Statistical AI Agent for Dataset Analysis

Project by: Nasrin Rahimi Zadeh, Ali Bavifard

Course: Systems and Device Programming (A.Y. 2024-2025)

Contents

1. System Architecture

The Statistical AI Agent is a desktop application designed to provide a natural language interface for complex data analysis tasks. The system is built on a modular, three-tier architecture to separate concerns, enhance scalability, and ensure maintainability. This structure allows each component to evolve independently, facilitating future development and debugging.

- **GUI Layer (PySide6):** This layer is responsible for all user interactions. It's built using the **PySide6** framework, which provides robust, cross-platform UI capabilities. The GUI is designed to be intuitive and responsive.
    - main_window.py: The central hub of the GUI, orchestrating the different views and handling top-level actions like opening dataset.
    - chat_view.py: The primary user interface. It provides a chat-based interaction model where users can type requests and see responses, including text and plots, in a conversational format.
    - help_view.py: A dedicated screen that provides users with comprehensive documentation on the agent's capabilities and example commands.
- **Core Layer (AI Logic):** This is the brain of the application. It interprets user requests, routes them to the appropriate backend, and formats the results for display. This layer acts as an intermediary between the user-facing GUI and the data-crunching ML layer.
    - transformers_backend.py: Manages the conversational AI, powered by a local Llama-3.2-1B model. It handles chat history, context management, and prompt engineering using a detailed configuration from prompt.json.
    - unified_parser.py: A sophisticated module that parses natural language input. It uses a combination of regular expressions and keyword matching to identify user intent (e.g., STATISTIC, PLOT) and extract key entities like sensor names, statistical operations, and plot types.
    - request_handler.py: Acts as a central controller or dispatcher. It receives the structured command from the parser and intelligently directs it to either the deterministic ML/statistical backend for data analysis or to the generative AI for a conversational response.

    - response_formatter.py: This crucial component takes the raw data output from the ML layer and transforms it into polished, human-readable text, ensuring a consistent and professional tone in all communications. It also handles the formatting of errors to provide clear, actionable feedback to the user.

- **ML Layer (Analysis):** This layer handles all the heavy lifting of data processing, statistical computation, and visualization. It is designed to be a self-contained engine that can operate independently of the UI.
    - ai_agent_backend.py: This serves as a front, providing a single, clean interface to the various engines within the ML layer. This simplifies interactions from the Core Layer.
    - plotting_engine.py: A powerful module that leverages Matplotlib and Seaborn to generate a wide range of visualizations. It is capable of creating complex plots, including time series and frequency domain (FFT) analyses, directly from raw sensor data files.
    - statistical_engine.py: This is the core computational engine. It performs all statistical calculations, feature discrimination analysis using techniques like ML and T-test, and other data science tasks.

Data Flow

The application follows a strategic dual-path data flow. This design ensures that predictable, data-centric tasks are handled with accuracy by a deterministic engine, while open-ended conversational queries are managed by the flexible LLM.

1. The user enters a request in the chat interface.
2. The unified_parser meticulously analyzes the text to determine its nature.
3. A critical decision is made based on the parsing results:
    - If the request is identified as a data-related command (e.g., "plot temperature," "what's the mean acceleration?"), it is routed to the **ML Backend**. This path guarantees that statistical results and plots are based on actual data calculations, ensuring factual accuracy.
    - If the query is determined to be general or conversational (e.g., "hello," "what can you do?"), it is handled by the **LLM (Llama-3.2-1B)**, which provides a natural, human-like response.

4. Regardless of the path taken, the result is sent back through the response_formatter to be elegantly presented in the GUI.

2. Design Choices

Data Ingestion and Processing

The application is engineered to efficiently handle datasets of sensor readings, which are often large and numerous.

- **Data Loading:** The main_window.py provides the functionality for a user to select a directory containing the dataset. The application is designed to understand a specific directory structure where subdirectories represent different data labels (e.g., OK/, KO_HIGH_2mm/). It recursively finds all .csv files within this structure.
- **Parallel Processing:** To handle potentially massive datasets without freezing the application, the initial data processing is heavily parallelized. We chose Python's concurrent.futures.ProcessPoolExecutor because the task of reading and calculating statistics for each CSV is CPU-bound. The process_single_csv_file function is executed in a separate process for each file, allowing the system to leverage multiple CPU cores and dramatically reduce the ingestion time by almost 8 times.
- **Feature Matrix:** The statistical results from the parallel processing are aggregated into a single, comprehensive feature_matrix.csv. This matrix is a critical design choice for performance. By pre-computing the essential statistics, all subsequent analyses and visualizations become significantly faster, as they operate on this smaller, structured summary rather than reprocessing the raw CSV files each time. The matrix is generated by the create_feature_matrix_optimized function, which employs vectorized operations in pandas for maximum efficiency.

