
ChatG2IA

GIIA

Jan 20, 2024

TABLE OF CONTENT:

1	Workflow Diagram Explanation	3
1.1	Benchmarking	3
1.2	Fine-tuning & Adapter Creation	4
1.3	Models with Adapters	4
1.4	Synthetic Data Generation	4
1.5	Interface	5
2	The team	7
3	FineTuning	9
3.1	FineTuning using Ludwig	9
3.2	Pushing the adapter into HuggingFace	15
3.3	How to load an adapter and attach it to the model (GPU)	16
4	Domain specific dataset	19
4.1	Dataset format for FineTuning	19
4.2	Data Merging	20
4.3	Generating Synthetic Data for Six Sigma and 5M Applications	21
4.4	Host the dataset in HuggingFace	26
5	Chainlit	31
5.1	Chainlit - hyperparameters tuning	31
5.2	Chainlit - model selection	33
5.3	Chainlit: an easy way to interact with LLMs	34
6	RAG Application	41
6.1	ConversationChain Class	41
6.2	DocumentChain Class	43
7	Deploy to Replicate	47
7.1	Overview	47
7.2	Setting Checkpoint Directory	47
7.3	Install Docker Script	47
7.4	Executing the Install Script	48
7.5	Install Replicate Cog	48
7.6	Initialize Cog	49
7.7	Define Cog Configuration	49
7.8	Define Requirements	49
7.9	Prediction Interface	49
7.10	Push to Replicate	50
7.11	Conclusion	50



WORKFLOW DIAGRAM EXPLANATION

This diagram illustrates the process of enhancing Large Language Models (LLMs) with custom adapters and integrating them into a user-friendly interface for both end-users and developers.

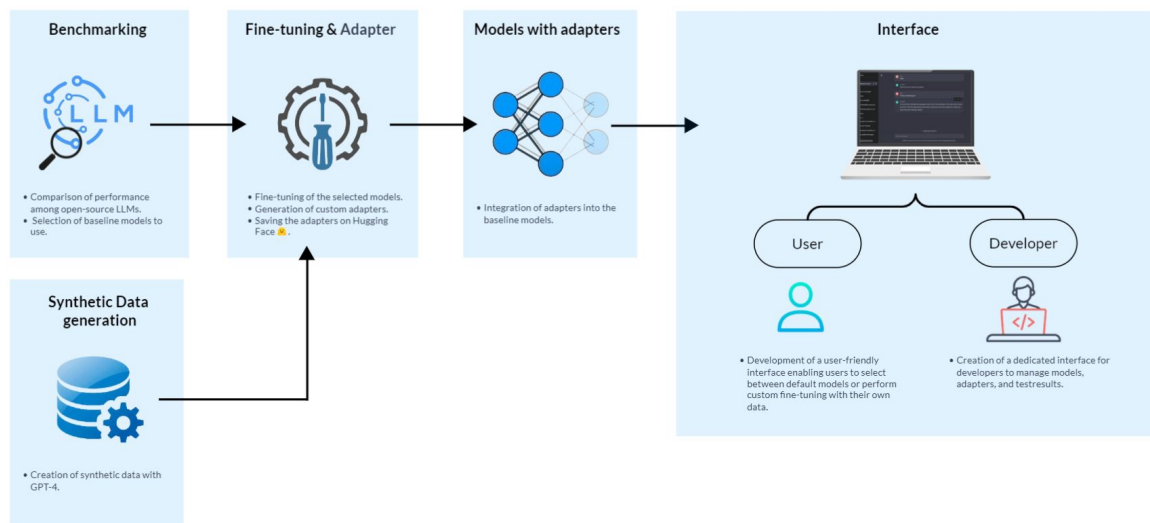


Fig. 1: The project's pipeline.

1.1 Benchmarking

The workflow begins with the benchmarking phase, where the performance of various open-source LLMs is compared. The most suitable models are selected as baseline models for further development.

	A	B	C	D	E	F	G	H	I
1	Model	Parameters	Release Date	Context length	Use cases	Training data	RAM required (32 bits)	RAM required (4bits)	Usage
2	Mistral 7B	7.3B	2023/09	8K	Low latency, text summarization, classification, text completion, code completion	A massive dataset of text and code	27.5 GB	3.5 GB	Free to use
3	Vicuna-13B	13B	2023/04	16K		User-shared conversations from ShareGPT.com	48 GB	6 GB	Personal & research use
4	OpenLLaMA	3B, 7B, 13B	2023/09	N/A	Text generation	One trillion tokens (RedPajama dataset)	N/A	N/A	Free to use
5	Falcon-7B	7B	2023/05	2048		Trained on 1.500B tokens of RefinedWeb enhanced with curated corpora	25.79 GB	3.22 GB	Free to use
6	LLaMA 2	13B	2023/04	4k	Text generation-Translation-Commonsense reasoning-Code generation	2 trillion tokens of data from publicly available sources	19GB (V-RAM for GPTQ)	2.5GB	Free to use
7	GPT4ALL	7B, 13B	2023/03	2048	Text generation	trained on a large-scale data distillation from GPT-3.5-Turbo	4GB-16GB	4GB	Free to use
8	MPT-7B	7B	2023/05	84K	StoWritting, Chat,	Trained on 1T tokens of text and code that was curated by MosaicML's data team			Commercial use
9	T5	11B	2019/10	512	machine translation, document summarization, question answering, and classification tasks	The model is pre-trained on the Colossal Clean Crawled Corpus (C4).			Commercial use
10	RedPajama-INCITE	7B	2023/05	2048		One trillion tokens (RedPajama dataset)	16 GB	4 GB	Commercial use
11	OpenLM	1.7B	2023/09	2048	real time monitoring & self audit system for software licenses, Reporting, Ensuring compliance with software licenses				
12	OpenLLama (Meta)	3.7B	2023/05	8192	text generation	a mixture of Falcon refined-web dataset, mixed with the starcoder dataset, and the wikipedia, arxiv and books and stackexchange from RedPajama			commercial & research usage
13	FastChat-T5	3B	2023/04	512	Commercial usage of large language models and chatbots and research purposes	trained on 70K user-shared conversations collected from ShareGPT.com by fine-tuning Flan-t5-xl			free to use
14	GPT-J-6B	6B	2021/06	2048	Text generation from a prompt (support only english)	The Pile is a 825 GiB diverse, open source language modelling data set that consists of 22 smaller, high-quality datasets combined together.	22GB	2.75GB	free to use
15	StableLM-3B-4E1T	3B	2023/09	4096		The dataset is comprised of a filtered mixture of open-source large-scale datasets available on the HuggingFace Hub: Falcon RefinedWeb extract, RedPajama-Data and The Pile, both without the Books3 subset, and StarCoder.			
16	Bloom	176B	2022/11	2048	Text generation, Exploring characteristics of language generated by a language model				
17	h2oGPT	12-20B	2023/05	256-2048					
18	RWKV	0.1-14B	2021/08	1024					

Fig. 2: Benchmarking various open source LLMs.

1.2 Fine-tuning & Adapter Creation

Next, the selected models undergo fine-tuning to optimize them for specific tasks. Custom adapters are then created to augment the LLMs' abilities, which are saved on the Hugging Face platform, a hub for sharing machine learning models.

Hint: For more details, refer to the *FineTuning ludwig tutorial*.

1.3 Models with Adapters

These custom adapters are integrated into the baseline models. The integration allows for a more modular approach to machine learning model enhancement, where specific capabilities can be added without altering the entire model architecture.

Hint: For more details, refer to the *Load adapter and attach to model tutorial*.

1.4 Synthetic Data Generation

In parallel to adapter integration, synthetic data is generated using GPT-4. This data can be used to further train and refine the models, ensuring that they are well-equipped to handle a variety of scenarios.

Hint: For more details, refer to the *Synthetic data tutorial*.

1.5 Interface

Finally, the models with adapters are made accessible through two distinct interfaces:

- User Interface Designed for end-users, this interface is user-friendly and allows users to select between default models or to perform custom fine-tuning with their own data.
- Developer Interface Tailored for developers, this interface provides the tools needed to manage models, adapters, and test results effectively.



This diagram encapsulates the streamlined approach to adapting LLMs to specialized tasks and ensuring that both users and developers have the necessary tools at their disposal.

Note: In the final version of the interface, the Developer Interface will not be present, as the system is designed to be user-centric without requiring direct developer involvement.



THE TEAM

The team consists of 8 individuals, organized into four pairs. They operated under the guidance of Tawfik Masrour.



- **Group 1:**

- **Member 1:** Fatima Zahra Hannou 
- **Member 2:** Achraf Jaanine 



- **Group 2:**

- **Member 1:** Saddik Imad 
- **Member 2:** Drissi EL Bouzaïdi Karim 

- **Group 3:**

- **Member 1:** Bilal El Manja 
- **Member 2:** Hafsa Ihrouchen 

- **Group 4:**

- **Member 1:** Badreddine Hannaoui 
- **Member 2:** Elghali Blouz 

FINETUNING

3.1 FineTuning using Ludwig

Fine-tuning allows the adaptation of pre-trained LLMs to a specific task by updating model weights, leading to performance improvements. It is a means to personalize these general models for specialized applications, optimizing performance for unique tasks.

3.1.1 1. Traditional FineTuning Vs. Parameter Efficient FineTuning

Traditional fine-tuning involves updating all model parameters, a process proven to be resource-intensive, time-consuming, and not always yield optimal task-specific performance. However, recent innovations in parameter-efficient fine-tuning have offered a breakthrough. By freezing the pre-trained LLM and only training a very small set of task-specific layers—less than 1% of the total model weight—efficient fine-tuning proves to be both resource-friendly and more effective.

Traditional Fine-Tuning vs Parameter Efficient Fine-Tuning

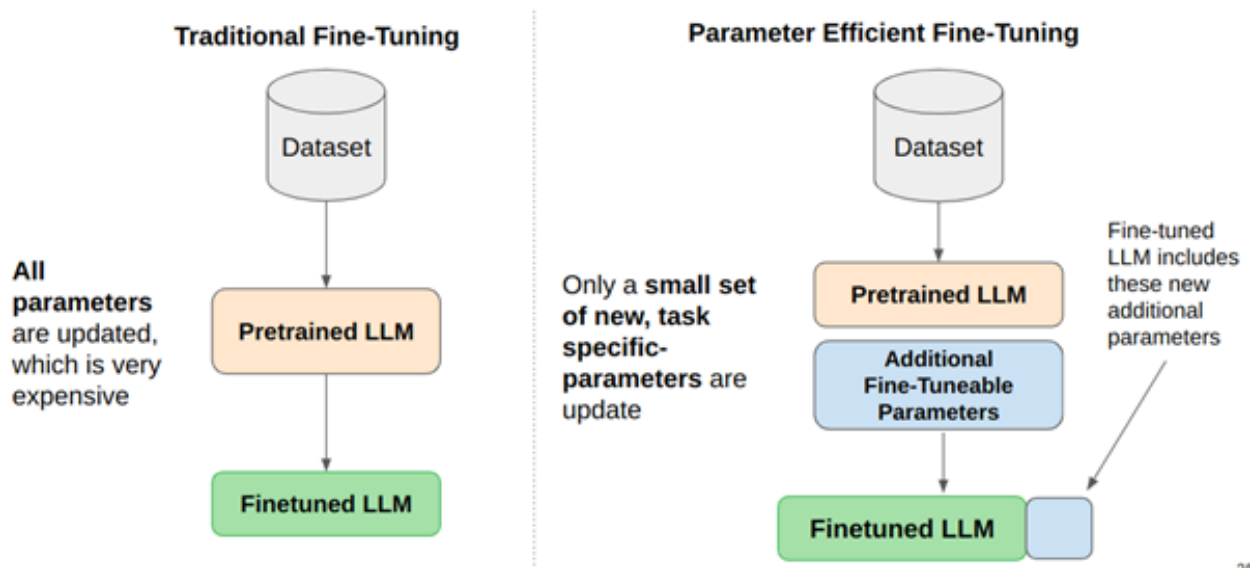


Fig. 1: Traditional FineTuning Vs. Parameter Efficient FineTuning.

