Malicious Mail Detector

Iván Iroslavov Petkov
Emilio Rodrigo Carreira Villalta

Monday 9th September, 2024

Table of Contents

List of Figures

1. Introduction

This document outlines the process of developing a system aimed at detecting malicious emails using large language models (LLMs) and cutting-edge machine learning techniques. The goal is to create a highly accurate solution that identifies and flags potentially harmful email content, leveraging advanced tools to enhance cybersecurity defenses. The system integrates several state-of-the-art technologies, including a custom-trained Llama model for natural language understanding, alongside techniques for email content analysis, pattern recognition, and threat detection. By combining these approaches, the system can not only identify malicious emails but also provide detailed explanations of the detected threats, highlighting suspicious elements, such as phishing attempts, embedded malware, or social engineering tactics. This demonstrates its value in enhancing email security and mitigating risks posed by harmful digital communication.

The project, accessible in this GitHub repository [1], showcases the development and training of a custom Llama model tailored for detecting malicious email content. It covers key features including data preprocessing, model architecture, training pipelines, and evaluation procedures to assess the system's performance. Additionally, it includes scripts for model training and testing, as well as techniques for handling large datasets, ensuring that the system is robust and scalable. The final output not only identifies potential threats but also provides insights into the nature of the risks, enhancing interpretability and making the tool useful for cybersecurity experts and automated defense systems. To further enhance usability, the system is integrated into a Flask-based web interface, making it accessible and easy to deploy in real-world environments.

This document details the comprehensive steps for setting up and running the project, explaining the preprocessing methods used for training data, the architecture of the Llama model, and the training and evaluation strategies. Each component of the implementation is thoroughly documented to ensure that readers can grasp both the practical aspects and the theoretical underpinnings of the system.

This project is part of a larger initiative to improve cybersecurity through the application of advanced machine learning techniques and LLMs. Inspired by the work of experts in both natural language processing and security domains, the objective is to provide a guide that encourages further experimentation and innovation in the development of secure and reliable digital communication systems.

[1] https://github.com/rorro6787/malicious-mail-detector

2. LLama, Llama2

Contents

LLaMA, which stands for "Large Language Model Meta AI," is a family of advanced natural language processing models developed by Meta (formerly Facebook). LLaMA models are designed to generate human-like text based on given prompts, and they are widely utilized in various language-related tasks due to their high efficiency and scalability. Here's a detailed overview of LLaMA 2:

- **Single Transformer Model Approach**: Unlike earlier models that relied on more complex architectures, LLaMA 2 employs a single transformer-based architecture to generate coherent, context-aware text outputs. This streamlined approach allows the model to handle a wide range of tasks from translation to summarization with remarkable efficiency.
- **Real-time Text Generation**: LLaMA 2 is optimized for fast inference, enabling real-time text generation capabilities. This makes it suitable for applications that require instant responses, such as chatbots, virtual assistants, and live customer support systems. LLaMA 2 can generate text with low latency, even on relatively modest hardware.

LLaMA 2 represents the latest iteration in Meta's family of large language models, offering improvements in natural language understanding and generation capabilities. As a continuation of the LLaMA series, LLaMA 2 enhances both text generation quality and efficiency. Here's a detailed look at what makes LLaMA 2 stand out, particularly the Llama-2-7b-chat-hf.

2.1 Why Llama-2-7b-chat-hf?

The model Llama-2-7b-chat-hf is a specialized variant in the LLaMA 2 family, and its name conveys important information about its purpose and characteristics:

- **NousResearch**: This prefix indicates that the model has been fine-tuned and released by the NousResearch team, who have made specific modifications to the base LLaMA 2 model for improved performance in chat-based scenarios.
- **Llama-2**: This signifies that the base model comes from Meta's LLaMA 2 series, which is known for its transformer-based architecture and strong performance across a wide variety of natural language processing tasks.
- **7b**: This refers to the model's size, which contains 7 billion parameters. It is a medium-sized model in the LLaMA 2 family, balancing computational efficiency with robust text generation capabilities. While smaller than the 70B variant, the

7B model provides a good balance of speed and accuracy, making it well-suited for many real-time chat applications.

