DevOops: Speeding Up Debugging from Logs

Final Report

Mondo Jiang
University of Wisconsin, Madison
`mondo.jiang@wisc.edu`

Daniel Smedema
University of Wisconsin, Madison
`djsmedema@wisc.edu`

Abstract

In the modern software development landscape, complex tech stacks and automated CI/CD pipelines present unique challenges in error identification and debugging. We explore the potential of leveraging large language models to interpret log streams and assist in debugging, addressing the inefficiencies of current manual methods. We propose a fully-local, fine-tuned language model to identify errors within log texts, preserving proprietary information and providing context-specific insights. We utilize data from the open-source BugSwarm dataset to fine-tune a Llama-3.2 3B model using LoRA. We do not show notable improvement in model performance after fine-tuning. Ultimately, we conclude that alternative approaches will likely be more practical.

1 Introduction

In the modern era, software applications are developed using complex "tech stacks" that feature multiple interacting components, and these are often deployed into a production application using another automated process, often called a CI/CD pipeline (Continuous Integration/Continuous Deployment). This pipeline itself is code that may fail, and its failure could be due to problems in the pipeline code or problems one or more of the system's constituent components. The error logs generated by these failures can be difficult to read and reason about.

There is often no way to test whether you have fixed a pipeline error other than running the pipeline again, and this may take minutes or even tens of minutes even for relatively small applications. A human troubleshooter will likely not be able to spend these minutes spent waiting on pipeline runs productively on other tasks. Thus it is costly to be wrong about what the actual error is.

With the advent of large language models, the possibility of automated assistance in interpreting semi-structured and dynamic log streams is on the horizon. It is already possible to copy and paste your logs into ChatGPT and ask it questions, often receiving fairly helpful results. However, there are a few downsides to this: a) it may leak your proprietary information to third parties, and b) ChatGPT does not have the full context of your codebase, which may be necessary to properly contextualize the errors.

A fully-local language model (LM) avoids the first issue, and fine-tuning it on a single specific codebase addresses the second. In order for this to be economically viable for a small organization, the model would need to be small and based on a pretrained open-source model. For example, the recently released Llama 3.2 with 1B parameters is advertised as fitting onto edge and mobile devices.

As a first step towards generating natural-language "advice" on how to fix issues or even automatically generating fix code, the model must be able to identify the error from within the log text. This will be our focus in this project.

Our core research questions are:

- RQ1: Can a local version of a (consumer grade) open-source LM be used to identify errors?
- RQ2: Can fine-tuning the LM on build log datasets increase its usefulness?

Our general approach is to take a consumer-grade, open-source LM, fine-tune it on open-source data using consumer-grade hardware, and compare the performance of the base model with the fine-tuned model on a downstream evaluation task that asks the LM to essentially summarize a DevOps log by repeating back only the section relevant to the error. We reason that the first step towards machine-assisted solutions is identification of the problem.

2 Related Work

Many approaches have been proposed for various code-related tasks. Q. Sun *et al.* [1] provided a comprehensive summary of code intelligence, where most efforts focus on building connections between code-to-code and text-to-code generation, enhancement, and understanding. However, there are no well-defined tasks for handling logs generated during build and deployment pipelines.

C. E. Jimenez *et al.* [2] defined a benchmark to test the code auto-fixing capabilities of models, while C. Le Goues *et al.* [3] focused on automatically repairing bugs already fixed by patches. Their approaches are limited to predefined, well-constructed bugs, rather than identifying issues directly from information generated during building and running processes.

S. Kang, B. Chen, S. Yoo, and J.-G. Lou [4] focused on explainable automated debugging to improve interpretability. Their hypothesis-testing cycle model is particularly insightful, and we may explore applying it to automate the repair process once the core error log is identified.

M. Beller, G. Gousios, and A. Zaidman [5] and D. A. Tomassi *et al.* [6] provided datasets that include pipeline logs, historical data, and fix patches, but they do not offer insights into the logs themselves. In the later stages of our experiments, we plan to use their datasets to enhance our model's ability to suggest fixes.

Finally, W. Meng *et al.* [7] applied NLP methods to log summarization and introduced the first gold-standard dataset for this task. However, their approach is not label-efficient and does not leverage the vast knowledge encoded in pretrained LLMs. We will focus on summarizing core error logs and aim to propose a more label-efficient method for generating ground truth.