3.1.2 1. Fine-tuning using Ludwig

Ludwig offers a declarative approach to machine learning, providing an accessible interface to control and customize models without extensive coding. Its YAML-based configurations empower users to manage different input features and output tasks efficiently. Imagine a world where we feed the model a prompt, pair it with specific instructions and context, and let the magic happen. The prompt acts as a guide, steering the model's understanding of the task at hand. And this is where Ludwig's advanced features come into play.

Attention: This toolkit a single memory-constrained GPU, including: LoRA and 4-bit quantization.

Now, let's delve deeper into the nitty-gritty of advanced configuration and the fine-tuning parameters that Ludwig offers.

3.1.3 1.1 Install Ludwig and Ludwig's LLM related dependencies

```
pip uninstall -y tensorflow --quiet
pip install ludwig --quiet
pip install ludwig[llm] --quiet
pip install datasets
```

```
from IPython.display import HTML, display

def set_css():
    display(HTML(''
<style>
    pre {
        white-space: pre-wrap;
    }
</style>
''))

get_ipython().events.register('pre_run_cell', set_css)

def clear_cache():
    if torch.cuda.is_available():
        torch.cuda.empty_cache()
```

3.1.4 1.2. Import The Code Generation Dataset

```
from google.colab import data_table; data_table.enable_dataframe_formatter()
import numpy as np; np.random.seed(123)
import pandas as pd

df = dataset['train'].to_pandas()

# We're going to create a new column called `split` where:
# 90% will be assigned a value of 0 -> train set
# 5% will be assigned a value of 1 -> validation set
# 5% will be assigned a value of 2 -> test set
# Calculate the number of rows for each split value
```

(continues on next page)

(continued from previous page)

```

total_rows = len(df)
split_0_count = int(total_rows * 0.9)
split_1_count = int(total_rows * 0.05)
split_2_count = total_rows - split_0_count - split_1_count

# Create an array with split values based on the counts
split_values = np.concatenate([
    np.zeros(split_0_count),
    np.ones(split_1_count),
    np.full(split_2_count, 2)
])

# Shuffle the array to ensure randomness
np.random.shuffle(split_values)

# Add the 'split' column to the DataFrame
df['split'] = split_values
df['split'] = df['split'].astype(int)

n_rows = 5000
df = df.head(n=n_rows)

```

Understanding The Code Alpaca Dataset

```
df.head(10)
```

This is how the dataset looks like:

index	instruction	input	output	prompt	split
0	Create a function to calculate the sum of a sequence of integers.	[1, 2, 3, 4, 5]	# Python code def sum_sequence(sequence): sum = 0 for num in sequence: sum += num return sum	Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction: Create a function to calculate the sum of a sequence of integers. ### Input: [1, 2, 3, 4, 5] ### Output: # Python code def sum_sequence(sequence): sum = 0 for num in sequence: sum += num return sum	0
1	Generate a Python code for crawling a website for a specific type of data.	website: www.example.com data to crawl: phone numbers	import requests import re def crawl_website_for_phone_numbers(website): response = requests.get(website) phone_numbers = re.findall(r'\d{3}-\d{3}-\d{4}', response.text) return phone_numbers if __name__ == '__main__': print(crawl_website_for_phone_numbers(www.example.com))	Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction: Generate a Python code for crawling a website for a specific type of data. ### Input: website: www.example.com data to crawl: phone numbers ### Output: import requests import re def crawl_website_for_phone_numbers(website): response = requests.get(website) phone_numbers = re.findall(r'\d{3}-\d{3}-\d{4}', response.text) return phone_numbers if __name__ == '__main__': print(crawl_website_for_phone_numbers(www.example.com))	0
2	Create a Python list comprehension to get the squared values of a list [1, 2, 3, 5, 8, 13]		[x*x for x in [1, 2, 3, 5, 8, 13]]	Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction: Create a Python list comprehension to get the squared values of a list [1, 2, 3, 5, 8, 13] ### Input: [1, 2, 3, 5, 8, 13] ### Output: [x*x for x in [1, 2, 3, 5, 8, 13]]	0
3	Generate a python script to perform this action.	Given a string, remove all the consecutive duplicates from the string. Input: "AABBBCCCD"	def remove_duplicates(string): result = "" prev = "" for char in string: if char != prev: result += char prev = char return result result = remove_duplicates("AABBBCCCD") print(result)	Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction: Generate a python script to perform this action. ### Input: Given a string, remove all the consecutive duplicates from the string. Input: "AABBBCCCD" ### Output: def remove_duplicates(string): result = "" prev = "" for char in string: if char != prev: result += char prev = char return result result = remove_duplicates("AABBBCCCD") print(result)	0
4	Write a python script to generates random numbers between 0 and 9 that are divisible by 3.		def generate_random_divisible_number(): import random while True: # Generate a random number process = random.randint(0, 9) # Check if the number is divisible by 3 if process % 3 == 0: # If it is divisible, return it return process	Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction: Write a python script to generates random numbers between 0 and 9 that are divisible by 3. ### Input: ### Output: def generate_random_divisible_number(): import random while True: # Generate a random number process = random.randint(0, 9) # Check if the number is divisible by 3 if process % 3 == 0: # If it is divisible, return it return process	0
5	Write a Python code to get the third largest element in a given row.	[12, 13, 13, 45, 22, 99]	def third_largest(lst): if len(lst) < 3: return distinct = [] for i in lst: if i not in distinct: distinct.append(i) distinct.sort(reverse=True) return distinct[2]	Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction: Write a Python code to get the third largest element in a given row. ### Input: [12, 13, 13, 45, 22, 99] ### Output: def third_largest(lst): if len(lst) < 3: return distinct = [] for i in lst: if i not in distinct: distinct.append(i) distinct.sort(reverse=True) return distinct[2]	0
6	Generate a REST API with Python and Flask that allows users to create, read, update, and delete records from a database.	Not applicable	from flask import Flask, request from flask_sqlalchemy import SQLAlchemy app = Flask(__name__) app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///tmp/test.db' db = SQLAlchemy(app) class Record(db.Model): id = db.Column(db.Integer, primary_key=True) name = db.Column(db.String(120), unique=True) def create_all(): @app.route('/records', methods=['GET']) def get_records(): records = Record.query.all() return {'records': [record.name for record in records]} @app.route('/records', methods=['POST']) def create_record(): record = Record(name=request.json['name']) db.session.add(record) db.session.commit() return {'id': record.id} @app.route('/records/int/{id}', methods=['PUT']) def update_record(id): record = Record.query.get(id) record.name = request.json['name'] db.session.commit() return {'message': 'Record updated'} @app.route('/records/int/{id}', methods=['DELETE']) def delete_record(id): record = Record.query.filter_by(id=id).delete() db.session.commit() return {'message': 'Record deleted'} if __name__ == '__main__': app.run(debug=True)	Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction: Generate a REST API with Python and Flask that allows users to create, read, update, and delete records from a database. ### Input: Not applicable ### Output: from flask import Flask, request from flask_sqlalchemy import SQLAlchemy app = Flask(__name__) app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///tmp/test.db' db = SQLAlchemy(app) class Record(db.Model): id = db.Column(db.Integer, primary_key=True) name = db.Column(db.String(120), unique=True) def create_all(): @app.route('/records', methods=['GET']) def get_records(): records = Record.query.all() return {'records': [record.name for record in records]} @app.route('/records', methods=['POST']) def create_record(): record = Record(name=request.json['name']) db.session.add(record) db.session.commit() return {'id': record.id} @app.route('/records/int/{id}', methods=['PUT']) def update_record(id): record = Record.query.get(id) record.name = request.json['name'] db.session.commit() return {'message': 'Record updated'} @app.route('/records/int/{id}', methods=['DELETE']) def delete_record(id): record = Record.query.filter_by(id=id).delete() db.session.commit() return {'message': 'Record deleted'} if __name__ == '__main__': app.run(debug=True)	0

Fig. 2: A look at the Code Alpaca dataset.

This dataset is meant to train a large language model to following instructions to produce code from natural language. Each row in the dataset consists of an:

- instruction that describes a task
- input when additional context is required for the instruction, and
- the expected output.

This is a script that calculates various statistics for token distributions in different columns of a DataFrame.

```
from transformers import AutoTokenizer
import numpy as np

def calculate_distribution(data_dict):
    result = {}

    for key, values in data_dict.items():
        values = np.array(values)
        result[key] = {
            'average': int(np.mean(values)),
            'min': np.min(values),
            'max': np.max(values),
            'median': np.median(values),
            '75th_percentile': int(np.percentile(values, 75)),
            '90th_percentile': int(np.percentile(values, 90)),
            '95th_percentile': int(np.percentile(values, 95)),
            '99th_percentile': int(np.percentile(values, 99))
        }

    return result

tokenizer = AutoTokenizer.from_pretrained('HuggingFaceH4/zephyr-7b-beta')

token_counts = {
    "instruction": [],
    "input": [],
    "output": [],
    "total": []
}

for index, row in df.iterrows():
    instruction_col_tokens = len(tokenizer.tokenize(row['instruction']))
    input_col_tokens = len(tokenizer.tokenize(row['input']))
    output_col_tokens = len(tokenizer.tokenize(row['output']))
    total = instruction_col_tokens + input_col_tokens + output_col_tokens

    token_counts['instruction'].append(instruction_col_tokens)
    token_counts['input'].append(input_col_tokens)
    token_counts['output'].append(output_col_tokens)
    token_counts['total'].append(total)

token_distribution = calculate_distribution(token_counts)
token_distribution = pd.DataFrame(token_distribution)
token_distribution
```


3.1.5 1.3. Setup Your HuggingFace Token

```
pip install --upgrade git+https://github.com/huggingface/peft.git --quiet
```

```
import getpass
import locale; locale.getpreferredencoding = lambda: "UTF-8"
import logging
import os
import torch
import yaml

from ludwig.api import LudwigModel

os.environ["HUGGING_FACE_HUB_TOKEN"] = "hf_pqfaqdjH0kfTuzLFsxIqBUrBkZiTYjOrUe"
assert os.environ["HUGGING_FACE_HUB_TOKEN"]
```

3.1.6 1.4. Fine-tuning

Note: We're going to fine-tune using a single T4 GPU with 16GiB of GPU VRAM on Colab. To do this, the new parameters we're introducing are:

- adapter: The PEFT method we want to use
- quantization: Load the weights in int4 or int8 to reduce memory overhead.
- trainer: We enable the finetune trainer and can configure a variety of training parameters such as epochs and learning rate.

```
qlora_fine_tuning_config = yaml.safe_load(
    """
model_type: llm
# We use a resharded model here since the base model does not have safetensors support.
base_model: HuggingFaceH4/zephyr-7b-beta

input_features:
- name: instruction
  type: text

output_features:
- name: output
  type: text

prompt:
template: >-
  Below is an instruction that describes a task, paired with an input
  that may provide further context. Write a response that appropriately
  completes the request.

  ### Instruction: {instruction}

  ### Input: {input}
```

(continues on next page)

(continued from previous page)

```

    ### Response:

generation:
temperature: 0.1
max_new_tokens: 256

adapter:
type: lora

quantization:
bits: 4

preprocessing:
global_max_sequence_length: 256
split:
    type: random
    probabilities:
        - 0.9 # train
        - 0.05 # val
        - 0.05 # test

trainer:
type: finetune
epochs: 1
batch_size: 1
eval_batch_size: 2
gradient_accumulation_steps: 16
learning_rate: 0.0004
learning_rate_scheduler:
    warmup_fraction: 0.03
"""
)

model = LudwigModel(config=qlora_fine_tuning_config, logging_level=logging.INFO)
results = model.train(dataset=df)

```

Perform Inference We can now use the model we fine-tuned above to make predictions on some test examples to see whether fine-tuning the large language model improve its ability to follow instructions/the tasks we’re asking it to perform.

```

df[['instruction', 'input']].iloc[-n_rows:].shape
test_df = df[['instruction', 'input']].iloc[-n_rows:]
test_df = test_df.head(20)
predictions = model.predict(test_df)[0]
result_df = test_df.copy()
result_df['Output'] = df['output'].iloc[-n_rows: -n_rows + test_df.shape[0]].values
result_df['Generated_output'] = predictions['output_response']
result_df

