Technical Report

Executive Summary

This paper explores the implementation of advanced RAG features on a naïve RAG pipeline, the comparison of these respective pipelines, and then a comparison of the RAGAS results for both these pipelines. This paper also assesses the advanced RAG pipeline's deployability. The key finding was that implementing advanced RAG does incrementally improve the Exact Match and F1 scores of the pipeline. The improvement, however, comes at a cost according to the RAGAS results with the answer relevancy reducing by several percentage points. But this cost is outweighed by the benefits.

Data Understanding

The dataset from HuggingFace contains two parts. The first is the text-corpus that consists of the Wikipedia passages and information. This collection of data is what will be embedded into the vector database. The second part is the collection of question-answer pairs that will be used to test the accuracy of the RAG pipeline's output. There are 918 question-answer pairs or queries total with indices between 0 and 1,714. This means that technically there are a total of 1,715 indices but for only 918 values. This implies that data has been removed for one reason or another and is part of Exploratory Data Analysis or EDA (see Appendix B). That means the creators of this dataset have done their own data cleaning. Further analysis of the dataset showed that the data has no empty rows, which means that the dataset creator was thorough in their own EDA.

I chose the "all-MiniLM-L6-v2" for my sentence transformer because I have not used any sentence transformers before and wanted to utilize the recommended one. It was well documented and provided clear guidance on how to use it. I chose Milvus Lite for my vector database under a similar reason. Since I did not have prior experience with any vector databases in general, I wanted to begin with the recommended database and familiarize myself with its functions. Milvus was also well documented with many examples and explanations on how to prepare and build the database. I chose Flan T5 for my LLM because Llama 3.2 was taking too long to load. Flan T5's transformers loaded almost instantaneously. Flan T5 also had great documentation on huggingface in regards to how to access its transformers and then utilize it.

System Architecture

I designed the RAG pipeline to utilize the naïve RAG module as its primary module for instantiating the RAG features. The enhanced RAG module served more or less as a light utility class for naïve RAG to use. The reason for this was to avoid repeating code when instantiating the pipeline. If statements to handle enhanced RAG features were included in the naïve rag module to facilitate this. The trade-off was that in a sense naïve RAG and enhanced RAG features were coupled tightly together and ideally the code should be decoupled as much as possible. The decision was also made to maintain the embedding code in the actual implementation of the modules in the notebooks. This was due to difficulties in passing strings to methods and using them to access columns in the pandas data frame. The notebooks for enhanced RAG features and naïve RAG features were also kept separate. This was to assist in differentiating between the differing RAG calls that would occur for both pipelines.

Experimental Results

The initial naïve RAG evaluation began without content retrieval due to inexperience with RAG pipelines. The starting Exact Match score began at 39 out of 918 answers and after incorporating content retrieval the score increased to 41. The evaluation, however, had not been properly implementing the top-1 retrieval method required for the initial evaluation. After top-1 was applied, the Exact Match score actually increased to 107.

After calculating the Exact Match (EM) scores for three separate prompting styles, the instruction prompt style performed the best. The instruction prompt also incorporated a persona but not a distinct one. All three prompts, however, scored within one point of one another for EM. The instruction prompt scored the highest for the F1 score. All three prompts also scored within a few points of one another for the F1 score as well.

Overall, the instruction prompt scored the best.

Increasing the embedding size decreased performance and so did increasing the passage selection to 10. For top-5, the RAG pipeline was able to maintain the same scores as the top-1. This was most likely due to the introduction of too much noise to the SLM.

Enhancement Analysis

I originally began with chunking 500 characters at a time but that lowered the accuracy level due to loss in information from the embedding model's maximum token limit of 256. To avoid this and maximize chunking coverage, I reduced the chunking size to 200. The scores increased across the board for all three prompt types.

While implementing the second RAG feature of reranking distance scores, I tested varying thresholds of permissive and strict scoring. I discovered that a threshold of 0.5 achieved the highest scores. With both features implemented, the third type of prompt using CoT ranked marginally highest in EM and F1 scores.

After applying the enhancements, the results confirmed that advanced RAG features do incrementally improve the pipeline results. But it also emphasized that even with certain features such as reranking, the right threshold needs to be found. This is likely to account for the capabilities of the LLM generating the answers.

As expected when increasing the passage selection to top-10, the scores dropped also likely due to noise. Although the advanced RAG features improved the scoring, the results were still limited by the capabilities of the SLM.

Advanced RAG Analysis

After running a RAGAS evaluation using a Langchain wrapper, the results between the naïve rag and enhanced rag showed that the faithfulness, context_recall, and context precision has increased by an average of 10% each. However, answer_relevancy had diminished by around 5%. This relevancy decrease might have occurred due to chunking. The SLM could handle up to 256 tokens of input, but I reduced it to 200 to avoid pushing its limits. This reduction in relevancy, however, is made up for by the improvement in the RAG pipeline's context retrieval.

Production Considerations

For production, a more reliable vector database is needed. During testing, the connection to the vector database was constantly dropping. Replacing Milvus with Pinecone is likely a much smoother and more consistent experience.

For the LLM used to generate responses, Flan-T5 is fast and uses very little compute power. However, its token limit is much too small as it is technically an SLM. It is not enough if we scale to larger datasets. The laptop would also need to be traded out for better hardware that can handle a larger model such as one from OpenAI or Gemini.

In other words, to scale and deploy a reliable RAG pipeline, more than likely payment will need to be involved to generate the computing power needed to handle larger amounts of data.

Appendices

Technical Specifications
Embedding Model: sentence-transformer/all-MiniLM-L6-v2
Generation Model: google/flan-t5-small
Vector Database: Milvus Lite
Collection: rag_mini, rag_mini_512
Dataset: rag-datasets/rag-mini-wikipedia
Enhanced RAG Features: Chunking and Reranking
Top-k tested: 1, 5, 10 for Naïve RAG; 3, 10 for Enhanced RAG
Reranking Threshold: 0.5
Queries Tested: 100
RAGAS LLM Judge: Gemini-2.0-flash
Hardware: Macbook Air 13 inch Apple M3

Reproducibility Requirements
Generate Gemini API key from Google AI Studio. Register for free trial in case of restricted requests per minute.
Create a .env file with the api key to prevent hard coding the key in the notebook.

