Cheat Sheet: Fundamentals of Building AI Agents using RAG and Lang Chain

Package/ Method	Description	Code Example
Generate text	This code snippet generates text sequences based on the input and doesn’t compute the gradient to generate output.	# Generate text output_ids = model.generate(inputs.input_ids, attention_mask=inputs.attention_mask, pad_token_id=tokenizer.eos_token_id, max_length=50, num_return_sequences=1) output_ids or with torch.no_grad(): outputs = model(**inputs) outputs
formatting_prompts_ func_no_response function	The prompt function generates formatted text prompts from a data set by using the instructions from the data set. It creates strings that include only the instruction and a placeholder for the response.	def formatting_prompts_func(mydataset): output_texts = [] for i in range(len(mydataset['instruction'])): text = (f"### Instruction:\n{mydataset['instruction'][i]}" f"\n\n### Response:\n{mydataset['output'][i]}</s>"

<table>
<tr>
<td></td>
<td></td>
<td>

```
        )
        output_texts.append(text)
    return output_texts

def formatting_prompts_func_no_response(mydataset):
    output_texts = []
    for i in range(len(mydataset['instruction'])):
        text = (
            f"### Instruction:\n{mydataset['instruction'][i]}"
            f"\n\n### Response:\n"
        )
        output_texts.append(text)
    return output_texts
```

</td>
</tr>
<tr>
<td>torch.no_grad()</td>
<td>This code snippet helps generate text sequences from the pipeline function. It ensures that the gradient computations are disabled and optimizes the performance and memory usage.</td>
<td>

```
with torch.no_grad():
    # Due to resource limitation, only apply the function on 3 records using "instructions_torch[:10]"
    pipeline_iterator= gen_pipeline(instructions_torch[:3],
                                    max_length=50, # this is set to 50 due to resource constraint, using a GPU, you can increase it to the length of your choice
                                    num_beams=5,
                                    early_stopping=True,)
generated_outputs_lora = []
for text in pipeline_iterator:
    generated_outputs_lora.append(text[0]["generated_text"])
```

</td>
</tr>
<tr>
<td>mixtral-8x7b-instruct-v01</td>
<td>Adjusts the parameters to push the limits of creativity and response length.</td>
<td>

```
model_id = 'mistralai/mixtral-8x7b-instruct-v01'

parameters = {
```

</td>
</tr>
</table>

watsonx.ai inference model object		`GenParams.MAX_NEW_TOKENS: 256,  # this controls the maximum number of tokens in the generated output` `GenParams.TEMPERATURE: 0.5, # this randomness or creativity of the model's responses` `}` `credentials = {` `"url": "https://us-south.ml.cloud.ibm.com"` `}` `project_id = "skills-network"` `model = ModelInference(` `model_id=model_id,` `params=parameters,` `credentials=credentials,` `project_id=project_id` `)`
String prompt templates	Used to format a single string and are generally used for simpler inputs.	`from langchain_core.prompts import PromptTemplate` `prompt = PromptTemplate.from_template("Tell me one {adjective} joke about {topic}")` `input_ = {"adjective": "funny", "topic": "cats"}  # create a dictionary to store the corresponding input to placeholders in prompt template` `prompt.invoke(input_)`
Chat prompt templates	Used to format a list of messages. These "templates" consist of a list of templates themselves.	`from langchain_core.prompts import ChatPromptTemplate` `prompt = ChatPromptTemplate.from_messages([` `("system", "You are a helpful assistant"),`

		`("user", "Tell me a joke about {topic}")` `])` `input_ = {"topic": "cats"}` `prompt.invoke(input_)`
Messages place holder	This prompt template is responsible for adding a list of messages in a particular place. But if you want the user to pass in a list of messages that you would slot into a particular spot, the given code snippet is helpful.	`from langchain_core.prompts import MessagesPlaceholder` `from langchain_core.messages import HumanMessage` `prompt = ChatPromptTemplate.from_messages([` `    ("system", "You are a helpful assistant"),` `    MessagesPlaceholder("msgs")` `])` `input_ = {"msgs": [HumanMessage(content="What is the day after Tuesday?")]}` `prompt.invoke(input_)`
Example selectors	If you have many examples, you may need to select which ones to include in the prompt. The Example Selector is the class responsible for doing so.	`from langchain_core.example_selectors import LengthBasedExampleSelector` `from langchain_core.prompts import FewShotPromptTemplate, PromptTemplate` `# Examples of a pretend task of creating antonyms.` `examples = [` `    {"input": "happy", "output": "sad"},` `    {"input": "tall", "output": "short"},` `    {"input": "energetic", "output": "lethargic"},` `    {"input": "sunny", "output": "gloomy"},` `    {"input": "windy", "output": "calm"},` `]`