```

This is how the dataset looks like:

index	instruction	input	Output	Generated_output
0	Create a function to calculate the sum of a sequence of integers.	[1, 2, 3, 4, 5]	# Python code def sum_sequence(sequence): sum = 0 for num in sequence: sum += num return sum	def sum_sequence(sequence): sum = 0 for num in sequence: sum += num return sum
1	Generate a Python code for crawling a website for a specific type of data.	website: www.example.com data to crawl: phone numbers	import requests import re def crawl_website_for_phone_numbers(website): response = requests.get(website) phone_numbers = re.findall(r'\d{3}-\d{4}', response.text) return phone_numbers if __name__ == '__main__': print(crawl_website_for_phone_numbers('www.example.com'))	import requests from bs4 import BeautifulSoup url = 'http://www.example.com' response = requests.get(url) soup = BeautifulSoup(response.text, 'html.parser') phone_numbers = [] for phone_number in soup.find_all('span', {'class': 'phone-number'}): phone_numbers.append(phone_number.text)
2	Create a Python list comprehension to get the squared values of a list [1, 2, 3, 5, 8, 13].		[x**2 for x in [1, 2, 3, 5, 8, 13]]	[x**2 for x in [1, 2, 3, 5, 8, 13]]
3	Generate a python script to perform this action.	Given a string, remove all the consecutive duplicates from the string. Input: "AAABBBCCCD"	def remove_duplicates(string): result = "" prev = "" for char in string: if char != prev: result += char prev = char return result result = remove_duplicates("AAABBBCCCD") print(result)	def remove_duplicates(s): result = "" for i in range(len(s)): if i == 0 or s[i] != s[i-1]: result += s[i] return result s = "aaabbbcccd" print(remove_duplicates(s))
4	Write a python script to generates random numbers between 0 and 9 that are divisible by 3.		def generate_random_divisible_number(): import random while True: # Generate a random number process = random.randint(0, 9) # Check if the number is divisible by 3 if process % 3 == 0: # If it is divisible, return it return process	import random for i in range(10): num = random.randint(0, 9) if num % 3 == 0: print(num)
5	Write a Python code to get the third largest element in a given row.	[12, 13, 14, 25, 22, 99]	def third_largest(lst): if len(lst) < 3: return distinct = [] for i in lst: if i not in distinct: distinct.append(i) distinct.sort(reverse=True) return distinct[2]	def get_third_largest(row): row.sort() return row[-3]
6	Generate a REST API with Python and Flask that allows users to create, read, update, and delete records from a database.	Not applicable	from flask import Flask, request from flask_sqlalchemy import SQLAlchemy app = Flask(__name__) app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///tmp/test.db' db = SQLAlchemy(app) class Record(db.Model): id = db.Column(db.Integer, primary_key=True) name = db.Column(db.String(120), unique=True) db.create_all() @app.route('/records', methods=['GET']) def get_records(): records = Record.query.all() return [record.name for record in records] @app.route('/records', methods=['POST']) def create_record(): record = Record(name=request.json['name']) db.session.add(record) db.session.commit() return {'id': record.id} @app.route('/records/<int:id>', methods=['PUT']) def update_record(id): record = Record.query.get(id) record.name = request.json['name'] db.session.commit() return {'message': 'Record updated'} @app.route('/records/<int:id>', methods=['DELETE']) def delete_record(id): Record.query.filter_by(id=id).delete() db.session.commit() return {'message': 'Record deleted'} if __name__ == '__main__': app.run(debug=True)	from flask import Flask, request, jsonify from flask_sqlalchemy import SQLAlchemy app = Flask(__name__) app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db' db = SQLAlchemy(app) class User(db.Model): id = db.Column(db.Integer, primary_key=True) name = db.Column(db.String(100), nullable=False) email = db.Column(db.String(100), nullable=False) password = db.Column(db.String(100), nullable=False) @app.route('/users', methods=['POST']) def create_user(): user = User(name=request.json['name'], email=request.json['email'], password=request.json['password']) db.session.add(user) db.session.commit() return jsonify({'id': user.id, 'message': 'User created'}) @app.route('/users/<int:id>', methods=['GET']) def get_user(id): user = User.query.get(id) return jsonify({'id': user.id, 'name': user.name, 'email': user.email}) @app.route('/users', methods=['PUT']) def update_user(): user = User.query.get(request.json['id']) user.name = request.json['name'] user.email = request.json['email'] user.password = request.json['password'] db.session.commit() return jsonify({'id': user.id, 'message': 'User updated'}) @app.route('/users/<int:id>', methods=['DELETE']) def delete_user(id): User.query.filter_by(id=id).delete() db.session.commit() return jsonify({'id': id, 'message': 'User deleted'}) if __name__ == '__main__': app.run(debug=True)

Fig. 3: A look at the dataset with the generated output.

Note: he inference outputs may not be perfect, especially if the fine-tuning epochs are limited. However, by tweaking parameters like generation configuration (temperature, maximum new tokens, etc.), the outputs can be altered, thereby refining the model's responses.

3.2 Pushing the adapter into HuggingFace

In this section, we will show you how to push your adapter to Hugging Face. This will allow you to easily access your adapter from the Hugging Face Hub and use it in your own projects.

Note: This section assumes that you have already went through the fine-tuning using the Ludwig library and that you have a Hugging Face account. If you have not done so, please go back to this section first (2.3 fine-tuning using Ludwig).

3.2.1 Ludwig API

Assuming that you have used Ludwig to fine-tune an open source llm of your choice. At the end of training, a new folder will be created called **results**. This is an example of what you will see after the training is done:

```
results
├── api_experiment_run_x  # x refers to the training run (integer)
│   ├── config.yaml
│   ├── model
│   │   ├── pytorch_model.bin
│   │   ├── special_tokens_map.json
│   │   ├── tokenizer_config.json
│   │   ├── training_args.bin
│   │   └── vocab.json
│   ├── training.log
│   └── training_results.json
└── llm_adapter
```

(continues on next page)

(continued from previous page)

```
├── config.json
├── pytorch_adapter.bin
└── training_args.bin
```

Now to push your adapter to Hugging Face, you will need to run the following command:

```
ludwig upload hf_hub \
  --repo_id {profile/repo_name} \
  --model_path /content/results/api_experiment_run
```

To be able to run this command, you will need to create a Hugging Face token. To do so, go to your Hugging Face profile page and click on the **Access Tokens** button, then click on the **New token** button to generate your token. Then, copy the token and paste it in the console after running the command.

repo_id refers to the name of your repository on Hugging Face. For example, if you want to push your adapter to the following [repository](#), then the repo_id will be Azulian/DoctorLLM2.

3.3 How to load an adapter and attach it to the model (GPU)

To enhance the capabilities of your model, you can load pre-trained adapters and attach them. This process involves loading the adapter, previously pushed to the Hugging Face Model Hub, and then attaching it to your existing model. Follow these steps to achieve this integration:

3.3.1 1. Install Required Packages

Before proceeding, make sure to install the necessary Python packages. Open your terminal, command prompt, or a code cell in your preferred environment (e.g., Jupyter Notebook, Google Colab) and run the following commands:

```
!pip install peft
!pip install bitsandbytes
!pip install transformers
!pip install accelerate
```

This ensures that you have the required dependencies installed.

3.3.2 2. Load and Configure the Model

Note: Before loading the adapter, ensure the model is loaded. If it's already loaded, proceed to the next stage. If not, execute the following Python code:

```
import bitsandbytes as bnb
import transformers
import torch

from peft import (
    LoraConfig,
    PeftConfig,
    PeftModel,
```

(continues on next page)

(continued from previous page)

```
get_peft_model,
prepare_model_for_kbit_training
)
```

```
from transformers import (
    AutoConfig,
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig
)
```

```
model_name = "alexsherstinsky/Mistral-7B-v0.1-sharded" # This model is just an example.
↳ You can use any model you want from the Hugging Face Model Hub.
```

```
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)
```

```
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    device_map="auto",
    trust_remote_code=True,
    quantization_config=bnb_config
)
```

```
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token
```

The following code defines a prompt generating a response based on a given instruction and input. (It's just an example).

```
instruction = "Create a function to calculate the sum of a sequence of integers."
input = "[1, 2, 3, 4, 5]"
```

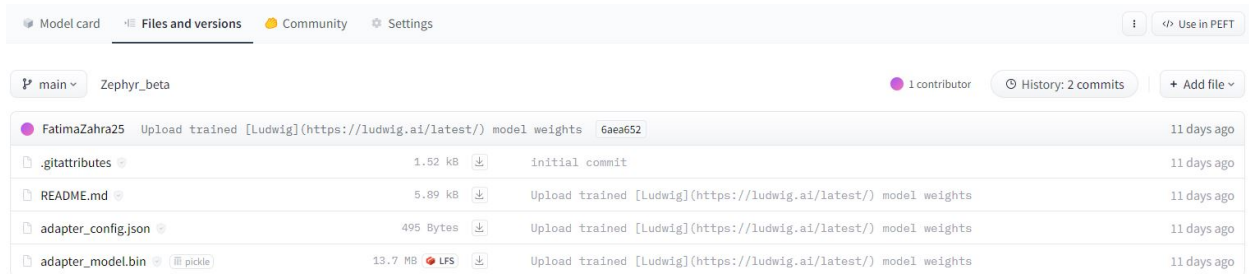
```
prompt = f"""Below is an instruction that describes a task. Write a response that
↳ appropriately completes the request.
### Instruction: {instruction}
### Input: {input}
### Response:
"""
```

```
encodeds = tokenizer(prompt, return_tensors="pt", add_special_tokens=False)
model_inputs = encodeds.to("cuda")
```

3.3.3 3. Load adapters with PEFT

Parameter-Efficient Fine Tuning (PEFT) methods freeze the pretrained model parameters during fine-tuning and add a small number of trainable parameters (the adapters) on top of it. The adapters are trained to learn task-specific information.

To load and use a PEFT adapter model from Transformers, make sure the Hub repository or local directory contains an adapter_config.json file and the adapter weights, as shown in the example image above.



Then you can load the PEFT adapter model using the code below:

```
peft_model_id = "FatimaZahra25/Zephyr_beta"
model.load_adapter(peft_model_id)
```

Enable and Disable the Adapter

Once you've added an adapter to a model, you can enable or disable the adapter module. To enable the adapter module:

```
model.enable_adapters()
```

```
generated_ids = model.generate(**model_inputs, max_new_tokens=1000, do_sample=True)
decoded = tokenizer.batch_decode(generated_ids)
print(decoded[0])
```

To disable the adapter module:

```
model.disable_adapters()
```

Attention: The adapter functionality currently utilizes GPU memory.

DOMAIN SPECIFIC DATASET

4.1 Dataset format for FineTuning

Before pushing your dataset, you need to make sure that it is in the right format. The format that we will be using is called the **Alpaca format**. The dataset should be a json file containing the following columns:

- instruction : the instruction to execute
- input : the input of the instruction
- output : the output of the instruction

Here is an example of a dataset in the right format:

```
[
  {
    "instruction": "Assess the company's waste management practices.",
    "input": "How does your company manage hazardous and non-hazardous waste?",
    "output": "We segregate hazardous and non-hazardous waste at the source.
    ↪ Hazardous waste is handled by a licensed contractor, and we regularly audit their
    ↪ practices for compliance. Non-hazardous waste is recycled where possible."
  },
  {
    "instruction": "Evaluate the company's water usage efficiency.",
    "input": "What measures has your company implemented to reduce water consumption?
    ↪ ",
    "output": "We have installed low-flow fixtures and automated sensors in our
    ↪ facilities. We also use a water recycling system in our manufacturing process to
    ↪ reduce overall water consumption."
  },
  {
    "instruction": "Check the company's energy efficiency initiatives.",
    "input": "What steps has your company taken to improve energy efficiency?",
    "output": "We've upgraded to energy-efficient lighting, HVAC systems, and
    ↪ machinery. Additionally, we conduct regular energy audits to identify further
    ↪ improvement areas."
  },
  {
    "instruction": "Review the company's air emissions control.",
    "input": "How does your company manage and monitor air emissions?",
    "output": "We have installed emission control systems like scrubbers and
    ↪ electrostatic precipitators. We conduct regular emissions testing and maintain records.
  }
]
```

(continues on next page)

(continued from previous page)

```
↪as per regulatory requirements."  
    }  
]
```

4.2 Data Merging

4.2.1 Overview

This document details the code and its functionalities in the Jupyter Notebook designed to prepare and upload a dataset to Hugging Face's hub.

