranks second, following contrastive learning, showcasing our method as an intermediate solution between the two.

6.3. Intra-Class Distance vs. Inter-Class Distance

To validate our method further, we examined the intra-class and inter-class distances of command representations in the unit hypersphere across different training phases, as depicted in Figure 5. For representations of the same category, one was randomly chosen as the anchor. The intra-class distance represents the average distance between instances of the same category and the anchor. In contrast, the inter-class distance measures the distance between the anchor and its closest neighboring class.

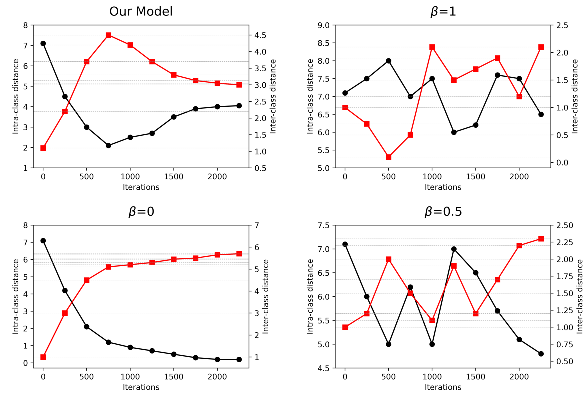


Figure 5: intra-class(black) and inter-class(red) distances throughout training periods

To evaluate the effectiveness of our proposed loss form $\mathcal{L}$, we compared it against the model's performance using $\mathcal{L}'$ as the loss. Tests were conducted with β values of 1, 0.5, and 0 for $\mathcal{L}'$. A beta value of 0 implies the model performs only classification. For $\mathcal{L}$, the optimal beta value was chosen based on performance. Key observations: (1) Using $\mathcal{L}$ as the loss, the inter-class distance initially increases substantially. This suggests that the model initially clusters representations of the same class in the neighboring feature space, and later distinguishes at the instance level, causing the inter-class distance to reduce and the intra-class distance to increase. (2) For $\mathcal{L}'$, when β is 0, the inter-class distance diminishes while the intra-class distance grows, indicating the model's sole focus on classification. With β values of 1 or 0.5, the model struggles to converge.

7. Human Evaluation

We perform a human evaluation to ensure that our increase in scores is also followed by an increase in human readability and quality. In particular, we want to know whether the category constraints in conjunction with the MMA loss function did improve readability compared to other baselines.

Model	Readability	Relevance
Pointer Network	4.81	4.32
BART	7.32	4.76
Dual Model	6.32	6.54
CodeBERT	6.48	7.14
Our Model	7.21	7.43

Table 4: Manual evaluation results

Evaluation Setup To perform human evaluation, we randomly selected 100 test examples from the <code,nl> dataset. For each example, we showed the evaluator the source code, the ground truth summary as well as summaries generated by different models. The human evaluator does not know which summary is the ground truth and which model the summary is generated from. We employed two metrics, scored from 1 to 10, assessing relevance (summary's alignment with source code content) and readability. Five evaluators from network operations and maintenance scored each summary, with final results being the average across evaluators and examples. The evaluation criteria focus on several key dimensions: the precision of the summary content, the accurate use of technical vocabulary, the clear expression of configuration instructions, and the coherence and readability of the entire summary structure. This assessment design aims to ensure that the generated summaries are rigorous on a technical level and can be easily understood and adopted by network engineers and developers.

Results While CodeBERT excels in METEOR and ROUGE-L metrics, its readability lags. A BART-like model yields fluent sentences but lacks task-specific relevance. Conversely, our model scores high on both metrics, underscoring the efficacy of our approach for code summarization.

8. Hyper-Parameter Analysis

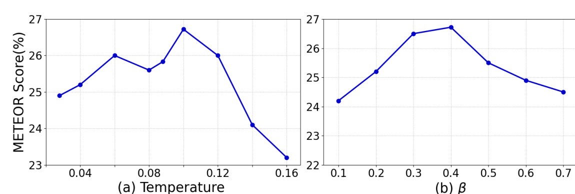


Figure 6: METEOR score under different temperature coefficients and β

8.1. Effect of Temperature in Loss Function

The temperature coefficient is instrumental in regulating attention towards challenging samples. Specifically, a smaller coefficient amplifies the emphasis on distinguishing such samples from their most similar counterparts. In our exploration of supervised contrastive learning, we found that this temperature, when optimally set in the loss function, can enhance model performance by nearly 3%. Experimental results indicated that a temperature coefficient of 0.1 yielded the best METEOR score. Detailed results across varying temperature values are illustrated in Figure 6(a).

