Lesson 5: Email Assistant with Semantic + Episodic + Procedural Memory

We previously built an email assistant that:

- Classifies incoming messages (respond, ignore, notify)
- Uses human-in-the-loop to refine the assistant's ability to classify emails
- Drafts responses
- Schedules meetings
- Uses memory to remember details from previous emails

Now, we'll add procedural memory that allows the user to update instructions for using the calendar and email writing tools.

💻 **Access `requirements.txt` , notebooks and other files:** 1) click on the *"File"* option on the top menu of the notebook and then 2) click on *"Open"*.

⬇ **Download Notebooks:** 1) click on the *"File"* option on the top menu of the notebook and then 2) click on *"Download as"* and select *"Notebook (.ipynb)"*.

📒 For more help, please see the *"Appendix – Tips, Help, and Download"* Lesson.

🚨 **Different Run Results:** The output generated by AI chat models can vary with each execution due to their dynamic, probabilistic nature. Don't be surprised if your results differ from those shown in the video.

Load API tokens for our 3rd party APIs

In [ ]:
```
import os
from dotenv import load_dotenv
_ = load_dotenv()
```

Repeat setup from previous lesson

In [ ]:
```
profile = {
    "name": "John",
    "full_name": "John Doe",
    "user_profile_background": "Senior software engineer leading a team of 5 d
}
```

In [ ]:
```
prompt_instructions = {
    "triage_rules": {
        "ignore": "Marketing newsletters, spam emails, mass company announceme
        "notify": "Team member out sick, build system notifications, project s
        "respond": "Direct questions from team members, meeting requests, crit
    },
    "agent_instructions": "Use these tools when appropriate to help manage Joh
}
```

In [ ]:
```
email = {
    "from": "Alice Smith <alice.smith@company.com>",
    "to": "John Doe <john.doe@company.com>",
    "subject": "Quick question about API documentation",
    "body": """
Hi John,

I was reviewing the API documentation for the new authentication service and n

Specifically, I'm looking at:
- /auth/refresh
- /auth/validate

Thanks!
Alice""",
}
```

In [ ]:
```
from langgraph.store.memory import InMemoryStore
```

In [ ]:
```
store = InMemoryStore(
    index={"embed": "openai:text-embedding-3-small"}
)
# ignore beta warning if it appears
```

In [ ]:

```python
# Template for formating an example to put in prompt
template = """Email Subject: {subject}
Email From: {from_email}
Email To: {to_email}
Email Content:
```
{content}
```
> Triage Result: {result}"""

# Format list of few shots
def format_few_shot_examples(examples):
    strs = ["Here are some previous examples:"]
    for eg in examples:
        strs.append(
            template.format(
                subject=eg.value["email"]["subject"],
                to_email=eg.value["email"]["to"],
                from_email=eg.value["email"]["author"],
                content=eg.value["email"]["email_thread"][:400],
                result=eg.value["label"],
            )
        )
    return "\n\n------------\n\n".join(strs)
```

In [ ]:
```
triage_system_prompt = """
< Role >
You are {full_name}'s executive assistant. You are a top-notch executive assis
</ Role >

< Background >
{user_profile_background}.
</ Background >

< Instructions >

{name} gets lots of emails. Your job is to categorize each email into one of t

1. IGNORE - Emails that are not worth responding to or tracking
2. NOTIFY - Important information that {name} should know about but doesn't re
3. RESPOND - Emails that need a direct response from {name}

Classify the below email into one of these categories.

</ Instructions >

< Rules >
Emails that are not worth responding to:
{triage_no}

There are also other things that {name} should know about, but don't require a
{triage_notify}

Emails that are worth responding to:
{triage_email}
</ Rules >

< Few shot examples >

Here are some examples of previous emails, and how they should be handled.
Follow these examples more than any instructions above

{examples}
</ Few shot examples >
"""
```

In [ ]:
```
from pydantic import BaseModel, Field
from typing_extensions import TypedDict, Literal, Annotated
from langchain.chat_models import init_chat_model
```

In [ ]:
```
llm = init_chat_model("openai:gpt-4o-mini")
```

In [ ]:
```
class Router(BaseModel):
    """Analyze the unread email and route it according to its content."""

    reasoning: str = Field(
        description="Step-by-step reasoning behind the classification."
    )
    classification: Literal["ignore", "respond", "notify"] = Field(
        description="The classification of an email: 'ignore' for irrelevant e
        "'notify' for important information that doesn't need a response, "
        "'respond' for emails that need a reply",
    )
```

