Code representation learning

Cyprien DERUELLE and **Julien PEREZ**
and **Idir BENOUARET**
Laboratory of Epita (LRE)

July 2024

Contents

1 Abstract

Machine-assisted programming tools, such as type predictors and code summaries, increasingly rely on learning techniques. However, most current approaches for learning code representations depend on task-specific annotated data, which limits their generalizability. During my semester in the lab, I explored and tested various CodeLLM models with the goal of improving performance. I drew inspiration from the self-supervised method used by ContraCode (Paras Jain et al. 2021), an innovative approach that creates robust semantic representations without needing human annotations. This method enhances the model's ability to understand and generate code, making it more versatile and effective.

2 Introduction

2.1 Context

Language Models for Code (CodeLLM) are advanced AI models that utilize deep learning to analyze and generate computer code. These models have a profound understanding of both the syntax and semantics of various programming languages. By leveraging this capability, CodeLLMs can significantly enhance development tools, offering features such as intelligent code suggestions, automatic error detection, and correction. These improvements can streamline the coding process, increase productivity, and reduce the likelihood of bugs, making them invaluable assets for developers.

2.2 Llama 2

In order to better understand how these models work, I began studying the Code Llama paper (Meta-Llama. 2023. CodeLlama). This model offers fine-tuning of Llama 2 on specific tasks such as code infilling and code completion. It also includes specialized training on the Python language and the use of longer prompts. This first approach to improving Code LLM models places particular emphasis on a rigorous selection of the training dataset, thus ensuring the quality and relevance of the data used to optimize the performance of the model.

By studying these techniques, I was able to understand how careful data preparation and targeted training can significantly improve the capabilities of language models in specific programming tasks. This has allowed me to design strategies to adapt these methods to our own needs, aiming for greater accuracy and efficiency in the machine-assisted programming tools we develop.

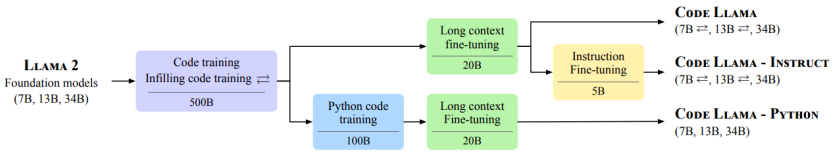

Figure 1: Code Llama

2.3 Goals

In my research, I found that the results obtained by Code Llama were particularly promising, outperforming most other models in tasks such as code infilling or generating code in Python. The method used by Code Llama allowed me to better understand the challenges associated with Language Models for Code (CodeLLM). However, this approach quickly showed its limits when it became less efficient than Llama 2 in certain specific areas.

This observation led me to ask myself an essential question: does the improvement of CodeLLMs only involve the selection of training data? And what would be the best technique to have the most efficient LLM code possible? At this point, I decided to study Contra Code, a method that quickly provided concrete answers to my questions.

3 Tool

3.1 LightningIA

As I am going to manipulate LLM code models I need to have access to GPUs to be able to launch my code more quickly for this I use Lightning IA (Lightning AI. 2023).

LightningAI is a platform designed to streamline the development and deployment of machine learning models. It provides tools for building, training, and scaling AI models with ease. The platform leverages PyTorch Lightning, an open-source framework that simplifies the process of writing high-performance, scalable deep learning code. LightningAI supports distributed training, automated logging, and experiment tracking, which helps in managing complex workflows efficiently. It also offers integrations with popular data and compute services, facilitating seamless end-to-end model development.

	Code compression		Identifier modification
✓	Reformatting (R)	✓	Variable renaming (VR)
✓	Beautification (B)	✓	Identifier mangling (IM)
✓	Compression (C)		**Regularization**
✓	Dead-code elimination (DCE)	✓	Dead-code insertion (DCI)
✓	Type upconversion (T)	✓	Subword regularization (SW)
✓	Constant folding (CF)	✗	Line subsampling (LS)

✓ = semantics-preserving transformation ✗ = lossy transformation

Figure 2: Augmented programs from ContraCode with 11 automated source-to-source compiler transforms. 10 are correct by-construction and preserve operational semantics.