Positive	1	3	5	7	9
METEOR	25.2	25.6	26.7	26.4	26.2

Table 5: METEOR score under different number of positive samples

8.2. Effect of Hyperparameter β

The hyperparameter β balances the two loss functions. As illustrated in Figure 6(b), an optimal outcome is achieved with a value of 0.3.

8.3. Effect of Number of Positives

We conducted ablation experiments to assess the influence of the number of positive samples on our results. As presented in Table 5, optimal results were achieved with five positive samples. It's noteworthy that each batch of positive samples consists of commands from the same category. A single positive sample equates to self-supervised contrastive learning.

9. Conclusion

In this study, we introduce a distantly supervised contrastive learning approach designed for scripting language summarization. This methodology is meticulously crafted, taking into account not only the categorical constraints that govern scripting languages but also the characteristic limited atomic commands that define this style of coding. Our approach is particularly tailored to harness the power of minimal parallel data to deliver summarization results that stand out in terms of quality and precision. Through a series of comprehensive experiments, we rigorously test and validate the efficacy of our method, demonstrating its significant advantages and the potential it holds for advancing the field of code summarization.

10. Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under, Grant U23B2001, Grant 62201072, Grant 62171057, Grant 62101064, Grant 62001054, and Grant 62071067; in part by the Ministry of Education and China Mobile Joint Fund under Grant MCM20200202 and Grant MCM20180101; in part by the BUPT-China Mobile Research Institute Joint Innovation Center.

11. Bibliographical References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Philip Bachman, R Devon Hjelm, and William Buchwalter. 2019. Learning representations by maximizing mutual information across views. *Advances in neural information processing systems*, 32.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Piotr Bojanowski and Armand Joulin. 2017. Unsupervised learning by predicting noise. In *International Conference on Machine Learning*, pages 517–526. PMLR.

Sergiy V Borodachov, Douglas P Hardin, and Edward B Saff. 2019. *Discrete energy on rectifiable sets*. Springer.

BSI. 1973a. *Natural Fibre Twines*, 3rd edition. British Standards Institution, London. BS 2570.

BSI. 1973b. Natural fibre twines. BS 2570, British Standards Institution, London. 3rd. edn.

A. Castor and L. E. Pollux. 1992. The use of user modelling to guide inference and learning. *Applied Intelligence*, 2(1):37–53.

Huangxun Chen, Yukai Miao, Li Chen, Haifeng Sun, Hong Xu, Libin Liu, Gong Zhang, and Wei Wang. 2022. Software-defined network assimilation: bridging the last mile towards centralized network configuration management with nassim. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 281–297.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR.

J.L. Chercheur. 1994. *Case-Based Reasoning*, 2nd edition. Morgan Kaufman Publishers, San Mateo, CA.

N. Chomsky. 1973. Conditions on transformations. In *A festschrift for Morris Halle*, New York. Holt, Rinehart & Winston.

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

Umberto Eco. 1990. *The Limits of Interpretation*. Indian University Press.

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

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*.

Paul Gerhard Hoel. 1971a. *Elementary Statistics*, 3rd edition. Wiley series in probability and mathematical statistics. Wiley, New York, Chichester. ISBN 0 471 40300.

Paul Gerhard Hoel. 1971b. *Elementary Statistics*, 3rd edition, Wiley series in probability and mathematical statistics, pages 19–33. Wiley, New York, Chichester. ISBN 0 471 40300.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50.

Otto Jespersen. 1922. *Language: Its Nature, Development, and Origin*. Allen and Unwin.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *Advances in Neural Information Processing Systems*, 33:18661–18673.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.

Yuding Liang and Kenny Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. *arXiv preprint arXiv:1802.08979*.

Ralph Linsker. 1988. Self-organization in a perceptual network. *Computer*, 21(3):105–117.

Shikun Liu, Edward Johns, and Andrew J Davison. 2019a. End-to-end multi-task learning with attention. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1871–1880.

