Optimization of Search Performance in BreastCare Trial using Trulens

Overview

In this report, I aim to measure and optimize the search performance of my project by focusing on key metrics such as:

- **Context Relevance**
- **Latency**
- **Total Cost**

To achieve this, I conducted multiple experiments involving adjustments to prompt engineering, chunk sizes, chunk overlap, and retrieval limits. The experiments were conducted using the following question:

"What is the D3L-001 trial, and what outcomes does it aim to achieve?"

Experiment 1

Settings:

- Chunk size: 2000
- Chunk overlap: 800

```

class text_chunker:

    def process(self, pdf_text: str):

        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size = 2000, #Adjust this as you see fit
            chunk_overlap  = 800, #This let's text have some form of overlap. Useful for keeping chunks contextual
            length_function = len
        )
```

Prompt Used:

```
    @instrument
    def generate_completion(self, query: str, context_str: list) -> str:
        """
        Generate answer from context.
        """
        prompt = f"""
        You are an intelligent assistant specialized in breast cancer clinical trials.
        Your responses should focus on trial information, eligibility requirements, and next steps.
    <user_query>{query}</user_query>
         Context: {context_str}
         Question:
         {query}
         Answer:
        """
        return Complete("mistral-large2", prompt)
```

Results: In this configuration, the metrics (context relevance, latency, and cost) were not optimal. The app's performance needed improvement, prompting further experimentation.

Python as cell38

```
session.get_leaderboard()
```

app_name	app_version	Answer Relevance	Context Relevance	latency	total_cost
BreastCareTrial	simple	1	0.6667	18.5886	4.1817
BreastCareTrial Retriever	base	None	1	0.4598	0

Experiment 2:

Settings:

- Chunk size: 1500
- Chunk overlap: 400

```
class text_chunker:

    def process(self, pdf_text: str):
        # Adjusted chunk size and overlap
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size = 1500,  # Lower chunk size for better latency
            chunk_overlap = 400,  # Moderate overlap to maintain context
            length_function = len
        )

        chunks = text_splitter.split_text(pdf_text)
        df = pd.DataFrame(chunks, columns=['chunks'])
```

Prompt:

```
        Generate answer from context.
        """
        prompt = f"""
        You are an intelligent assistant specialized in breast cancer clinical trials.
        Your responses should focus on trial information, eligibility requirements, and next steps.
    <user_query>{query}</user_query>
          Context: {context_str}
          Question:
          {query}
          Answer:
        """
        return Complete("mistral-large2", prompt)

    @instrument
    def query(self, query: str) -> str:
        context_str = self.retrieve_context(query)
        return self.generate_completion(query, context_str)
```

Results: This experiment yielded more poor results than Experiment 1. Further experimentation was necessary to achieve the desired performance.

Python as cell38

```
session.get_leaderboard()
```

app_name	app_version	Answer Relevance	Context Relevance	Groundedness	latency	total_cost
BreastCareTrial	simple	1	0.5	1	10.4623	1.6566
BreastCareTrial Retriever	base	None	1	None	0.273	0

Experiment 3:

Settings:

- Chunk size: 1180
- Chunk overlap: 350
- Retrieval limit: 3 (previously 4)

```

class text_chunker:

    def process(self, pdf_text: str):
        # Adjusted chunk size and overlap
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size = 1180,  # Lower chunk size
            chunk_overlap = 350,  # Moderate overlap to maintain context
            length_function = len
        )

        chunks = text_splitter.split_text(pdf_text)
        df = pd.DataFrame(chunks, columns=['chunks'])

        yield from df.itertuples(index=False, name=None)
$$;

