Titanic: Machine Learning from Disaster

System Analysis Project

Julián Carvajal Garnica
20242020024

Andrés Mauricio Cepeda Villanueva
20242020010

Jhonatan David Moreno Barragán
20201020094

Andrés Camilo Ramos Rojas
20242020005

School of Engineering, Universidad Distrital Francisco José de Caldas
System Analysis Course
Teacher: Carlos Andrés Sierra
Bogotá D.C.
2025

Part 2 of the System Analysis Project

1. Review and Refine System Architecture

Based on the high-level architecture defined in Part 2, this section refines the design to incorporate robust engineering principles, as requested in Workshop 3. The goal is to evolve our predictive system from a prototype to a sustainable solution, explicitly addressing reliability, scalability, and maintainability.

1.1 Refinement of Robust Design Principles

The original architecture established a modular flow. We now refine this with an explicit focus on robustness:

- **Modularity and Maintainability (CMMI):** We are taking modularity a step further. Each component (Ingestion, Preprocessing, Modeling, Evaluation) [cite: 96-100] will be designed as a containerized microservice (e.g., using Docker). This not only reinforces modularity but also aligns with **CMMI** principles by allowing independent configuration management and facilitating maintainability, as one component can be updated without destabilizing the entire system.
- **Scalability:** The scalability NFR will be addressed through the orchestration of these containers (e.g., with Kubernetes). Specifically, the *Modeling Layer* and *Deployment Layer* will be able to scale horizontally. If the demand for Titanic predictions increases, we can dynamically launch more instances of the *Deployment* service.
- **Fault-Tolerance:** This is a critical refinement.
  - **Data Ingestion:** A message queue (e.g., RabbitMQ or Kafka) will be implemented between the ingestion and preprocessing layers. If the *Preprocessing Layer* fails, the raw Kaggle data is not lost; it waits in the queue, ensuring **reliability**.

  - **Deployment:** A load balancer will be used to distribute prediction requests among multiple model replicas. If one replica fails, traffic is automatically rerouted to healthy ones.
  - **Feedback Loop:** The *Feedback Layer* will monitor model health. If the error rate (an indicator of chaos) exceeds a threshold, the system can automatically roll back to a previously stable model version, ensuring **homeostasis**.

1.2 Components and their Support for Quality (ISO 9000)

Following the process management principle of **ISO 9000**, we define how each component ensures system quality and reliability:

- **Data Ingestion Layer:**
  - *Quality:* Implements data validation schemas (e.g., Pydantic) to ensure incoming data meets the expected format (data types, ranges).
  - *Reliability:* Uses retries and the message queue (see above) to handle network failures or Kaggle service downtime.
- **Preprocessing Layer:**
  - *Quality:* Versioning of preprocessing artifacts (e.g., Scikit-learn scalers or encoders) is implemented. Each trained model will be linked to the exact version of the preprocessing pipeline it used, ensuring **transparency** and reproducibility.
- **Modeling & Evaluation Layer:**
  - *Quality:* We apply a *Six Sigma* approach to "defect reduction" (prediction errors). The *Evaluation Layer* will not only measure accuracy but also use SHAP and LIME to analyze the *root cause* of errors and bias, aligning with the "Analyze" phase of DMAIC.
- **Deployment & Feedback Layer:**
  - *Reliability:* Implements "Canary" deployments. A new model version is first released to a small percentage of users. The *Feedback Layer* monitors its real-time performance. If stable, it is rolled out to everyone; if it fails, it is rolled back without affecting the majority of users.

2. Define System Requirements

Building on the analytical understanding from Workshop #1, the following requirements define the foundations of the Kaggle system design. These requirements translate the system's complexity, sensitivity, and feedback dynamics into concrete engineering objectives.

2.1 Functional Requirements

- The system must ingest, preprocess, and validate data from the selected Kaggle dataset.
- It must train and evaluate predictive models capable of handling nonlinear and sensitive data patterns.
- It must incorporate monitoring and feedback to sustain model performance over time.
- It should provide interfaces or visualization tools to track the evolution and stability of predictions.

2.2 Non-Functional Requirements

- **Scalability:** The system should grow dynamically with increasing data size or model complexity.
- **Modularity:** Each subsystem must operate independently while maintaining interconnection.
- **Reliability:** The design must ensure homeostasis—operational stability amid perturbations.
- **Transparency:** Outputs and internal states should be interpretable and traceable for verification.
- **Maintainability:** Components must be easily updated or replaced without destabilizing the whole.

