1 Objetivo

Desenvolver uma ferramenta de inteligência artificial (IA) generativa capaz de recuperar informações de um ou mais arquivos em formato PDF, utilizando a capacidade dos grandes modelos de linguagem (LLMs) para gerar textos semelhantes à linguagem humana, com base no conteúdo específico dos PDFs fornecidos.

2 Introdução

Neste tutorial, vamos explorar e implementar soluções de IA generativa, utilizando técnicas avançadas. Ao final, você será capaz de:

- **Compreender** o que são Grandes Modelos de Linguagem (LLM) e suas aplicações práticas.
- **Entender** o funcionamento do Retrieval-Augmented Generation (RAG).
- **Aplicar** os conceitos aprendidos para implementar um sistema de RAG usando R e um pouco de Python.
- **Criar** um chatbot capaz de interagir com o conteúdo de um arquivo PDF.

Atualmente, até onde eu sei, não há implementação de LLMs na linguagem R. Entretanto, é possível integrá-los com Python através do pacote `reticulate`, como veremos no tutorial.

Todo trecho de código apresentado aqui terá uma marcação no título indicando se é R (cor azul) ou Python (cor amarela).

2.1 Conceitos Básicos

Esses são os conceitos fundamentais para o entendimento do tutorial:

- **Grandes Modelos de Linguagem (LLM)**: Modelos de inteligência artificial treinados em grandes volumes de texto para entender e gerar linguagem natural de forma semelhante aos humanos. Exemplos incluem GPT-3 e GPT-4.

- **ChatGPT**: Um chatbot avançado baseado no modelo de linguagem GPT da OpenAI, capaz de realizar conversas complexas, responder perguntas e executar tarefas de linguagem natural.

- **Prompt**: Uma instrução ou conjunto de instruções fornecidas a um modelo de linguagem para orientar a geração de respostas ou execução de tarefas específicas.

- **Embedding**: Representação numérica de dados, como palavras ou frases, em um espaço vetorial multidimensional, permitindo que modelos de IA compreendam relações semânticas entre eles.

- **RAG**: Retrieval-Augmented Generation, uma técnica que combina recuperação de informações e geração de texto, onde um modelo de linguagem é usado para gerar respostas com base em documentos recuperados de uma base de dados.

- **API**: Interface de Programação de Aplicações, um conjunto de definições e protocolos que permite a integração entre diferentes softwares, permitindo que eles se comuniquem e compartilhem funcionalidades.
- **Langchain**: Um framework de desenvolvimento que facilita a criação de aplicações que utilizam inteligência artificial generativa, especialmente modelos de linguagem. O Langchain oferece componentes modulares para integrar LLMs, como aqueles da OpenAI, com diversas fontes de dados e ferramentas de processamento, permitindo uma implementação eficiente de fluxos de trabalho de IA complexos, incluindo o uso de embeddings e a geração de texto baseada em contexto.

2.2 Requisitos

Os requisitos necessários para que você consiga reproduzir o código apresentado neste tutorial são:

- **R e RStudio Instalados**: Para instalação, faça o download do R em http://www.r-project.org. Em seguida, instale a IDE (Integrated Development Environment) R Studio.
- **Python Instalado**: Caso não tenha o Python instalado na sua máquina, você pode instalá-lo usando o RStudio por meio do pacote `reticulate` usando o comando `reticulate::install_miniconda()`. Detalhes podem ser encontrados aqui.
- **Conta OpenAI**: para usar a API do ChatGPT, você deve se cadastrar. Além disso você deve adicionar créditos para conseguir utilizar a API da OpenAI. Entre nesta página e adicione. Sugiro adicionar US$ 1, que será mais do que suficiente para execução deste tutorial.
- **Conta Pinecone**: para usar a API da Pinecone, que é uma plataforma para armazenamento de dados vetoriais (onde vamos armazenar os dados do PDF). Crie sua conta usando este link. A conta gratuita permite armazenar até 2GB de dados.

Para configurar o Python dentro do RStudio, siga os passos abaixo:

1. Abra o RStudio.
2. Vá para o menu `Tools` e selecione `Global Options`.
3. No painel esquerdo, selecione `Python`.
4. Clique no botão `Select...`.
5. Navegue até o executável do Python que você deseja usar e selecione-o. Este caminho é geralmente algo como `C:\Users\SeuUsuario\AppData\Local\Programs\Python\PythonXX\python.exe` no Windows, ou `/usr/bin/python3` no Linux/Mac.
6. Clique em `Apply` e depois em `OK` para salvar as configurações.

