AI Fact-Checker Bot

AI Fact-Checker Bot Project Assignment

Overview

Build an intelligent fact-checking bot that uses advanced AI techniques including prompt chaining, web search capabilities, and structured reasoning to verify claims and provide accurate information.

Learning Objectives

- Master LangChain framework for building AI applications
- Implement prompt chaining techniques for complex reasoning
- Integrate web search tools for real-time information retrieval
- Design effective prompting strategies for fact verification
- Create user-friendly interfaces with Streamlit or Gradio
- Apply critical thinking and verification methodologies in AI systems

Project Requirements

Core Features (Required - 70 points)

1. LangChain Integration (20 points)

- Use LangChain chat models (OpenAI GPT, Anthropic Claude, or open-source alternatives)
- Implement proper API key management and error handling
- Configure model parameters (temperature, max tokens, etc.)

2. Prompt Chaining Implementation (25 points)

Your bot must implement the following chain:

1. **Initial Response**: Generate preliminary answer to user's question/claim
2. **Assumption Extraction**: Identify key assumptions made in the initial response
3. **Verification Loop**: For each assumption, determine if it's true/false/uncertain
4. **Evidence Gathering**: Use web search to find supporting/contradicting evidence
5. **Final Synthesis**: Generate refined answer incorporating verification results

3. Web Search Integration (15 points)

- Integrate search tools (DuckDuckGo, SerpAPI, or similar)
- Parse and process search results effectively
- Handle rate limiting and API errors gracefully

4. User Interface (10 points)

Choose one implementation:

- **Streamlit**: Web-based interface with chat history and interactive elements
- **Gradio**: Simple web interface with input/output components
- **CLI**: Command-line interface with clear formatting and progress indicators

Advanced Features (Choose 2 for full credit - 20 points)

A. Source Credibility Assessment (10 points)

- Evaluate reliability of information sources
- Assign credibility scores based on domain authority, publication date, etc.
- Display source quality indicators to users

B. Claim Classification (10 points)

- Categorize claims as: Factual, Opinion, Mixed, Unverifiable
- Use classification prompts or fine-tuned models
- Provide appropriate responses based on claim type

C. Evidence Synthesis (10 points)

- Aggregate information from multiple sources
- Identify conflicting information and present balanced views
- Generate confidence scores for fact-check results

D. Interactive Clarification (10 points)

- Ask follow-up questions when claims are ambiguous
- Allow users to provide additional context
- Refine searches based on user feedback

E. Fact-Check History & Analytics (10 points)

- Store previous fact-checks in a database
- Provide analytics on common claim types
- Allow users to revisit previous checks

Bonus Features (5 points each)

- Multi-language support
- Voice input/output integration
- Batch fact-checking from uploaded documents
- Integration with social media APIs for real-time claim detection
- Custom knowledge base integration

Technical Specifications

Required Libraries

```
langchain≥0.1.0
langchain-community
streamlit≥1.25.0  # if using Streamlit
gradio≥3.40.0     # if using Gradio
python-dotenv
requests
beautifulsoup4
duckduckgo-search  # or your preferred search tool
```

Project Structure

```
fact_checker_bot/
├── src/
│   ├── __init__.py
│   ├── fact_checker.py      # Main fact-checking logic
│   ├── prompt_chains.py     # Prompt templates and chains
│   ├── search_tools.py      # Web search integration
│   ├── utils.py             # Helper functions
│   └── ui/
│       ├── streamlit_app.py # Streamlit interface
│       ├── gradio_app.py    # Gradio interface
│       └── cli.py           # Command-line interface
├── config/
│   ├── prompts.yaml         # Prompt templates
│   └── settings.py          # Configuration settings
├── tests/
│   ├── test_fact_checker.py
│   └── test_search_tools.py
├── examples/
│   ├── example_queries.txt
│   └── demo_notebook.ipynb
├── requirements.txt
├── .env.example
├── README.md
└── main.py                  # Entry point
```

Implementation Guidelines

1. Prompt Engineering Best Practices

- Use clear, specific instructions in prompts
- Implement few-shot examples for better performance
- Design prompts that encourage step-by-step reasoning
- Include safety instructions to avoid harmful content

2. Error Handling

- Implement comprehensive try-catch blocks
- Provide meaningful error messages to users
- Include fallback mechanisms for API failures
- Log errors for debugging purposes

3. Performance Optimization

- Implement caching for repeated queries
- Use async operations where possible
- Optimize search query construction

- Limit unnecessary API calls

4. Code Quality

- Follow PEP 8 style guidelines
- Include comprehensive docstrings
- Write unit tests for core functions
- Use type hints throughout the codebase

Test Cases

Your bot should handle these example scenarios:

1. **Simple Factual Claim**: "The capital of France is Paris."
2. **Common Misconception**: "What type of mammal lays the biggest eggs?"
3. **Recent Event**: "What happened in the latest SpaceX launch?"
4. **Controversial Topic**: "Is climate change caused by human activities?"
5. **Ambiguous Claim**: "Python is the best programming language."

Evaluation Rubric

Criteria	Excellent (A)	Good (B)	Satisfactory (C)	Needs Work (D/F)
LangChain Implementation	Proper use of chains, agents, and tools	Good integration with minor issues	Basic implementation works	Significant technical problems
Prompt Engineering	Sophisticated, well-designed prompts	Effective prompts with good results	Functional prompts	Poor prompt design affects results
Search Integration	Seamless web search with smart result processing	Good search integration	Basic search functionality	Search feature unreliable
User Interface	Intuitive, polished interface	Good UX with minor issues	Functional interface	Poor usability
Code Quality	Clean, well-documented, tested code	Good structure with documentation	Functional code	Poor code quality
Accuracy & Reliability	Consistently accurate fact-checking	Generally reliable results	Mostly accurate	Frequent errors

Submission Requirements

Documentation (Required)

1. **README.md** with setup instructions and usage examples
2. **Technical Report** (2-3 pages) explaining:
   - Architecture and design decisions
   - Prompting strategies used
   - Challenges encountered and solutions

- Testing methodology and results

3. **Demo Video** (5 minutes max) showing your bot in action

Code Submission

- Complete source code with proper structure
- Requirements.txt file
- Environment variable template (.env.example)
- Test files demonstrating functionality

Presentation (10 minutes)

- Live demonstration of your fact-checker
- Explanation of technical approach
- Discussion of results and limitations
- Q&A session

Timeline & Milestones

- **Week 1:** Project setup, LangChain basics, initial prompt design
- **Week 2:** Implement core prompt chaining logic
- **Week 3:** Integrate web search tools, build UI
- **Week 4:** Add advanced features, testing, documentation
- **Week 5:** Final testing, presentation preparation

Academic Integrity

- All code must be original or properly attributed
- You may use online resources and documentation
- Collaboration is encouraged, but each student submits individual work
- Clearly cite any external libraries, prompts, or code snippets used

Resources

- LangChain Documentation
- Streamlit Documentation
- Gradio Documentation
- Prompt Engineering Guide

Good luck building your fact-checker bot! Focus on creating a system that not only works technically but also provides genuine value in combating misinformation.