4 ContraCode

4.1 Model overview

ContrasCode or Contrastive Code is an LLM Code model released in January 2022 which offers an interesting approach to our code representation problem. This paper presents a new method for learning semantic representations of code using self-supervised learning.

Instead, it uses contrastive learning, a technique that creates multiple semantically equivalent variants of code fragments through automated transformations.

These transformations help the model distinguish between different versions based on functionality rather than syntax, it compares codes based on their substance and not their form.

Contra Code's approach significantly improves traditional supervised learning methods by being pre-trained on 1.8 million unannotated JavaScript functions from different open source databases such as GitHub, improving performance in various tasks, such as code summarization and type prediction. Notably, this paper reports a 7.9% improvement in code synthesis accuracy over standard supervised methods and a 4.8% improvement over BERT-based pre-training (Simplilearn. 2019. BERT).

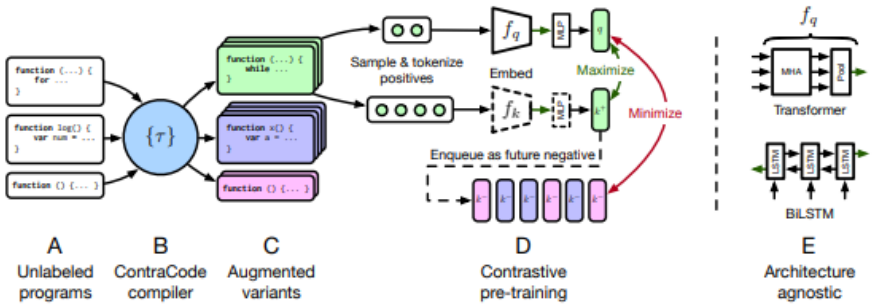

Figure 3: ContraCode pre-trains a neural program encoder fq and transfers it to downstream tasks. A-B. Unlabeled programs are transformed C. into augmented variants. D. We pre-train fq by maximizing similarity of projected embeddings of positive program pairs–variants of the same program–and minimizing similarity with a queue of cached negatives. E. ContraCode supports any architecture for fq that produces a global program embedding such as Transformers and LSTMs. fq is then fine-tuned on smaller labeled datasets

The model architecture used for ContraCode is flexible (Figure 3). Pre-trained models can be fine-tuned for specific tasks, demonstrating consistent accuracy improvements across different architectures. The Contra Code repository provides the tools and scripts needed to augment JavaScript programs, pre-train models, and fine-tune them for specific tasks.

4.2 Model testing

In the further study of this model I carried out various evaluation tests on this model, thanks to a database provided in the ContraCode paper I was able to reproduce the tests which were carried out on this model which is are described in his paper. In addition to the tests carried out in the paper, I did others on my own. Among the evaluation tests we find the Gram matrix, the MIPS (Wikipedia. 2023.), the confusion matrix (Scikit-learn. 2023.), pathological pairs or even the NDCG (Scikit-learn. 2023. NDCG).

4.2.1 Dataset

First, let's look at the dataset. It is made up of 100 files which each represent a 'type' of function. In each file there are several implementations of the same function. We are certain that two functions belonging to the same file, or in other words to the same type, have the same functionality (we are not interested in complexity).

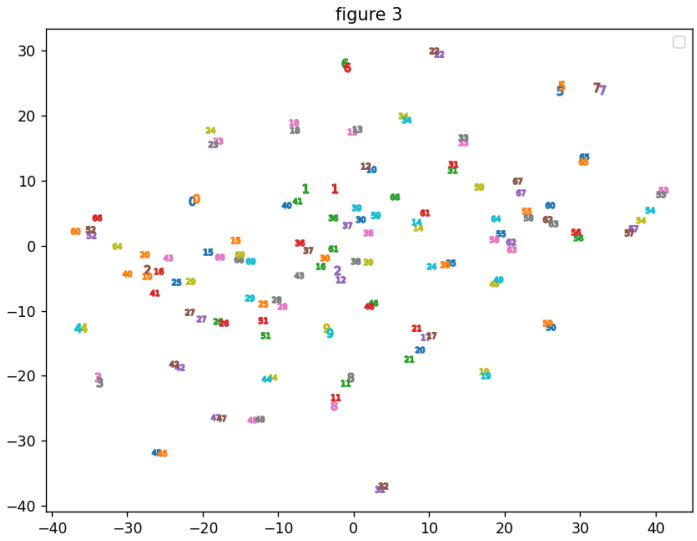

Figure 4: a UMAP visualization of JavaScript method representations learned by ContraCode, in R2.