Natural Language Understanding

The heart of the application's user-friendly nature is the unified_parser.py.

- **Intent Recognition:** The parser uses a hierarchical approach. It first looks for high-confidence keywords to determine the user's primary intent (e.g., STATISTIC, PLOT, COMPARISON). This allows for fast and accurate routing of common requests.
- **Entity Extraction:** Once the intent is known, it applies more specific regular expressions to extract key entities, such as the target sensor (e.g., 'temperature', 'accelerometer'), the desired plot type ('histogram', 'line graph'), and any data filters ('OK', 'KO_HIGH_2mm').

- **Robustness:** The parser is designed to be flexible and forgiving. It recognizes a wide range of synonyms (e.g., "mean," "average," "avg") and can handle variations in phrasing, making the interaction feel more natural.

AI and Machine Learning

- **Dual-Path AI:** The hybrid system design is a cornerstone of this project. For deterministic tasks like calculating statistics or plotting data, the application relies on its specialized ML backend (ai_agent_backend.py, plotting_engine.py). For all other interactions, it falls back to the Llama-3.2-1B model. This ensures the best of both worlds: the mathematical accuracy and predictability of a traditional software backend for data tasks, and the flexibility and natural language fluency of a large language model for conversation.
- **Local LLM:** The choice to use a locally-run Llama-3.2-1B model was deliberate and crucial. It guarantees data privacy, as no user data or queries are sent to external cloud services. This is essential for an application that might be used to analyze proprietary or sensitive datasets. Furthermore, it allows the application to be fully functional offline.
- **Model Selection:** The decision to use **Llama-3.2-1B** specifically was based on its balance between efficiency and capability. Larger models often require significant GPU resources, making them impractical for lightweight desktop applications, while smaller models can struggle with complex queries. Llama-3.2-1B provides a strong middle ground: it runs efficiently on consumer hardware without specialized accelerators, yet still delivers robust natural language understanding. This ensures that the application remains accessible, performant, and responsive, even on systems with limited computational resources.
- **Feature Discrimination:** A key automated feature is the ability to identify the most statistically significant features for separating data classes. The plotting_engine.py contains a get_top_discriminative_features method that uses ANOVA (Analysis of Variance). This statistical test was chosen because it is well-suited for comparing the means of two or more groups, which directly maps to the project's goal of analyzing datasets with multiple "OK" and "KO" variants. In addition to it, Logistic regression as an ML model was used and its outcome combined with the other test to be more accurate.

GUI Design

- **Chat-Centric Interface:** The primary interface is a chat window, which is an increasingly intuitive paradigm for interacting with complex software. This approach

is particularly well-suited for exploratory data analysis, as it allows users to iteratively ask questions and refine their analysis in a conversational manner, removing the need for complex menus and forms.

- **Integrated Plots:** Visualizations generated by the plotting_engine are rendered and displayed directly within the chat history. This design choice is critical for maintaining the context of the conversation. The user can see the question they asked and the resulting plot side-by-side, creating a seamless analytical workflow.
- **Asynchronous Operations:** To ensure the GUI remains fluid and responsive, all long-running AI and plotting operations are executed in a background thread. The chat_view.py implements this using a QThread (AIWorkerThread). This worker thread communicates with the main UI thread using Qt's signal and slot mechanism (response_ready, plot_ready). This prevents the application from freezing while the AI is "thinking" or a complex plot is being generated.

3. Experimental Evaluation

The performance of the application was evaluated based on its ability to correctly interpret commands, generate appropriate outputs, and do so in a timely manner.