- **chat**: The "chat" designation highlights that this model has been specifically fine-tuned for dialogue-based tasks. This includes generating coherent, contextually aware responses in a conversational format, making it ideal for virtual assistants, chatbots, and customer support systems.
- **hf (Hugging Face)**: This indicates that the model is hosted and can be deployed via the Hugging Face platform. Hugging Face provides easy-to-use APIs and infrastructure for deploying and scaling NLP models, making it accessible for a wide range of users.

Despite being a 7B-parameter model, NousResearch/Llama-2-7b-chat-hf delivers high-quality conversational abilities with low computational overhead. It is particularly well-suited for applications requiring real-time interaction, such as live chat systems, AI-driven customer support, and personal assistants. While not as resource-intensive as larger models, it still offers significant capabilities for understanding and responding to user inputs in natural language, making it an excellent choice for developers who need efficient, scalable conversational AI solutions.

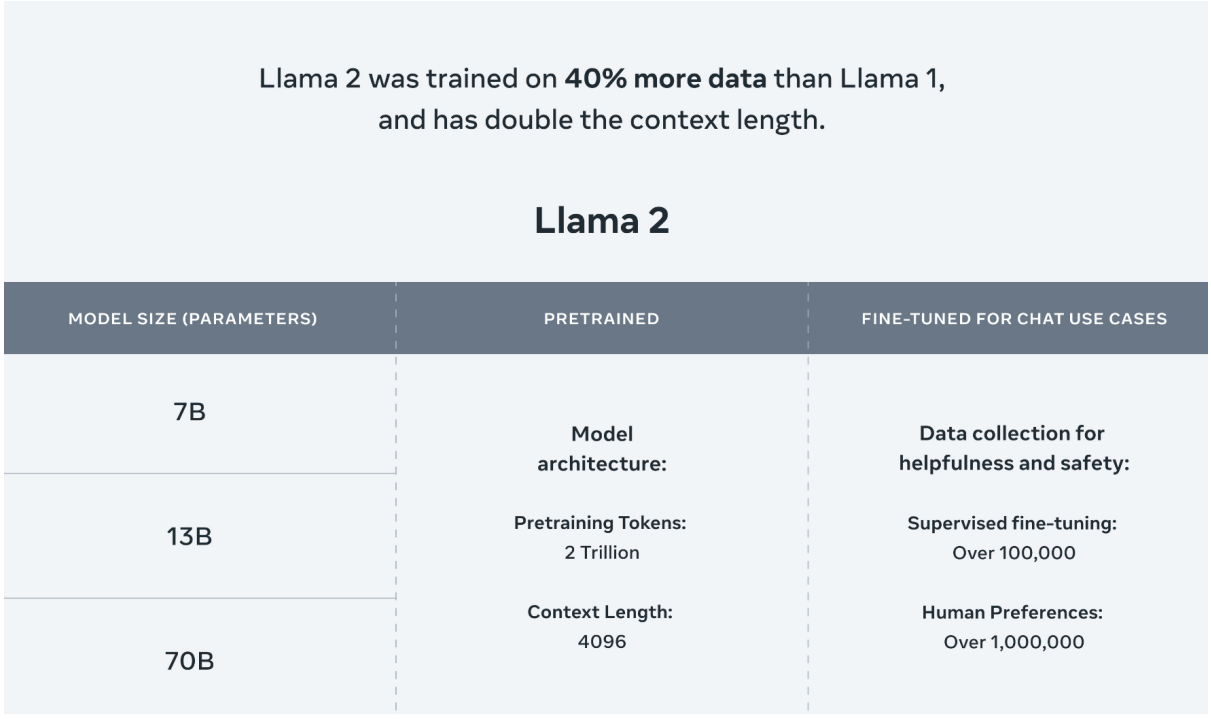

Figure 2.1: LLama2 models features

3. Training the Model

Contents

The main goal is to be able to a upervised fine-tuning (SFT), which adjusts a pre-trained model to a new dataset or specific task. The base model is ”NousResearch/Llama-2-7b-chat-hf” from Hugging Face, and it is being fine-tuned for a particular task, in this case, ”malicious mail detection”.

