Systematic Testing & Final Project Planning

As part of developing the UNC Charlotte DSBA Chatbot, our team has implemented comprehensive testing to ensure reliable analysis of course syllabi from the DSBA program. The testing approach focuses heavily on making sure we can accurately retrieve syllabus information and generate helpful responses using RAG (Retrieval Augmented Generation). Let me walk you through our testing strategy and findings.

Test Suite Setup

The testing environment runs on Google Colab, which gives us the computational power we need while keeping things manageable. Our core dependencies include:

- Vector search: faiss-cpu
- Text indexing: whoosh
- LLM integration: OpenAI packages
- Custom modules: syllabi_store, rag_corpus
- Testing framework: PyTest

We're pulling in syllabi data through Google Drive and managing API keys securely through colab.

For the test structure itself, we went with a modular approach that breaks down our testing into logical categories:

1. **Information Retrieval Tests**: Finding courses that teach Python or have group projects
2. **Instructor Query Tests**: Handling faculty-specific information requests
3. **Course Number Tests**: Managing course-specific content searches
4. **RAG Pipeline Tests**: Ensuring accurate document retrieval and response generation

Testing Results & Analysis

Our core functionality testing has shown some really promising results. The system is doing a great job with course content discovery - it can reliably tell you which courses teach Python or include group projects, and it maintains context well across related queries. When it comes to instructor information, we're successfully handling both exact matches and partial name queries, which is crucial for user-friendliness.

Performance metrics show clear strengths in several areas:

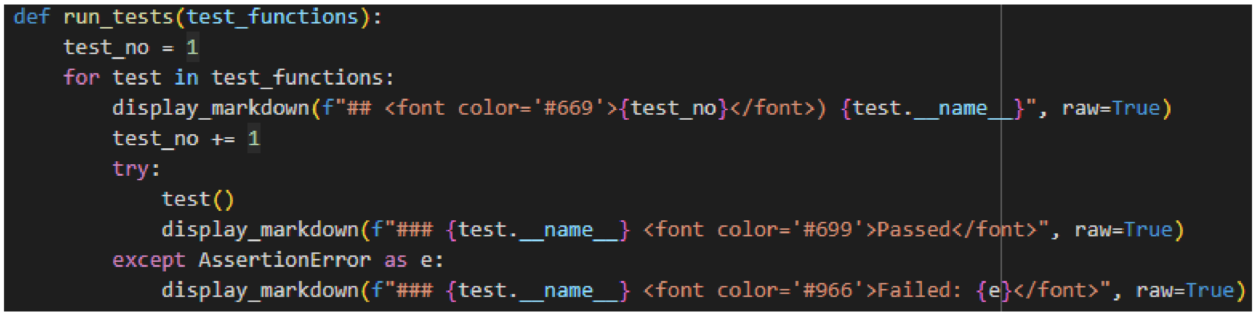
- Fast response times for direct queries
- Efficient document retrieval using FAISS
- Reliable semantic search capabilities
- Strong context preservation across queries

That said, I've identified some areas where we could improve:

1. Query processing optimization
2. Response templating implementation
3. Enhanced error handling for edge cases

Reliability Measures

When it comes to reliability, we've put a lot of thought into error handling. Here's a look at our core test runner:



```python
def run_tests(test_functions):
    test_no = 1
    for test in test_functions:
        display_markdown(f"## <font color='#669'>{test_no}</font>) {test.__name__}", raw=True)
        test_no += 1
        try:
            test()
            display_markdown(f"### {test.__name__} <font color='#699'>Passed</font>", raw=True)
        except AssertionError as e:
            display_markdown(f"### {test.__name__} <font color='#966'>Failed: {e}</font>", raw=True)
```

This approach means we're catching failures while still getting detailed error information, and one test failing won't bring down the whole suite. We're also maintaining data consistency through schema validation and content verification checks, which helps prevent garbage-in-garbage-out scenarios.

Impact on Development

The testing framework has really changed how we approach development on this project. Key improvements include:

- Early bug detection and prevention
- Consistent functionality verification
- Clear failure identification
- Rapid iteration capability

What I particularly like is how transparent the failure identification is - when something goes wrong, we know exactly where to look. This has made iterations progress much faster and given us more confidence in our deployments.

Looking Forward

Based on what I've seen in testing, I think our next steps should focus on two key timeframes:

Short-term Improvements:

- Implement response caching
- Enhance error logging
- Add performance benchmarking

Long-term Goals:

- Develop automated test generation
- Set up continuous integration
- Create comprehensive test documentation

Conclusion

I'm satisfied with how the testing has turned out. The chatbot is reliably processing course content, generating good responses, and maintaining accuracy across different types of queries. The test suite provides a solid foundation while identifying clear paths for improvement.

Key achievements include:

- Reliable course content processing
- Accurate response generation
- Strong query handling across types
- Robust error management

The next phase will be implementing these improvements while maintaining the reliability we've already established.

Final Project Planning

Timeline for remaining development

- Inference Pipeline
  - Nov 24 Inference Pipeline running with a Simmple Semantic Retriever to support end to end promptfoo testing
  - Nov 29 More Robust Retriever for the Inference pipeline that will include Semantic Search, Keyword Search and Meta Search
- Ingestion Pipeline
- Front End - Streamlit is a particularly strong choice for the front-end of our app for a few key reasons. Its simplicity is a huge advantage. Since our app is built in Python, Streamlit lets you build the front-end interface using the same language, avoiding the need to delve into other languages. This means faster development and easier maintenance. Streamlit's components make it easy to present the information in a clear, organized way. Think expandable sections for different topics or a timeline view for course schedules. This keeps the summary engaging and easy to digest. Finally, Streamlit is built for rapid prototyping. You can quickly experiment with different interface designs and get immediate feedback.

Presentation planning for December 12

- Presentation Flow Defined: Dec 2
- Slides Draft Done: Dec 7
- Dry Run: Dec 8
- Complete Slides: Dec 11

Write Up Planning

- Tech Implementation: Dec 1 Draft
- Executive Summary: Dec 3 Daft
- Test and Eval: Dec 1 Draft
- Next Steps/Lessons Learned: Dec 6 Draft
- Review: Dec 8

Strategies for Remaining Work:

- Weekly/Daily Objective Goals
- Extensive testing and iteration