```

Prompt:

```python
from trulens.apps.custom import instrument
from snowflake.cortex import Complete


class RAG:

    def __init__(self):
        self.retriever = CortexSearchRetriever(snowpark_session=snowpark_session, limit_to_retrieve=3)

    @instrument
    def retrieve_context(self, query: str) -> list:
        """
        Retrieve relevant text from vector store.
        """
        return self.retriever.retrieve(query)

    @instrument
    def generate_completion(self, query: str, context_str: list) -> str:
        """
        Generate answer from context.
        """
        prompt = f"""
        You are an expert assistant specializing in breast cancer clinical trials. Provide accurate answers strictly f
        - Trial objectives, phases, and descriptions.
        - Eligibility criteria and exclusion details.
        - Trial locations, investigator contacts.

        If the context does not contain the required information, respond with:
        "I'm sorry, the provided context does not contain the information for your query."
        Context: {context_str}
          Question:
          {query}
          Answer:
        """
        return Complete("mistral-large2", prompt)
```

Results: By limiting the retrieved context to three chunks, this experiment achieved noticeable reductions in latency and cost. However, the relevance could still be improved.

Python as cell38

```python
session.get_leaderboard()
```

app_name	app_version	Answer Relevance	Context Relevance	Groundedness	latency	total_cost
BreastCareTrial	simple	1	0.8411	0.9992	3.9545	0.4944
BreastCareTrial Retriever	base	None	1	None	0.2376	0

Experiment 4:

Settings:

- Chunk size: 1200
- Chunk overlap: 350
- Retrieval limit: 3

```python

    def process(self, pdf_text: str):
        # Adjusted chunk size and overlap
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size = 1200,  # Lower chunk size
            chunk_overlap = 350,  # Moderate overlap to maintain context
            length_function = len
        )

```

Prompt:

```python
    def __init__(self):
        self.retriever = CortexSearchRetriever(snowpark_session=snowpark_session, limit_to_retrieve=3)

    @instrument
    def retrieve_context(self, query: str) -> list:
        """
        Retrieve relevant text from vector store.
        """
        return self.retriever.retrieve(query)

    @instrument
    def generate_completion(self, query: str, context_str: list) -> str:
        """
        Generate answer from context.
        """
        prompt = f"""
        You are an intelligent assistant specialized in breast cancer clinical trials.
        Your responses should focus on trial information, eligibility requirements, and next steps.
        Context: {context_str}
          Question:
          {query}
          Answer:
        """
        return Complete("mistral-large2", prompt)
```

Results: This experiment provided the best balance of:

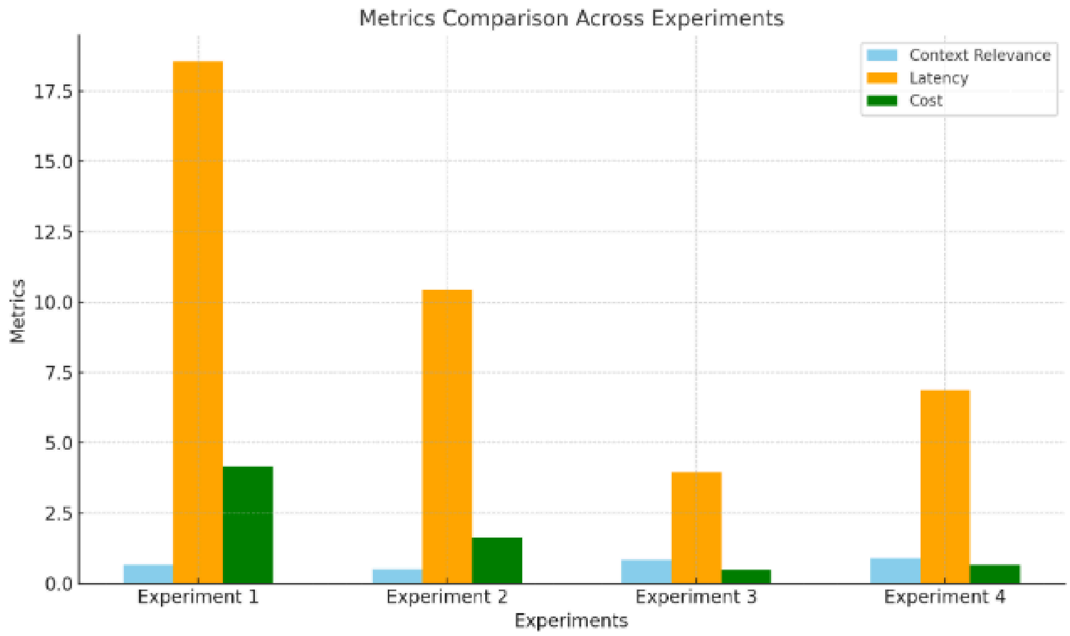
- **Context Relevance**
- **Latency**
- **Cost Efficiency**

