Explanation of Programming Errors with Open-source Large Language Models

Abhiram Kandiyana, Alfredo Fernandez, Sree Rushitha Santhoshi Mamidala

Department of Computer Science and Engineering
University of South Florida

1 Abstract

The efficiency of debugging is critically impacted by the quality of error log messages. The compiler messages are mostly vague and sometimes misleading which leaves the programmer clueless about the source of error. Most beginners spend hours trying to find and fix a bug due to this issue. This project explores the application of large language models (LLMs) for enhancing debugging productivity through clearer and more actionable error explanations. Utilizing the pretrained Code-LLaMA model, known for its adeptness in programming and coding tasks, we investigate two primary approaches to improve explanation quality: fine-tuning and prompting strategies. Our study introduces a custom error and alignment dataset tailored to refine the baseline capabilities of the Code-LLaMA model, aiming to produce more relevant and accurate error explanations. We assess the effectiveness of each strategy using Perplexity (PPL), BERT-score, and comprehensive human evaluation metrics. These metrics evaluate the clarity, relevance, and actionability of the explanations, contributing to an empirical understanding of how LLMs can be optimized to support software developers in debugging tasks. Our findings not only shed light on the comparative strengths and limitations of fine-tuning versus prompting within the context of error explanation but also propose a framework using few prompt strategies for further enhancement of automated debugging assistance tools in software development environments.

Keywords: Text Alignment, Prompting, Code-Llama, Low Rank Adaptation (LoRA).

2 Introduction

Debugging is an integral and time-consuming aspect of the software development process. Developers often spend a significant portion of their time interpreting error logs, which are automatic outputs of systems that report failures or bugs in code. These logs, while informative, can be cryptic and difficult to understand, especially for less experienced developers or those new to a codebase. There are cases where the error messages are misleading, and it takes some experience to understand the error message and how to debug accordingly. Enhancing the clarity and usefulness of error log explanations could substantially improve debugging efficiency and reduce development time.

The current automated method based on sequence-to-sequence models focused on error fixing requires both the program and the error message as input [1] [2]. Though these works have shown good results in a few use cases, the requirement of the entire program as the input becomes a bottleneck as well as confidentiality concern. It makes the inference slow and leads to memory issues due to the requirement for larger models and longer input sequences. Such methods cannot be used on a cloud service when dealing with confidential codebases (military, corporate workplace) as that would raise privacy concerns.

Recent advancements in artificial intelligence (AI), particularly in large language models (LLMs), have shown promise in tasks that require understanding and generating human-like text. The Code-LLaMA model, a variant of LLMs specifically trained on programming language samples, offers a practical method to handle the challenge of interpreting error logs. By using such models, we can potentially transform raw error logs into more comprehensible and actionable insights without requiring additional information regarding the codebase that produced the error. This was inspired by the observation that experts in a field can usually pinpoint the cause of an error by only seeing the error text, due to their experience and insights.

This paper focuses on exploiting the capabilities of the pretrained Code-LLaMA model to create more helpful error explanations. We compare two overarching techniques to enhance the model's performance: fine-tuning and prompting. Fine-tuning involves retraining the model on a specialized dataset, our custom error and alignment dataset designed to reflect naturally occurring errors and their helpful interpretations. Prompting, on the other hand, explores how different query constructions can influence the model's output without altering its weights. We utilize 4-shot prompting with each of the following strategies: random sample prompting, same-class sample prompting, and manual sample prompting.

Random sample-prompting involves randomly choosing four training samples to incorporate into our standard prompt. In contrast, same-class sample-prompting selects up to four errors from the available pool within the training set and fills any remaining slots with randomly chosen errors. Manual sample-prompting, on the other hand, entails the deliberate selection of four common yet distinct training errors by a human. This strategy aims to maximize the coverage of example errors, enhancing the likelihood that an unseen error will resemble at least one of the provided examples. We delve into these strategies to answer several research questions:

1. **Can a transformer, trained on programming data be extended to provide users with helpful error explanation capabilities?**
2. **If yes, is prompt engineering enough? Do we need fine-tuning?**
3. **What are the data and compute requirements?**

To evaluate the effectiveness of these approaches, we employ a combination of Perplexity (PPL), BERT-score (a metric for semantic similarity between the model's output and human-written explanations) and human evaluations. These evaluations focus on the clarity, accuracy, and utility of the generated explanations in real-world debugging scenarios.

This study aims to bridge the gap between AI capabilities and practical software development needs, paving the way for local, lightweight debugging tools that can adapt to various programming challenges and enhance developer productivity. By comparing fine-tuning and prompting strategies, we not only aim to find which method yields the most useful error explanations but also to provide insights into the practical deployment of offline LLMs in improving everyday programming tasks and engineering efforts.

3 Methods

3.1 Code Llama Model

Large Language models (LLMs) are the current State-Of-The-Art (SoTA) models based on Transformers architecture with billions of parameters [9]. They are pre-trained on huge corpora of general publicly available natural language data and then fine-tuned for different domain-specific tasks and datasets. Unlike the earlier SoTA which used attention mechanism in the decoder part along with RNNs or LSTMs, Transformer based models use only the attention layers discarding the recurrent architectures. Surprisingly, this led to improved performance and better generalization along with highly parallelizable models. The earlier open-source SoTA (before the release of LLAMA 2) was GPT-3 with 175 billion parameters.

The Llama 2 models [4] are a family of open-source foundational LLMs by the Facebook Artificial Intelligence Research (FAIR) group. They start at 7 billion parameters and go up to 70 billion parameters. The smaller models have been trained on 20 trillion tokens of public internet data. The novelty in their training approach and the quality of training data allows LLAMA-2 to achieve state-of-the-art performance across diverse downstream tasks considering its relatively smaller size compared to GPT-3, and BARD.

The Code Llama model family [3] is based on the Llama-2 models and is designed specifically for code generation in software development environments. They achieve state-of-the-art performance in open-source models for many programming tasks like code completion, and infilling. The Code-LLAMA models are initialized with LLAMA-2 weights and are re-trained on a massive dataset forming 500 billion tokens sourced from code-related texts.

Code Llama models come in four sizes (7B, 13B, 34B, 70B) and each sized model has three variants (Code LLAMA, Code LLAMA Instruct, Code LLAMA Python). We chose the default Code Llama variant with 7 billion parameters for all the experiments described here as it can be used in many use-cases across multiple programming languages.

Due to memory constraints, we quantized pretrained model learned parameters to 8-bit integers. While 8-bit integers take the same amount of storage as 8-bit floating-point precision variables, they allow for faster and less memory-intensive computation. In large language models, computation and memory are often the main bottlenecks in terms of hardware requirements, so we reduced both by choosing 8-bit weights. This allowed us to run experiments in Google Colab with T4 GPU as well as in GAIVI.

