assignment_6

October 25, 2025

[ ]:
```python
# === IMPORTS ===
import torch
from transformers import pipeline, AutoTokenizer
from sentence_transformers import SentenceTransformer, CrossEncoder, util
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
import os
import pandas as pd
from tqdm import tqdm
import time
import requests
import zipfile
import io
import re
```

[ ]:
```python
# === DATA LOADING (Needle in a Haystack) ===
def extract_expected_from_needle(needle: str) -> str:
    """Extracts the core answer from the 'needle' sentence."""
    needle = needle.strip()
    # Try to match quoted text
    m = re.search(r'\bis\s+(?:a|an|the)?\s*"([^"]+)"', needle, flags=re.
 ↪IGNORECASE)
    if m:
        return m.group(1).strip()

    # Try to match text after 'is' until a period
    m2 = re.search(r'\bis\s+(.*?)(?:\.\s*$|$)', needle, flags=re.IGNORECASE)
    if m2:
        val = m2.group(1).strip()
        # Remove articles
        val = re.sub(r'^(?:a|an|the)\s+', '', val, flags=re.IGNORECASE).strip()
        val = val.strip(' ."\'' )
        return val
    return needle
```

[ ]:
```python
def load_test_cases(company_path):
```

```
    """Loads all test cases (PDFs, questions, needles) from the directory␣
↪structure."""
    test_cases = []
    company_name = os.path.basename(company_path)
    subfolders = [f for f in os.listdir(company_path) if os.path.isdir(os.path.
↪join(company_path, f))]
    # Sort folders by page number
    subfolders = sorted(subfolders, key=lambda x: int(''.join(filter(str.
↪isdigit, x))) if any(ch.isdigit() for ch in x) else 0)

    for subfolder in tqdm(subfolders, desc="Processing subfolders"):
        subfolder_path = os.path.join(company_path, subfolder)
        # Define paths
        pdf_file_name = f"{subfolder}_TextNeedles.pdf"
        pdf_path = os.path.join(subfolder_path, pdf_file_name)
        needles_file = os.path.join(subfolder_path, "needles.csv")
        questions_file = os.path.join(subfolder_path, "prompt_questions.txt")

        # Load needles and questions
        try:
            needles_df = pd.read_csv(needles_file, header=None)
            needles_df.columns = ['needle']

            with open(questions_file, 'r', encoding="utf-8") as f:
                questions = [line.strip() for line in f.readlines() if line.
↪strip()]
        except Exception as e:
            print(f"Error loading metadata for {subfolder}: {e}")
            continue

        # Load PDF text using PyPDFLoader
        try:
            loader = PyPDFLoader(pdf_path)
            docs = loader.load_and_split()
            full_text = " ".join(page.page_content for page in docs)
            print(len(full_text))
            doc_id = pdf_file_name
        except Exception as e:
            print(f"Error loading PDF {pdf_path}: {e}")
            continue

        # Create test cases
        if len(needles_df) != len(questions):
          print(f"Warning: mismatch in {subfolder} - needles={len(needles_df)},␣
↪questions={len(questions)}")

        for ( _, row), question in zip(needles_df.iterrows(), questions):
```

```
            needle = row['needle']

            try:
                expected_value = extract_expected_from_needle(needle)

            except Exception:
                expected_value = needle

            test_cases.append({
                "company": company_name,
                "doc_id": doc_id,
                "full_text": full_text,
                "question": question,
                "expected_answer": expected_value,
                "needle": needle
            })
    print(f"\nTotal test cases loaded: {len(test_cases)}")
    return test_cases
```

[ ]:
```
# Colab
company_path = "/content/drive/MyDrive/Colab Notebooks/GoldmanSachs"
test_cases = load_test_cases(company_path)
```

```
Total test cases loaded: 165
```

[ ]:
```
# Load the data
company_path = "/kaggle/input/glodmansachs/GoldmanSachs"
test_cases = load_test_cases(company_path)
```