4.2.2 Reproduction of Figure 3

The figure 3 in the ContraCode paper is a UMAP visualization of JavaScript method representations learned by ContraCode, in R2 (figure 4 in this paper). For each type of function, we take two examples, we calculate their embedding using the model and we project them into a two-dimensional grid. Using the TSNE library (Scikit-learn. 2023.) from Python greatly helps in this task and ensures consistent results. This graph shows us that most of the numbers are grouped together but we can start to see anomalies like numbers 11 and 8 which are abnormally close to each other.

4.2.3 Assessment tests

To better test the model I made a Gram matrix and a Confusion matrix. For this I had to respect a strict protocol to guarantee the quality of the results. For each function I calculated their embedding associated with the model then calculated their distances between them using a maximum inner product search (MIPS) which in our case corresponds to the matrix product. By seeking to maximize this result and classifying the figures obtained from largest to smallest, we obtain the Gram matrix (figure 5). This part of the Gram matrix shows us that the model is quite good at recognizing functions of the same type by classifying the good types in the top 3 of almost each function.

N° sample	1	2	3
Sample_0	0-4 (1698)	0-1 (1695)	1-9 (1658)
Sample_0	0-4 (1726)	1-9 (1708)	0-0 (1695)
Sample_0	1-0 (1596)	0-11 (1594)	0-14 (1585)
Sample_0	1-2 (1653)	1-5 (1642)	0-9 (1632)
Sample_0	0-1 (1726)	0-0 (1698)	1-9 (1685)
Sample_0	1-2 (1617)	1-5 (1606)	0-3 (1599)

Figure 5: Gram matrix, Each Sample number corresponds to one copy of a type (here several copies of type 0). The format "x-y (0000)" gives us information about the functions, 'x' corresponds to the type of function, 'y' to the copy number and the number in parentheses corresponds to the distance between the two functions.

Starting from this Gram matrix we can identify pathological pairs which correspond to a bad response in the Gram matrix. For example, the 3rd copy of type 0 is according to the table closer to type 1 than to type 0, it is therefore a pathological pair where the two functions are type 0 and 1. The pathological pairs that I created contain 3 parameters, the 'score' and the two functions concerned. The score corresponds to the distance between the two embedding functions. On this test dataset I identified 96 pathological pairs in the top ten of the Gram matrix. Which is quite a good score because there are a total of 1383 functions which are all associated with 10 other functions.

Pathological pairs can also be identified graphically using the confusion matrix (figure 6) which was used to evaluate the accuracy of the classifications made by the model. Each entry in the matrix represents the number of correct and incorrect predictions for each class, thereby identifying the strengths and weaknesses of the model. It summarizes the Gram matrix by selecting the ith results of a row and grouping each copy into its associated type. By doing this we end up with a list by type containing for each box the number of similarities with another type of function.

Of course, we must also establish a degree of selection to know up to which classification of the Gram matrix we take. In my case I chose to take the first 15. The reason for this comes from the fact that the database is not equally distributed, for each type there is not the same number of copies. In total there are 1383 functions but some types have more than 25 copies while others only have 15, so it is to obtain consistent results that I chose this number.

