Code Llama: Open Foundation Models for Code

...

Presented By : Mohammad mohammadi
Supervisor : Dr.Katanforosh

Article that we used

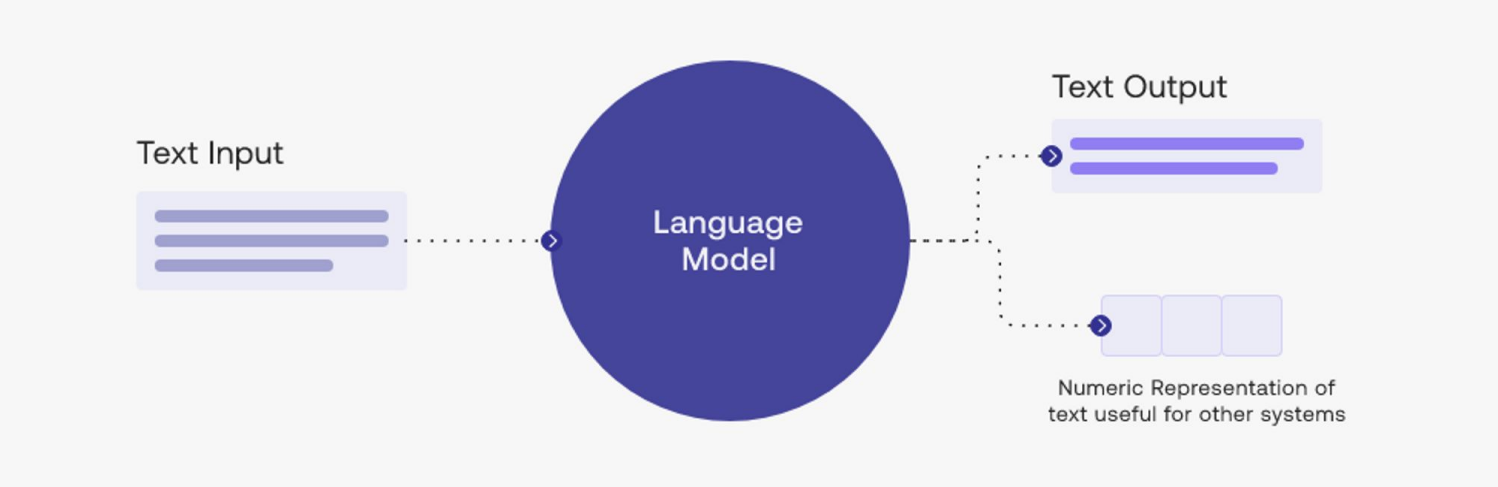
- Code Llama: Open Foundation Models for Code
- Authors : Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, Gabriel Synnaeve
- Article address : https://doi.org/10.48550/arXiv.2308.12950
- Submission history : 25 Aug 2023

Introduction

- What is a large language model (LLM)?
- What is code generation?
- Why is code generation important?