AI Usage Log
Appendix A
Tool: Claude Code
Purpose:
Input: For data_exploration.ipynb, I'm looking to do the following: Document dataset structure, sample entries, and data quality observations. What do you suggest?
Output: For your data exploration notebook, here's what I suggest adding to document dataset structure, sample entries, and data quality:

Dataset Structure Analysis

```
# Dataset dimensions and basic info
print("=== PASSAGES DATASET ===")
print(f"Shape: {passages.shape}")
print(f"Columns: {list(passages.columns)}")
print(f"Data types:\n{passages.dtypes}")
print(f"Index range: {passages.index.min()} to {passages.index.max()}")

print("\n=== QUERIES DATASET ===")
print(f"Shape: {queries.shape}")
print(f"Columns: {list(queries.columns)}")
print(f"Data types:\n{queries.dtypes}")
print(f"Index range: {queries.index.min()} to {queries.index.max()}")
```

Text Length Analysis

```
# Passage length statistics
passages['passage_length'] = passages['passage'].str.len()
passages['word_count'] = passages['passage'].str.split().str.len()

print("=== PASSAGE LENGTH STATISTICS ===")
print(passages[['passage_length', 'word_count']].describe())

# Question/answer length statistics
queries['question_length'] = queries['question'].str.len()
queries['answer_length'] = queries['answer'].str.len()
queries['question_words'] = queries['question'].str.split().str.len()

print("\n=== QUERY LENGTH STATISTICS ===")
```

```
print(queries[['question_length', 'answer_length',
'question_words']].describe())
```

Data Quality Checks

```
# Missing values
print("=== DATA QUALITY CHECKS ===")
print("Passages missing values:", passages.isnull().sum().sum())
print("Queries missing values:", queries.isnull().sum().sum())

# Empty strings
print("Empty passages:", (passages['passage'] == '').sum())
print("Empty questions:", (queries['question'] == '').sum())
print("Empty answers:", (queries['answer'] == '').sum())

# Duplicates
print("Duplicate passages:", passages.duplicated().sum())
print("Duplicate queries:", queries.duplicated().sum())
```

Sample Entries

```
# Display diverse samples
print("=== SAMPLE PASSAGES ===")
print("Shortest passage:")
print(passages.loc[passages['passage_length'].idxmin(), 'passage'][:200]
 + "...")

print("\nLongest passage:")
print(passages.loc[passages['passage_length'].idxmax(), 'passage'][:200]
 + "...")

print("\n=== SAMPLE Q&A PAIRS ===")
sample_queries = queries.sample(3)
for idx, row in sample_queries.iterrows():
   print(f"Q: {row['question']}")
   print(f"A: {row['answer']}\n")
```

Topic Distribution (Optional)

```
# Quick topic analysis
import re
print("=== CONTENT THEMES ===")
# Extract common entities/topics
topics = ['Lincoln', 'Wilson', 'Uruguay', 'President', 'War', 'America']
```

```
for topic in topics:
    count = passages['passage'].str.contains(topic, case=False).sum()
    print(f"{topic}: {count} passages")
```

This gives you comprehensive documentation of both datasets' structure, quality, and content characteristics.

Output Usage:

Verification: Verified using my own experience in EDA.

Appendix B

Tool: Claude Code

Purpose: To figure out why the index range of the test queries is so much higher than the shape.

Input: Is the index range showing how many qa pairs there are and passages of information?

Output: Not exactly. The index range shows the ID values used to identify each row, not the count.

What Index Range Shows

- Passages: index.min() to index.max() = ID range (likely 0 to 3199)
- Queries: index.min() to index.max() = ID range (likely 0 to 1714, but sparse)

What Actually Shows Count

- Shape: (3200, 1) = 3200 passages
- Shape: (918, 2) = 918 Q&A pairs

Why the Difference?

The queries dataset has sparse indices - notice from your earlier output that query IDs jump around: 0, 2, 4, 6, 8, ..., 1710, 1711, 1712, 1713, 1714.

So:
- 918 actual Q&A pairs exist
- IDs range from 0 to 1714 (with gaps)
- 3200 passages with consecutive IDs 0-3199

The index range tells you the ID numbering system, while shape tells you the actual count of data points.

> If there are 918 Q&A pairs, then why are there 1715 ids? Index says 0 to

1714

Great observation! The dataset has 918 Q&A pairs but 1715 possible IDs (0 to 1714) because it uses sparse indexing.

Why This Happens

The original dataset likely had 1715 Q&A pairs, but some were:
- Filtered out during preprocessing (low quality, duplicates, etc.)
- Removed for data cleaning
- Split into different subsets

What Remains

- Original IDs preserved: 0, 2, 4, 6, 8, ..., 1710, 1711, 1712, 1713, 1714
- Gaps in sequence: Missing IDs like 1, 3, 5, 7, etc.
- Final count: 918 surviving Q&A pairs

This is common in ML datasets - they maintain original IDs for traceability even after filtering. You can verify this:

```
print(f"Unique IDs: {len(queries.index.unique())}")  # Should be 918
print(f"ID range: {queries.index.min()} to {queries.index.max()}")  # 0
to 1714
print(f"Missing IDs: {1715 - len(queries.index.unique())}")  # Should be
 797
```

The sparse indexing helps maintain consistency with the original dataset structure and any reference systems that might use these specific ID numbers.

Output Usage: I used the reasoning for dropped indices to conclude that the creators of the dataset did their own EDA.

Verification: Verified using my own experience in data cleaning.

Appendix C

Tool Used: Claude 4.0 Sonnet

Purpose: To debug the sentence transformer's encode function.

Input: I'm getting an invalidindexerror for the encode call in my naive_rag.ipynb. What do you recommend I do to fix it?

Output: I can see the issue! You're getting a NameError: name 'passages' is not defined because the passages variable from the earlier cell isn't available in the current cell execution context.