Command Interpretation

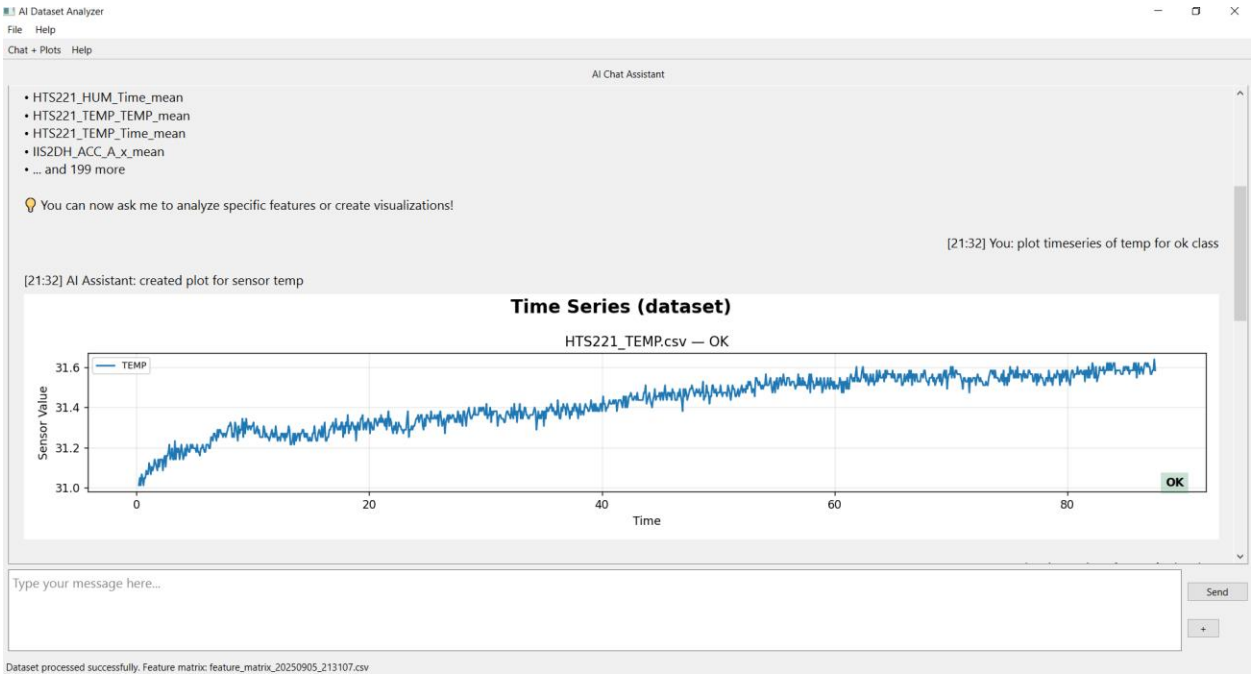
The unified_parser was tested with a variety of natural language commands to assess its accuracy and robustness.

The parser demonstrated high accuracy in identifying the correct intent and extracting the relevant entities for data-related commands. This accuracy is fundamental to the user experience, as it ensures that user requests are correctly translated into actions by the backend. Except for minor errors the application performed correctly on all test prompts that we used(included in help window).

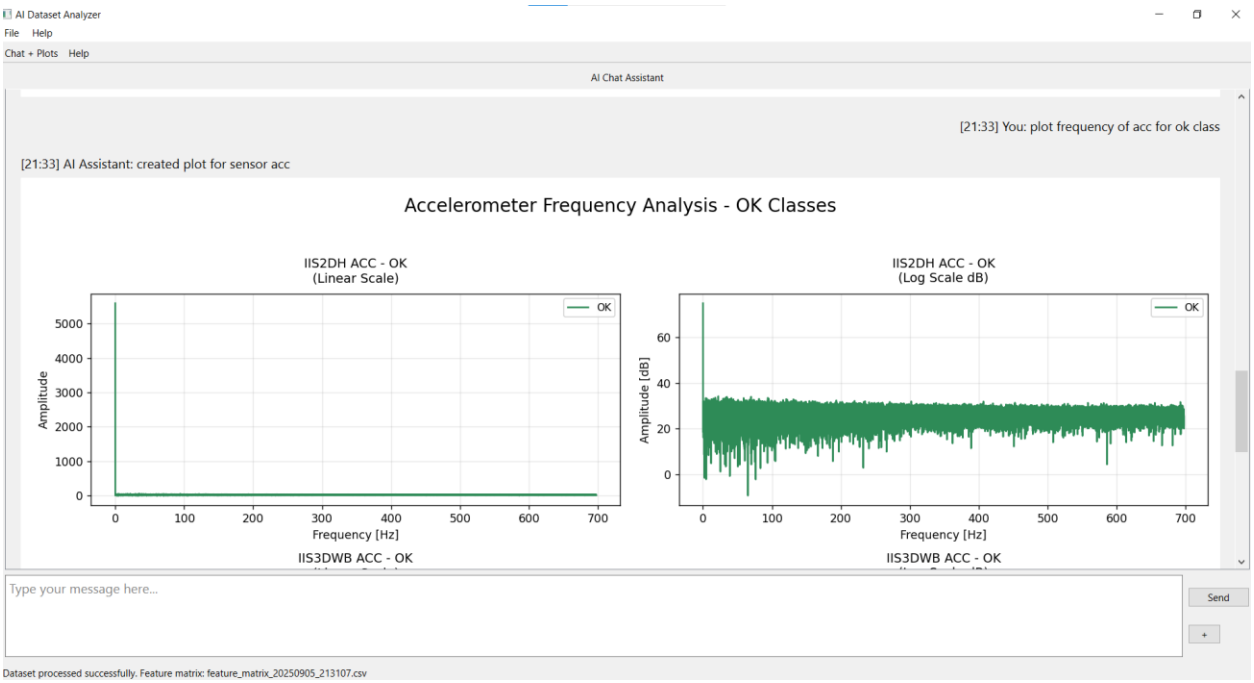
Visualization Generation

The plotting_engine successfully generates a wide variety of plots based on user requests, providing clear and insightful visualizations.