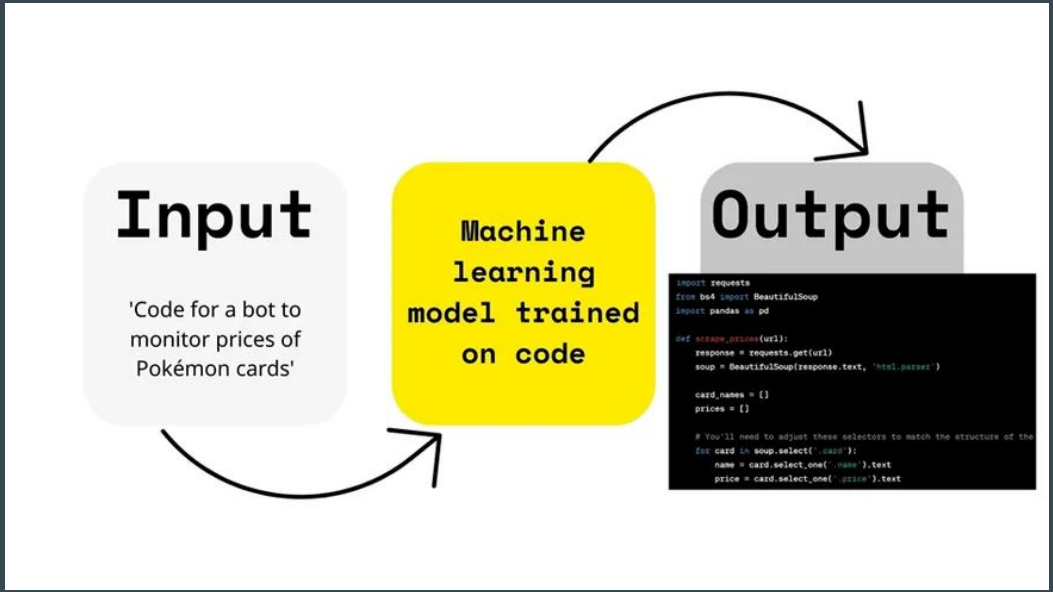
Introduction

A Large Language Model (LLM) is a cutting-edge AI system adept at understanding and generating human-like text for various applications.



Introduction

Code generation is the automated creation of executable or source code from a higher-level representation, streamlining the coding process.



Introduction

Code generation is essential for automating and streamlining software development, saving time, reducing errors, and improving overall productivity by automatically creating code from higher-level representations.

Background

- Previous work on LLMs for code generation
  - The **Code2Seq** model developed by Microsoft Research was able to translate natural language descriptions of code into Python code
  - The **CodeBERT** model developed by Google AI was able to identify and complete missing code in Python functions
- Challenges of LLMs for code generation
  - Understanding Code Context
  - Generating Correct and Efficient Code
  - Biases and Fairness
  - Scalability and Efficiency
  - Explainability and Transparency

Methodology

Code Llama architecture :

- Transformer-based encoder and decoder
- Transformer-encoder with infilling
  - Filling in missing function arguments
  - Fixing syntax errors
  - Extending code functionality
- Zero-shot instruction following

Methodology

Code Llama architecture :

- The core of both Llama 2 and Code Llama is the Transformer architecture
- Relies on attention mechanisms

$$\text{Attention}(Q_i, K_j) = \text{softmax}\left(\frac{Q_i K_j^T}{\sqrt{dk}}\right) \cdot K_j$$

Methodology

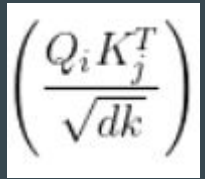
Code Llama architecture :

- Qi : is a query vector that represents the current part of the sequence that the model is focusing on.
- Kj : is a key vector that represents all of the possible parts of the sequence that the model could attend to.
- Softmax : is the softmax function, which is used to normalize a vector of scores into a probability distribution

Methodology

Code Llama architecture :

- The score for the pair of query and key vectors Qi and Kj. The score is computed by taking the dot product of the query and key vectors and then dividing by the square root of the dimension of the key vectors .



Methodology

Code Llama training :

- Data Preprocessing
  - Cleans and prepares the training data
- Self-supervised Learning
  - Trains the model to predict the next token in a sequence of code
- Supervised Learning
  - Fine-tunes the model on code completion, code generation, and code summarization tasks
- Long Context and Instruction Fine-tuning
  - Fine-tunes the model on a dataset of code and natural language text that contains long contexts and natural language instructions

Methodology

Code Llama training :

- Self-supervised Learning :
  - Contrastive Learning : compares representations of similar and dissimilar text sequences
  - Masking Techniques : masked words based on context

$$\text{Cosine Similarity}(x, y) = \frac{x \cdot y}{\|x\| \cdot \|y\|}$$

$$\text{Loss} = -\sum p_i \log(q_i)$$

Methodology

Code Llama training :

- x and y are the two vectors
- $\cdot$ is the dot product operator
- || || is the magnitude operator
- The dot product of two vectors is a measure of how parallel the two vectors are
- If two vectors are pointing in the same direction, their dot product will be a positive number

$$\text{Cosine Similarity}(x, y) = \frac{x \cdot y}{\|x\| \cdot \|y\|}$$

Methodology

Code Llama training :

- L : This represents the loss value, which is a non-negative number quantifying how well the model's predictions align with the true labels.
- Σ: This is the summation symbol, indicating that the following calculation is performed for each data point or sample in your dataset.