		`` example_prompt = PromptTemplate(input_variables=["input", "output"], template="Input: {input}\nOutput: {output}",) example_selector = LengthBasedExampleSelector(examples=examples, example_prompt=example_prompt, max_length=25, # The maximum length that the formatted examples should be.) dynamic_prompt = FewShotPromptTemplate(example_selector=example_selector, example_prompt=example_prompt, prefix="Give the antonym of every input", suffix="Input: {adjective}\nOutput:", input_variables=["adjective"],) ```
JSON parser	This output parser allows users to specify an arbitrary JSON schema and query LLMs for outputs that conform to that schema.	``` from langchain_core.output_parsers import JsonOutputParser from langchain_core.pydantic_v1 import BaseModel, Field # Define your desired data structure. class Joke(BaseModel): setup: str = Field(description="question to set up a joke") punchline: str = Field(description="answer to resolve the joke") # And a query intented to prompt a language model to populate the data structure. ```

		`joke_query = "Tell me a joke."` `# Set up a parser + inject instructions into the prompt template.` `output_parser = JsonOutputParser(pydantic_object=Joke)` `format_instructions = output_parser.get_format_instructions()` `prompt = PromptTemplate(` `    template="Answer the user query.\n{format_instructions}\n{query}\n",` `    input_variables=["query"],` `    partial_variables={"format_instructions": format_instructions},` `)` `chain = prompt | mixtral_llm | output_parser` `chain.invoke({"query": joke_query})`
Comma separated list parser	This output parser can be used when you want to return a list of comma-separated items.	`from langchain.output_parsers import CommaSeparatedListOutputParser` `output_parser = CommaSeparatedListOutputParser()` `format_instructions = output_parser.get_format_instructions()` `prompt = PromptTemplate(` `    template="Answer the user query. {format_instructions}\nList five {subject}.",` `    input_variables=["subject"],`

<table>
<tr><td></td><td></td><td><pre><code>   partial_variables={"format_instructions": format_instructions},
)

chain = prompt | mixtral_llm | output_parser</code></pre></td></tr>
<tr><td>Document object</td><td>Contains information about some data in LangChain. It has two attributes:

page_content: str: This attribute holds the content of the document.
metadata: dict: This attribute contains arbitrary metadata associated with the document. It can be used to track various details such as the document id, file name, and so on.</td><td><pre><code>from langchain_core.documents import Document

Document(page_content="""Python is an interpreted high-level general-purpose programming language.
                        Python's design philosophy emphasizes code readability with its notable use of significant indentation.""",
         metadata={
             'my_document_id' : 234234,
             'my_document_source' : "About Python",
             'my_document_create_time' : 1680013019
         })</code></pre></td></tr>
<tr><td>text_splitter</td><td>At a high level, text splitters work as follows:
Split the text into small, semantically meaningful chunks (often sentences).
Start combining these small chunks into a larger chunk until you reach a certain size (as measured by some function).
Once you reach that size, make that chunk its own piece of text and start creating a new chunk with</td><td><pre><code>text_splitter = CharacterTextSplitter(chunk_size=200, chunk_overlap=20, separator="\n") # define chunk_size which is length of characters, and also separator.
chunks = text_splitter.split_documents(document)
print(len(chunks))</code></pre></td></tr>
</table>

<table>
<tr><td></td><td>some overlap (to keep context between chunks).</td><td></td></tr>
<tr><td>Embedding models</td><td>Embedding models are specifically designed to interface with text embeddings.