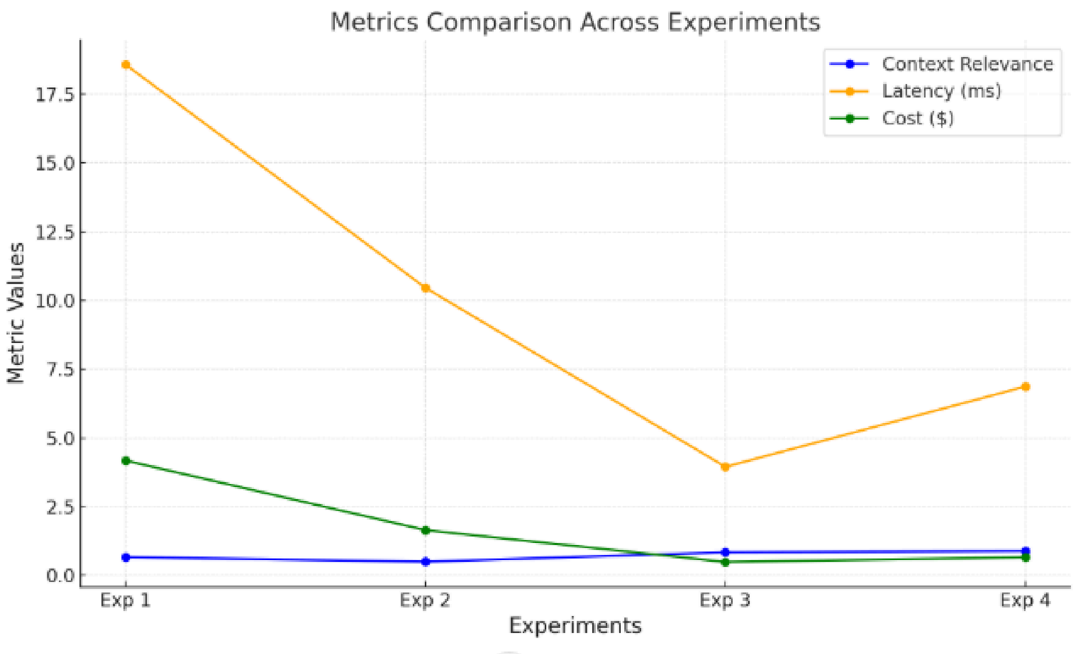
Experiment 4 is the most optimal configuration achieved during this analysis.

Python as cell38 0.0s

```python
session.get_leaderboard()
```

app_name	app_version	Answer Relevance	Context Relevance	latency	total_cost
BreastCareTrial	simple	1	0.8889	6.8685	0.666
BreastCareTrial Retriever	base	None	1	0.2945	0





Conclusion

Based on the metrics analyzed, Experiment 3 and Experiment 4 show the most promising results.

- **Experiment 3** demonstrates a strong balance between high context relevance (0.84), low latency (3.95 seconds), and a minimal cost (0.49), making it highly efficient for performance optimization.
- **Experiment 4** further improves context relevance slightly (0.89) while maintaining reasonable latency (6.87 seconds) and cost (0.66).