AI/ML Coding Round

Train a LLM that can answer queries about JFrog Pipelines' native steps. When posed with a question like "How do I upload an artifact?" or "What step should I use for an Xray scan?", the model should list the appropriate native step(s) and provide an associated YAML for that step.

Requirements

1. *Data Collection:* Acquire publicly available information on Native Steps from JFrog's website that contain information on native steps for building pipelines. Data that is not publicly accessible falls outside the scope of this coding challenge. (https://jfrog.com/help/r/jfrog-pipelines-documentation/pipelines-steps)
2. *Data Preprocessing:* Process the text to make it suitable for training. This might involve tokenization, stemming, and other NLP techniques.
3. *Model Training:* Train a LLM on the (preprocessed) dataset. You can choose one of the freely available open source model like BERT or any other model available
4. *Query Handling:* Implement a function that takes a user query as input and returns the appropriate native step(s) and a sample YAML configuration.
5. *YAML Generation:* Implement a function that can generate a sample YAML configuration based on the identified native step(s).

Expected Output

1. A trained LLM model that can answer queries about JFrog Pipelines native steps.
2. A sample function to interact with the model, which takes a query as input and returns a list of recommended steps and a sample YAML configuration.

Bonus

● Implement an API that exposes this functionality

1. Data Extraction:

Data Understanding

1. I have started exploring the documentation sites given in the problem and noted the below points.
    - The site has multiple navigation links for various pipeline procedures.
    - Each discipline procedure has two main things, like instructions and yaml pipeline examples we can use them for training
    - If we can extract these links, we can use them to scrape the data from each document and use them for preparation.

ToDO: Link Extraction

- Extracting the pipeline documentation links from following site

https://jfrog.com/help/r/jfrog-pipelines-documentation/pipelines-steps

I have Extracted 244 links from above link and steps present in Data scraping folder

We have extracted our data and we can use jFrog_pipline.csv for training.

2. Data preparation for training:

Experiment 1

- we have got the raw data from the site for various pipelines
- we have data that can talk about the pipeline procedure and sample yaml files
- I have chosen the meta's opensource codellama-instruct-7B model for finetuning since its current best model for code completion and instruction
- Since we are using the above model we need to pre-process for below format

---

Prompt:

```
[INST] Your task is to write a Python function to solve a programming problem.
The Python code must be between [PYTHON] and [/PYTHON] tags.
You are given one example test from which you can infere the function signature.

Problem: Write a Python function to get the unique elements of a list.
Test: assert get_unique_elements([1, 2, 3, 2, 1]) == [1, 2, 3]
[/INST]
[PYTHON]
def get_unique_elements(my_list):
    return list(set(my_list))
[/PYTHON]
```

- As you can see we have know what part is YAML which we tagged inside this blocks [code][/code] we need to converts them into [YAML]tag in order extract from trained model

Experiment 1 Results:

```
# Generate Text
system_message = "You are a helpful, respectful and honest assistant. Helps user to write jFrog pipline and
answers about jFr
query = "Write a pipeline to do a Docker Build & Publish?"
text_gen = pipeline(task="text-generation",
                    model=refined_model,
                    torch_dtype=torch.float16,
                    tokenizer=llama_tokenizer,
                    max_length=200,
                    device_map='auto')
output = text_gen(f"<s>[INST]<<SYS>>{system_message} <</SYS>> {query} [/INST]",
                  do_sample=True,
                  top_k=10,
                  top_p = 0.9,
                  temperature = 0.2,
                  num_return_sequences=1,
                  eos_token_id=llama_tokenizer.eos_token_id,
                  max_length=200) # can increase the length of sequence
print(output[0]['generated_text'])
```

Loading checkpoint shards: 100% 2/2 [00:05<00:00, 2.52s/it]

```
     Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.
     <s>[INST]<<SYS>>You are a helpful, respectful and honest assistant. Helps user to write jFrog pipline
and answers about j
     ```
     pipeline {

agent any

         stages {
             stage('Build') {
                 steps {
                     sh 'docker build -t my-image.'

} }

             stage('Publish') {
                 steps {
                     sh 'docker push my-image'
                 }

} }

     }
     ```
```

Data Experiment 2

- From the First experiment we can see the model not performing well so we try to limit the data
- Remove pipeline documents where see not required for training
- we will add questions instead of splitting the data

Remarks:

- Failed due to data extracted not useful or missing some features

- will try to trim the data or only we take important procedures where clear cut answers mentioned
- will adjust the prompt and training structure as well

Data Experiment 2. Results:

```
  <s>[INST]Write a jfrog pipeline to do a docker push? [/INST]  Sure! Here is an example JFrog Pipeline
that pushes a Docke
     1. Create a new pipeline in JFrog Artifactory by going to the "Pipelines" section in the top menu and
clicking "New Pipel
     2. Give the pipeline a name, such as "Docker Push".
     3. Add a new stage to the pipeline by clicking the "Add Stage" button. Select "Docker" from the list
of available stages.
     4. In the "Docker" stage, you will need to provide the following configuration:
             * "Image": the name of the Docker image that you want to push.
             * "Repository": the name of the Docker registry where you want to push the image.
             * "Tag": the tag or label that you want to assign to the image.
             * "Push": set this to "true" to push the image to the registry.
     5. Add any additional stages to the pipeline as needed, such as a "Build" stage to build the Docker
image or a "Deploy" s
     6. Save and activate the pipeline.
     Here is an example of a JFrog Pipeline that pushes a Docker image to a registry:
     ```
     {

"stages": [{
```

### *Data Experiment 3:*

- Since we are training more complex dataset let me try to make it simple for single GPU inferences
- Experiment 3 also failed for small epochs ran on minimal time
- However same data did performed well during long running GPU training.

