Transfer of Knowledge

Peyton Smith

Information

GitHub: https://github.com/Peyton-Smith05/llm_code_vulnerability_detection
HuggingFace Dataset: https://huggingface.co/datasets/peytonwsmith/lava2llama
HuggingFace LLM: https://huggingface.co/peytonwsmith/llama2lavafinetune/tree/main
LAVA GitHub: https://github.com/panda-re/lava
LAVA Paper: https://ieeexplore.ieee.org/document/7546498
LLaMA2 GitHub: https://github.com/facebookresearch/llama
LLaMA2 HuggingFace: https://huggingface.co/meta-llama
LLaMA2 Paper: https://ai.meta.com/llama/

Introduction

The goal for research this semester was to determine how viable Large Language Models (LLMs) are as an option for autonomously detecting and correcting vulnerabilities in code. The secondary goal was to create/improve a current LLM in this functionality. To accomplish this, we needed to download and run various models, collect a dataset that can test a model's ability to detect and correct code, collect a dataset that can train/fine-tune the model to detect and correct code, and finetune an LLM.

Topics

Large Language Models

There are many types of models. The two main ones are:

Text-2-text generation: This type of model maps one piece of text to another. An example would be mapping conversations such as an input and a response. This is what ChatGPT is.

Text generation: This purely generates the next token. It looks at what has been said and outputs what is most likely to come next. Code LLaMA is like this. It completes functions.

Read More: https://medium.com/@sharathhebbar24/text-generation-v-s-text2text-generation-3a2b235ac19b)

Text generation models are not the best at performing requests because they don't understand conversation format. Text-2-text generation understands conversation format and understands requests. If we wanted to use text generation for the current task, the model would have to be shown a lot of text that has code and then an output of the correction in the format wanted. Generally, a lot of the finetuned models out there that are good are good at generating code from a function definition. This makes it hard to show it how to generate corrections from faulty input code.

To use text-2-text generation, we just need to show it conversations of a user and a code analyzer. This model already maps conversations. Since it already has a lot of other conversations mapped to it, it is much better at abstracting and inferring new information. This is ultimately why I chose to pursue using LLaMA2 because it does text-2-text and will probably perform better.

Juliet Dataset

This dataset has various vulnerabilities classified with the CWE convention. All this code is written in C and C++. There is a good function and a bad function. I wanted to use this dataset to generate a vulnerability and a patch to the code. This could be put in conversation as a dataset and trained further on top of a current LLM.

There are a few problems with this dataset though. One, is that there are repeat examples of code files. Two, the functions are all labeled “good function” and “bad function”. Three, the functions do not have an input to trigger the bug. This means it is hard to test whether the bug was fixed or not. Because of these problems we decided to pursue the use of the LAVA dataset. However, before doing this I attempted to see how good it is to purely classify bugs from the Juliet dataset. I gave it 3000 pieces of code from the buggy function and for each one asked it to classify the CWE error. Of those 3000 queries it classified 19.2% of the buggy code correctly. This was using LLaMA2. The code that I did this with is described in both the Juliet and LLaMA folders of the repository.

One challenge I faced while doing this was parsing the output. Because the LLM is nondeterministic, it sometimes had output that the code could not parse. This was rare and for the most part output what I wanted when I specified how the output should be. This will always be a problem with the system we are trying to build. See more about Pydantic in the continuation section for a potential solution that I did not get a chance to explore.

LLaMA2

LLaMA2 has multiple versions of its model. There are 7b, 13b, and 70b parameter models with chat and completion version. The difference between the number of parameters is purely the size of the model. The 70b parameter model is the largest and most effective model. It requires about 4 NVIDIA A100s to run it because the model is about 130 GB. The largest model I was able to query was the 13b chat model.

The difference between the chat and completion models is purely the type of text generation it performs. The chat model is used to complete text in conversation. The input looks different. There can be a System prompt, a User, and Output field that the model then completes. The text generation model simply continues whatever text it was given. It continues code, sentences, or paragraphs.