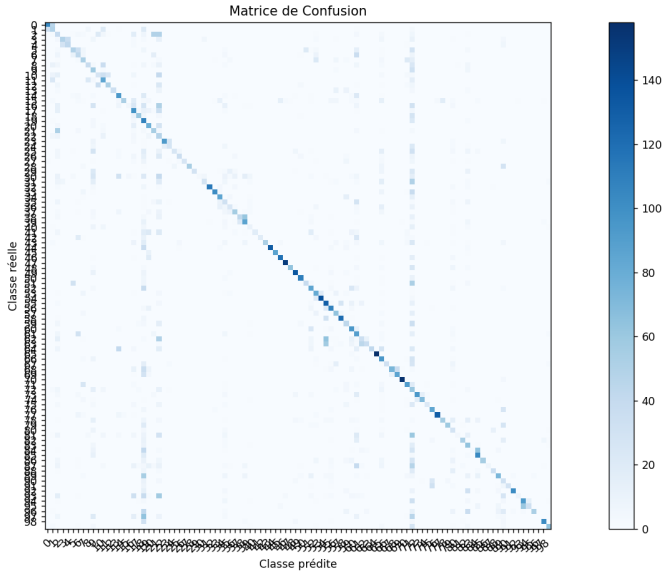

Figure 6: ContraCode degree 15 confusion matrix. As we can see, a diagonal appears to appear, which is the expected result. We can also identify pathological pairs which correspond to all the colored boxes outside this diagonal.

These matrices allowed me to get an overview of the model's performance, highlighting its ability to capture semantic relationships between code functions and perform accurate classifications.

To go further, we can also calculate NDCG (Normalized Discounted Cumulative Gain), an essential metric for evaluating the effectiveness of ranking systems, especially in the areas of information retrieval and recommendation systems. The NDCG measures the quality of a ranking by comparing the relevance of items in the predicted order versus the ideal order.

The DCG (Discounted Cumulative Gain) is calculated by summing the relevance scores of the elements, weighted logarithmically according to their position in the list. The general DCG formula is as follows:

$$\mathrm{DCG} = \Sigma_{i=1}^{p} \frac{rel_i}{log_2()i+1}$$

where rel_i is the relevance score of the element at position i, and p is the position up to which we want to calculate the DCG.

The NDCG is obtained by normalizing the DCG by the ideal DCG (IDCG), which is the DCG for the perfect rank order. The formula is:

$$\text{NDCG} = \frac{DCG}{IDCG}$$

The NDCG ranges between 0 and 1, where 1 indicates a perfect ranking corresponding exactly to the ideal order. This normalization makes it possible to compare the performance of different rankings on various queries and datasets.

In the tests carried out, ContraCode achieved an NDCG of 0.92, which indicates exceptional performance. This means that the model is very effective at producing code rankings that are close to the ideal order in terms of semantic relevance. This high NDCG score shows ContraCode's ability to understand and prioritize code in meaningful ways, outperforming traditional approaches based on annotated datasets.

4.3 Results and analysis

The ContraCode model shows impressive performance thanks to its innovative self-supervised learning approach. By generating code variants through augmentation functions, the model becomes particularly effective for various programming applications. These hypotheses were verified during the tests previously discussed.

To go further, we will compare these results with those of other models following the same test protocols. Using metrics like NDCG and confusion matrices, we will be able to evaluate differences in performance and identify the strong and weak points of each model. This comparison will help determine whether ContraCode's approach offers significant advantages over traditional supervised learning methods based on annotated data.

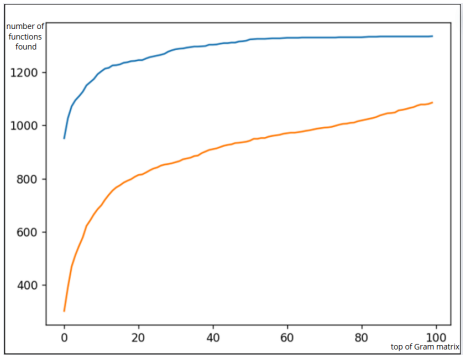

Figure 7: Curve representing the number of functions found following the selected top. The abscissa axis represents the number of functions found and the ordinate axis the top of the Gram matrix in which we search. The orange curve is that of Phi 1.5 and the blue one corresponds to ContraCode. We can clearly see that ContraCode recognizes more functions and faster.

4.4 Comparison

In order to compare the results obtained with those of another model, I carried out the same tests on the Microsoft phi 1.5 7B model (Microsoft. 2023.). The Gram matrix as well as the confusion matrix seems less precise than that of ContraCode and the NDCG reinforces this feeling with a result of 0.85.

In Figure 7 we can see that the Contracode model manages to find more functions than phi 1.5.

In summary, the results show that Contacode outperforms phi 1.5 in terms of performance according to NDCG metrics, with a higher score indicating better recommendation quality. Additionally, the confusion matrix comparison also suggests that Contacode is more accurate in its predictions compared to phi 1.5.

