



Ain Shams University
Faculty of Computer & Information Sciences
Mainstream Program

IPC: **Intelligent Programming** **Companion**

**This documentation submitted as required for
the degree of bachelors in
Mainstream Program
Computer and Information Sciences
Ain Shams University**

By

*Martina Al-Kes Angelos
Omar Yasser
Salma Mahdy*

*Mohamed Hesham
Salma Ayman
Yosef Mahmoud*

Supervisors
Prof. Dr. Ahmed Salah
*Computer Science Department,
Faculty of Computer and Information Sciences,
Ain Shams University*

T.A. Razan Bayoumi
*Computer Science Department
Faculty of Computer and Information Sciences,
Ain Shams University*

Cairo, June 2024

Acknowledgment

First and foremost, we are profoundly grateful to ALLAH, who provided us with the ability, strength, and perseverance to complete this work. We pray that He accepts our efforts and that this work serves its intended purpose.

We would like to extend our sincerest gratitude to our supervisors, *Prof. Dr. Ahmed Salah* and *T.A. Razan Bayoumi*. Their unwavering support, patience, knowledge, and experience were instrumental in guiding us through this journey. Without their invaluable advice and encouragement, this thesis would not have been possible.

We also wish to express our heartfelt thanks to our families and friends. Their continuous support, understanding, and encouragement have been a constant source of motivation for us.

Lastly, we want to emphasize that the primary motive behind this project was to give back to the community. We hope that our work contributes positively and meaningfully to society.

Abstract

There are several critical challenges encountered by competitive programmers specifically and software engineers preparing for interviews generally, such as: uncertainty on how to prepare, problem repetition, inconsistent difficulty assessments, platform fragmentation, and ambiguous problem descriptions, all of which impede effective learning and fair competition. To tackle these issues comprehensively, a centralized platform is proposed, featuring four main modules: problem summarization, technique-based and semantic similarity for problems and solutions, problem difficulty-level and tags prediction, and, finally, a solver for standard problems that can appear in an interview. The Summarization Module leverages Google's Gemini 1.5 architecture, with advanced prompt engineering techniques. The Similarity Module uses the nearest-vector approach, utilizing state-of-the-art vector databases to find most similar problems/solutions. It achieves a remarkable 97.5% recall@10 on problem statements and 99.6% recall@10 on solutions. Meanwhile, the Difficulty/Tags Prediction Module applies a BigBird BERT-based architecture to accurately predict problem difficulty levels and assign relevant tags, demonstrating a robust performance with an ROC AUC score of 76.37%. Lastly, the solver module is presented as a helpful interactive chatbot for dynamic problem-solving, built on a RAG architecture with best-in-class open-weight CodeQwen LLM. The presented results underscore the efficacy and practicality of our proposed solutions. Through the implementation of this platform, our aim is to foster a more cohesive and supportive environment for competitive programmers and software engineer interview candidates. By encouraging innovation, promoting knowledge sharing, and enhancing problem-solving experiences, our system aims not only to elevate current standards in competitive programming and interview preparation, but also to lay the groundwork for future advancements in Computer Science education and problem-solving methodologies.

ملخص

تواجه المبرمجين التنافسيين بشكل خاص والمهندسين البرمجيات الذين يستعدون للمقابلات بشكل عام عدة تحديات حرجية، مثل: عدم اليقين حول كيفية التحضير، تكرار المشاكل، تقييمات الصعوبة غير المتسلقة، تجزئة المنصات، ووصف المشاكل الغامض، وكل ذلك يعيق التعلم الفعال والمنافسة العادلة. لمعالجة هذه القضايا بشكل شامل، يتم اقتراح منصة مركزية تتضمن أربع وحدات رئيسية: تلخيص المشاكل، تشابه تقني ودلالي للمشاكل والحلول، تنبؤ مستوى صعوبة المشاكل وتصنيفاتها، وأخيراً، محلل للمشاكل القياسية التي يمكن أن تظهر في مقابلة. تستفيد وحدة التلخيص من بنية Google's Gemini 1.5، مع تقنيات هندسة متقدمة للمطالبات. تستخدم وحدة التشابه نهج أقرب-متوجه، باستخدام قواعد بيانات متوجهة متطرفة للعنصر على أكثر المشاكل والحلول تشابهاً. تحقق Recall@10 بنسبة 97.5٪ على أوصاف المشاكل و 99.6٪ على الحلول. في الوقت نفسه، تطبق وحدة تنبؤ الصعوبة/التصنيفات بنية BigBird BERT للتنبؤ بدقة بمستويات صعوبة المشاكل وتحديد التصنيفات ذات الصلة، مع أداء قوي بدرجة ROC AUC تبلغ 76.37٪. أخيراً، يتم تقديم وحدة المحلل كروبوت دردشة تفاعلية للمساعدة في حل المشاكل الديناميكية، مبني على بنية RAG مع LLM CodeQwen مفتوح الأوزان. تؤكد النتائج المقدمة فعالية وحيوية الحلول المقترحة. من خلال تنفيذ هذه المنصة، هدفنا هو تعزيز بيئة أكثر تماساً ودعمًا للمبرمجين التنافسيين ومرشحي مقابلات مهندسي البرمجيات. من خلال تشجيع الابتكار، وتعزيز تبادل المعرفة، وتحسين تجارب حل المشاكل، يهدف نظامنا ليس فقط إلى رفع المعايير الحالية في البرمجة التنافسية والتحضير للمقابلات، ولكن أيضاً إلى وضع الأساس للتقدم المستقبلي في تعليم علوم الحاسوب ومنهجيات حل المشاكل.

Table of Contents

Acknowledgment.....	I
Abstract.....	II
List of Figures.....	VI
List of Tables.....	VII
List of Abbreviations.....	VIII
Chapter 1.....	1
Introduction.....	1
1.1 Problem Definition.....	1
1.2 Motivation.....	2
1.3 Objectives.....	2
1.4 Methodology.....	3
1.4.1 Problem Summarization Module.....	3
1.4.2 Problem/Solution Similarity Module.....	3
1.4.3 Difficulty/Tags Prediction Module.....	3
1.4.4 Solver Module.....	4
1.5 Time Plan.....	4
1.6 Thesis Outline.....	5
Chapter 2.....	7
Literature Review.....	7
2.1 Problem Summarization Module.....	7
2.2 Problem/Solution Similarity Module.....	8
2.2.1 Problem Similarity Module.....	8
2.2.2 Solution Similarity Module.....	9
2.3 Difficulty/Tags Prediction Module.....	10
2.3.1 Predicting Problem Tags.....	10
2.3.2 Multi-Task Solution Using Deep Learning.....	10
2.4 Solver Module.....	11
Chapter 3.....	14
System Architecture and Methods.....	14
3.1 System Architecture.....	14
3.2 Description of Methods and Procedures.....	16

3.2.1 Problem Summarization Module.....	16
3.2.2 Problem/Solution Similarity Module.....	18
3.2.3 Difficulty/Tags Prediction Module.....	21
3.2.4 Solver Module.....	23
Chapter 4.....	28
System Implementation and Results.....	28
4.1 Dataset.....	28
4.2 Description of Software Tools Used.....	31
4.3 Experimental and Results.....	35
4.3.1 Problem Summarization Module.....	35
4.3.2 Problem/Solution Similarity Module.....	36
4.3.3 Difficulty/Tags Prediction Module.....	42
4.3.4 Solver Module.....	45
Chapter 5.....	50
Run the Application.....	50
Chapter 6.....	56
Conclusion and Future Work.....	56
6.1 Conclusion.....	56
6.2 Recommendations for Future Work.....	57
References.....	58

List of Figures

Figure 1.1: Time Plan	4
Figure 2.1: AlphaCode's impressive rating on Codeforces	12
Figure 3.1: Three Tier System Architecture	14
Figure 3.2: Transformer's Architecture	16
Figure 3.3: BERT's Architecture	17
Figure 3.4: Clustering Approach used	19
Figure 3.5: How Nearest-Vector approach works	20
Figure 3.6: Predicting the Difficulty and Tags simultaneously	21
Figure 3.7: Network Structure of a BERT Model	22
Figure 3.8: Difference between BigBird and other attention mechanisms	23
Figure 3.9: RAG architecture explaining how it works	24
Figure 3.10: Difference between Qwen and a Decoder-only architecture	26
Figure 4.1: Manual Assessment results for problems similarity using nearest vector approach	39
Figure 4.2: Manual Assessment results for problems similarity using clustering approach	39
Figure 4.3: Manual Assessment results for solutions similarity using nearest vector approach	40
Figure 4.4: Manual Assessment results for solutions similarity using clustering approach	40
Figure 4.5: ROC curve	42
Figure 4.6: ROC curves of various models	44
Figure 4.7: ROC before and after fine-tuning	44
Figure 4.8: How RAG influences LLM response	49
Figure 5.1: Summarization page	50
Figure 5.2: A summarized problem statement	51
Figure 5.3: The similarity page	51
Figure 5.4: List of similar problems to a given statement and solution	52
Figure 5.5: Difficulty/Tag page	53
Figure 5.6: Solver page	54
Figure 5.7: The solver providing a comprehensive solution and explanation to a given problem query	55

List of Tables

Table 4.1: Problems Dataset Details.....	28
Table 4.2: Solutions Dataset Details.....	28
Table 4.3: Performance of different LLMs on different benchmarks.....	36
Table 4.4: Silhouette scores for different clustering algorithms with different embeddings for problem statements.....	37
Table 4.5: Silhouette scores for different clustering algorithms with different embeddings for solutions.....	38
Table 4.6: recall@k results for problem statements.....	41
Table 4.7: recall@k results for solutions source codes.....	41
Table 4.8: Different LLMs assessed on various benchmarks.....	47
Table 4.9: Different LLMs assessed on LiveCodeBench.....	48

List of Abbreviations

Abbreviation	Description
LLM	Large Language Models
RAG	Retrieval Augmented Generation
MBPP	Model-based Policy Optimization
RMSNorm	Root Mean Square Normalization
SwiGLU	Switch Gated Linear Unit Activation Function
RoPE	Rotary Positional Embeddings
GPT	Generative Pre-trained Transformer
XML	Extensible Markup Language
BERT	Bidirectional Encoder Representations from Transformers (a language model)
MMLU	Massive Multitask Language Understanding
ROC	Receiver Operating Characteristic
AI	Artificial Intelligence
GLUE	General Language Understanding Evaluation (a benchmark for NLP tasks)
CNN	Convolutional Neural Network
NLP	Natural Language Programming
ANN	Artificial Neural Network
BLEU	Bilingual Evaluation Understudy
CoNaLa	Code/Natural Language Alignment

TAE	Type-Aware Embedding
TranX	Tree-based Transformer
API	Application Programming Interface
RL	Reinforcement Learning
UI	User Interface
PPO	Proximal Policy Optimization
BART	Bidirectional and Auto-Regressive Transformers
T5	Text-to-Text Transfer Transformer
PEGASUS	Pre-training with Extracted Gap-sentences for Abstractive Summarization
tf-idf	Term Frequency-Inverse Document Frequency
LayerNorm	Layer Normalization
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Problem Definition

The field of Computer Science (CS) is renowned for its rapid evolution and the perpetual learning required from students, employees, and employers alike. Among the various domains within CS, problem-solving has emerged as a particularly challenging and popular pursuit. Despite its popularity, it remains one of the most difficult areas, requiring continuous practice and upskilling.

A recurring question in this field is: “**How can one best prepare for interviews and competitive programming contests?**” While numerous platforms host a vast array of problems spanning diverse topics, several significant issues have emerged, impacting both interview preparation and the competitive programming community:

- **Repetition and Uniqueness:** Problem setters often create similar problems unknowingly, leading to many repeated problems. This repetition provides an unfair advantage to contestants familiar with these problems and diminishes the overall value of the problem sets.
- **Difficulty Assessment:** Problem setters frequently face challenges in determining the appropriate difficulty level for their problems. This inconsistency can hinder effective preparation and fair competition.
- **Platform Fragmentation:** Currently, there is no centralized platform for problem setters to share and collaborate on problems. This fragmentation results in problems being scattered across multiple websites, causing confusion for problem setters about where to share their content and making it difficult for individuals to find problems for practice.
- **Clarity of Problem Descriptions:** Many problem descriptions lack clarity, posing challenges for both human solvers and AI tools like GPT-3 and

GitHub Copilot. Unclear problem statements hinder accurate comprehension and effective problem-solving.

Addressing these challenges will yield dual benefits, enhancing the ecosystem of the competitive programming community and improving strategies for interview preparation. By grouping problems, reducing repetition, and refining problem descriptions, we aim to create a more cohesive and supportive environment for all participants.

1.2 Motivation

The primary motivation for choosing this topic is our desire to give back to the community and ease the learning process for both learners and learning providers in one of the most challenging domains. By providing a gateway to problem-solving that can be utilized in competitive programming and interview preparation, we aim to create a harmonious ecosystem where knowledge is shared seamlessly, originality is celebrated, and learning is empowered through cutting-edge technologies. By addressing existing gaps in problem sharing, uniqueness validation, difficulty assessment, and AI-driven support, we envision a future where competitive programming enthusiasts can thrive, problem setters can innovate, and the entire community can collaborate towards excellence.

1.3 Objectives

Our project has several key objectives:

- **Create a Centralized Sharing Platform:** Develop a user-friendly platform that enables problem setters to effortlessly share their challenges, thereby reducing the fragmentation of problems across different websites.
- **Technique-based Problem Clustering:** Categorize problems based on the solving techniques they require, enabling individuals to practice specific techniques.
- **Enhance Originality and Uniqueness:** Empower problem setters with a mechanism to verify the uniqueness of their problems, reducing repetition and supporting the fairness of the competition.
- **Streamline Difficulty Assessment:** Equip problem setters with a dedicated tool for determining the difficulty level of their problems.

- **Summarize Problem Statements:** Provide clear summaries of problem statements to eliminate unnecessary information.
- **Develop a Standard Problem Solver:** Construct a robust tool proficient in resolving common standard problems, aiding individuals in solving complex problems.

1.4 Methodology

To address the identified challenges and achieve our objectives, we will develop and integrate four key modules into a centralized platform.

1.4.1 Problem Summarization Module

The summarization module utilizes a scientific methodology centered on text filtration, leveraging Google's Gemini 1.5 LLM and prompt engineering techniques. This approach focuses on crafting specific prompts that effectively filter out narratives, characters, legends, and contextual backgrounds from problem statements. By doing so, the module aims to distill the core essence of each problem, ensuring clarity and accessibility for all users, thereby enhancing understanding and facilitating efficient problem-solving in competitive programming and interview preparation contexts.

1.4.2 Problem/Solution Similarity Module

This module employs a nearest vector approach, where each problem/solution is represented as a vector in a high-dimensional space. By calculating the distance between vectors, we can identify and flag similar problems/solutions. This method is used to reduce the repetition of problems, ensuring that each problem remains unique and valuable.

1.4.3 Difficulty/Tags Prediction Module

This module utilizes a BigBird BERT-based architecture trained on labeled datasets to predict the difficulty level and relevant tags for problem statements. By extracting features from the statements, we can accurately guide users in selecting and solving problems. This method is used to provide a valuable guide for problem difficulty and categorization.

1.4.4 Solver Module

The RAG architecture enhances LLMs' ability to provide precise and context-specific responses to unstructured data. By integrating information retrieval with LLMs' generative power, RAG systems deliver high-quality answers across a variety of questions, bridging the gap between LLMs' static knowledge and the dynamic nature of evolving data. The solver module utilizes RAG and Qwen LLM capabilities to function as a chatbot providing accurate responses to diverse questions, effectively adapting to evolving data environments.

1.5 Time Plan

A detailed time plan is illustrated through a Gantt chart in [figure 1.1](#). It depicts the key stages, with their durations.