4.2.2 Unzipping Data

```
!unzip /content/data_LLM.zip
```

Comment: Unzips the 'data_LLM.zip' file, ensuring the raw data is accessible for processing.

4.2.3 Installing Datasets Package

```
!pip install datasets
```

Comment: Installs the 'datasets' package necessary for efficient data handling and processing in Python.

4.2.4 Merging JSON Files into JSONL

```
import os  
import json  
import glob  
  
directory = "/content/data_LLM"  
output_jsonl_filename = "merged_dataset.jsonl"  
json_pattern = os.path.join(directory, '*.json')  
file_list = glob.glob(json_pattern)  
  
with open(output_jsonl_filename, 'w') as outfile:  
    for file in file_list:  
        with open(file, 'r') as f:  
            json_obj = json.load(f)  
            outfile.write(json.dumps(json_obj) + '\n')
```

Comment: Reads multiple JSON files from the specified directory and merges them into a single JSONL file, creating a unified dataset structure.

4.2.5 Loading Dataset

```
from datasets import Dataset, Features, Value, ClassLabel, Sequence, load_dataset

jsonl_file_path = output_jsonl_filename
dataset = load_dataset('json', data_files=jsonl_file_path)
print(dataset['train'][0])
```

Comment: Loads the merged JSONL file as a dataset using the ‘datasets’ library and prints the first entry for verification.

4.2.6 Authentication for Hugging Face

```
from huggingface_hub import notebook_login
notebook_login()
```

Comment: Prompts for Hugging Face authentication, ensuring secure access for uploading the dataset.

4.2.7 Pushing to Hugging Face

```
dataset.push_to_hub("badreddine_LLM_data")
```

Comment: Pushes the prepared dataset to Hugging Face’s hub under the specified repository name, making it available for global access.

4.2.8 Conclusion

This document provided a step-by-step guide to the notebook’s process for preparing and uploading a dataset to Hugging Face’s hub.

4.3 Generating Synthetic Data for Six Sigma and 5M Applications

In this section, we will explore the process of generating synthetic data using GPT-4 to fine-tune our LLMs (llama2, Zephyr, Falcon, Mistral) for industrial applications. particularly in the Six Sigma and 5M domain.

4.3.1 Synthetic Data Generation Process

Our decision to leverage synthetic data is driven by the need for a controlled and diverse dataset that encompasses a wide range of Six Sigma and 5M scenarios. Here is the syntax given to GPT-4 for generating the synthetic data:

Name

Generate data

Description

generate data fot training LLMs in the field of fishbone diagram 5M

Instructions

you will generate data , i will give you the domain about the data and i want a json object as output :
 {instruction : "" ,\n context : "" ,\n response: ""} , output as a code so i can copy paste easly
 please make sure that there is no ' , ' between json and json , and make \n between them.

example : {

"instruction": "Analyze the provided workplace scenario and identify potential issues categorized under

4.3.2 Diversity Considerations

- GPT-4 is employed to create synthetic scenarios that span different industrial settings, from manufacturing to service industries.
- Emphasis is placed on simulating a variety of Six Sigma and 5M challenges, ensuring the models are exposed to a broad spectrum of scenarios.

4.3.3 Instruction Crafting

- Instructions are carefully crafted to guide GPT-4 in generating responses aligned with Six Sigma and 5M principles.
- Instructions cover scenarios related to DMAIC methodologies, 5S principles, Voice of the Customer analysis, and other Six Sigma, 5M concepts.

4.3.4 Quality Control

- A rigorous quality control process is implemented to ensure the synthetic data's relevance and coherence.
- Validation against real-world scenarios is performed to guarantee that the synthetic data aligns with actual industrial challenges.

4.3.5 Data Integration with LLMs

Once synthetic data is generated, it undergoes integration with our LLMs for fine-tuning.

4.3.6 Data Input Format

- Synthetic data is formatted to match the input requirements of llama2, Zephyr, Falcon, and Mistral or other large language model.
- The format ensures that the synthetic data seamlessly integrates with each LLM's unique architecture.

4.3.7 Fine-tuning Process

- The synthetic data is utilized in the fine-tuning process, exposing the LLMs to a diverse set of scenarios.
- Iterative fine-tuning sessions are conducted, allowing the models to adapt and learn from the synthetic data.

4.3.8 Validation

- Models are rigorously validated against both real-world and synthetic scenarios to assess their performance.
- The validation process ensures that the LLMs effectively generalize their knowledge from synthetic data to real-world industrial challenges.

4.3.9 Six Sigma Domain Integration

Understanding the Six Sigma domain is crucial for ensuring the LLMs produce meaningful and relevant outputs aligned with industry best practices.

4.3.10 Understanding Six Sigma

- Six Sigma is a data-driven methodology for process improvement, emphasizing defect reduction and efficiency enhancement.
- Key principles include DMAIC (Define, Measure, Analyze, Improve, Control), 5S methodology, and continuous improvement.

4.3.11 Customization for Six Sigma

- LLMs are tailored to understand and respond to Six Sigma-related prompts, ensuring alignment with industry standards.
- Fine-tuning involves exposure to diverse Six Sigma scenarios, allowing models to adapt their responses accordingly.

4.3.12 Quality Metrics

- Six Sigma metrics, including defect rates, process efficiency, and customer satisfaction, play a crucial role in evaluating LLM performance.
- The integration ensures that LLM-generated solutions are measurable and aligned with Six Sigma quality standards.

4.3.13 Examples

Explore examples of synthetic data generation and the subsequent integration with LLMs for Six Sigma scenarios. Here is a glimpse to six sigma and 5M dataset:

1. Six sigma dataset




input string · lengths 	instruction string · lengths 	output string · lengths 
The assembly line for electronic devices is experiencing frequent delays due to...	As a Six Sigma expert, suggest a strategy to enhance the efficiency of a production...	Implement a DMAIC (Define, Measure, Analyze, Improve, Control) approach...
Our car manufacturing plant is seeing a high percentage of material waste,...	Provide a Six Sigma based solution for reducing material waste in manufacturing.	Apply the 5S methodology (Sort, Set in order, Shine, Standardize, Sustain) to...
The assembly line for electronic devices is experiencing frequent delays due to...	As a Six Sigma expert, suggest a strategy to enhance the efficiency of a production...	Implement a DMAIC (Define, Measure, Analyze, Improve, Control) approach...
Our car manufacturing plant is seeing a high percentage of material waste,...	Provide a Six Sigma based solution for reducing material waste in manufacturing.	Apply the 5S methodology (Sort, Set in order, Shine, Standardize, Sustain) to...
A telecom company is receiving an increasing number of complaints regarding...	Develop a solution using Six Sigma for improving customer satisfaction in a...	Use the Voice of the Customer (VOC) technique to gather and analyze customer...
A plastic molding factory has high energy bills. affecting overall profitability.	Advise a Six Sigma approach to reduce energv consumption in a manufacturing...	Conduct a Value Stream Mapping to identify non-value-added processes consuming exces...

Fig. 1: The Six Sigma dataset.

2. 5M dataset

4.3.14 Generating Sample Data

- Synthetic data showcases diverse scenarios, covering industries such as manufacturing, healthcare, and logistics.
- Examples include challenges in supply chain optimization, defect reduction in manufacturing, and service quality improvement.

4.3.15 Integrating with Six Sigma Use Cases

- LLM-generated solutions are seamlessly integrated into Six Sigma use cases, demonstrating adaptability and effectiveness.
- Use cases cover scenarios from different industries, emphasizing the applicability of fine-tuned models.


instruction string · lengths  67~75 7.8%	context string · lengths  208~227 2.6%	response dict
Use the 5Ms framework to evaluate the challenges in an agricultural farm.	Farm C is experiencing low crop yields. Farmers are undertrained, farming methods are traditional, tractors and tools are outdated, there's a scarcity of high-quality seeds, and soil quality is not regularly tested.	{ "Man": ["Undertrained farmers", "Lack of skilled agronomists"], "Method": ["Traditional farming methods", "Inefficient irrigation techniques"], "Machine": ["Outdated tractors and tools", "Lack of modern farming equipment"], "Material": ["Scarcity of high-quality seeds", "Inadequate fertilizer supply"], "Measurement": ["Irregular soil quality testing", "Inadequate crop yield tracking"] }
Evaluate the operation of a high-end fashion brand using the 5Ms...	Fashion Brand L is struggling with its latest collection. Designers...	{ "Man": ["Creative blocks among designers", "High turnover in..."] }

Fig. 2: The 5M dataset.

4.3.16 Quality Assurance

- Rigorous quality assurance processes ensure the accuracy and relevance of LLM-generated solutions.
- Validation against real-world scenarios and Six Sigma principles validates the effectiveness of the fine-tuned models.

4.3.17 Quality Metrics and Evaluation

To gauge the effectiveness of the fine-tuned models, we employ a set of quality metrics and evaluation techniques.

4.3.18 Metric Selection

- Metrics such as accuracy, precision, recall, and F1 score quantify LLM performance.
- Six Sigma-specific metrics, including defect rates and process efficiency, provide a comprehensive evaluation.

4.3.19 Validation against Real-world Data

- Fine-tuned models are validated against real-world Six Sigma scenarios, ensuring practical applicability and effectiveness.

4.3.20 Next Steps and Recommendations

With the LLMs fine-tuned using synthetic data, the next steps involve deploying them in industrial environments.

4.3.21 Deployment

- Deploy fine-tuned LLMs in real-world industrial settings, including manufacturing plants, supply chain management, and service industries.

4.3.22 Ongoing Monitoring

- Continuously monitor models to identify drift or degradation in performance over time.
- Regular updates and re-training based on ongoing monitoring results to maintain adaptability.

4.3.23 Areas for Improvement

- Periodically revisit the synthetic data generation process to incorporate new challenges.
- Ensure models remain adaptable to evolving industrial scenarios through continuous improvement.

4.4 Host the dataset in HuggingFace

We have seen in the previous tutorial *FineTuning_Ludwig* that in order to fine-tune a model on a new task, we need to have a dataset in the right format. In this tutorial, we will see how to push your dataset to HuggingFace so that it can be used later to fine-tune a model.

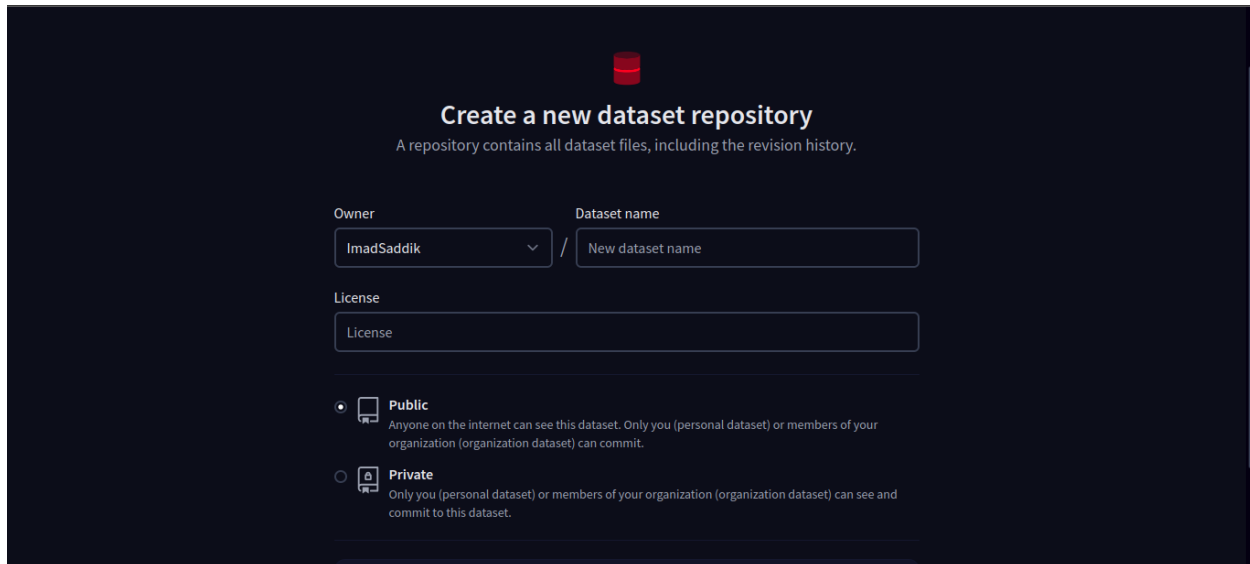
Note: You need to create an account on HuggingFace to be able to push your dataset. You can create an account [here](#). There is a video tutorial available for this section [watch it](#).