The fix is simple - you need to pass the actual text data to the encode function. Here's what you should change:
⎿ User rejected replace cell in src/naive_rag.ipynb at cell f39adefe

from sentence_transformers import SentenceTransformer

embedding_model = SentenceTransformer("all-MiniLM-L6-v2")

# Encode Text - pass the passage text, not the dataframe
embeddings = embedding_model.encode(passages['passage'].tolist())
print(f"Embeddings shape: {embeddings.shape}")
print(f"Sample embedding: {embeddings[0][:5]}...") # Show first 5 values of first
embedding
Output Usage: I used only the portion telling me to convert the contents of the passages column to a list.
Verification: I verified in VSCode that the output was what I wanted.

Appendix D
Tool Used: Claude 4.0 Sonnet
Purpose: To find a solution to the DataNotMatchException I was getting for my rag_data variable.
Input: Now I'm getting this error: DataNotMatchException: <DataNotMatchException: (code=1,
message=Insert missed an field `vector` to collection without set nullable==true or set default_value)>
for this code: res = client.insert(collection_name="rag_mini", data=rag_data)
⎿ Selected 1 lines from src/naive_rag.ipynb in Visual Studio Code
Output: The error indicates that Milvus is expecting a field named "vector" but you defined it as
"embedding". This suggests there might be a mismatch between your schema definition and the
actual collection.

Try this fix - change your FieldSchema to use "vector" as the field name:

# Update your schema definition
id_ = FieldSchema(name="id", dtype=DataType.INT64, is_primary=True)
passage = FieldSchema(name="passage", dtype=DataType.VARCHAR, max_length=65535)
embedding = FieldSchema(name="vector", dtype=DataType.FLOAT_VECTOR, dim=384) # Changed to
"vector"

And update your rag_data accordingly:

rag_data = [{"id": id_[i], "passage": passage[i], "vector": embedding[i]} for i in range(len(embeddings))]

Alternatively, you can check what the actual schema expects:

print("Collection schema:", client.describe_collection("rag_mini"))

This will show you the exact field names that the collection is expecting, and you can adjust your data dictionary keys to match.

Output Use: I made the name change from “embedding” to “vector”.
Verification: I ran the code and the bug was fixed.

Appendix E
Tool Used: Claude Opus 3
Purpose: To learn how to find the queries related to Uruguay, which was the very first output from the passages list.
Input: How do I find a query related to uruguay? I meant from my queries that were pulled from huggingface's datasets: queries = pd.read_parquet("hf://datasets/rag-datasets/rag-mini-wikipedia/data/test.parquet/part.0.parquet")

Output: I apologize for the confusion. If you have a DataFrame named queries that contains a list of queries loaded from the "rag-mini-wikipedia" dataset on Hugging Face, you can search for queries related to Uruguay using the following code:

python

```
import pandas as pd


# Assuming you have already loaded the queries DataFrame

# queries = pd.read_parquet("hf://datasets/rag-datasets/rag-mini-wikipedia/data/test.parquet/part.0.parquet")


# Search for queries related to Uruguay

uruguay_queries = queries[queries['question'].str.contains('Uruguay', case=False)]


# Print the matching queries

print("Queries related to Uruguay:")

for index, row in uruguay_queries.iterrows():
```

```python
    print(f"Query: {row['question']}")
    print(f"Answer: {row['answer']}")
    print("---")
```

Explanation:

- We assume that you have already loaded the queries DataFrame using the provided code.
- The code uses boolean indexing to filter the queries DataFrame based on the condition queries['question'].str.contains('Uruguay', case=False).
  - queries['question'] selects the 'question' column of the DataFrame.
  - .str.contains('Uruguay', case=False) checks if the string 'Uruguay' is present in each question, ignoring case sensitivity.
- The resulting uruguay_queries DataFrame contains only the rows where the question contains the word 'Uruguay'.
- We then iterate over the rows of uruguay_queries using iterrows() and print the 'question' and 'answer' for each matching query.

This code will search for queries related to Uruguay within the loaded DataFrame and display the matching questions and answers.

Make sure you have the necessary dependencies installed (pandas) and have successfully loaded the queries DataFrame before running this code.

Let me know if you have any further questions!

Output Use: I used the query search python code provided by Claude Opus 3.
Verification: I ran the code in my IDE and confirmed the output was what I wanted.

Appendix F
Tool Used: Claude Code
Purpose: To help debug the 0 exact matches from comparing all query answers.
Input: Will this affect how I compare each index of the queries_list to the queries.answer results? I got a total of 0 for exact match
Output: Yes, this will definitely affect your comparison! There are several issues with your current approach:

1. Format mismatch: queries_list contains lists like ['no'], while queries.answer contains strings like 'no'
2. Index increment: You're not incrementing index, so you're always comparing to queries_list[0]

Here's the corrected version:

```
# Extract strings from your list of lists
queries_strings = [answer[0] for answer in queries_list]  # Convert
['no'] to 'no'

counter = 0
for i, ground_truth in enumerate(queries.answer):
   if ground_truth == queries_strings[i]:
      counter += 1

print(f"Exact matches: {counter}")
print(f"Total questions: {len(queries.answer)}")
print(f"Exact Match Score: {counter / len(queries.answer):.4f}")
```

Or even simpler:

```
# Flatten your results first
flat_answers = [answer[0] for answer in queries_list]

# Count exact matches
exact_matches = sum(1 for pred, truth in zip(flat_answers,
queries.answer) if pred == truth)

print(f"Exact matches: {exact_matches}")
print(f"Exact Match Score: {exact_matches / len(queries.answer):.4f}")
```

The key fixes:
- Extract strings from your nested lists: answer[0] instead of answer
- Use proper indexing with enumerate() or zip()
- Compare the same data types (string to string)

Output Use: I used Claude’s recommended code for flattening my answers.
Verification: I ran the code in VS Code and confirmed a good output.

Appendix G
Tool Used: Claude Code
Purpose: To figure out why the number of exact matches is so low.
Input: Is getting 36 right out of 918 normal?
Output: 36 out of 918 (about 3.9% exact match) is actually quite typical for a naive RAG system, especially with the current setup you have. Here's why this score makes sense:

Reasons for low exact match:

1. No context retrieval: Your current loop generates answers without

retrieving relevant passages for each question
2. Small model: flan-t5-small has limited capabilities
3. Exact match is strict: Requires perfect string matching (case, punctuation, etc.)

