Flight Delay Prediction System

Atharva Purohit, Jehan Kotwal, Nirupama Balasubramaniam, Dhevdharsan Bhavani Satish Kumar

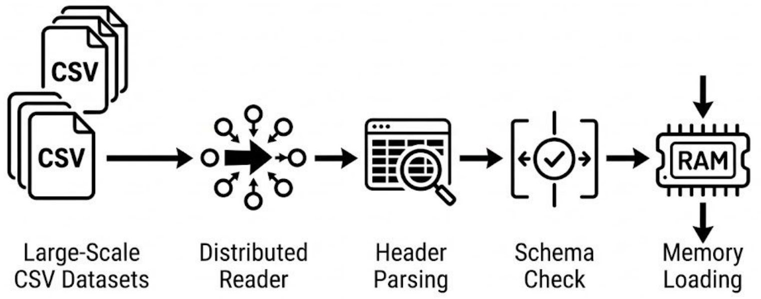
COMPSCI 532 - Fall 2025

Design Description

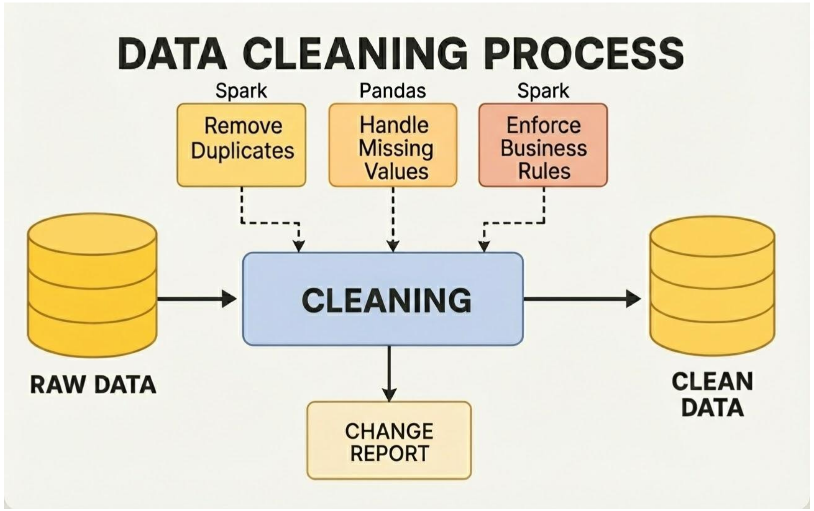
- Developed a scalable machine learning system to predict flight delays using distributed computing with PySpark.
- Benchmarked PySpark against Pandas to illustrate significant performance differences and validate the need for distributed systems.
- Processed over 1 million flight records to demonstrate system efficiency in computationally intensive tasks that are unfeasible in single-node environments.

Design Description - Part 2

- Data Ingestion
  - Dataset contains date, airline, origin, time, etc. (some useful columns others not so)
  - The Spark ingests CSV datasets using a distributed reader. Schema inference is enabled to automatically detect data types.

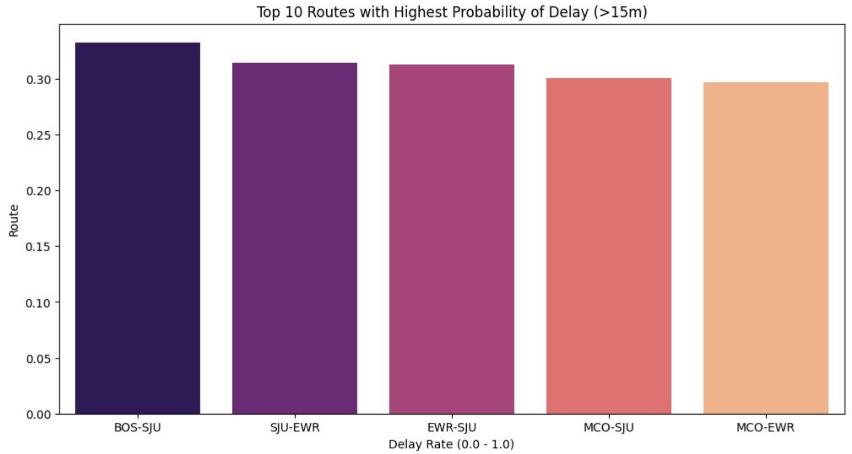


- Data Cleaning
  - Drop columns that are not needed.
  - Removed any flights that were cancelled or diverted.
  - Validated that flights traveled between 10 and 10,000 miles.
  - Checked that flight speeds were within the 50 to 700 mph range.
  - Ensured a consistent 24 hour time format.