In summary, our work is different from prior work in that it brings together the power of pre-trained LLMs and label-efficient training techniques on code-related tasks with the domain of build logs, a combination which has not been explored before to our knowledge.

3 Methods

3.1 Data: BugSwarm

We selected the BugSwarm dataset to use for fine-tuning (D. A. Tomassi *et al.* [6]). The dataset itself contains raw logs from paired failing and successful runs. We have generated a derived dataset using a diffing program to extract the differences in the log between the successful and failed runs. We believe lines that are added in the diff will be significantly more likely to contain relevant errors.

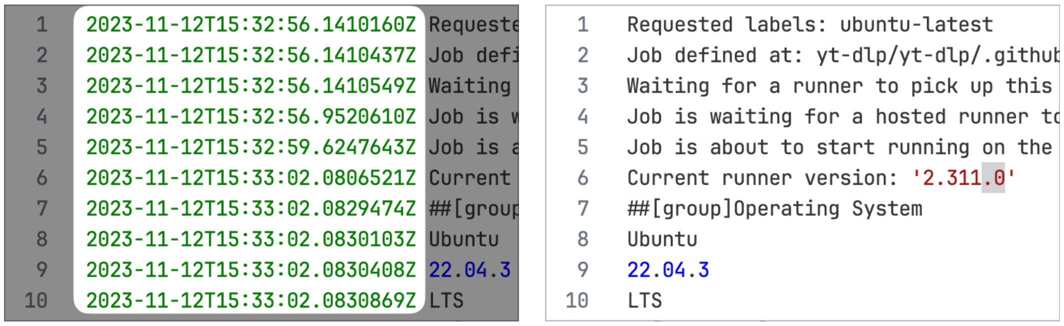
BugSwarm contains build jobs from various Python and Java open-source projects, and also raw logs from paired failing and successful runs. We imported 4,455 pairs of build logs and generated silver labels by distilling the raw logs into diff hunks that capture only the "changes" between failed and passed logs. An overview of the dataset is provided in Table 1. The methods used to clean the data are detailed in the following sections. After distilling, we managed to reduce the log size by approximately 50%.

Table 1: Dataset Overview: This table shows the key statistics about the logs before and after our distillation process, including the average line and character count for both passed and failed logs. After noise reduction, the average length of the logs was significantly reduced.

	Sourcecode Type		**Total**
	Python	**Java**	
Task Count	1980	2405	4455
Avg. Line Count (passed)	2613	8258	5655
Avg. Line Count (passed, distilled)	893	4836	3032 (-46.4%)
Avg. Line Count (failed)	2803	5196	4084
Avg. Line Count (failed, distilled)	1107	3117	2198 (-46.2%)
Avg. Char Count (passed)	202301	740882	492095
Avg. Char Count (passed, distilled)	65735	450832	274139 (-44.3%)
Avg. Char Count (failed)	215747	484425	359556
Avg. Char Count (failed, distilled)	78639	287113	191431 (-46.8%)
Avg. Hunk Count (per task)	1.76	1.76	1.76
Avg. Hunk Char Size (anwser)	180	476	338
Avg. Context Char Size (question)	246	629	452

3.1.1 Timestamp & Garbage Removal

To clean and preprocess raw logs, our first step is to remove irrelevent log lines, as shown in Figure 1. Using a pre-defined blacklist of regular expressions, we identified patterns that commonly appear in logs, such as branch updates, timestamps and random paths. By iterating through lines and applying the regular expression to filter out matches, we remove those non-informative lines and reduce noise level significantly.



```
2023-11-12T15:32:56.1410160Z Requeste
2023-11-12T15:32:56.1410437Z Job defi
2023-11-12T15:32:56.1410549Z Waiting
2023-11-12T15:32:56.9520610Z Job is w
2023-11-12T15:32:59.6247643Z Job is a
2023-11-12T15:33:02.0806521Z Current
2023-11-12T15:33:02.0829474Z ##[group
2023-11-12T15:33:02.0830103Z Ubuntu
2023-11-12T15:33:02.0830408Z 22.04.3
2023-11-12T15:33:02.0830869Z LTS
```