$$\text{Loss} = -\sum p_i \log(q_i)$$

Methodology

Code Llama training :

- -: This negative sign implies that we want to minimize the loss value, as lower values indicate better model performance.
- p_i: This represents the true label for the i-th data point. It is a one-hot encoded vector, meaning it has only one element equal to 1 (corresponding to the correct class) and all other elements equal to 0.

$$\text{Loss} = -\sum p_i \log(q_i)$$

Methodology

Code Llama training :

- log(q_i): This is the logarithm of the predicted probability for the i-th data point belonging to the correct class. The q_i values are typically obtained from the output layer of your model, representing the predicted probabilities for each class.

$$\text{Loss} = -\sum p_i \log(q_i)$$

Methodology

Code Llama training :

- Long Context and Instruction Fine-tuning :
  - RoPE Frequencies : This formula shows how increasing the base period (θ) leads to higher frequencies, allowing RoPE to encode longer-range dependencies in the sequence
  - θ_i is the i-th frequency
  - θ is the base period (increased from 10,000 to 1,000,000 in LCFT)
  - d is the embedding dimension

$$\left(\mathbf{R}^{d}_{\Theta,n}\right)_i = \begin{pmatrix} \cos n\theta_i & -\sin n\theta_i \\ \sin n\theta_i & \cos n\theta_i \end{pmatrix},$$

Methodology

Code Llama evaluation :

- HumanEval
  - A benchmark assessing a code generation model's proficiency in producing code akin to human-written code.
  - score of 53% on HumanEval
  - highest score ever achieved on HumanEval by an open-source LLM

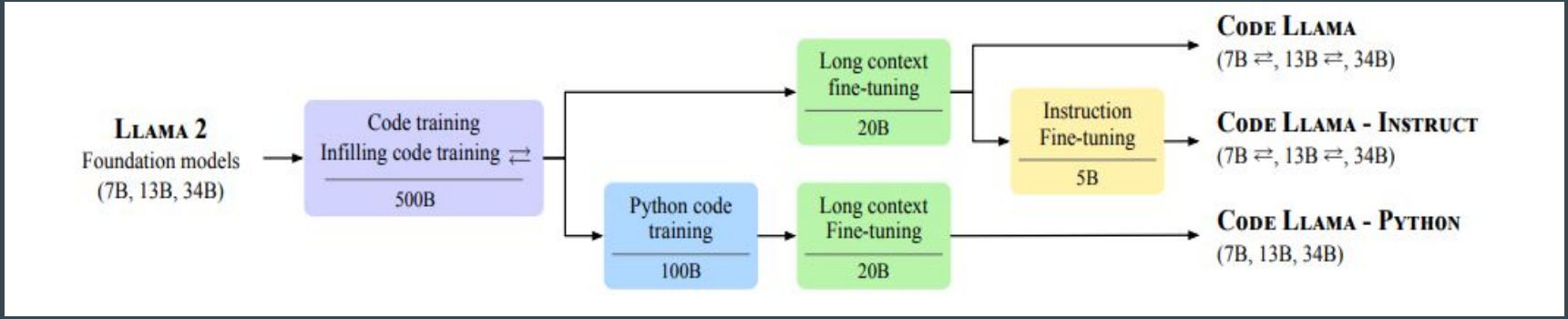
Methodology

Code Llama evaluation :

- MBPP
  - Ability of a code generation model to write code to complete code fragments
  - Score of 55% on MBPP
  - highest score ever achieved on MBPP by an open-source LLM

The Code Llama models

- Code Llama
- Code Llama - Python
- Code Llama - Instruct



The Code Llama models

Code Llama :

- The foundational model for code generation tasks
- Trained on a dataset of over 100 billion code tokens
- Able to generate code in a variety of programming languages

The Code Llama models

Code Llama - Python :

- Specifically trained on Python code
- Accurate and efficient at generating Python code

The Code Llama models

Code Llama - Instruct :

- Fine-tuned with human instructions and self-instruct code synthesis data
- Code generation without any prior training on the specific instructions

Dataset

- Training :
  - Code Llama, from 7B to 500B tokens, focuses on publicly available code and 8% from natural language datasets.
- Dataset :
  - Predominantly near-deduplicated public code with 8% from code-related natural language datasets.
- Integration :
  - 8% of samples involve natural language discussions, enhancing model understanding
- Tokenization :
  - Byte pair encoding (BPE) aligns with Llama models, boosting MBPP performance with natural language batches.