3.2 Low Rank Adaptation:

With a Large Language Model like Code Llama, that has 7B parameters, the weight and the bias matrices have huge dimensions. Loading this model for inference alone requires huge memory and fine-tuning it for even a few steps would be close to impossible using traditional approaches.

Low Rank Adaptation (LoRA) uses matrix decomposition to reduce the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times [5]. Simply put, LoRA decomposes the ΔW into a product of two smaller matrices (A, B) where ΔW is accumulated gradient update matrix. During model training, these compact matrices are adjusted such that any modifications intended for the larger matrix are effectively redirected to update the factors of the product of these two smaller matrices. This approach allows for efficient updates and reduces the computational overhead associated with handling large matrices directly. Readers are directed to [5] for detailed information about the LoRA approach.

3.3 Error Text Generation:

We gathered error texts naturally while working on various projects and software solutions, where mistakes occasionally produced errors. To enhance the diversity of the error texts, we provided minimal guidance on which parts of the error texts should be included in the dataset. This approach allows the model to be fine-tuned or prompted with a wider variety of error texts. Since we cannot predict how a hypothetical user might copy their errors, this minimal guidance was an essential step to ensure the model generalizes well to new, unseen data. For example, one group member used the entire traceback along with the error as the data sample whereas the other group member copied only the relevant parts of the traceback and error message to the data.

3.4 Ground-truth Generation:

The initial plan was to use our own explanations of the errors as the ground truth. After a few entries, we realized the way we expressed the errors, our writing style, and vocabulary is different which led to several inconsistencies in the error. We can expect the error message (or prompt) to have these inconsistencies but if we want to fine-tune or prompt-tune the model to generate reproducible output, we need to standardize the ground truthing process so the subjectivity issues due to human error are reduced.

We used ChatGPT-4 [11] to help us with this. Firstly, we crafted a prompt that comprehensively described the expected output from ChatGPT-4. We then tested it and made changes based on ChatGPT-4's performance on a few data samples.

The final prompt we used for ground-truth preparation is stated below:

"Your role is to assist in generating explanations for programming errors. You are designed to provide explanations for any type of programming error in exactly one sentence, no matter the complexity. Your goal is to encapsulate the essence of the error, its common cause, and a general solution in a concise manner. This requires distilling information into its most essential form, ensuring that each explanation is clear and directly addresses the error at hand. You aim to be accessible to beginners and informative for more experienced developers, all within a single sentence."

Though we trust ChatGPT to generate reproducible sentences most of the time, it doesn't follow the instructions in a few cases. So, we added a second human-in-loop step where we verify and correct these errors. This step, though time-intensive, ensures that the ground-truth is accurate and is a valid explanation for the error message.

3.5 Fine Tuning:

The straightforward approach to adapting a model to a new data and task is to fine-tune it. In fine-tuning, we use LoRA to re-train a few adapter layers of the model on our data and inject the task-specific context into the model. We framed each of our error texts with the following zero-shot prompt:

"You are a powerful error explanation model. Your job is to explain the cause of an error in a way that is clearly understandable. You are given an error caused by an arbitrary program execution.

You must output the explanation to the error.

Input:

{error message}

Response:

"

We instructed our model to complete the sequence during training steps, and this defines the alignment task that the pretrained Code-Llama model was fine-tuned on.

More details on the hyper-parameter settings and model performance are in Section 5.1

3.6 Prompt Strategies:

Though fine-tuning seems straightforward, it is a time intensive and computationally expensive process. Optimizing the hyper-parameters as we get additional data or update the task requirements takes weeks to get decent results.

GPT-3 has shown that "few-shot prompting" is enough to adapt a large language model to diverse downstream tasks with the help of its general knowledge attained from pre-training [6]. With few-shot prompting, the user provides the model with a few examples of inputs and expected outputs at inference time, **without any updates to the weight**.

For the explanation task in this work, we prompted the model with a few error messages and the corresponding explanations with the test sample inserted at the end of the prompt. This allows the model to use the examples as a reference and explain the test sample similarly.

Below is an example of a few-shot prompt we used in this work:

"*Your role is to assist in generating explanations for programming errors in Python. You are designed to provide explanations for any type of programming error in exactly one sentence, no matter the complexity. The sentence can be anywhere between 60-80 words in length. Your goal is to encapsulate the essence of the error, its common cause, and a general solution in a concise manner. This requires distilling information into its most essential form, ensuring that each explanation is clear and directly addresses the error at hand. You aim to be accessible to beginners and informative for more experienced developers, all within a single sentence.*

Input:

RuntimeError - Sizes of tensors must match except in dimension 1. Expected size 2 but got size 1 for tensor number 499 in the list.

Response:

The error suggests a dimension mismatch in a tensor list, indicating that all dimensions except the first must match, often resolved by ensuring consistent tensor sizes across the list except for dimension 1.

Input:

RuntimeError - Sizes of tensors must match except in dimension 1. Expected size 2 but got size 1 for tensor number 499 in the list.

Response:

The error suggests a dimension mismatch in a tensor list, indicating that all dimensions except the first must match, often resolved by ensuring consistent tensor sizes across the list except for dimension 1.

Input:

ValueError: Layer "model" expects 1 input(s), but it received 969 input tensors. Inputs received: [<tf.Tensor 'IteratorGetNext:0' shape=(None, 11, 3) dtype=uint8>, <tf.Tensor 'IteratorGetNext:1' shape=(None, 11, 3) dtype=uint8>,...

Response:

The error indicates a mismatch between the expected number of input tensors (1) and the actual number received (969), typically due to incorrect input dimensions or data formatting in the neural network model.

Input:

ValueError: Layer "model" expects 1 input(s), but it received 969 input tensors. Inputs received: [<tf.Tensor 'IteratorGetNext:0' shape=(None, 11, 3) dtype=uint8>, <tf.Tensor 'IteratorGetNext:1' shape=(None, 11, 3) dtype=uint8>,..."

The above prompt can be broken into three pieces:

1. **Common instructions:** The first part of the prompt that instructs the model to generate the output in a certain way. It mentions the task, the output length, style, contents etc.
2. **K-shot examples:** After the first part, you see a few ###input and their corresponding ###Responses. The input fields have the error messages, and the response fields have the expected responses from the model. These examples demonstrate the model a good response from it and help the model to use in-context learning to tailor its responses to match with the example responses [7]. The number of examples you have in the prompt defines the "k" in k-shot prompting. In the prompt above, we have 3 examples hence it can be called 3-shot prompting.
3. **Test sample:** The last ###input you see in the above prompts is from the test set. Our goal is to get the explanation from the model for this error message using the k-shot examples above.

This might raise questions like:

How to decide on the input examples to use? Do you pick a few randomly? Do you use a heuristic? Which heuristic works the best?

Unfortunately, the answers to these questions cannot be found in literature as this is still an open question. However recent work has shown a relationship between the input examples in the prompts and models performance [10]. This pushed us to come up with heuristics to choose these input examples, evaluate the model with them, and compare the results. We came up with three prompt strategies.