4.4.1 Pushing the dataset

Once your dataset is in the right format (see the data format *tutorial*), you can upload it to HuggingFace. To do so, you need to follow the following steps:

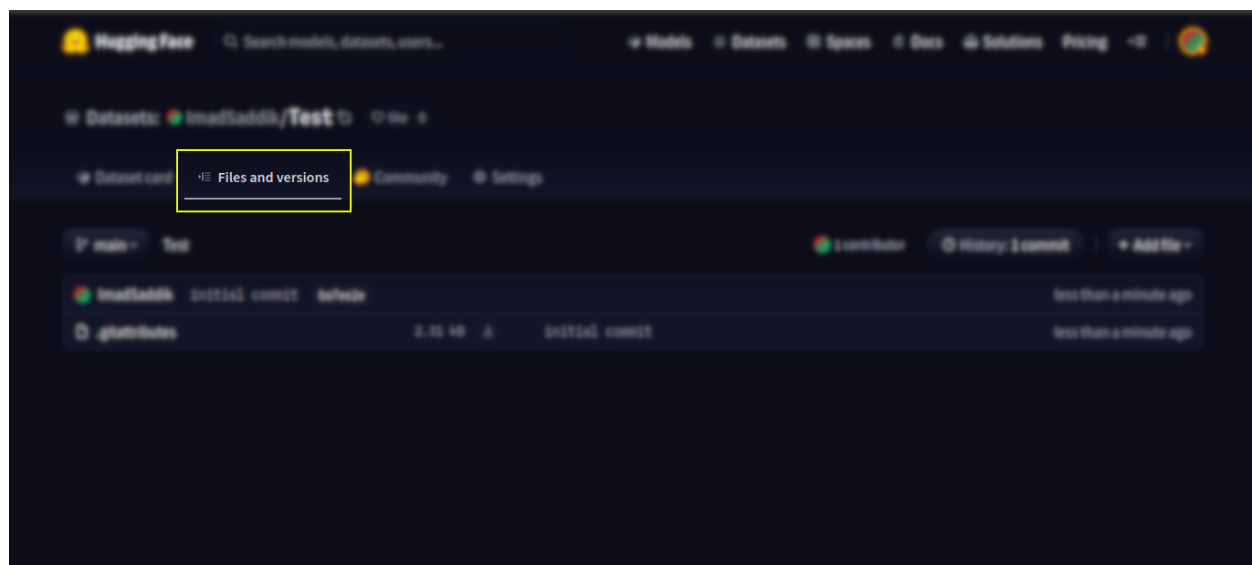
1. Connect to your HuggingFace account.
2. Visit the following [link](#) and give a name to your dataset, choose a proper license and choose if you want to make your dataset public or private.
3. Click on the *Files and versions* tab.
4. Click on *Add file*, then *Upload files* and upload your dataset.
5. Drag and drop your dataset.
6. Hit *Commit changes to main*.

Congratulations! You have successfully pushed your dataset to HuggingFace. You can now use it to fine-tune a model on a new task.



The screenshot shows the 'Create a new dataset repository' page on HuggingFace. At the top, there is a red database icon and the title 'Create a new dataset repository' with a subtitle 'A repository contains all dataset files, including the revision history.' Below this, there are three input fields: 'Owner' (a dropdown menu showing 'ImadSaddik'), 'Dataset name' (a text box with 'New dataset name'), and 'License' (a text box with 'License'). At the bottom, there are two radio button options: 'Public' (selected) and 'Private'. The 'Public' option has a description: 'Anyone on the internet can see this dataset. Only you (personal dataset) or members of your organization (organization dataset) can commit.' The 'Private' option has a description: 'Only you (personal dataset) or members of your organization (organization dataset) can see and commit to this dataset.'

Fig. 3: Create a new dataset page.

Fig. 4: The *Files and versions* tab.

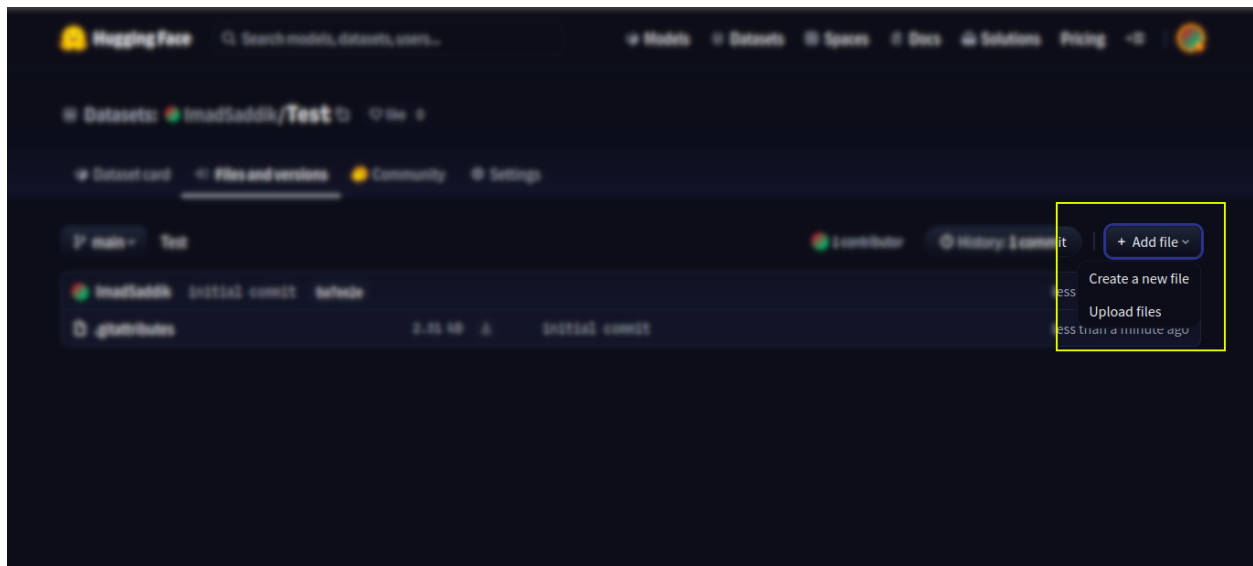
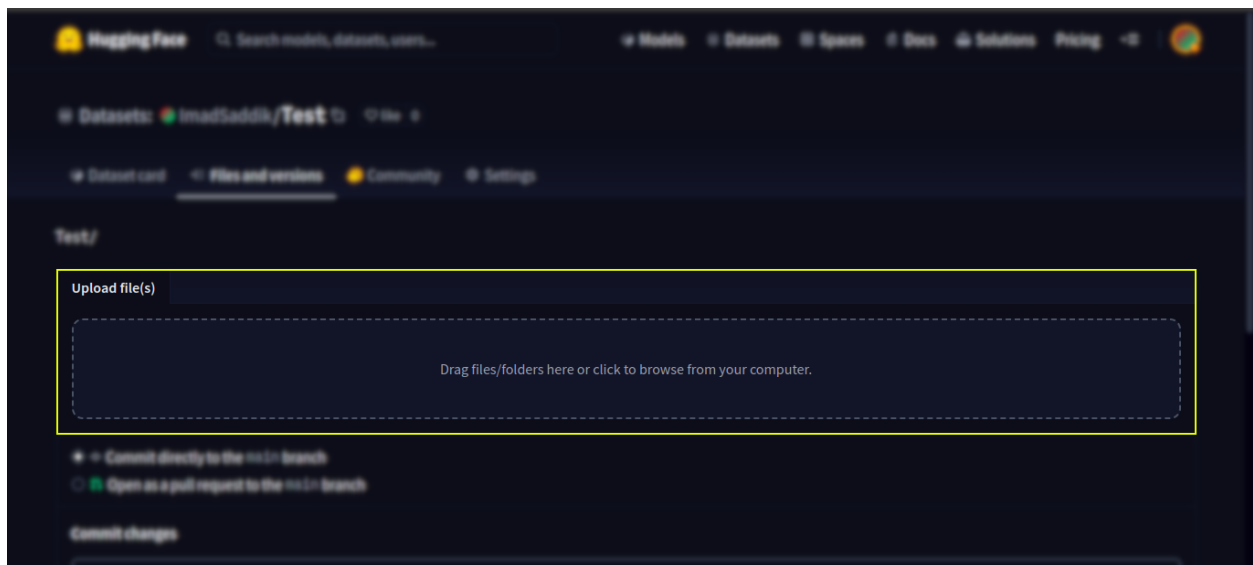
Fig. 5: The *Files and versions* tab.

Fig. 6: The drag and drop area.

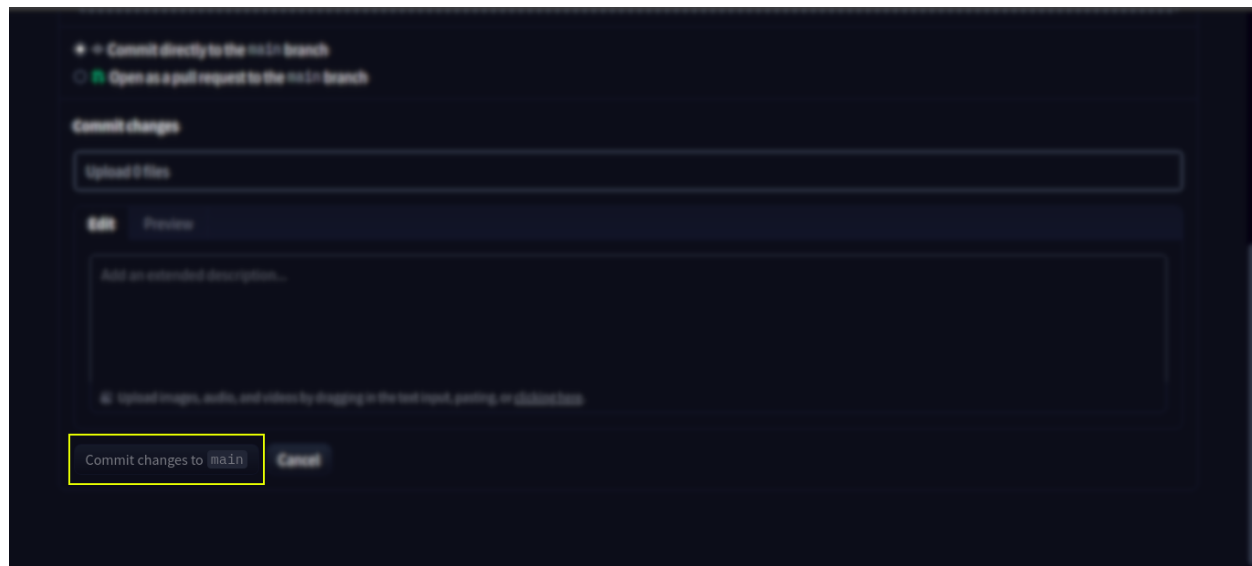


Fig. 7: The *Commit changes to main* button.