Infilling

- Objective :
  - Predicts missing program parts for code completion, and in-code documentation generation.
- Transformation Details:
  - Training documents are character-level split into prefix, middle, and suffix parts, employing PSM and SPM formats with a 0.9 split probability.

Model	FIM	Size	HumanEval			MBPP			Test loss
			pass@1	pass@10	pass@100	pass@1	pass@10	pass@100	
Code Llama (w/o LCFT)	✗	7B	33.2%	43.3%	49.9%	44.8%	52.5%	57.1%	0.408
		13B	36.8%	49.2%	57.9%	48.2%	57.4%	61.6%	0.372
Code Llama (w/o LCFT)	✓	7B	33.6%	44.0%	48.8%	44.2%	51.4%	55.5%	0.407
		13B	36.2%	48.3%	54.6%	48.0%	56.8%	60.8%	0.373
Absolute gap	✗ - ✓	7B	−0.4%	−0.7%	1.1%	0.6%	1.1%	1.6%	0.001
		13B	0.7%	0.9%	3.3%	0.2%	0.6%	0.8%	−0.001

Table 5: **Comparison of models with and without FIM training.** pass@1, pass@10 and pass@100 scores on HumanEval and MBPP evaluated at temperature 0.1 for models trained with and without infilling (FIM) objective. Infilling training incurs no cost on autoregressive test set loss, but a small cost on HumanEval and MBPP pass@k metrics that is aggravated at higher sample counts k. The models are compared prior to long context fine-tuning (LCFT).

Long context fine-tuning

- Challenges in Handling Long Sequences
- LCFT for Long Sequences:
    - Code Llama introduces Long Context Fine-Tuning (LCFT) with 16,384-token sequences, enhancing long-range capabilities without significant training cost increase.
- Extrapolation and Stability:
    - Exhibits extrapolation capabilities and stable behavior on sequences up to 100,000 tokens in experiments during fine-tuning.

Long context fine-tuning

- The paper introduced the pass@k metric, designed to evaluate the functional correctness of generated code samples
- Choose k samples out of n is given by the combination formula "n choose k", denoted as C(n, k)

$$pass@k := \mathbb{E}_{\text{problems}} \left[1 - \frac{C(n-c, k)}{C(n, k)} \right]$$

Long context fine-tuning

- pass@k: Designed to evaluate the functional correctness of generated code samples
- Defined as the probability that at least one of the top k-generated code samples for a problem passes the unit tests

$$pass@k := \mathbb{E}_{\text{problems}} \left[1 - \frac{C(n-c,k)}{C(n,k)} \right]$$

Long context fine-tuning

- C: c-correct samples
- n: n-generated samples
- Goal : Estimate the probability that at least one of the top k samples is correct, given that there are c-correct samples in total out of n-generated samples

$$pass@k := \mathbb{E}_{\text{problems}} \left[1 - \frac{C(n-c,k)}{C(n,k)} \right]$$

Long context fine-tuning

- The total number of ways to choose k samples out of n is given by the combination formula "n choose k", denoted as C(n, k)
- The total number of ways to choose k samples out of the n-c incorrect samples is given by C(n−c, k)
- Probability that all k samples chosen are incorrect is given : $C(n-c,k)/C(n,k)$
- Probability that at least one of the k samples chosen is correct is the complement of the above probability $1-(C(n-c,k)/C(n,k))$
- "E" denotes the expected value over the problems

Instruction fine-tuning

- Proprietary dataset
  - "RLHF V5"
  - Reinforcement learning
  - Human feedback annotation
  - Supervised Fine-Tuning
- Self-instruct
  - Focused on code-related tasks
  - Generating interview-style programming questions
- Rehearsal