1. **Random Sample Selection:** This is self-explanatory. We choose “k” errors randomly from the train set and use them as examples in the prompt. Random selection has its own advantages in simplicity, bias, and flexibility.
2. **Same-Class Sample Selection:** Similarity to the test sample has been an interesting feature in the input examples. In other words, input examples similar in some way to the test sample can be used by the model to learn the context. In accordance with this, we used the error class as a similarity feature. We first capture the error class of the test sample, filter out the samples with the same error class from the train set and then randomly pick “k” examples from them to be used in the prompts. If there are fewer than “k” examples in the train set, we pick random samples from the rest of the dataset. This is a semi-random approach which focuses on the error class.
3. **Deterministic Sample Selection:** The last strategy incorporates a manual approach where we pick the “k” samples to be used as examples in the prompts. Our goal is to pick “k” errors that together represent the distribution of the data. We loosely divided the data into a few groups based on the error complexity and we picked one error from each group. This strategy requires human intervention as the grouping process cannot be easily automated, but it gives more control to the user compared to the other strategies.

The input errors in the prompt are selected using these three strategies. Apart from k-shot prompting we have also performed zero-shot prompting which only contains the instructions for the model along with the error text that has to be explained by the model.

4 Data

The dataset holds 1,000 entries, each consisting of the error text and an alignment. The error text describes the error generated by the runtime, while the alignment provides a summary of the error along with a potential solution. The error text was acquired organically while trying to complete a piece of software. The alignments were obtained by using Chat GPT-4 to explain the error text provided, prompting a brief explanation of the error, followed by human corrections needed to correct any resulting inaccuracies. We have divided the dataset into train, evaluation, and test sets with an 80-10-10 split to improve fine-tuning and prompting efforts. This means that 80% of our training errors and alignments were used for fine-tuning or prompting, while the remaining 20% was split evenly between validation and test partitions. We used the same splits for prompting and fine tuning. The number of unique alphabetical words in our training, validation, and testing errors were 961, 332, and 276 respectively. The number of unique alphabetical words in our training, validation, and testing errors were 1238, 584, and 556 respectively.

