Plano de Projeto: CLI de Estudo Interativo de Livros/PDFs com IA Local

Introdução

Este documento detalha o plano de projeto para o desenvolvimento de uma Interface de Linha de Comando (CLI) inovadora, projetada para facilitar o estudo interativo de livros e documentos em formato PDF ou TXT. A ferramenta proposta integrará capacidades de Inteligência Artificial (IA) local, utilizando o framework Ollama para processamento de modelos de linguagem grandes (LLMs) diretamente no ambiente do usuário. O objetivo principal é oferecer uma experiência de estudo enriquecida, permitindo a criação de resumos estruturados, a geração de quizzes personalizados e a capacidade de realizar perguntas contextuais sobre o conteúdo do material de estudo. Este projeto será desenvolvido por uma equipe de três programadores ao longo de um período de três meses, com um foco em boas práticas de desenvolvimento Python e modularidade. A seguir, serão apresentados os detalhes técnicos, a arquitetura, o fluxo de trabalho, as tecnologias empregadas e um cronograma de sprints para guiar o desenvolvimento.

Visão Geral do Projeto

O `study-cli` é uma ferramenta de linha de comando que visa transformar a maneira como os usuários interagem com seus materiais de estudo digitais. Ao invés de apenas ler passivamente, o usuário poderá carregar PDFs ou arquivos de texto, e a CLI se encarregará de processar esse conteúdo para gerar resumos concisos e estruturados, além de criar quizzes e perguntas para testar o conhecimento. A grande inovação reside na utilização de LLMs rodando localmente via Ollama, garantindo privacidade dos dados e independência de serviços em nuvem. Isso é particularmente útil para estudantes, pesquisadores e profissionais que lidam com grandes volumes de texto e necessitam de métodos eficientes para assimilar e revisar informações. A modularidade do projeto, o uso de tipagem estática e a saída de terminal aprimorada com Rich são pilares para garantir um produto robusto e de fácil manutenção.

Tecnologias Empregadas

O projeto `study-cli` será construído utilizando um conjunto de tecnologias Python modernas e eficientes, cada uma escolhida por sua capacidade de resolver um problema específico no fluxo de trabalho. A seguir, detalhamos cada uma delas:

Ollama

Descrição: Ollama é um framework leve e extensível que permite executar e gerenciar modelos de linguagem grandes (LLMs) diretamente na máquina local do usuário [1]. Isso elimina a necessidade de depender de serviços de nuvem, garantindo privacidade dos dados e a capacidade de trabalhar offline. Ele empacota os pesos do modelo, configuração e dados em um único pacote, facilitando a distribuição e o uso de LLMs de código aberto como Mistral, Llama2 e Phi.

Uso no Projeto: No `study-cli`, Ollama será o motor por trás de todas as operações de IA. Ele será responsável por:

- Gerar resumos estruturados do conteúdo dos livros/PDFs.
- Criar perguntas e quizzes com base no texto processado.
- Responder a perguntas livres do usuário, buscando informações no índice de documentos.

Exemplo Introdutório:
Para usar o Ollama, o usuário precisaria primeiro instalá-lo e baixar um modelo, por exemplo, o `llama2`:

```Bash
ollama run llama2
```

No código Python, a interação com o Ollama pode ser feita diretamente ou, como faremos, através do LangChain.

LangChain

Descrição: LangChain é um framework poderoso e flexível para o desenvolvimento de aplicações que utilizam Large Language Models (LLMs) [2]. Ele simplifica a orquestração de prompts, o encadeamento de operações com LLMs e a integração com outras fontes de dados e ferramentas. LangChain oferece componentes modulares que podem ser combinados para criar fluxos de trabalho complexos, como a geração de resumos seguida pela criação de perguntas.

Uso no Projeto: LangChain será crucial para:

- Orquestrar o fluxo de trabalho de processamento do texto: extração -> chunking -> embedding -> resumo -> quiz.
- Gerenciar os prompts enviados ao Ollama para garantir respostas coerentes e estruturadas.
- Integrar o Ollama com o ChromaDB para a funcionalidade de busca e resposta a perguntas.