Embeddings generate a vector representation for a given piece of text. This is advantageous as it allows you to conceptualize text within a vector space. Consequently, you can perform operations such as semantic search, where you identify pieces of text that are most similar within the vector space.</td><td>

```
from ibm_watsonx_ai.metanames import
EmbedTextParamsMetaNames

embed_params = {
    EmbedTextParamsMetaNames.TRUNCATE_INPUT_TOKENS: 3,
    EmbedTextParamsMetaNames.RETURN_OPTIONS:
{"input_text": True},
}

from langchain_ibm import WatsonxEmbeddings

watsonx_embedding = WatsonxEmbeddings(
    model_id="ibm/slate-125m-english-rtrvr",
    url="https://us-south.ml.cloud.ibm.com",
    project_id="skills-network",
    params=embed_params,
)
```

</td></tr>
<tr><td>Vector store-backed retriever</td><td>A retriever that uses a vector store to retrieve documents. It is a lightweight wrapper around the vector store class to make it conform to the retriever interface. It uses the search methods implemented by a vector store, like similarity search and MMR (maximum marginal relevance), to query the texts in the vector store.</td><td>

```
retriever = docsearch.as_retriever()

docs = retriever.invoke("Langchain")A vector store
retriever is a retriever that uses a vector store to
retrieve documents. It is a lightweight wrapper around
the vector store class to make it conform to the
retriever interface. It uses the search methods
implemented by a vector store, like similarity search
and MMR (Maximum marginal relevance), to query the texts
in the vector store.
```

</td></tr>
</table>

<table>
<tr><td></td><td>Since we've constructed a vector store docsearch, it's very easy to construct a retriever.</td><td><pre><code>Since we've constructed a vector store docsearch, it's
very easy to construct a retriever.</code></pre></td></tr>
<tr><td>ChatMessageHistory class</td><td>One of the core utility classes underpinning most (if not all) memory modules is the ChatMessageHistory class. This super lightweight wrapper provides convenient methods for saving HumanMessages, AIMessages, and then fetching them all.</td><td><pre><code>from langchain.memory import ChatMessageHistory
chat = mixtral_llm

history = ChatMessageHistory()

history.add_ai_message("hi!")

history.add_user_message("what is the capital of
France?")</code></pre></td></tr>
<tr><td>langchain.chains</td><td>This code snippet uses a LangChain, library for building language model applications, creating a chain to generate popular dish recommendations based on the specified locations. It also configures model inference settings for further processing.</td><td><pre><code>from langchain.chains import LLMChain
template = """Your job is to come up with a classic dish
from the area that the users suggests.
                {location}

                YOUR RESPONSE:
"""
prompt_template = PromptTemplate(template=template,
input_variables=['location'])

chain 1
location_chain = LLMChain(llm=mixtral_llm,
prompt=prompt_template, output_key='meal')

location_chain.invoke(input={'location':'China'})</code></pre></td></tr>
<tr><td>Simple sequential chain</td><td>Sequential chains allow the output of one LLM to be used as the input for another. This approach is</td><td><pre><code>from langchain.chains import SequentialChain

template = """Given a meal {meal}, give a short and
simple recipe on how to make that dish at home.</code></pre></td></tr>
</table>

	beneficial for dividing tasks and maintaining the focus of your LLM.	<pre> YOUR RESPONSE: """ prompt_template = PromptTemplate(template=template, input_variables=['meal']) # chain 2 dish_chain = LLMChain(llm=mixtral_llm, prompt=prompt_template, output_key='recipe') template = """Given the recipe {recipe}, estimate how much time I need to cook it. YOUR RESPONSE: """ prompt_template = PromptTemplate(template=template, input_variables=['recipe']) # chain 3 recipe_chain = LLMChain(llm=mixtral_llm, prompt=prompt_template, output_key='time') # overall chain overall_chain = SequentialChain(chains=[location_chain, dish_chain, recipe_chain], input_variables=['location'], output_variables=['meal', 'recipe', 'time'], verbose= True)</pre>

load_summarize_chain	This code snippet uses LangChain library for loading and using a summarization chain with a specific language model and chain type. This chain type will be applied to web data to print a resulting summary.	`from langchain.chains.summarize import load_summarize_chain` `chain = load_summarize_chain(llm=mixtral_llm, chain_type="stuff", verbose=False)` `response = chain.invoke(web_data)` `print(response['output_text'])n`
TextClassifier	Represents a simple text classifier that uses an embedding layer, a hidden linear layer with a ReLU avtivation, and an output linear layer. The constructor takes the following arguments: num_class: The number of classes to classify. freeze: Whether to freeze the embedding layer.	`from torch import nn` `class TextClassifier(nn.Module):` `    def __init__(self, num_classes,freeze=False):` `        super(TextClassifier, self).__init__()` `        self.embedding = nn.Embedding.from_pretrained(glove_embedding.vectors.to(device),freeze=freeze)` `        # An example of adding additional layers: A linear layer and a ReLU activation` `        self.fc1 = nn.Linear(in_features=100, out_features=128)` `        self.relu = nn.ReLU()` `        # The output layer that gives the final probabilities for the classes` `        self.fc2 = nn.Linear(in_features=128, out_features=num_classes)` `    def forward(self, x):` `        # Pass the input through the embedding layer` `        x = self.embedding(x)` `        # Here you can use a simple mean pooling`