The code includes steps for setting up a training environment with specific parameters for QLoRA, precision adjustments (4-bit), and the use of LoRA (Low-Rank Adaptation) to improve efficiency during fine-tuning. The model is then trained on a labeled dataset, adjusting the pre-existing model for the new task.

3.1 The Dataset

The most important aspect of the training process is both the structure and the dataset itself. Here's what it would look like:

```
[
    {
        "question": "mail1",
        "answer": {
            "spam": false,
            "spam_type": "",
            "explanation": ""
        }
    },
    {
```

```
        "question": "mail2",
        "answer": {
            "spam": true,
            "spam_type": "IPaO",
            "explanation": "explanation2"
        }
    },
    ...
]
```

The dataset is structured this way because it is designed for a spam detection task. Each email, represented by the "question" field, is labeled as either spam or not spam in the "answer" field. Here's why this structure is necessary:

- **"question" field**: This contains the content of the email, including the subject and body. The structure ensures that the full text of the email is available for analysis, allowing the model to evaluate various features (e.g., keywords, phrases, or patterns) that might indicate whether the email is spam.
- **"answer" field**: It contains a boolean field indicating whether the email is considered spam (true) or not (false). This is essential for training a model to classify emails as spam or not, an "spam_type" field that is useful in cases where there are different categories of spam (e.g., phishing, promotional, malicious), providing more granularity to the classification task. In this example, it is left empty since the emails are labeled as non-spam. Lastly, an "explanation" is included for interpretability, allowing the dataset to store explanations of why an email is labeled as spam or not. In this example, it's empty, but it can help when fine-tuning models or for later analysis when explanations are needed for decision-making.
- **Consistency**: The uniform structure allows for efficient parsing and processing of the data. Each entry follows the same format, making it easier for a machine learning model to consume the dataset during training.
- **Supervised Learning**: This structure supports supervised learning, where the model is trained using examples of both spam and non-spam emails. The labels ("spam" and "not spam") provide the necessary supervision for the model to learn the differences.

3.2 Fine-Tunning Process

For the fine-tuning process, it's important to highlight that even though we are not using the largest Llama2 model version, the one we are using is still a 10 GB model. Therefore, we cannot train it on the GPU available on our local computer. Instead, we will use the T4 Nvidia graphics card provided by Google Colab. Given this, the first step is to download the dataset and the model, and then configure the various parameters needed for training:

```
# The model that you want to train from the Hugging Face hub
model_name = "NousResearch/Llama-2-7b-chat-hf"
```

```
# Fine-tuned model name
new_model = "llama-2-7b-malicious-mail-detection"
...
# Compute dtype for 4-bit base models
bnb_4bit_compute_dtype = "float16"

# Quantization type (fp4 or nf4)
bnb_4bit_quant_type = "nf4"

# Activate nested quantization for 4-bit base models (double
   ↪ quantization)
use_nested_quant = False
...

# Number of training epochs
num_train_epochs = 24
...

def process_dataset():
    from datasets import Dataset
    import os

    cd = os.getcwd()

    data = create_dataset(os.path.join(cd, "dataset.txt"))

    # Create Dataset with combined 'text' field
    dataset = Dataset.from_dict({
        "text": [f"<s>[INST]: This is an email, it may or not
           ↪ include subject and sending information ->
           ↪ {item['question']} [/INST] {item['answer']}
           ↪ </s>" for item in data]
    })

    # dataset.save_to_disk("os.path.join(cd,
       ↪ 'custom_dataset')")

    return dataset

def create_dataset(file_path: str):
    import json

    # Read the .txt file
    with open(file_path, 'r', encoding='utf-8') as file:
        file_content = file.read()

    # Clean up the content if necessary, for example,
       ↪ deleting blank lines or unwanted characters
```

```
    data = json.loads(file_content)
    return data

dataset = process_dataset()
```