Exemplo Introdutório:
Um exemplo básico de como o LangChain pode interagir com um LLM (neste caso, via Ollama) para gerar uma resposta:

Python

```python
from langchain_community.llms import Ollama
from langchain.prompts import ChatPromptTemplate

llm = Ollama(model="llama2")
prompt = ChatPromptTemplate.from_messages([
    ("system", "Você é um assistente útil. Responda a todas as perguntas com precisão."),
    ("user", "{question}")
])

chain = prompt | llm
response = chain.invoke({"question": "Qual a capital da França?"})
print(response)
```

FAISS (via LangChain)

Descrição: FAISS (Facebook AI Similarity Search) é uma biblioteca para busca eficiente de similaridade e agrupamento de vetores densos. No contexto do LangChain, o FAISS pode ser utilizado como um *vectorstore* local, permitindo o armazenamento e a busca rápida de embeddings de texto. Ele é ideal para cenários onde a persistência de um banco de dados vetorial completo não é necessária ou onde se prefere uma solução mais leve e em memória para a busca de similaridade [3].

Uso no Projeto: FAISS será utilizado para:

- Armazenar os embeddings dos chunks de texto extraídos dos PDFs/TXTs em memória (ou persistir em disco, se necessário).
- Realizar buscas de similaridade eficientes para encontrar os chunks de texto mais relevantes para uma dada consulta do usuário.
- Facilitar a recuperação de passagens relevantes do texto original quando o usuário fizer perguntas livres, alimentando o LLM com contexto.

Exemplo Introdutório:
Criar um FAISS vectorstore e adicionar documentos:

Python

```python
from langchain_community.vectorstores import FAISS
from langchain_community.embeddings import OllamaEmbeddings
```

```python
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Supondo que 'docs' são os documentos já carregados e divididos
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=200)
texts = text_splitter.split_text("Seu texto longo aqui...")

embeddings = OllamaEmbeddings(model="llama2")
vectorstore = FAISS.from_texts(texts, embeddings)

# Para buscar documentos similares
query = "Qual é o principal tópico?"
docs = vectorstore.similarity_search(query)
print(docs[0].page_content)
```

pdfplumber ou pypdf2

Descrição: `pdfplumber` e `pypdf2` são bibliotecas Python para extração de texto de arquivos PDF. `pdfplumber` é conhecido por sua capacidade de extrair informações detalhadas sobre caracteres, retângulos e linhas, além de tabelas, sendo mais robusto para PDFs com layouts complexos [4]. `pypdf2` é uma biblioteca mais simples e eficaz para extração de texto, mas pode ter limitações com PDFs complexos [5].

Uso no Projeto: Uma dessas bibliotecas será utilizada na fase de carregamento para:

- Extrair o texto bruto de arquivos PDF fornecidos pelo usuário.
- Lidar com diferentes estruturas de PDF para garantir a máxima extração de conteúdo.

Exemplo Introdutório (pdfplumber):

Python

```python
import pdfplumber

with pdfplumber.open("path/to/your/document.pdf") as pdf:
    first_page = pdf.pages[0]
    text = first_page.extract_text()
    print(text)
```

Exemplo Introdutório (pypdf2):

Python

```python
from PyPDF2 import PdfReader

reader = PdfReader("path/to/your/document.pdf")
page = reader.pages[0]
```

```
text = page.extract_text()
print(text)
```

Typer

Descrição: Typer é uma biblioteca Python para construir interfaces de linha de comando (CLIs) de forma rápida e fácil, baseada em type hints do Python [6]. Ele simplifica a criação de comandos, argumentos e opções, tornando o desenvolvimento de CLIs mais intuitivo e a aplicação mais robusta e fácil de usar.

Uso no Projeto: Typer será a base para a construção da CLI, permitindo:

- Definir os comandos principais (`load` , `summary` , `quiz` , `ask`).
- Gerenciar argumentos e opções de linha de comando de forma clara e tipada.
- Organizar a estrutura da CLI em módulos, como `cli.py` .