This is normal for:
- Naive implementations without proper context retrieval
- Small language models
- Complex QA datasets

To improve your scores:

1. Implement proper RAG retrieval (as I suggested earlier):
```
for question in queries['question']:
   # Retrieve relevant passages for each question
   query_embedding = embedding_model.encode([question])
   search_results = client.search(collection_name="rag_mini",
data=query_embedding)
   # Use retrieved context in your prompt
```

2. Use F1 score alongside EM - F1 is more forgiving for partial matches
3. Try different passage selection strategies (top-3, top-5 passages)

Your 3.9% EM is a good baseline to improve from. Many research papers start with similar naive baselines before implementing more sophisticated techniques.

Output Use: I added in the content retrieval as per Claude's suggestion. I used my existing code to implement the retrieval tailored to all queries.
Verification: I ran the code in VS Code to confirm the output and verified Claude's assurance in the Naïve Rag's performance using this blog: https://blog.knowledgator.com/run-t5-model-on-100k-tokens-20x-faster-5ac3aa405d3e.

Appendix H
Tool Used: Claude Code
Purpose: To verify if I'm actually using the top-1 retrieval method.
Input: How do I know I'm using top-1 retrieved document to prompt?
Output: You're currently using the raw search results object as context, not the actual passage text. The client.search() returns metadata, not the readable text. You need to extract the actual passage content.

Here's how to properly use top-1 retrieved document:

```
queries_list = []
```

for row in queries.question:
 new_query_embedding = embedding_model.encode([row])
 search_results = client.search(collection_name="rag_mini",
data=new_query_embedding, limit=1) # limit=1
for top-1

 # Extract the actual passage text from top result
 top_result_id = search_results[0][0]['id'] # Get the ID of top result
 top_passage = passage[top_result_id] # Get actual passage text

 new_prompt = f"""{system_prompt} \n Context: {top_passage} \n Question: {row}"""
 all_queries_input = tokenizer(new_prompt, return_tensors="pt")
 all_queries_output = model.generate(**all_queries_input)
 queries_answer = tokenizer.batch_decode(all_queries_output,
skip_special_tokens=True)
 queries_list.append(queries_answer)

Key changes:
1. limit=1 ensures you only get the top result
2. search_results[0][0]['id'] extracts the passage ID from search results
3. passage[top_result_id] gets the actual readable text
4. Use the text as context, not the search object

To verify you're using top-1:
print(f"Using top-1 result ID: {top_result_id}")
print(f"Context preview: {top_passage[:100]}...")
Output Use: I applied the suggested code from Claude to change my code to retrieving the top-1 document.
Verification: The Exact Match calculation jumped from 39 and 41 to 100.

Appendix I
Tool Used: Claude Code
Purpose: To figure out how to address the token sequence length error.
Input: I got this message: Token indices sequence length is longer than the specified maximum sequence length for
this model (557 > 512). Running this sequence through the model will result in indexing errors
Output: This error occurs because your prompt is too long for the flan-t5-small model, which has a maximum sequence
length of 512 tokens. You need to truncate the input.

Fix by adding truncation:

queries_list = []

```
for row in queries.question:
    new_query_embedding = embedding_model.encode([row])
    search_results = client.search(collection_name="rag_mini",
data=new_query_embedding, limit=1)

    top_result_id = search_results[0][0]['id']
    top_passage = passage[top_result_id]

    new_prompt = f"""{system_prompt} \n Context: {top_passage} \n Question: {row}"""

    # Add truncation and padding
    all_queries_input = tokenizer(new_prompt, return_tensors="pt", max_length=512,
truncation=True,
 padding=True)
    all_queries_output = model.generate(**all_queries_input, max_length=50)  # Limit
output length too
    queries_answer = tokenizer.batch_decode(all_queries_output,
skip_special_tokens=True)
    queries_list.append(queries_answer)
```

Key additions:

- max_length=512 - Limits input to model's max capacity
- truncation=True - Automatically cuts off excess tokens
- padding=True - Ensures consistent input size
- max_length=50 in generate - Limits output length

Alternative - Truncate passage manually:

```
# Truncate passage to fit within token limit
top_passage = top_passage[:200]  # Keep first 200 characters
```

The truncation parameter is the cleanest solution as it handles tokenization limits automatically.

Output Use: I implemented the code recommended by Claude to truncate the sequence length.

Verification: The error message went away after I implemented this code.

Appendix J

Tool Used: Claude Code

Purpose: To discover the reason why my F1 score code was not calculating properly.

Input: Here's how I did it [my code here], what do you suggest I adjust to make it come out with the same results as your suggested code?

Output: Your approach has the right idea but several issues that will give incorrect results. Here are the key problems and fixes:

Issues:
1. You're calculating one overall F1 instead of F1 per question-answer pair
2. Wrong precision/recall calculation - you're dividing by list length instead of token counts
3. num_same gets overwritten in the loop instead of accumulated
4. Missing normalization (lowercase, punctuation removal)

Here's the corrected version:

```
# F1 Score Calculation
from collections import Counter
import string

def normalize_answer(s):
    """Normalize answer for comparison"""
    # Remove punctuation and convert to lowercase
    s = s.lower()
    s = ''.join(ch for ch in s if ch not in string.punctuation)
    return ' '.join(s.split())  # Remove extra whitespace

f1_scores = []

for pred, truth in zip(flatten_answer, queries.answer):
    # Normalize and tokenize
    pred_tokens = normalize_answer(pred).split()
    truth_tokens = normalize_answer(truth).split()

    # Handle empty cases
    if len(pred_tokens) == 0 or len(truth_tokens) == 0:
        f1_scores.append(int(pred_tokens == truth_tokens))
        continue

    # Count common tokens
    common_tokens = Counter(pred_tokens) & Counter(truth_tokens)
    num_same = sum(common_tokens.values())

    if num_same == 0:
        f1_scores.append(0)
        continue

    # Calculate precision, recall, F1 for this pair
    precision = num_same / len(pred_tokens)
```

```
    recall = num_same / len(truth_tokens)
    f1 = (2 * precision * recall) / (precision + recall)
    f1_scores.append(f1)

  # Average F1 across all questions
  avg_f1 = sum(f1_scores) / len(f1_scores)
  print(f"Average F1 Score: {avg_f1:.4f}")
```

Key changes:
1. Calculate F1 per question-answer pair, then average
2. Correct precision/recall: divide by token counts, not list lengths
3. Add normalization for fair comparison
4. Handle edge cases (empty answers)

This will give you the standard QA F1 score that matches research papers.
Output Usage: I implemented the suggested F1 score calculations since my own F1 score calculation was not working properly.
Verification: The code ran fine and I received my average F1 score.