2.3 Systemic Requirements

- **Homeostasis:** Self-regulation mechanisms must control sensitivity and prevent instability.

- **Permeability:** The system should remain open to new data or environmental feedback.
- **Equifinality:** Different models or pathways can achieve the same predictive goal.
- **Adaptability:** The system must evolve through learning, maintaining equilibrium amid change.
- **Holism and Synergy:** The overall performance emerges from the coordinated function of parts.

These requirements establish a foundation for designing a complex adaptive system, not just a static machine learning pipeline, where chaos and order coexist under structured feedback control.

3. High-Level Architecture

The proposed architecture represents an open, adaptive, and self-organizing system. Its modular design ensures scalability, sensitivity management, and dynamic equilibrium. The flow of data and information follows a systemic pattern that resembles biological and ecological models—interconnected, feedback-driven, and hierarchically organized.

3.1 System Components

- **Data Ingestion Layer:** Collects raw Kaggle data, ensuring permeability to external sources.
- **Preprocessing Layer:** Filters noise, normalizes inputs, and stabilizes data variability.
- **Modeling Layer:** Executes machine learning algorithms, balancing deterministic computation and stochastic exploration.
- **Evaluation Layer:** Assesses model accuracy, detects chaotic deviations, and quantifies sensitivity.
- **Deployment & Feedback Layer:** Delivers the model to users or environments while receiving performance feedback to adjust internal states—representing homeostasis and autoregulation.

This architecture embodies complexity with control: chaos is not eliminated but harnessed through adaptive loops that preserve systemic balance.