Exemplo Introdutório:

Python

```python
import typer

app = typer.Typer()

@app.command()
def hello(name: str):
    print(f"Olá {name}")

@app.command()
def goodbye(name: str, formal: bool = False):
    if formal:
        print(f"Adeus, Sr./Sra. {name}.")
    else:
        print(f"Tchau {name}!")

if __name__ == "__main__":
    app()
```

Rich

Descrição: Rich é uma biblioteca Python para adicionar texto rico e formatação bonita ao terminal [7]. Ele permite exibir cores, estilos, tabelas, barras de progresso, Markdown e até mesmo realce de sintaxe, melhorando significativamente a experiência do usuário com aplicações de linha de comando.

Uso no Projeto: Rich será empregado para:

- Melhorar a saída visual de todos os comandos da CLI.
- Exibir resumos e quizzes de forma formatada e legível.
- Apresentar mensagens de status e erros de maneira clara e atraente.

Exemplo Introdutório:

Python

```python
from rich.console import Console
from rich.table import Table

console = Console()

console.print("[bold green]Resumo Gerado com Sucesso![/bold green]")

table = Table(title="Quiz")
table.add_column("Pergunta", style="cyan")
table.add_column("Resposta Correta", style="magenta")

table.add_row("Qual a capital da França?", "Paris")
table.add_row("Quem escreveu 'Dom Quixote'?", "Miguel de Cervantes")

console.print(table)
```

Fluxo do Projeto Detalhado

O fluxo de trabalho do `study-cli` pode ser dividido nas seguintes etapas:

1. **Input do Usuário:** O usuário inicia o processo executando um comando como `study-cli load livro.pdf`.
2. **Extração de Texto:** O `loaders.py` utiliza `pdfplumber` ou `pypdf2` para extrair o texto completo do arquivo PDF ou TXT.
3. **Fragmentação (Chunking):** O texto extraído é dividido em pedaços menores (chunks) de 200-500 tokens. Essa fragmentação é crucial para o processamento eficiente pelos LLMs e para a busca de passagens relevantes no ChromaDB.
4. **Armazenamento e Indexação no FAISS:** Os chunks de texto são enviados para o `vectorstore.py`, que os armazena no FAISS (em memória ou persistido em disco). Durante o armazenamento, embeddings são gerados para cada chunk, permitindo a busca semântica.
5. **Comandos do Usuário:**

- `study-cli summary`: O `study.py` orquestra, via LangChain, o envio dos chunks relevantes (ou do texto completo, dependendo da estratégia de resumo) para o Ollama (`llm.py`) para gerar um resumo estruturado. A saída é formatada com Rich.
- `study-cli quiz`: Similar ao resumo, o `study.py` instrui o Ollama a gerar 5-10 perguntas e suas respostas com base no conteúdo indexado. A apresentação é feita com Rich.
- `study-cli ask "Qual a tese do capítulo 3?"`: O `study.py` recebe a pergunta do usuário. O ChromaDB é consultado para recuperar os chunks de texto mais relevantes para a pergunta. Esses chunks, juntamente com a pergunta, são enviados ao Ollama via LangChain para gerar uma resposta contextualizada. A resposta é exibida com Rich.

6. **Output e Exportação:** Todas as saídas são exibidas no terminal com formatação Rich. O usuário terá a opção de exportar resumos e quizzes para arquivos `.txt` através de `utils.py`.

Estrutura Inicial de Pastas

Plain Text

```
study-cli/
|── cli.py          # Entrada principal com Typer
|── loaders.py      # Extrai texto de PDF/TXT
|── vectorstore.py  # Configuração do FAISS
|── llm.py          # Conexão com Ollama via LangChain
|── study.py        # Funções de resumo, quiz, QA
|── utils.py        # Funções auxiliares (ex: salvar em txt)
|── data/           # PDFs e textos do usuário
|── outputs/        # Resumos e quizzes salvos
|── .env            # Variáveis de ambiente (ex: modelo Ollama)
|── requirements.txt # Dependências do projeto
```

Boas Práticas Pythonicas

Para garantir a qualidade, manutenibilidade e escalabilidade do projeto, as seguintes boas práticas Pythonicas serão adotadas:

- **Tipagem estática com `typing`:** Utilização extensiva de type hints (`List[str]`, `Dict`, `Optional`, etc.) para melhorar a legibilidade do código, facilitar a detecção de erros e auxiliar no desenvolvimento com IDEs.
- **Estrutura modular:** Cada arquivo terá uma responsabilidade clara e bem definida, seguindo o princípio da separação de preocupações. Isso facilita a colaboração entre os programadores e a manutenção do código.

