Cheat Sheet: Foundations of Generative AI and LangChain

Estimated time needed: 10 minutes

Package/Method	Description	Code Example
pip install	Installs the necessary Python libraries required for the course.	`%%capture` `!pip install "ibm-watsonx-ai==1.0.8" --user` `!pip install "langchain==0.2.11" --user` `!pip install "langchain-ibm==0.1.7" --user` `!pip install "langchain-core==0.2.43" --user`
warnings	Suppresses warnings generated by the code to keep the output clean.	`import warnings` `warnings.filterwarnings('ignore')`
WatsonxLLM	Facilitates interaction with IBM's Watsonx large language models.	`from langchain_ibm import WatsonxLLM` `granite_llm = WatsonxLLM(` `  model_id="ibm/granite-3-2-8b-instruct",` `  url="https://us-south.ml.cloud.ibm.com",` `  project_id="skills-network",` `  params={` `  "max_new_tokens": 256,` `  "temperature": 0.5,` `  "top_p": 0.2` `  }` `)`
llm_model	Invokes IBM Watsonx LLM with a given prompt and parameters.	`def llm_model(prompt_txt, params=None):` `  model_id = "ibm/granite-3-2-8b-instruct"` `  default_params = {` `  "max_new_tokens": 256,` `  "temperature": 0.5,` `  "top_p": 0.2` `  }` `  if params:` `  default_params.update(params)` `granite_llm = WatsonxLLM(` `  model_id=model_id,` `  url="https://us-south.ml.cloud.ibm.com",` `  project_id="skills-network",`

		params=default_params) response = granite_llm.invoke(prompt_txt) return response</code></pre>
GenParams	A class from the ibm_watsonx_ai.metanames module that provides parameters for controlling text generation, including max_new_tokens, min_new_tokens, temperature, top_p, and top_k.	from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams // Get example values GenParams().get_example_values() // Use in parameters parameters = { GenParams.MAX_NEW_TOKENS: 256, GenParams.TEMPERATURE: 0.5, }
Basic Prompt	The simplest form of prompting, in which you provide a short text or phrase to the model without special formatting or instructions. The model then generates a continuation based on patterns it has learned during training.	params = { "max_new_tokens": 128, "min_new_tokens": 10, "temperature": 0.5, "top_p": 0.2, "top_k": 1 } prompt = “The wind is” response = llm_model(prompt, params) print(f”prompt: {prompt}\n”) print(f”response : {response}\n”)
Zero-shot Prompt	A technique in which the model performs a task without any examples or prior specific training on that task. This approach tests the model's ability to understand instructions and apply its knowledge to a new context without demonstration.	prompt = """Classify the following statement as true or false: 'The Eiffel Tower is located in Berlin.' Answer: “”” response = llm_model(prompt, params) print(f”prompt: {prompt}\n”) print(f”response : {response}\n”)

One-shot Prompt	Provides the model with a single example of the task before asking it to perform a similar task. This technique gives the model a pattern to follow, improving its understanding of the desired output format and style.	``` 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) ```
Few-shot Prompt	Extends the one-shot approach by providing multiple examples (typically 2-5) before asking the model to perform the task. These examples establish a clearer pattern and context, helping the model better understand the expected output format, style, and reasoning.	``` params = { "max_new_tokens": 10, } 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) ```
Chain-of-thought (CoT) Prompting	Encourages the model to break down complex problems into step-by-step reasoning before arriving at a final answer. By explicitly showing or requesting intermediate steps, this technique improves the model's problem-solving abilities and reduces errors in tasks requiring multi-step reasoning.	``` params = { "max_new_tokens": 512, "temperature": 0.5, } 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) ```

Self-consistency	An advanced technique where the model generates multiple independent solutions or answers to the same problem, then evaluates these different approaches to determine the most consistent or reliable result. This method helps improve accuracy by leveraging the model's ability to approach problems from different angles.	``` 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) ```
PromptTemplate	A class from langchain_core.prompts module that acts as a reusable structure for generating prompts with dynamic values. It allows you to define a consistent format while leaving placeholders for variables that change with each use case.	``` from langchain_core.prompts import PromptTemplate template = """Tell me a {adjective} joke about {content}.""" prompt = PromptTemplate.from_template(template) // Format the prompt formatted_prompt = prompt.format(adjective="funny", content="chickens") ```
RunnableLambda	A class from langchain_core.runnables that wraps a Python function into a LangChain runnable component. It's used to create transformation steps in a chain, especially for formatting or processing data.	``` from langchain_core.runnables import RunnableLambda // Define a function to ensure proper formatting def format_prompt(variables): return prompt.format(**variables) // Use in a chain joke_chain = (RunnableLambda(format_prompt) | llm | StrOutputParser()) ```
StrOutputParser	A class from langchain_core.output_parsers that simply extracts string outputs from LLM responses. It's commonly used as the final step in a LangChain chain to ensure a clean string is returned.	``` from langchain_core.output_parsers import StrOutputParser // Create a chain that returns a string chain = (RunnableLambda(format_prompt) | llm | StrOutputParser()) // Run the chain response = chain.invoke({"variable": "value"}) ```

LCEL Pattern	LangChain Expression Language (LCEL) is a pattern for building LangChain applications using the pipe operator (|) for more flexible composition. It offers better composability, clearer visualization of data flow, and more flexibility when constructing complex chains.	``` // Basic LCEL pattern chain = (RunnableLambda(format_prompt) # Format input | llm # Process with LLM | StrOutputParser() # Parse output) // Run the chain result = chain.invoke({“variable”: “value”}) // More complex example template = “”” Answer the {question} based on the {content}. Respond “Unsure about answer” if not sure. Answer: “”” prompt = PromptTemplate.from_template(template) qa_chain = (RunnableLambda(format_prompt) | llm | StrOutputParser()) answer = qa_chain.invoke({ “question”: “Which planets are rocky?”, “content”: “The inner planets are rocky.” }) ```

Author

Hailey Quach

Skills Network