<table>
<tr>
<td></td>
<td></td>
<td><pre><code>   x = torch.mean(x, dim=1)
        # Pass the pooled embeddings through the
additional layers
        x = self.fc1(x)
        x = self.relu(x)
        return self.fc2(x)</code></pre></td>
</tr>
<tr>
<td>Train the model</td>
<td>This code snippet outlines the function to train a machine learning model using PyTorch. This function trains the model over a specified number of epochs, tracks them, and evaluates the performance on the data set.</td>
<td><pre><code>def train_model(model, optimizer, criterion,
train_dataloader, valid_dataloader, epochs=100,
model_name="my_modeldrop"):
    cum_loss_list = []
    acc_epoch = []
    best_acc = 0
    file_name = model_name

    for epoch in tqdm(range(1, epochs + 1)):
        model.train()
        cum_loss = 0
        for _, (label, text) in
enumerate(train_dataloader):
            optimizer.zero_grad()
            predicted_label = model(text)
            loss = criterion(predicted_label, label)
            loss.backward()

torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)
            optimizer.step()
            cum_loss += loss.item()
        #print("Loss:", cum_loss)
        cum_loss_list.append(cum_loss)
        acc_val = evaluate(valid_dataloader, model,
device)</code></pre></td>
</tr>
</table>

		`        acc_epoch.append(acc_val)` `        if acc_val > best_acc:` `            best_acc = acc_val` `            print(f"New best accuracy: {acc_val:.4f}")` `            #torch.save(model.state_dict(),` `f"{model_name}.pth")` `    #save_list_to_file(cum_loss_list,` `f"{model_name}_loss.pkl")` `    #save_list_to_file(acc_epoch,` `f"{model_name}_acc.pkl")`
llm_model	This code snippet defines function ‘llm_model’ for generating text using the language model from the mistral.ai platform, specifically the ‘mitral-8x7b-instruct-v01’ model. The function helps in customizing generating parameters and interacts with IBM Watson’s machine learning services.	`def llm_model(prompt_txt, params=None):` `    model_id = 'mistralai/mixtral-8x7b-instruct-v01'` `    default_params = {` `        "max_new_tokens": 256,` `        "min_new_tokens": 0,` `        "temperature": 0.5,` `        "top_p": 0.2,` `        "top_k": 1` `    }` `    if params:` `        default_params.update(params)` `    parameters = {` `        GenParams.MAX_NEW_TOKENS:` `default_params["max_new_tokens"],  # this controls the` `maximum number of tokens in the generated output`

		`` GenParams.MIN_NEW_TOKENS: default_params["min_new_tokens"], # this controls the minimum number of tokens in the generated output GenParams.TEMPERATURE: default_params["temperature"], # this randomness or creativity of the model's responses GenParams.TOP_P: default_params["top_p"], GenParams.TOP_K: default_params["top_k"] } credentials = { "url": "https://us-south.ml.cloud.ibm.com" } project_id = "skills-network" model = Model(model_id=model_id, params=parameters, credentials=credentials, project_id=project_id) mixtral_llm = WatsonxLLM(model=model) response = mixtral_llm.invoke(prompt_txt) return response ```
Zero-shot prompt	Zero-shot learning is crucial for testing a model's ability to apply its pre-trained knowledge to new, unseen tasks without additional training. This capability is valuable	``` prompt = """Classify the following statement as true or false: 'The Eiffel Tower is located in Berlin.' Answer: ```

	for gauging the model's generalization skills.	`"""` `response = llm_model(prompt, params)` `print(f"prompt: {prompt}\n")` `print(f"response : {response}\n")`
One-shot prompt	One-shot learning example where the model is given a single example to help guide its translation from English to French. The prompt provides a sample translation pairing, "How is the weather today?" translated to "Comment est le temps aujourd'hui?" This example serves as a guide for the model to understand the task context and desired format. The model is then tasked with translating a new sentence, "Where is the nearest supermarket?" without further guidance.	`params = {` `    "max_new_tokens": 20,` `    "temperature": 0.1,` `}` `prompt = """Here is an example of translating a sentence from English to French:` `            English: “How is the weather today?”` `            French: “Comment est le temps aujourd'hui?”` `            Now, translate the following sentence from English to French:` `            English: “Where is the nearest supermarket?”` `"""` `response = llm_model(prompt, params)` `print(f"prompt: {prompt}\n")` `print(f"response : {response}\n")`
Few-shot prompt	This code snippet classifies emotions using a few-shot learning approach. The prompt includes various examples where statements are associated with their respective emotions.	``#parameters  `max_new_tokens` to 10, which constrains the model to generate brief responses`` `params = {` `    "max_new_tokens": 10,` `}`