Appendix K
Tool Used: Claude Code
Purpose: To figure out why my python module wasn’t importing properly.
Input: from naive_rag import NaiveRAG gave me an error when I tried it
Output: The error is likely because the notebook can't find the naive_rag.py file. Try these solutions:

Option 1: Add the src directory to Python path

```
import sys
sys.path.append('/Users/shoyou100/Visual-Code-Studio-workspace/assignment2-rag/src')
from naive_rag import NaiveRAG
```

Option 2: Use relative path (if notebook is in notebooks/ folder)

```
import sys
sys.path.append('../src')
from naive_rag import NaiveRAG
```

Option 3: Change working directory

```
import os
os.chdir('/Users/shoyou100/Visual-Code-Studio-workspace/assignment2-rag/src')
from naive_rag import NaiveRAG
```

Option 4: Import as module with full path

```
import importlib.util
```

```
spec = importlib.util.spec_from_file_location("naive_rag",
"/Users/shoyou100/Visual-Code-Studio-workspace/assignment2-rag/src/naive_rag.py")
naive_rag_module = importlib.util.module_from_spec(spec)
spec.loader.exec_module(naive_rag_module)
NaiveRAG = naive_rag_module.NaiveRAG
```

Option 5: Simple fix - move to src directory first

```
%cd /Users/shoyou100/Visual-Code-Studio-workspace/assignment2-rag/src
from naive_rag import NaiveRAG
```

Try Option 1 first - it's the most common solution for this path issue.

Output Use: I used Option 1.

Verification: The code worked once I added the code recommendation.

Appendix L

Tool Used: Claude Code

Purpose: To ask for a recommendation in converting my python notebook code to a function in my python module.

Input: How can I convert this [insert my code here] from the system_evaluation.ipynb to a function in the naive_rag.py?

Output: I'll add an EM score calculation method to the NaiveRAG class:

Update(src/naive_rag.py)

⎿ User rejected update to src/naive_rag.py

```
          # Step 4: Store the answer in the list
          self.queries_list.append(queries_answer)

  +     def calculate_em_score(self, ground_truths, limit=100):
  +         """Calculate Exact Match (EM) score between generated answers and ground truths."""
  +         counter = 0
  +         flatten_answer = [actual_answer[0] for actual_answer in self.queries_list]
  +
  +         for index, truth in enumerate(ground_truths):
  +             if index >= limit:
  +                 break
  +             if index < len(flatten_answer) and truth == flatten_answer[index]:
  +                 counter += 1
  +
  +         em_score = counter / min(limit, len(flatten_answer), len(ground_truths))
  +         print(f"EM Score: {counter}/{min(limit, len(flatten_answer), len(ground_truths))} = {em_score:.4f}")
  +         return em_score
  +
```

```
def sanityCheck(self, collection_name):
    print("Entity count:",
self.client.get_collection_stats(collection_name)["row_count"])
    print("Collection schema:", self.client.describe_collection(collection_name))
```

Output Use: I implemented the calculate exact method function in my python module.
Verification: I ran the code and the calculation matched my original EM calculation.

Appendix M
Tool Used: Claude Sonnet 4.5
Purpose: To learn how to incorporate chunking for advanced RAG.
Input (Part 1): So how is the LM embedding the information as a default? How large are default chunks is basically what I'm asking or is there no chunking at all?

Output: **There is NO default chunking by the embedding model** - it just embeds whatever text you give it, up to its max token limit.

How all-MiniLM-L6-v2 Works

No chunking, just a token limit:

python

```python
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')

# Case 1: Short text (under 256 tokens)
text1 = "Abraham Lincoln was the 16th president."
embedding1 = model.encode(text1)  # Embeds the whole thing → 384-dim vector

# Case 2: Long text (over 256 tokens)
text2 = "... 5000 word Wikipedia article ..."
embedding2 = model.encode(text2)  # TRUNCATES to first ~256 tokens, embeds that
```

What happens:

- Text ≤256 tokens → Embeds the entire text as one vector
- Text >256 tokens → **Silently truncates** to first 256 tokens, embeds only that

The Problem Without Chunking

If you just throw entire Wikipedia articles at the model:

- Only the **first ~200-250 words get embedded**
- The rest is **lost/ignored**
- You lose all information from the middle and end of documents

Why YOU Need to Chunk

You must chunk documents yourself BEFORE embedding:

python

```
# Without chunking (BAD for long docs)
doc = "3000 word article..."
embedding = model.encode(doc)  # Only first 256 tokens embedded!

# With chunking (GOOD)
chunks = split_into_chunks(doc, chunk_size=500)  # e.g., 6 chunks of 500 chars each
embeddings = model.encode(chunks)  # Now you get 6 vectors covering the whole doc
```

TL;DR

- **The model has no "default chunk size"** - it just has a max input (256 tokens)
- **YOU decide chunking** - common sizes: 200-1000 characters, with 50-200 overlap
- **If you don't chunk**, long documents get truncated and you lose most content

For your RAG project, you MUST implement chunking as part of your data preprocessing pipeline.

Make sense?

Input (Part 2): So instead of doing this: passage_embeddings = naive_rag.embedding_model.encode(passages['passage'].tolist()), I'm chunking the queries pandas dataframe contents instead? Not sure what text_splitter.split_text is expecting

Output (Part 2): **Yes! You should chunk the passages before embedding them.** Right now you're embedding entire passages, which might exceed the 256 token limit and get truncated.

Current Problem

python

```python
# This might truncate long passages
passage_embeddings = naive_rag.embedding_model.encode(passages['passage'].tolist())
```

If a passage is >256 tokens, only the first 256 get embedded and the rest is lost.