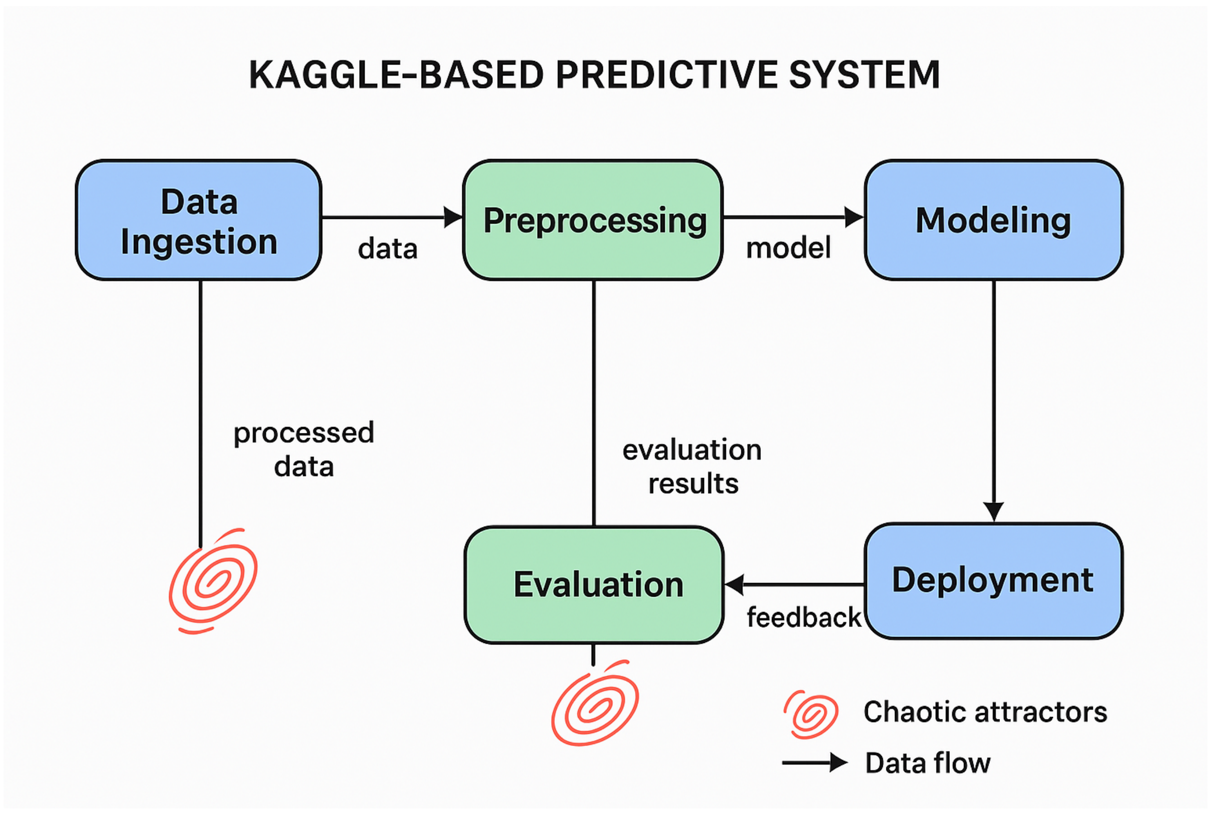

Figure 1: System architecture diagram for a Kaggle-based predictive system showing interconnected modules for data ingestion, preprocessing, modeling, evaluation, and deployment.

4. Addressing Sensitivity and Chaos

Machine learning systems are inherently sensitive to input variation. Small perturbations—missing values, biased samples, or random fluctuations—can produce nonlinear effects that propagate through the model. This phenomenon, akin to chaotic attractors, requires systemic mitigation strategies.

The proposed design incorporates feedback control mechanisms inspired by cybernetics and systems theory. These loops continuously compare expected outputs with actual results, triggering self-regulation and parameter adjustments when discrepancies exceed tolerance thresholds. This process emulates homeostasis, enabling the system to stabilize despite uncertainty.

Furthermore, by analyzing feature sensitivity and error propagation, the system learns to identify fragile variables—those that disproportionately affect output. These are managed through adaptive preprocessing, ensemble modeling, and reweighting strategies that strengthen robustness.

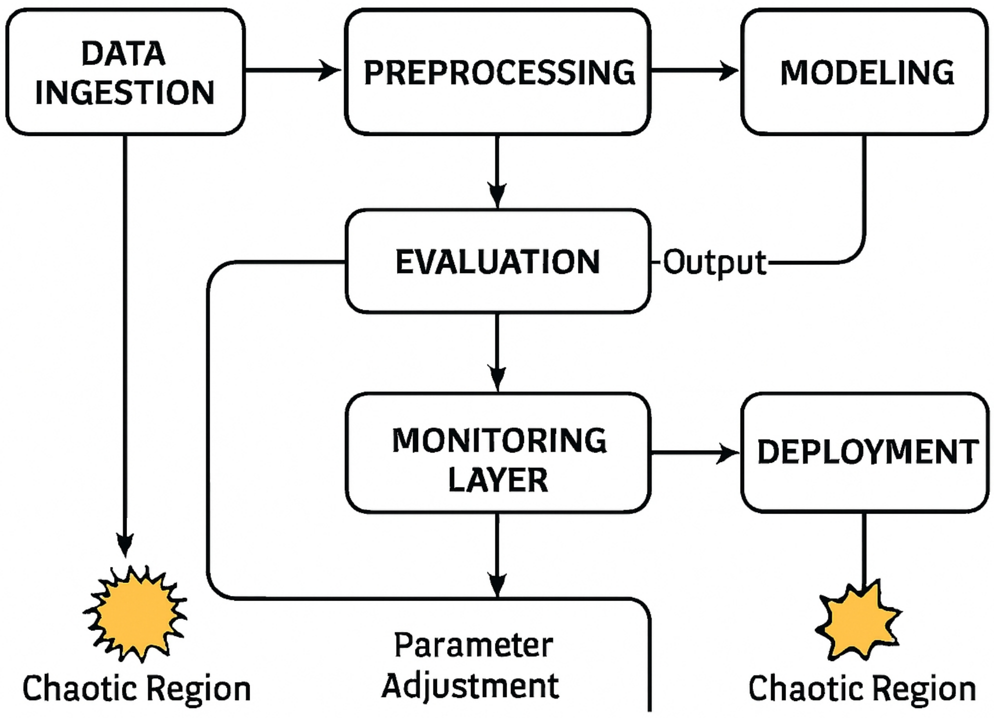

Figure 2: Feedback control diagram for a machine learning system illustrating self-regulation mechanisms. The diagram shows loops between *Model Output*, *Monitoring Layer*, and *Parameter Adjustment*, representing feedback and self-regulation. Chaotic regions highlight areas where small input changes lead to large output variations.

5. Technical Stack and Implementation Sketch

The technical foundation is selected to ensure modularity, interoperability, and adaptive scalability. Each layer corresponds to a subsystem within the architecture, integrating computational intelligence and systems engineering principles.