<table>
<tr><td></td><td></td><td><pre>prompt = """Here are few examples of classifying
emotions in statements:

            Statement: 'I just won my first marathon!'
            Emotion: Joy

            Statement: 'I can't believe I lost my keys
again.'
            Emotion: Frustration

            Statement: 'My best friend is moving to
another country.'
            Emotion: Sadness

            Now, classify the emotion in the following
statement:
            Statement: 'That movie was so scary I had to
cover my eyes.’

"""
response = llm_model(prompt, params)
print(f"prompt: {prompt}\n")
print(f"response : {response}\n")</pre></td></tr>
<tr><td>Chain-of-thought (CoT) prompting</td><td>The Chain-of-Thought (CoT) prompting technique, designed to guide the model through a sequence of reasoning steps to solve a problem.

The CoT technique involves structuring the prompt by</td><td><pre>params = {
    "max_new_tokens": 512,
    "temperature": 0.5,
}</pre></td></tr>
</table>

<table>
<tr><td></td><td>instructing the model to “Break down each step of your calculation.” This encourages the model to include explicit reasoning steps, mimicking human-like problem-solving processes.</td><td><pre>prompt = """Consider the problem: 'A store had 22 apples. They sold 15 apples today and got a new delivery of 8 apples.
            How many apples are there now?’

            Break down each step of your calculation

"""
response = llm_model(prompt, params)
print(f"prompt: {prompt}\n")
print(f"response : {response}\n")</pre></td></tr>
<tr><td>Self-consistency</td><td>This code snippet determines the consistent result for age-related problems and generates multiple responses. The ‘params’ dictionary specifies the maximum number of tokens to generate responses.</td><td><pre>params = {
    "max_new_tokens": 512,
}

prompt = """When I was 6, my sister was half of my age. Now I am 70, what age is my sister?

            Provide three independent calculations and explanations, then determine the most consistent result.

"""
response = llm_model(prompt, params)
print(f"prompt: {prompt}\n")
print(f"response : {response}\n")</pre></td></tr>
<tr><td>Prompt template</td><td>A key concept in langchain, it helps to translate user input and parameters into instructions for a language model. This can be used to guide a model's response, helping it understand the context</td><td><pre>model_id = 'mistralai/mixtral-8x7b-instruct-v01'

parameters = {
    GenParams.MAX_NEW_TOKENS: 256, # this controls the maximum number of tokens in the generated output</pre></td></tr>
</table>

	and generate relevant and coherent language-based output.	``` GenParams.TEMPERATURE: 0.5, # this randomness or creativity of the model's responses } credentials = { "url": "https://us-south.ml.cloud.ibm.com" } project_id = "skills-network" model = Model(model_id=model_id, params=parameters, credentials=credentials, project_id=project_id) mixtral_llm = WatsonxLLM(model=model) mixtral_llm ```
Text summarization	Text summarization agent designed to help summarize the content you provide to the LLM. You can store the content to be summarized in a variable, allowing for repeated use of the prompt.	``` content = """ The rapid advancement of technology in the 21st century has transformed various industries, including healthcare, education, and transportation. Innovations such as artificial intelligence, machine learning, and the Internet of Things have revolutionized how we approach everyday tasks and complex problems. For instance, AI-powered diagnostic tools are improving the accuracy and speed of medical diagnoses, while smart transportation systems are making cities more efficient and reducing traffic congestion. ```