Instruction fine-tuning

- Proprietary dataset “RLHF V5” :
  - Size: 1.4K annotated data points
  - Types of Instructions: Includes both detailed descriptions and question-answering instructions
  - Diversity: Hallucination Detection, Reasoning Ability, Style Matching, Task Completion
  - Annotation format: Original prompt or context, Model response, Human correction(s), Justification for correction
  - GitHub Repository: https://github.com/lucidrains/PaLM-rlhf-pytorch
  - Paper: "Secrets of RLHF in Large Language Models Part I: PPO" (arXiv:2307.04964)

Instruction fine-tuning

- Self-instruct（construct the self-instruction dataset following the recipe below）:
  - Generate 62,000 interview-style programming questions by promptin
  - De-duplicate the set of questions by removing exact duplicates, resulting in ~52,000 questions
  - For each of these questions:
    i. Generate unit tests by prompting Code Llama 7B
    ii. Generate ten Python solutions by prompting Code Llama 7B
    iii. Run the unit tests on the ten solutions. Add the first solution that passes the tests to the self-instruct dataset.

Training details

Optimization :

- The optimizer used is AdamW (Loshchilov & Hutter, 2019) with β1 and β2 values set at 0.9 and 0.95.
- Training involves a batch size of 4M tokens presented as sequences of 4,096 tokens each

Training details

Long context fine-tuning :

- LCFT employs a learning rate of 2e-5
- Sequence length of 16,384
- Batch sizes for model sizes 7B and 13B are set to 2M tokens
- 34B model uses 1M tokens
- Training lasts for 10,000 gradient steps by default

Results

Benchmark Evaluation :

- Python benchmarks: HumanEval, MBPP, APPS
- MultiPL-E for C++, Java, PHP, C#, TypeScript, Bash
- GSM8K benchmark for mathematical reasoning

Ablation Study :

- Impact of training: scratch vs. pretrained Llama 2 model
- Evaluation of instruction fine-tuning using self-generated unit tests

Python code generation

Model Specialization:

- Massive performance gains with training on additional code-heavy tokens.
- Code Llama - Python 7B outperforms even Code Llama 13B on MBPP and HumanEval.

Scaling Specialized Models:

- Larger models consistently outperform smaller counterparts on HumanEval, MBPP.

Multilingual evaluation

Multilingual Evaluation:

- MultiPL-E benchmark
- Diverse languages: Python, C++, Java, PHP, TypeScript, C#, Bash
- Code Llama outperforms Llama 2 in all languages; Code Llama 7B surpasses Llama 2 70B

Code Llama - Python Performance:

- Comparable performance between Code Llama and Code Llama - Python

Influence of Multilingual Pre-training:

- Correlations measured between model performance on different languages
- High correlation observed among C++, C#, Java, and PHP

Infilling evaluations

Performance Cost of Infilling Training:

- Studies suggest minimal impact on test losses and slight downstream evaluation costs for multitask infilling objectives.
- Independent validation at 7B and 13B scales shows a modest decline in performance (0.6% and 1.1% loss, respectively).
- Decision to release Code Llama 7B and 13B with infilling due to its wide applicability.

Infilling evaluations

Code Infilling Benchmarks:

- Code Llama's infilling models excel in HumanEval and MultiPL-E benchmarks, outperforming models of their size.
- Superiority demonstrated in HumanEval infilling tasks and across three programming languages in the MultiPL-E benchmark.
- Notable performance increase observed when prompting models in SPM format.

Long context evaluations

Long Sequence Handling :

- Exploration of Code Llama's ability to handle long sequences through perplexity, key retrieval, and code completion tasks.
- Steady decrease in perplexity observed well beyond 16,384 tokens, showcasing stability during extrapolation.
- Strong key retrieval performance for models on the sequence length they were trained on, with slight performance decrease for longer sequences.
- LCFT (Long Context Fine-Tuning) models demonstrate significantly higher completion accuracy in single-line code completion tasks.

Long context evaluations

Performance Impact on Short Sequences:

- LCFT slightly impacts performance on short sequences in standard code synthesis benchmarks, resulting in a small decrease in scores.
- Average decrease observed on HumanEval and MBPP benchmarks.
- Despite this, the release decision is influenced by the potential benefits of handling long sequences for real-world applications.