Once all the downloads and configurations are complete, the next step is to prepare and initiate the fine-tuning process as outlined below:

```
# Load tokenizer and model with QLoRA configuration
compute_dtype = getattr(torch, bnb_4bit_compute_dtype)
...

# Check GPU compatibility with bfloat16
if compute_dtype == torch.float16 and use_4bit:
    ...

# Load base model
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map=device_map
)
...

# Load LLaMA tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name,
   ↪ trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"
...

# Set training parameters
training_arguments = TrainingArguments(
    output_dir=output_dir,
    num_train_epochs=num_train_epochs,
    ...
)

# Set supervised fine-tuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    ...
)

# Train model
trainer.train()

# Save trained model
```

```
trainer.model.save_pretrained(new_model)
```

3.3 Model Inferences

The training time will vary depending on how you configure the hyperparameters, particularly the number of epochs. In my case, with 24 epochs, it will take approximately 45 minutes. Once the training is complete, we will be able to perform inferences with the model:

```
def process_generated_message(prompt):
    import re
    from transformers import pipeline
    # Run text generation pipeline with our next model

    pipe = pipeline(task="text-generation", model=model,
       ↪ tokenizer=tokenizer, max_length=1000)
    result = pipe(f"<s>[INST] {prompt} [/INST]")
    generated_text = result[0]['generated_text']
    generated_answer = generated_text.replace(f"<s>[INST]
       ↪ {prompt} [/INST]", "").strip()
    process_generated_answer = re.search(r'\{[^{}]*\}',
       ↪ generated_answer).group(0)
    return process_generated_answer

# Ignore warnings
logging.set_verbosity(logging.CRITICAL)

# Run text generation pipeline with our next model
prompt = "prompt"
process_generated_answer = process_generated_message(prompt)
print(process_generated_answer)
```

If the training process has been successful, we should receive an answer from the model with the same structure as learned from the dataset. This response will indicate whether an email is malicious or not.

```
{'spam': True,'spam_type': 'RPFI', 'explanation': '...'}
```

3.4 GPU Issue

We've developed a Python project utilizing a finely-tuned Llama2 model to determine whether an email is malicious. This project culminates in a Flask server application that integrates HTML, CSS, and JavaScript to facilitate user interactions. The core functionality revolves around downloading the trained model and using it to make inferences about email content.

3.4.1 Training and Deployment Constraints

The model was trained using Google Colab due to limitations in local GPU resources. Google Colab provided the necessary computational power, but deploying the model presents a new challenge. Inferences with this model require GPU support, specifically CUDA, for efficient processing. This requirement poses a problem for users without CUDA-compatible GPUs, as the project must still function correctly on their systems.

3.4.2 Dockerization and Compatibility Issues

To address deployment challenges, I Dockerized the application. This approach simplifies distribution and setup, ensuring that users can run the project without having to configure their environments manually. However, this solution brings its own set of complications. If a user's system lacks CUDA support, the application will not be able to perform inference as intended.

3.4.3 Handling CUDA Absence

To mitigate the issue of GPU unavailability, I implemented a workaround within the project's logic. If CUDA is not detected on the user's machine, the application generates random responses instead of raising an error. This ensures that the application remains functional, albeit with degraded performance.

3.4.4 Dockerfile Adjustments for GPU Support

Users with CUDA-enabled GPUs face a different challenge. To leverage GPU capabilities, the Dockerfile must be modified to align with the specific Python version and PyTorch version compatible with the CUDA drivers installed on the user's system. This requires users to update the Dockerfile to match their environment, ensuring proper integration with the GPU.