Layer	Tools / Technologies	Purpose and Role in System
Data Layer	Python (Pandas, NumPy)	Data ingestion, cleaning, and transformation.
Processing Layer	Scikit-learn, TensorFlow	Model training and performance optimization.
Evaluation Layer	Matplotlib, SHAP, LIME	Interpretability and sensitivity analysis.
Deployment Layer	Flask, FastAPI, Streamlit	Visualization, deployment, and user interaction.
Feedback Layer	Airflow, Prefect, or custom pipelines	Continuous monitoring, retraining, and control loops.
Versioning	Git + GitHub	Ensures traceability, reproducibility, and collaboration.

Table 1: Technical Stack and System Implementation Sketch

The implementation follows a modular microservice structure. Each component functions semi-independently, communicating through defined interfaces, promoting permeability and synergy. The balance between deterministic modules (data preprocessing, evaluation) and stochastic modules (modeling, retraining) mirrors a chaotic yet self-organized ecosystem.

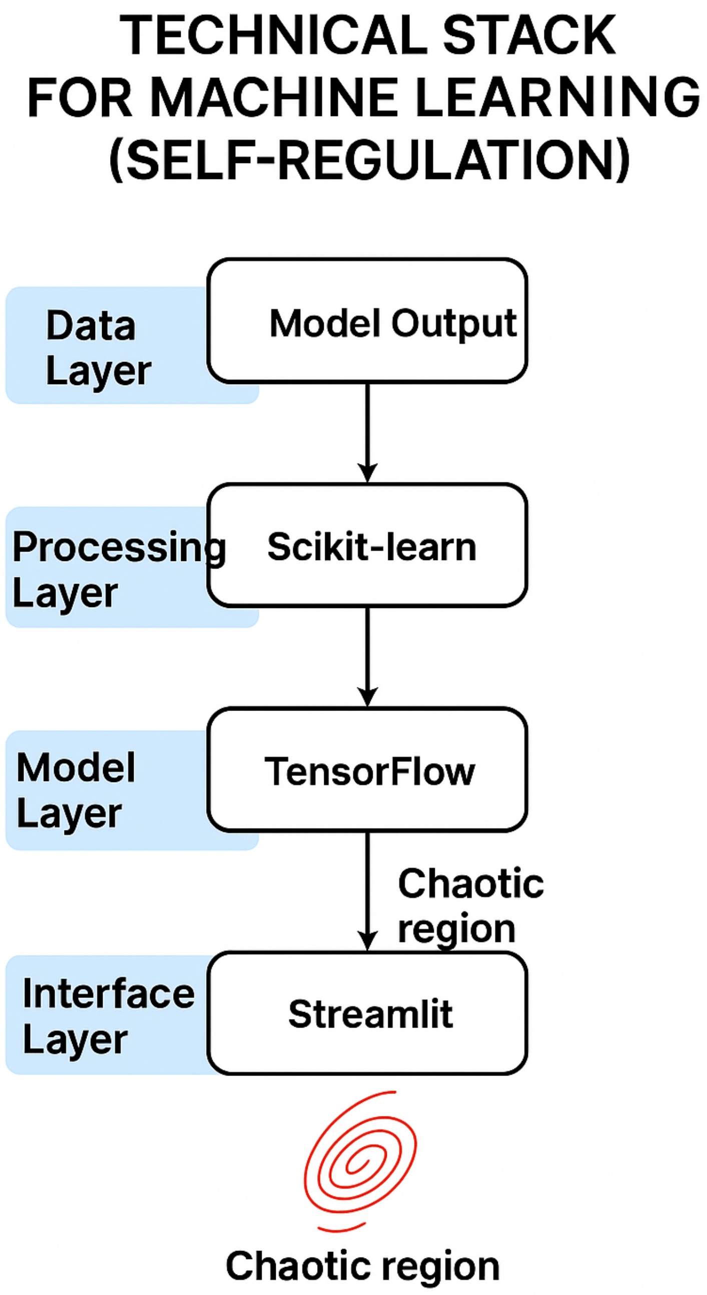

Figure 3: Technical stack diagram linking each architectural layer (*Data Layer*, *Processing Layer*, *Model Layer*, and *Interface Layer*) with corresponding technologies and frameworks such as Pandas, Scikit-learn, TensorFlow, and Streamlit. This diagram illustrates how each component integrates within the system to ensure modularity, interoperability, and adaptive scalability.