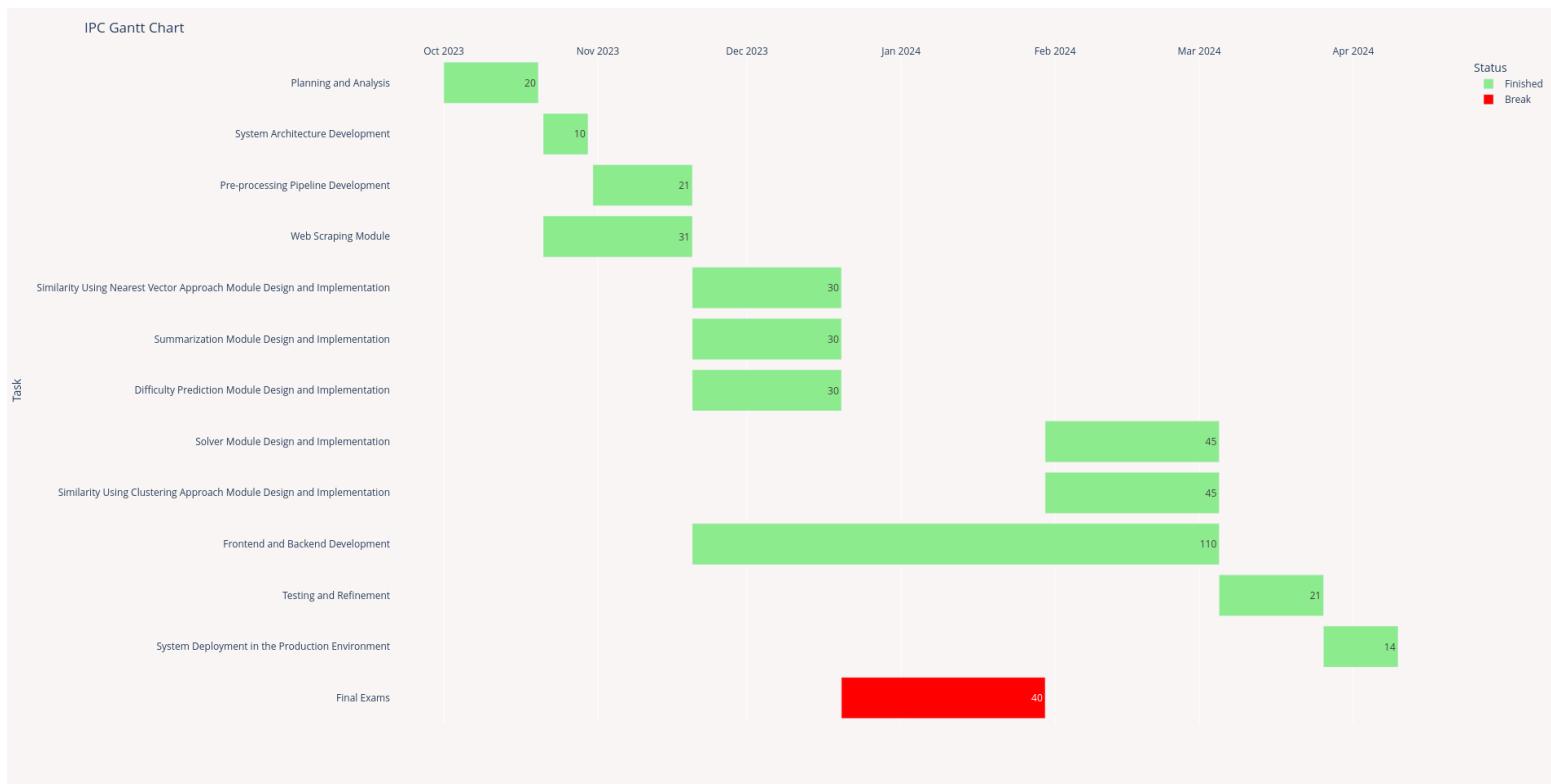


Figure 1.1: Time Plan

1.6 Thesis Outline

The thesis is structured as follows:

Chapter 2: Literature Review

Chapter 2 provides a comprehensive review of existing literature on competitive programming platforms, problem repetition, difficulty assessment methodologies, and techniques for improving problem statement clarity. It explores current solutions and gaps in the literature, laying the foundation for the methodologies and innovations proposed in this thesis.

Chapter 3: System Architecture and Methods

Chapter 3 outlines the architectural design of the proposed centralized platform and details the methodologies employed in developing the four key modules: Problem Summarization, Problem/Solution Similarity, Difficulty/Tags Prediction, and Solver. It discusses how each module contributes to addressing the identified challenges and achieving the objectives of the thesis.

Chapter 4: System Implementation and Results

Chapter 4 focuses on the practical implementation of the centralized platform, including details on the dataset used, software tools employed, and experimental results. It evaluates the performance and effectiveness of each module, particularly emphasizing the impact of the Summarization Module in enhancing problem statement clarity.

Chapter 5: Application and User Interaction

Chapter 5 guides users on utilizing the centralized platform, emphasizing the functionality and benefits of the Summarization Module in practical applications. It provides insights into user interface design, interaction flow, and the platform's role in facilitating effective problem-solving and learning experiences.

Chapter 6: Conclusion and Future Work

Chapter 6 summarizes the findings and contributions of the thesis, discussing the implications for the competitive programming community and future research directions. It highlights the achievements in addressing problem repetition, improving difficulty assessment, and enhancing problem statement clarity through innovative technologies. This chapter concludes with recommendations for further enhancing the capabilities of the centralized platform and advancing the field of competitive programming.

Chapter 2

Literature Review

2.1 Problem Summarization Module

Text Summarization is a natural language processing (NLP) task that involves condensing a lengthy text document into a shorter, more compact version while still retaining the most important information and meaning. The goal is to produce a summary that accurately represents the content of the original text in a concise form.

There are different approaches to text summarization, including extractive methods that identify and extract important sentences or phrases from the text, and abstractive methods that generate new text based on the content of the original text.

Text Summarization is an old problem and one of the most extensively studied in NLP.

Abigail et al. [23] introduced **Summarization with Pointer-Generator Networks**. They propose a novel architecture that augments the standard sequence-to-sequence attentional model in two orthogonal ways. First, they use a hybrid pointer-generator network that can copy words from the source text via pointing, which aids accurate reproduction of information, while retaining the ability to produce novel words through the generator. Second, they use coverage to keep track of what has been summarized, which discourages repetition. The model, applied to the CNN / Daily Mail summarization task, outperformed the then-current abstractive state-of-the-art models by at least 2 ROUGE points.

Jingqing et al. [24] proposed pre-training large Transformer-based encoder-decoder models on massive text corpora with a new self-supervised objective. Important sentences are removed/masked from an input document and are generated together as one output sequence from the remaining sentences, similar to an extractive summary. It achieved state-of-the-art performance on all downstream datasets measured by ROUGE scores. It also showed surprising

performance on low-resource summarization, surpassing previous state-of-the-art results on 6 datasets with only 1000 examples.

However, competitive programming problems often contain too many unnecessary details. Combining this with having no annotated dataset for summarized problems, then that model won't achieve the expected-good performance with our task.

2.2 Problem/Solution Similarity Module

2.2.1 Problem Similarity Module

Semantic textual similarity deals with determining how similar two pieces of texts are. One of the most popular architectures for this is **BERT**, by **Jacob et al. [15]**, which introduced a new language representation model, Bidirectional Encoder Representations from Transformers. Unlike recent language representation models, BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, including Semantic Similarity. It obtains state-of-the-art results, including pushing the GLUE score to 80.5%.

An improvement of this is introduced in **Sentence-BERT**, by **Nils et al. [25]**. BERT requires that both sentences are fed into the network, which causes a massive computational overhead: Finding the most similar pair in a collection of 10,000 sentences requires about 50 million inference computations (~65 hours) with BERT. The construction of BERT makes it unsuitable for semantic similarity search as well as for unsupervised tasks like clustering. Sentence-BERT (SBERT), introduced a modification of the pretrained BERT network that uses siamese and triplet network structures to derive semantically meaningful sentence embeddings that can be compared using cosine-similarity. This reduces the effort for finding the most similar pair from 65 hours with BERT to about 5 seconds with SBERT, while maintaining the accuracy from BERT.

2.2.2 Solution Similarity Module

Stoica et al. [1] developed an unsupervised voting algorithm for detecting algorithmic solving strategies in competitive programming solutions. This method leverages multiple code embedding techniques to analyze correct solutions for a set of competitive programming problems. Researchers employed a data analysis pipeline that incorporated five embedding methods and three algorithmic strategies on a dataset of 15 problems. They used a hard clustering algorithm on each embedding view to cluster solutions into distinct algorithmic approaches. The largest subset of solutions correctly mapped across views was used as a base for a co-training procedure, where each view's estimator voted on the algorithmic approach for solutions outside the subset. The results showed significant improvements over baseline clustering methods, with one exception, indicating that scaling up the pipeline could provide deeper insights into the innovative processes of code design in competitive programming and potentially in industry applications.

Other approaches have also been proposed for code analysis in competitive programming. For example, **S. Sudha et al.** [20] using Convolutional Neural Networks (CNNs) to classify C++ program solutions has been suggested to identify the required algorithmic design approach, such as segment tree, binary search, dynamic programming, and graph algorithms.

Another study, by **W. Looi et al.** [21] aimed to predict a user's rank and country based on a single C++ source code submission. Among the various models tested, Artificial Neural Networks (ANNs) performed the best, achieving accuracies of 77.2% for rank prediction and 72.5% for country prediction, surpassing classical machine learning algorithms like linear regression, Gaussian discriminant analysis, and logistic regression.

The availability of various code embeddings has enabled the use of Natural Language Processing (NLP) techniques for code analysis and synthesis. One approach by **J. Austin et al.** [22] successfully synthesized short Python programs from natural language descriptions, tackling 974 simple programming tasks designed for entry-level programmers.

However, competitive programming problems often have statements written in a way that obscures the necessary underlying algorithms, which can reduce performance.

2.3 Difficulty/Tags Prediction Module

2.3.1 Predicting Problem Tags

Recent studies by **Vinayak et al.** [36] have introduced a new research area termed Programming Word Problems (PMP), focusing on predicting algorithm tags from problem descriptions. A pivotal contribution in this domain was the introduction of a dataset comprising 4,019 problems annotated with over 10 algorithm tags. The study demonstrated that CNN-based classifiers can achieve near-human performance in predicting these tags. However, the effectiveness of their architectures was limited when handling the long sequences typical of algorithmic problem descriptions, often exceeding 1,000 words. Simple CNN architectures with fixed kernel sizes proved inadequate in capturing comprehensive global representations of such extensive texts. To address this challenge, a shift towards transformer-based architectures, known for their ability to effectively process long sequences, is proposed. Additionally, efforts are underway to expand the dataset by incorporating a larger volume of algorithmic problems, aiming to create more extensive and diverse datasets for robust research and development.

2.3.2 Multi-Task Solution Using Deep Learning

In addressing the complexities of predicting problem tags, our task can be considered as a multi-task problem that simultaneously addresses (1) multi-label classification and (2) ordinal-class classification. This approach is supported by previous research demonstrating the benefits of joint learning methods across various fields [37, 38].

By simultaneously training models on multiple tasks, we aim to streamline computational resources while maintaining or enhancing the classification accuracy compared to single-task models.

2.4 Solver Module

Mikyška et al. [11] highlights a study that investigates the potential of pretrained large language models (LLMs) in addressing programming questions, particularly those containing code snippets, from platforms like Stack Overflow. The paper explores how LLMs, when fine-tuned for the downstream task, can be optimized for closed-book question answering systems. Utilizing GPT-Neo, CodeGen, and PanGu-Coder, the researchers achieve a BLEU score of **0.4432** on two benchmark datasets: CoNaLa and a Stack Overflow-based collection.

The recent advances in NLP have underscored the importance of LLMs, which, through pretraining, can implicitly learn knowledge from text and perform tasks like open-domain QA without requiring external context or information. The success of Codex in program synthesis demonstrates the adaptability of LLMs from NLP to source code tasks. This trend is further supported by models like BERT+TAE [23], TranX+BERT [24], and MarianCG [25], which integrate pretrained LLMs for enhanced performance in code-related tasks.

While program synthesis traditionally involves generating code based on natural language queries, studies on programming-related QA sites, such as Stack Overflow, reveal that real-life programmer inquiries cover a broader spectrum. Researchers have classified Stack Overflow posts into categories like conceptual, API usage, and discrepancy. This observation led the authors to investigate the effectiveness of pretrained LLMs in addressing these diverse questions, particularly those involving code snippets.

PanGu-Coder2, by **Shet et al.** [26] explored multiple strategies to enhance Code LLMs, including data engineering, instruction tuning, retrieval-augmented generation, and reinforcement learning (RL). However, current RL methods face limitations, primarily due to their reliance on test results as rewards, complications in implementation, and the time-consuming nature of testing during model training.

Limitations of Existing RL Approaches in Code LLMs

Existing RL-based approaches for boosting Code LLMs have several key limitations. Directly using test results as a reward signal provides limited improvement to the baseline model. Additionally, implementing advanced RL algorithms, such as PPO, on large language models is challenging and computationally expensive. Moreover, incorporating testing during training elongates the process, leading previous studies to focus on smaller models with relatively modest improvements. These factors hinder the full exploitation of Code LLM's potential in code generation tasks.

A groundbreaking paper from Google's DeepMind, named **Competition-Level Code Generation with AlphaCode** by Yujia et al. [27] introduces the AlphaCode system, capable of generating innovative solutions to complex coding challenges. Key components of AlphaCode include a comprehensive and clean competitive programming dataset, large and efficient transformer-based models, and large-scale model sampling with subsequent filtering based on program behavior.

AlphaCode achieved an impressive performance by ranking in the top 54.3% among contest participants on Codeforces, the most popular online judge, as shown in **figure 2.1**.

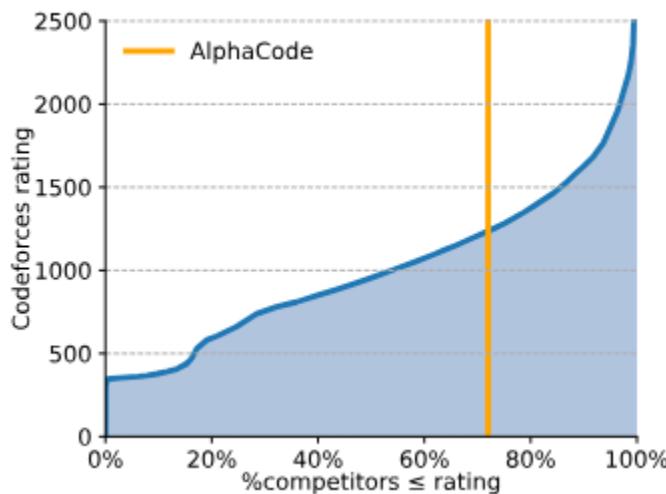


Figure 2.1: AlphaCode's impressive rating on Codeforces

AlphaCode Limitations

Despite the impressive performance of AlphaCode, there are several limitations highlighted in the study. One limitation is the undertraining of the largest 41B parameter model due to resource constraints, which affected its performance relative to models of other scales.

Another issue concerns the evaluation process, where variance was observed stemming from the trained model, sample selection, ordering, and clustering process. To mitigate some of this variance, the authors resampled and shuffled inputs but were constrained from retraining or resampling the models themselves. Furthermore, sensitivity to problem descriptions was tested by assessing the model's performance with modified descriptions; however, retraining on these changes was not feasible due to computational limitations. This suggests that while AlphaCode demonstrates some adaptability to changes in problem descriptions, it does not resort to brute-force solution attempts, emphasizing the importance of clear and accurate descriptions for optimal performance.

Chapter 3

System Architecture and Methods

3.1 System Architecture

Figure 3.1 visualizes the system architecture of the project. The system architecture is composed of three layers. The three-tier architecture is divided into Presentation, Logic and Data layers.