		`        Moreover, online learning platforms are making education more accessible to people around the world, breaking down geographical and financial barriers.` `        These technological developments are not only enhancing productivity but also contributing to a more interconnected and informed society.` `"""` `template = """Summarize the {content} in one sentence.` `"""` `prompt = PromptTemplate.from_template(template)` `llm_chain = LLMChain(prompt=prompt, llm=mixtral_llm)` `response = llm_chain.invoke(input = {"content": content})` `print(response["text"])`
Question answering	An agent that enables the LLM to learn from the provided content and answer questions based on what it has learned. Occasionally, if the LLM does not have sufficient information, it might generate a speculative answer. To manage this, you'll specifically instruct it to respond with "Unsure about the answer" if it is uncertain about the correct response.	`content = """` `        The solar system consists of the Sun, eight planets, their moons, dwarf planets, and smaller objects like asteroids and comets.` `        The inner planets—Mercury, Venus, Earth, and Mars—are rocky and solid.` `        The outer planets—Jupiter, Saturn, Uranus, and Neptune—are much larger and gaseous.` `"""` `question = "Which planets in the solar system are rocky and solid?"` `template = """`

		<pre> Answer the {question} based on the {content}. Respond "Unsure about answer" if not sure about the answer. Answer: """ prompt = PromptTemplate.from_template(template) output_key = "answer" llm_chain = LLMChain(prompt=prompt, llm=mixtral_llm, output_key=output_key) response = llm_chain.invoke(input = {"question":question ,"content": content}) print(response["answer"])</pre>
Code generation	An agent that is designed to generate SQL queries based on given descriptions. It interprets the requirements from your input and translates them into executable SQL code.	<pre>description = """ Retrieve the names and email addresses of all customers from the 'customers' table who have made a purchase in the last 30 days. The table 'purchases' contains a column 'purchase_date' """ template = """ Generate an SQL query based on the {description} SQL Query: """</pre>

<table>
<tr><td></td><td></td><td>

```
prompt = PromptTemplate.from_template(template)
output_key = "query"

llm_chain = LLMChain(prompt=prompt, llm=mixtral_llm,
output_key=output_key)
response = llm_chain.invoke(input =
{"description":description})
print(response["query"])
```

</td></tr>
<tr><td>Role playing</td><td>Configures the LLM to assume specific roles as defined by us, enabling it to follow predetermined rules and behave like a task-oriented chatbot.</td><td>

```
role = """
        game master
"""

tone = "engaging and immersive"

template = """
            You are an expert {role}. I have this
question {question}. I would like our conversation to be
{tone}.

            Answer:

"""
prompt = PromptTemplate.from_template(template)
output_key = "answer"

llm_chain = LLMChain(prompt=prompt, llm=mixtral_llm,
output_key=output_key)
```

</td></tr>
<tr><td>class_names</td><td>This code snippet maps numerical labels to their corresponding textual descriptions to classify tasks. This code helps in machine</td><td>

```
class_names = {0: "negative", 1: "positive"}
class_names
```

</td></tr>
</table>

<table>
<tr>
<td></td>
<td>learning to interpret the output model, where the model’s predictions are numerical and should be presented in a more human-readable format.</td>
<td></td>
</tr>
<tr>
<td>read_and_split_text</td>
<td>Involves opening the file, reading its contents, and splitting the text into individual paragraphs. Each paragraph represents a section of the company policies. You can also filter out any empty paragraphs to clean your data set.</td>
<td><pre><code>def read_and_split_text(filename):
    with open(filename, 'r', encoding='utf-8') as file:
        text = file.read()
    # Split the text into paragraphs (simple split by newline characters)
    paragraphs = text.split('\n')
    # Filter out any empty paragraphs or undesired entries
    paragraphs = [para.strip() for para in paragraphs if len(para.strip()) > 0]
    return paragraphs

Read the text file and split it into paragraphs
paragraphs = read_and_split_text('companyPolicies.txt')
paragraphs[0:10]</code></pre></td>
</tr>
<tr>
<td>encode_contexts</td>
<td>This code snippet encodes a list of texts into embeddings using content_tokenizer and context_encoder. This code helps iterate through each text in the input list, tokenizes and encodes it, and then appends the pooler_output to the embeddings list. The resulting embeddings get stored in the context_embeddings variables and generate embeddings from text data for</td>
<td><pre><code>def encode_contexts(text_list):
    # Encode a list of texts into embeddings
    embeddings = []
    for text in text_list:
        inputs = context_tokenizer(text, return_tensors='pt', padding=True, truncation=True, max_length=256)
        outputs = context_encoder(**inputs)
        embeddings.append(outputs.pooler_output)
    return torch.cat(embeddings).detach().numpy()</code></pre></td>
</tr>
</table>