Will see the results in the Model training section

# 3. Model Training:

### *Experiment 1:*

- We Used Codellama2 from metas opensource model since it was out performing other open source LLMs in the market
- And we used Codellama2 Tokenizer for better results

Experiment 1 failed we have adjusted the data(Data experiment 2) and will train on new data with multiple epochs.

### *Experiment 2 :*

- We used same model with modified new data trimmed data
- With the same model used in experiment 1

Experiment 2 failed  Could be model not performing in this dataset Will try with different llama2 model and further simplified data(Data experiment 3)

### *Experiment 2 result:*

```
Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.
```
```

```
     <s>[INST]<s>[INST]<<SYS>> You are a helpful, respectful and honest assistant. Helps user to write
jFrog pipline and answe
     ```
     pipeline {

agent any

         stages {
             stage('Build') {
                 steps {
                     sh'mvn clean package'

} }

             stage('Deploy') {
                 steps {
                     sh'mvn deploy'
                 }

} }

     }
     ```
     This pipeline defines two stages: "Build" and "Deploy". The "Build" stage uses the `mvn
```

Experiment 3 :

- We Used llama2 instead of CodeLLama2 from metas opensource model.
- And we used llama2 Tokenizer for better results for this model.
- Still our data dosent get high probability and model hallucinated with the data already trained on.

  Note: hallucinated means LLM will answer from randomly from billions of data it was trained on.

```
      "image": "my-docker-image",
<s>[INST]<<SYS>>You are a helpful, respectful and honest assistant. Helps user to write jFrog pipline and
answers about j

```

```
# Define the pipeline
      "push": true,
pipeline {
      "docker": {
    agent any
        "push": "docker push my-registry/my-docker-image:latest"

}

    # Define the stages
    }
  ] stages {
        stage('Build') {

```

```
Loading checkpoint shards: 100%

2/2 [00:01<00:00, 1.37it/s]

}

steps {
    # Run the Gradle build
```

```
    sh 'gradle build'
"repository": "my-registry",
"tag": "latest",
This will push the Docker image directly to the registry without the need to specify the repository or tag.
                }
    I hope this helps! Let me know if you have any questions

} }

    }
```

Experiment 4 :

- After running long GPU hours model with Data (data experiment3) now able to shape the outputs as YAML from our data
- But still its giving wide answers like some made-up answers not completely from trained data
- Since LLMs needs to be trained more time we also need to train them on Long running servers for expected results ex: 6-8 Hours GPU runtime
- Also some data finetuning required - for better results

Note:- Due to No GPU subscription left I have stopped the further training and Inferencing

Final Results:

Q1:

<s>[INST]<<SYS>>You are a helpful, respectful and honest assistant. Helps user to write jFrog pipline and answers about jFrog pipline process<</SYS>>Write a jfrog pipeline to do a HelmBlueGreenDeploy? [/INST] Sure, I'd be happy to help you with that! Here's an example jFrog pipeline that you can use to perform a Helm BlueGreen deploy:

```yaml
# Define the pipeline stages
stages:
  - stage: prepare
    displayName: 'Prepare deployment'
    jobs:
      - job: download-helm
        displayName: 'Download Helm'
        steps:
          - name: Download Helm
            url: https://raw.githubusercontent.com/helm/helm/v2.16.0/bin/helm
            path: helm
      - job: install-helm
        displayName: 'Install Helm'
        steps:
          - name: Install Helm
            run: |
```

```
          chmod +x./helm
          ./helm

  - stage: deploy
    displayName: 'Deploy application'
    jobs:
      - job: deploy-blue
        displayName: 'Deploy Blue'
        steps:
          - name: Deploy Blue
            helm upgrade --set-image-name blue=blue:latest --set-image-port 80 blue

  - stage: deploy-green
    displayName: 'Deploy Green'
    jobs:
      - job: deploy-green
        displayName: 'Deploy Green'
        steps:
          - name: Deploy Green
            helm upgrade --set-image-name green=green:latest --set-image-port 80 green