4.5 To go further (BEIR)

To go further in the tests I could draw inspiration from what BEIR (et al. 2021.) did for artificial intelligence textual models.

BEIR (Benchmarking Information Retrieval) is a comprehensive dataset and benchmarking framework designed to evaluate information retrieval models. It includes 18 datasets covering various domains and tasks, such as document and passage retrieval, question answering, and recommendation systems. The framework provides a standardized way to compare the performance of different retrieval models across these diverse tasks. BEIR emphasizes robustness and generalizability, aiming to push the development of retrieval models that perform well in a wide range of real-world scenarios. It supports various evaluation metrics and offers a consistent baseline for researchers to assess and improve their models.

All the tests I performed above could be a category of a larger benchmark to test CodeLLM more deeply. The tests performed this based on the latent representation generated by a model and analyses. We could therefore take inspiration from the types of tests on the BEIR paper to do new tests such as code completion or others.

5 Fined-tuned a model

Following the technique used by ContraCode I took the first steps to refine a model on the C language. I chose the C language because it is a language that I know well given that we used it throughout semester 5. Moreover, it is a language that I particularly like. To do this I had to follow a certain number of steps. First, I looked for a dataset, then a model to improve. Then I made increase functions and created a loss function

5.1 C Dataset

To carry out my research, I first identified a C language code dataset available on GitHub (TheAlgorithms. 2023.). This repository is a collection of diverse algorithms spanning mathematics, machine learning, computer science, physics, and other fields, all implemented in C. Criteria for selecting the dataset include that it be open source, moderate in size and sufficiently comprehensive.

5.1.1 Dataset Structure

The dataset is organized in a structured way, making it easy to navigate and use for researching Language Models for Code (CodeLLM). Here is an overview of the structure of the dataset:

1. Function Categories:
   The dataset contains a list of folders, each corresponding to a specific category of C functions. For example, there is a folder named sort for sorting algorithms, and another named game for game-related algorithms.

2. .c files:
   Each folder contains several .c files. Each file represents a different implementation of a function falling under the category of the folder. For example, the sort folder may contain files implementing different sorting algorithms like quick sort, insertion sort, etc.

5.1.2 Content Examples

To illustrate, here is how the content could look:

1. Folder output:
   quick_sort.c: Implementation of the quick sort algorithm.
   insertion_sort.c: Implementation of the insertion sort algorithm.
   merge_sort.c: Implementation of the merge sort algorithm.

2. Game folder:
   tic_tac_toe.c: Implementation of the Tic-Tac-Toe game.
   snake.c: Implementation of the Snake game.
   minesweeper.c: Implementation of the Minesweeper game.

5.2 Model to improve

After selecting the C code dataset, the next step was to choose a suitable LLM Code model for retraining. Several criteria guided this selection, notably performance optimization and cost. It was crucial to choose a model that performed well but was compact enough to be retrained effectively. For these reasons, I opted for the phi 1.5 model with 3 billion parameters (phi 1.5 3B).

The model had to offer high performance in terms of understanding and generating code. Phi 1.5 3B is known for its ability to understand complex code structures and generate relevant suggestions, making it a wise choice for a variety of programming tasks.

Very large models, with hundreds of billions of parameters, although extremely efficient, require considerable computational resources for training and inference. Phi 1.5 3B, with its 3 billion parameters, represents a good compromise between performance and cost. It is small enough to be retrained without requiring excessively powerful infrastructure, while still being large enough to capture complex nuances in the code.

Retraining large models can be costly in terms of time and hardware resources. By opting for an intermediate size model like phi 1.5 3B, it is possible to reduce these costs while maintaining high quality performance. This allows for faster and more cost-effective iterations, making it easier to experiment and improve the model.

5.3 Augmentation functions

Once the dataset and model are loaded, the next crucial step is to implement augmentation functions for the C code. Drawing inspiration from the techniques available for Python and JavaScript in ContraCode, I developed four augmentation functions. C-specific augmentation. These functions, when combined, generate hundreds of possible variants for a single function, thus enriching the dataset and improving the robustness of the model. Here is a detailed development of these four augmentation functions:

1. Deleting Comments

   The first augmentation function is to remove comments from the source code. Comments, while useful for developers, do not affect code execution. By removing them, one can create variations of the code that remain functionally the same but differ in appearance. This helps the model focus on the actual code rather than annotations.