(a) Original log

```
Requested labels: ubuntu-latest
Job defined at: yt-dlp/yt-dlp/.githut
Waiting for a runner to pick up this
Job is waiting for a hosted runner tc
Job is about to start running on the
Current runner version: '2.311.0'
##[group]Operating System
Ubuntu
22.04.3
LTS
```

(b) After removal

Figure 1: An example of timestamp removal

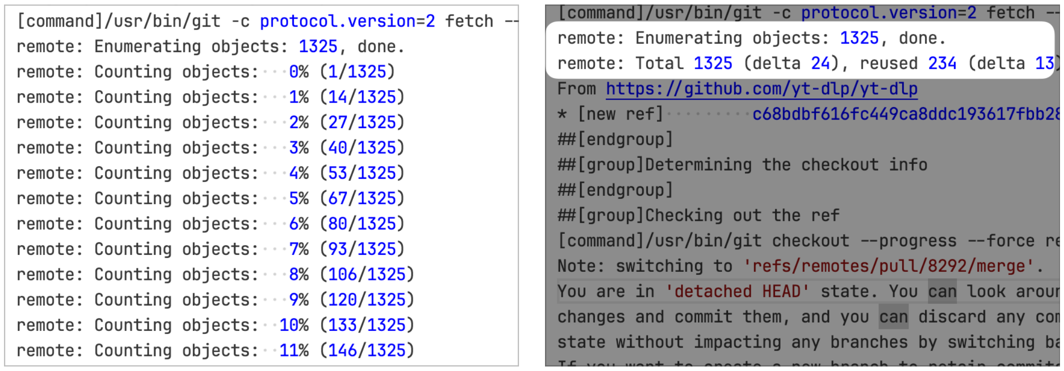
3.1.2 ASCII Escape Code Detection

ASCII escape codes[1] are a significant source of noise in the logs. These codes are typically generated by progress bars during the build process and, due to their repetitive nature, can occupy a substantial portion of the logs. However, they are irrelevant for error detection and debugging tasks.

To address this, we systematically remove lines containing 'Erase in Line' and 'Scroll Up & Erase Line' sequences, effectively simulating the behavior of a terminal. This cleaning step ensures that

[1]https://en.wikipedia.org/wiki/ANSI_escape_code#Control_Sequence_Introducer_commands

the resulting dataset is free from visual artifacts, allowing the focus to remain on meaningful log content. An example of these escape codes and their removal is shown in Figure 2.



```
[command]/usr/bin/git -c protocol.version=2 fetch --
remote: Enumerating objects: 1325, done.
remote: Counting objects:   0% (1/1325)
remote: Counting objects:   1% (14/1325)
remote: Counting objects:   2% (27/1325)
remote: Counting objects:   3% (40/1325)
remote: Counting objects:   4% (53/1325)
remote: Counting objects:   5% (67/1325)
remote: Counting objects:   6% (80/1325)
remote: Counting objects:   7% (93/1325)
remote: Counting objects:   8% (106/1325)
remote: Counting objects:   9% (120/1325)
remote: Counting objects:  10% (133/1325)
remote: Counting objects:  11% (146/1325)
```

(a) Original log

```
[command]/usr/bin/git -c protocol.version=2 fetch --
remote: Enumerating objects: 1325, done.
remote: Total 1325 (delta 24), reused 234 (delta 13)
From https://github.com/yt-dlp/yt-dlp
* [new ref]         c68bdbf616fc449ca8ddc193617fbb28
##[endgroup]
##[group]Determining the checkout info
##[endgroup]
##[group]Checking out the ref
[command]/usr/bin/git checkout --progress --force re
Note: switching to 'refs/remotes/pull/8292/merge'.
You are in 'detached HEAD' state. You can look arour
changes and commit them, and you can discard any com
state without impacting any branches by switching ba
```

(b) After timestamp removal

Figure 2: An example of terminal simulation

3.1.3 Label Generation

To identify the root cause of failures within build logs, we generate labels by analyzing the differences between paired failed and passed logs. The key assumption here is that the "removed part" from failed to passed logs represents the root cause of the failure. This method allows us to extract the most meaningful information about what caused the build to fail.