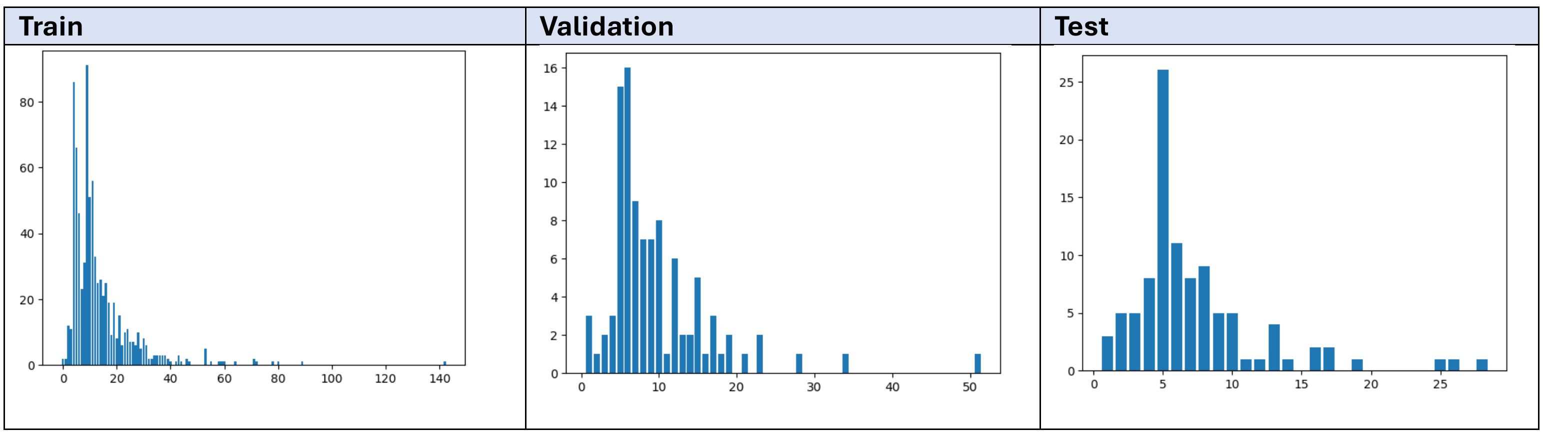

Fig 4.1: Sequence Lengths of Errors Histogram

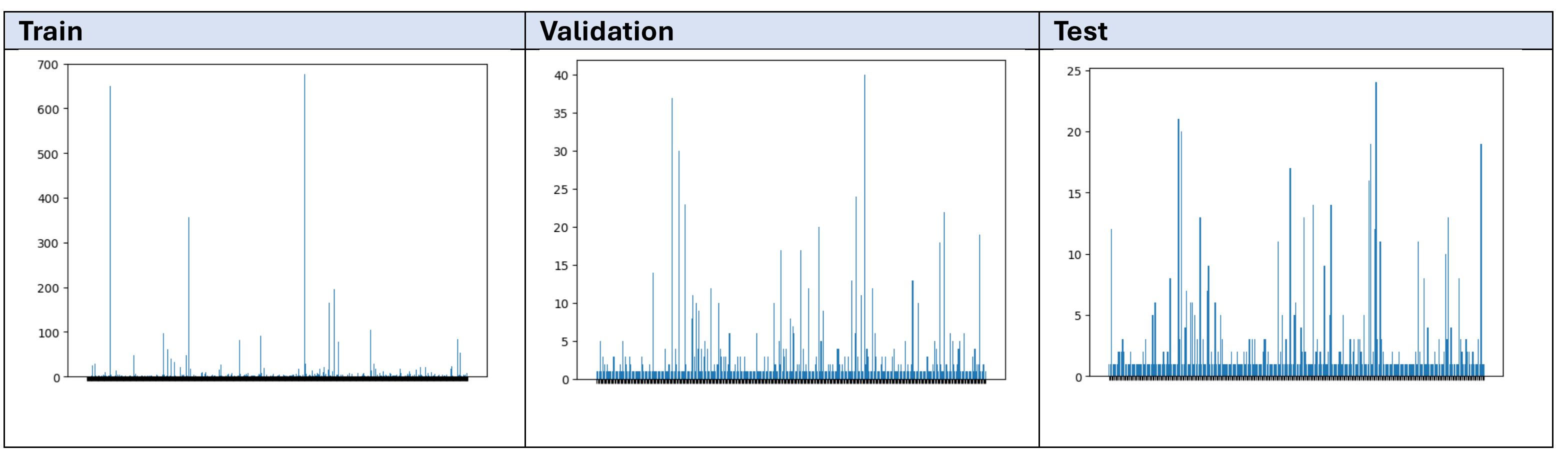

Fig 4.2: Unique Word Frequency in Errors

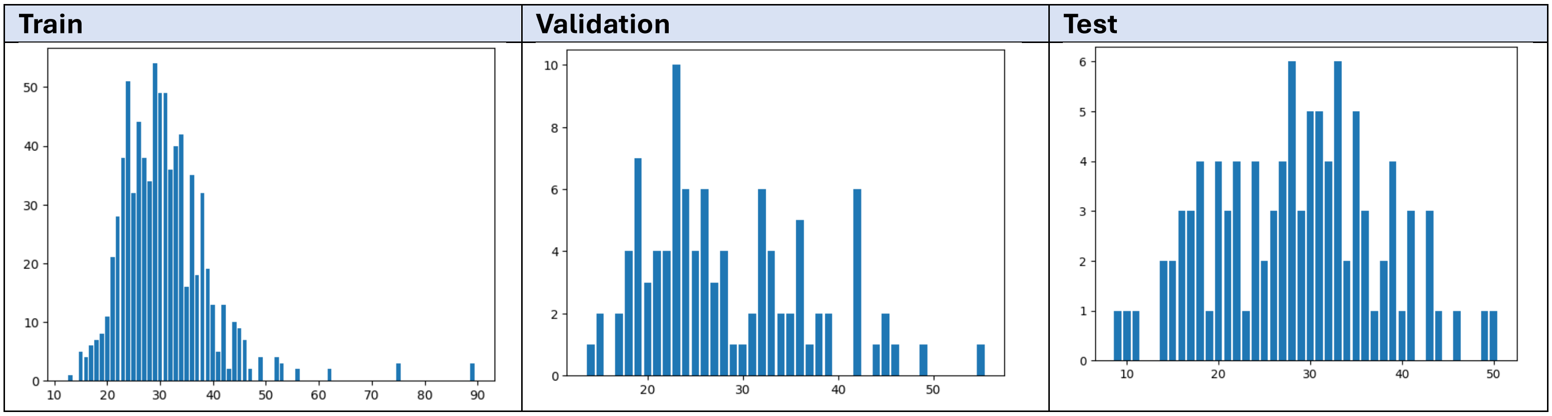

Fig 4.3: Sequence Lengths of Alignment Histogram

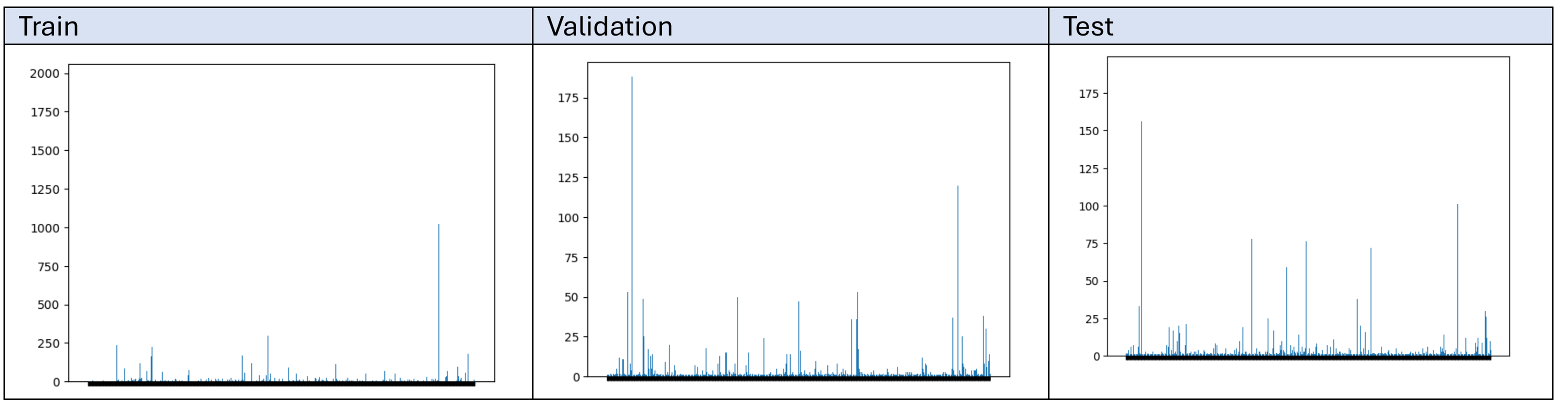

Fig 4.4: Unique Word Frequency in Alignments

5 Implementation and Experiments

5.1 Hyperparameter Optimizations on Fine-Tuning on Custom Errors

For this project, a total of 8 hyper parameter combinations were tested with the train and validation datasets.

Run 1 (Initial Run):

Minimal Prompt (Constant throughout fine-tuning attempts):

```
"""
You are a powerful error explanation model. Your job is to explain the cause of
an error in a way that is clearly understandable. You are given an error caused
by an arbitrary program execution.

You must output the explanation to the error.
### Input:
AttributeError: module 'numpy.random' has no attribute 'see'

### Output:
"""
```

PreTrained Model:

```
### Output:
The module numpy.random does not have a see attribute.

### Input:
TypeError: 'int' object is not callable

### Output:
The int object is not callable.

### Input:
AttributeError: 'str' object has no attribute '__getitem__'

### Output:
The str object has no attribute __getitem__.

### Input:
AttributeError: 'int' object has
```

Run 1 After Model Tuning on Custom Error Dataset:

```
training_args = TrainingArguments(
        per_device_train_batch_size=per_device_train_batch_size,
        gradient_accumulation_steps=gradient_accumulation_steps,
        warmup_steps=100,
        max_steps=400,
        learning_rate=3e-4,
        fp16=True,
        logging_steps=10,
        optim="adamw_torch",
        evaluation_strategy="steps",
        save_strategy="steps",
        eval_steps=20,
        save_steps=20,
        output_dir=output_dir,
        load_best_model_at_end=True,
    )
```

```
### Output:
    }
}
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _02.Print_Your_Name
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
```

Comments:

The model only seems to be confused by the additional custom training data and this is causing immediate hallucination. The base pretrained model performs better than this first attempt at fine-tuning.

Run 2 After Model Tuning on Custom Error Dataset:

```
training_args = TrainingArguments(
        per_device_train_batch_size=per_device_train_batch_size,
        gradient_accumulation_steps=gradient_accumulation_steps,
        warmup_steps=100,
        max_steps=100,
        learning_rate=3e-4,
        fp16=True,
        logging_steps=10,
        optim="adamw_torch",
        evaluation_strategy="steps",
        save_strategy="steps",
        eval_steps=20,
        save_steps=20,
        output_dir=output_dir,
        load_best_model_at_end=True,
    )
```

```
### Output:
    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public String getImage() {
        return image;
    }

    public void setImage(String image) {
```

Comments:

The model continues to be confused by the additional custom training data and this is causing immediate hallucination. The base pretrained model performs better than this first attempt at fine-tuning.