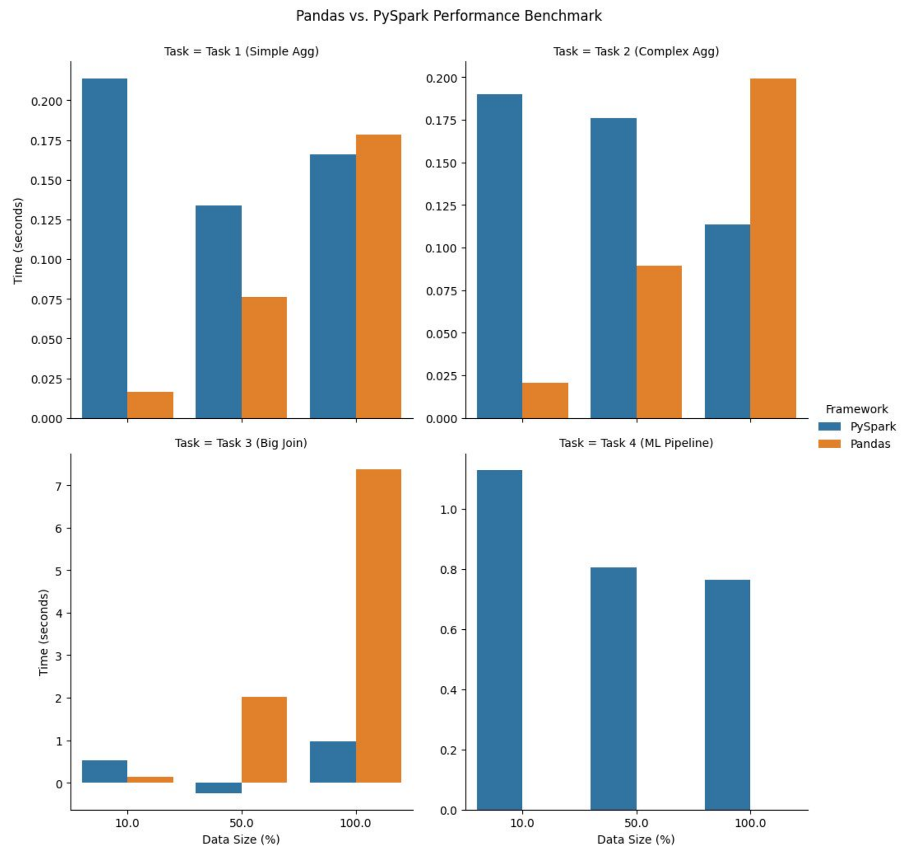
Design Description - Part 3(EDA)

- SQL-based analysis to gain an understanding of the data set we analyzed -
    - Delay by Route
    - Delay by Airline
    - Delay by Distance
    - Delay by Day of Week
    - Delay by Hour
    - Delay by Origin



Tests - PySpark vs Pandas

- Compared execution time of PySpark vs. Pandas across data volumes (10%, 50%, 100%).
- We found Pandas works well at low load and easy tasks, but degrades with load or fails outright while PySpark scales linearly.
- Tasks Tested:
    - Simple Aggregation: GroupBy Origin count.
    - Complex Aggregation: Stats by Airline.
    - Heavy Join: Self-join on Date/Origin.
    - ML Pipeline: One hot encoding



Tests - Model Experimentation

Phase 1: Baseline Modeling (Imbalanced Data)

- **The Approach:** Trained Logistic Regression, FMC, Random Forest, GBT, and Linear SVC on raw, imbalanced data (~4.65:1 ratio).
- **The Result:** High Accuracy (~82%) but Low AUC (~0.52 - 0.62).
- **Interpretation:** Demonstrated the **Accuracy Paradox**; models prioritized the majority class ("No Deay") to maximize accuracy but failed to distinguish actual delays.