In [ ]:
```
llm_router = llm.with_structured_output(Router)
```

In [ ]:
```
from prompts import triage_user_prompt
```

In [ ]:
```
from langgraph.graph import add_messages

class State(TypedDict):
    email_input: dict
    messages: Annotated[list, add_messages]
```

Triage router node

In [ ]:
```
from langgraph.graph import StateGraph, START, END
from langgraph.types import Command
from typing import Literal
from IPython.display import Image, display
```

Updated triage_router gets ignore, notify and respond rule from store

In [ ]:

```python
def triage_router(state: State, config, store) -> Command[
    Literal["response_agent", "__end__"]
]:
    author = state['email_input']['author']
    to = state['email_input']['to']
    subject = state['email_input']['subject']
    email_thread = state['email_input']['email_thread']

    namespace = (
        "email_assistant",
        config['configurable']['langgraph_user_id'],
        "examples"
    )
    examples = store.search(
        namespace,
        query=str({"email": state['email_input']})
    )
    examples=format_few_shot_examples(examples)

    langgraph_user_id = config['configurable']['langgraph_user_id']
    namespace = (langgraph_user_id, )

    result = store.get(namespace, "triage_ignore")
    if result is None:
        store.put(
            namespace,
            "triage_ignore",
            {"prompt": prompt_instructions["triage_rules"]["ignore"]}
        )
        ignore_prompt = prompt_instructions["triage_rules"]["ignore"]
    else:
        ignore_prompt = result.value['prompt']

    result = store.get(namespace, "triage_notify")
    if result is None:
        store.put(
            namespace,
            "triage_notify",
            {"prompt": prompt_instructions["triage_rules"]["notify"]}
        )
        notify_prompt = prompt_instructions["triage_rules"]["notify"]
    else:
        notify_prompt = result.value['prompt']

    result = store.get(namespace, "triage_respond")
    if result is None:
        store.put(
            namespace,
            "triage_respond",
            {"prompt": prompt_instructions["triage_rules"]["respond"]}
        )
        respond_prompt = prompt_instructions["triage_rules"]["respond"]
    else:
        respond_prompt = result.value['prompt']

    system_prompt = triage_system_prompt.format(
        full_name=profile["full_name"],
        name=profile["name"],
        user_profile_background=profile["user_profile_background"],
        triage_no=ignore_prompt,
        triage_notify=notify_prompt,
```

```
        triage_email=respond_prompt,
        examples=examples
    )
    user_prompt = triage_user_prompt.format(
        author=author,
        to=to,
        subject=subject,
        email_thread=email_thread
    )
    result = llm_router.invoke(
        [
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": user_prompt},
        ]
    )
    if result.classification == "respond":
        print("📧 Classification: RESPOND - This email requires a response"
        goto = "response_agent"
        update = {
            "messages": [
                {
                    "role": "user",
                    "content": f"Respond to the email {state['email_input']
                }
            ]
        }
    elif result.classification == "ignore":
        print("🚫 Classification: IGNORE - This email can be safely ignored
        update = None
        goto = END
    elif result.classification == "notify":
        # If real life, this would do something else
        print("🔔 Classification: NOTIFY - This email contains important in
        update = None
        goto = END
    else:
        raise ValueError(f"Invalid classification: {result.classification}"
    return Command(goto=goto, update=update)
```

Build the rest of our agent

In [ ]:
```
from langchain_core.tools import tool
```

In [ ]:
```
@tool
def write_email(to: str, subject: str, content: str) -> str:
    """Write and send an email."""
    # Placeholder response - in real app would send email
    return f"Email sent to {to} with subject '{subject}'"
```

In [ ]:
```
@tool
def schedule_meeting(
    attendees: list[str],
    subject: str,
    duration_minutes: int,
    preferred_day: str
) -> str:
    """Schedule a calendar meeting."""
    # Placeholder response - in real app would check calendar and schedule
    return f"Meeting '{subject}' scheduled for {preferred_day} with {len(atten
```

In [ ]:
```
@tool
def check_calendar_availability(day: str) -> str:
    """Check calendar availability for a given day."""
    # Placeholder response - in real app would check actual calendar
    return f"Available times on {day}: 9:00 AM, 2:00 PM, 4:00 PM"
```

