Aspect-Based Sentiment Analysis Project

Course: Applied Computer Science – Data & AI 6
Team Size: 2 students
Deadline: 2025-10-28 23:59h

Project Overview

In this project, you will develop a comprehensive Aspect-Based Sentiment Analysis (ABSA) system that can identify aspects (features, attributes) mentioned in text and determine the sentiment expressed toward each aspect. You will implement three different approaches and expose them through a unified API.

What is Aspect-Based Sentiment Analysis?

Unlike traditional sentiment analysis that determines overall sentiment, ABSA identifies specific aspects mentioned in text and the sentiment toward each. For example:

- **Input:** "The pizza was delicious but the service was terrible."
- **Output:**
    - Aspect: "pizza" → Sentiment: POSITIVE
    - Aspect: "service" → Sentiment: NEGATIVE

Learning Objectives

By completing this project, you will:

- Understand different approaches to sentiment analysis (rule-based, ML-based, and LLM-based)
- Gain hands-on experience with NLP libraries and frameworks
- Learn to work with pre-trained transformer models
- Explore local LLM deployment and prompting strategies
- Design and implement a unified API interface
- Compare and evaluate different AI/ML approaches

Project Requirements

Implementation 1: Lexicon-Based Approach with spaCy

Implement a classic NLP solution using:

- **spaCy** for text processing (tokenization, POS tagging, dependency parsing)
- A **sentiment lexicon** (e.g., VADER, SentiWordNet, or AFINN)

Approach Options: You may choose either direction:

1. **Aspect → Opinion:** Identify aspects first (using POS patterns, noun extraction), then find opinion words related to them using dependency parsing
2. **Opinion → Aspect:** Identify opinion/sentiment words first (using lexicon), then find the aspects they modify using dependency relations

Required Components:

- Text preprocessing pipeline based on spaCy
- Aspect extraction using linguistic rules
- Opinion word identification using sentiment lexicon
- Sentiment aggregation for each aspect
- Handling of negations and modifiers

Implementation 2: Pre-trained ABSA Model

Use a pre-trained transformer-based ABSA model from Hugging Face.

Recommended Model:

- yangheng/deberta-v3-base-absa-v1.1 (*or similar ABSA-specific model*)

Required Components:

- Model loading and initialization
- Input preprocessing for the model's expected format
- Post-processing of model predictions
- Proper handling of aspect-sentiment pairs

Note: Research the model's input format (some ABSA models expect aspect terms as part of the input, while others extract them automatically).

Implementation 3: Local LLM with Ollama

Implement ABSA using a locally running Large Language Model through Ollama.

Requirements:

- Use **Ollama** to run a local LLM (e.g., Llama 3, Mistral, or similar)
- Design effective prompts for aspect extraction and sentiment classification
- Parse structured output from the LLM (consider using JSON output formatting)
- Implement error handling for LLM responses

Considerations:

- Prompt engineering is crucial for consistent results
- Consider few-shot prompting with examples
- Balance between model size and performance
- Implement retry logic for failed generations

Unified API Design

All three implementations must conform to a single, unified API interface. This enables easy comparison and switching between implementations.

Required API Structure

```
class ABSAAnalyzer:
    """Base class/interface for all ABSA implementations"""

    def analyze(self, text: str) -> List[AspectSentiment]:
        """
        Analyze text and extract aspect-sentiment pairs
        Args:
            text: Input text to analyze
        Returns:
            List of AspectSentiment objects
        """
        raise NotImplementedError

class AspectSentiment:
```

```
"""Data class for aspect-sentiment pairs"""
aspect: str          # The aspect/feature mentioned
sentiment: str       # Sentiment label: 'positive', 'negative', 'neutral'
confidence: float    # Confidence score (0.0 to 1.0)
text_span: tuple     # Optional: (start, end) position in original text
```

Implementation Classes

Create three concrete implementations:

- LexiconABSA(ABSAAnalyzer)
- ML_ABSA(ABSAAnalyzer)
- LLMABSA(ABSAAnalyzer)

Technical Requirements

Dependencies

- Python 3.8+
- spaCy with English language model (en_core_web_sm or larger)
- transformers and torch (PyTorch)
- ollama-python
- A sentiment lexicon library (e.g. vaderSentiment, nltk, or similar)
- Additional libraries as needed (pandas, numpy, etc.)

File Organization

```
project/
├── README.md
├── requirements.txt
├── doc/
│   └── report.pdf           # Unit tests
├── notebooks/
│   ├── exploration.ipynb    # Data exploration
│   └── comparison.ipynb     # Model comparison
├── data/
│   ├── test_samples.json
│   └── evaluation_data.json (optional)
├── src/
│   ├── __init__.py
│   ├── base.py              # Base classes and interfaces
│   ├── lexicon_absa.py      # Implementation 1
│   ├── transformer_absa.py  # Implementation 2
│   ├── llm_absa.py          # Implementation 3
│   └── utils.py             # Helper functions
└── tests/
    └── test_absa.py         # Unit tests
```

Deliverables

0. Single ZIP file uploaded in Canvas

- Files organized in the structure shown above
- Don't include your complete development environment artifacts!

1. Source Code

- Well-documented, clean code following Python best practices
- All three implementations working correctly
- Unified API properly implemented
- Unit tests for core functionality

2. Documentation (README.md)

Include:

- Project overview and setup instructions
- Dependencies and installation guide
- Usage examples for each implementation
- API documentation

- Design decisions and rationale

3. Demonstration Notebook

Create a Jupyter notebook that:

- Demonstrates all three implementations on sample data
- Compares their outputs on the same inputs
- Analyzes strengths and weaknesses of each approach
- Includes performance metrics (accuracy, speed, etc.)