Solution: Chunk First

python

```python
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Create text splitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,     # Adjust based on experiments
    chunk_overlap=50    # Retain some context between chunks
)

# Chunk all passages
all_chunks = []
chunk_metadata = []  # Track which passage each chunk came from

for idx, passage_text in enumerate(passages['passage']):
    chunks = text_splitter.split_text(passage_text)
    all_chunks.extend(chunks)

    # Keep track of source passage for each chunk
    for chunk in chunks:
        chunk_metadata.append({
            'passage_id': passages['id'][idx] if 'id' in passages else idx,
            'chunk_text': chunk
        })

# Now embed the chunks (not full passages)
```

```
passage_embeddings = naive_rag.embedding_model.encode(all_chunks)
```

What split_text() Expects

Input: A single string (one passage) **Output:** A list of strings (the chunks)

python

```python
# Example
passage = "Abraham Lincoln was born in 1809. He became president in 1861. He was assassinated in 1865."

chunks = text_splitter.split_text(passage)
# Returns: ['Abraham Lincoln was born in 1809. He became president in 1861.',
#          'He became president in 1861. He was assassinated in 1865.']
```

For Your Vector Store

You'll need to adjust your indexing:

python

```python
# Instead of storing one embedding per passage
# Store multiple embeddings (one per chunk)

for i, chunk_embed in enumerate(passage_embeddings):
    vector_store.add(
        embedding=chunk_embed,
        text=chunk_metadata[i]['chunk_text'],
        metadata={'passage_id': chunk_metadata[i]['passage_id']}
    )
```

Experimenting with Chunk Sizes

For your assignment, try different values:

- chunk_size=200 (small chunks, more granular)
- chunk_size=500 (medium)
- chunk_size=1000 (larger, more context)

This is exactly the kind of experiment your professor wants!

Does this make sense now?

Output Use: I used the combined information from both queries to implement Claude's suggested chunking technique using langchain.
Verification: I confirmed that the code was valid by running it successfully.

Appendix N
Tool Used: Claude Sonnet 4.5
Purpose: To figure out why Im getting a list index out of range error when running the advanced RAG chunking feature.
Input: I implemented your code exactly and now I'm getting a list index out of range error for the following code: [My code here]

Output: The error is happening because **the number of embeddings doesn't match the number of passages**. This is because you chunked the passages, creating MORE chunks than original passages.

The Problem

python

```python
# Before chunking:
len(passages) = 3200  # Original passages

# After chunking:
len(all_chunks) = ~10000  # Way more chunks than passages!
len(passage_embeddings) = ~10000  # Embeddings for chunks

# In create_dataBase:
# Trying to access passages['id'][10000] → IndexError!
```

Solution: Track Chunk Metadata

You need to store chunk information alongside passage information. Here's the fix:

python

```python
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

```python
# Create text splitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50
)

# Chunk all passages and track metadata
all_chunks = []
chunk_metadata = []

for idx in range(len(passages)):
    passage_text = passages['passage'].iloc[idx]
    passage_id = passages['id'].iloc[idx] if 'id' in passages.columns else idx

    # Split passage into chunks
    chunks = text_splitter.split_text(passage_text)

    for chunk_idx, chunk in enumerate(chunks):
        all_chunks.append(chunk)
        chunk_metadata.append({
            'passage_id': passage_id,
            'chunk_id': f"{passage_id}_chunk_{chunk_idx}",
            'chunk_text': chunk
        })

# Embed chunks
passage_embeddings = naive_rag.embedding_model.encode(all_chunks)

print(f"Original passages: {len(passages)}")
print(f"Total chunks: {len(all_chunks)}")
print(f"Embeddings: {len(passage_embeddings)}")
```

Update Your create_dataBase Call

Now you need to pass the chunk data, not the original passages:

python

```
# Create a DataFrame-like structure for chunks
import pandas as pd

chunks_df = pd.DataFrame(chunk_metadata)

# Now call create_dataBase with chunk data
naive_rag.create_dataBase(
    corpus=chunks_df,
    corpus_id='chunk_id',  # Use chunk_id instead of passage
    embeddings=passage_embeddings,
    dim=384
)
```

Or Modify create_dataBase to Accept Lists

Alternatively, modify your create_dataBase method to work with the metadata list:

python

```
# In naive_rag.py
def create_dataBase(self, chunk_metadata, embeddings, dim):
    self.schema.add_field(field_name="id", datatype=DataType.VARCHAR, is_primary=True,
max_length=100)
    self.schema.add_field(field_name="passage_id", datatype=DataType.VARCHAR,
max_length=100)
    self.schema.add_field(field_name="chunk_text", datatype=DataType.VARCHAR,
max_length=5000)
    self.schema.add_field(field_name="vector", datatype=DataType.FLOAT_VECTOR,
dim=dim)

    self.client.create_collection(collection_name=self.collection_name, dimension=dim,
schema=self.schema)
```

```python
    rag_data = [
        {
            "id": meta['chunk_id'],
            "passage_id": str(meta['passage_id']),
            "chunk_text": meta['chunk_text'],
            "vector": embeddings[i].tolist()
        }
        for i, meta in enumerate(chunk_metadata)
    ]

    self.client.insert(collection_name=self.collection_name, data=rag_data)
    print(f"Inserted {len(rag_data)} chunks successfully.")
```

Then Call It Like This

python

```python
naive_rag.create_dataBase(
    chunk_metadata=chunk_metadata,
    embeddings=passage_embeddings,
    dim=384
)
```

TL;DR

The issue: You have **more embeddings (for chunks) than original passages**, causing the index mismatch.

The fix: Store chunk metadata (chunk_id, passage_id, chunk_text) alongside embeddings, then update create_dataBase to use this metadata instead of the original passages DataFrame.

Which approach would you prefer - modifying the method or creating a chunks DataFrame?

Output Usage: I added in the code incorporating chunk_ids and converting all chunks into a dataframe that replaces the code from Naïve RAG.

Verification: I ran the code and verified through EM and F1 scores that chunking improved the results.

Appendix O
Input: I want a fresh database everytime I create an instance of naive rag
Output: Here are a few ways to ensure a fresh database every time:

Option 1: Drop Collection in __init__ (Recommended)

Modify your NaiveRAG.__init__ method:

python

