Data Preprocessing to Model Selection Rationale

Sales Forecasting for Pharmaceutical Products Using Databricks

> **This document explains the logical connection between specific data preprocessing techniques using PySpark/Databricks and the choice of ARIMA for time series forecasting, demonstrating why enterprise-scale preprocessing informs model selection for pharmaceutical demand prediction.**

Table of Contents

1. Executive Summary

The Core Question

Why use PySpark/Databricks for preprocessing, and why does ARIMA emerge as the model choice?

Quick Answer

Preprocessing Technique	Problem Solved	How It Enables ARIMA
PySpark Data Loading	Handles large-scale data in DBFS	Validates data availability and quality
Schema Inference	Automatic type detection	Identifies date and numeric columns
Date Conversion	String → DateType	Creates proper time series index
Fill Missing with Zero	Handles NULL values	ARIMA requires complete series
Aggregation (Monthly/Weekly)	Multiple granularities	Enables forecasting at business-relevant levels

Preprocessing Technique	Problem Solved	How It Enables ARIMA
Spark → Pandas	Converts for statsmodels	ARIMA requires pandas Series

Model Selection Rationale

```
┌──────────────────────────────────────────────────────────────┐
│  After preprocessing, the data has:                          │
│  ✓ Clean datetime index                                      │
│  ✓ No missing values (filled with 0)                         │
│  ✓ Single-column time series per category                    │
│  ✓ Consistent temporal ordering                              │
│  ✓ Pandas DataFrame format (required by statsmodels)         │
│                                                              │
│  → ARIMA is optimal for single-variable forecasting          │
└──────────────────────────────────────────────────────────────┘
```

2. Data Characteristics & Challenges

2.1 Original Data Structure

Characteristic	Value
Storage	DBFS (Databricks File System)
Format	CSV files at multiple granularities
Drug Categories	8 ATC classifications
Time Range	Multi-year (2014-2019)
Granularities Available	Hourly, Daily, Weekly, Monthly

Available Files:

File	DBFS Path	Rows	Use Case
`salesdaily.csv`	`/FileStore/tables/salesdaily.csv`	~2,160	Primary forecasting
`saleshourly.csv`	`/FileStore/tables/saleshourly.csv`	~52,560	Intraday analysis
`salesweekly.csv`	`/FileStore/tables/salesweekly.csv`	~302	Trend analysis
`salesmonthly.csv`	`/FileStore/tables/salesmonthly.csv`	~72	High-level planning

Drug Categories:

Code	Category	Business Relevance

Code	Category	Business Relevance
M01AB	Anti-inflammatory (Diclofenac)	Weather-sensitive
M01AE	Anti-inflammatory (Ibuprofen)	OTC demand
N02BA	Analgesics (Aspirin)	Consistent demand
N02BE	Analgesics (Paracetamol)	Seasonal (flu)
N05B	Anxiolytics	Prescription-driven
N05C	Sedatives	Prescription-driven
R03	Respiratory	Seasonal (allergies)
R06	Antihistamines	Seasonal (allergies)

2.2 Key Challenges Requiring Specific Preprocessing

Challenge 1: Enterprise-Scale Data Environment

Problem:

- Data stored in distributed DBFS
- Multiple file formats and granularities
- Need for scalable processing

Solution: Use PySpark for distributed data loading

Challenge 2: Inconsistent Data Types

Problem:

- Date column stored as string (`"2019-01-15"`)
- Schema varies across files
- Type mismatches between files

Solution: Schema inference + explicit date conversion

Challenge 3: Missing Values

Problem:

- NULL values in sales columns
- Days with no sales recorded as missing
- ARIMA cannot handle NaN values

Solution: Fill missing values with 0

Challenge 4: Model Library Requirements

Problem:

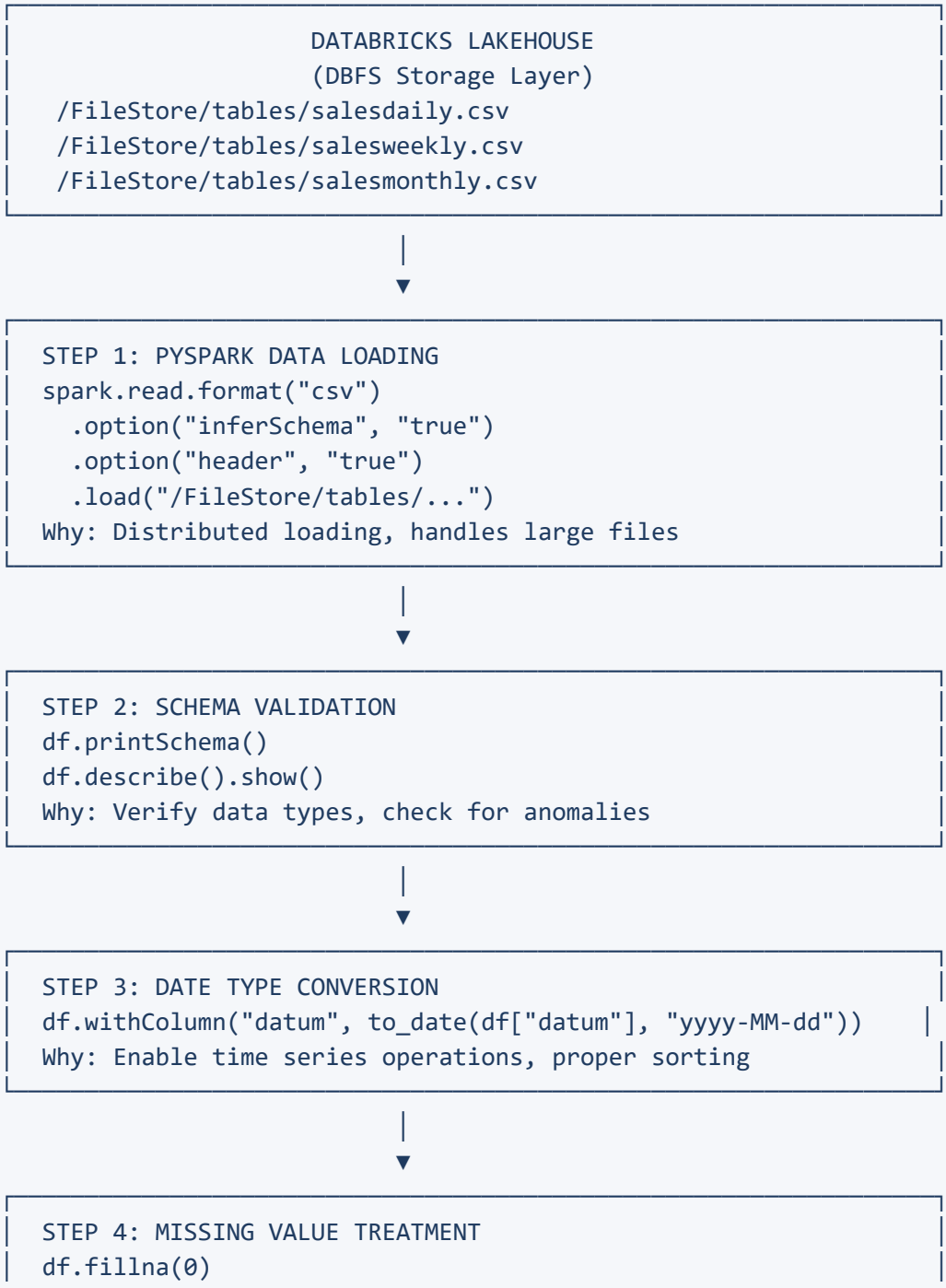
- `statsmodels.ARIMA` requires pandas input
- PySpark DataFrames not directly compatible
- Need datetime index for time series

Solution: Convert Spark → Pandas with proper indexing

---

3. Preprocessing Pipeline Overview

Complete Databricks Pipeline



```
│  Why: ARIMA requires complete series, 0 = no sales                │
└──────────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌──────────────────────────────────────────────────────────────────┐
│  STEP 5: AGGREGATION (OPTIONAL)                                  │
│  df.groupBy("Year", "Month").sum(...)                            │
│  Why: Create business-relevant time periods                      │
└──────────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌──────────────────────────────────────────────────────────────────┐
│  STEP 6: SPARK → PANDAS CONVERSION                               │
│  category_df = df.select("datum", "M01AB").toPandas()            │
│  category_df.set_index("datum", inplace=True)                    │
│  Why: statsmodels requires pandas with datetime index            │
└──────────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌──────────────────────────────────────────────────────────────────┐
│                    ARIMA FORECASTING                             │
│  model = ARIMA(category_df['M01AB'], order=(1,1,1))              │
│  model_fit = model.fit()                                         │
│  forecast = model_fit.forecast(steps=30)                         │
└──────────────────────────────────────────────────────────────────┘
```