Our process relies on a unified diff-based approach. For each log pair, we compute multiple **diff hunks** that encapsulate the changes between failed and passed logs. Each hunk is associated with its corresponding **context**, which includes surrounding lines from the logs for additional information. We use context window lengths of 20 and 50 to provide different levels of detail in the generated hunks.

By iterating over the tasks and computing the diffs, we generate a labeled dataset in JSON format, which includes the task ID, hunk index, hunk content, and context. This structured data is then used for training and evaluation.

An example of the hunk and its context is shown in Appendix Section B.

3.2 Fine-tuning

We elected to use the Unsloth python library in order to fine-tune using a Low-Rank Adaptation (LoRA) method (E. J. Hu *et al.* [8]). We are using unsloth's version of the Llama-3.2 3B Instruct model, quantized, loaded from huggingface as unsloth/Llama-3.2-3B-Instruct-bnb-4bit. Our hardware was an NVIDIA GeForce RTX 2080 Ti. It had about 12GB of VRAM (11264MiB). We use mostly Unsloth's default recommended hyperparameters. An excerpt of our fine-tuning code is in Appendix A[2].

We use a custom template, similar to the one described in Listing 1, to format the input-output pairs for the model. The hunks and their corresponding contexts are passed through this template, which structures them into a Hugging Face trl-compatible format. This transformation ensures that the data is properly formatted for training, with the hunks as the answer and the corresponding context as question. We split our data along a 90/10 train/test split. We ran our fine-tuning process on 4,000 randomly selected samples from the training set.

3.3 Experiment: Zero-shot prompting

Our primary experimental methodology was to compare models using a zero-shot prompt, because this better matches the expected final use case of a tool like this (though it is possible a few-shot model could be implemented in a way that is opaque to the user by hiding it within a system prompt).

[2]The full code used for the entire project is available at https://github.com/uwm-cs561/stunning-goggles.

Our prompt template is:

```
prompt_template = """You are an expert DevOps engineer. You are examining some
logs. Your first task is to find which lines, if any, contain errors. Examine the
following logs.

### Logs:
{}

### Instructions:
Find the lines that contain the errors and repeat them verbatim, and then stop. Do
not include the irrelevant lines that do not contain any errors.

### Response:
{}"""
```

Listing 1:Prompt template

The output of the LM was then compared against our expected output using the BLEU metric. Although designed to be a measure of translation accuracy, this metric is a general-purpose measure of string similarity. Any particular value here should not be interpreted as having any particular meaning, but a higher value tells us that the strings are more similar than a lower value. Values range between 0 and 1.

4 Results

4.1 Exploratory Testing

We ran simple exploratory experiments by setting up the base model and prompting it in a fairly naive way; for example, providing it with context containing a subset of log lines from BugSwarm, and a simple prompt like "Identify the error in the following logs." This kind of exploratory testing gave us an idea of the model's baseline capabilities. The pretrained model without any additional fine-tuning seems to produce reasonable output to these sorts of questions, often giving seemingly helpful advice for how to fix the error in question.

4.2 Experiment: Zero-shot prompting

We selected 512 random samples out of our test test, and evaluated the LM's performance on these (due to resource constraints, we could not evaluate on the whole set, but 512 was a somewhat arbitrary choice).

First, a non-negligible fraction of samples caused our LM to crash during inference, both the base model and the fine-tuned one. Some of our samples were simply too large for the memory that was allocated on the device we were using. The fine-tuned model crashed with a slightly more diverse set of errors, but the overall number of crashes was almost identical.

In addition, both models had a tendency to produce results that were evaluated to a BLEU score of 0, indicating no overlap between the generated output and the reference text. After our fine-tuning, the model produced notably more 0 results. Anecdotally, we saw an increase in very short negative responses, like "None yet. Please proceed to examine the logs." and "No errors found." We did not analyze the output systematically to determine any trends among low-scoring responses. However, it is possible that there was a systemic qualitative difference in the kind of response generated by the LMs when it did not match the label that would not be captured by a BLEU score. Therefore we also report the mean BLEU score when excluding scores of 0.

Table 2: Summary Results

	Error count (approx. fraction)	**Zero count** (approx. fraction)	**Mean BLEU** excl. errors	**Mean BLEU** excl. errors and zeros
Llama-3b	**41** (0.08)	**242** (0.47)	**0.076**	**0.156**
Fine-tuned	**39** (0.08)	**326** (0.64)	**0.054**	**0.174**

The distribution of scores for the base and fine-tuned models was similar, besides the fine-tuned model's increased number of 0 scores. A histogram of scores is shown in Figure 3.