```python
def __init__(self, embedding_model_name, model_name, database_name, collection_name):
    self.embedding_model = SentenceTransformer(embedding_model_name)
    self.tokenizer = AutoTokenizer.from_pretrained(model_name)
    self.schema = MilvusClient.create_schema(auto_id=False, enable_dynamic_field=False)
    self.model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
    self.client = MilvusClient(uri=f"./{database_name}")
    self.collection_name = collection_name

    # Drop collection if it exists (fresh start)
    if self.client.has_collection(collection_name=self.collection_name):
        print(f"Dropping existing collection '{self.collection_name}'...")
        self.client.drop_collection(collection_name=self.collection_name)
        print("Collection dropped.")

    self.queries_list = []
```

Option 2: Delete Entire Database File (Nuclear Option)

python

```python
import os

def __init__(self, embedding_model_name, model_name, database_name, collection_name):
```

```
    self.embedding_model = SentenceTransformer(embedding_model_name)
    self.tokenizer = AutoTokenizer.from_pretrained(model_name)
    self.schema = MilvusClient.create_schema(auto_id=False, enable_dynamic_field=False)
    self.model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

    # Delete database file if it exists
    if os.path.exists(database_name):
        os.remove(database_name)
        print(f"Deleted existing database file '{database_name}'")

    self.client = MilvusClient(uri=f"./{database_name}")
    self.collection_name = collection_name
    self.queries_list = []
```

Option 3: Add a reset Parameter

Give yourself control over when to reset:

python

```
def __init__(self, embedding_model_name, model_name, database_name,
collection_name, reset=True):
    self.embedding_model = SentenceTransformer(embedding_model_name)
    self.tokenizer = AutoTokenizer.from_pretrained(model_name)
    self.schema = MilvusClient.create_schema(auto_id=False, enable_dynamic_field=False)
    self.model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
    self.client = MilvusClient(uri=f"./{database_name}")
    self.collection_name = collection_name

    # Optional reset
    if reset and self.client.has_collection(collection_name=self.collection_name):
        print(f"Resetting collection '{self.collection_name}'...")
        self.client.drop_collection(collection_name=self.collection_name)
```

```python
    self.queries_list = []

# Usage:
naive_rag = NaiveRAG(..., reset=True)  # Fresh start
naive_rag = NaiveRAG(..., reset=False) # Keep existing data
```

My Recommendation

Use **Option 1** (drop collection in __init__) because:

- ✅ Fast - no file deletion overhead
- ✅ Clean - only removes the specific collection
- ✅ Safe - other collections in the same DB file are preserved
- ✅ Explicit - clear what's happening

Then in your notebook:

python
```python
# This will always start fresh
naive_rag = NaiveRAG('all-MiniLM-L6-v2', 'google/flan-t5-small',
             'rag_wikipedia_mini.db', 'rag_mini')


# Now create your database
naive_rag.create_dataBase(...)
```

Which option fits your workflow best?

Appendix P
Tool Used: Claude Code
Purpose: To assist in implementing the confidence scoring feature for my enhanced RAG.
Input: I'd like to incorporate confidence scoring for the second feature in my enhanced_rag.ipynb file and enhanced_rag.py file. What do you suggest?
Output: Based on your code, I can see you're implementing chunking (Feature 1) but don't have a second feature yet. Here are my suggestions for incorporating **confidence scoring** as your second feature:
Confidence Scoring Suggestions:
1. Retrieval Confidence Score

Calculate confidence based on similarity scores from your vector search:
1 Use the cosine similarity scores from Milvus

2 Normalize scores to 0-1 range
3 Set a threshold (e.g., 0.7) below which you warn about low confidence

2. Answer Confidence Score

Combine multiple signals:

- **Semantic similarity**: Distance between query and retrieved chunks
- **Cross-encoder reranking**: Use a cross-encoder model to rescore query-context pairs
- **Answer consistency**: Generate multiple answers and check agreement
- **Retrieval score aggregation**: Average/max of top-k retrieval scores

3. Implementation Approach

For enhanced_rag.py, add:

```
def calculate_confidence(self, similarity_scores, threshold=0.7):
    """Calculate confidence from similarity scores"""
    max_score = max(similarity_scores)
    avg_score = sum(similarity_scores) / len(similarity_scores)

    confidence = {
        'max_similarity': max_score,
        'avg_similarity': avg_score,
        'confidence_level': 'high' if max_score > threshold else 'low'
    }
    return confidence
```

For the notebook, modify your search to:

- Return similarity scores from Milvus
- Calculate confidence metrics
- Include confidence in your output
- Optionally filter/flag low-confidence answers

4. Alternative: Cross-Encoder Reranking

A more sophisticated approach:

- Retrieve top-k chunks (e.g., k=10)
- Use a cross-encoder model (cross-encoder/ms-marco-MiniLM-L-6-v2) to rescore
- Select top-1 with confidence score
- This gives better relevance estimates than bi-encoder similarity alone

Would you like me to implement one of these approaches? The **retrieval confidence + cross-encoder reranking** combination would be most effective for improving both accuracy and confidence scoring.

Output Use: I implemented the method recommended for enhanced_rag.py
Verification: I ran the code and confirmed that the function was working in naïve_rag.py.

Appendix Q
Tool Used: Claude Code
Purpose: To solve the issue of why the confidence score was making the results worse.

Input: Yes but you'll likely see that there's no difference between chunking by itself and chunking with confidence scoring added.
Output: You're absolutely right! That's because **confidence scoring alone doesn't change the retrieval results** - it only provides **metadata about answer quality**.
The Real Value of Confidence Scoring:
Confidence scoring is meant to:
4 **Identify unreliable answers** - Flag low-confidence predictions
5 **Enable filtering** - Skip or handle low-confidence answers differently
6 **Provide transparency** - Tell users when the system is uncertain
To Actually Improve Performance with Feature 2:

You need to **use the confidence scores to make different decisions**. Here are actionable approaches:
Option 1: Multi-passage context aggregation
Instead of picking one passage, **combine multiple high-confidence passages**:
```
if use_enhanced:
  top_results = new_context[0][:3]