- **Docstrings e comentários concisos:** Todas as funções, classes e módulos importantes terão docstrings explicando seu propósito, argumentos e retornos. Comentários serão usados para explicar lógicas complexas ou decisões de design específicas.
- **Uso de `logging` em vez de `print` brutos:** Para um controle mais granular sobre as mensagens de saída (informações, avisos, erros) e para facilitar a depuração em diferentes ambientes, o módulo `logging` será preferido em detrimento de `print` statements.
- **Configurações via `.env` + `pydantic`:** Variáveis de configuração sensíveis ou que podem mudar entre ambientes (como o modelo Ollama a ser usado) serão gerenciadas através de arquivos `.env` e carregadas de forma segura e validada usando a biblioteca `pydantic`.
- **Testes unitários e de integração:** Serão desenvolvidos testes para garantir a funcionalidade de cada componente e a integração entre eles, utilizando frameworks como `pytest`.

Cronograma de Sprints (3 Meses)

Este cronograma é uma estimativa e pode ser ajustado conforme o progresso e os desafios encontrados. Ele é dividido em 3 meses, com cada mês contendo 2 sprints de 2 semanas. A equipe de 3 programadores trabalhará em paralelo nas tarefas designadas.

Mês 1: Fundação e Extração

Sprint 1 (Semanas 1-2)

Foco: Configuração do ambiente, estrutura básica da CLI e extração de texto.

Objetivos:

- Configurar o repositório do projeto e ambiente de desenvolvimento.
- Implementar a estrutura básica da CLI com Typer (`cli.py`).
- Desenvolver o módulo `loaders.py` para extrair texto de arquivos TXT.
- Pesquisar e selecionar a melhor biblioteca para extração de PDF (`pdfplumber` ou `pypdf2`).
- Implementar a extração de texto de PDFs usando a biblioteca escolhida.

Tarefas por Programador:

- **Programador 1:** Configuração do projeto (repositório, `requirements.txt` , `.env` com `pydantic`), estrutura básica da CLI (`cli.py` com comando `load` inicial).
- **Programador 2:** Desenvolvimento do `loaders.py` para TXT, pesquisa e prototipagem de extração de PDF com `pypdf2` .

- **Programador 3:** Desenvolvimento do `loaders.py` para PDF com `pdfplumber`, criação de testes unitários para `loaders.py`.

Entregáveis:

- Repositório Git inicializado.
- `cli.py` funcional com comando `load` para TXT.
- `loaders.py` capaz de extrair texto de TXT e PDF.
- `requirements.txt` e `.env` configurados.
- Testes unitários iniciais para `loaders.py`.

Sprint 2 (Semanas 3-4)

Foco: Chunking, Embeddings e FAISS.

Objetivos:

- Implementar a fragmentação de texto (chunking).
- Configurar e integrar o FAISS (`vectorstore.py`).
- Gerar embeddings para os chunks de texto.
- Refinar o comando `load` para incluir chunking e armazenamento no FAISS.

Tarefas por Programador:

- **Programador 1:** Implementação da lógica de chunking em `loaders.py` ou `utils.py`.
- **Programador 2:** Configuração do FAISS em `vectorstore.py`, integração com o processo de embedding (via LangChain/Ollama).
- **Programador 3:** Refinamento do comando `load` em `cli.py` para orquestrar extração, chunking e armazenamento no FAISS, testes de integração para o fluxo de carregamento.

Entregáveis:

- Funcionalidade de chunking implementada.
- FAISS configurado e integrado.
- Comando `load` completo, armazenando chunks com embeddings no FAISS.
- Testes de integração para o fluxo de carregamento.

Mês 2: Funcionalidades de IA e Interação

Sprint 3 (Semanas 5-6)

Foco: Integração com Ollama e funcionalidade de resumo.

Objetivos:

- Configurar a conexão com Ollama via LangChain (`llm.py`).
- Desenvolver a funcionalidade de resumo (`study.py`).
- Implementar o comando `summary` na CLI.
- Utilizar Rich para formatação da saída do resumo.