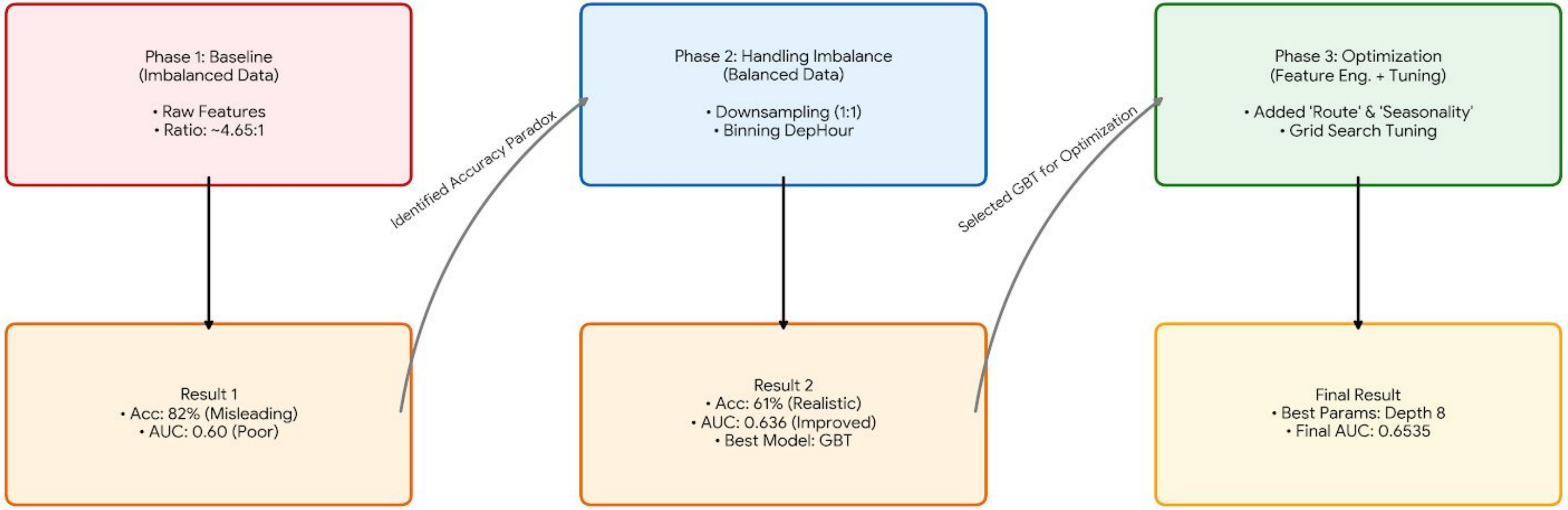
Phase 2: Handling Class Imbalance (Balanced Data)

- **The Approach:** Rebalanced training data to a 1:1 ratio via downsampling and added feature grouping.
- **The Result:** Accuracy normalized to ~55-61% while AUC stabilized (~0.63).
- **Interpretation:** Fixed the model bias, forcing it to learn actual delay patterns. **Gradient Boosted Trees (GBT)** emerged as the best performer (AUC 0.636).

Phase 3: Optimization (Feature Engineering + Tuning)

- **The Approach:** Enhanced the balanced GBT model with interaction features (`Route`, `Month`, `DayOfWeek`) and Grid Search tuning.
- **The Result:** Identified optimal parameters (Max Depth 8, Iterations 20) with a **Final AUC of 0.6535**.
- **Interpretation:** Though computationally expensive (~3 hours), this phase yielded the best predictive performance, improving AUC by ~3.5% over the baseline.

Experimental Workflow: Flight Delay Prediction



Tests - How Spark Enabled this Project

1. Efficient Data Ingestion - The SparkSession.read.csv command ingested the 3GB+ dataset into a distributed DataFrame structure instantly. Unlike single-node libraries that load all data into RAM at once, Spark managed the data pointers efficiently, allowing for immediate EDA and schema validation.