3 Conexão com a API da OpenAI

Na área de API Keys, crie uma nova chave usando o botão *"Create new secret key"*. Salve essa chave, que será usada neste tutorial.

Use o código abaixo para salvar a chave como uma variável de ambiente do RStudio.

```
Sys.setenv(`OPENAI_API_KEY`= "COLE SUA CHAVE AQUI")
```

4 Conexão com a API da Pinecone

Pinecone é uma plataforma para armazenamento de dados vetoriais, permitindo a criação de "*vector databases*". Esses bancos de dados são projetados para armazenar e gerenciar vetores de alta dimensão, como aqueles gerados por embeddings em aplicações de aprendizado de máquina e inteligência artificial. Essa plataforma facilita a busca e a recuperação de informações com base em similaridade vetorial, que é o que vamos fazer para recuperar de informações e implementar um chat que possa responder perguntas específicas.

Na página inicial da sua conta Pinecone, localize no menu do lado esquedo a opção de "API keys", abra e clique no botão "+ Create API key". Salve essa chave, que será usada neste tutorial.

Use o código abaixo para salvar a chave como uma variável de ambiente do RStudio.

```
Sys.setenv(`PINECONE_API_KEY`= "COLE SUA CHAVE AQUI")
```

5 Configuração de ambiente para rodar Python no RStudio

O pacote `reticulate` do R facilita a integração entre R e Python, permitindo que pessoas programadoras de R utilizem funcionalidades avançadas disponíveis em Python diretamente no RStudio. Isso é particularmente útil para trabalhar com grandes modelos linguagens (LLM), que possuem implementações mais robustas e atualizadas em Python.

Embora `reticulate` torne possível essa integração, é importante ter um conhecimento básico da sintaxe de Python para aproveitar ao máximo as capacidades dos modelos de IA generativa, como o ChatGPT, em suas análises.

No código abaixo, vamos criar um ambiente virtual Python chamado `"langchain_rag_pdf"` e instalar os pacotes necessários para a nossa análise usando o `reticulate`. Um ambiente no Python é um espaço isolado onde você pode instalar pacotes e dependências específicas para um projeto, sem interferir em outros projetos. Isso garante que cada projeto possa ter suas próprias versões de pacotes, evitando conflitos e problemas de compatibilidade. No R, normalmente não é necessário criar ambientes separados, pois o gerenciamento de pacotes é feito de forma global. No entanto, a criação de ambientes no Python oferece vantagens significativas, como isolamento, reprodutibilidade e melhor gestão de dependências.

```
library(reticulate)

virtualenv_create(
  envname = "langchain_rag_pdf",
  packages = c(
    "langchain",
    "langchain-community",
    "pypdf",
    "pinecone",
    "langchain_pinecone",
    "langchain-openai",
    "pinecone-client[grpc]"
  )
)

reticulate::use_virtualenv("langchain_rag_pdf")
```

Como exemplo de funcionamento da integração entre R e Python usando o pacote reticulate, podemos usar o código abaixo:

```
reticulate::py_run_string('
print("Estou no SINAPE em Fortaleza!")
')
```

Para finalizar a configuração, vamos definir duas variáveis de ambiente com as chaves da API da OpenAI e da Pinecone que foram geradas anteriormente. Altere o código abaixo conforme necessário, substituindo “COLE SUA CHAVE AQUI” pela sua chave de API correspondente.

```
chave_api_openai <- Sys.getenv("OPENAI_API_KEY")
chave_api_pinecone <- Sys.getenv("PINECONE_API_KEY")
```

Com isso, as chaves das APIs estarão configuradas corretamente e você estará pronto para utilizar a OpenAI API e a Pinecone API no seu ambiente de desenvolvimento.

6 Download e importação do arquivo em PDF

Vamos usar como arquivo de exemplo a documentação do pacote ggplot2, que é amplamente utilizado para criar gráficos no R usando a gramática de gráficos.

Baixe o arquivo PDF que será utilizado no tutorial usando o código abaixo.

```
if(!(dir.exists("docs"))) {
  dir.create("docs")
}

download.file("https://cran.r-
project.org/web/packages/ggplot2/ggplot2.pdf",
              destfile = "docs/ggplot2.pdf", mode = "wb")
```

Importe o arquivo como um objeto no LangChain e carregue seu conteúdo para processamento.