Weiyang Liu, Rongmei Lin, Zhen Liu, Lixin Liu, Zhiding Yu, Bo Dai, and Le Song. 2018. Learning towards minimum hyperspherical energy. *Advances in neural information processing systems*, 31.

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

Paul W McBurney and Collin McMillan. 2015. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119.

Paul McGuire. 2007. *Getting started with pyparsing*. " O'Reilly Media, Inc.".

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

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

Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2021. Neural code summarization: How far are we? *arXiv preprint arXiv:2107.07112*.

Charles Joseph Singer, E. J. Holmyard, and A. R. Hall, editors. 1954–58. *A history of technology*. Oxford University Press, London. 5 vol.

Jannik Strötgen and Michael Gertz. 2012. Temporal tagging on different domains: Challenges, strategies, and gold standards. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, pages 3746–3753, Istanbul, Turkey. European Language Resource Association (ELRA).

S. Superman, B. Batman, C. Catwoman, and S. Spiderman. 2000. *Superheroes experiences with books*, 20th edition. The Phantom Editors Associates, Gotham City.

Joseph John Thomson. 1904. Xxiv. on the structure of the atom: an investigation of the stability and periods of oscillation of a number of corpuscles arranged at equal intervals around the circumference of a circle; with application of the results to the theory of atomic structure. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 7(39):237–265.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. *Advances in neural information processing systems*, 28.

Feng Wang, Xiang Xiang, Jian Cheng, and Alan Loddon Yuille. 2017. Normface: L2 hypersphere embedding for face verification. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1041–1049.

Tongzhou Wang and Phillip Isola. 2020. Understanding contrastive representation learning through alignment and uniformity on the hypersphere. In *International Conference on Machine Learning*, pages 9929–9939. PMLR.

Zhennan Wang, Canqun Xiang, Wenbin Zou, and Chen Xu. 2020. Mma regularization: Decorrelating weights of neural networks by maximizing the minimal angles. *Advances in Neural Information Processing Systems*, 33:19099–19110.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32.

Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976.

Rui Xie, Wei Ye, Jinan Sun, and Shikun Zhang. 2021. Exploiting method names to improve code summarization: A deliberation multi-task learning approach. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 138–148. IEEE.

Jiacheng Xu, Zhe Gan, Yu Cheng, and Jingjing Liu. 2020. Discourse-aware neural extractive text summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5021–5031, Online. Association for Computational Linguistics.

Yuxiang Zhu and Minxue Pan. 2019. Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352*.

12. Appendices

12.1. Case Study

To demonstrate the superiority of our method, several examples on the test set, which are generated by our method and the existing competitive baselines, are given in Table 6. Specifically, in the first example, both Pointer Network and CodeBERT are missing the scenario information “BGP”, and CodeBERT summarizes “selection” as “add ”, which is an inaccurate representation of the information. In the second example, Pointer Network simply picks words from the source code and the sentence does not make sense, and CodeBERT omits the function description of “receive capability”.

router bgp [as-number] # address-family vpnv4 unicast # additional-paths selection -route-policy [route-policy-name] **Gold**: enable selection of additional paths under BGP routing process. **Pointer Network**: enable additional paths selection. **CodeBERT**: it show how to add additional paths. **Our method**: it enable selection of additional paths when config BGP.
router bgp [as-number] # neighbor [ip-address] # address-family ipv4 unicast # aigp **Gold**: enable aigp send and receive capability under neighbor address family ipv4 unicast. **Pointer Network**: enable address family ipv4 unicast under neighbor. **CodeBERT**: enable aigp under ipv4 unicast. **Our method**: enable aigp and receive capability under ipv4 .
dhcp ipv4 # profile client proxy # helper-address vrf vrf1 foo [ip-address] **Gold**: enter the DHCP ipv4 profile proxy submode and forward UDP broadcastrs **Pointer Network**: enter DHCP ipv4 profile client proxy **CodeBERT**: enter DHCP profile proxy submode and forward broadcastrs **Our method**: enter DHCP ipv4 proxy submode and forwarf UDP broadcastrs

Table 6: Summarizations generate by our method, Pointer Network and CodeBERT from the Cisco test dataset.

Length	5	10	15	20	25	≥ 25
%	8.4	29.3	25.8	20.3	11.5	4.7

Table 7: Code length statistics

12.2. Statistics Analysis

Code length statistics Table 7 illustrates the distribution of code lengths, indicating the proportion of codes with varying numbers of commands. For instance, ’5’ represents code lengths ranging from 0-5 commands, and so forth. The majority of the code lengths fall within the 5-25 command range.