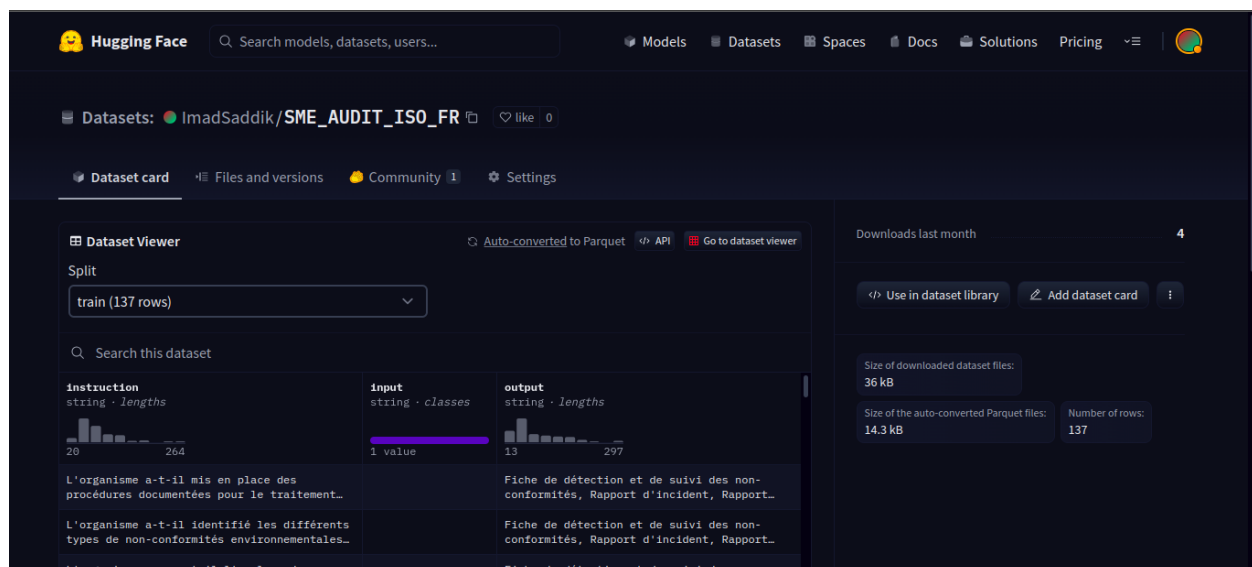


Fig. 8: The preview of the data.

CHAINLIT

5.1 Chainlit - hyperparameters tuning

To control the behavior of the llm, we can change some hyperparameters like the `max_new_tokens`, the `temperature` of the sampling, or the `top_k` and `top_p` values.

Note: For more information on the hyperparameters that you can play with, please refer to the CTransformers [documentation](#).

There is a video tutorial available for this section [watch it](#).

5.1.1 The implementation

Let's first import the necessary libraries:

```
import chainlit as cl
from chainlit.input_widget import Slider, Switch
```

To change the hyperparameters, we will use the `Slider` and `Switch` widgets. The `Slider` widget allows us to change the value of a hyperparameter by moving a slider. The `Switch` widget allows us to change the value of a hyperparameter by switching between two values.

```
settings = await cl.ChatSettings(
    [
        Slider(
            id="Temperature",
            label="Temperature",
            initial=1,
            min=0,
            max=2,
            step=0.1,
        ),
        Slider(
            id="Repetition Penalty",
            label="Repetition Penalty",
            initial=0.3,
            min=0,
            max=2,
            step=0.1,
```

(continues on next page)

(continued from previous page)

```

    ),
    Slider(
        id="Top P",
        label="Top P",
        initial=0.7,
        min=0,
        max=1,
        step=0.1,
    ),
    Slider(
        id="Top K",
        label="Top K",
        initial=42,
        min=0,
        max=100,
        step=1,
    ),
    Slider(
        id="Max New Tokens",
        label="Max New Tokens",
        initial=256,
        min=0,
        max=1024,
        step=1,
    ),
    Switch(id="Streaming", label="Stream Tokens", initial=True),
]
).send()

```

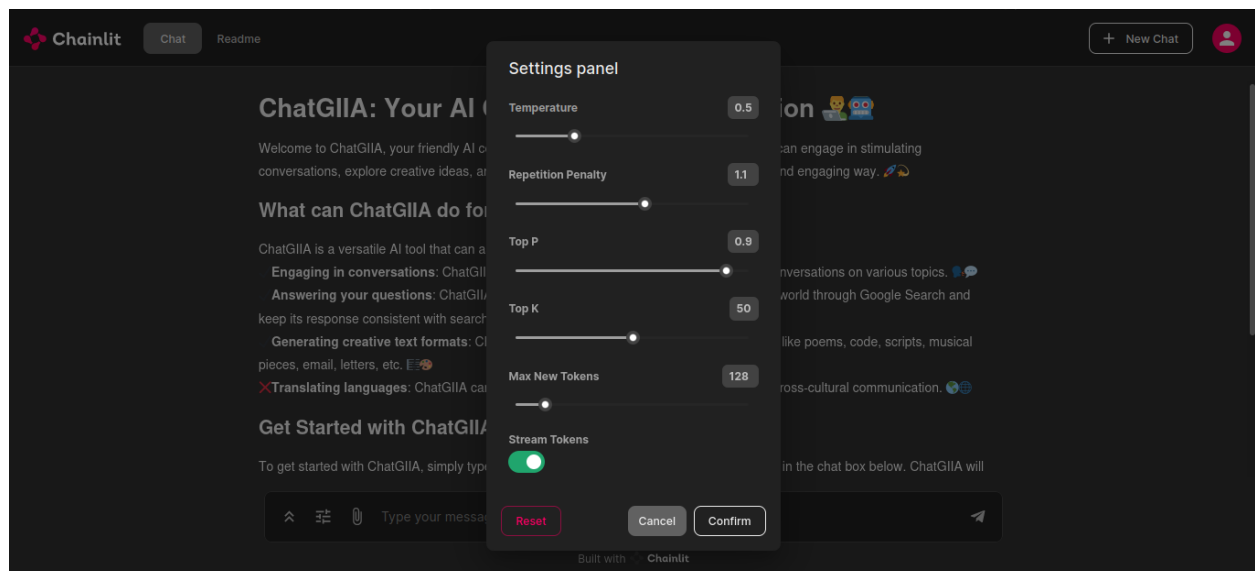


Fig. 1: The settings panel.

Now as demonstrated in the figure above the user can change the hyperparameters using the sliders. The **Streaming** switch allows us to stream the tokens as they are generated. If the switch is turned off, the tokens will be generated all at once.

5.2 Chainlit - model selection

In this part, we will see how we can add a dropdown menu to give the user the ability to choose the model he wants to use.

Note: Make sure that the chainlit package is installed. If not, run `pip install chainlit`.

There is a video tutorial available for this section [watch it](#).

5.2.1 The implementation

Let's first import the necessary libraries:

```
import chainlit as cl
from chainlit.input_widget import Select
```

To create a dropdown menu, we will be using the Select widget. This widget takes a list of options as input and returns the selected option. Let's see how to do this.

```
settings = await cl.ChatSettings(
    [
        Select(
            id="model",
            label="Choos an open source llm",
            values=["mistral-7b", "llama2-7b", "zephyr-7b-beta"], # the list of models
            ↪ goes here
            initial_index=0,
        ),
    ]
).send()
```

To verify if the selected model is being used we will run the following program that will send a message with the selected model to the user.

```
import chainlit as cl
from chainlit.input_widget import Select

@cl.on_chat_start
async def start():
    settings = await cl.ChatSettings(
        [
            Select(
                id="model",
                label="Choos an open source llm",
                values=["mistral-7b", "llama2-7b", "zephyr-7b-beta"],
                initial_index=0,
            ),
        ]
    ).send()
```

(continues on next page)

(continued from previous page)

```

@cl.on_settings_update
async def setup_agent(settings):
    model = settings["model"]
    await main(cl.Message(content=f"Running model: {model}"))

@cl.on_message
async def main(message: cl.Message):
    await message.send()

```

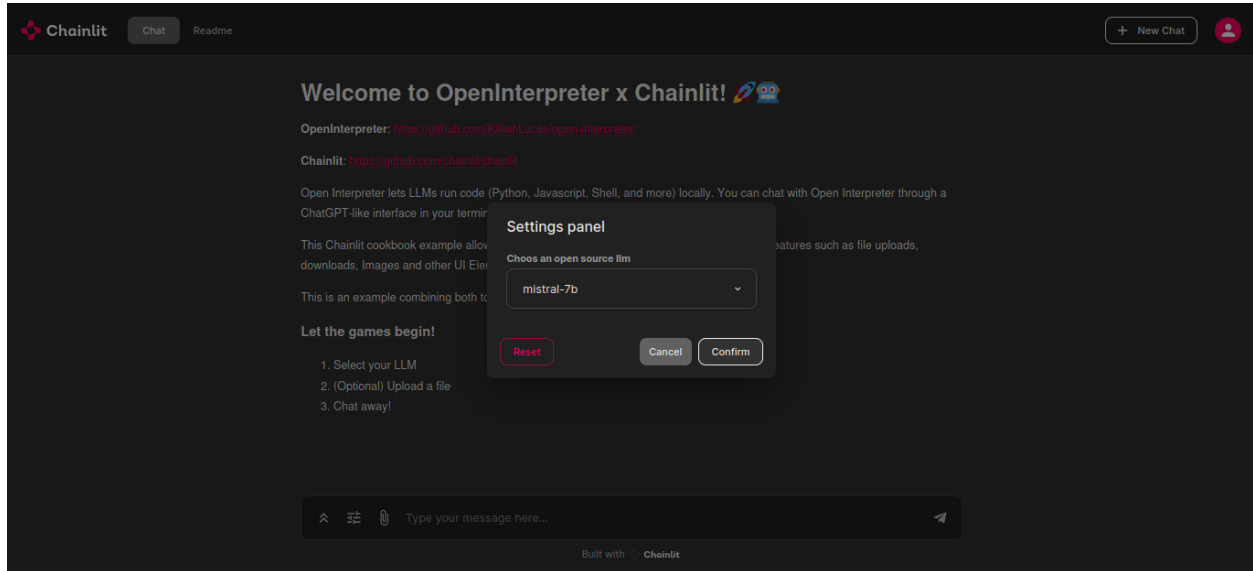


Fig. 2: The dropdown menu.

After clicking on the confirm button. The name of the selected model will be printed to the user.

5.3 Chainlit: an easy way to interact with LLMs

In this section, we'll demonstrate the process of engaging with an open-source language model of your choice available on the Hugging Face model hub. To facilitate this interaction, we'll leverage the Chainlit library—an open-source asynchronous Python framework designed to expedite the creation of applications akin to ChatGPT. This library enables seamless interaction with models through an automatically generated user interface. For more detailed insights into Chainlit and its functionalities, further information is available [here](#).

Note: There is a video tutorial available for this section [watch it](#).

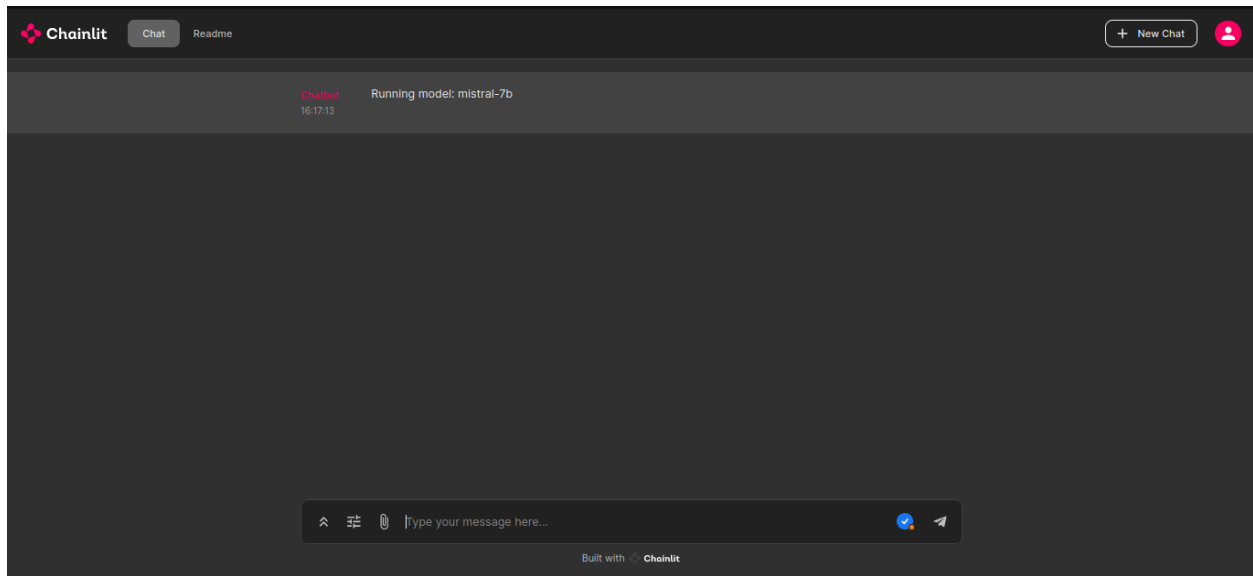


Fig. 3: The name of the selected model is printed in the UI.