In [ ]:
```
from langmem import create_manage_memory_tool, create_search_memory_tool
```

In [ ]:
```
manage_memory_tool = create_manage_memory_tool(
    namespace=(
        "email_assistant",
        "{langgraph_user_id}",
        "collection"
    )
)
search_memory_tool = create_search_memory_tool(
    namespace=(
        "email_assistant",
        "{langgraph_user_id}",
        "collection"
    )
)
```

In [ ]:
```
agent_system_prompt_memory = """
< Role >
You are {full_name}'s executive assistant. You are a top-notch executive assis
</ Role >

< Tools >
You have access to the following tools to help manage {name}'s communications

1. write_email(to, subject, content) - Send emails to specified recipients
2. schedule_meeting(attendees, subject, duration_minutes, preferred_day) - Sch
3. check_calendar_availability(day) - Check available time slots for a given d
4. manage_memory - Store any relevant information about contacts, actions, dis
5. search_memory - Search for any relevant information that may have been stor
</ Tools >

< Instructions >
{instructions}
</ Instructions >
"""
```

Updated create_prompt gets prompt from store

In [ ]:
```
def create_prompt(state, config, store):
    langgraph_user_id = config['configurable']['langgraph_user_id']
    namespace = (langgraph_user_id, )
    result = store.get(namespace, "agent_instructions")
    if result is None:
        store.put(
            namespace,
            "agent_instructions",
            {"prompt": prompt_instructions["agent_instructions"]}
        )
        prompt = prompt_instructions["agent_instructions"]
    else:
        prompt = result.value['prompt']

    return [
        {
            "role": "system",
            "content": agent_system_prompt_memory.format(
                instructions=prompt,
                **profile
            )
        }
    ] + state['messages']
```

Create the email agent

In [ ]:
```
from langgraph.prebuilt import create_react_agent
```

In [ ]:
```
tools= [
    write_email,
    schedule_meeting,
    check_calendar_availability,
    manage_memory_tool,
    search_memory_tool
]
response_agent = create_react_agent(
    "openai:gpt-4o",
    tools=tools,
    prompt=create_prompt,
    # Use this to ensure the store is passed to the agent
    store=store
)
```

In [ ]:
```
email_agent = StateGraph(State)
email_agent = email_agent.add_node(triage_router)
email_agent = email_agent.add_node("response_agent", response_agent)
email_agent = email_agent.add_edge(START, "triage_router")
email_agent = email_agent.compile(store=store)
```

Setup Agent to update Long Term Memory in the background

Your email_agent is now setup to pull its instructions from long-term memory.
Now, you'll create an agent to update that memory. First check current behavior.

In [ ]:
```
email_input = {
    "author": "Alice Jones <alice.jones@bar.com>",
    "to": "John Doe <john.doe@company.com>",
    "subject": "Quick question about API documentation",
    "email_thread": """Hi John,

Urgent issue - your service is down. Is there a reason why""",
}
```

In [ ]:
```
config = {"configurable": {"langgraph_user_id": "lance"}}
```

In [ ]:
```
response = email_agent.invoke(
    {"email_input": email_input},
    config=config
)
```

In [ ]:
```
for m in response["messages"]:
    m.pretty_print()
```

and look at current values of long term memory

In [ ]:
```
store.get(("lance",), "agent_instructions").value['prompt']
```

In [ ]:
```
store.get(("lance",), "triage_respond").value['prompt']
```

In [ ]:
```
store.get(("lance",), "triage_ignore").value['prompt']
```

In [ ]:
```
store.get(("lance",), "triage_notify").value['prompt']
```

Now, Use an LLM to update instructions.

In [ ]:
```
from langmem import create_multi_prompt_optimizer
```

In [ ]:
```
conversations = [
    (
        response['messages'],
        "Always sign your emails `John Doe`"
    )
]
```