4. Report (PDF, 6-10 pages)

Required Sections:

1. **Introduction:** Problem description and project goals
2. **Methodology:** Detailed explanation of each implementation
3. **Design Decisions:** Why certain choices were made
4. **Experimental Setup:** Data, evaluation metrics, test scenarios
5. **Results & Analysis:**
    - Qualitative comparison (examples)
    - Quantitative comparison (if evaluation data available)
    - Performance metrics (speed, resource usage)
6. **Discussion:**
    - Strengths and weaknesses of each approach
    - Use case recommendations
    - Challenges faced and solutions
7. **Conclusion:** Key learnings and potential improvements

Evaluation Rubric

Total: 100 points

Component	Points
Implementation 1: Lexicon-Based ABSA	20
Implementation 2: Transformer-Based ABSA	20
Implementation 3: LLM-Based ABSA	20
Code Quality (API, Architecture, Testing)	20
Analysis & Comparison Report	20
TOTAL	**100**

1. Implementation 1: Lexicon-Based ABSA (20 points)

Points	Criteria
18-20	Correctly extracts aspects and sentiments; sophisticated use of spaCy (dependency parsing, POS tags); well-integrated lexicon with proper negation handling; works on complex sentences
14-17	Extracts aspects and sentiments with minor issues; good use of spaCy features; lexicon properly integrated; basic negation handling
10-13	Basic extraction works; uses spaCy for tokenization/POS; basic lexicon lookup; struggles with complex cases
0-9	Incomplete, frequently incorrect, or doesn't work; minimal use of required tools

2. Implementation 2: Transformer-Based ABSA (20 points)

Points	Criteria
18-20	Properly loads and uses pre-trained model; correct input/output formatting; excellent preprocessing and parsing; handles edge cases; shows clear understanding of transformers
14-17	Model works correctly with minor issues; good preprocessing; proper output handling; demonstrates understanding

Points	Criteria
10-13	Model runs but with suboptimal setup; basic preprocessing; output parsing works for simple cases
0-9	Model fails, produces poor results, or doesn't integrate properly; limited understanding

3. Implementation 3: LLM-Based ABSA (20 points)

Points	Criteria
18-20	Excellent prompt engineering with consistent structured output; properly configured Ollama; robust parsing of LLM responses; handles errors gracefully; uses few-shot examples effectively
14-17	Good prompts with mostly reliable output; Ollama works correctly; good output parsing with minor edge case issues
10-13	Basic prompts work inconsistently; basic Ollama integration; parsing works for well-formed output only
0-9	Poor prompts with unreliable results; Ollama not properly configured; fragile or broken parsing

4. Code Quality: API, Architecture & Testing (20 points)

Points	Criteria
18-20	Clean, well-designed unified API; all implementations conform perfectly; excellent project structure; comprehensive error handling; unit tests with good coverage; follows best practices; clean requirements.txt
14-17	Good interface design; minor inconsistencies between implementations; good organization; adequate error handling; basic tests present; generally good code style
10-13	Basic unified interface works; some inconsistencies; acceptable structure; minimal error handling; few or basic tests; inconsistent style
0-9	Poor or inconsistent API; major design issues; disorganized code; missing error handling; no tests; poor code quality

5. Analysis & Comparison Report (20 points)

Points	Criteria
18-20	Insightful comparison with rich examples; comprehensive metrics (accuracy, speed, resources); deep understanding of trade-offs; clear recommendations for use cases; considers limitations; well-written and organized
14-17	Good comparison with relevant examples; clear metrics presented; good understanding of trade-offs; reasonable recommendations; clear writing
10-13	Basic comparison present; some metrics provided; basic understanding of differences; adequate writing
0-9	Superficial or missing analysis; poor or missing metrics; limited critical thinking; poorly written

---

Penalty Conditions

- **Late Submission**: 0
- **Missing Implementation**: -20 points per missing implementation
- **Non-functional Code**: -15 points if code doesn't run without fixes
- **Plagiarism**: 0 for the assignment + integrity proceedings

Test Dataset Suggestions

Use review data from domains like:

- Restaurant reviews (e.g., Yelp dataset)
- Product reviews (e.g., Amazon reviews)
- Laptop/phone reviews (SemEval ABSA datasets)

Your test set should include:

- Simple cases (single aspect, clear sentiment)
- Complex cases (multiple aspects, mixed sentiments)
- Edge cases (sarcasm, implicit aspects, etc.)

Tips for Success

1. **Start Simple:** Get each implementation working on basic cases before handling complex scenarios
2. **Iterative Development:** Build incrementally and test frequently
3. **Prompt Engineering:** For the LLM approach, spend time crafting and refining prompts
4. **Error Handling:** All implementations should handle edge cases gracefully
5. **Performance:** Consider efficiency, especially for the LLM approach
6. **Documentation:** Document as you code, not at the end
7. **Team Coordination:** Divide work clearly but ensure both members understand all implementations

Resources

- spaCy Documentation: https://spacy.io
- Hugging Face Transformers: https://huggingface.co/docs/transformers
- Ollama Documentation: https://ollama.ai
- VADER Sentiment: https://github.com/cjhutto/vaderSentiment
- SemEval ABSA tasks: Papers and datasets on aspect-based sentiment analysis

Submission

Submit via [platform] by [deadline]:

1. Source code (GitHub repository link or ZIP file)
2. Report (PDF)
3. Presentation slides (PDF)
4. Demo video (optional but recommended, max 5 minutes)

Academic Integrity

- This is a team project, but collaboration between teams is not allowed
- You may use online resources and libraries, but cite your sources
- Do not copy code without attribution
- Pre-trained models are allowed and encouraged