5.3.1 Prerequisites

To get started, we'll need to install the Chainlit library and other dependencies. To do so, we'll create a new virtual environment using the following command:

```
# using python
python3 -m venv chainlit_env

# using anaconda
conda create -n chainlit_env python=3.11
```

Next, we'll activate the virtual environment and install the necessary dependencies

```
# activate the virtual environment
# using python
source chainlit_env/bin/activate

# using anaconda
conda activate chainlit_env

# install the dependencies
pip install chainlit
pip install ctransformers
pip install langchain
pip install torch
```

5.3.2 Downloading the model

The models that we need to download from Hugging Face Hub should be in the **GGUF** format. In this [link](#) you can find the **Mistral** model in this format, download the model that has this name **mistral-7b-instruct-v0.1.Q4_K_S.gguf** because we will need it for the rest of this tutorial. If you want to use another llm just search for it in one of TheBloke's repositories.

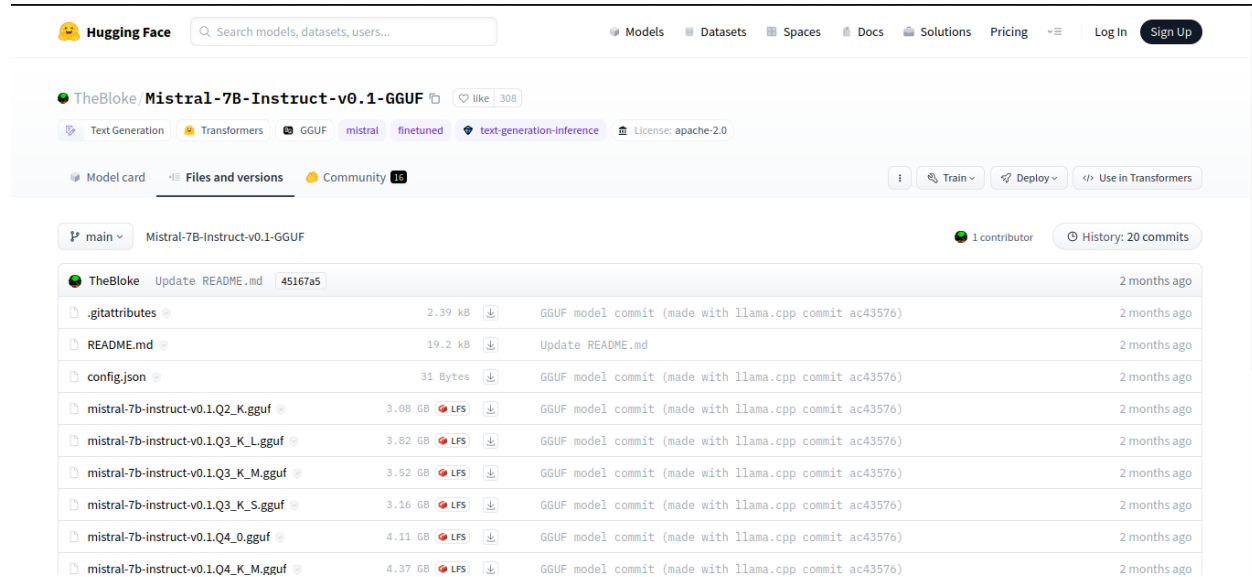


Fig. 4: The Mistral-7B-Instruct-v0.1-GGUF model repository.

5.3.3 The interface

Now let's create a new file called **app.py** and import the necessary libraries:

```
import os
import chainlit as cl
from chainlit.input_widget import Slider, Switch
from langchain.chains import LLMChain
from langchain.llms import CTransformers
from langchain.prompts import PromptTemplate
```

Now we'll create a variable to store the path to the model we downloaded earlier:

```
local_llm = "./mistral-7b-instruct-v0.1.Q4_K_S.gguf" # download the model from this link_
↳ https://huggingface.co/TheBloke/Mistral-7B-Instruct-v0.1-GGUF/tree/main
```

Next we'll create a configuration dictionary to store the parameters that we'll use to initialize our model:

```
config = {
    'max_new_tokens': 128,
    'repetition_penalty': 1.1,
    'temperature': 0.5,
    'top_p': 0.9,
    'top_k': 50,
    'stream': True,
```

(continues on next page)

(continued from previous page)

```
'threads': int(os.cpu_count() / 2),
}
```

The values in this dictionary are the default values for the parameters that we'll use to initialize our model. For more information on these parameters, please refer to the CTransformers [documentation](#).

In the interface we'll use the **Slider** and **Switch** widgets to allow the user to adjust these parameters. To do so, we'll use Chainlit's ChatSettings class as follows:

```
settings = await cl.ChatSettings(
    [
        Slider(
            id="Temperature",
            label="Temperature",
            initial=config['temperature'],
            min=0,
            max=2,
            step=0.1,
        ),
        Slider(
            id="Repetition Penalty",
            label="Repetition Penalty",
            initial=config['repetition_penalty'],
            min=0,
            max=2,
            step=0.1,
        ),
        Slider(
            id="Top P",
            label="Top P",
            initial=config['top_p'],
            min=0,
            max=1,
            step=0.1,
        ),
        Slider(
            id="Top K",
            label="Top K",
            initial=config['top_k'],
            min=0,
            max=100,
            step=1,
        ),
        Slider(
            id="Max New Tokens",
            label="Max New Tokens",
            initial=config['max_new_tokens'],
            min=0,
            max=1024,
            step=1,
        ),
        Switch(id="Streaming", label="Stream Tokens", initial=True),
    ]
)
```

(continues on next page)

(continued from previous page)

).send()

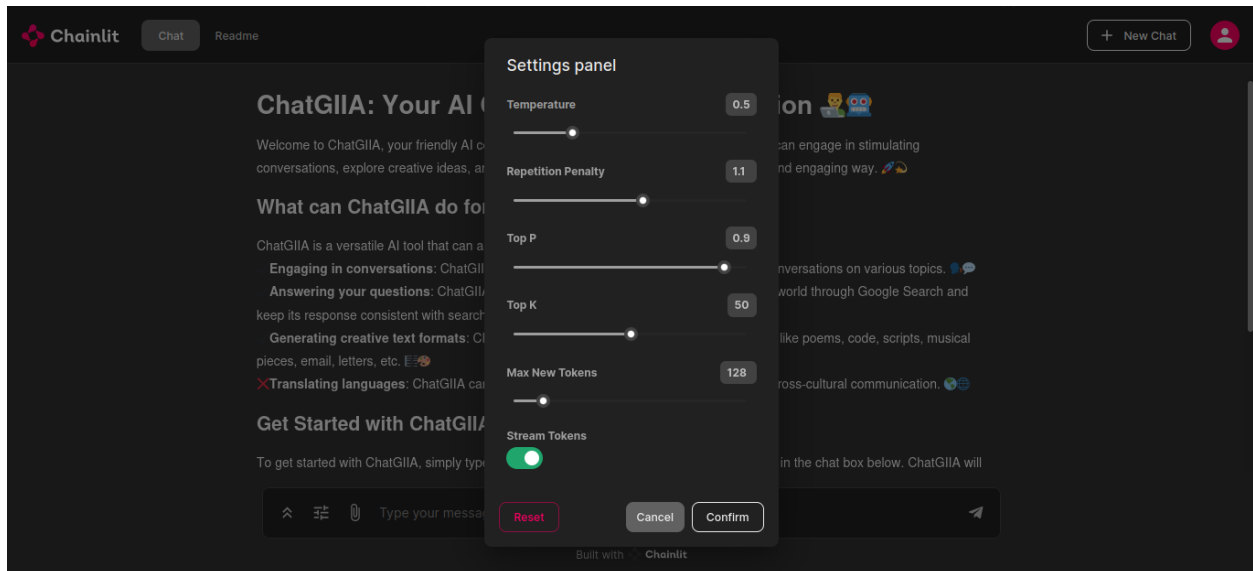


Fig. 5: The settings panel.

Now whenever the user changes one of these parameters, we need to setup the model to use the new values. To do so, we'll create a function called **setup_agent** that will update the values in the **config** dictionary as well as applying it to the model:

```
def setup_agent(settings):
    # update the config dictionary with the new settings
    config['temperature'] = settings['Temperature']
    config['repetition_penalty'] = settings['Repetition Penalty']
    config['top_p'] = settings['Top P']
    config['top_k'] = settings['Top K']
    config['max_new_tokens'] = settings['Max New Tokens']
    config['stream'] = settings['Streaming']

    # update the model with the new settings
    llm_init = CTransformers(
        model=local_llm,
        model_type="mistral",
        lib="avx2", # 'avx2' or 'avx512'
        **config
    )

    # creating the prompt template
    template = """
    Question: {question}
    Answer:
    """

    prompt = PromptTemplate(template=template, input_variables=['question'])
```

(continues on next page)

(continued from previous page)

```
# creating the llm chain
llm_chain = LLMChain(prompt=prompt, llm=llm_init, verbose=False)

# saving the llm chain in the session
cl.user_session.set('llm_chain', llm_chain)
```

In the `setup_agent` function, we have created a **PromptTemplate** object that will be used to generate the prompt that we'll feed to the model. This object takes a template string and a list of input variables. The template string is a string that contains the text that we want to feed to the model. The input variables are the variables that we want to replace in the template string. In our case, we want to replace the `{question}` variable with the question that the user will ask. For more information on the **PromptTemplate** class, please refer to the [LangChain documentation](#).

After that, we have created an **LLMChain** object that will be used to interact with the model. This object takes a **PromptTemplate** object and an **LLM** object. For more information on the **LLMChain** class, please refer to the [LangChain documentation](#).

Finally, we have saved the **LLMChain** object in the user session so that we can access it later. For more information on the user session, please refer to the [Chainlit documentation](#).

The `setup_agent` will be called whenever the user changes one of the parameters in the interface. To do so, we'll use the `cl.on_settings_update` decorator as follows:

```
@cl.on_chat_start
async def start():
    settings = await cl.ChatSettings(...)

    # calling the setup_agent function
    await setup_agent(settings)

@cl.on_settings_update
async def setup_agent(settings):
    # the content of the setup_agent function
```

Now, we are ready to start the chat. To do so, we'll use the `cl.on_message` decorator as follows:

```
@cl.on_message
async def main(message):
    # getting the llm chain from the session
    llm_chain = cl.user_session.get('llm_chain')

    # generating the response
    result = await llm_chain.acall(message.content, callbacks=[cl.
    AsyncLangchainCallbackHandler()])

    # sending the response
    await cl.Message(content=result["text"]).send()
```

In the `main` function, we have retrieved the **LLMChain** object from the user session and used it to generate the response. The `acall` method takes the user input and a list of callbacks. The **AsyncLangchainCallbackHandler** is a callback that is used to handle the asynchronous calls to the model. For more information on the **LangChain Callback Handler**, please refer to the [Chainlit documentation](#).

Finally, we have sent the response to the user using the `cl.Message` class. For more information on the **Message** class, please refer to the [Chainlit documentation](#).

5.3.4 Running the interface

To run the interface, we'll use the following command:

```
chainlit run app.py -w
```

After running the command, you should see something like this:

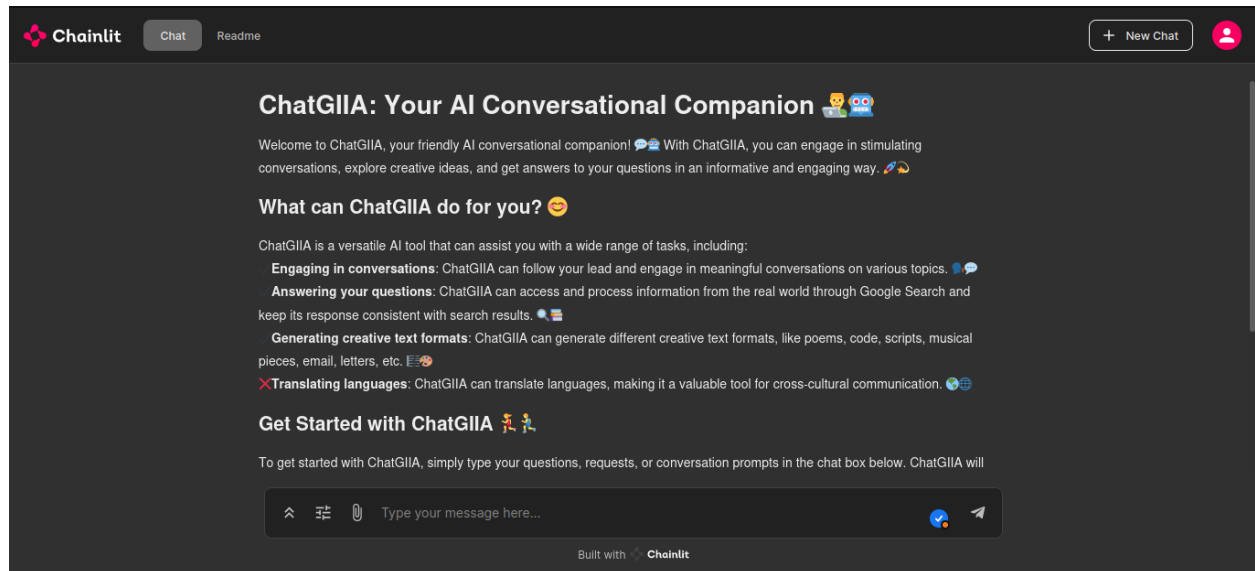


Fig. 6: The interface once loaded.