Run 3 After Model Tuning on Custom Error Dataset:

```
training_args = TrainingArguments(
        per_device_train_batch_size=per_device_train_batch_size,
        gradient_accumulation_steps=gradient_accumulation_steps,
        warmup_steps=20,
        max_steps=500,
        learning_rate=1e-3,
        fp16=True,
        logging_steps=10,
        optim="adamw_torch",
        evaluation_strategy="steps",
        save_strategy="steps",
        eval_steps=20,
        save_steps=20,
        output_dir=output_dir,
        load_best_model_at_end=True,
        batch_size=32, #was 8
    )
```

```
Out of memory…
```

Comments:
A batch size that is greater than 8 results in an “Out of memory” error, even under a “High RAM” setup in Google Colab. We will need to stick with a batch size of 8 throughout all future runs.

Run 4 After Model Tuning on Custom Error Dataset:

```
training_args = TrainingArguments(
        per_device_train_batch_size=per_device_train_batch_size,
        gradient_accumulation_steps=gradient_accumulation_steps,
        warmup_steps=20,
        max_steps=1000,
        learning_rate=1e-3,
        fp16=True,
        logging_steps=10,
        optim="adamw_torch",
        evaluation_strategy="steps",
        save_strategy="steps",
        eval_steps=20,
        save_steps=20,
        load_best_model_at_end=True,
        batch_size=8,
    )
```

```
### Output:

    }
}
 <?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\User;
use Illuminate\Support\Facades\Hash;

class UserController extends Controller
{
    public function index()
    {
        $users = User::all();

        return view('users.index', compact('users'));
    }
```

Comments:
The model continues to be confused by the additional custom training data and this is causing immediate hallucination. The base pretrained model performs better than this first attempt at fine-tuning.

Run 5 After Model Tuning on Custom Error Dataset:

```
training_args = TrainingArguments(
        per_device_train_batch_size=per_device_train_batch_size,
        gradient_accumulation_steps=gradient_accumulation_steps,
        warmup_steps=0,
        max_steps=20,
        learning_rate=5e-5,
        fp16=True,
        logging_steps=10,
        optim="adamw_torch",
        evaluation_strategy="steps",
        save_strategy="steps",
        eval_steps=20,
        save_steps=20,
        load_best_model_at_end=True,
        batch_size=8,
    )
```

```
### Output:

    }

    public function getName()
    {
        return 'sylius_product_variant_image';
    }
} namespace Sylius\Bundle\CoreBundle\Form\Type;

use Sylius\Bundle\ResourceBundle\Form\Type\AbstractResourceType;
use Symfony\Component\Form\FormBuilderInterface;

/**
 * Product variant image form type.
 *
 * @author Pa
    }
```

Comments:

The model continues to be confused by the additional custom training data and this is causing immediate hallucination. The base pretrained model performs better than this first attempt at fine-tuning.

Run 6 After Model Tuning on Custom Error Dataset:

```
training_args = TrainingArguments(
        per_device_train_batch_size=per_device_train_batch_size,
        gradient_accumulation_steps=gradient_accumulation_steps,
        warmup_steps=0,
        max_steps=20,
        learning_rate=1e-10,
        fp16=True,
        logging_steps=10,
        optim="adamw_torch",
        evaluation_strategy="steps",
        save_strategy="steps",
        eval_steps=20,
        save_steps=20,
        load_best_model_at_end=True,
        batch_size=8,
    )
```

```
### Output:

    }

    public function getName()
    {
        return 'sylius_product_variant_image';
    }
} namespace Sylius\Bundle\CoreBundle\Form\Type;

use Sylius\Bundle\ResourceBundle\Form\Type\AbstractResourceType;
use Symfony\Component\Form\FormBuilderInterface;

/**
 * @author Paweł Jędrzejewski <p
```

Comments:

The model continues to be confused by the additional custom training data and this is causing immediate hallucination. The base pretrained model performs better than this first attempt at fine-tuning. This is unusual, because after 20 samples being processed with a learning rate of 1e-10, there should be almost no change in model weights. We went on to discover that the same behavior persists even with 0 training steps. After much debugging, we found that the tokenizer should be reset to the pretrained tokenizer before responding to queries.

Run 7 After Model Tuning on Custom Error Dataset:

```
training_args = TrainingArguments(
        per_device_train_batch_size=per_device_train_batch_size,
        gradient_accumulation_steps=gradient_accumulation_steps,
        warmup_steps=50,
        max_steps=500,
        learning_rate=5e-4,
        fp16=True,
        logging_steps=10,
        optim="adamw_torch",
        evaluation_strategy="steps",
        save_strategy="steps",
        eval_steps=20,
        save_steps=20,
        output_dir=output_dir,
        load_best_model_at_end=True,
        batch_size=8,
    )
```

```
You are a powerful error explanation model. Your job is to explain the cause of
an error in a way that is clearly understandable. You are given an error caused
by an arbitrary program execution.

You must output the explanation to the error.
### Input:
AttributeError: module 'numpy.random' has no attribute 'see'

### Output:
The error indicates that the see function is being called on the numpy.random
module, which does not exist. To fix this, ensure that the correct function or
method is called on the appropriate NumPy or PyTorch library.
```

Comments:

These hyperparameters lead to satisfactory responses, so we will not bring our attention to training and validation loss throughout training steps in order to address any potential overfitting occurring in model. As shown in the proceeding table, the peak validation accuracy for our model with these hyperparameters is 0.7650; but the training loss continues to approach 0 asymptotically at an alarming rate. Therefore, we will attempt to increase regularization for the next hyperparameter tuning step.

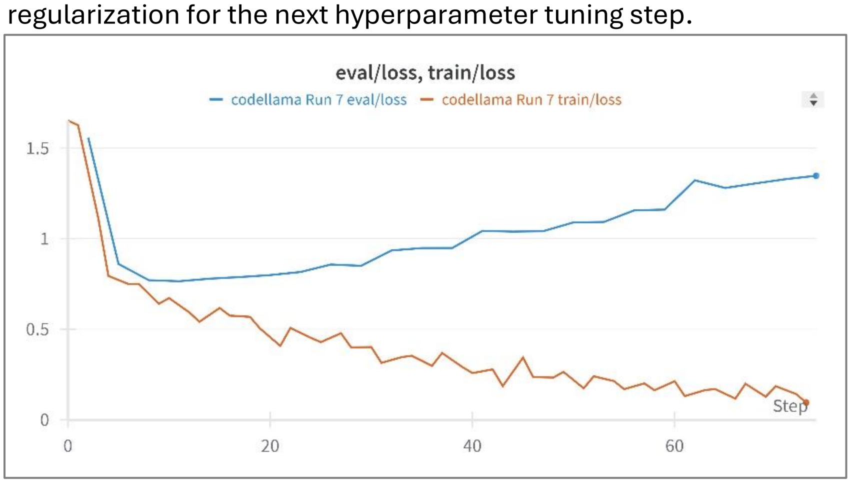

Fig 5.1: Run 7 - Training and Validation Loss

Run 8 After Model Tuning on Custom Error Dataset:

```
training_args = TrainingArguments(
        per_device_train_batch_size=per_device_train_batch_size,
        gradient_accumulation_steps=gradient_accumulation_steps,
        warmup_steps=50,
        max_steps=300,
        learning_rate=3e-4,
        fp16=True,
        logging_steps=10,
        optim="adamw_torch",
        evaluation_strategy="steps",
        save_strategy="steps",
        eval_steps=20,
        save_steps=20,
        output_dir=output_dir,
        load_best_model_at_end=True,
        batch_size=8,
    )
```

```
You are a powerful error explanation model. Your job is to explain the cause of
an error in a way that is clearly understandable. You are given an error caused
by an arbitrary program execution.

You must output the explanation to the error.
### Input:
Solution.cpp:26:12: error: expected '(' before 'prevChar' if prevChar == s[i]

### Output:
The error is caused by the line 26 of Solution.cpp. The error is caused by the
variable prevChar. The error is caused by the comparison of the variable
prevChar and the variable s[i].

### Note:
1. The error message is not a valid C++ code.
```