> **Atenção**
>
> Atenção! o código abaixo está em Python.

```
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader('codigos_novo/docs/ggplot2.pdf')

paginas = loader.load()
print(type(paginas))
print(len(paginas))
```

Você pode examinar o objeto `paginas` do Python no R usando o objeto `py` do `reticulate`. O código R a seguir armazena o objeto Python paginas em uma variável R chamada `paginas_em_r`. Você pode então trabalhar com o objeto como qualquer outro objeto R. Neste caso, é uma lista.

```
paginas_em_r <- py$paginas

# metadatos da primeira pagina
paginas_em_r[[1]]$metadata

# quantidade de caracteres da centésima pagina
nchar(paginas_em_r[[100]]$page_content)
```

7 Divisão do documento em pedaços

O próximo passo é dividir o arquivo PDF em partes menores para facilitar o processamento pelos modelos de IA generativa.
O trecho de código abaixo utiliza o `CharacterTextSplitter` do `LangChain` para realizar essa tarefa. Nele definimos o `tamanho_pedaco` que especifica o tamanho máximo de cada parte em 4000 caracteres, e o `tamanho_intersecao`, que determina uma sobreposição de 150 caracteres entre as partes consecutivas. Após a divisão, as partes são armazenadas na variável `partes_pdf`, que contém o texto dividido. O código finaliza imprimindo o número de partes geradas.

```
from langchain.text_splitter import CharacterTextSplitter

tamanho_pedaco = 4000
tamanho_intersecao = 150

divisor_documentos = CharacterTextSplitter(
  chunk_size = tamanho_pedaco,
  chunk_overlap = tamanho_intersecao,
  separator = " "
)
```

```
partes_pdf = divisor_documentos.split_documents(paginas)

print(len(partes_pdf))
partes_pdf
```

8 Avaliação do custo da aplicação

Vamos calcular o custo da aplicação com base no número de tokens gerados. O pacote `TheOpenAIR` do R possui uma função `count_tokens()` (certifique-se de instalar tanto este pacote quanto o `purrr` para usar o código R abaixo):

```
partes_pdf <- py$partes_pdf
length(partes_pdf)

total_tokens <- purrr::map_int(partes_pdf,
                               ~
TheOpenAIR::count_tokens(.x$page_content)) |>
  sum()
```

Ao executar o código acima, veremos que há 153.172 tokens.

Atualmente (agosto de 2024), o custo do modelo **ada v2** usado para criação dos *embeddings* da OpenAI é de US$ 0,10 por 1 milhão de tokens. Como temos 153.172 tokens, o custo dessa etapa será de aproximadamente US$ 0,0153172.

9 Geração de *embeddings*

Agora vamos criar embeddings a partir das partes do PDF utilizando a API da OpenAI e o serviço Pinecone para armazenamento e busca de vetores.

Primeiro, configuramos as chaves de API para acessar os serviços necessários:

```
import os
os.environ["OPENAI_API_KEY"] = r.chave_api_openai
os.environ['PINECONE_API_KEY'] = r.chave_api_pinecone
```

> **Dica**
>
> Veja que no código python acima usamos o objeto `r` para acessar as variáveis R que armazanam as chaves das apis.

Vamos utilziar o modelo “text-embedding-ada-002” da OpenAI para gerar embeddings a partir do conteúdo do PDF:

```
from langchain_openai import OpenAIEmbeddings
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

resultado = embeddings.embed_query(partes_pdf[100].page_content)
print(resultado)
```

Este último comando acima imprime o vetor de embeddings gerado para a parte específica do PDF. Cada número no vetor representa uma dimensão no espaço de embeddings, refletindo a semântica do texto. Esses vetores são usados para comparar a similaridade entre diferentes textos com base em suas representações vetoriais.

Em seguida, configuramos o Pinecone, o serviço de banco de dados vetorial que vamos usar para armazenar os embeddings gerados. Aqui, criamos um índice específico para nossos dados chamado `"tutorial-sinape"`:

```
from pinecone.grpc import PineconeGRPC as Pinecone
from pinecone import ServerlessSpec

pc = Pinecone(os.environ['PINECONE_API_KEY'])

nome_indice = "tutorial-sinape"

if nome_indice not in pc.list_indexes().names():
    pc.create_index(
        name = nome_indice,
        dimension = 1536,
        metric = "cosine",
        spec = ServerlessSpec(
            cloud = 'aws',
            region = 'us-east-1'
        )
    )
```

O valor definido da dimensão de `1536` para o índice de vetores é o requerido pelo modelo de *embedding* da OpenAI.