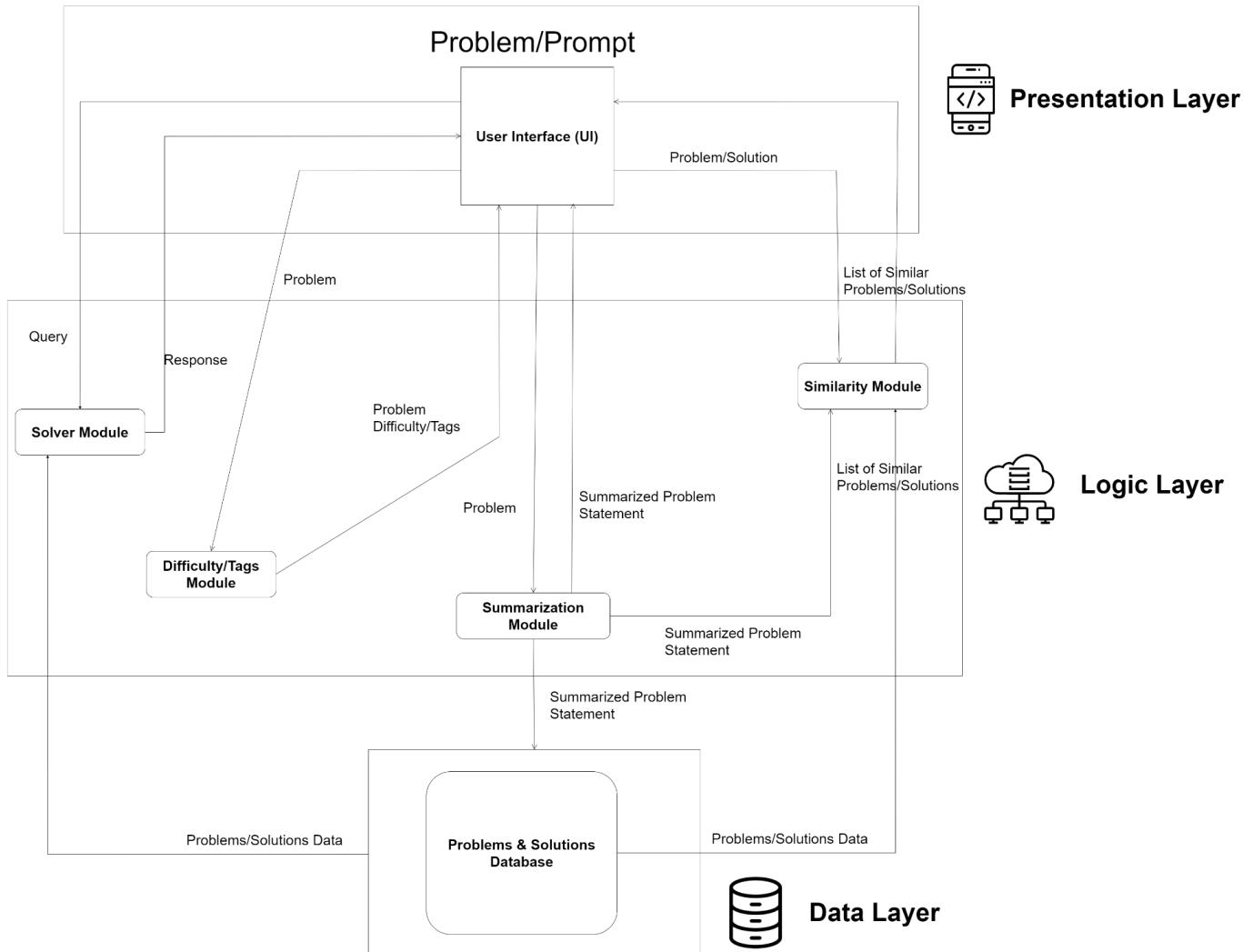


Figure 3.1: Three Tier System Architecture

- **Presentation Layer**

Users interact with the system by entering queries, problems, or solutions through the user interface (UI). The UI then communicates these inputs to various modules within the logic layer for processing. Specifically, the UI sends:

- Queries to the Solver Module for responses
- Problems to the Summarization Module for summaries
- Problems to the Difficulty/Tags Prediction Module to determine their difficulty levels and tags
- Problems or solutions to the Similarity Module to retrieve lists of similar problems or solutions

- **Logic Layer**

The logic layer handles the processing of information received from the presentation layer using four integrated modules:

- Problem Summarization Module
- Problem Statements/Solution Source Codes Similarity Module
- Difficulty/Tag Prediction Module
- Solver Module

Each Module has a specific role and collectively contributes to the overall functionality of the system.

- **Data Layer**

The data layer manages and stores all data related to problems and solutions. This database includes problem texts, problem inputs and outputs, summarized problem statements, difficulty levels, tags, solutions, time, memory, and other relevant metadata. It ensures that all necessary information is readily available for retrieval and processing by various modules in the logic layer, such as the Summarization Module, Difficulty/Tags Module, and Similarity Module.

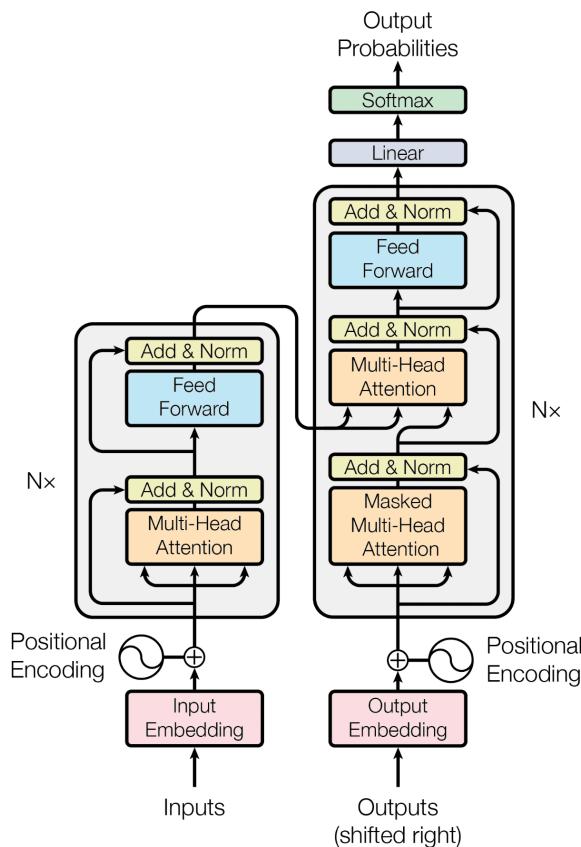
3.2 Description of Methods and Procedures

3.2.1 Problem Summarization Module

The primary objective of this module is to strip off all the stories, legends, characters, and backgrounds from the statement while enabling everyone to understand the problem. Two approaches were explored; classical pre-LLM methods, and prompt engineering an LLM.

Pre-LLM era

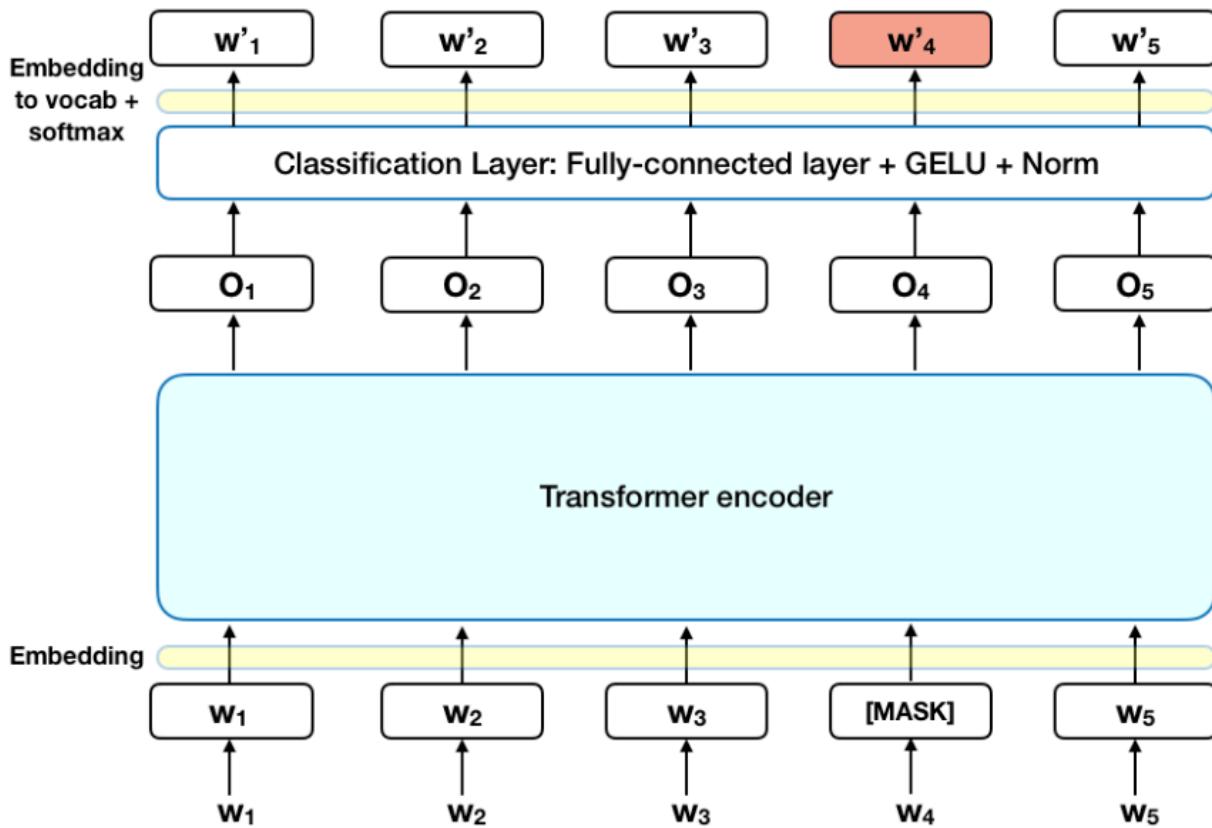
Prior to the historic rise of LLMs, architectures like BERT, and its family, which is based on the transformer architecture shown in [figure 3.2](#), were the standard for many NLP tasks.



[Figure 3.2: Transformer's Architecture](#)

BERT is an “encoder-only” transformer architecture. At a high level, BERT consists of three modules, as shown in [figure 3.3](#).

- **Embedding:** This module converts an array of one-hot encoded tokens into an array of real-valued vectors representing the tokens. It represents the conversion of discrete token types into a lower-dimensional Euclidean space.
- **Encoder stack:** A sequence of Transformer encoder blocks. They perform transformations over the array of representation vectors, one of which is bi-directional self-attention.
- **Un-embedding/decoder:** This module converts the final representation vectors into one-hot encoded tokens again by producing a predicted probability distribution over the token types. It can be viewed as a simple decoder, decoding the latent representation into token types.



[Figure 3.3: BERT’s Architecture](#)

The decoder module is necessary for pre-training, but it is often unnecessary for so-called “downstream tasks” such as text summarization or sentiment classification. Instead, one removes the decoder module and replaces it with a newly initialized module suited for the task. The latent vector representation of the model is directly fed into this new module, allowing for sample-efficient transfer learning.

Architectures used:

- BERT
- BART
- T5
- PEGASUS

Prompt Engineering an LLM

LLMs have shown spectacular performance on downstream tasks (like semantic analysis, summarization, etc...) even if it wasn't specifically trained on.

This method uses LLMs and prompt engineering, writing detailed tailored prompts that instructs the LLM in a clear way to strip all the legends, background stories, etc..., resulting in a concise and clear statement.

LLMs used:

- Gemma-7b
- Mixtral-8x7b
- Gemini 1.5 flash
- Vicuna-33b

3.2.2 Problem/Solution Similarity Module

The primary objective of this module is to identify similar problems or solutions to a given query. Two approaches were explored: the Clustering Approach and the Nearest Vectors Approach.

- **Clustering Approach**

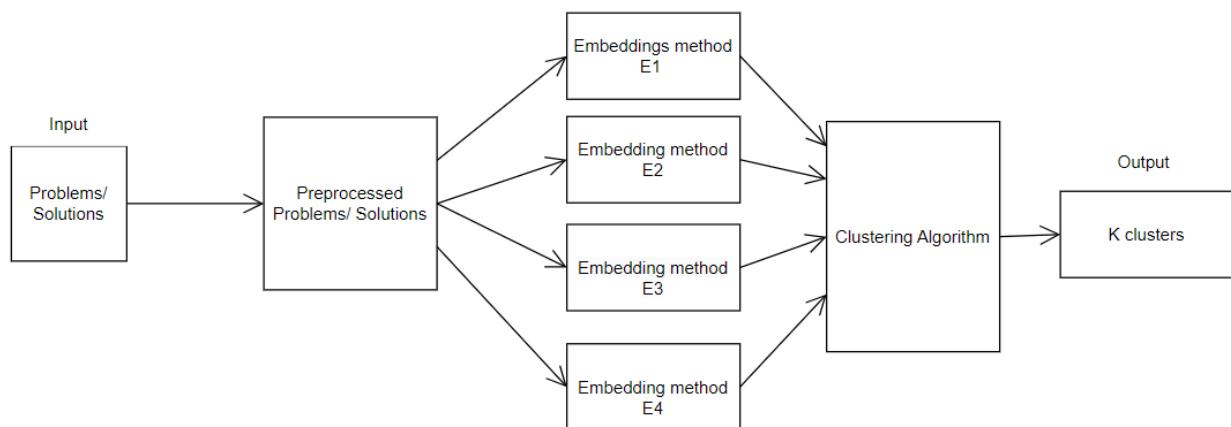
The clustering approach used is illustrated in [figure 3.4](#). This method involves grouping similar problems/solutions into clusters, with each cluster representing a set of similar problems/solutions. Various clustering algorithms were applied to each embedding technique to partition the source codes into K clusters.

Embeddings Used for Problems:

- all-mpnet-base-v2
- bge-small-en-v1.5
- all-MiniLM-L6-v2

Embeddings Used for Solutions:

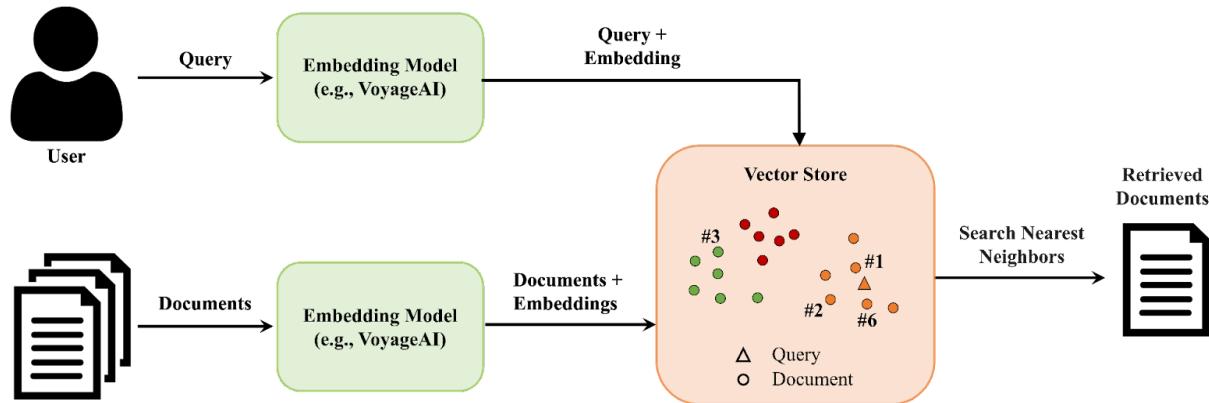
- voyage-code-2
- all-mpnet-base-v2
- word2vec
- doc2vec
- tf-idf



[Figure 3.4: Clustering Approach used](#)

- **Nearest Vectors Approach**

The Nearest Vectors approach used is illustrated in [figure 3.5](#). In this method, each problem/solution is represented as a vector in a high-dimensional space. Similarity is determined by calculating the distance between vectors. The nearest vectors to the query vector are identified as the most similar problems/solutions.



[Figure 3.5: How Nearest-Vector approach works](#)

Embeddings Used for Problems:

- all-mpnet-base-v2
- bge-small-en-v1.5
- all-MiniLM-L6-v2

Embeddings Used for Solutions:

- voyage-code-2
- all-mpnet-base-v2

Attempts were made to use the following models for experimentation, but memory constraints prevented their use:

- jina-embeddings-v2-small-en: This model consists of 33 million parameters.
- jina-embeddings-v2-base-en: This model consists of 137 million parameters.