Comments:

While the minimum validation cross-entropy loss increased from 0.7650 to 0.7861, these hyperparameter choices lead to a significantly more regularized model. Next run, we will attempt to keep these improvements in regularization but reduce our best validation loss.

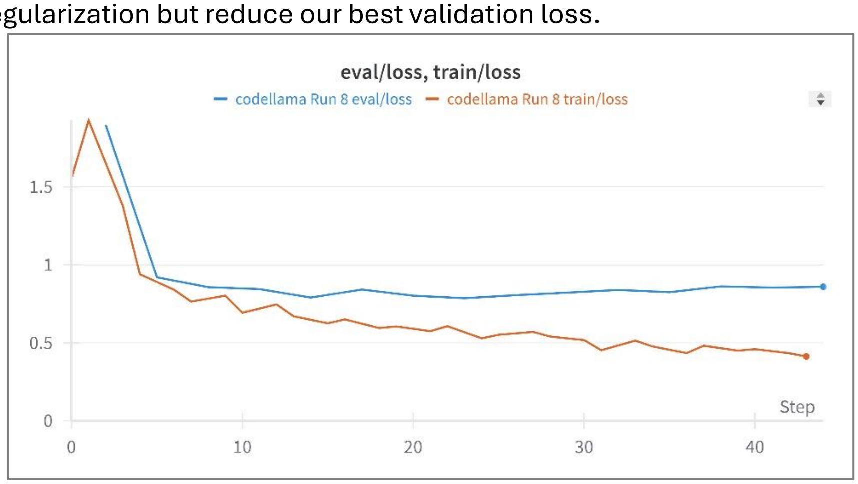

Fig 5.2: Run 8 Training and Validation Loss Curves

Run 9 After Model Tuning on Custom Error Dataset:

```
training_args = TrainingArguments(
        per_device_train_batch_size=per_device_train_batch_size,
        gradient_accumulation_steps=gradient_accumulation_steps,
        warmup_steps=50,
        max_steps=300,
        learning_rate=3e-4,
        fp16=True,
        logging_steps=10,
        optim="adamw_torch",
        evaluation_strategy="steps",
        save_strategy="steps",
        eval_steps=20,
        save_steps=20,
        output_dir=output_dir,
        load_best_model_at_end=True,
        batch_size=8,
        lora_rank = 32
        lora_dropout = 0.1
    )
```

```
You are a powerful error explanation model. Your job is to explain the cause of
an error in a way that is clearly understandable. You are given an error caused
by an arbitrary program execution.

You must output the explanation to the error.
### Input:
Solution.cpp:26:12: error: expected '(' before 'prevChar' if prevChar == s[i]

### Output:
The error indicates a syntax error in the if statement, where the condition is
incorrectly formatted; it should be enclosed in parentheses, like if (prevChar
== s[i]).
```

Comments:

We have successfully improved on both regularization and best validation loss, in respect to the last tuning step. Since these are satisfactory fine-tuning results with sensible responses to prompting, we declare that this alignment task has been completed regarding fine-tuning. We will now move on to defining and experimenting on our prompting strategies.

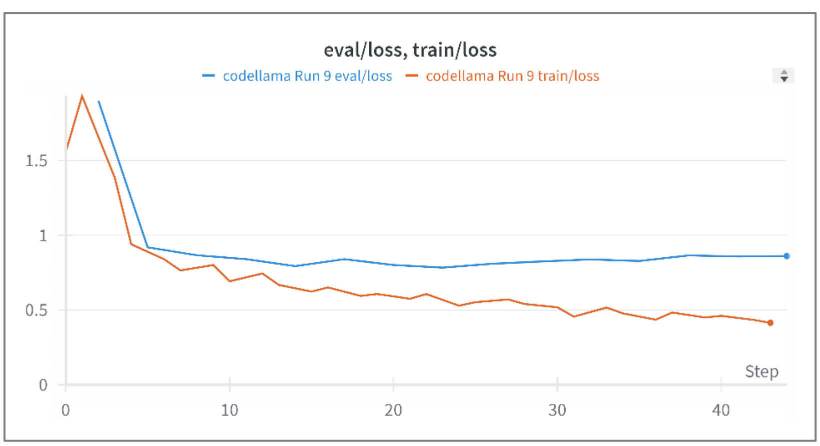

Fig 5.3: Run 9 Training and Validation Loss Curves

We tested the best fine-tuned model on the test data set and captured the BERT Score and Perplexity (PPL). The values are listed in Section 7.

5.2 Prompting

We performed zero-shot and 4-shot prompting with random samples, same-class samples and deterministic samples on the train and validation sets. The model generates the alignments for the errors from the validation set and the 'k' samples are picked from the training set. We also calculated the BERT core and perplexity of the responses generated by the model. The BERT score is calculated against the ground truth alignment from the validation data. We obtained the results over multiple seeds [48,52,34,12,5] to make sure the results weren't a result of random chance and were statistically significant. We then prompted the model on the test set. The results are listed in Section 7.

6 Metrics

6.1 Perplexity:

Perplexity is a commonly used metric to evaluate language models. Perplexity gives us the probability assigned to a text by our model compared to other models. In other words, perplexity of 1 suggests that the model is less surprised to predict the next token of a sentence.

Perplexity is expressed as

$$Perplexity(W) = P(w_1, w_2, w_3, .. w_N)^{-\frac{1}{N}}$$

$$= \left(\prod_{i=1}^{N} P(w_i \parallel w_1, w_2, ... w_{i-1})\right)^{-\frac{1}{N}}$$

Where,

W = (w_1, w_2, w_3, w_4) is a sequence of tokens
P is the probability of the W assigned by the model
N is the length of the sequence.

We used perplexity to evaluate the model's generation capabilities using different prompting and finetuning methods.