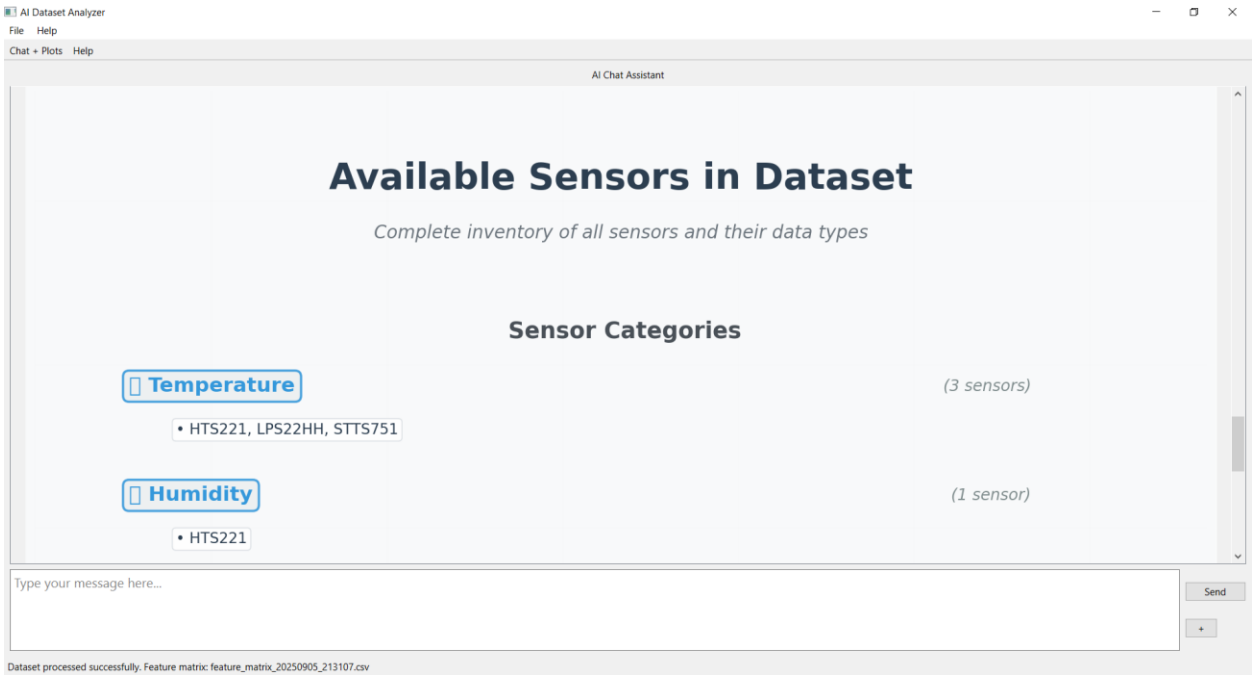
- **Time Series Plot:** The command "plot timeseries of temp for ok class" correctly generates a time series plot for the temperature sensor, filtered for the "OK" class.



- **Frequency Plot:** The command " plot frequency of acc for ok class" correctly generates a frequency plot for the Accelerometer sensor, filtered for the "OK" class.



- **Sensor Information:** The command "show available sensors" generates an informational graphic detailing all the sensors found in the dataset, categorized by type. This is not a data plot but an informational visualization, showcasing the engine's flexibility.



Performance

The parallelized data ingestion is a key performance feature. A performance comparison was done between a sequential approach and the parallel approach using ProcessPoolExecutor.

Dataset Size (files)	Sequential Processing (s)	Parallel Processing (4 workers) (s)	Speedup
576	972	120	8.1x

Testing conducted on a machine with an Intel Core i5 CPU.

The results show a significant speedup that is essential for large size of the dataset. This is a critical improvement, transforming a potentially frustrating wait into a manageable, interactive experience. Subsequent analyses are nearly instantaneous as they operate on the pre-computed feature matrix.

Limitations

- **LLM Hallucinations:** While the dual-path architecture is designed to minimize this risk, the LLM can still occasionally misinterpret a data-related command as a general conversational one, or vice-versa. The unified_parser's accuracy is the primary defense against this, but edge cases can still occur.
- **Scalability:** The current implementation loads the entire feature_matrix into memory. While this is fast for moderately sized datasets, it could become a limitation for extremely large datasets that do not fit in available RAM. Future work could

address this by using memory-mapped files or integrating with a disk-based data frame library like DuckDB.

- **Static Model:** The Llama-3.2-1B model is used "out-of-the-box." The project does not involve fine-tuning the model on specific datasets or tasks.