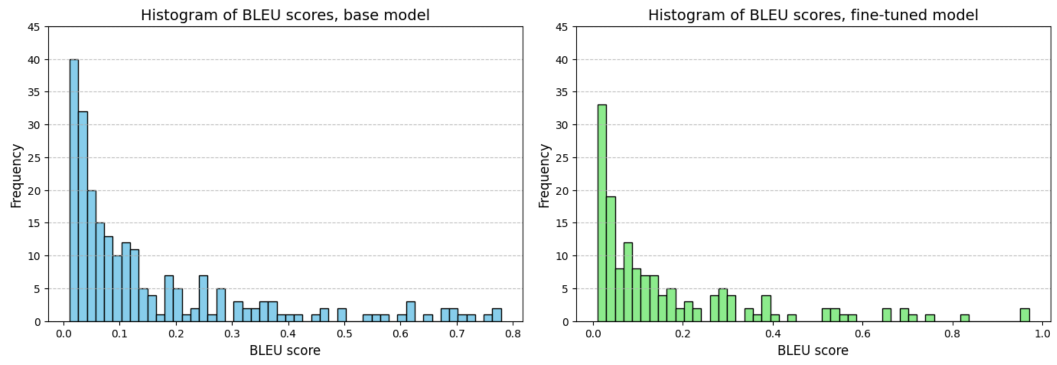


Figure 3: Histogram of BLEU scores, with errors and scores of 0 excluded because the number of these is at a vastly different scale.

Additionally, we compared the models' performance on a per-sample basis. Since each model was given the same evaluation set of 512 prompts, we calculated the increase or decrease in BLEU score for each sample (excluding samples where either model ran into an error). The results can be seen in Figure 4.

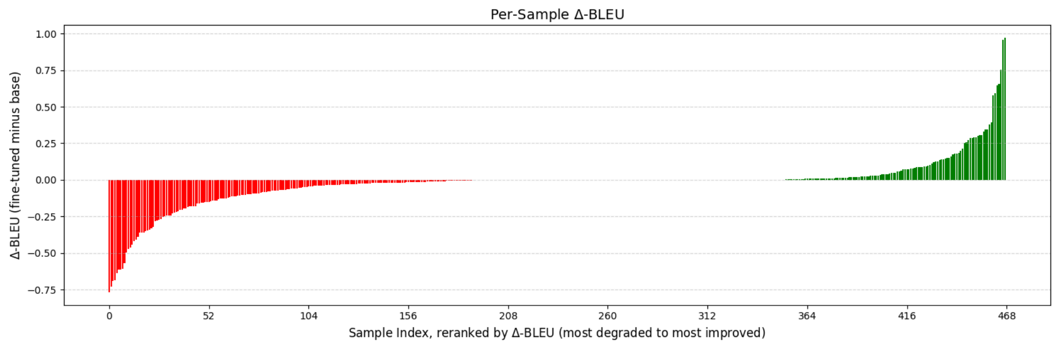


Figure 4: The degradation or improvement in BLEU score on a per-sample basis. The deltas were calculated, and then the results were reranked from "worst" to "best" - that is, from the sample where fine-tuning had the largest degradation, to the sample where fine-tuning showed the most improvement. Samples where either model crashed were excluded.

5 Discussion

We sought to answer the following research questions:

- RQ1: Can a local version of a (consumer grade) open-source LM be used to identify errors?
- RQ2: Can fine-tuning the LM on build log datasets increase its usefulness?

RQ1 was a "buy-in" question. If a consumer-grade LM was not powerful enough to do anything useful in this domain, then no amount of fine-tuning would grant it the capability. We answered this question through qualitative and exploratory testing; frankly, it is not the interesting research question.

The answer to RQ2 remains inconclusive. We did not demonstrate a method to fine-tune an LM to increase its performance on our log-summarization task. After fine-tuning, the model performed worse averaged across our samples. However, excluding the responses that got a BLEU score of 0, the fine-tuned model performed better. This could be justified by qualitative analysis of the actual content of the responses; perhaps the fine-tuned model was more likely to say "there is no error" and just be flat-out wrong, but when it did report an error, it gave better responses.