[ ]:
```
# === RAG COMPONENT DEFINITIONS ===
class Chunker:
    def __init__(self, chunk_size=500, chunk_overlap=100):
        self.splitter = RecursiveCharacterTextSplitter(
            chunk_size=chunk_size,
            chunk_overlap=chunk_overlap
        )
    def chunk(self, document_text):
        return self.splitter.split_text(document_text)

class Embedder:
    def __init__(self, model_name="BAAI/bge-small-en-v1.5"):
        self.model = SentenceTransformer(model_name)
    def embed(self, texts, batch_size=16):
        return self.model.encode(texts, convert_to_tensor=True,␣
↪batch_size=batch_size, show_progress_bar=False)

class Retriever:
    def __init__(self, embedder):
        self.embedder = embedder
```

```
    def retrieve(self, query, chunks, chunk_embeddings, top_k=5):
        query_embedding = self.embedder.embed(query)
        similarities = util.pytorch_cos_sim(query_embedding,␣
 ↪chunk_embeddings)[0]
        actual_k = min(top_k, len(chunks))
        top_k_indices = torch.topk(similarities, k=actual_k).indices
        return [chunks[i] for i in top_k_indices]

class Reranker:
    def __init__(self, model_name='cross-encoder/ms-marco-MiniLM-L-6-v2'):
        self.model = CrossEncoder(model_name)
    def rerank(self, query, chunks, top_k=5):
        pairs = [[query, chunk] for chunk in chunks]
        scores = self.model.predict(pairs)

        # Combine chunks with scores and sort
        reranked_chunks = sorted(zip(chunks, scores), key=lambda x: x[1],␣
 ↪reverse=True)

        actual_k = min(top_k, len(reranked_chunks))
        return [chunk for chunk, score in reranked_chunks[:actual_k]]

from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

class Generator:
    def __init__(self, model_name="HuggingFaceTB/SmolLM-135M-Instruct",␣
 ↪device=None):
        self.model_name = model_name
        if device is None:
            self.device = 0 if torch.cuda.is_available() else -1
        else:
            self.device = device
        model_lower = self.model_name.lower()
        self.is_smol = "smol" in model_lower

        if self.is_smol:
            print(f"Loading SmolLM model: {model_name}")
            self.pipeline = pipeline(
                "text-generation",
                model=model_name,
                device=self.device,
            )
            self.tokenizer = None
            print(f"SmolLM loaded on device: {self.device}")

        else:
            # General Causal LM (Gemma, etc.)
```

```
            print(f"Loading general Causal LM: {model_name}. This may take a␣
↪moment...")
            try:
                self.tokenizer = AutoTokenizer.from_pretrained(model_name)
                self.model = AutoModelForCausalLM.from_pretrained(
                    model_name,
                    device_map="auto",
                    torch_dtype="auto"
                )

                self.pipeline = pipeline(
                    "text-generation",
                    model=self.model,
                    tokenizer=self.tokenizer
                )
                print(f"Successfully loaded {model_name}.")
                if torch.cuda.is_available():
                     print(f"Model is on device: {self.model.device}")
            except Exception as e:
                print(f"Failed to load model {model_name} with␣
↪AutoModelForCausalLM.")
                raise e

        self.system_prompt = (
            "You are a helpful assistant that answers questions based on the␣
↪provided context. "
            "Provide direct, concise answers."
        )

    def generate(self, query, context_chunks, max_new_tokens=150):
        context_text = "\n\n".join(context_chunks)
        user_question = f"Context:\n{context_text}\n\nQuestion: {query}\nAnswer:
↪"

        if self.is_smol:
            prompt = f"{self.system_prompt}\n{user_question}"

            output = self.pipeline(
                prompt,
                max_new_tokens=max_new_tokens,
                return_full_text=False,
                do_sample=False,
                temperature=0.0,
                top_p=1.0
            )
            generated_text = output[0]['generated_text']
            if generated_text.startswith(prompt):
```

```
                return generated_text[len(prompt):].strip()
            return generated_text.strip()

        else:
            # General Causal LM (Gemma) template
            full_user_prompt = f"{self.system_prompt}\n\n{user_question}"
            messages = [
                # NO 'system' role
                {"role": "user", "content": full_user_prompt}
            ]
            output = self.pipeline(
                messages,
                max_new_tokens=max_new_tokens,
                return_full_text=False,
                do_sample=False,
                temperature=0.0,
                top_p=1.0
            )
            return output[0]['generated_text'].strip()

class RAGPipeline:
    def __init__(self, chunker, embedder, retriever, generator, reranker=None):
        self.chunker = chunker
        self.embedder = embedder
        self.retriever = retriever
        self.generator = generator
        self.reranker = reranker

    def prepare_document(self, document_text):
        """Chunks and embeds a single document."""
        chunks = self.chunker.chunk(document_text)
        embeddings = self.embedder.embed(chunks)
        return chunks, embeddings

    def query(self, query, chunks, chunk_embeddings, top_k=5, rerank_top_k=3):
        # 1. Retrieve
        retrieved_chunks = self.retriever.retrieve(query, chunks,␣
 ↪chunk_embeddings, top_k=top_k)

        # 2. (Optional) Rerank
        if self.reranker:
            final_chunks = self.reranker.rerank(query, retrieved_chunks,␣
 ↪top_k=rerank_top_k)
        else:
            final_chunks = retrieved_chunks

        # 3. Generate
```

```
        answer = self.generator.generate(query, final_chunks)
        return answer, final_chunks
```