2. Extracting Subsections from Code

   The second augmentation function extracts specific parts of the code, such as function blocks or loop segments. This approach allows generating code fragments that can be used to train the model on local code understanding tasks.

3. Introduction of Human Errors

   The third augmentation function imitates common human errors, such as typos or minor syntax errors. By introducing these errors, we can make the model more robust to error handling.

4. Code Reorganization

   The fourth augmentation function rearranges parts of the code, such as changing the order of variable declarations or functions, while maintaining the same functionality. This method helps the model understand different code structures and learn that the order of declarations

```python
def __call__(self, sample):
    text = sample.function
    lines = text.split("\n")
    is_next_comment = False

    list_lines = []
    for i in range(len(lines)):
        if "/*" in lines[i]:
            is_next_comment = True
        elif not is_next_comment:
            list_lines.append(i)
            is_next_comment = False
        if "*/" in lines[i]:
            is_next_comment = False
    sample.function = "\n".join([lines[i] for i in list_lines])
    return sample
```

Figure 8: Python code for a call to an increase function. This function removes comments from a C function.

can vary without changing the behavior of the program.

By combining these four augmentation functions, we can create a multitude of variations for a single function. For example, one can take a snippet of code without comments and introduce human error, or reorganize the code and extract a specific subsection. This multiplies the possibilities for variations, thus enriching the dataset significantly.

5.4 Loss Function

Once the data was increased and the model chosen, it was necessary to select a Loss function. The loss function is a crucial indicator for adjusting model weights during training. Depending on the situation, we either seek to maximize or minimize this function. In the case where the function is associated with a positive variant, the objective is to maximize it, while for a negative variant, we seek to minimize it.

For tasks involving semantic similarity between code fragments, a commonly used loss function is inspired by contrast techniques, such as those used in ContraCode models. Here is the formula for the chosen loss function:

$$Loss = -log \frac{exp(q*k^+/t)}{exp(q*k^+/t)+\sum_{k-} exp(q*k^-/t)}$$

where :

q is the representation of the query.

k^+ is the positive representation (positive variant).

k^- is the positive representation (negative variant).

t is a temperature which controls the concentration of the distribution (scaling factor).

6 Conclusion

My semester project in the laboratory allowed me to explore advanced learning techniques for Code LLMs. Through this study, I compared traditional and innovative approaches, including those of Code Llama and ContraCode, to draw valuable lessons. To then turn to the beginning of improvement of the phi 1.5 model where I still have the whole retraining part to do.
In conclusion, the lessons drawn from advanced learning techniques in Code LLMs underscore the potential for significant improvements in the phi 1.5 model. By leveraging data diversity the retraining process can substantially enhance the model's performance, making it a more powerful and reliable tool for code generation and completion tasks.

Acknowledgments

I thank Julien PEREZ, Idir BENOUARET, Lucas COLLEMARE, Ilyas OULKADDA for their valuable help.

References

Microsoft. 2023. Phi-1_5 on Hugging Face: A Specialized Natural Language Processing Model for Coding Tasks. Designed for code generation, infilling, and code understanding while remaining compact and efficient.

PyTorch Team. 2023. PyTorch-Transformers: Library providing state-of-the-art implementations for various transformer models, making it easy to train and use advanced NLP models in research and industrial application projects.

OpenAI. 2021. HumanEval: a benchmark used to evaluate the perfor-

mance of code generation models based on human programming tasks, providing a measure of the quality and accuracy of the code generated by the models.

Paras Jain et al. 2021. ContraCode: a self-supervised learning framework for code generation, using syntactic transformations to improve the robustness and generalization of code models, without requiring human annotations. (https://github.com/parasj/contracode)

Ahmed Raza et al. 2023. A Survey on Machine Learning for Code Intelligence: an in-depth study of different machine learning techniques applied to code intelligence, covering the latest advances and challenges in the field. (https://arxiv.org/abs/2310.11511)

Simplilearn. 2019. BERT Neural Network – EXPLAINED! : an explanatory video on YouTube detailing how the BERT (Bidirectional Encoder Representations from Transformers) model works, its applications and its impact on natural language processing.