The larger question motivating this research, however, is whether LMs can be useful to DevOps engineers, not whether LMs can perform better or worse on this specific task. In our exploratory experiments, it seemed that the model was capable of providing useful feedback. It may be the case that our framing of the problem as a smaller, more measurable question is simply unnecessarily limiting to an LM that has already trained a more broad and useful capability.

What's more, our experiments indicate that fine-tuning an LM on commodity hardware leads to significant challenges, especially in terms of resource usage and performance. Despite applying LoRA techniques to minimize memory usage and improve efficiency, we encountered several limitations during the training process. The hardware constraints, including limited GPU memory, led to frequent crashes and freezes, especially when processing larger log samples.

It may be more practical to explore alternative approaches, such as using specialized models designed for log analysis or error detection. They could provide better performance without the heavy computational burden of LMs. Moreover, leveraging non-LM models like task-specific neural architectures might give us more reliable and scalable solutions for our task.

Another alternative is simply to integrate off-the-shelf LMs with an open-ended chat process and ask them open-ended questions you want answers to, rather than jumping through a lot of hoops trying to get them to produce some precise response in a highly-quantified task.

6 Future Work

The most immediate next step would be to refactor (and reimplement as needed) our training and evaluation process, perhaps acquiring better hardware, either directly or by farming out the training to a service. We could use more larger models, quantize them less, and fine-tune them using different LoRA parameters or even run full fine-tuning. The biggest obstacle this research faced was lack of resources.

Another very useful direction would be the acquisition or production of better datasets. M. Beller, G. Gousios, and A. Zaidman [5] and W. Meng *et al.* [7] both present datasets that we believe could be usefully applied to this task. In general, a dataset with human-generated gold labels would highly useful, but expensive to acquire, because it would require experts to produce the labels.

On the topic of data, implementing dataset selection, active learning, or other label-efficient methods would be the obvious next step to handle the resource issues. Using Fisher Embeddings for active learning as proposed by J. T. Ash, S. Goel, A. Krishnamurthy, and S. M. Kakade [9] or selecting a subset of data to train on based on diversity measurements as proposed by P. Wang *et al.* [10] could be fruitful.

References

[1] Q. Sun *et al.*, "A Survey of Neural Code Intelligence: Paradigms, Advances and Beyond," Mar. 2024, doi: 10.48550/ARXIV.2403.14734.

[2] C. E. Jimenez *et al.*, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," Oct. 2023, doi: 10.48550/ARXIV.2310.06770.

[3] C. Le Goues *et al.*, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015, doi: 10.1109/TSE.2015.2454513.

[4] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, "Explainable Automated Debugging via Large Language Model-driven Scientific Debugging," Apr. 2023, doi: 10.48550/ARXIV.2304.02195.

[5] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 447–450. doi: 10.1109/MSR.2017.24.

[6] D. A. Tomassi *et al.*, “BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes,” in *ICSE*, IEEE / ACM, 2019, pp. 339–349.

[7] W. Meng *et al.*, “LogSummary: Unstructured Log Summarization for Software Systems,” *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 3803–3815, 2023, doi: 10.1109/TNSM.2023.3236994.

[8] E. J. Hu *et al.*, “LoRA: Low-Rank Adaptation of Large Language Models.” [Online]. Available: https://arxiv.org/abs/2106.09685

[9] J. T. Ash, S. Goel, A. Krishnamurthy, and S. M. Kakade, “Gone Fishing: Neural Active Learning with Fisher Embeddings,” *CoRR*, 2021, [Online]. Available: https://arxiv.org/abs/2106.09675

[10] P. Wang *et al.*, “Diversity Measurement and Subset Selection for Instruction Tuning Datasets.” [Online]. Available: https://arxiv.org/abs/2402.02318

