**Methodology Chapter: Sentiment Dashboard Web Application**

**1. Research Design & Methodology**

This project adopted a structured software development methodology with a focus on creating a sentiment analysis dashboard tailored for content creators. The primary aim was to bridge the gap between traditional sentiment analysis tools, which are typically geared toward financial, health, or customer support sectors, and the needs of content creators seeking direct, real-time insight into public reception of their social media content.

The research methodology combines both software engineering principles and aspects of experimental design. On one hand, the implementation of a Natural Language Processing (NLP) model necessitated exploration, tuning, and evaluation akin to a research experiment. On the other hand, the end product required a user-friendly, scalable, and visually intuitive full-stack web application. Hence, the methodology comprised iterative development, rapid prototyping, and repeated testing.

The project followed an Agile-inspired methodology, using bi-weekly sprint cycles. This allowed for the frequent refinement of features, particularly the sentiment analysis module and visualization components. Agile was favored over Waterfall due to the non-linear nature of ML integration, where feature performance and usability can be evaluated only after functional implementation. The Agile methodology also supported ongoing interaction with the supervisor, enabling timely feedback and necessary redirection.

The methodology was divided into four key phases. The Foundation Phase focused on requirements gathering, technology evaluation, and early experimentation with sentiment models. The Core Development Phase emphasized backend setup using Flask and integration of the chosen CardiffNLP RoBERTa model. During the Frontend Phase, user interfaces, charts, and authentication mechanisms were built and tested. Finally, the Integration and Testing Phase involved system-level testing, performance tuning, and initial deployment. Feedback collected at each phase informed adjustments and iterative improvements.

**2. Tools, Technologies & Resources**

The development stack was chosen based on compatibility, efficiency, and ease of integration with NLP and visualization libraries. For the backend, Python was used with the Flask framework due to its lightweight routing system and seamless integration with NLP libraries such as Hugging Face’s Transformers. SQLAlchemy was employed as the ORM for handling interactions with Supabase, a PostgreSQL-as-a-service platform that provided cloud-hosted database access and real-time features. Environment configurations were managed securely using Python-dotenv.

On the frontend, the application used HTML5 and CSS3 for structure and styling. JavaScript was used for interactive behavior and client-side logic. Chart.js was the visualization tool of choice, due to its ease of use, variety of chart types, and real-time data rendering capabilities. Bootstrap was utilized to implement responsive design across mobile, tablet, and desktop screens.

For sentiment analysis, the CardiffNLP Twitter RoBERTa model was selected. It was particularly well-suited for analyzing social media text, outperforming general-purpose models like BERT on informal, brief, and slang-heavy content. PyTorch was required as the deep learning framework to load and run the model. Hugging Face’s Transformers and Tokenizers libraries were used for preprocessing and inference.

Additional development tools included Visual Studio Code (VS Code) as the primary IDE, Git for version control, and GitHub for repository hosting. Cursor AI was occasionally used for smart code suggestions, while GitHub Copilot assisted in boilerplate generation. These tools significantly accelerated development without compromising manual code review and quality.

Alternative technologies were considered and rejected based on fit. Django was deemed too heavyweight for this project’s needs. MongoDB was ruled out due to the relational nature of sentiment records and the need for structured joins, which PostgreSQL handled better. Finally, BERT-base was tested but underperformed on domain-specific tasks compared to RoBERTa.

**3. Development Process & Timeline**

The development process was designed around sprints, with each sprint comprising planning, development, testing, feedback, and documentation. This iterative loop ensured that features could evolve organically based on technical challenges or newly discovered user needs.

In Weeks 1–2, foundational tasks such as development environment setup, technology research, and base Flask structure were completed. Initial documentation, such as the PDD and ethics form, were finalized in this period. Weeks 3–4 were dedicated to backend development, including the creation of RESTful endpoints, model integration, and database schema design. CRUD operations and connection tests with Supabase were also implemented.

Weeks 5–6 focused on the frontend. The MVP version of the dashboard was built using HTML, CSS, and JavaScript. Sentiment results were rendered dynamically using Chart.js. Responsiveness was tested, and user authentication was added to enable basic personalization. API integration ensured that analysis results could be displayed in real-time.

The final two weeks (7–8) were devoted to polishing and optimization. This included implementing error-handling middleware, optimizing sentiment response times through caching, and finalizing the contact form feature on the homepage. User testing was conducted to refine the UI and sentiment summaries.

**4. Testing & Evaluation Methods**

A robust multi-layer testing strategy was applied throughout development. Unit tests were written for backend functions including sentiment classification, database insertion, and user authentication. These tests were scripted in Python and executed using pytest.