2. Lazy Evaluation & Optimization - Spark utilized Lazy Evaluation, meaning it didn't execute cleaning or transformation steps until absolutely necessary (the .fit() call). This allowed the Catalyst Optimizer to combine operations efficiently, minimizing data passes and computational overhead.

3. Unified ML Pipelines - Spark allowed us to wrap distinct complex steps—StringIndexer, OneHotEncoder, and VectorAssembler—into a single Pipeline object. This ensured that the exact same feature engineering logic applied to the Training set was automatically applied to the Test set, eliminating data leakage.

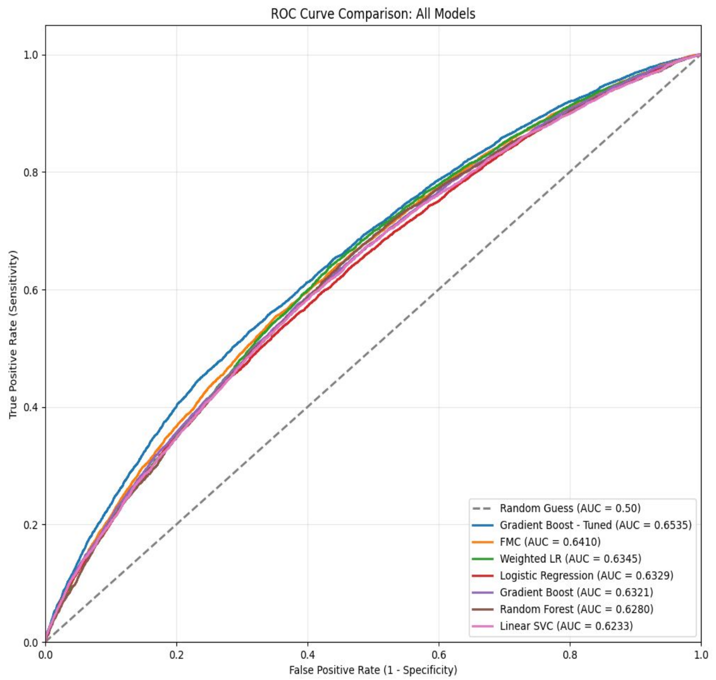
4. Handling High-Dimensionality at Scale - Processing 3 million rows with high-cardinality features (like the One-Hot Encoded Route column) creates massive sparse vectors. Spark prevented memory crashes by processing data in partitions and spilling to disk when RAM was full, a task that often fails in standard Pandas.

5. Parallelized Hyperparameter Tuning - Spark’s TrainValidationSplit revolutionized the optimization phase. Instead of training models sequentially, Spark leveraged available resources to train multiple parameter combinations (e.g., Depth 5 vs. 8) simultaneously, drastically reducing tuning time.

6. Distributed Algorithm Execution - Within the models themselves, specifically Random Forest, Spark distributed the construction of individual decision trees across different CPU cores. This parallelization meant that increasing the number of trees improved accuracy without a linear increase in training time.

Experimental Results

Models	AUC Values
Gradient Boost - After Hyperparameter Tuning	**0.6535**
Factorization Machine Classifier (FMC)	0.6410
Weighted LR	0.6345
Logistic Regression	0.6329
Gradient Boost	0.6320
Random Forest	0.6280
Linear SVC	0.6233



Goals

- COMPLETE - Environment Setup
- COMPLETE - Data Ingestion
- COMPLETE - Initial Cleaning and Target Creation
- COMPLETE - Exploratory Data Analysis (EDA)
- COMPLETE - Benchmarking
- COMPLETE - Feature Transformation
- COMPLETE - Pipeline Construction
- COMPLETE - Model Training
- COMPLETE - Distributed Training
- COMPLETE - Evaluation

Incomplete Goals and Possible Improvements

- Possible Improvements
  - Ingest real-time weather API data for origin/destination airports to improve predictive power.
  - Migrate to Spark Structured Streaming for live delay predictions as flight status updates.
  - Experiment with Deep Learning for complex pattern recognition