Appendix

A Contributions

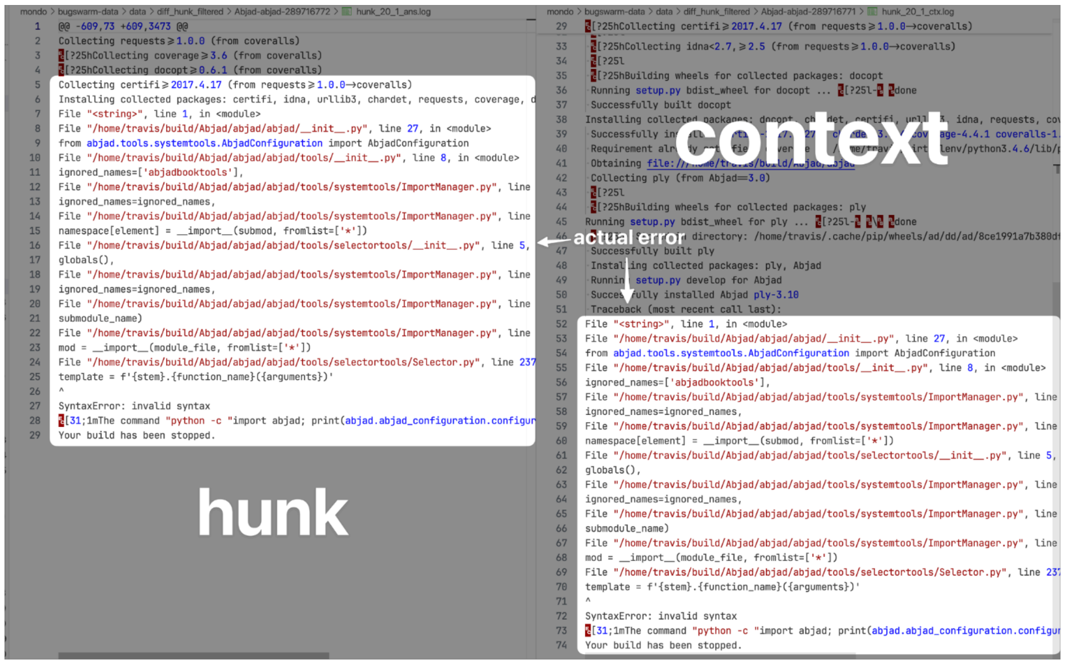
A.1 Mondo

- Infrastructure setup (docker container with ssh access)
- Data downloading (non-trivial due to rate limit)
- Data preprocessing
- Model fine-tuning (primary)
- Writing this report

A.2 Daniel

- Infrastructure setup (department-provided GPU machines)
- Exploratory testing
- Model fine-tuning (some assistance)
- Model evaluation
- Writing this report