4. Preprocessing Step 1: Distributed Data Loading

4.1 PySpark CSV Loading

```
# Define file locations
file_location_daily = "/FileStore/tables/salesdaily.csv"
file_type = "csv"

# CSV reading options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# Load DataFrame
df_daily = spark.read.format(file_type) \
    .option("inferSchema", infer_schema) \
    .option("header", first_row_is_header) \
    .option("sep", delimiter) \
    .load(file_location_daily)
```

4.2 Why PySpark Instead of Pandas?

Aspect	PySpark	Pandas
Scalability	Handles TB+ data	Limited by memory
Distributed	Runs on cluster	Single machine
DBFS Integration	Native support	Requires workarounds
Enterprise Standard	Databricks default	Manual setup

4.3 Loading Multiple Granularities

```
# Load all granularities
df_daily = spark.read.format(file_type)...load(file_location_daily)
df_hourly = spark.read.format(file_type)...load(file_location_hourly)
df_weekly = spark.read.format(file_type)...load(file_location_weekly)
df_monthly = spark.read.format(file_type)...load(file_location_monthly)
```

Why Load All Four?

Granularity	Purpose	Forecasting Use
Daily	Primary forecasting	30-day predictions
Hourly	Intraday patterns	Peak hour analysis
Weekly	Smoothed trends	Weekly ordering cycles
Monthly	Strategic planning	Budget forecasting

4.4 How This Enables ARIMA

Loading Choice	ARIMA Benefit
Daily granularity	Sufficient data points for parameter estimation
Schema inference	Identifies numeric columns for modeling
Multiple files	Flexibility to choose optimal granularity

5. Preprocessing Step 2: Schema Validation & Type Conversion

5.1 Schema Inspection

```
# Print schemas to understand structure
df_daily.printSchema()
df_daily.describe().show()
```

Schema Output:

```
root
 |-- datum: string (nullable = true)     ← Problem: Should be date
 |-- M01AB: double (nullable = true)
 |-- M01AE: double (nullable = true)
 |-- N02BA: double (nullable = true)
 |-- N02BE: double (nullable = true)
 |-- N05B: double (nullable = true)
 |-- N05C: double (nullable = true)
 |-- R03: double (nullable = true)
 |-- R06: double (nullable = true)
 |-- Year: integer (nullable = true)
 |-- Month: integer (nullable = true)
 |-- Weekday Name: string (nullable = true)
```

5.2 Date Type Conversion

```
from pyspark.sql.functions import to_date

# Convert datum from string to date
df_daily = df_daily.withColumn("datum", to_date(df_daily["datum"], "yyyy-MM-
dd"))
```

Why Convert to DateType?

String Date	Date Type
Sorts alphabetically	Sorts chronologically
No date operations	Supports year(), month(), dayofweek()
Cannot be time series index	Valid index for ARIMA

Date Format Pattern:

Pattern	Meaning	Example

Pattern	Meaning	Example
`yyyy`	4-digit year	2019
`MM`	2-digit month	01-12
`dd`	2-digit day	01-31

5.3 How This Enables ARIMA

Preprocessing	ARIMA Benefit
Schema validation	Confirms numeric target columns exist
Date conversion	Creates proper time series index
Type checking	Prevents runtime errors in model

---

6. Preprocessing Step 3: Missing Value Treatment

6.1 Fill Missing with Zero

```
# Fill all NULL values with 0
df_daily = df_daily.fillna(0)
```

6.2 Why Zero is the Correct Fill Value

Data Type	Missing Meaning	Correct Fill
Sales volume	No sales occurred	**0**
Revenue	No revenue	**0**
Count	Zero count	**0**

Alternative Approaches Considered:

Method	Code	Why Not Used
Mean imputation	`df.fillna(df.mean())`	Creates artificial sales that didn't happen
Forward fill	`df.ffill()`	Propagates potentially incorrect values
Drop nulls	`df.dropna()`	Loses data points, creates gaps
Fill with 0 ✓	`df.fillna(0)`	Semantically correct for sales

6.3 Missing Value Analysis

```
from pyspark.sql.functions import col, sum

# Count NULLs per column
df_daily.select([sum(col(c).isNull().cast("int")).alias(c)
                 for c in df_daily.columns]).show()
```

Example Output:

datum	M01AB	M01AE	N02BA	...
0	5	3	2	...