Integration testing focused on verifying the connection between modules. For instance, the pipeline from frontend input to backend processing to database storage was tested as a continuous flow. API requests and responses were validated using tools like Postman.

Manual User Acceptance Testing (UAT) was performed to simulate typical use cases. The application was accessed on different browsers (Chrome, Firefox, Safari) and devices (desktop, tablet, mobile) to ensure responsive layout and consistent chart rendering. Basic testing scripts helped measure API response times and check for page load delays.

Performance testing revealed that caching batch predictions of similar text improved response time by over 30%. Frontend rendering via Chart.js was optimized by deferring heavy DOM operations until the data was ready. Database queries were indexed to speed up retrieval of historical sentiment records.

To evaluate the RoBERTa model’s accuracy, sample tweets were annotated manually and compared against model predictions. Results showed over 85% match with human sentiment, confirming suitability for MVP use. Confidence scores were also visualized to help users understand borderline cases.

**5. Ethical, Legal & Professional Considerations**

Given the nature of processing public social media data, the project maintained strict adherence to ethical and legal guidelines. All collected data was public, and no user login information or private content was scraped. No usernames, email addresses, or profile data were stored, thus ensuring compliance with GDPR and institutional research ethics policies.

Authentication systems followed secure practices, including hashing passwords and using environment variables for secret keys. Supabase’s built-in row-level security further restricted unauthorized access. All data at rest was encrypted, and connections were made over HTTPS.

Bias in AI models is a known concern. Regular assessments were conducted to evaluate the RoBERTa model's handling of controversial or ambiguous text. Documentation was provided to inform users that the model may not always capture context such as sarcasm or satire.

Open-source compliance was maintained by attributing all reused code and pre-trained models. Hugging Face’s license was followed, and third-party libraries used in the frontend and backend were properly documented and credited.

**6. Use of AI Tools**

AI-powered development tools played a significant role in supporting this project’s workflow. ChatGPT was used during planning stages to outline milestones, break down features into sprint tasks, and identify potential edge cases in testing. It was also consulted for resolving technical bugs in Flask and data parsing.

Cursor AI and GitHub Copilot were used inside VS Code to generate code suggestions for UI components and API calls. Their output was treated as helper templates—every line was verified, tested, and documented manually.

AI assistance was also employed in writing and formatting documentation. This included initial drafts of user guides, README files, and test plans. All AI-generated content was logged in a separate appendix to ensure transparency and traceability.

**7. Technical Implementation Details**

The application’s core pipeline began with user input (URL or comment text), which was preprocessed for emojis, hashtags, and mentions. This cleaned text was passed to the RoBERTa model, and the predicted sentiment label and confidence score were stored in Supabase along with a timestamp and metadata.

The database schema included a users table (for authentication), a sentiment\_results table (storing predictions), and a logs table (tracking interactions). Supabase's dashboard allowed easy inspection and role-based access control.

The frontend was designed with responsiveness in mind, allowing sentiment charts to adjust on different screen sizes. Dynamic chart updates were achieved through asynchronous fetch calls to the backend. Users could switch between light and dark themes for better accessibility.

The API followed RESTful principles, with endpoints for login, logout, analysis submission, and history retrieval. Rate limiting and error handling were implemented to prevent abuse. JWT tokens were used to maintain user sessions securely.

**8. Challenges & Solutions**

Several challenges arose during development. Integrating the PyTorch-backed RoBERTa model into a lightweight Flask app proved tricky due to version conflicts. This was resolved by isolating dependencies in a virtual environment and pinning package versions.

Performance was another concern. Initial response times for the model exceeded acceptable limits, especially when analyzing multiple comments. Batch processing and result caching were introduced to mitigate this.

The frontend posed layout issues on small screens. Adopting a mobile-first design and testing on emulators helped create a more consistent UI. Additionally, setting up the contact form to route user queries via email required configuring backend logic and form validation.

Version control was challenging due to frequent updates to third-party libraries. A strict version pinning strategy was adopted using requirements.txt and automated deployment testing via GitHub Actions.

Testing ML functionality presented another hurdle. Unlike traditional software, there’s inherent uncertainty in model predictions. This was addressed by setting confidence thresholds, testing edge cases (e.g. sarcasm, emojis), and using explainable AI tools for visualization.

**Conclusion**

The methodology employed in this project successfully combined Agile development practices with AI-based experimentation to create a real-time sentiment analysis dashboard for content creators. The structured phases—spanning planning, development, testing, and deployment—ensured iterative improvement and alignment with initial project goals.

Tools were selected for their suitability to the task, and extensive testing validated both the frontend performance and the backend model accuracy. Ethical considerations were rigorously followed, and AI tools were used responsibly with human oversight. Overall, this methodology enabled the development of a robust, scalable, and user-centered application with potential for future expansion.