3.4.5 Issue Conclusion

In summary, while Dockerization has streamlined the deployment process, it has introduced the need for careful configuration to support different hardware capabilities. Users must adapt the Dockerfile based on their GPU configuration to ensure optimal performance, particularly for those with CUDA support. For users without CUDA, the application's fallback to random responses maintains usability but does not utilize the full potential of the trained model.

The complexities encountered are a result of the project's personal nature and resource constraints. Hosting the application on cloud platforms like AWS or Azure could resolve these issues by providing the necessary GPU resources and simplifying deployment. However, due to limited resources, the project must remain a local solution accessible to anyone who wishes to use it.

4. Using Flask to deploy our system

Contents

Flask is a micro web framework for Python, designed to make web application development simple and efficient. Unlike full-stack frameworks, Flask is lightweight and provides the essentials to get an application up and running. This makes it an excellent choice for building and deploying small to medium-sized web applications and APIs. Flask was developed by Armin Ronacher as part of the Pallets Project and has gained widespread popularity due to its simplicity and flexibility.

Figure 4.1: Flask framework logo

4.1 Key Features of Flask

Flask provides several features that make it suitable for web application development:

- **Lightweight and Modular:** Flask is a micro-framework, meaning it has a minimalistic core, making it easy to extend with additional modules and libraries.

- **Flexible and Easy to Use:** It allows developers to build web applications quickly with minimal boilerplate code.
- **Built-in Development Server:** Flask includes a development server for testing and debugging, making it easy to develop and test applications locally.
- **Jinja2 Templating:** Flask integrates Jinja2, a powerful templating engine, to help create dynamic HTML pages.
- **RESTful Request Handling:** Flask supports RESTful URL routing, which is essential for building APIs.
- **Extensive Documentation:** Flask has comprehensive documentation and a large community, which helps developers find solutions and support easily.

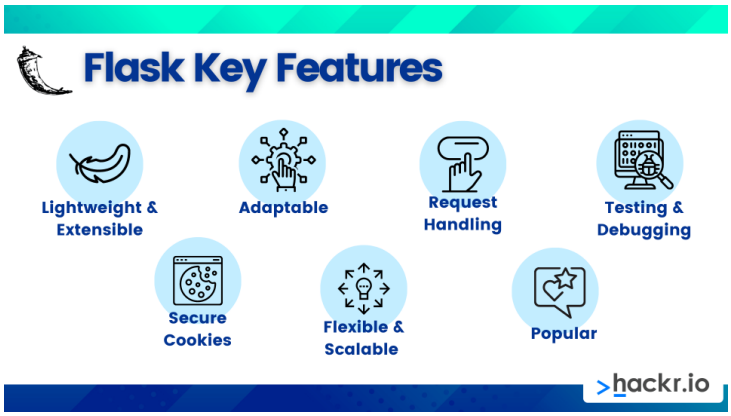

Figure 4.2: Flask framework main features

4.2 Rationale for Using Flask

In the context of our project, Flask was chosen for several reasons:

- **Simplicity:** The lightweight nature of Flask allows for a straightforward and clean architecture, which is ideal for our image description system.
- **Flexibility:** Flask's modular design means we can easily integrate additional features as needed, such as authentication, database connectivity, and API endpoints.
- **Rapid Development:** Flask's minimal setup and ease of use enable quick prototyping and development, which aligns well with our agile development approach.
- **Scalability:** While being simple, Flask can be scaled with the use of extensions and third-party libraries, making it suitable for scaling the image description system if needed.

4.3 Setting Up a Flask Application

Setting up a Flask application is straightforward. The following steps outline the basic setup for a Flask application:

4.3.1 Installation

First, Flask needs to be installed. This can be done using pip, the Python package manager:

```
pip install Flask
```

4.3.2 Creating a Basic Flask Application

Below is a simple example of a Flask application that serves a basic web page:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return 'Welcome to the Image Description System!'