A próxima etapa é enviar os *embeddings* para o Pinecone:

```
from langchain_pinecone import PineconeVectorStore

vectorstore_from_docs = PineconeVectorStore.from_documents(
    partes_pdf,
    index_name = nome_indice,
    embedding = embeddings
)
```

Agora, podemos usar diretamente o objeto armezanado na variável `vectorstore_from_docs` ou então, carregar o vetor que está armazenado na Pinecone.

```
base_conhecimento = PineconeVectorStore.from_existing_index(
    index_name=nome_indice,
    embedding=embeddings
)
```

Finalmente, testamos a busca por similaridade com uma pergunta de exemplo, recuperando as partes do documento mais relevantes:

```
pergunta = "Como rotacionar o texto no eixo x de um gráfico ggplot?"
resultado = base_conhecimento.similarity_search(pergunta, k = 3)
resultado[0]
resultado[1]
resultado[2]
```

```
Document(metadata={'page': 135.0, 'source':
'codigos_novo/docs/ggplot2.pdf'}, page_content='136
geom_map\nannotate(\n"text", label = "plot mpg vs. wt",\nx = 2, y = 15,
size = 8, colour = "red"\n)\n# Aligning labels and bars ----------------
---------------------------------\ndf <- data.frame(\nx = factor(c(1, 1,
2, 2)),\ny = c(1, 3, 2, 1),\ngrp = c("a", "b", "a", "b")\n)\n# ggplot2
doesn /quotesingle.Vart know you want to give the labels the same virtual
width\n# as the bars:\nggplot(data = df, aes(x, y, group = grp))
+\ngeom_col(aes(fill = grp), position = "dodge") +\ngeom_text(aes(label =
y), position = "dodge")\n# So tell it:\nggplot(data = df, aes(x, y, group
= grp)) +\ngeom_col(aes(fill = grp), position = "dodge")
+\ngeom_text(aes(label = y), position = position_dodge(0.9))\n# You can
/quotesingle.Vart nudge and dodge text, so instead adjust the y
position\nggplot(data = df, aes(x, y, group = grp)) +\ngeom_col(aes(fill
= grp), position = "dodge") +\ngeom_text(\naes(label = y, y = y +
0.05),\nposition = position_dodge(0.9),\nvjust = 0\n)\n# To place text in
the middle of each bar in a stacked barplot, you\n# need to set the vjust
parameter of position_stack()\nggplot(data = df, aes(x, y, group = grp))
+\ngeom_col(aes(fill = grp)) +\ngeom_text(aes(label = y), position =
position_stack(vjust = 0.5))\n# Justification ----------------------------
---------------------------------\ndf <- data.frame(\nx = c(1, 1, 2, 2,
1.5),\ny = c(1, 2, 1, 2, 1.5),\ntext = c("bottom-left", "top-left",
"bottom-right", "top-right", "center")\n)\nggplot(df, aes(x, y))
+\ngeom_text(aes(label = text))\nggplot(df, aes(x, y))
+\ngeom_text(aes(label = text), vjust = "inward", hjust =
"inward")\ngeom_map Polygons from a reference map')
```

O resultado acima mostra o `resultado[0]` a busca por similaridade vetorial com base na `pergunta`, ou seja, encontra o conteúdo no PDF que mais se assemelha com a pergunta feita.

10 Integração com ChatGPT para Geração de Respostas

Agora, integramos o sistema com o ChatGPT para gerar respostas às consultas baseadas no conteúdo armazenado. É importante notar que, nesta etapa, as consultas *ainda* não possuem memória de contexto, ou seja, cada pergunta é tratada de forma independente, sem considerar interações anteriores como em uma conversa contínua de chat.