[ ]:
```
# === EVALUATION FUNCTION ===
def evaluate_rag(test_cases, rag_pipeline, top_k=5, rerank_top_k=3,␣
 ↪use_llm=True, log_errors=True):
    total_correct = 0
    total_time = 0
    errors = []

    print("Preparing all documents (chunking and embedding)...")
    # Pre-process all unique documents
    embeddings_dict, chunks_dict = {}, {}
    docs = pd.DataFrame(test_cases)

    for _, row in tqdm(docs.iterrows(), total=len(docs)):
        chunks, embeddings = rag_pipeline.prepare_document(row['full_text'])
        chunks_dict[row['doc_id']] = chunks
        embeddings_dict[row['doc_id']] = embeddings

    print("\nStarting evaluation...")

    for case in tqdm(test_cases):
        doc_id = case['doc_id']
        chunks = chunks_dict[doc_id]
        embeddings = embeddings_dict[doc_id]
        start_time = time.time()

        if not use_llm:
            # Retrieval-Only evaluation
            context_chunks = rag_pipeline.retriever.retrieve(
                case['question'], chunks, embeddings, top_k=top_k
            )
            # Check if needle is in any retrieved chunk
            pattern = re.escape(case['needle'].lower())
            is_correct = any(re.search(pattern, chunk.lower()) for chunk in␣
 ↪context_chunks)
        else:
            # End-to-end RAG evaluation
            answer, context_chunks = rag_pipeline.query(
                case['question'], chunks, embeddings, top_k=top_k,␣
 ↪rerank_top_k=rerank_top_k
            )
            # Check if expected answer is in the generated answer
            pattern = re.escape(case['expected_answer'].lower())
            is_correct = bool(re.search(pattern, answer.lower()))
```

```
        total_time += time.time() - start_time
        if is_correct:
            total_correct += 1
        elif log_errors:
            errors.append({
                "company": case["company"],
                "doc_id": doc_id,
                "question": case["question"],
                "expected_answer": case["expected_answer"],
                "needle": case["needle"],
                "retrieved_context": "\n---\n".join(context_chunks[:2])
            })

    # Report results
    accuracy = total_correct / len(test_cases) * 100
    print(f"Accuracy: {accuracy:.2f}% ({total_correct}/{len(test_cases)}␣
 ↪correct)")
    print(f"Total Time: {total_time:.2f}s ({total_time/len(test_cases)*1000:.
 ↪2f} ms/query)")

    return accuracy, errors
```

[ ]:
```
# === BASELINE CONFIGURATION ===
print("--- Configuring Baseline RAG Pipeline ---")
baseline_chunker = Chunker(chunk_size=500, chunk_overlap=100)
baseline_embedder = Embedder(model_name="BAAI/bge-small-en-v1.5")
baseline_retriever = Retriever(baseline_embedder)
baseline_generator = Generator("HuggingFaceTB/SmolLM-135M-Instruct")

baseline_pipeline = RAGPipeline(baseline_chunker, baseline_embedder,␣
 ↪baseline_retriever, baseline_generator)
```