B Example of Hunk and Context


mondo > bugswarm-data > data > diff_hunk_filtered > Abjad-abjad-289716772 > hunk_20_1_ans.log
mondo > bugswarm-data > data > diff_hunk_filtered > Abjad-abjad-289716771 > hunk_20_1_ctx.log
hunk
context
actual error

```
@@ -609,73 +609,3473 @@
Collecting requests≥1.0.0 (from coveralls)
[?25hCollecting coverage≥3.6 (from coveralls)
[?25hCollecting docopt≥0.6.1 (from coveralls)
Collecting certifi≥2017.4.17 (from requests≥1.0.0→coveralls)
Installing collected packages: certifi, idna, urllib3, chardet, requests, coverage, d
File "<string>", line 1, in <module>
File "/home/travis/build/Abjad/abjad/abjad/__init__.py", line 27, in <module>
from abjad.tools.systemtools.AbjadConfiguration import AbjadConfiguration
File "/home/travis/build/Abjad/abjad/abjad/tools/__init__.py", line 8, in <module>
ignored_names=['abjadbooktools'],
File "/home/travis/build/Abjad/abjad/abjad/tools/systemtools/ImportManager.py", line
ignored_names=ignored_names,
File "/home/travis/build/Abjad/abjad/abjad/tools/systemtools/ImportManager.py", line
namespace[element] = __import__(submod, fromlist=['*'])
File "/home/travis/build/Abjad/abjad/abjad/tools/selectortools/__init__.py", line 5,
globals(),
File "/home/travis/build/Abjad/abjad/abjad/tools/systemtools/ImportManager.py", line
ignored_names=ignored_names,
File "/home/travis/build/Abjad/abjad/abjad/tools/systemtools/ImportManager.py", line
submodule_name)
File "/home/travis/build/Abjad/abjad/abjad/tools/systemtools/ImportManager.py", line
mod = __import__(module_file, fromlist=['*'])
File "/home/travis/build/Abjad/abjad/abjad/tools/selectortools/Selector.py", line 237
template = f'{stem}.{function_name}({arguments})'
^
SyntaxError: invalid syntax
[31;1mThe command "python -c "import abjad; print(abjad.abjad_configuration.configur
Your build has been stopped.
```

```
[?25hCollecting certifi≥2017.4.17 (from requests≥1.0.0→coveralls)
 [?25hCollecting idna<2.7,≥2.5 (from requests≥1.0.0→coveralls)
 [?25l
 [?25hBuilding wheels for collected packages: docopt
 Running setup.py bdist_wheel for docopt ... [?25l- done
 Successfully built docopt
Installing collected packages: docopt, ch[illegible]det, certifi, urll[illegible]3, idna, requests, co
 Successfully in[illegible] age-4.4.1 coveralls-1.
 Requirement alr[illegible] lenv/python3.4.6/lib/p
 Obtaining file:///[illegible]/travis/build/Abjad/abjad
 Collecting ply (from Abjad==3.0)
 [?25l
 [?25hBuilding wheels for collected packages: ply
Running setup.py bdist_wheel for ply ... [?25l- \ done
 [illegible] directory: /home/travis/.cache/pip/wheels/ad/dd/ad/8ce1991a7b380d
 Successfully built ply
 Installing collected packages: ply, Abjad
 Running setup.py develop for Abjad
 Successfully installed Abjad ply-3.10
 Traceback (most recent call last):
File "<string>", line 1, in <module>
File "/home/travis/build/Abjad/abjad/abjad/__init__.py", line 27, in <module>
from abjad.tools.systemtools.AbjadConfiguration import AbjadConfiguration
File "/home/travis/build/Abjad/abjad/abjad/tools/__init__.py", line 8, in <module>
ignored_names=['abjadbooktools'],
File "/home/travis/build/Abjad/abjad/abjad/tools/systemtools/ImportManager.py", line
ignored_names=ignored_names,
File "/home/travis/build/Abjad/abjad/abjad/tools/systemtools/ImportManager.py", line
namespace[element] = __import__(submod, fromlist=['*'])
File "/home/travis/build/Abjad/abjad/abjad/tools/selectortools/__init__.py", line 5,
globals(),
File "/home/travis/build/Abjad/abjad/abjad/tools/systemtools/ImportManager.py", line
ignored_names=ignored_names,
File "/home/travis/build/Abjad/abjad/abjad/tools/systemtools/ImportManager.py", line
submodule_name)
File "/home/travis/build/Abjad/abjad/abjad/tools/systemtools/ImportManager.py", line
mod = __import__(module_file, fromlist=['*'])
File "/home/travis/build/Abjad/abjad/abjad/tools/selectortools/Selector.py", line 237
template = f'{stem}.{function_name}({arguments})'
^
SyntaxError: invalid syntax
[31;1mThe command "python -c "import abjad; print(abjad.abjad_configuration.configur
Your build has been stopped.
```

Figure 5: An example of Hunk and Context

C Training Code Excerpt

```python
dataset = load_dataset("json", data_files=data_files, split="train")

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/Llama-3.2-3B-Instruct-bnb-4bit",
    max_seq_length=max_seq_length,
    dtype=None,
    load_in_4bit=True,
)

model = FastLanguageModel.get_peft_model(
    model,
    r=16,
    target_modules=[
        "q_proj",
        "k_proj",
        "v_proj",
        "o_proj",
        "gate_proj",
        "up_proj",
        "down_proj",
    ],
    lora_alpha=16,
    lora_dropout=0,
    bias="none",

    use_gradient_checkpointing="unsloth",
    random_state=3407,
    max_seq_length=max_seq_length,
    use_rslora=False,
    loftq_config=None,
)

trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    tokenizer=tokenizer,
    args=TrainingArguments(
        per_device_train_batch_size=2,
        gradient_accumulation_steps=4,
        warmup_steps=10,
        max_steps=60,
        fp16=not is_bfloat16_supported(),
        bf16=is_bfloat16_supported(),
        logging_steps=1,
        output_dir="outputs",
        optim="adamw_8bit",
        seed=3407,
    ),
)

trainer.train()
```