Primeiro, configuramos o modelo de linguagem (LLM). Aqui vamos usar o modelo `"gpt-4o-mini"`, que é o mais recente da OpenAI com o melhor custo-benefício.

```
from langchain.chains import RetrievalQA
from langchain_openai import ChatOpenAI
```

```
from langchain_openai import OpenAIEmbeddings
from langchain_pinecone import PineconeVectorStore

llm = ChatOpenAI(
    openai_api_key = os.environ["OPENAI_API_KEY"],
    model_name = "gpt-4o-mini",
    temperature = 0.0
)
```

Em seguida, carregamos o repositório de conhecimento armazenado no Pinecone:

```
nome_indice = "tutorial-sinape"
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")
base_conhecimento = PineconeVectorStore.from_existing_index(
    index_name = nome_indice,
    embedding = embeddings
)
```

Utilizamos a cadeia de perguntas e respostas `RetrievalQA` para configurar como as perguntas serão tratadas:

```
qa = RetrievalQA.from_chain_type(
    llm = llm,
    chain_type = "stuff",
    retriever = base_conhecimento.as_retriever()
)
```

Por fim, fazemos uma pergunta ao sistema e obtemos a resposta gerada com base nos documentos armazenados:

```
pergunta = "Como fazer gráfico de linha no ggplot?"
qa.run(pergunta)
```

E obtemos como resposta:

```
Para fazer um gráfico de linha no ggplot, você pode usar a função
`geom_line()`.
Aqui está um exemplo básico:

  ` ` `
  library(ggplot2)
  # Suponha que você tenha um dataframe chamado 'dados' com colunas
'data' e 'valor'
  ggplot(dados, aes(x = data, y = valor)) +
  geom_line()
  ` ` `


Se você tiver múltiplas linhas para diferentes grupos,
pode usar a estética `group` para diferenciá-las:\n\n
  ` ` `
  ggplot(dados, aes(x = data, y = valor, group = grupo)) +
```

```
  geom_line()
  ```

Certifique-se de substituir `dados`, `data`, `valor` e `grupo` pelos
nomes reais das suas colunas e dataframe."
```

Neste exemplo, a resposta será gerada com base no conteúdo relacionado do PDF, mas sem considerar qualquer contexto de perguntas ou respostas anteriores. Se você fizer qualquer outra pergunta cujo assunto não esteja relacionado ao pacote ggplot2, deverá receber como resposta algo como “Está fora de contexto”.

## 11 Criação de um Chat com Histórico de Conversa

Nesta seção final, vamos configurar um sistema de chat que pode considerar o histórico de conversas, permitindo que o assistente forneça respostas mais contextualmente relevantes. Isso é feito através de uma cadeia de recuperação e resposta que utiliza tanto o contexto dos documentos recuperados quanto o histórico de interação da pessoa utilizando o serviço.

Primeiro, importamos as bibliotecas necessárias e configuramos o modelo de linguagem e o recuperador:

```
from langchain_core.prompts import ChatPromptTemplate,
MessagesPlaceholder
from langchain.chains import create_retrieval_chain
from langchain.chains.combine_documents import
create_stuff_documents_chain
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_core.chat_history import BaseChatMessageHistory
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.prompts import ChatPromptTemplate,
MessagesPlaceholder
from langchain.chains import create_history_aware_retriever

llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)
recuperador = base_conhecimento.as_retriever()
```

Definimos um prompt para contextualizar as perguntas e outro para formular as respostas:

```
prompt_contextualizacao = (
  "Dado um histórico de chat e a última pergunta do usuário "
  "que pode referenciar o contexto no histórico do chat, "
  "formule uma pergunta independente que possa ser compreendida "
  "sem o histórico do chat. NÃO responda à pergunta, "
  "apenas reformule-a se necessário e, caso contrário, retorne-a como
está."
)

prompt_template = ChatPromptTemplate.from_messages(
```
```

```
  [
    ("system", prompt_contextualizacao),
    MessagesPlaceholder("chat_history"),
    ("human", "{input}"),
  ]
)

recuperador_historico = create_history_aware_retriever(
    llm, recuperador, prompt_template
)

prompt_final = (
  "Você é um assistente para tarefas de perguntas e respostas. "
  "Use os seguintes trechos de contexto recuperado para responder "
  "à pergunta. Se você não souber a resposta, diga que você "
  "não sabe. Se a pergunta estiver fora do contexto recuperado, "
  "não responda e apenas diga que está fora do contexto."
  "A sua resposta deve ser em português brasileiro.\n\n"
  "{context}"
)

qa_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", prompt_final),
        MessagesPlaceholder("chat_history"),
        ("human", "{input}"),
    ]
)
```

Criamos as cadeias para gerenciar a recuperação de documentos e a geração de respostas:

```
cadeia_perguntas_e_respostas = create_stuff_documents_chain(
  llm,
  qa_prompt
)

cadeia_rag = create_retrieval_chain(
  recuperador_historico,
  cadeia_perguntas_e_respostas
)
```