6.4 How This Enables ARIMA

Preprocessing	ARIMA Requirement
No NULL values	ARIMA cannot handle NaN in series
Complete time series	All time periods must have values
Zero-fill	Model learns that some days have no sales

---

7. Preprocessing Step 4: Aggregation for Analysis

7.1 Monthly Aggregation

```
# Monthly total sales for each drug category
monthly_sales = df_daily.groupBy("Year", "Month") \
    .sum("M01AB", "M01AE", "N02BA", "N02BE", "N05B", "N05C", "R03", "R06") \
    .orderBy("Year", "Month")
```

SQL Equivalent:

```
SELECT Year, Month,
       SUM(M01AB), SUM(M01AE), SUM(N02BA), ...
FROM df_daily
GROUP BY Year, Month
ORDER BY Year, Month
```

7.2 Weekly Trend Analysis

```
weekly_sales_trends = df_weekly.groupBy("datum") \
    .sum("M01AB", "M01AE", "N02BA", "N02BE", "N05B", "N05C", "R03", "R06")
```

7.3 Why Multiple Aggregations?

Aggregation	Business Purpose	Forecasting Use
Daily	Operational planning	Short-term (30-day) forecasts
Weekly	Procurement cycles	Weekly ordering patterns
Monthly	Financial planning	Budget projections

7.4 How This Enables ARIMA

Aggregation Level	ARIMA Consideration
Daily (used)	More data points = better parameter estimation
Weekly/Monthly	Smoother series, fewer observations
Choice: Daily	Balances granularity with statistical power

8. Preprocessing Step 5: Spark to Pandas Conversion

8.1 The Critical Conversion

```
# Convert Spark DataFrame to Pandas (required for statsmodels)
category_df = df_daily.select("datum", "M01AB").toPandas()
category_df.set_index('datum', inplace=True)
```

8.2 Why This Conversion is Necessary

Library	Required Input	Native Format
`statsmodels.ARIMA`	pandas Series with DatetimeIndex	pandas
PySpark	Spark DataFrame	Spark

Incompatibility:

```
# This will NOT work:
model = ARIMA(df_daily['M01AB'], order=(1,1,1))  # Spark column, fails!
```

```
# This works:
pandas_df = df_daily.toPandas()
model = ARIMA(pandas_df['M01AB'], order=(1,1,1))  # pandas Series, works!
```

8.3 Selecting Only Required Columns

```
# Only select what's needed for forecasting
category_df = df_daily.select("datum", "M01AB").toPandas()
```

Why Not Convert Entire DataFrame?

Approach	Memory	Time
`df_daily.toPandas()` (all columns)	High	Slow
`df_daily.select("datum", "M01AB").toPandas()`	**Low**	**Fast**

Memory Efficiency:

- Full DataFrame: 12 columns × 2,160 rows = ~25,920 values
- Selected: 2 columns × 2,160 rows = ~4,320 values
- **83% memory reduction**

8.4 Setting DateTime Index

```
category_df.set_index('datum', inplace=True)
```

Why Set Index?

Without Index	With Index
Date is regular column	Date is row identifier
ARIMA doesn't know time order	ARIMA understands temporal sequence
Cannot slice by date	Can slice: `df['2018':'2019']`

8.5 How This Enables ARIMA

Preprocessing	ARIMA Requirement
`.toPandas()`	statsmodels requires pandas

Preprocessing	ARIMA Requirement
Single column selection	ARIMA models univariate series
DateTime index	Temporal ordering for forecasting

9. Why ARIMA Was Chosen

9.1 Data Characteristics After Preprocessing

After all preprocessing steps, the data has:

Characteristic	Value	ARIMA Suitability
Format	pandas Series	☑ Required by statsmodels
Index	DatetimeIndex	☑ Temporal ordering
Missing values	None (filled with 0)	☑ Complete series
Granularity	Daily	☑ Sufficient observations
Type	Univariate	☑ ARIMA specialty

9.2 Why ARIMA is Appropriate

Advantage 1: Univariate Forecasting

The data is **single-variable** per drug category:

- Input: Historical sales of M01AB
- Output: Future sales of M01AB

ARIMA excels at univariate time series.