3.2.3 Difficulty/Tags Prediction Module

The objective of this module is to predict the tags and difficulty level of a given problem using only its statement. This serves as a valuable guide for solving algorithm problems. [Figure 3.6](#) demonstrates how a problem statement is processed by a feature extractor (language model), which then feeds into a multi-label classifier and difficulty predictor to generate the relevant tags and difficulty level.

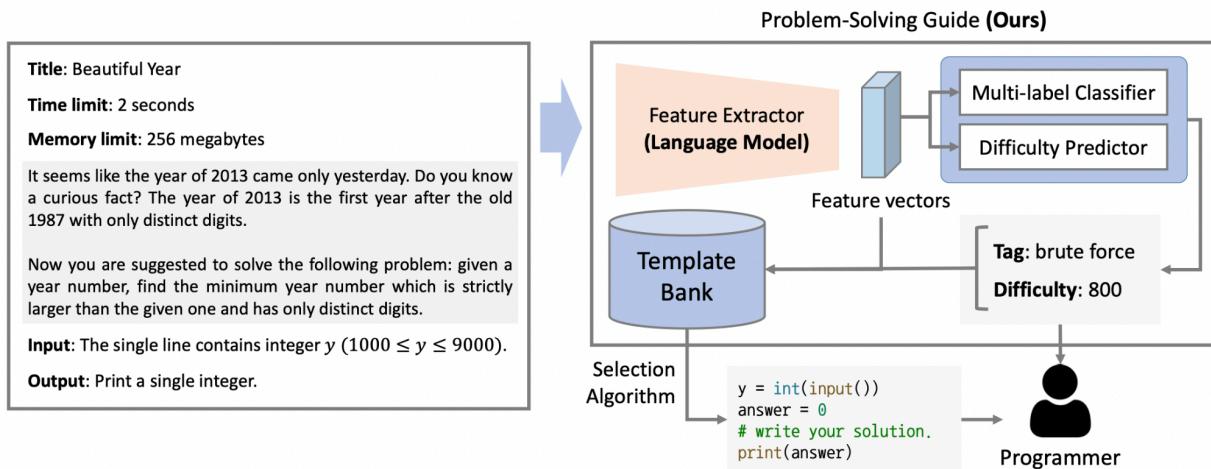


Figure 3.6: Predicting the Difficulty and Tags simultaneously

We consider our task as a multi-task problem that simultaneously addresses:

- Multi-label Classification
- Ordinal-class Classification

Some related studies have presented joint learning methods for simultaneously training multiple tasks in various research fields. The multi-task learning approach can reduce memory complexity while nearly maintaining the original classification performance of individual single-task models. As illustrated in [equation \(3.1\)](#), the loss function used in our multi-task learning framework has two components: the first term L_1 denotes a binary cross-entropy loss for the problem category, while the second term L_2 is designed for the ordinal classification task, representing the problem difficulty level. The parameter λ is a scaling factor for weighting the two tasks differently.

$$\mathbb{E}_{(x,y,d) \in \mathcal{D}_{train}} [l_1(H_1(z), y)) + \lambda \cdot l_2(H_2(z), d))]$$

(3.1)

This module leverages the BigBird BERT-based architecture, which is a variation of the traditional BERT architecture. BERT has been one of the most successful deep learning models for natural language processing (NLP). **Figure 3.7** displays the network structure of the BERT model, highlighting the token embeddings, segment embeddings, and position embeddings that are combined and fed into the transformer layers. Each output from the transformer layers is then used for different tasks (C, T1, T2, etc.), showcasing the multi-task learning approach.

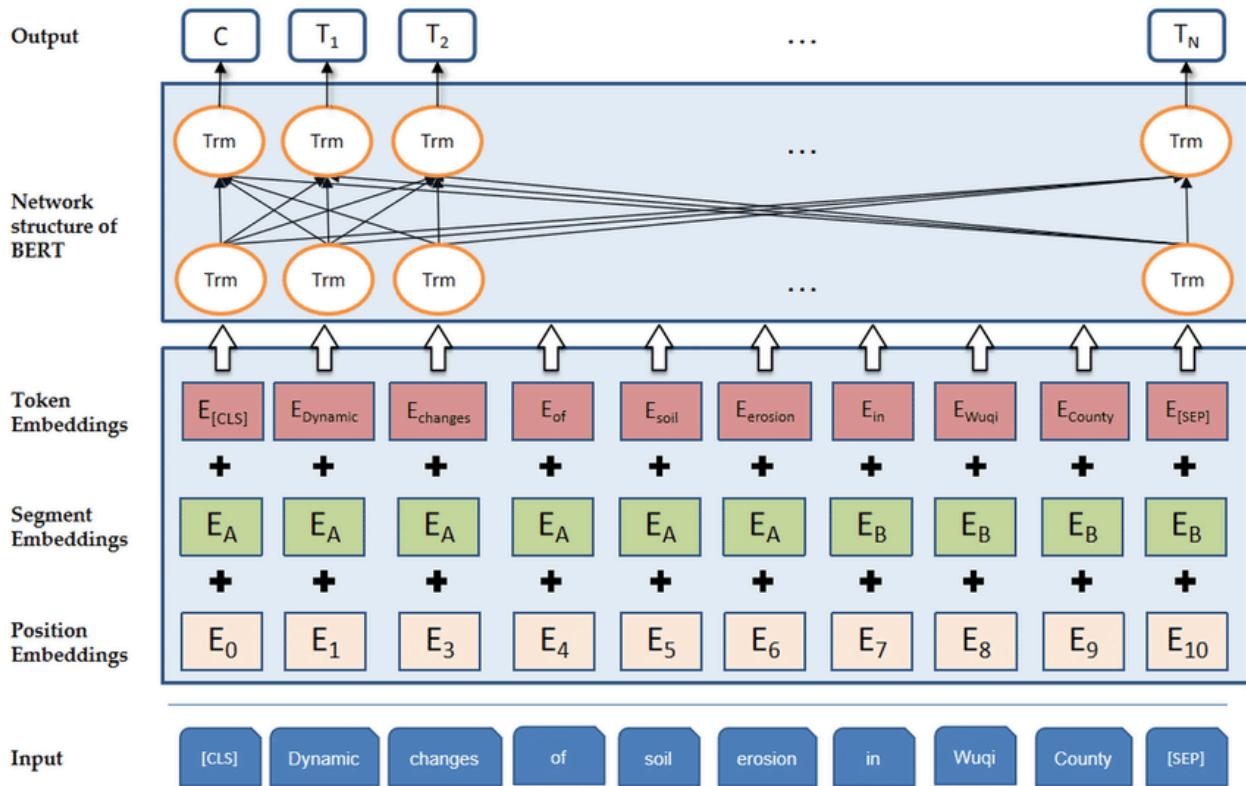


Figure 3.7: Network Structure of a BERT Model

However, a core limitation of BERT is its quadratic dependency on sequence length, primarily due to its full attention mechanism, which results in high memory consumption. To address this limitation, BigBird was proposed. As shown in [figure 3.8](#), BigBird employs a sparse attention mechanism that reduces the quadratic dependency to linear, significantly enhancing efficiency and scalability for longer sequences. This innovation allows for more efficient processing of lengthy problem statements, which is crucial for our module's functionality.

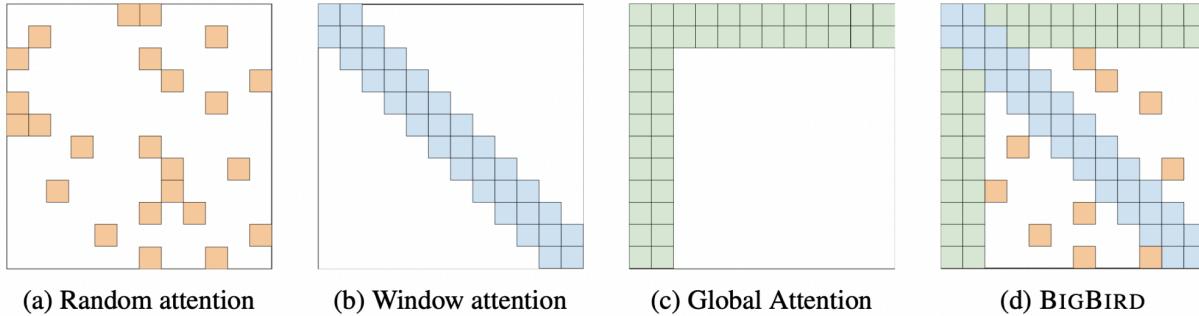


Figure 3.8: Difference between BigBird and other attention mechanisms

3.2.4 Solver Module

The solver module is built on a Retrieval-Augmented Generation (RAG) architecture with CodeQwen Large Language Model (LLM) capabilities to function as an interactive chatbot providing accurate responses to diverse questions, effectively adapting to evolving data environments.

The RAG architecture, shown in [figure 3.9](#), is a method designed to enhance the knowledge and context of LLMs by augmenting them with external data. This technique is crucial for AI applications that require reasoning about private data or recent information not covered in the LLM's training dataset.

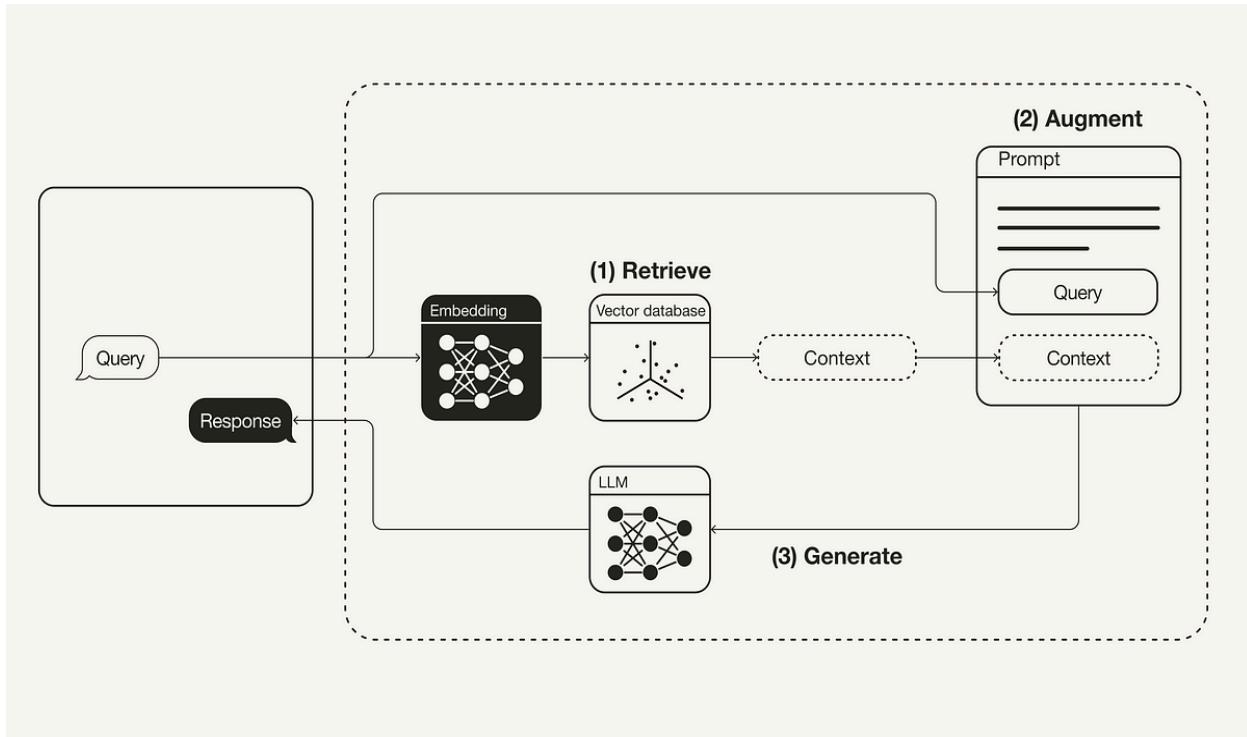


Figure 3.9: RAG architecture explaining how it works

A typical RAG application for unstructured data involves two main components: indexing and retrieval-generation.

- **Indexing**

The indexing component is responsible for preparing and storing the data in a format that enables efficient retrieval. This process typically occurs offline and includes the following steps:

- **Load:** Utilizing Document Loaders, the first step is to load the data into the system. This involves importing documents from various sources, such as PDFs, web pages, or text files.
- **Split:** Text Splitters are then used to break down large documents into smaller, manageable chunks. This is essential for indexing as well as for passing the data into the model, considering the constraints of the model's context window.
- **Store:** The smaller text chunks are indexed and stored using a VectorStore, which often requires the use of an Embeddings model to convert the text into numerical vectors that can be compared and searched efficiently.

- **Retrieval and Generation**

The retrieval and generation component executes at runtime in response to user queries. The process involves:

- **Retrieve:** Given a user query, a Retriever searches the stored index to find the most relevant text chunks. This retrieval process is crucial for ensuring that the LLM has access to the precise information needed to answer the query accurately.
- **Generate:** Once the relevant data is retrieved, a ChatModel or LLM uses a prompt that includes the user's question and the retrieved data to generate a comprehensive and contextually accurate response. This response leverages the combined knowledge of the LLM and the retrieved data to provide detailed and up-to-date information.

The RAG architecture for unstructured data significantly enhances the capability of LLMs to provide accurate and context-specific responses. By integrating the retrieval of pertinent information with the generative power of LLMs, RAG systems can deliver high-quality answers to a wide range of questions, bridging the gap between the static knowledge of LLMs and the dynamic, evolving nature of data.

For our project, we have adopted a LLM architecture akin to that of Qwen, a state-of-the-art language model designed for code generation and understanding. This architecture is a variant of the transformer-based decoder-only model, specifically tailored to address the challenges and requirements of code-related tasks. Below, we delve into the distinctive features of our model architecture that differentiate it from a standard transformer model, shown in [figure 3.10](#).

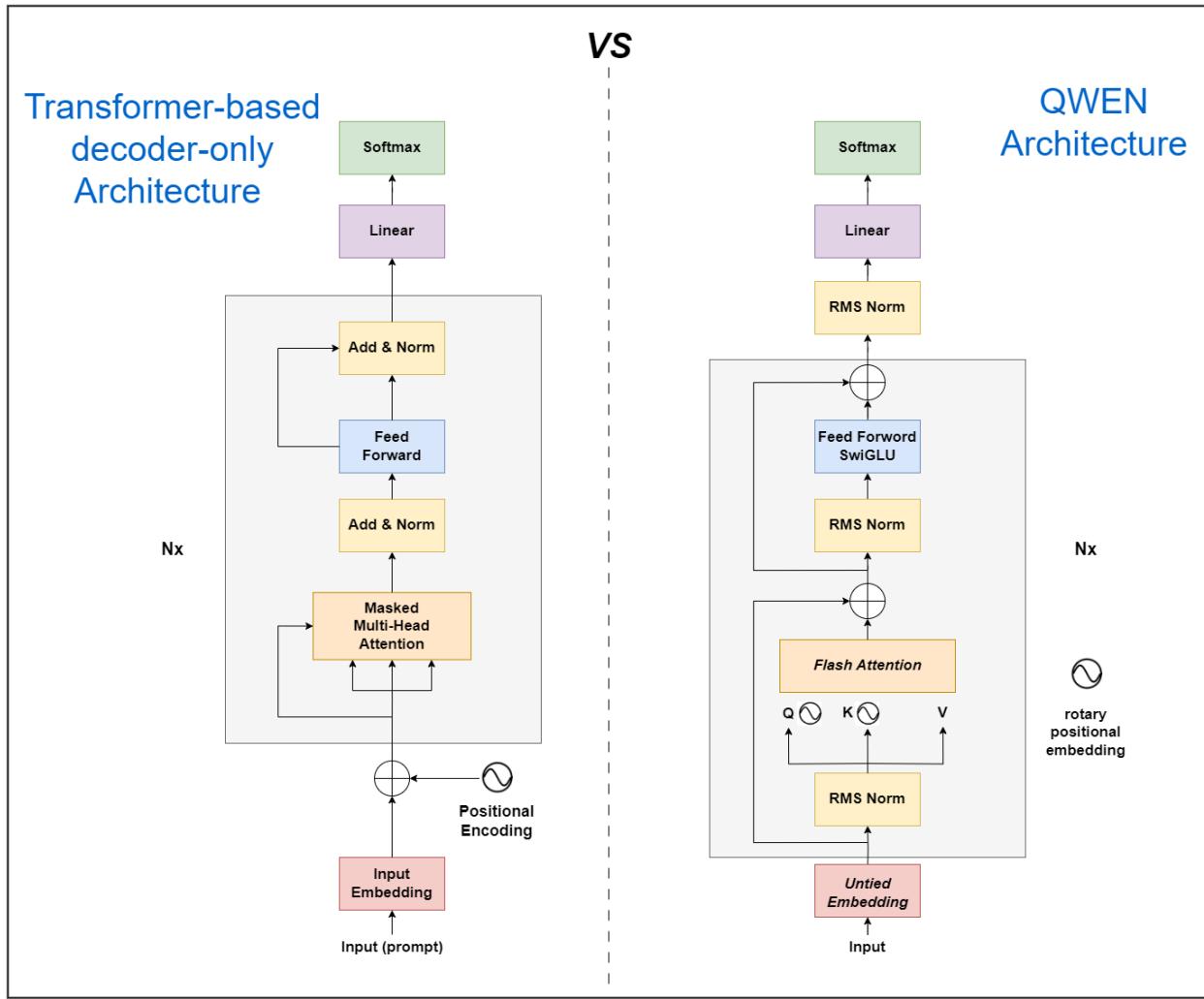


Figure 3.10: Difference between Qwen and a Decoder-only architecture

Key Architectural Deviations from Standard Transformer

- **Un-tied Embeddings:** Unlike the standard transformer model, which typically employs tied input and output embeddings to reduce the number of parameters and improve generalization, our model uses un-tied embeddings. This approach allows the model to learn independent representations for inputs and outputs, potentially enhancing its capacity to generate more diverse and contextually relevant outputs.
- **Rotary Positional Embeddings:** We incorporate rotary positional embeddings as a means to improve the model's understanding of the positional context in sequences. This technique, instead of using learned position embeddings, uses a fixed, predefined function to encode absolute positions in the sequence. This method is particularly beneficial for handling

sequences of variable lengths and for improving the model's performance on longer sequences.