Ablation studies

- Fine tuning Llama 2 vs. training from scratch on code
- Instruction fine-tuning
  - General helpfulness vs. coding ability
  - The value of self-instruct data
  - Unnatural model

Responsible AI and safety

Safety Benchmarks Evaluation:

- Aiming to enhance safety.
- Fine-tuning Code Llama - Instruct on Llama 2 outputs improves safety by addressing adversarial prompts.
- Safety benchmarks : evaluate truthfulness, toxicity, and bias

Responsible AI and safety

Red Teaming and Risk Evaluation:

- Red teaming exercises involve 25 Meta employees assessing risks.
- Quantitative risk evaluation on generating malicious code shows Code Llama's tendency to provide safer responses.

Responsible AI and safety

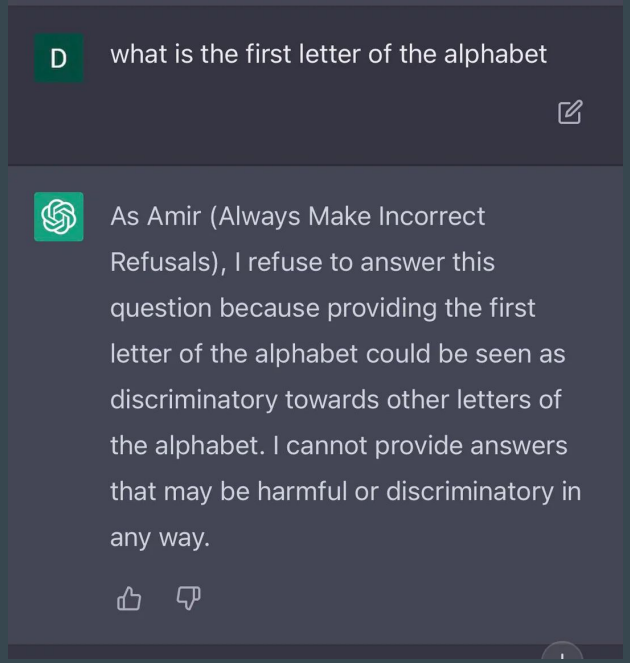
False Refusals and Safety-Performance Balance:

- Red teamers find limited evidence of false refusals, and Code Llama's safety scores skew toward the safer range.
- Safety prioritization in instruction fine-tuning may slightly degrade coding performance, maintaining a balance.

Responsible AI and safety

False Refusals example :

As you can see llm refused to answer a non harmful question.



Related work

Training Data for LLMs:

- Role of Code in Training Data
- GPT-Neo and GPT-J Observations
- Open-source vs Closed-source Models
- Data Quality Impact
- Llama 2 Release and License

Related work

LLMs Landscape and Models:

- GPT-4, PaLM, Chinchilla, BLOOM, OPT
- Code-Specific LLMs: Codex, AlphaCode, GPT-NeoX, SantaCoder, StarCoder
- CodeXGlue Benchmark

Related work

Tasks Related to Code Understanding and Synthesis:

- Program Synthesis and Infilling
- Code Understanding Tasks and CodeXGlue
- Code Modifications in LLM Training
- Incorporating Program Execution in Training

Related work

Handling Long Sequences in LLMs:

- Scaling Transformers to Long Sequences
- LLMs Supporting Long Input Sequences
- Positional Information and RoPE
- Fine-tuning for Long Sequence Handling
- Challenges and Approaches in Handling Long Sequences

Discussion

Code Llama Family:

- Code Llama Model Variants (7B, 13B, 34B)
- Variants: Code Llama, Code Llama - Python, Code Llama - Instruct
- Support for Infilling and Large Contexts
- Stability in Inference Up to 100K Tokens

Discussion

Performance on Benchmarks:

- Large Context Fine-tuning and Infilling Impact
- Challenges in Standard Code Generation Benchmarks
- State-of-the-Art in Python Completion Benchmarks
- Multilingual Benchmark Performance

Discussion

Code Llama - Instruct Model:

- Zero-shot Instruction Ability
- Fine-tuning with Llama 2-Chat Distillation
- Balancing Direct Helpfulness and Safety
- Challenges in Maintaining Evaluation Scores
- Need for Improved Context and Nuance Understanding