Scikit-learn. 2023. t-SNE: A dimensionality reduction method for high-dimensional data visualization using the t-Distributed Stochastic Neighbor Embedding (t-SNE) algorithm, integrated into the scikit-learn library. (https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html)

Liwei Wu et al. 2023. Code Transforms for Large-Scale Self-Supervised Learning: An article introducing large-scale code transformations for self-supervised learning, improving the performance of code generation and understanding models.
(https:/ /arxiv.org/abs/2306.03234)

Scikit-learn. 2023. NDCG Score: a scikit-learn library function used to calculate the normalized Discounted Cumulative Gain (DCG) score, a measure of the effectiveness of recommendation and information retrieval systems. (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.ndcg_score.html)

Hugging Face. 2023. Feature Extraction: An overview of feature extraction tasks available on the Hugging Face platform, including pre-trained models for various natural language processing and data analysis applications. (https://huggingface.co/tasks/feature-extraction)

Scikit-learn. 2023. Confusion Matrix: A function in the scikit-learn library to generate confusion matrices, providing a visualization of model classification performance by showing the rates of true positives, false positives, true negatives, and false negatives.
(https ://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html)

Tom B. Brown et al. 2020. Language Models are Few-Shot Learners:

an article detailing the capabilities of language models such as GPT-3 in learning tasks with few examples, demonstrating their effectiveness and flexibility in various NLP applications.
(https://arxiv.org/pdf/2104.08663.pdf)

Nils Reimers et al. 2021. BEIR: A collection of benchmarks for evaluating information retrieval models, providing a standardized platform for comparing the effectiveness of document retrieval and question answering methods.
(https://huggingface.co/BeIR)

Wikipedia. 2023. Maximum Inner Product Search: A page explaining the maximum inner product search technique, used to find the most similar elements in a vector space, often used in recommendation systems and search engines.
(https://en.wikipedia.org/wiki/Maximum_inner-product_search)

Lightning AI. 2023. Lightning AI: An artificial intelligence model development and deployment platform, offering tools to accelerate model development cycles from experimentation to production. (https://lightning.ai/)

Ham Vocke. 2019. A Quick and Easy Guide to tmux: A practical guide to using tmux, a terminal multiplexer that allows multiple terminal sessions to be managed in a single window, improving the efficiency of command line work.
(https://hamvocke.com/blog/a-quick -and-easy-guide-to-tmux/)

Veera Lakrishna. 2021. 150k Python Dataset: a dataset available on Kaggle containing 150,000 Python code fragments, used for code analysis tasks, machine learning and evaluation of code generation models.
(https://www.kaggle.com/datasets/veeralakrishna/150k-python-dataset)

TheAlgorithms. 2023. Collection of C Programs: an open-source collection of various algorithms implemented in C, covering areas such as mathematics, machine learning, computer science and physics, available for educational and research purposes.

CUHK-ARISE. 2023. ML4Code Dataset: a dataset compiled by the University of Hong Kong containing programs and annotations used for machine learning applied to code analysis and generation.
(https://github.com/CUHK -ARISE/ml4code-dataset)

Papers with Code. 2023. Information Retrieval Datasets: A list of information retrieval datasets available from Papers with Code, providing resources for evaluating and comparing information retrieval models.
(https://paperswithcode.com/datasets?task=information-retrieval)

BigCode Project. 2023. The Stack v2: a multi-language source code dataset from various open-source repositories, intended for training and evaluating code processing models, hosted on Hugging Face. (https://huggingface.co/datasets/bigcode/the-stack-v2)

BigCode Project. 2023. The Stack v2 Deduplicated: a deduplicated version of The Stack v2 dataset, cleaned to eliminate redundancies and ensure the quality of the data used for training code models. (https://huggingface.co/datasets/bigcode/the-stack-v2-dedup)

Meta-Llama. 2023. CodeLlama: an advanced language model for code generation and understanding, leveraging deep learning techniques to analyze and generate computer code. (https://github.com/Meta-Llama/codellama)