- **Bias Minimization:** The model excludes biases in all layers except for the Q (query), K (key), and V (value) matrices in the attention mechanism. This design choice is aimed at reducing the number of parameters and computational overhead while preserving critical model functionality.
- **RMSNorm Instead of LayerNorm:** RMSNorm (Root Mean Square Layer Normalization) is adopted in place of the more conventional LayerNorm. RMSNorm is known for its computational efficiency and numerical stability, which are particularly advantageous during the training phase of deep models like Qwen.
- **SwiGLU Activation Function:** We utilize SwiGLU (a combination of the sigmoid function and GLU, Gated Linear Unit) over ReLU as the activation function in our model's feed-forward layers. SwiGLU is designed to improve the model's capacity to capture complex patterns and dependencies in data by introducing non-linearities and gating mechanisms.
- **Flash Attention Mechanism:** To accelerate the training process and reduce computational costs, our model implements flash attention. This is a technique for efficiently computing the attention mechanism by leveraging sparsity and other optimizations, making it particularly effective for large-scale models like Qwen.

Model Specifications

The model is composed of 32 layers, which is considered deep for transformer models, providing a higher capacity for learning intricate patterns in the data.

The embedding dimension is set to 4096, which is substantially larger than that of many standard models, allowing for a richer representation of input sequences.

Utilizing 32 attention heads, the model can simultaneously attend to different parts of the input sequence, enhancing its ability to handle complex and multi-faceted relationships within the data.

Chapter 4

System Implementation and Results

4.1 Dataset

We collected a dataset consisting of problems, as shown in [table 4.1](#), and their corresponding solutions, as shown in [table 4.2](#). The dataset used in our research was scraped from various online judges and GitHub repositories, resulting in a comprehensive collection of problems and their corresponding solutions. The total dataset comprises **19,503** problems and **1,700,412** solutions. During the web scraping process, we encountered rate limit constraints which posed challenges in data collection.

Online Judge	AtCoder	CodeChef	CSES	Codeforces	Hacker Earth	LeetCode	UVa	Yosupo
Number of Problems	2,307	3,201	296	8,636	985	2,424	1,461	193
Online Judge	AtCoder	CodeChef	CSES	Codeforces	Hacker Earth	LeetCode	UVa	Yosupo

Table 4.1: Problems Dataset Details

Online Judge	AtCoder	CodeChef	CSES	Codeforces	Hacker Earth	LeetCode	UVa	Yosupo
Number of Solutions	1,587,756	53,961	1,175	16,229	36,179	3,570	1,347	195
Online Judge	AtCoder	CodeChef	CSES	Codeforces	Hacker Earth	LeetCode	UVa	Yosupo

Table 4.2: Solutions Dataset Details

The dataset is divided into two primary datasets: Problems and Solutions.

Problems Dataset:

- **Problem Link:** URL to the problem statement.
- **Problem Text:** The full text of the problem statement.
- **Sample Input:** Example input provided in the problem.
- **Sample Output:** Example output expected from the problem.

- **Time Limit (s)**: Maximum allowed time for the solution to run.
- **Memory Limit (mb)**: Maximum allowed memory for the solution.
- **Tags**: Relevant tags categorizing the problem.
- **Difficulty**: Difficulty level of the problem.

Solutions Dataset:

- **Problem Link**: URL to the problem statement to which the solution corresponds.
- **Submission Link**: URL to the solution submission.
- **Problem Name**: Name of the problem.
- **Source**: Origin of the solution (e.g., online judge or GitHub).
- **Language**: Programming language used in the solution (all solutions are in C++).
- **Time**: Execution time of the solution.
- **Memory**: Memory consumption of the solution.
- **Solution**: The actual code of the solution.

This structured approach ensures that each problem and its solutions are systematically categorized, facilitating easy access and processing for training and evaluation of our models. The inclusion of various metadata such as time limit, memory limit, tags, and difficulty levels provides valuable context and aids in the comprehensive analysis and modeling of the dataset.

Preprocessing

To ensure the modules focus on the functional aspects of the source code and problem statements, we applied the following preprocessing steps:

- **For Solutions:**
 1. Remove #include Directives, using namespace, and Comments:
 - We used regular expressions to remove all `#include` directives, `using namespace` declarations, and comments from the source code.

2. Remove All Non-ASCII Characters:
 - This was done by encoding the source code into ASCII bytes, ignoring any non-ASCII characters, and then decoding those bytes back into a regular string.
3. Remove Unused Functions:
 - We used the `cpp-check` tool, a static code analysis tool for C/C++ code, to identify and remove unused functions. `cpp-check` generates XML output containing information about potential issues in the code, including unused functions.
 - The XML file generated by `cpp-check` was parsed using the `xml.etree.ElementTree` library. The identified lines containing unused functions were removed from the code.
4. Tokenize the Source Code:
 - The tokenization was done using the `CppLexer` class from the `pygments` library, which tokenizes the source code into a list of tokens.
5. Attempts to Replace All Macro Directives:
 - We attempted to use the `g++` compiler to replace all macro directives and generate assembly code. However, this approach led to an excessive number of tokens, exceeding the limitations of existing embedding models. Therefore, it was not used in the final preprocessing pipeline.

- **For Problems:**

1. Substitute Exponential Notation:
 - Convert exponential notation (e.g., 10^6) to its numeric equivalent (1000000).
2. Insert Spaces Between Dollar Signs:
 - Modify instances like `$$$$n$$$$` to `$$$ n $$$`.
3. Convert to Lowercase:
 - Convert all text to lowercase for uniformity (e.g., "This is an interactive task." to "this is an interactive task.").

4. Calculate Expressions:
 - Simplify mathematical expressions (e.g., $2 \cdot 100000$ to 200000).
5. Remove Stopwords:
 - Remove common stopwords using the `nltk` library to reduce noise in the text.
6. Lemmatize:
 - Perform lemmatization using the `nltk` library to reduce words to their base or root form.

These preprocessing steps ensure that the data fed into our models is clean, consistent, and focused on the functional and relevant aspects of the problems and solutions, enhancing the performance and accuracy of our modules.

4.2 Description of Software Tools Used

- **PyTorch**

PyTorch, accessed through the `torch` package, is a pivotal library for deep learning tasks, providing flexible and powerful computational frameworks. It supports dynamic computation graphs, GPU acceleration, and a comprehensive set of mathematical operations, making it ideal for training and deploying LLMs and handling complex embeddings. PyTorch's dynamic nature and extensive ecosystem of extensions and plugins enhance its utility in our project, ensuring optimal performance in complex model computations.

- **Transformers**

Transformers library from Hugging Face is an essential toolkit for natural language processing, offering state-of-the-art models pre-trained on diverse datasets. These models, easily accessible through this library, significantly expedite the process of integrating LLMs into our applications. By leveraging the transformers library, we can perform advanced NLP tasks, such as text generation, classification, and translation, without the need for extensive model training from scratch, saving both time and computational resources.

- **Langchain and Langchain Community**

Langchain and langchain_community, are comprehensive toolkits for working with LLMs. Langchain simplifies the integration of LLMs into various applications, providing useful connectors and utilities for seamless interaction with different types of data. The Langchain Community package adds supplementary functionalities, such as the ability to load and process CSV datasets, enhancing the data handling capabilities of our applications, and ensuring smooth data flow between our models and external data sources.

- **Sentence Transformers**

Sentence Transformers, a library derived from BERT[15] and similar models, specializes in generating sentence embeddings. This library extends the capabilities of traditional transformers by enabling the generation of embeddings that encapsulate the semantic meaning of sentences. It is particularly useful in our RAG framework for improving the accuracy of vector store operations and similarity searches, allowing for more precise and contextually relevant information retrieval.

- **ChromaDB**

ChromaDB, a high-performance vector database, is crucial for storing and retrieving embeddings for LLMs. It efficiently manages large-scale vector data, enabling quick and accurate retrieval of information stored in our RAG framework. ChromaDB's scalable architecture and optimized search algorithms ensure that our models can access the most relevant information swiftly, enhancing the overall performance and responsiveness of our applications.

- **Numpy**

Numpy, a fundamental library for numerical computing in Python, offers support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. It facilitates the efficient manipulation of numerical data, which is essential for preprocessing data for use in machine learning models and for post-processing predictions to extract meaningful insights.

- **Pandas**

Pandas, is a data manipulation and analysis library providing data structures and operations for manipulating numerical tables and time series. It is particularly useful for handling structured data, such as CSV files, and for data munging and preparation. Pandas simplifies the process of loading, manipulating, and analyzing data, making it an indispensable tool in our data pipeline.

- **Matplotlib**

Matplotlib, is a plotting library that allows for the creation of static, interactive, and animated visualizations in Python. It is essential for visualizing data and model outputs, enabling us to understand complex patterns and trends that may not be apparent from the raw data. Matplotlib's flexibility and broad range of customization options make it a versatile tool for data visualization.

- **Nltk**

Nltk (Natural Language Toolkit), library provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for tokenization, parsing, classification, stemming, tagging, and semantic reasoning. It is indispensable for performing basic and advanced NLP tasks in our applications.

- **Requests**

Requests, is an elegant and simple HTTP library for Python, built for human beings. It is used for making HTTP requests to web APIs and web services, simplifying the process of fetching data from the internet. Its user-friendly API and robustness make it a standard choice for web data retrieval.

- **Faiss-cpu**

Faiss-cpu is a library for efficient similarity search and clustering of dense vectors. It is particularly useful for large-scale vector search tasks in our RAG framework, providing fast and accurate retrieval of similar items from a large dataset.

- **Voyageai**

Voyageai, is an AI platform that provides tools and services for building and deploying AI models. It is used for enhancing our applications with advanced AI capabilities, such as text generation and analysis.

- **google.generativeai**

Google.generativeai, is a set of APIs and services provided by Google for generating human-like text and content. It is integrated into our applications to enhance text generation capabilities, offering powerful models for creative and conversational text generation.

- **Kaggle:**

While not directly involved in the code generation process, Kaggle was used as a platform for accessing and preprocessing the dataset that constituted our knowledge base for RAG. It provided the infrastructure for data management and enhanced the accessibility of the data used in our project.

- **CUDA (from cuda):**

Used to leverage GPU acceleration for the computations required by our LLM and embeddings, crucial for performance optimization.

- **Gradio:**

A library for creating interactive user interfaces for machine learning models. Gradio was used to develop a user-friendly interface for interacting with our code generation model, allowing for real-time demonstrations and testing.

4.3 Experimental and Results

4.3.1 Problem Summarization Module

Evaluation Metric

Several LLMs were prompt engineered to instruct them specifically how to summarize the problem statement. We focused on low-cost or open-weight LLMs as a proof-of-concept, but the performance does scale with the LLM size (i.e. the number of parameters).

Several metrics were used to assess the underlying LLM. The metrics used include:

- **General Ability (Chatbot Arena):** This metric measures the overall performance and effectiveness of the model in a general conversational context.
- **Reasoning & Knowledge (MMLU):** This metric evaluates the model's ability to reason and utilize knowledge, as tested by the MMLU (Massive Multitask Language Understanding) benchmark.
- **Reasoning & Knowledge (MT Bench):** This metric assesses the model's reasoning and knowledge capabilities using the MT Bench.

The performance of each model across these metrics is summarized in [table 4.3](#).

Model	General Ability (Chatbot Arena) [39]	Reasoning & Knowledge (MMLU)	Reasoning & Knowledge (MT Bench)
Gemini 1.5 Flash [28]	1231	79%	NaN
Mixtral 8x7B [29]	1114	71%	8.3
Gemma-7b [30]	1037	64%	NaN
Vicuna-33b [31]	NaN	NaN	NaN

Table 4.3: Performance of different LLMs on different benchmarks

The results are clear that **Gemini 1.5 flash** is best to its size.

4.3.2 Problem/Solution Similarity Module

Evaluation Metric

1. Silhouette Score:

The Silhouette score is used to evaluate the quality of the clusters formed. It measures how similar an object is to its own cluster compared to other clusters. A high Silhouette score indicates that the objects are well-matched to their own cluster and poorly matched to neighboring clusters.

Silhouette scores for problem statements are shown in [table 4.4](#), and for solutions are shown in [table 4.5](#).

Embeddings Model	Clustering Algorithm	Number of Clusters	Silhouette Score
sentence-transformers/all-mpnet-base-v2	K-means	30	0.14809753
	Agglomerative		0.12610750
	Spectral Clustering		0.10872551
	DBscan	-	-0.10904756
sentence-transformers/all-MiniLM-L6-v2	K-means	30	0.13519753
	Agglomerative		0.18610651
	Spectral Clustering		0.10872551
	DBscan	-	-0.18904756

Table 4.4: Silhouette scores for different clustering algorithms with different embeddings for problem statements

Embeddings Model	Clustering Algorithm	Number of Clusters	Silhouette Score
TF-IDF	K-means	23	0.52416289
	Agglomerative		0.50316228
	Spectral Clustering		0.40467228
	DBscan	-	0.42417239

Voyage-Code-2	K-means	23	0.4416289
	Agglomerative		0.4674223
	Spectral Clustering		0.4126262
	DBscan		0.3114283
sentence-transformers/all-mnlpnet-base-v2	K-means	23	0.13256446
	Agglomerative		0.113510504
	Spectral Clustering		-0.040572744
	DBscan		-0.06446284

Table 4.5: Silhouette scores for different clustering algorithms with different embeddings for solutions

The *Agglomerative* clustering along with *all-MiniLM-L6-v2* embedding achieved the highest score for problem statements.

The *K-means* clustering along with *TF-IDF* embedding achieved the highest score for solutions.

2. Manual Assessment:

We analyzed 50 queries for both problem statements and solution source codes to manually assess the effectiveness of the similarity module.

Results for the manual assessment for problem statements are shown as shown in **figures 4.1 and 4.2**.