	various natural language processing (NLP) applications.	# you would now encode these paragraphs to create embeddings. context_embeddings = encode_contexts(paragraphs)
import faiss	FAISS (Facebook AI Similarity Search) is an efficient library developed by Facebook for similarity search and clustering of dense vectors. FAISS is designed for fast similarity search, which is particularly valuable when dealing with large data sets. It is highly suitable for tasks in natural language processing where retrieval speed is critical. It effectively handles large volumes of data, maintaining performance even as data set sizes increase.	import faiss # Convert list of numpy arrays into a single numpy array embedding_dim = 768 # This should match the dimension of your embeddings context_embeddings_np = np.array(context_embeddings).astype('float32') # Create a FAISS index for the embeddings index = faiss.IndexFlatL2(embedding_dim) index.add(context_embeddings_np) # Add the context embeddings to the index
search_relevant_contexts	This code snippet is useful in searching relevant contexts for a given question. It tokenizes the question using the question_tokenizer, encodes the question using question_encoder, and searches an index for retrieving the relevant context based on question embedding.	def search_relevant_contexts(question, question_tokenizer, question_encoder, index, k=5): """ Searches for the most relevant contexts to a given question. Returns: tuple: Distances and indices of the top k relevant contexts. """ # Tokenize the question question_inputs = question_tokenizer(question, return_tensors='pt')

		`` # Encode the question to get the embedding question_embedding = question_encoder(**question_inputs).pooler_output.detach().numpy() # Search the index to retrieve top k relevant contexts D, I = index.search(question_embedding, k) return D, I ```
generate_answer_without_context	This code snippet generates responses using the entered prompt without requiring additional context. It tokenizes the input questions using the tokenizer, generates the output text using the model, and decodes the generated text to obtain the answer.	``` def generate_answer_without_context(question): # Tokenize the input question inputs = tokenizer(question, return_tensors='pt', max_length=1024, truncation=True) # Generate output directly from the question without additional context summary_ids = model.generate(inputs['input_ids'], max_length=150, min_length=40, length_penalty=2.0, num_beams=4, early_stopping=True) # Decode and return the generated text answer = tokenizer.decode(summary_ids[0], skip_special_tokens=True) return answer ```
Generating answers with DPR contexts	Answers are generated when the model utilizes contexts retrieved via DPR, which are expected to enhance the answer's relevance and depth:	``` def generate_answer(contexts): # Concatenate the retrieved contexts to form the input to BART input_text = ' '.join(contexts) inputs = tokenizer(input_text, return_tensors='pt', max_length=1024, truncation=True) ```

		`` # Generate output using BART summary_ids = model.generate(inputs['input_ids'], max_length=150, min_length=40, length_penalty=2.0, num_beams=4, early_stopping=True) return tokenizer.decode(summary_ids[0], skip_special_tokens=True) ```
aggregate_embeddings function	The function aggregate_embeddings takes token indices and their corresponding attention masks, and uses a BERT model to convert these tokens into word embeddings. It then filters out the embeddings for zero-padded tokens and computes the mean embedding for each sequence. This helps in reducing the dimensionality of the data while retaining the most important information from the embeddings.	``` def aggregate_embeddings(input_ids, attention_masks, bert_model=bert_model): """ Converts token indices and masks to word embeddings, filters out zero-padded embeddings, and aggregates them by computing the mean embedding for each input sequence. """ mean_embeddings = [] # Process each sequence in the batch print('number of inputs',len(input_ids)) for input_id, mask in tqdm(zip(input_ids, attention_masks)): input_ids_tensor = torch.tensor([input_id]).to(DEVICE) mask_tensor = torch.tensor([mask]).to(DEVICE) with torch.no_grad(): # Obtain the word embeddings from the BERT model word_embeddings = bert_model(input_ids_tensor, attention_mask=mask_tensor)[0].squeeze(0) ```

<table>
<tr><td></td><td></td><td><pre>
            # Filter out the embeddings at positions
where the mask is zero
            valid_embeddings_mask=mask_tensor[0] != 0
            valid_embeddings =
word_embeddings[valid_embeddings_mask,:]
            # Compute the mean of the filtered
embeddings
            mean_embedding =
valid_embeddings.mean(dim=0)

mean_embeddings.append(mean_embedding.unsqueeze(0))