  - stage: verify
    displayName: 'Verify deployment'
    jobs:
      - job: verify
        displayName: 'Verify deployment'
        steps:
          - name: Verify deployment
            run: |
              helm list
              helm status
```
Let me explain each stage of the pipeline:

1. `stage: prepare`: This stage is used to prepare the deployment environment. In this stage, we download the Helm binary and install it on the machine.
2. `stage: deploy`: This stage is used to deploy the application. In this stage, we use the `helm upgrade` co mmand to deploy the blue and green deployments. We use the `--set-image-name` flag to specify the name of the deployment, and the `--set-image-port` flag to specify the port that the deployment should listen on.
3. `stage: verify`: This stage is used to verify that the deployment was successful. In this stage, we use the ` helm list` and `helm status` commands to verify that the deployments are properly configured and runnin g.

Note that this is just an example pipeline, and you may need to modify it to fit your specific use case. For e xample, you may need to specify additional flags or options when using `helm upgrade`, or you may need t o modify the `helm list` and `helm status

Q:2

<s>[INST]<<SYS>>You are a helpful, respectful and honest assistant. Helps user to write jFrog pipline and answers about jFrog pipline process<</SYS>>Write a jfrog pipeline to do a GradleBuild? [/INST] Sure, I'd be happy to help you with that! Here's an example jFrog pipeline that you can use to automate a Gradle bu ild:
```yaml
# Define the pipeline
pipeline:
```

```
  - step:
      name: Checkout Code
       uses: actions/checkout@v2
      - step:
       name: Install Gradle
       uses: actions/install-gradle@v1
      - step:
       name: Run Gradle Build
       run: |
         gradle build
```
Let me explain each step in the pipeline:

1. `Checkout Code`: This step uses the `actions/checkout` action to check out the code from your repositor
y. You can specify the repository URL, branch, and other options as needed.
2. `Install Gradle`: This step uses the `actions/install-gradle` action to install Gradle on the machine runni
ng the pipeline. This is necessary because the `gradle build` command requires Gradle to be installed.
3. `Run Gradle Build`: This step runs the `gradle build` command to build your project. The `|` character is
used to pass the output of the command to the next step in the pipeline.

You can customize this pipeline to fit your needs by modifying the `uses` keywords to use different action
s or by adding additional steps as needed. For example, you might want to add a step to test your project
or to deploy it to a server.

I hope this helps! Let me know if you have any questions or need further assistance.

Q3:

<s>[INST]<<SYS>>You are a helpful, respectful and honest assistant. Helps user to write jFrog pipline and answers about jFrog pipline process<</SYS>>Write a jfrog pipeline to do a forceXrayScan? [/INST] Sure, I'd be happy to help you with that!

To create a JFrog pipeline for a forceXrayScan, you will need to perform the following steps:

1. Install the necessary dependencies:
        * `pip install jfrog-cli`
        * `pip install xray-scan`
2. Create a new JFrog pipeline file (`jfrog-pipeline.yml`) in your project directory:
```yaml
# jfrog-pipeline.yml

pipelines:
  force-xray-scan:
    - step:
       name: Install dependencies
       script:
         - pip install xray-scan
         - pip install jfrog-cli
    - step:
       name: Run forceXrayScan
       script:
         - xray-scan --force
```
Explanation:

* `pipelines`: This is the top-level key that defines the pipeline.
* `force-xray-scan`: This is the name of the pipeline.
* `- step`: This is the key that defines a step in the pipeline.

* `name`: This is the name of the step.
* `script`: This is the script that will be executed in the step.

In this example, we are installing the `xray-scan` package and the `jfrog-cli` package, and then running the `xray-scan` command with the `--force` flag.

3. Save the pipeline file and run it using the `jfrog-cli` command:
```
jfrog-cli run --pipeline-file=jfrog-pipeline.yml
```
This will execute the pipeline and run the `forceXrayScan` step.

Note: The `xray-scan` command is not included in the JFrog pipeline by default, so you will need to install i t using the `pip install` command before running the pipeline.

I hope this helps! Let

4.Model Inferencing :

- Issue With LLMs inferencing also needs to be done on GPUs. Unless we quantize the model for CPU inferencing
- Since I Don't have the GPU resources further I have created Inferencing API only and also have created API for converting LLM result to YAML file.

  Files can be found in Model Inferencing Folder.

5. Future work and Improvements :

1. If we prepare proper dataset results will show better
2. We can use RAG also for better results but model will not be trained on our data but can create synthetic answers with correct pipeline process [Reference link here]
3. We need extreme model GPU training in our current process to get expected results

6. System Requirement for Training and Inferencing

CPU : Minimum 16 GB RAM
GPU : Minimum 24 GB RAM A100 Tesla GPUs

7. References and Topics in this Study.

- QLORA
- PEFT – Performance efficient finetuning
- SFT – Supervised finetuning
- CodeLLAMA2 - Model
- LLAMA2 - Model

https://www.labellerr.com/blog/comprehensive-guide-for-fine-tuning-of-llms/
https://arxiv.org/pdf/2109.01652.pdf
https://huggingface.co/blog/codellama
https://huggingface.co/docs/peft/index
https://huggingface.co/docs/peft/conceptual_guides/lora
https://www.cloudbooklet.com/qlora-efficient-finetuning-of-quantized-llms/
https://github.com/facebookresearch/codellama/blob/main/MODEL_CARD.md