LLaMA2 can be downloaded and run multiple ways. One of the main ways is through HuggingFace.com. HuggingFace is a company dedicated to fostering the AI/ML community. On their website you can upload/download models, datasets, and more. The LLaMA2 models are all uploaded onto this website. A big advantage to HuggingFace is their Transformers library. This library has prebuilt functions for loading models and datasets off HuggingFace repositories. It also has libraries for training, querying, and finetuning models.

The second way to access LLaMA2 is directly from the GitHub repository. Clone the repository and follow the setup instructions to access it and query it locally there.

Transformers Library

Link: https://huggingface.co/docs/transformers/index

The transformers library is a library developed by HuggingFace that allows for easy integration of repositories on HuggingFace. It allows models and datasets to easily be downloaded and trained then reuploaded again. The documentation for this library is linked above. The code in my repository has many examples of loading models from HF and loading datasets.

Querying

The best way to query the model that I found was using the example_chat_completion.py file that they have in their repository. This code sets up the model after it has been downloaded and allows it to be queried. Then, running a command in the terminal the script runs its queries. All of this can be seen in my repository in the LLaMA README.md file.

Fine-Tuning

I tried several forms of finetuning. I was able to successfully finetune a LLaMA2 model with the transformers library and the SAMSUM dataset which is a dataset that teaches it how to summarize. This dataset was preprocessed by META in the transformers library, so it was ready to be put into a pipeline for training. However, when I collected the data from LAVA and put it into a dataset and tried the same code, there was an issue with the format of the data. After finding a different tutorial that recommended putting the data into a system prompt and response format, I still could not get the model to finetune. Both code attempts are outlined in the repository under the extra folder.

The issue with formatting is just the finicky way the model accepts parameterized data. With more time I am sure that this will have an easy solution.

LAVA

LAVA is a system meant to inject bugs/vulnerabilities into code bases. It has various pre-setup repositories such as Blecho, LibYaml, and File. The code picks 50 potential vulnerabilities and then adds them to the code base to taint it. This was picked because there is a good and buggy code base to compare to in the dataset. In addition, it also provides an input to trigger the buggy code. This input can be used to test the code base.

I wrote code to parse the output of LAVA. All instructions to set up LAVA and parse the data with my code are in the repository.

Code

During the research project I sought to run and query an LLM locally, finetune an LLM, run LAVA locally, parse LAVA output, and parse the Juliet dataset. I was able to successfully do all those things. The code and steps to repeat any work I have done have been outlined in the repository link at the beginning of this document. The repository contains code and instructions to run LLaMA2, run LAVA, parse LAVA, and finetune models.

Continuation

There are multiple areas or paths to continue pursuing in this project. The first is to finish formatting the LAVA data so that it can finetune an LLM model. If that is successful it is worth finetuning multiple models and seeing if better performance can be achieved. It's also needed to experiment with the types of prompts the model is finetuned with. More or less context of code, better system prompts, better user prompts, etc.

Another avenue worth exploring is learning how to vectorize repositories of code. Vectorizing multiple files of code and performing vector searches on the data can be a great way to provide the correct context to the code to catch more errors.

Another resource I came across is Pydantic. I do not know much about it but from the video below it seems to have solutions to parsing LLM output data. Since LLM output data can be random and vary, it is difficult to use in systems that need consistency in output. This library seems to allow for more exact queries to the model and allows the model to be queried until the output matches a desired format.
LangChain is another developing package that assists in making the model agentic, and conversational. Making an LLM agentic can be useful for things like code injections/replacements for automatic correction of vulnerabilities.

Links to learn more:

- Pydantic: https://www.youtube.com/watch?v=yj-wSRJwrrc&t=963s
- LangChain: https://www.youtube.com/watch?v=hLQ8DAkcygI&t=2242s
- Transformers library HF: https://huggingface.co/docs/transformers/index
- More articles:
    - https://medium.com/@gurpartap.sandhu3/using-ai-to-find-security-bugs-in-code-and-generate-code-fixes-3b0da2a21bb4