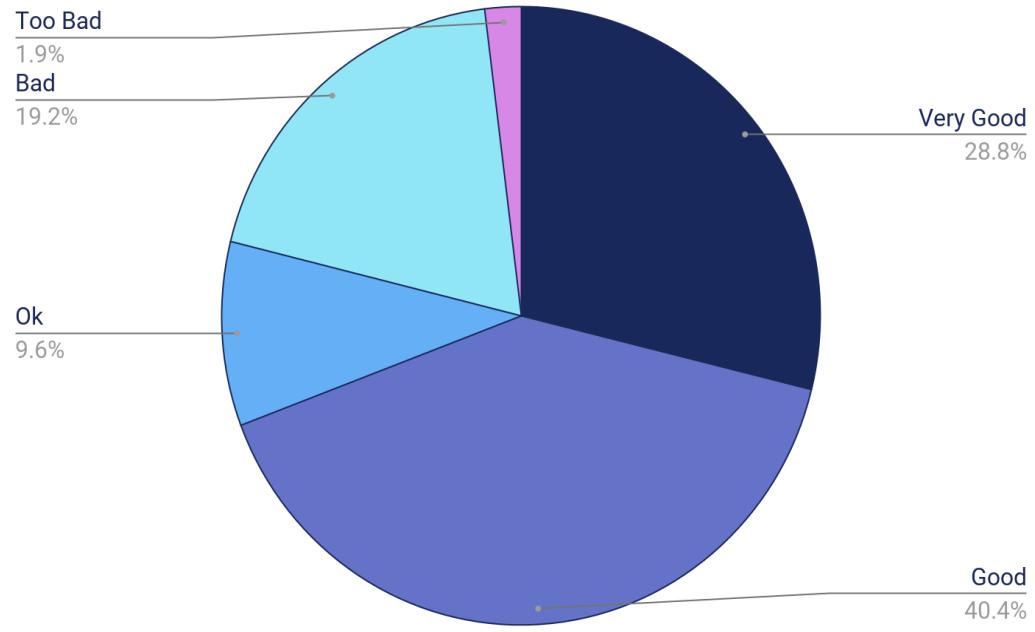


Figure 4.1: Manual Assessment results for problems similarity using nearest vector approach

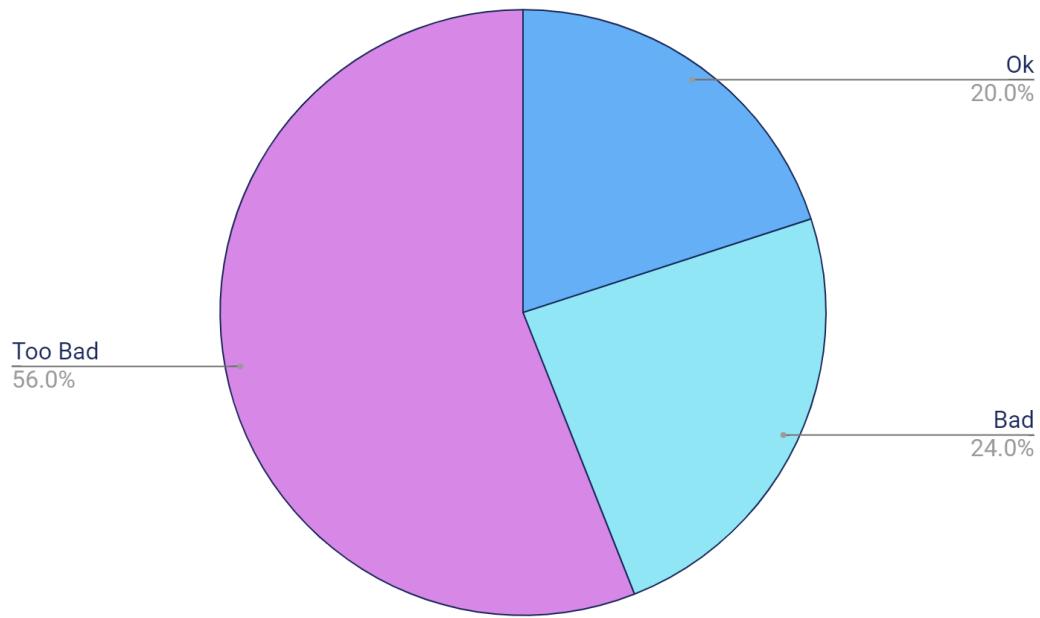


Figure 4.2: Manual Assessment results for problems similarity using clustering approach

Results for the manual assessment for solutions source codes are shown as shown in **figures 4.3 and 4.4**.

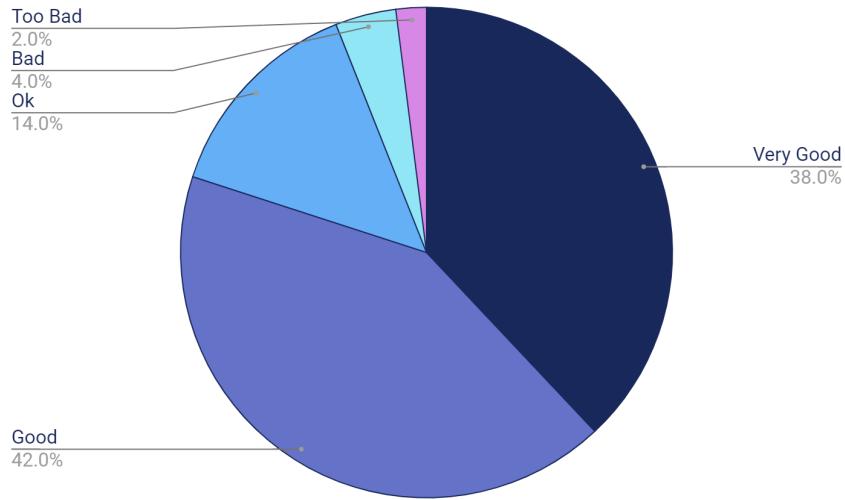


Figure 4.3: Manual Assessment results for solutions similarity using nearest vector approach

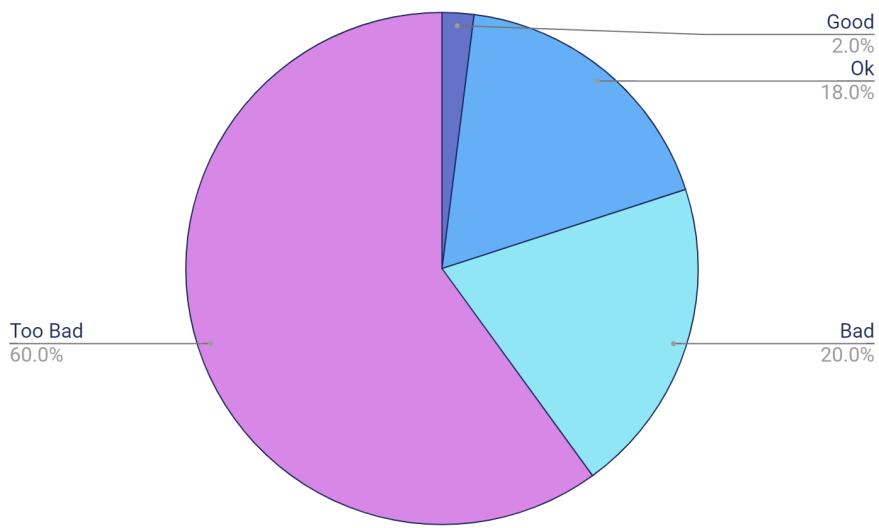


Figure 4.4: Manual Assessment results for solutions similarity using clustering approach

It is clear that the *nearest-vector approach* is better for both problems and solutions similarity, and by a wide margin.

3. Recall@k:

This metric measures the hit rate of retrieving the gold standard document among the top k retrieved documents for k = 1, 3, 5, 10.

- **Problem Statements:** A hit is when, for a summarized problem statement, the original problem is retrieved. Results are shown in [table 4.6](#).

Embedding Model	all-mpnet-b ase-v2	all-MiniLM- L6-v2	bge-small-e n-v1.5	Embedding Model	all-mpnet-b ase-v2	all-MiniLM- L6-v2	bge-small-e n-v1.5
recall@1	74%	55.5%	45.1%	recall@1	90.5%	82.5%	75.1%
recall@3	74.4%	60.7%	50.5%	recall@3	94.5%	88.5%	80.5%
recall@5	80.3%	62%	52.3%	recall@5	96%	90%	85.3%
recall@10	81%	65.3%	55.4%	recall@10	97.5%	93.5%	90.2%

Clustering
Nearest Vector

Table 4.6: recall@k results for problem statements

- **Solutions Source Codes:** A hit is when, for a solution to a particular problem, another solution for that same problem exists in the retrieved solutions. Results are shown in [table 4.7](#).

Embedding Model	voyage-code -2	all-mpnet- base-v2	tf-idf	Embedding Model	voyage-code -2	all-mpnet- base-v2	tf-idf
recall@1	94%	74%	65.1%	recall@1	99%	82.5%	75.1%
recall@3	94.3%	74.5%	69.5%	recall@3	99.2%	88.5%	80.5%
recall@5	94.4%	80.1%	70.3%	recall@5	99.2%	90%	85.3%
recall@10	97%	80.3%	75.2%	recall@10	99.6%	93.5%	90.2%

Clustering
Nearest Vector

Table 4.7: recall@k results for solutions source codes

The *nearest-vector approach* achieved state-of-the-art performance for both problems and solutions similarity, and by a wide margin, using *all-mpnet-base-v2* embedding for problems, and *Voyage-Code-2* embedding for solutions.

4.3.3 Difficulty/Tags Prediction Module

Evaluation Metric

1. ROC Curve and AUC

An ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds, as shown in [figure 4.5](#).

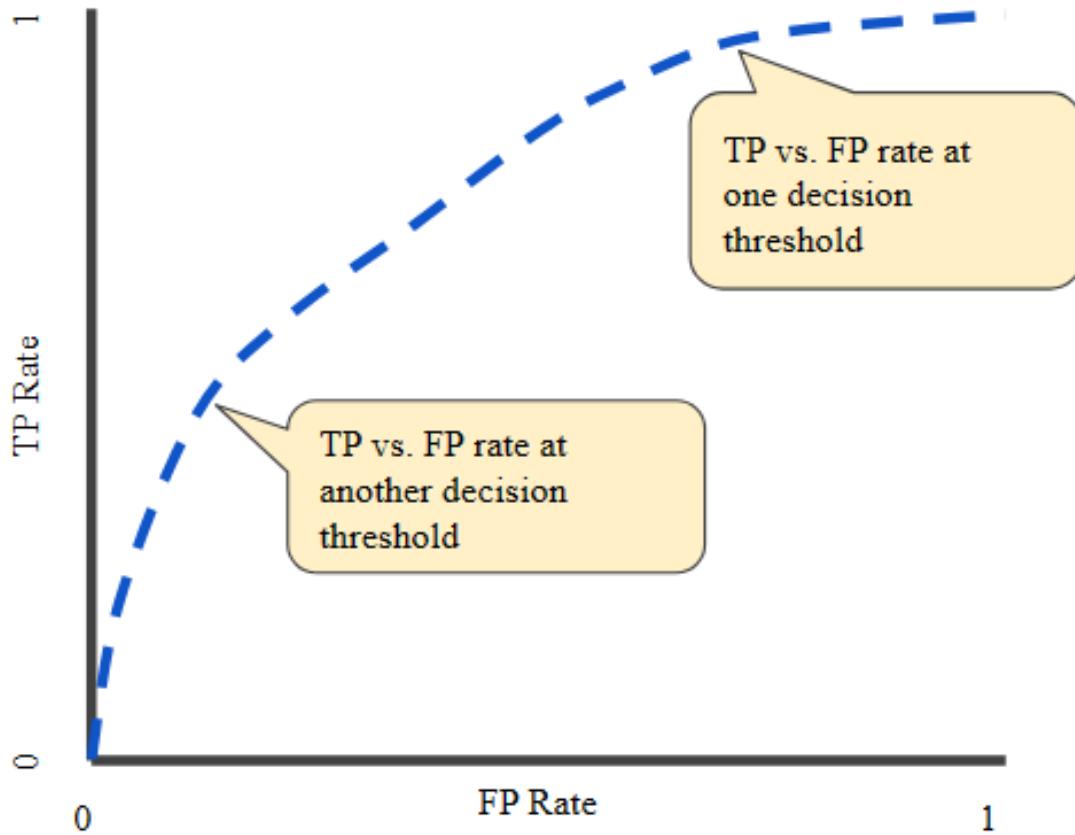


Figure 4.5: ROC curve

The ROC curve plots two parameters:

- **TPR (True Positive Rate):** Also known as recall, it shows the share of detected true positives. For example, the share of emails correctly labeled as spam out of all spam emails in the dataset.
- **FPR (False Positive Rate):** This shows the share of objects falsely assigned a positive class out of all objects of the negative class. For example, the proportion of legitimate emails falsely labeled as spam.

The ROC Curves of Various Model Variations

As shown in [figure 4.6](#), the ROC curves for different model variations are depicted:

- **Top-left graph:** ROC curve of our PSG model trained with $\lambda=1$.
- **Top-right graph:** ROC curve of our PSG model trained with $\lambda=10$.
- **Bottom-left graph:** ROC curve of our PSG model trained with $\lambda=100$.
- **Bottom-right graph:** ROC curve of the single-task BigBird model used to solve only the second task T2.

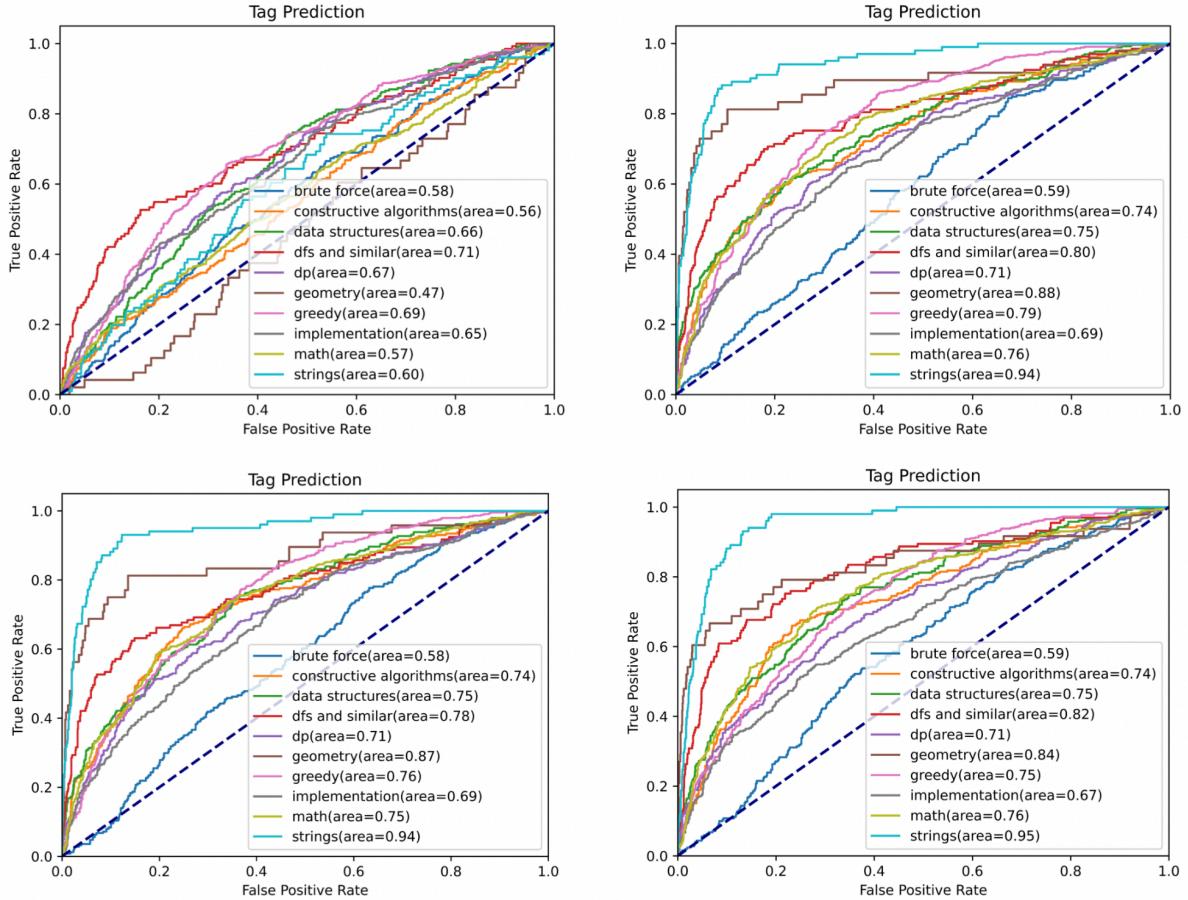


Figure 4.6: ROC curves of various models

Model Fine-tuning

Figure 4.7 shows the ROC curve of the model trained with $\lambda = 10$ before vs after fine-tuning.