Implementamos uma função para gerenciar o histórico de mensagens de cada sessão de chat:

```
sessoes = {}

def obter_historico_sessao(id_sessao: str) -> BaseChatMessageHistory:
  if id_sessao not in sessoes:
```

```
    sessoes[id_sessao] = ChatMessageHistory()
  return sessoes[id_sessao]
```

Finalmente, integramos o histórico de conversas com a cadeia de respostas para manter o contexto:

```
chat_cadeia_rag = RunnableWithMessageHistory(
    cadeia_rag,
    obter_historico_sessao,
    input_messages_key = "input",
    history_messages_key = "chat_history",
    output_messages_key = "answer",
)

def obter_resposta(pergunta, id_sessao = "abc123"):
  resultado = chat_cadeia_rag.invoke(
    {"input": pergunta},
    config = {"configurable": {"session_id": id_sessao}},
  )
  return resultado
```

Com este sistema, o assistente pode considerar o histórico de conversas ao fornecer respostas, criando uma experiência de chat mais natural e contextualmente informada.

Para utilizar o chat com o sistema configurado, vamos integrar o R com Python usando o pacote `reticulate`. Abaixo está o código para iniciar o chat em R:

```
library(reticulate)
resposta <- py$obter_resposta("Qual a capital do Ceará?")
print(resposta$answer)

iniciar_chat <- function(id_sessao = "abc123") {
  print("Iniciando chatGGPLOT, digite 'sair' para terminar.")

  while(TRUE){
    pergunta <- readline("Diga: ")

    if (stringr::str_to_lower(pergunta) == 'sair'){
      print("Encerrando o chatGGPLOT.")
      break
    }

    resultado <- py$obter_resposta(pergunta, id_sessao)
    print(resultado$answer)
  }
}

iniciar_chat()
```

Esse script em R permite iniciar uma sessão de chat interativa, onde os usuários podem fazer perguntas e receber respostas contextuais, mantendo um histórico de conversa.

12 Usando IA generativa brasileira

MaritacaAI é uma plataforma de inteligência artificial desenvolvida no Brasil, focada em fornecer soluções personalizadas de processamento de linguagem natural e aprendizado de máquina. Ela é projetada para atender às necessidades específicas do mercado brasileiro, oferecendo suporte completo ao português brasileiro e incorporando nuances culturais e linguísticas que são únicas para a região.

Para criar uma conta, acesse a plataforma aqui. Você ganhará R$20 de créditos para fazer seus testes. Atualmente, o modelo mais avançado, o Sabiá-3, custa R$10 por milhão de tokens. Antes de usar, entretanto, você deve cadastrar um cartão de crédito para usos futuros. Após feito isso, acesse a área de API e clique em "Crie Nova Chave".

Com a chave em mãos e usando o `langchain`, fica muito fácil alterar o modelo de LLM usado em nossa aplicação. Basta alterar a parte do código em que criamos a variável `llm`, como mostramos abaixo:

```
chave_api_maritaca = r.chave_api_maritaca

llm = ChatMaritalk(
    model = "sabia-2-small",
    api_key = chave_api_maritaca,
    temperature = 0.1,
    max_tokens = 2000,
)
```

O resto do código continua exatamente o mesmo. Aqui escolhemos trabalhar com o modelo `"sabia-2-small"`, o que apresenta o melhor custo-benefício, custando R$1 por milhão de tokens de entrada e R$3 por milhão de token de saída.

13 Conclusão

Neste tutorial, exploramos a criação de uma ferramenta de inteligência artificial generativa capaz de interagir com documentos PDF para recuperar informações específicas e responder a perguntas. Ao longo do processo, aprendemos a integrar modelos de linguagem avançados, como os da OpenAI, com o ambiente R utilizando o pacote reticulate. Implementamos uma solução que busca e extrai informações relevantes de documentos, e também mantém um histórico de interação para melhorar a contextualização das respostas em conversas subsequentes.

Este tutorial serve como um ponto de partida para quem deseja aplicar IA generativa em cenários reais, especialmente na automação de respostas e análise de documentos textuais.

14 Referências

- Building a Portuguese Language RAG Pipeline using Sabia-7B, Qdrant, and LangChain;
- Call ChatGPT (or really any other API) from R;
- Curso LangChain Chat with Your Data - DeepLearning.AI Short;
- Documentação LangChain Maritalk;
- Documentação LangChain Memory;
- Documentação Plataforma OpenAI;
- Documentação Plataforma Pinecone.
- O que é a geração aumentada de recuperação?