[ ]:
```
# === RUN EXPERIMENTS ===
# --- Running Baseline Evaluation (Retrieval-Only) ---
print("\n--- [Baseline] Starting Retrieval-Only Evaluation ---")
results_baseline_retrieval = evaluate_rag(test_cases, baseline_pipeline,␣
 ↪top_k=5, use_llm=False)
```

```
==================================
EVALUATION RESULTS
==================================
Mode: Retrieval-only
Accuracy: 83.64% (138/165 correct)
Total Time: 2.14s (12.99 ms/query)
```

[ ]:
```
# --- Running Baseline Evaluation (End-to-End) ---
print("\n--- [Baseline] Starting End-to-End Evaluation ---")
results_baseline_full = evaluate_rag(test_cases, baseline_pipeline, top_k=5,␣
↪use_llm=True)
```

```
==================================
EVALUATION RESULTS
==================================
Mode: End-to-end
Accuracy: 36.36% (60/165 correct)
Total Time: 983.63s (5961.42 ms/query)
```

[ ]:
```
# --- EXPERIMENT 1: UPGRADE GENERATOR ---
print("--- Configuring Experiment 1: Upgrading Generator ---")
# Reuse components from baseline
exp1_generator = Generator(model_name="google/gemma-2b-it")
exp1_pipeline = RAGPipeline(baseline_chunker, baseline_embedder,␣
↪baseline_retriever, exp1_generator)
# --- Running Experiment 1 Evaluation (End-to-End) ---
print("\n--- [Experiment 1] Starting End-to-End Evaluation ---")
results_exp1 = evaluate_rag(test_cases, exp1_pipeline, top_k=5, use_llm=True)
```

```
==================================
EVALUATION RESULTS

==================================
Mode: End-to-end
Accuracy: 76.97% (127/165 correct)
Total Time: 314.37s (1905.30 ms/query)
```

[ ]:
```
# --- EXPERIMENT 2: INTRODUCE RERANKER ---
print("--- Configuring Experiment 2: Introducing Reranker ---")
exp2_reranker = Reranker(model_name='cross-encoder/ms-marco-MiniLM-L-6-v2')
exp2_pipeline = RAGPipeline(
    chunker=baseline_chunker,
    embedder=baseline_embedder,
  retriever=baseline_retriever,
    generator=exp1_generator, # Use the upgraded generator
    reranker=exp2_reranker
)
# --- Running Experiment 2 Evaluation (End-to-End) ---
# Note: Increase retrieval top_k to 10 to feed reranker
print("\n--- [Experiment 2] Starting End-to-End Evaluation ---")
results_exp2 = evaluate_rag(test_cases, exp2_pipeline, top_k=10,␣
↪rerank_top_k=3, use_llm=True)
```

```
==================================
EVALUATION RESULTS
==================================
Mode: End-to-end
Accuracy: 83.03% (137/165 correct)
Total Time: 250.39s (1517.54 ms/query)
```

[ ]:
```python
# --- EXPERIMENT 3: SMALLER CHUNKS ---
print("--- Configuring Experiment 3a: Smaller Chunks (size=256) ---")
exp3a_chunker = Chunker(chunk_size=256, chunk_overlap=50)


exp3a_pipeline = RAGPipeline(
    chunker=exp3a_chunker,
    embedder=baseline_embedder,
    retriever=baseline_retriever,
    generator=exp1_generator,
    reranker=exp2_reranker
)

# --- Running Experiment 3a Evaluation ---
print("\n--- [Experiment 3a] Starting End-to-End Evaluation ---")
results_exp3a = evaluate_rag(test_cases, exp3a_pipeline, top_k=10,␣
 ↪rerank_top_k=3, use_llm=True)
```

```
==================================
EVALUATION RESULTS
==================================
Mode: End-to-end
Accuracy: 94.55% (156/165 correct)
Total Time: 223.56s (1354.88 ms/query)
```