  # Get all passages with high confidence
  high_conf_passages = []
  all_confidences = []

  for result in top_results:
    result_id = result['id']
    distance = result['distance']
    passage = corpus[result_id]
    confidence = enhancedRag.calculate_confidence(distance)

    if confidence['confidence_level'] == 'high':
      high_conf_passages.append(passage)
    all_confidences.append(confidence)

  # Combine multiple passages or fallback to top-1
  if len(high_conf_passages) > 0:
    top_passage = " ".join(high_conf_passages[:2])  # Use top 2 high-conf passages
  else:
    top_passage = corpus[top_results[0]['id']]

  # Store aggregate confidence
  self.confidence_scores.append(all_confidences[0])
```
Option 2: Confidence-based answer filtering

Generate answer but flag/modify low-confidence ones:
```
if use_enhanced:
```

    # ... calculate confidence ...

    # After generating answer
    if confidence['confidence_level'] == 'low':
        queries_answer = ["[LOW CONFIDENCE] " + queries_answer[0]]

Option 3: Dynamic retrieval strategy

Retrieve more passages when confidence is low:

```
# First, check top-1 confidence
initial_result = new_context[0][0]
initial_confidence = enhancedRag.calculate_confidence(initial_result['distance'])

if initial_confidence['confidence_level'] == 'low' and top_k == 1:
    # Low confidence, retrieve more
    new_context = self.client.search(..., limit=3)
    # Then aggregate or rerank
```

My Recommendation:

Implement Option 1 - it will actually improve performance because you're giving the LLM more context when multiple passages are high-confidence. This makes your Feature 2 genuinely valuable, not just informational. Would you like me to help implement Option 1 (multi-passage aggregation)?

Output Use: I implemented option but adjusted it to be more malleable and take in k number of passages.

Verification: I ran the code in VSCode and confirmed that the EM and F1 scores at least stayed the same if not increased.

Appendix R

Tool Usage: Claude Code

Purpose: Learn how to implement RAGAS using Gemini API.

Input: I have the free tier Gemini API key and was also given starter code for implementing RAGAs. I've been thinking through what to do but already ran into an issue where the name Dataset is not defined but I have not had a reason to define Dataset in my enhanced_rag.ipynb so far. data = {

```
    "question": ... ,               # Question
    "answer": ... ,                 # Generated Answer
    "contexts": ... ,               # Context you pass in. You can just use top-1 here
    "ground_truths": ...            # Reference Answer in the dataset (Human annotated)
}

# Convert dict to dataset
```

dataset = Dataset.from_dict(data). Here's the starter code and here's the instructions for what to do next: # Pass the dataset above to the evaluate method in RAGAs

Your code here

... I am unfamiliar with using Gemini's free api key
Output: Good question! Dataset is from the Hugging Face datasets library. You need to import it. Let me help you implement the RAGAs evaluation with your Gemini API key.

Step 1: Import Dataset

Add this to your imports at the top of the notebook:

```
from datasets import Dataset
```

Step 2: Set up Gemini API for RAGAs

RAGAs needs an LLM for evaluation. With Gemini free tier:

```
import os
from langchain_google_genai import ChatGoogleGenerativeAI

# Set your Gemini API key
os.environ["GOOGLE_API_KEY"] = "your-gemini-api-key-here"

# Initialize Gemini model for RAGAs
gemini_model = ChatGoogleGenerativeAI(model="gemini-1.5-flash")
```

Step 3: Prepare your data for RAGAs

You need to collect the contexts (passages) you used during search. First, modify naive_rag.py to store contexts: Add to __init__:

```
self.contexts_list = []  # Add this
```

In your search method, after getting top_passage:

```
# Store context for RAGAs evaluation
self.contexts_list.append([top_passage])  # RAGAs expects list of contexts
```

Step 4: Create the RAGAs dataset in your notebook

```
from datasets import Dataset

# Prepare data for RAGAs (using first 100 queries)
data = {
   "question": queries.question[:100].tolist(),
   "answer": naive_rag.flatten_answer,  # Generated answers
   "contexts": naive_rag.contexts_list,  # Retrieved passages (list of lists)
   "ground_truths": [[truth] for truth in queries.answer[:100].tolist()]  # Wrap in lists
}

# Convert dict to dataset
ragas_dataset = Dataset.from_dict(data)
print(ragas_dataset)
```

Step 5: Run RAGAs evaluation with Gemini

```
from ragas import evaluate
from ragas.metrics import (
```

```
    faithfulness,
    answer_relevancy,
    context_recall,
    context_precision,
)

# Evaluate using Gemini
results = evaluate(
    ragas_dataset,
    metrics=[
        faithfulness,
        answer_relevancy,
        context_recall,
        context_precision,
    ],
    llm=gemini_model,
)

print("RAGAs Evaluation Results:")
print(results)
```

Complete Code Block for your notebook:

```
# Import Dataset
from datasets import Dataset
import os
from langchain_google_genai import ChatGoogleGenerativeAI

# Set up Gemini API
os.environ["GOOGLE_API_KEY"] = "YOUR_GEMINI_API_KEY_HERE"
gemini_model = ChatGoogleGenerativeAI(model="gemini-1.5-flash")

# Prepare data for RAGAs
data = {
    "question": queries.question[:100].tolist(),
    "answer": naive_rag.flatten_answer,
    "contexts": naive_rag.contexts_list,
    "ground_truths": [[truth] for truth in queries.answer[:100].tolist()]
}

# Convert to dataset
ragas_dataset = Dataset.from_dict(data)

# Evaluate
results = evaluate(
```

```
    ragas_dataset,
    metrics=[faithfulness, answer_relevancy, context_recall, context_precision],
    llm=gemini_model,
)

print("RAGAs Results:")
print(results)
```

Important: You'll need to update naive_rag.py to track contexts_list first, otherwise you won't have the passages to pass to RAGAs. Would you like me to help you modify naive_rag.py to store the contexts?

Output Usage: I implemented the code recommended.

Verification: I ran the code and it worked.