if __name__ == '__main__':
    app.run(debug=True)
```

This script sets up a basic Flask application that responds with a welcome message when accessed at the root URL. The `app.run(debug=True)` line runs the application in debug mode, which is useful during development.

4.4 Deploying the Mail Detection System with Flask

In our project, Flask is used to deploy an malicuous mail detector system. The system is designed to process mails and provide detailed with the desired prediction within the description of the answer.

4.4.1 Flask Application Structure

The Flask application serves as the backbone of the system, handling image uploads, processing requests, and returning descriptions. Below is an overview of the Flask application structure:

- **Routes:** Defines different endpoints for uploading images, processing images, and returning descriptions.
- **Templates:** Uses Jinja2 for rendering HTML templates to display results.
- **Static Files:** Manages CSS, JavaScript, and images used in the web interface.
- **Error Handling:** Includes mechanisms to handle exceptions and provide meaningful error messages.

4.4.2 Implementation Example

Here is a snippet demonstrating how images are uploaded and processed using Flask:

```
from flask import Flask, render_template, request
import os
from src.malicious_mail_detector import test_model, load_model
import ast

load_model()
app = Flask(__name__)
app.config['PROCESSED_FOLDER'] = 'processed'
description = ""

@app.route('/', methods=['GET', 'POST'])
def index():
    global description
    if request.method == 'POST':
        input_sender = request.form['sender']
        input_subject = request.form['subject']
        input_text = request.form['inputText']
        if input_text:
            ...
            return render_template('index.html',
                ...
            )
        return render_template('index.html')
    return render_template('index.html')

if __name__ == '__main__':
    app.run(debug=True)
```

4.5 Showcase

Whether you installed the virtual environment on your computer or created the Docker image, once everything is set up, you will be able to test the application on your localhost. To begin, you will see the following screen:

Figure 4.3: First screen

In this app, you paste the sender, subject information, and the email content, then

press the 'Upload' button. After about 30 seconds, you will see a screen displaying the email information, an analysis of whether the email is potentially malicious or not, and a detailed explanation of the analysis. Additionally, the app will show how long it took to process the information. There's also a button to have the detailed explanation read aloud through the microphone, making it accessible for visually impaired users.

Figure 4.4: Result screen

5. Docker for Deployment

As the final step for our project, we want to deploy it. However, due to hardware and software limitations, instead of using a hosted AWS service or Microsoft Azure, we will use Docker. This way, users who download the repository can simply follow two instructions to access a virtual machine with all the required software pre-installed, ensuring it works perfectly regardless of their system. If you have Docker installed on your system and configured everything related to the Nvidia drivers and prefer not to install all the dependencies and the specific Python version required to test this app. Below is the Dockerfile that will allow us to achieve this:

```
# Change it to the version compatible with your Nvidia drivers
FROM nvidia/cuda:11.7.1-base-ubuntu20.04

WORKDIR /app

RUN apt-get update && \
    apt-get install -y \
    nano \
    python3-pip \
    python3-dev \
    && rm -rf /var/lib/apt/lists/*

# Change it to the version compatible with your Nvidia drivers
RUN pip3 install torch==1.13.1+cu117 \
                 torchvision==0.14.1+cu117 \
                 torchaudio==0.13.1+cu117 \
                 --extra-index-url
                    ↪ https://download.pytorch.org/whl/cu117

COPY . /app

RUN pip3 install --no-cache-dir -r requirements.txt

EXPOSE 5000

CMD ["python3", "app.py"]
```

With that, the user will need to run the following two commands to use the application. They also have the option to download the repository and set up a virtual environment to install all the dependencies:

```
docker build -t your_image_name .
docker run --gpus all -p 5000:5000 your_image_name
# If you did not configure the drivers use:
# docker run -it -p 5000:5000 your_image_name
```

6. Bibliography

[1] **Llama3 Repository**, *LLama3 Repository*, Aug 2024, https://github.com/meta-llama/llama3 pages

[2] **Llama2 Documentation**, *LLama2 Documentation*, Aug 2024, https://llama.meta.com/docs/get-started/ pages

[3] **Docker Documentation**, *Docker Documentation*, Aug 2024, https://docs.docker.com/ pages