    # Concatenate the mean embeddings from all sequences
in the batch
    aggregated_mean_embeddings =
torch.cat(mean_embeddings)
    return aggregated_mean_embeddings
</pre></td></tr>
<tr><td>text_to_emb</td><td>Designed to convert a list of text strings into their corresponding embeddings using a pre-defined tokenizer.</td><td><pre>
def text_to_emb(list_of_text,max_input=512):
    data_token_index =
tokenizer.batch_encode_plus(list_of_text,
add_special_tokens=True,padding=True,truncation=True,max
_length=max_input)

question_embeddings=aggregate_embeddings(data_token_inde
x['input_ids'], data_token_index['attention_mask'])
    return question_embeddings
</pre></td></tr>
<tr><td>process_song</td><td>Convert both the predefined appropriateness questions and the song lyrics into "RAG embeddings" and measure the similarity</td><td><pre>
import re
def process_song(song):
    # Remove line breaks from the song
    song_new = re.sub(r'[\n]', ' ', song)
</pre></td></tr>
</table>

<table>
<tr>
<td></td>
<td>between them to determine the appropriateness.</td>
<td><pre><code>
    # Remove single quotes from the song
    song_new = [song_new.replace("\'", "")]

    return song_new
</code></pre></td>
</tr>
<tr>
<td>RAG_QA</td>
<td>This code snippet performs question-answering using question embeddings and provides embeddings. It helps reshape the results for processing, sorting the indices in descending order, and printing the top ‘n-responses’ based on the highest dot product values.</td>
<td><pre><code>def RAG_QA(embeddings_questions, embeddings,
n_responses=3):
    # Calculate the dot product between the question
embeddings and the provided embeddings (transpose of the
second matrix for proper alignment).
    dot_product = embeddings_questions @ embeddings.T

    # Reshape the dot product results to a 1D tensor for
easier processing.
    dot_product = dot_product.reshape(-1)

    # Sort the indices of the dot product results in
descending order (setting descending to False should be
True for typical similarity tasks).
    sorted_indices = torch.argsort(dot_product,
descending=True)

    # Convert sorted indices to a list for easier
iteration.
    sorted_indices = sorted_indices.tolist()

    # Print the top 'n_responses' responses from the
sorted list, which correspond to the highest dot product
values.
    for index in sorted_indices[:n_responses]:
        print(yes_responses[index])
</code></pre></td>
</tr>
</table>

<table>
<tr>
<td>model_name_or_path</td>
<td>This code snippet defines the model name to ‘gpt2’ and initializes the token and model using the GPT-2 model. In this code, add special tokens for padding by keeping the maximum sequence length to 1024.</td>
<td><pre><code># Define the model name or path
model_name_or_path = "gpt2"

Initialize tokenizer and model
tokenizer = GPT2Tokenizer.from_pretrained(model_name_or_path, use_fast=True)
model = GPT2ForSequenceClassification.from_pretrained(model_name_or_path, num_labels=1)

Add special tokens if necessary
tokenizer.pad_token = tokenizer.eos_token
model.config.pad_token_id = model.config.eos_token_id

Define the maximum length
max_length = 1024</code></pre></td>
</tr>
<tr>
<td>add_combined_columns</td>
<td>This code snippet combines the prompt with chosen and rejected responses in a data set example. It combines with the ‘Human:’ and ‘Assistant:’ for clarity. This function modifies each example in the ’train’ split the data set by creating new columns ‘prompt_chosen’ and ‘prompt_rejected’ with the combined text.</td>
<td><pre><code># Define a function to combine 'prompt' with 'chosen' and 'rejected' responses
def add_combined_columns(example):
    # Combine 'prompt' with 'chosen' response, formatting it with "Human:" and "Assistant:" labels
    example['prompt_chosen'] = "\n\nHuman: " + example["prompt"] + "\n\nAssistant: " + example["chosen"]

    # Combine 'prompt' with 'rejected' response, formatting it with "Human:" and "Assistant:" labels
    example['prompt_rejected'] = "\n\nHuman: " + example["prompt"] + "\n\nAssistant: " + example["rejected"]</code></pre></td>
</tr>
</table>

		`` # Return the modified example return example # Apply the function to each example in the 'train' split of the dataset dataset['train'] = dataset['train'].map(add_combined_columns) ```
RetrievalQA	This code snippet creates an example for ‘RetrievalQA’ using a language model and document retriever.	``` qa = RetrievalQA.from_chain_type(llm=flan_ul2_llm, chain_type="stuff", retriever=docsearch.as_retriever(), return_source_documents=False) query = "what is mobile policy?" qa.invoke(query) ```