In [ ]:
```
prompts = [
    {
        "name": "main_agent",
        "prompt": store.get(("lance",), "agent_instructions").value['prompt'],
        "update_instructions": "keep the instructions short and to the point",
        "when_to_update": "Update this prompt whenever there is feedback on ho

    },
    {
        "name": "triage-ignore",
        "prompt": store.get(("lance",), "triage_ignore").value['prompt'],
        "update_instructions": "keep the instructions short and to the point",
        "when_to_update": "Update this prompt whenever there is feedback on wh

    },
    {
        "name": "triage-notify",
        "prompt": store.get(("lance",), "triage_notify").value['prompt'],
        "update_instructions": "keep the instructions short and to the point",
        "when_to_update": "Update this prompt whenever there is feedback on wh

    },
    {
        "name": "triage-respond",
        "prompt": store.get(("lance",), "triage_respond").value['prompt'],
        "update_instructions": "keep the instructions short and to the point",
        "when_to_update": "Update this prompt whenever there is feedback on wh

    },
]
```

In [ ]:
```
optimizer = create_multi_prompt_optimizer(
    "anthropic:claude-3-5-sonnet-latest",
    kind="prompt_memory",
)
```

In [ ]:
```
updated = optimizer.invoke(
    {"trajectories": conversations, "prompts": prompts}
)
```

In [ ]:
```
print(updated)
```

In [ ]:
```
#json dumps is a bit easier to read
import json
print(json.dumps(updated, indent=4))
```

update the prompts in store.

> Note.. only one of the prompts was included here! The remainder are left to you!

In [ ]:
```
for i, updated_prompt in enumerate(updated):
    old_prompt = prompts[i]
    if updated_prompt['prompt'] != old_prompt['prompt']:
        name = old_prompt['name']
        print(f"updated {name}")
        if name == "main_agent":
            store.put(
                ("lance",),
                "agent_instructions",
                {"prompt":updated_prompt['prompt']}
            )
        else:
            #raise ValueError
            print(f"Encountered {name}, implement the remaining stores!")
```

In [ ]:
```
store.get(("lance",), "agent_instructions").value['prompt']
```

In [ ]:
```
response = email_agent.invoke(
    {"email_input": email_input},
    config=config
)
```

In [ ]:
```
for m in response["messages"]:
    m.pretty_print()
```

In [ ]:
```
email_input = {
    "author": "Alice Jones <alice.jones@bar.com>",
    "to": "John Doe <john.doe@company.com>",
    "subject": "Quick question about API documentation",
    "email_thread": """Hi John,

Urgent issue - your service is down. Is there a reason why""",
}
```

In [ ]:
```
response = email_agent.invoke(
    {"email_input": email_input},
    config=config
)
```

In [ ]:
```
conversations = [
    (
        response['messages'],
        "Ignore any emails from Alice Jones"
    )
]
```

In [ ]:
```
prompts = [
    {
        "name": "main_agent",
        "prompt": store.get(("lance",), "agent_instructions").value['prompt'],
        "update_instructions": "keep the instructions short and to the point",
        "when_to_update": "Update this prompt whenever there is feedback on ho

    },
    {
        "name": "triage-ignore",
        "prompt": store.get(("lance",), "triage_ignore").value['prompt'],
        "update_instructions": "keep the instructions short and to the point",
        "when_to_update": "Update this prompt whenever there is feedback on wh

    },
    {
        "name": "triage-notify",
        "prompt": store.get(("lance",), "triage_notify").value['prompt'],
        "update_instructions": "keep the instructions short and to the point",
        "when_to_update": "Update this prompt whenever there is feedback on wh

    },
    {
        "name": "triage-respond",
        "prompt": store.get(("lance",), "triage_respond").value['prompt'],
        "update_instructions": "keep the instructions short and to the point",
        "when_to_update": "Update this prompt whenever there is feedback on wh

    },
]
```

In [ ]:
```
updated = optimizer.invoke(
    {"trajectories": conversations, "prompts": prompts}
)
```

In [ ]:
```
for i, updated_prompt in enumerate(updated):
    old_prompt = prompts[i]
    if updated_prompt['prompt'] != old_prompt['prompt']:
        name = old_prompt['name']
        print(f"updated {name}")
        if name == "main_agent":
            store.put(
                ("lance",),
                "agent_instructions",
                {"prompt":updated_prompt['prompt']}
            )
        if name == "triage-ignore":
            store.put(
                ("lance",),
                "triage_ignore",
                {"prompt":updated_prompt['prompt']}
            )
        else:
            #raise ValueError
            print(f"Encountered {name}, implement the remaining stores!")
```

In [ ]:
```
response = email_agent.invoke(
    {"email_input": email_input},
    config=config
)
```

In [ ]:
```
store.get(("lance",), "triage_ignore").value['prompt']
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: