Google Gemini LangChain Cheatsheet

April 28, 2025
7 minute read
View Code

The `langchain-google-genai` provides access to Google's powerful Gemini models directly via the Gemini API & Google AI Studio. Google AI Studio enables rapid prototyping and experimentation, making it an ideal starting point for individual developers. LangChain is a framework for developing AI applications. The `langchain-google-genai` package connects LangChain with Google's Gemini models. LangGraph is a library for building stateful, multi-actor applications with LLMs.

All examples use the `gemini-2.0-flash` model. Gemini 2.5 Pro and 2.5 Flash can be used via `gemini-2.5-pro-preview-03-25` and `gemini-2.5-flash-preview-04-17`. All model ids can be found in the Gemini API docs.

Start for free and get your API key from Google AI Studio.

1. Install the package `langchain-google-genai`

```
%pip install langchain-google-genai
```

2. Set your API key

```python
import getpass
import os

if "GOOGLE_API_KEY" not in os.environ:
    os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter your Google AI API key: ")
```

3. Get Started with LangChain.

Google Gemini with LangChain Chat Models

Learn how to use Google Gemini chat models within LangChain for basic text generation and conversation tasks.

```python
from langchain_google_genai import ChatGoogleGenerativeAI

# Initialize model
llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",
    temperature=0,
    max_tokens=None,
    timeout=None,
    max_retries=2,
)

# Simple invocation
messages = [
    ("system", "You are a helpful assistant that translates English to French."),
    ("human", "I love programming."),
]
response = llm.invoke(messages)
print(response.content)  # Output: J'adore la programmation.
```

Chain calls with Prompt Template

Discover how to chain LangChain prompt templates with Gemini models for flexible and dynamic input processing.

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate

# Initialize model
llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",
    temperature=0,
)

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant that translates {input_language} to {out
    ("human", "{input}"),
])

chain = prompt | llm
result = chain.invoke({
    "input_language": "English",
    "output_language": "German",
    "input": "I love programming.",
})
print(result.content)  # Output: Ich liebe Programmieren.
```

Image Input

Explore using image inputs (URLs or local files) with multimodal Gemini models in LangChain for vision tasks.

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.messages import HumanMessage
import base64

# Initialize model
llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash")

# Using an image URL
message_url = HumanMessage(
    content=[
        {"type": "text", "text": "Describe this image."},
        {"type": "image_url", "image_url": "https://picsum.photos/seed/picsum/200/30
    ]
)
result_url = llm.invoke([message_url])
print(result_url.content)

# Using a local image
local_image_path = "../assets/react.png"
with open(local_image_path, "rb") as image_file:
    encoded_image = base64.b64encode(image_file.read()).decode('utf-8')

message_local = HumanMessage(
    content=[
        {"type": "text", "text": "Describe this image."},
        {"type": "image_url", "image_url": f"{encoded_image}"}
    ]
)
result_local = llm.invoke([message_local])
print(result_local.content)
```

Audio Input

Understand how to provide audio file data to Gemini models via LangChain for audio processing like transcription.

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.messages import HumanMessage
import base64

# Initialize model
llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash")

audio_file_path = "../path/to/your/audio/file.mp3"
audio_mime_type = "audio/mpeg"

with open(audio_file_path, "rb") as audio_file:
    encoded_audio = base64.b64encode(audio_file.read()).decode('utf-8')

message = HumanMessage(
    content=[
        {"type": "text", "text": "Transcribe this audio."},
        {"type": "media", "data": encoded_audio, "mime_type": audio_mime_type}
    ]
)
response = llm.invoke([message])
print(response.content)
```

Video Input

See how to utilize video file input with Gemini models in LangChain for video understanding and analysis.

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.messages import HumanMessage
import base64

# Initialize model
llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash")

video_file_path = "../path/to/your/video/file.mp4"
video_mime_type = "video/mp4"

with open(video_file_path, "rb") as video_file:
    encoded_video = base64.b64encode(video_file.read()).decode('utf-8')

message = HumanMessage(
    content=[
        {"type": "text", "text": "Describe what's happening in this video."},
        {"type": "media", "data": encoded_video, "mime_type": video_mime_type}
    ]
)
response = llm.invoke([message])
print(response.content)
```

Image Generation

Generate images from text prompts using specialized Gemini models integrated with LangChain.

```python
from langchain_google_genai import ChatGoogleGenerativeAI
import base64
from IPython.display import Image, display

# Initialize model for image generation
llm = ChatGoogleGenerativeAI(model="models/gemini-2.0-flash-exp-image-generation")

message = {
    "role": "user",
    "content": "Generate an image of a cat wearing a hat.",
}

response = llm.invoke(
    [message],
    generation_config=dict(response_modalities=["TEXT", "IMAGE"]),
)

# Display the generated image
image_base64 = response.content[0].get("image_url").get("url").split(",")[-1]
image_data = base64.b64decode(image_base64)
display(Image(data=image_data, width=300))
```

Tool Calling/Function Calling

Learn to implement tool calling (function calling) with Gemini models in LangChain to execute custom functions.