Tarefas por Programador:

- **Programador 1:** Desenvolvimento de `llm.py` para conexão com Ollama e LangChain, prototipagem de prompts para resumo.
- **Programador 2:** Implementação da lógica de resumo em `study.py` utilizando `llm.py` e LangChain.
- **Programador 3:** Implementação do comando `summary` em `cli.py` , integração com Rich para formatação da saída, testes unitários para `study.py` (resumo).

Entregáveis:

- `llm.py` funcional para interação com Ollama.
- Comando `summary` gerando resumos formatados com Rich.
- Testes unitários para a funcionalidade de resumo.

Sprint 4 (Semanas 7-8)

Foco: Funcionalidade de Quiz e Perguntas Livres.

Objetivos:

- Desenvolver a funcionalidade de geração de quizzes (`study.py`).
- Implementar o comando `quiz` na CLI.
- Desenvolver a funcionalidade de perguntas livres (`study.py`) com busca no ChromaDB.
- Implementar o comando `ask` na CLI.

Tarefas por Programador:

- **Programador 1:** Desenvolvimento da lógica de geração de quizzes em `study.py` , prototipagem de prompts para quiz.
- **Programador 2:** Desenvolvimento da lógica de perguntas livres em `study.py` (integração ChromaDB + Ollama via LangChain).
- **Programador 3:** Implementação dos comandos `quiz` e `ask` em `cli.py` , integração com Rich para formatação das saídas, testes unitários para `study.py` (quiz e ask).

Entregáveis:

- Comando `quiz` gerando quizzes formatados com Rich.
- Comando `ask` respondendo a perguntas livres com base no contexto.
- Testes unitários para as funcionalidades de quiz e perguntas livres.

Mês 3: Refinamento, Exportação e Documentação Final

Sprint 5 (Semanas 9-10)

Foco: Exportação, tratamento de erros e melhorias de UX.

Objetivos:

- Implementar a funcionalidade de exportação para `.txt` (`utils.py`).
- Melhorar o tratamento de erros e mensagens para o usuário.
- Refinar a experiência do usuário com Rich (barras de progresso, spinners).
- Revisão e otimização de prompts para Ollama.

Tarefas por Programador:

- **Programador 1:** Desenvolvimento da funcionalidade de exportação em `utils.py` , integração com `cli.py` .
- **Programador 2:** Implementação de tratamento de erros robusto e mensagens de feedback amigáveis, adição de elementos Rich (progresso, spinners).
- **Programador 3:** Revisão e otimização de todos os prompts utilizados com Ollama, testes de ponta a ponta.

Entregáveis:

- Funcionalidade de exportação para `.txt` .
- Tratamento de erros aprimorado.
- Melhorias na UX da CLI com Rich.
- Prompts otimizados.

Sprint 6 (Semanas 11-12)

Foco: Testes finais, documentação e preparação para lançamento.

Objetivos:

- Realizar testes de aceitação do usuário (UAT) internos.
- Finalizar a documentação técnica e de usuário.
- Preparar o projeto para distribuição (empacotamento).

- Revisão geral do código e refatoração, se necessário.

Tarefas por Programador:

- **Programador 1:** Coordenação dos testes de aceitação, coleta de feedback.
- **Programador 2:** Finalização da documentação técnica (README, exemplos de uso), empacotamento do projeto.
- **Programador 3:** Finalização da documentação de usuário, revisão de código e refatoração.

Entregáveis:

- Relatório de testes de aceitação.
- Documentação técnica e de usuário completa.
- Projeto empacotado e pronto para distribuição.
- Código revisado e refatorado.

Referências

[1] Ollama. Disponível em: https://ollama.com/
[2] LangChain. Disponível em: https://www.langchain.com/
[3] Chroma Docs. Disponível em: https://docs.trychroma.com/
[4] jsvine/pdfplumber. Disponível em: https://github.com/jsvine/pdfplumber
[5] PyPDF2 Documentation. Disponível em: https://pypdf2.readthedocs.io/
[6] Typer. Disponível em: https://typer.tiangolo.com/
[7] Rich. Disponível em: https://rich.readthedocs.io/