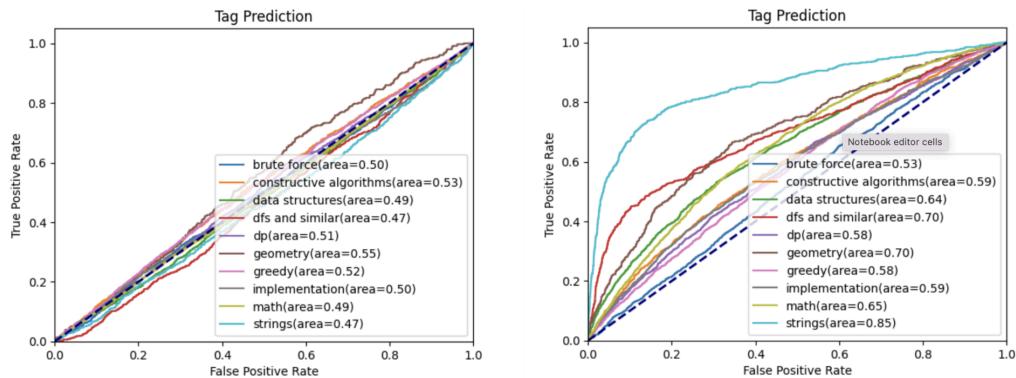


Figure 4.7: ROC before and after fine-tuning

By evaluating the ROC curve and calculating the AUC (Area Under the Curve), we can assess the overall performance of the classification model. The higher the AUC, the better the model is at distinguishing between positive and negative classes. The various model variations and fine-tuning steps help in understanding how different training strategies and hyperparameters affect the model's performance.

4.3.4 Solver Module

Evaluation Metric

- **HumanEval 0-shot**

HumanEval is a dataset designed to evaluate the ability of Large Language Models (LLMs) to generate correct and functional code snippets. It is composed of a set of programming problems that require the implementation of certain functionalities in Python. The **0-shot** evaluation variant of HumanEval assesses the model's ability to solve these problems without any fine-tuning or additional training examples. This metric directly measures the model's inherent understanding and generation capabilities in the context of programming tasks.

- **HumanEval+ 0-shot**

HumanEval+ builds upon the HumanEval dataset by introducing a more diverse set of programming tasks and languages. This expanded dataset not only evaluates Python code generation but also includes challenges in other programming languages, such as Java, C++, and JavaScript. The 0-shot evaluation in HumanEval+ maintains the premise of assessing the model's ability to generate correct code without prior instruction or examples, providing a more comprehensive evaluation of the model's multi-language programming proficiency.

- **MBPP 0-shot**

MBPP (Model-based Policy Optimization), although not originally a code generation dataset, has been adapted to assess LLMs in code-related tasks. In this context, MBPP contains more complex programming problems that often involve algorithmic thinking and optimization. The 0-shot evaluation in MBPP tests the model's capability to solve these intricate problems from scratch, without any specific training on the problem set. This metric is particularly challenging and

provides a measure of the model's advanced programming and logical reasoning skills.

- **MBPP+ 0-shot**

MBPP+ extends the MBPP dataset by increasing the diversity and complexity of the programming problems. This can include more advanced algorithmic challenges, real-world problems, and multi-language tasks. The 0-shot evaluation in MBPP+ demands that the model demonstrate an even higher level of programming expertise and adaptability across various domains and languages. It serves as a stringent test of the model's ability to reason, learn, and generate code effectively in unfamiliar situations.

- **MBPP 3-shot**

In contrast to the 0-shot evaluations, **MBPP 3-shot** provides the model with a small number of examples (typically three) to learn from before generating solutions to unseen challenges. This evaluation setup assesses the model's capacity for few-shot learning, where it can quickly adapt to new tasks with minimal guidance. The MBPP 3-shot evaluation measures not only the model's programming ability but also its ability to effectively utilize and generalize from a small number of examples, which is a practical scenario in many real-world applications.

- **LiveCodeBench**

LiveCodeBench is a dynamic evaluation metric for LLMs, focusing on real-time coding challenges. It aggregates problems from LeetCode [45], AtCoder [40], and Codeforces [41], offering a current and diverse test suite. This benchmark mirrors competitive programming environments, testing LLMs' adaptability and proficiency in algorithmic problem-solving.

Each of these datasets and evaluation methods offers a unique perspective on the capabilities of LLMs in the context of code generation and algorithmic problem-solving.

Different LLMs were assessed according to these evaluations, as shown in [tables 4.8 and 4.9](#).

Model	Size	HumanEv al 0-shot	HumanEv al+ 0-shot	MBPP 0-shot	MBPP+ 0-shot	MBPP 3-shot
CodeLlama- Base [32]	7B	33.5	25.6	52.1	41.6	38.6
StarCoder2 [33]	7B	35.4	29.9	54.4	45.6	51.0
DeepSeek-C oder-Base [34]	6.7B	47.6	39.6	70.2	56.6	60.6
CodeQwen1. 5-Chat [35]	7B	83.5	78.7	77.7	67.2	70.6

Table 4.8: Different LLMs assessed on various benchmarks

Model	Size	LiveCodeBench	
		All Time Pass@1 [x]	2023/9/1 ~ 2024/4/1 Pass@1 [x]
CodeLlama-Base [32]	7B	6.5	7.6
StarCoder2 [33]	7B	11.3	12.7
DeepSeek-Coder-Ba se [34]	6.7B	19.1	13.7
CodeQwen1.5 [35]	7B	21.8	19.3

Table 4.9: Different LLMs assessed on LiveCodeBench

CodeQwen was chosen due to its state-of-the-art results.

Enhancing LLM Responses With RAG

RAG grounds the chatbot to relevant problems, and solutions, to

1. Give it better context of problems it may have not trained on and seen before
2. Avoid hallucinations

An example of an LLM output enhanced with RAG is shown in [figure 4.8](#).

Query = “Could you solve this problem in C++?

One hot summer day Pete and his friend Billy decided to buy a watermelon. They chose the biggest and the ripest one, in their opinion. After that the watermelon was weighed, and the scales showed w kilos. They rushed home, dying of thirst, and decided to divide the berry, however they faced a hard problem.

Pete and Billy are great fans of even numbers, that's why they want to divide the watermelon in such a way that each of the two parts weighs an even number of kilos, at the same time it is not obligatory that the parts are equal. The boys are extremely tired and want to start their meal as soon as possible, that's why you should help them and find out if they can divide the watermelon in the way they want. For sure, each of them should get a part of the positive weight.”

LLM Response Without RAG	LLM Response With RAG
<pre>#include<bits/stdc++.h> using namespace std; int main() { int w; cin >> w; if(w % 2 == 0) { return true; } else { return false; } } return 0;</pre>	<pre>#include <iostream> int main() { int w; std::cin >> w; if ((w % 2 == 0) && (w!= 2)) { std::cout << "YES"; } else { std::cout << "NO"; } return 0; }</pre>
Rejected	Accepted

Figure 4.8: How RAG influences LLM response

As we have seen, RAG enhances the LLM output and gives better results.

Chapter 5

Run the Application

This chapter will explain how to run our web application locally and provide detailed information about its different pages, along with screenshots for clarity.

Summarization Page

- **Description:** The Summarization Page allows you to input a problem statement and provides a summarized version of the text, as shown in **figure 5.1**.

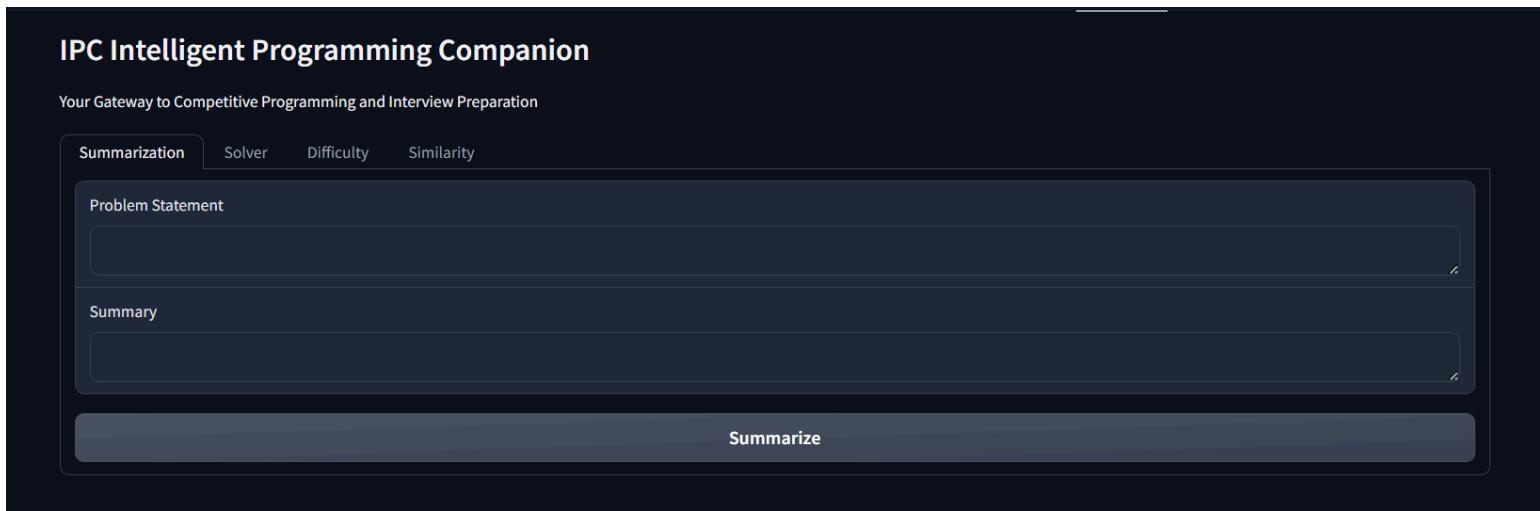


Figure 5.1: Summarization page

- **Usage:**
 1. Navigate to the Summarization Page.
 2. Enter the problem statement into the provided text box.
 3. Click the "Summarize" button.

View the summarized problem statement displayed in the Summary box as shown in **figure 5.2**.

The screenshot shows the 'IPC Intelligent Programming Companion' website. At the top, there is a navigation bar with tabs: 'Summarization' (which is active), 'Solver', 'Difficulty', and 'Similarity'. Below the navigation bar, there is a section titled 'Problem Statement' containing the original problem text. The problem text is as follows:

One hot summer day Pete and his friend Billy decided to buy a watermelon. They chose the biggest and the ripest one, in their opinion. After that the watermelon was weighed, and the scales showed w kilos. They rushed home, dying of thirst, and decided to divide the berry, however they faced a hard problem.

Pete and Billy are great fans of even numbers, that's why they want to divide the watermelon in such a way that each of the two parts weighs even number of kilos, at the same time it is not obligatory that the parts are equal. The boys are extremely tired and want to start their meal as soon as possible, that's why you should help them and find out, if they can divide the watermelon in the way they want. For sure, each of them should get a part of positive weight.

Below the 'Problem Statement' section is a 'Summary' section containing a simplified version of the problem:

Determine if a given weight 'w' can be divided into two positive even numbers.

At the bottom of the page is a large blue button labeled 'Summarize'.

Figure 5.2: A summarized problem statement

Similarity Model Page

- **Description:** The Similarity Model allows you to input either a problem or a solution and specify the number of similar problems/solutions (top k) you want to retrieve, as shown in **figure 5.3**.

The screenshot shows the 'IPC Intelligent Programming Companion' website. At the top, there is a navigation bar with tabs: 'Summarization' (which is active), 'Solver', 'Difficulty', and 'Similarity'. Below the navigation bar, there is a form for inputting a statement and a solution, and a section for specifying the number of similar problems to show. The form fields are as follows:

Statement
Paste your statement here!

Solution
Paste your solution here!

Number of similar problems to show
A slider set to 10.

Below the form is a large blue button labeled 'Submit'.

To the right of the form, there is a dark panel with a header 'Similar problems' and a list area that is currently empty.

Figure 5.3: The similarity page

- **Usage:**
 1. Navigate to the Similarity Model Page.
 2. Enter the problem or solution into the provided text box.
 3. Specify the value of top_k (the number of similar problems/solutions to retrieve).
 4. Click the "Submit" button.
 5. View the list of similar problems/solutions displayed in the output box as shown in **figure 5.4**.

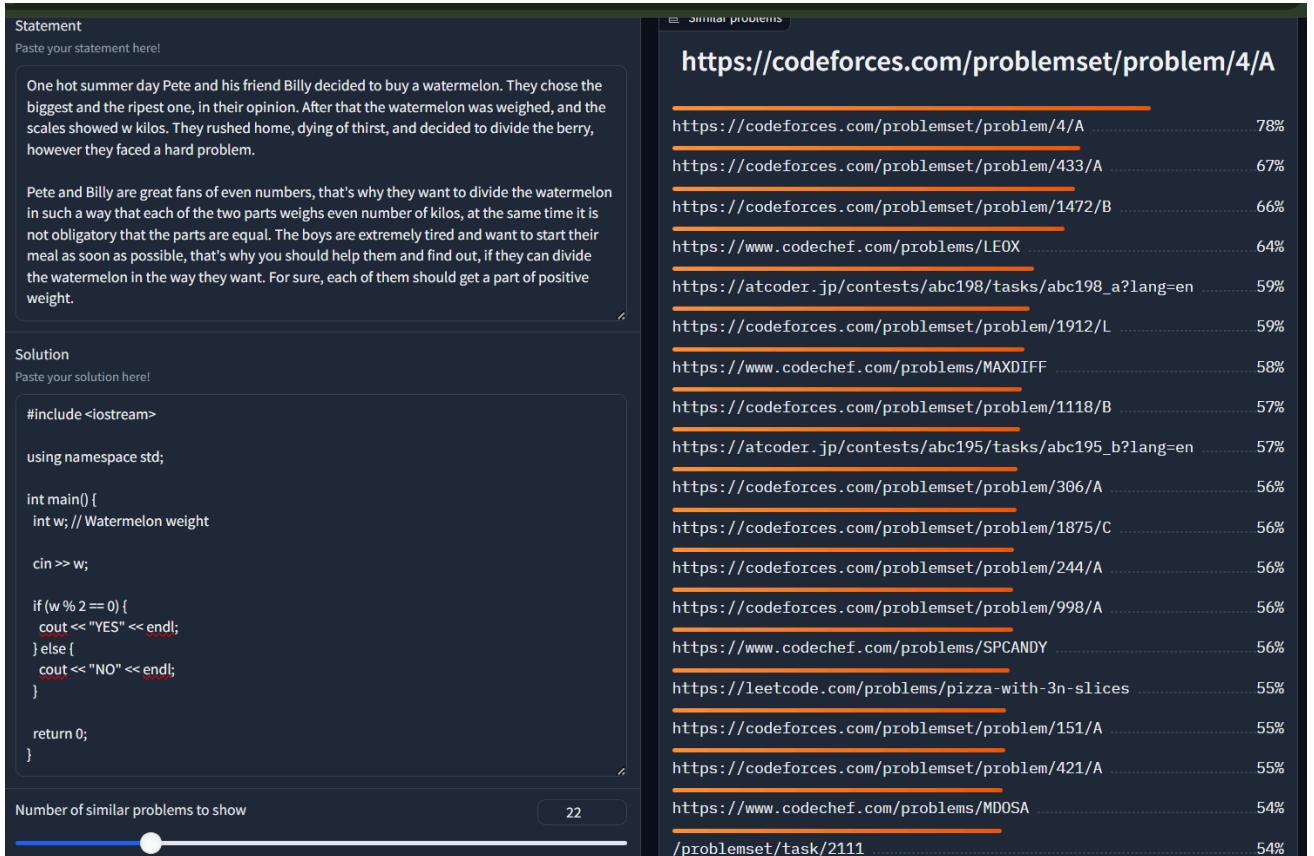


Figure 5.4: List of similar problems to a given statement and solution

Difficulty/Tag Prediction Page

- **Description:** The Difficulty/Tag Prediction Page allows you to input a problem and predicts its difficulty level and associated tags , as shown in **figure 5.5.**

The screenshot shows the IPC Intelligent Programming Companion interface. At the top, it says "IPC Intelligent Programming Companion" and "Your Gateway to Competitive Programming and Interview Preparation". Below this, there are four tabs: "Summarization", "Solver", "Difficulty" (which is highlighted in blue), and "Similarity". The main area has three input fields: "Problem Statement", "Tags", and "Difficulty", each with a text input box. At the bottom is a large "Predict" button.

