RELATÓRIO TÉCNICO - (UNIDADE 4)

Disciplina: Inteligência Artificial - Turma 01 (2025.2)

Professor: Hendrik Macedo

Instituição: Universidade Federal de Sergipe (UFS) / Departamento de Computação (DCOMP)

Integrantes:

Ellen Karolliny dos Santos

Ellen Vitória Menezes Lima

João Santos Rocha

Larissa Batista dos Santos

Tasso Marcel de Oliveira

1. Definição do Problema

O problema abordado foi a manutenção do histórico de conversa em implementações de chat puro, sem utilização de banco de dados. Em vários sistemas conversacionais baseados em Modelos de Linguagem de Grande Escala (LLMs), o histórico de conversa não é armazenado por padrão. O modelo não “lembra” interações anteriores de forma automática, apenas formula uma resposta com base na pergunta atual. Sendo assim, o desafio do projeto foi preservar o contexto da conversa para garantir coerência entre perguntas e respostas.

2. Solução do Problema

Foi implementado um mecanismo de memória a nível de aplicação, utilizando:

- Estrutura de dados do tipo lista *(history)*
- Identificação de papéis *(system, user, assistant)*
- Reconstrução manual do prompt a cada requisição
- Função de truncamento para controle de tamanho

A cada nova pergunta do usuário, ela é adicionada ao histórico. Após isso, o histórico é convertido em texto sequencial e o prompt completo é enviado ao modelo. Por fim, a resposta é adicionada ao histórico.

O modelo de linguagem utilizado possui a capacidade de compreender o contexto acumulado, manter coerência semântica, continuar raciocínios anteriores e gerar respostas de acordo com o contexto.

Inicialmente, foi utilizada a versão gratuita da API do Gemini para a realização de testes. Entretanto, o limite gratuito foi atingido rapidamente. Então migramos para o Ollama, escolhemos o **Orca Mini** por ser um modelo leve e ainda assim capaz de manter boa qualidade de tarefas. Mas ao implementar a lógica foi verificada uma demora para se obter resposta por parte da LLM.

Então migramos novamente para a API Gemini, agora no primeiro nível da versão paga, onde obtivemos respostas mais rápidas e com um limite maior de requisições.

3. Correspondência com o pseudocódigo:

3.1. A função do agente conversacional:

function CONVERSATIONAL-AGENT(*user_input*, *history*, *config*) **returns** a *response*, *updated_history*

return *reply_text*, *updated_history*

```python
# Agente Conversacional
def CONVERSATIONAL_AGENT(user_input, history, config):
    history.append({"role": "user", "content": user_input})
    payload = TRUNCATE_HISTORY(history, config["max_tokens"])

    api_response = CALL_LLM_API(config, payload)

    reply_text = api_response
    history.append({"role": "assistant", "content": reply_text})

    return reply_text, history
```

Essa função recebe a pergunta do usuário, o histórico da conversa e as configurações do modelo. Ela retorna a resposta da pergunta e atualiza o histórico de conversa.

3.2. Estrutura de configuração (hiperparâmetros do modelo):

```python
client = genai.Client(api_key=api_key)

#Configuração do Modelo
MODEL_CONFIG = {
    "model_name": "gemini-2.5-flash",
    "temperature": 0.7,
    "max_tokens": 5000
}

# Gestão de Contexto
```

```
structure MODEL-CONFIG
    provider string (e.g., "openai", "local-llama")
    model_name string (e.g., "gpt-4o", "dall-e-3", "whisper-1")
    temperature float [0.0, 2.0] // Criatividade
    max_tokens integer
    modality string { "text", "image", "audio" }
```