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.tools import tool
from langchain_core.messages import ToolMessage

# Define a tool
@tool(description="Get the current weather in a given location")
def get_weather(location: str) -> str:
    return "It's sunny."

# Initialize model and bind the tool
llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash")
llm_with_tools = llm.bind_tools([get_weather])

# Invoke with a query that should trigger the tool
query = "What's the weather in San Francisco?"
ai_msg = llm_with_tools.invoke(query)

# Access tool calls in the response
print(ai_msg.tool_calls)

# Pass tool results back to the model
tool_message = ToolMessage(
    content=get_weather(*ai_msg.tool_calls[0]['args']),
    tool_call_id=ai_msg.tool_calls[0]['id']
)
final_response = llm_with_tools.invoke([ai_msg, tool_message])
print(final_response.content)
```

Built-in Tools (Google Search, Code Execution)

Leverage Gemini's built-in tools like Google Search and Code Execution directly within your LangChain applications.

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from google.ai.generativelanguage_v1beta.types import Tool as GenAITool

# Initialize model
llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash")

# Google Search
search_resp = llm.invoke(
    "When is the next total solar eclipse in US?",
    tools=[GenAITool(google_search={})],
)
print(search_resp.content)

# Code Execution
code_resp = llm.invoke(
    "What is 2*2, use python",
    tools=[GenAITool(code_execution={})],
)

for c in code_resp.content:
    if isinstance(c, dict):
        if c["type"] == 'code_execution_result':
            print(f"Code execution result: {c['code_execution_result']}")
        elif c["type"] == 'executable_code':
            print(f"Executable code: {c['executable_code']}")
    else:
        print(c)
```

Structured Output

Control Gemini model output to conform to a specific Pydantic schema for reliable structured data extraction in LangChain.

```python
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_google_genai import ChatGoogleGenerativeAI

# Define the desired structure
class Person(BaseModel):
    '''Information about a person.'''
    name: str = Field(..., description="The person's name")
    height_m: float = Field(..., description="The person's height in meters")

# Initialize the model
llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0)
structured_llm = llm.with_structured_output(Person)

# Invoke the model with a query asking for structured information
result = structured_llm.invoke("Who was the 16th president of the USA, and how tall
print(result)  # Output: name='Abraham Lincoln' height_m=1.93
```

Token Usage Tracking

Track token usage for Gemini model calls within LangChain to monitor costs and API consumption.

```python
from langchain_google_genai import ChatGoogleGenerativeAI

# Initialize model
llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash")

result = llm.invoke("Explain the concept of prompt engineering in one sentence.")

print(result.content)
print("\nUsage Metadata:")
print(result.usage_metadata)
```

Google Gemini Embeddings with LangChain

Generate powerful text embeddings using Google Gemini models within the LangChain framework for semantic understanding.

```python
from langchain_google_genai import GoogleGenerativeAIEmbeddings

embeddings = GoogleGenerativeAIEmbeddings(model="models/gemini-embedding-exp-03-07")

# Embed a single query
vector = embeddings.embed_query("hello, world!")

# Embed multiple documents
vectors = embeddings.embed_documents([
    "Today is Monday",
    "Today is Tuesday",
    "Today is April Fools day",
])
```

Using with Vector Store

Integrate Gemini text embeddings with LangChain vector stores for efficient similarity search and information retrieval.

```python
from langchain_google_genai import GoogleGenerativeAIEmbeddings
from langchain_core.vectorstores import InMemoryVectorStore

# Initialize embeddings
embeddings = GoogleGenerativeAIEmbeddings(model="models/gemini-embedding-exp-03-07")

text = "LangChain is the framework for building context-aware reasoning applications

# Create vector store and retriever
vectorstore = InMemoryVectorStore.from_texts([text], embedding=embeddings)
retriever = vectorstore.as_retriever()

# Retrieve similar documents
retrieved_documents = retriever.invoke("What is LangChain?")
print(retrieved_documents[0].page_content)
```

Task Types

Optimize Gemini embedding performance by selecting the appropriate task type (e.g., retrieval, classification) in LangChain.

```
%pip install scikit-learn
```

```python
from langchain_google_genai import GoogleGenerativeAIEmbeddings
from sklearn.metrics.pairwise import cosine_similarity

# Different task types for different use cases
query_embeddings = GoogleGenerativeAIEmbeddings(
    model="models/gemini-embedding-exp-03-07",
    task_type="RETRIEVAL_QUERY"  # For queries
)
doc_embeddings = GoogleGenerativeAIEmbeddings(
    model="models/gemini-embedding-exp-03-07",
    task_type="RETRIEVAL_DOCUMENT"  # For documents
)

# Compare similarity
q_embed = query_embeddings.embed_query("What is the capital of France?")
d_embed = doc_embeddings.embed_documents(["The capital of France is Paris.", "Philip

for i, d in enumerate(d_embed):
    similarity = cosine_similarity([q_embed], [d])[0][0]
    print(f"Document {i+1} similarity: {similarity}")
```

Thanks for reading! If you have any questions or feedback, please let me know on Twitter or LinkedIn.