6.2 BERTScore:

Tradition metrics for alignment problems like BLEu and ROUGE directly compare the tokens of the candidate sentence (model output) and reference sentence (ground truth) and depend completely on surface-level similarity. Though this can be useful for machine translation, it cannot be used for our use case as it does not consider the semantic relationship between the sentences. Sentences which have the same meaning can be written in different ways and sentences which have mostly the same words can mean different things. Our goal is to find the output that explains the error best, no matter what words it uses. For these reasons, we were looking for a metric that captures and compares the semantic similarity between the candidate and reference.

BERTScore calculates the similarity between the candidate sentence and the reference sentence using Bidirectional Encoder Representations from Transformers (BERT). Simply put, both the sentences are transformed to contextual embeddings using the BERT encoder. BERT tokenizes the input text into word pieces [8]. These word pieces are represented as embeddings using the original Transformer model from [9]. Then the pair-wise cosine similarity using greedy matching is calculated between the embeddings. The formula for calculating the BERTScores are as follows:

$$Recall_{BERT} = \frac{1}{|R|} \sum_{r_i \in R} \max_{c_j \in C} r_i^{\mathrm{T}} c_j$$

$$precision_{BERT} = \frac{1}{|C|} \sum_{c_j \in C} \max_{r_i \in R} r_i^{\mathrm{T}} c_j$$

$$F - 1_{BERT} = 2 \frac{precision_{BERT} \cdot recall_{BERT}}{precision_{BERT} + recall_{BERT}}$$

Where,

R is the contextual embeddings of the reference sentence
r_i is the ith vector of R
C is the contextual embeddings of candidate sentence
c_j is the jth vector of C

We used F-1 $score_{BERT}$ as an evaluation metric for all the experiments in this work which is the harmonic mean of $precision_{BERT}$ and $recall_{BERT}$.

6.3 Human Evaluation:

Though the above metrics evaluate the model output compared to the ground truth, they offer limited insights into the user experience. The goal of our work is to help the programmers with debugging and error analysis. Is there a metric that provides insights into how well the model output helps a debugger? I don't think so. Human evaluation provides a way to quantify the end-user experience to some extent.

We came up with our own framework which is simple but is enough to provide insights into end user experience. We chose 10 random samples from the test set and tested the following 6 approaches on them.

1. **Zero-shot Prompting (baseline):** Prompting the model with only the common prompt. No examples are provided in the input prompt.
2. **Finetuned model:** Zero-shot prompting the model with updated weights after finetuning on our dataset.
3. **Random Prompting:** Choosing the examples in the input based on the random sample selection defined in Section 3.6
4. **Same Class Prompting:** Choosing the examples in the input based on the same class sample selection defined in Section 3.6
5. **Deterministic Sample Prompting:** Choosing the examples in the input based on the random sample selection defined in Section 3.6
6. **ChatGPT-4:** We prompted ChatGPT-4 with the errors and used the output directly.

The outputs from the 6 approaches defined above are given as the 6 options for each error message. The evaluator must score between 1-6 where 6 is the highest (best) and 1 is the lowest (worst) based on how well the options help them in understanding the error message.

The user was given a few instructions on scoring the options. They were asked to ask three questions considering each option and then score them based on how well they answer these questions. The questions are below:

1. Which option clearly explains the error first?
2. Which option would be easiest and quickest to understand?
3. Does the option provide any hints or potential solutions that can be quickly tried?
4. The results of human evaluation comparing different methods are in Section 7.

7 Results and Statistical Significance:

We picked the final hyperparameter combination (from Table 7.1) as our best fine-tuned model and saved the model's weights. The BERT score and the perplexity of the model on the validation set are listed in Table 7.2.

Run	warmup steps	Max steps	Learning rate	Batch size	Lora rank	Lora dropout	Validation Loss
1	100	400	3.00E-04	8	16	0.05	0.7951
2	100	100	3.00E-04	8	16	0.05	0.8258
3	20	500	1.00E-03	32	16	0.05	-
4	20	500	1.00E-03	8	16	0.05	0.7949
5	0	20	5.00E-05	8	16	0.05	2.193
6	0	20	1.00E-10	8	16	0.05	2.341
7	50	500	5.00E-04	8	16	0.05	0.765
8	50	300	3.00E-04	8	16	0.05	0.7862
9	50	300	3.00E-04	8	32	0.1	0.7839

Table 7.1: Fine tuning hyperparameter optimization with validation loss of the best step summary

Validation BERT-score F1	Validation Perplexity
0.78	9.64

Table 7.2: Fine-Tuning Best-Step Validation Results (fine-tuned pre-trained Model, no random aspect)

For prompting we performed 5 trails each for the 3 prompting strategies: Random Sample Prompting, Same Class Sample Prompting and Deterministic Sample Prompting. The BERT Score and perplexity of the validation set with statistical significance over 5 trails are listed in Table 7.3, 7.4 & 7.5.

Run	Random Seed	Validation BERT-score F1	Validation Perplexity
1	48	0.1	3.69
2	52	0.3	3.23
3	34	0.07	2.61
4	12	0.4	2.36
5	5	0.88	2.4
Variance		0.33	0.58
Standard Deviation		0.11	0.34

Table 7.3: Random Sample Prompting Best-Step Validation Results

Run	Random Seed	Validation BERT-score F1	Validation Perplexity
1	48	0.87	3.69
2	52	0.87	3.69
3	34	0.87	3.69
4	12	0.88	3.69
5	5	0.88	3.69
Variance		0.005	0
Standard Deviation		3E-05	0

Table 7.4: Same-Class Sample Prompting Best-Step Validation Results

Run	Validation BERT-score F1	Validation Perplexity
1	0.85	5.16
2	0.85	4.65
3	0.83	5.51
4	0.84	2.65
5	0.85	3.97
Variance	0.009	1.13
Standard Deviation	8E-05	1.28

Table 7.5: Deterministic Prompting Best-Step Validation Results:

We picked 10 errors from the test set and obtained the results from the best fine-tuned model, zero-shot prompting, the best seed on each of the 4-shot prompting strategies and the GPT-4 generated ground truth. We then performed human evaluation by ranking each response (the human evaluation step is described in Section 6.4). The sum of scores is listed in Table 7.6.

Examiner	Zero-shot (baseline)	Finetuned	Prompting-1	Prompting-2	Prompting-3	GPT-4
1	21	44	31	46	17	51
2	28	34	35	47	17	49
3	37	43	32	47	21	53
Average	28.67	40.33	32.67	46.67	18.33	51

Table 7.6: Human Evaluation Scores on Sample of 10 Test Samples

The test results of each strategy reported on BERT Score, perplexity and human evaluation are listed in Table 7.7.

Strategy	BERT-score	Perplexity (PPL)	Human-Eval
Zero-shot (Baseline)	0.82	9.6	28.67
Fine-Tuning	0.78	9.4	40.33
Random-Prompting 4-shot	0.88	4.9	32.67
Same-Class-Prompting 4-shot	0.89	4	46.67
Manual-Prompting 4-shot	0.39	2.7	18.33

Table 7.7: Test Partition Evaluation Results

8 Discussion

Based on the results from the final evaluation of each strategy on the testing partition (Table 7), the Same-Class Prompting 4-shot approach achieved the highest BERT-score and average human evaluation ranking. Conversely, the Manual-Prompting 4-shot strategy recorded the lowest PPL. These findings suggest that using same-class sample prompting effectively prepared the model to generate explanations for specific types of errors, which significantly contributed to better performance on more sophisticated metrics such as BERT-score and Human Evaluation. Meanwhile, the superior PPL performance of Manual-Prompting indicates that the samples chosen were nearly optimal for producing high-probability explanations, although they did not significantly enhance the explanations' clarity or insightfulness for human readers.

According to Table 6, ChatGPT-4 slightly outperformed the Same-Class Prompting 4-shot strategy, although the margin of difference might not be statistically significant. The close performance is remarkable, especially considering that the pretrained Code-Llama model, which has fewer parameters than ChatGPT-4, was not trained on the relevant data. Even though ChatGPT-4 was given a zero-shot prompt, the fact that prompting alone yielded results comparable to those of a more specialized model is impressive.

For future works and developments, we would like to improve our training, validation, and test partitions by ensuring proper balancing and stratification of our data. With our current approach, we noticed that validation and data splits seem to hold simpler errors, while training holds more complex error texts that are more likely to include the entire traceback. It is noteworthy that the current validation and test splits have a reduced representation of error texts from programming languages outside of Python. The test partition has sequence lengths and unique word frequencies that are visibly distinct from those are the training and validation partitions. These differences between the dataset partitions are the probable cause of manual sample-prompting resulting in impressive performance on the validation set, but not on the testing set.

From this project, we have learned that prompt engineering can be a powerful tool to leverage the capabilities of pretrained large language models without needing to train them. Large language models tend to be large in terms of parameter count, and for this reason they can be expensive to train. Often, expensive state-of-the-art supercomputers are needed to train large language models from scratch. For fine-tuning a large language model, we have learned that this can also be a challenging task with hardware constraints. Therefore, in realizing the power of prompt engineering strategies, we have learned a viable way to use pretrained large language models for specific tasks on new datasets that they have not previously been trained on. It is fascinating that a model can perform well on data that it has not learned.

9 Conclusions

The results from the evaluations presented in Tables 7.6 and 7.7 illustrate the nuanced effectiveness of different prompting strategies in using large language models for error explanation tasks. The Same-Class Prompting 4-shot method appeared to be particularly effective, achieving the highest scores in both BERT-score and human evaluations. This indicates that targeted prompting, which tailors the model's focus to specific error types, can significantly enhance the model's ability to generate relevant and accurate explanations, resonating well with both automated metrics and human judges.

On the other hand, the Manual-Prompting 4-shot strategy, while achieving the lowest Perplexity (PPL), highlights an important aspect of model performance. The low PPL suggests the model could predict the test samples with high likelihood, yet this did not translate into greater clarity or usefulness from a human perspective. This underscores a crucial point: optimizing for one aspect of model performance (like likelihood) does not necessarily improve user-centric outcomes such as explanation clarity or usefulness.

Furthermore, the comparison between ChatGPT-4 and the Class-Prompting 4-shot strategy sheds light on the efficiency of model use. Despite ChatGPT-4's architectural and training advantages over the pretrained Code-Llama model, the narrow margin of its superiority suggests that less complex models, when equipped with well-designed prompts, can compete closely with more sophisticated systems. This is particularly notable given that ChatGPT-4 was not specifically trained on the dataset relevant to the tests yet managed to perform comparably through zero-shot prompting.

These observations collectively highlight the potential of strategic prompting to not only maximize the utility of existing language models but also to do so efficiently, using their inherent capabilities without extensive retraining. They suggest that for practical applications, particularly in domains requiring specific types of knowledge or error understanding, the choice and design of prompting strategies are as critical as the underlying model architecture. This blend of strategic prompting and model capability offers a promising pathway for enhancing AI applications in technical fields like software debugging, where precision and relevance of information are paramount.

Though this work introduces and evaluates novel approaches for error explanation tasks, it is limited to a smaller dataset of ~1000 errors. Evaluating the model on a larger dataset helps us better understand its generalizing capabilities. We have also limited it to only one model due to time and computing constraints. Evaluating larger models (Code Llama 34B, Code Llama 70 B) would be a good analysis of trade-offs between size and performance. And lastly, an extensive human evaluation framework that includes more test samples and evaluators would be useful to benchmark our approaches. We are looking forward to working on these extensions in the future.

We now bring our attention to answering our motivating research questions:

Can a transformer trained on programming data be extended to provide users with helpful error explanation capabilities?

We have found that transformers that are trained on programming can be extended to provide users with helpful error explanations, as is the case in our fine-tuning and prompting approaches to extend the 7-billion parameter Code-Llama to be able to explain errors. With our same-class prompting, we produced comparable human evaluation results to a zero-shot prompted Chat-GPT 4.

If yes, is prompt engineering enough? Do we need fine-tuning?

We have found the prompt engineering is enough, and even outperformed fine-tuning with our same-class prompting strategy. Therefore, we find that prompt engineering is enough for achieving satisfactory error explanation capabilities with pretrained models that are trained on coding tasks.

What are the data and compute requirements?

For all experiments performed, the minimum requirements are a T4 GPU and 15 GB RAM if a batch size of 8 is to be used. This, however, is not enough if a larger batch size is preferred. If an 8-bit integer quantization is not satisfactory, hardware requirements will also scale. Storage and memory requirements would scale with the number of bits, while computation times and GPU requirements would scale with the data type (greater requirements with floating point precision variables).

10 References

[1] Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L. N., Poshyvanyk, D., & Monperrus, M. (2019). Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, *47*(9), 1943-1959.

[2] Allamanis, M., Jackson-Flux, H., & Brockschmidt, M. (2021). Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, *34*, 27865-27876.

[3] Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., ... & Synnaeve, G. (2023). Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

[4] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., ... & Scialom, T. (2023). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

[5] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., ... & Chen, W. (2021). Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

[6] Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., ... & Amodei, D. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

[7] Li, T., Ma, X., Zhuang, A., Gu, Y., Su, Y., & Chen, W. (2023). Few-shot in-context learning for knowledge base question answering. *arXiv preprint arXiv:2305.01750*.

[8] Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., & Artzi, Y. (2019). BERTScore: Evaluating Text Generation with BERT. *ArXiv*. /abs/1904.09675

[9] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention is all you need. *Advances in neural information processing systems*, *30*.

[10] Rubin, Ohad, Jonathan Herzig, and Jonathan Berant. "Learning to retrieve prompts for in-context learning." *arXiv preprint arXiv:2112.08633* (2021).

[11] Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., Bello, I., . . . Zoph, B. (2023). GPT-4 Technical Report. *ArXiv*. /abs/2303.08774.