3.3. Construção da Mensagem

```python
# Parte visual
st.title("Chatbot Gemini")
st.caption("Modelo: gemini-2.5-flash")

if "history" not in st.session_state:
    st.session_state.history = [
        {"role": "system", "content": "Você é um progr
    ]

for msg in st.session_state.history:
    if msg["role"] == "user":
        with st.chat_message("user"):
            st.markdown(msg["content"])
    elif msg["role"] == "assistant":
        with st.chat_message("assistant"):
            st.markdown(msg["content"])

user_input = st.chat_input("Digite uma pergunta ... ")

if user_input:
    with st.chat_message("user"):
        st.markdown(user_input)
```

```
2. Cenário A: Texto (Chatbot com memória)

Este pseudocódigo foca na manutenção do histórico de conversa, que
implementações de chat puro sem banco de dados.

function CONVERSATIONAL-AGENT(user_input, history, config) returns

    // 1. Construção da Mensagem
    current_msg CREATE-MESSAGE(role="user", content=user_input)
    APPEND(history, current_msg)

// 2. Gestão de Contexto (Obrigatório para não estourar tokens)
    // Mantém o System Prompt fixo e remove mensagens antigas se nece
    payload TRUNCATE-HISTORY(history, limit=config.max_tokens)
```

3.4. Gestão de contexto (função que faz a gestão do contexto para não estourar o limite de tokens):

```python
client = genai.Client(api_key=api_key)

#Configuração do Modelo
MODEL_CONFIG = {
    "model_name": "gemini-2.5-flash",
    "temperature": 0.7,
    "max_tokens": 5000
}

# Gestão de Contexto
def TRUNCATE_HISTORY(history, limit):
    if len(history) > limit:
        # Mantém o System Prompt fixo e remo
        history.pop(1)
    return history
```

```
implementações de chat puro sem banco de dados.

function CONVERSATIONAL-AGENT(user_input, history, config) returns
    // 1. Construção da Mensagem
    current_msg CREATE-MESSAGE(role="user", content=user_input)
    APPEND(history, current_msg)

// 2. Gestão de Contexto (Obrigatório para não estourar tokens)
    // Mantém o System Prompt fixo e remove mensagens antigas se nece
    payload TRUNCATE-HISTORY(history, limit=config.max_tokens)

// 3. Chamada de Inferência (Texto)
    api_response CALL-LLM-API(config, payload)
```

3.5. Chamada de Inferência

```python
def CALL_LLM_API(config, payload):
    # Constrói o prompt a partir do histórico
    contents = []
    for msg in payload:
        if msg["role"] == "user":
            contents.append({
                "role": "user",
                "parts": [{"text": msg["content"]}]
            })
        elif msg["role"] == "assistant":
            contents.append({
                "role": "model",
                "parts": [{"text": msg["content"]}]
            })

    response = client.models.generate_content(
        model=config["model_name"],
        contents=contents,
        config={
            "temperature": config["temperature"],
            "max_output_tokens": config["max_tokens"]
        }
    )

    return response.text
```

2. Cenário A: Texto (Chatbot com memória)

Este pseudocódigo foca na manutenção do **histórico de conversa**, que é o implementações de chat puro sem banco de dados.

function CONVERSATIONAL-AGENT(*user_input*, *history*, *config*) **returns** a re

// 1. Construção da Mensagem
current_msg CREATE-MESSAGE(role="user", content=*user_input*)
APPEND(*history*, *current_msg*)

// 2. Gestão de Contexto (Obrigatório para não estourar tokens)
// Mantém o System Prompt fixo e remove mensagens antigas se necessá
payload TRUNCATE-HISTORY(*history*, limit=*config*.max_tokens)

// 3. Chamada de Inferência (Texto)
api_response CALL-LLM-API(*config*, *payload*)

3.6. Pós processamento (envio da pergunta do usuário ao chat)

```python
user_input = st.chat_input("Digite uma pergunta ... ")

if user_input:
    with st.chat_message("user"):
        st.markdown(user_input)

    with st.spinner("Pensando ... "):
        try:
            resposta, st.session_state.history = CONVERSATIONAL_AGENT(
                user_input,
                st.session_state.history,
                MODEL_CONFIG
            )

            with st.chat_message("assistant"):
                st.markdown(resposta)

        except Exception as e:
            st.error(f"Erro de Conexão: {e}")
```

// 4. Pós-processamento
reply_text EXTRACT-CONTENT(*api_response*)
reply_msg CREATE-MESSAGE(role="assistant", content=*reply_text*)
APPEND(*updated_history*, *reply_msg*)