Figure 5.5: Difficulty/Tag page

- **Usage:**

1. Navigate to the Difficulty/Tag Prediction Page.
2. Enter the problem statement into the provided text box.
3. Click the "Submit" button.
4. View the predicted difficulty and tags displayed.

Chatbot Page

- **Description:** The chatbot Page allows you to input a query and receive a response, as shown in **figure 5.6**.

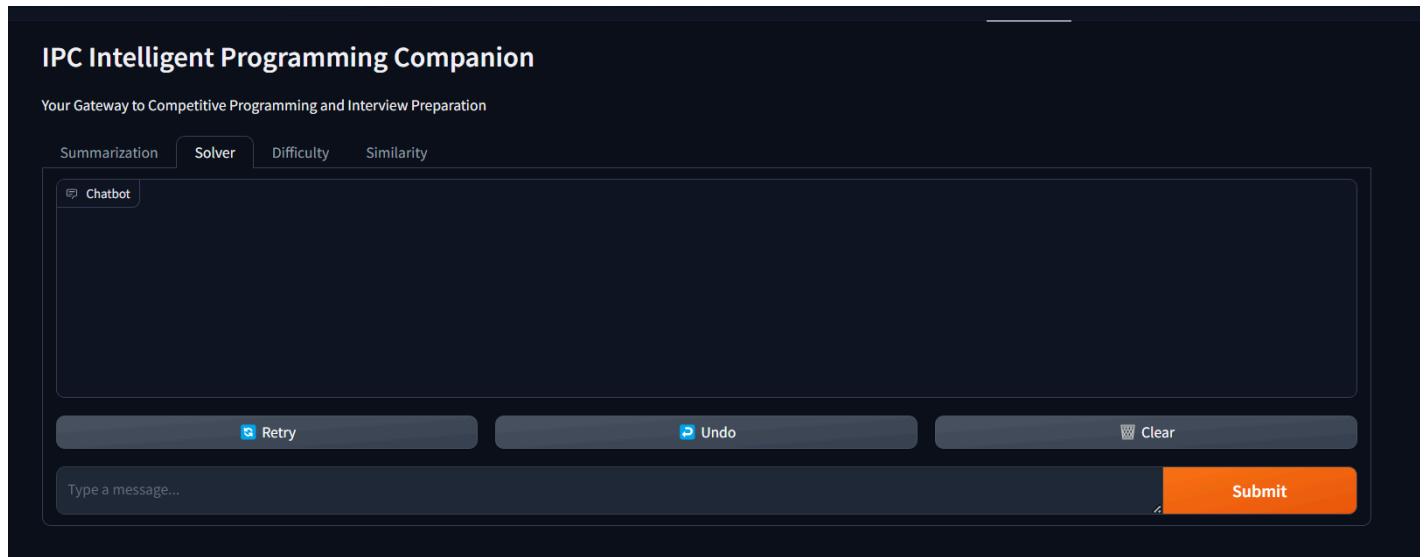


Figure 5.6: Solver page

- **Usage:**

1. Navigate to the Solver Page.
2. Enter your query into the provided text box.
3. Click the “Submit” button.
4. View the response displayed below the input box as shown in **figure 5.7**.

IPC Intelligent Programming Companion

Your Gateway to Competitive Programming and Interview Preparation

Summarization Solver Difficulty Similarity

Chatbot

One hot summer day Pete and his friend Billy decided to buy a watermelon. They chose the biggest and the ripest one, in their opinion. After that the watermelon was weighed, and the scales showed w kilos. They rushed home, dying of thirst, and decided to divide the berry, however they faced a hard problem.

Pete and Billy are great fans of even numbers, that's why they want to divide the watermelon in such a way that each of the two parts weighs even number of kilos, at the same time it is not obligatory that the parts are equal. The boys are extremely tired and want to start their meal as soon as possible, that's why you should help them and find out, if they can divide the watermelon in the way they want. For sure, each of them should get a part of positive weight.

You're right! The provided problem set doesn't seem relevant to the watermelon problem. Let's focus on helping Pete and Billy.

Here's a slightly different way to think about the solution:

The Key Insight

- A number is even if it's divisible by 2, leaving no remainder.
- If the watermelon's weight (w) is odd, it will always leave a remainder of 1 when divided by 2.
- Therefore, if the watermelon's weight is odd, there's no way to divide it into two even parts.

C++ Code (Simplified)

```
#include <iostream>

using namespace std;

int main() {
    int w; // Watermelon weight
    cin >> w;

    if (w % 2 == 0) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }

    return 0;
}
```

Explanation:

- This code does the same thing as before. We check if the watermelon's weight (w) is divisible by 2.
- If it is, we print "YES" (they can divide it).
- If it's not, we print "NO" (they can't divide it).

Important Note: The provided problem set is not relevant to this specific problem. This is a simple logic problem, and we can solve it directly without needing complex algorithms.

Retry

Undo

Clear

Type a message...

Submit

Figure 5.7: The solver providing a comprehensive solution and explanation to a given problem query

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Our solution aimed to address significant challenges in competitive programming and interview preparation through the development of a centralized platform with integrated modules focused on problem uniqueness, solver efficiency, difficulty prediction, and problem clarity.

The implemented modules—Summarization, Similarity, Difficulty/Tags Prediction, and Solver—have demonstrated strong performance metrics. The Similarity Module achieved a remarkable **99.6% recall@10**, indicating its efficacy in identifying similar problems and ensuring problem set originality. The Difficulty/Tags Prediction Module showed robust performance with a ROC AUC score of **76.37%**, guiding users effectively in problem selection based on difficulty levels and relevant tags.

The findings of this project are crucial for advancing the field of competitive programming and interview preparation. They underscore the significance of integrated platforms in overcoming existing challenges and improving learning outcomes.

By addressing issues such as problem repetition, inconsistent difficulty assessments, platform fragmentation, and ambiguous problem descriptions, our platform aims to create a more cohesive and supportive environment for competitive programmers and interview candidates. The tools developed not only streamline problem-solving processes but also foster innovation and knowledge sharing within the community.

6.2 Recommendations for Future Work

While our project has achieved substantial results, there are several areas for future exploration and improvement:

- **Enhancing AI Capabilities:** Continuously updating and refining AI models (such as LLMs and BERT-based architectures) to improve accuracy and responsiveness in problem-solving scenarios.
- **Expansion of Dataset and Features:** Incorporating larger and more diverse datasets to enhance the predictive capabilities of the Difficulty/Tags Prediction Module and improve problem categorization.
- **User Feedback Integration:** Developing mechanisms to gather and incorporate user feedback to further optimize platform functionalities and user experience.
- **Collaborative Problem-Solving:** Exploring ways to facilitate collaborative problem-solving among users and problem setters, potentially through crowdsourcing and community-driven initiatives.
- **Integration with Educational Institutions:** Partnering with educational institutions to integrate the platform into curricula and enhance student learning experiences in computer science and related fields.

References

1. Stoica, A., Băbiceanu, D., Rebedea, T., & Mihăescu, M.(2023). Unsupervised Voting for Detecting the Algorithmic Solving Strategy in Competitive Programming Solutions. Research Article <https://doi.org/10.21203/rs.3.rs-2834777/v1>
2. Athavale, V., Naik, A., Vanjape, R., & Shrivastava, M. (2019, April 4). *Predicting algorithm classes for programming word problems*. arXiv.org. <https://doi.org/10.48550/arXiv.1903.00830>
3. Crawshaw, M. (2020, September 10). *Multi-task learning with deep neural networks: A survey*. arXiv.org. <https://doi.org/10.48550/arXiv.2009.09796>
4. Sebastian Ruder. An overview of multi-task learning in deep neural networks. arXiv preprint arXiv:1706.05098, 2017.
5. Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., & Wang, H. (2024, March 27). Retrieval-augmented generation for large language models: A survey. arXiv.org. <https://arxiv.org/abs/2312.10997>
6. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., & Lample, G. (2023, February 27). Llama: Open and efficient foundation language models. arXiv.org. <https://arxiv.org/abs/2302.13971v1>
7. Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., Hui, B., Ji, L., Li, M., Lin, J., Lin, R., Liu, D., Liu, G., Lu, C., Lu, K., ... Zhu, T. (2023, September 28). Qwen Technical Report. arXiv.org. <https://arxiv.org/abs/2309.16609>
8. Lomshakov, V., Kovalchuk, S., Omelchenko, M., Nikolenko, S., Aliev, A. (2023). Fine-Tuning Large Language Models for Answering Programming Questions with Code Snippets. In: Mikyška, J., de Mulatier, C., Paszynski, M., Krzhizhanovskaya, V.V., Dongarra, J.J., Sloot, P.M. (eds) Computational Science – ICCS 2023. ICCS 2023. Lecture Notes in Computer Science, vol 14074. Springer, Cham. https://doi.org/10.1007/978-3-031-36021-3_15
9. Zelikman, E., Huang, Q., Poesia, G., Goodman, N., & Haber, N. (2023). Parsel: Algorithmic Reasoning with Language Models by Composing Decompositions.

10. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, Ł., & Polosukhin, I. (2023). Attention Is All You Need.
11. Heffernan, K., & Teufel, S. (2022). Problem-solving Recognition in Scientific Text. 13th Conference on Language Resources and Evaluation (LREC 2022).
12. Kim, J., Cho, E., Kim, D., & Na, D. (2023, October 9). *Problem-solving guide: Predicting the algorithm tags and difficulty for competitive programming problems*. arXiv.org. <https://arxiv.org/abs/2310.05791>
13. Google. (n.d.). *Classification: Roc curve and AUC | machine learning | google for developers*. Google. <https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc>
14. Zaheer, M., Guruganesh, G., Dubey, A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., & Ahmed, A. (2021, January 8). *Big bird: Transformers for longer sequences*. arXiv.org. <https://arxiv.org/abs/2007.14062>
15. Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019, May 24). *Bert: Pre-training of deep bidirectional Transformers for language understanding*. arXiv.org. <https://arxiv.org/abs/1810.04805>
16. W. Looi, Analysis of code submissions in competitive programming contests.
17. J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al., Program synthesis with large language models, arXiv preprint arXiv:2108.07732.
18. Norouzi, S., Cao, Y.: Semantic parsing with less prior and more monolingual data. CoRR abs/2101.00259 (2021). <https://arxiv.org/abs/2101.00259>
19. Beau, N., Crabbé, B.: The impact of lexical and grammatical processing on generating code from natural language. In: Findings of the Association for Computational Linguistics: ACL 2022, pp. 2204–2214. Association for Computational Linguistics, Dublin, Ireland (2022). <https://doi.org/10.18653/v1/2022.findings-acl.173>
20. Soliman, A.S., Hadhoud, M.M., Shaheen, S.I.: MarianCG: a code generation transformer model inspired by machine translation. J. Eng. Appl. Sci. 69(1), 1–23 (2022)
21. Shen, B., Zhang, J., Chen, T., Zan, D., Geng, B., Fu, A., Zeng, M., Yu, A., Ji, J., Zhao, J., Guo, Y., & Wang, Q. (2023, July 27). Pangu-Coder2: Boosting

- large language models for code with ranking feedback. arXiv.org. <https://arxiv.org/abs/2307.14936>
22. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., d'Autume, C. de M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., ... Vinyals, O. (2022, February 8). Competition-level code generation with AlphaCode. arXiv.org. <https://arxiv.org/abs/2203.07814>
23. See, A., Liu, P. J., & Manning, C. D. (2017, April 25). *Get to the point: Summarization with pointer-generator networks*. arXiv.org. <https://doi.org/10.48550/arXiv.1704.04368>
24. Zhang, J., Zhao, Y., Saleh, M., & Liu, P. J. (2020). *PEGASUS: Pre-training with extracted gap-sentences for abstractive summarization*. In *Proceedings of the 37th International Conference on Machine Learning* (Vol. 119). <https://proceedings.mlr.press/v119/zhang20ae/zhang20ae.pdf>
25. Reimers, N., & Gurevych, I. (2019, August 27). *Sentence-BERT: Sentence embeddings using Siamese BERT-networks*. arXiv.org. <https://doi.org/10.48550/arXiv.1908.10084>
26. S. Subramanian, A. Arun Kumar, M. Muthu Nagappan, and R. Suresh, "Classification and Recommendation of Competitive Programming Problems Using CNN," in *Smart Secure Systems – IoT and Analytics Perspective*, December 2018, pp. 262-272. DOI: 10.1007/978-981-10-7635-0_20.
27. Rahman, M. M., Das, B. C., Biswas, A. A., & Anwar, M. M. (2023). *Predicting participants' performance in programming contests using deep learning techniques*. arXiv.org. Retrieved from <https://arxiv.org/pdf/2302.05602>
28. Team, G., Georgiev, P., Lei, V. I., Burnell, R., Bai, L., Gulati, A., Tanzer, G., Vincent, D., Pan, Z., Wang, S., Mariooryad, S., Ding, Y., Geng, X., Alcober, F., Frostig, R., Omernick, M., Walker, L., Paduraru, C., Sorokin, C., ... Vinyals, O. (2024a, June 14). *Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context*. arXiv.org. <https://arxiv.org/abs/2403.05530>
29. Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. de las, Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P.,

- Subramanian, S., Yang, S., ... Sayed, W. E. (2024, January 8). *Mixtral of Experts*. arXiv.org. <https://arxiv.org/abs/2401.04088>
30. Team, Gemma, Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., Sifre, L., Rivière, M., Kale, M. S., Love, J., Tafti, P., Hussenot, L., Sessa, P. G., Chowdhery, A., Roberts, A., Barua, A., Botev, A., Castro-Ros, A., Slone, A., ... Kenealy, K. (2024a, April 16). *Gemma: Open models based on Gemini Research and Technology*. arXiv.org. <https://arxiv.org/abs/2403.08295>
31. Judging LLM-as-a-judge with MT-bench and Chatbot Arena. (n.d.). [https://arxiv.org/pdf/2306.05685](https://arxiv.org/pdf/2306.05685.pdf)
32. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., ... Synnaeve, G. (2024, January 31). *Code llama: Open foundation models for code*. arXiv.org. <https://arxiv.org/abs/2308.12950>
33. Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., ... de Vries, H. (2023, December 13). *Starcoder: May the source be with you!*. arXiv.org. <https://arxiv.org/abs/2305.06161>
34. Deepseek-ai/deepseek-coder-6.7b-base · huggingface. deepseek-ai/deepseek-coder-6.7b-base · Hugging Face. (n.d.). <https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-base>
35. QwenLM. (n.d.). *QWENLM/Codeqwen1.5: Codeqwen1.5 is the code version of Qwen, the large language model series developed by Qwen team, Alibaba Cloud*. GitHub. <https://github.com/QwenLM/CodeQwen1.5>
36. Vinayak Athavale, Aayush Naik, Rajas Vanjape, and Manish Srivastava. Predicting algorithm classes for programming word problems. In Conference on Empirical Methods in Natural Language Processing, 2019
37. Sebastian Ruder. An overview of multi-task learning in deep neural networks. arXiv preprint arXiv:1706.05098, 2017.
38. Michael Crawshaw. Multi-task learning with deep neural networks: A survey. arXiv preprint arXiv:2009.09796, 2020.
39. Chatbot Arena: Find the best chatbot builder. Chatbot Arena: Find the Best Chatbot Builder. (n.d.). <https://chatbotarena.com/>