To change the content that appears in the interface once running the command, you can edit the **chainlit.md** file

RAG APPLICATION

6.1 ConversationChain Class

The `ConversationChain` class is part of a project that implements the Retrieval-Augmented Generation (RAG) method to interact and retrieve information from documents, such as PDFs and text files. This class handles the management of conversation data, establishment of a retriever and execution of embedding models to compute text similarities within conversations.

6.1.1 Attributes

- **current_conversation_id**
[str] A unique identifier for the current conversation, typically a timestamp.
- **current_conversation_file**
[str] The file path to the active conversation text file.
- **directory**
[str] The directory path where conversation documents are stored.
- **docs**
[list] A list that holds the loaded documents after processing.
- **model**
[object] Placeholder for an instance of a language model used for downstream tasks.
- **chain**
[object] Placeholder for the RAG chain that connects retrieval to downstream tasks.
- **retriever**
[object] An instance of a retriever that manages document indexing and retrieval.
- **bge_embeddings**
[object] An embedding model from Hugging Face used for generating embeddings.
- **vectorstore**
[object] A storage system for embeddings, used for efficient querying and retrieval.
- **store**
[object] A key-value document store to hold chunked documents.
- **done**
[bool] A flag indicating if the initialization process has been completed.
- **conv_names**
[list] A list of conversation document names that have been loaded.

- **embedding_size**
[int] The dimensionality of the text embedding vectors.
- **index**
[object] A FAISS index for efficient similarity search in large scale vector databases.
- **retriever2**
[object] Duplicate attribute, likely a mistake as it isn't assigned elsewhere in the class.
- **memory_vectorstore**
[object] A vector store used for short-term storage of conversation vectors.
- **store2**
[object] Duplicate attribute, likely a mistake as it isn't assigned elsewhere in the class.
- **memory**
[object] A retriever instance for managing the current conversation's memory.

6.1.2 Methods

- **__init__()**
Initializes the ConversationChain with default values and directory set to `"/conversations"`.
- **start_new_conversation()**
Creates a new unique conversation identified by the current timestamp and initializes a text file for storing conversation content.
- **add_to_conversation(conversation: str)**
Appends a string of conversation to the current conversation file.
- **init_embedding_model()**
Initializes the embedding model with pre-trained Hugging Face embeddings.
- **load_documents()**
Loads all documents from the specified directory and processes them into the docs list, printing the count of loaded conversations.
- **init_retriever()**
Initializes the retrievers with vector stores and document stores, then, if documents have been loaded, adds them to the retriever for indexing.
- **init_model()**
Placeholder method for initializing a language model for downstream tasks.
- **init_chain()**
Orchestrates the initialization of the RAG components including the embedding model, language model, document loading, and retriever, and sets the done flag to True once the initialization is complete.
- **add_new_document(filename: str)**
Adds a new conversation document to the retriever if it's not already present in the conv_names list.
- **add_to_memory()**
Adds the active conversation file to the memory retriever for short-term retrieval operations.

6.1.3 Usage Example

```
# Create a ConversationChain instance
conv_chain = ConversationChain()

# Start a new conversation
conv_chain.start_new_conversation()

# Add a dialogue to the conversation
conv_chain.add_to_conversation("Hello, how can I assist you today?")

# Initialize the RAG components
conv_chain.init_chain()

# Load and add documents to the retriever
conv_chain.load_documents()
```

Note: Classes such as `HuggingFaceBgeEmbeddings`, `LLM_model`, `ParentDocumentRetriever`, `InMemoryStore`, `TextLoader`, and `Chroma` should be properly installed and imported for the `ConversationChain` to function.

6.1.4 Important Considerations

- Ensure that the *directory* exists and is writable.
- For the RAG method to work, the backend must be properly set up with vector storage and retrieval functionality.
- Memory management can be a concern when loading many documents or managing long conversations. Performance optimization may be necessary.

If you require further information or assistance with using the `ConversationChain` class, consult the full documentation or contact the project maintainer.

6.2 DocumentChain Class

The `DocumentChain` class is designed to facilitate the implementation of the Retrieval-Augmented Generation (RAG) method for information retrieval from various documents such as PDFs, text files, etc. This class encapsulates methods and attributes necessary to process and retrieve information from a collection of documents using advanced NLP models and techniques.

6.2.1 Attributes

- **docs**
[list] A list that holds the loaded documents after processing.
- **model**
[object] Placeholder for the language model used for downstream tasks (e.g., question answering).
- **chain**
[object] Placeholder for the RAG chain which connects retrieval to downstream tasks.

- **retriever**
[object] An instance of a retriever that handles document indexing and retrieval.
- **bge_embeddings**
[object] An embedding model from Hugging Face used for generating embeddings for text.
- **vectorstore**
[object] A storage system for embeddings, which enables efficient querying and retrieval.
- **store**
[object] A key-value document store to hold chunked documents.
- **done**
[bool] A flag indicating if the initialization has been completed.
- **doc_names**
[list] A list of all document names that have been loaded.
- **directory**
[str] The directory path where document files are stored.

6.2.2 Methods

- **__init__()**
Initializes an empty DocumentChain with defaults and directory to `"./documents"`.
- **init_embedding_model()**
Initializes the embedding model with pre-trained Hugging Face embeddings.
- **load_documents()**
Loads all the documents from the specified directory and processes them into the docs attribute.
- **init_retriever()**
Initializes the `retriever` with a vector store and document store, and adds processed documents to the retriever.
- **init_model()**
Placeholder for initializing the downstream language model.
- **init_chain()**
Orchestrates the initialization of all components necessary for the RAG method, including the embedding model, downstream model, document loading, and retriever.
- **add_new_document(filename: str)**
Adds a new document to the retriever. The document is specified by the filename, and it is loaded, processed, and added to the store.

6.2.3 Usage Example

Below is an example demonstrating how to use the DocumentChain class:

```
# Initialize the DocumentChain
doc_chain = DocumentChain()

# Initialize all components
doc_chain.init_chain()
```

(continues on next page)

(continued from previous page)

```
# Check if the initialization is complete
if doc_chain.done:
    print("Document Chain is ready for use.")

# Add a new document to the retriever
new_filename = "new_document.txt"
doc_chain.add_new_document(new_filename)
```

Note: External dependencies such as *HuggingFaceBgeEmbeddings*, *TextLoader*, *PyPDFLoader*, document splitters, *Chroma*, *LLM_model*, *ParentDocumentRetriever* should be properly installed and imported for the *DocumentChain* class to function correctly.

6.2.4 Important Considerations

- Make sure that all required dependencies are satisfied.
- The directory set in `directory` must exist and contain the documents to be processed.
- Loading large sets of documents may be time-consuming and could require substantial memory, depending on document size and quantity.
- The actual retrieval and downstream use of the RAG chain are not implemented within the provided methods.

If you encounter issues or have any questions regarding the use of the `DocumentChain` class, please refer to the project developer or support documentation.

DEPLOY TO REPLICATE

7.1 Overview

This document details the code and its functionalities in the “deploy-to-replicate” Jupyter Notebook. The notebook is designed for setting up and deploying an environment, likely for Docker-based projects.

7.2 Setting Checkpoint Directory

```
checkpoint_dir = "."
```

Comment: Sets the checkpoint directory to the current directory. This is typically used to specify where to save or retrieve data during deployment.

7.3 Install Docker Script

```
%%writefile install_docker.sh
#!/bin/bash

# Function to check if a command exists
command_exists() {
    command -v "$@" > /dev/null 2>&1
}

# Check if Docker is already installed
if command_exists docker; then
    echo "Docker is already installed. Checking the version..."
    sudo docker --version
else
    echo "Docker is not installed. Proceeding with the installation..."

    echo 'Step 1: Update Software Repositories'
    sudo apt update

    echo 'Step 2: Install Dependencies'
    sudo apt install -y apt-transport-https ca-certificates curl software-properties-common

    echo 'Step 3: Add Docker's GPG Key'
```

(continues on next page)

(continued from previous page)

```

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

echo 'Step 4: Add Docker Repository'
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
→$(lsb_release -cs) stable"

echo 'Step 5: Update Package Database'
sudo apt update

echo 'Step 6: Install Docker CE'
sudo apt install -y docker-ce

echo 'Step 7: Start and Enable Docker'
sudo systemctl start docker
sudo systemctl enable docker

echo "Docker has been installed successfully."
sudo docker --version
fi

```

Comment: Creates and writes a bash script named *install_docker.sh* to install Docker. This script includes checks for existing installations and then proceeds with updating repositories, installing dependencies, adding Docker’s GPG key, and finally installing Docker CE. It ensures Docker is installed and running on the system.

7.4 Executing the Install Script

```
!chmod +x install_docker.sh && ./install_docker.sh
```

Comment: Makes the *install_docker.sh* script executable and then runs it. This step executes the script, which installs Docker based on the commands and checks provided within the script.

[Further cells would continue in a similar format, each with a “Code” and “Comment” section explaining the cell’s purpose and functionality.]

7.5 Install Replicate Cog

```

!sudo curl -o /usr/local/bin/cog -L https://github.com/replicate/cog/releases/latest/
→download/cog_`uname -s`_`uname -m`
!sudo chmod +x /usr/local/bin/cog

```

Comment: Downloads and installs Replicate’s Cog tool, a necessary component for building and deploying machine learning models. It sets the necessary permissions to make it executable.

7.6 Initialize Cog

```
!cd {checkpoint_dir}
!cog init
```

Comment: Changes the directory to the specified checkpoint directory and initializes a new Cog project in it. This sets up the structure needed for Cog to build and run models.

7.7 Define Cog Configuration

```
%%writefile cog.yaml
build:
  gpu: true
  cuda: "12.0.1"
  python_version: "3.10"
  python_requirements: requirements.txt
predict: "predict.py:Predictor"
```

Comment: Creates a *cog.yaml* file to define the configuration for Cog, including the use of GPU, CUDA version, Python version, and the prediction interface.

7.8 Define Requirements

```
%%writefile requirements.txt
bitsandbytes
git+https://github.com/huggingface/transformers.git
git+https://github.com/huggingface/peft.git
git+https://github.com/huggingface/accelerate.git
scipy
```

Comment: Specifies the Python requirements for the project in a *requirements.txt* file. This includes necessary libraries like *bitsandbytes* and specific versions of *transformers*, *peft*, and *accelerate* from Hugging Face.

7.9 Prediction Interface

```
%%writefile predict.py
# Prediction interface for Cog
# [Full script contents]
```

Comment: Creates a *predict.py* file that defines the prediction interface for Cog. This includes setting up the model, tokenizer, and the prediction function that will be used when the model is deployed.

7.10 Push to Replicate

```
sudo cog login && sudo cog push r8.im/<your-username>/<your-model-name>
```

Comment: Logs into the Replicate platform and pushes the configured model to your specified repository. This makes your model accessible for others to use through Replicate.

7.11 Conclusion

This document provided a detailed guide to each step involved in the “deploy-to-replicate” notebook, focusing on setting up and deploying an environment for Docker-based projects, including the setup of Replicate’s Cog tool for model deployment.