Category distribution statistics As shown in the Figure 7(a), the horizontal axis represents the length of the source code and the vertical axis represents the number of identical categories. For each length, we randomly selected the same number of codes for counting. When the code length is 0-5, a segment of code will contain an average of 2 or 3 command categories that have appeared

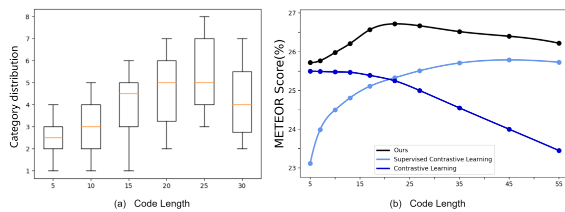


Figure 7: Category distribution and model performance under different lengths

```
word = OneOrMore(p.Word(p.printables, excludeChars='{}[]#\n|')).setParseAction(groupornot)
ele = p.Forward()
items = p.Group(ele) + p.ZeroOrMore('|' + p.Group(ele))
select = p.ZeroOrMore(word) + p.Group(Suppress('{') + items + Suppress('}'))
option = p.ZeroOrMore(word) + p.Group('['+ items + ']')
ele <<= p.OneOrMore(option^select^word)
syntax_parser = ele
```

Figure 8: Code snippet for syntax parser

in different segments of the same length. Furthermore, the number of identical categories increases as the length of the code increases, and the two are roughly positively correlated. This also demonstrates the feasibility of treating categories as "weak labels" in different code segments.

Impact of code length We investigated how code length (number of commands) influences summary performance. The test data was segmented based on code length. Given that most of our data has a length between 5 to 25 commands, we sampled more from this range to balance the data distribution. The comparative performance of the three methods, as per the METEOR metric, is showcased in Figure 7(b). Key observations include: As code length increases, there's an evident inverse relationship between supervised and unsupervised contrastive learning. The former emphasizes command categories, while the latter focuses on individual commands. Models prioritizing only categories tend to underperform for shorter code lengths. One plausible reason is the critical role each command plays in achieving a specific function in shorter codes, necessitating instance-level differentiation. For longer codes, the significance of a single command diminishes, making categorization more apt. Notably, supervised contrastive learning excels in longer code scenarios. Our method considers both categories and individual commands, ensuring robust performance across different code lengths.

12.3. Command Template Parsing

Figure 9 illustrates the basic syntax rules of the configuration language. The font requirements are merely for display distinction between variables and keywords; during parsing, we enclose parameter with '<>'. The specific parser can be easily written based on its syntax rules. For example, curly braces { } are used to denote selected branches, and square brackets [ ] indicate optional branches. We can utilize syntax parsing generators, such as pyparsing (McGuire, 2007) (Figure 8), to parse our command templates.

Template
clear ip bgp [ipv4 {unicast | multicast} | all] dampening [neighbor | prefix]
Commands
clear ip bgp dampening
clear ip bgp dampening neighbor
clear ip bgp dampening prefix
clear ip bgp ipv4 unicast dampening
clear ip bgp ipv4 unicast dampening neighbor
clear ip bgp ipv4 unicast dampening prefix
clear ip bgp ipv4 multicast dampening
clear ip bgp ipv4 multicast dampening neighbor
clear ip bgp ipv4 multicast dampening prefix
clear ip bgp all dampening
clear ip bgp all dampening neighbor
clear ip bgp all dampening prefix
Template
eigrp stub [direct | leak-map <map-name> | receive-only | redistributed]
Commands
eigrp stub
eigrp stub direct
eigrp stub leak-map <map-name>
eigrp stub receive-only
eigrp stub redistributed

Table 8: Parsing Examples of Templates

Convention	Description
boldface font	Commands and keywords are in boldface.
italic font	Arguments for which you supply values are in italics.
[ ]	Elements in square brackets are optional.
{x | y | z}	Alternative keywords are grouped in braces and separated by vertical bars.
[x | y | z]	Optional alternative keywords are grouped in brackets and separated by vertical bars.
string	A nonquoted set of characters. Do not use quotation marks around the string or the string will include the quotation marks.

Figure 9: The basic syntax structure of config code