Advantage 2: Handles Non-Stationarity

The `d` parameter handles differencing:

- `d=1` converts non-stationary → stationary
- No manual preprocessing required

Advantage 3: Captures Autocorrelation

Sales today often correlate with sales yesterday:

- `p=1` captures autoregressive patterns
- `q=1` captures moving average effects

Advantage 4: Industry Standard

ARIMA is the **traditional baseline** for demand forecasting:

- Well-documented methodology
- Easy to explain to stakeholders
- Reliable for short-term predictions

9.3 ARIMA(1,1,1) Parameter Justification

```
model = ARIMA(category_df['M01AB'], order=(1, 1, 1))
```

Parameter	Value	Reasoning
p=1	AR(1)	Yesterday's sales influence today
d=1	First difference	Makes non-stationary data stationary
q=1	MA(1)	Yesterday's error influences today

ARIMA(1,1,1) Mathematical Model:

$$\Delta y_t = c + \phi_1 \Delta y_{t-1} + \theta_1 \epsilon_{t-1} + \epsilon_t$$

Where:

- $\Delta y_t = y_t - y_{t-1}$ (first difference)
- ϕ_1 = Autoregressive coefficient
- θ_1 = Moving average coefficient
- ϵ_t = White noise error

---

10. Alternative Models Considered

10.1 Why Not Prophet?

Aspect	Prophet	ARIMA
Seasonality	Automatic detection	Manual (SARIMA)
Setup complexity	Simple	Simple
Interpretability	Components plot	Coefficients
Verdict	Good alternative	**Chosen for simplicity**

Prophet would be better if:

- Strong yearly seasonality exists
- Holiday effects are important
- Multiple seasonalities overlap

10.2 Why Not LSTM?

Aspect	LSTM	ARIMA
Data requirement	Large datasets	Small to medium
Complexity	Neural network	Statistical
Interpretability	Black box	White box
Verdict	Overkill for ~2,160 daily points	**Appropriate scale**

10.3 Why Not SARIMA?

Aspect	SARIMA	ARIMA
Seasonality	Explicit modeling	None
Parameters	(p,d,q)(P,D,Q,m)	(p,d,q)
Complexity	Higher	**Lower**
Verdict	Use if strong seasonality	**Baseline choice**

SARIMA would be better if seasonal patterns were detected during analysis.

10.4 Model Selection Decision Tree

```
Time Series Data
      │
      ├── Strong Seasonality?
      │   YES → SARIMA or Prophet
      │   NO ↓
      │
      ├── Multiple Variables?
      │   YES → VAR or Machine Learning
      │   NO ↓
      │
      ├── Non-Stationary?
      │   YES → ARIMA with d > 0 ✓
      │   NO → ARIMA with d = 0
      │
      └── Result: ARIMA(1,1,1)
```

11. Preprocessing-Model Synergy

11.1 How Each Preprocessing Step Enables ARIMA

Preprocessing Step	Creates	ARIMA Benefit
PySpark loading	Validated DataFrame	Confirms data quality
Schema inference	Typed columns	Identifies numeric targets
Date conversion	DateType column	Proper temporal ordering
fillna(0)	Complete series	No NaN errors in model
Column selection	Univariate series	ARIMA input format
toPandas()	pandas DataFrame	statsmodels compatibility
set_index()	DatetimeIndex	Time series structure

11.2 The Complete Feature Flow

```
DBFS CSV Files
      │
      ▼
[PySpark Load] → Spark DataFrame (distributed)
      │
      ▼
[Schema Check] → Confirmed: datum, M01AB, ..., R06
      │
      ▼
[Date Convert] → datum: DateType (not string)
      │
      ▼
[fillna(0)] → No missing values
      │
      ▼
[select("datum", "M01AB")] → Only needed columns
      │
      ▼
[toPandas()] → pandas DataFrame
      │
      ▼
[set_index("datum")] → DatetimeIndex
      │
      ▼
ARIMA Input Ready: pandas Series with DatetimeIndex
```

11.3 Why This Pipeline is Optimal

```
┌─────────────────────────────────────────────────────────────┐
│                    PREPROCESSING BENEFITS                   │
├─────────────────────────────────────────────────────────────┤
```

```
| 1. PySpark handles enterprise data scale                        |
|    → Can process TB of data before converting to pandas         |
|                                                                 |
| 2. Date conversion enables time series                          |
|    → ARIMA requires proper temporal index                       |
|                                                                 |
| 3. Zero-fill is semantically correct                            |
|    → Missing sales = no sales = 0                               |
|                                                                 |
| 4. Selective column conversion saves memory                     |
|    → Only convert what ARIMA needs                              |
|                                                                 |
| 5. DateTime index required by statsmodels                       |
|    → ARIMA.forecast() uses index for date predictions           |
├─────────────────────────────────────────────────────────────────┤
|                    ARIMA BENEFITS                               |
├─────────────────────────────────────────────────────────────────┤
| 6. Handles differencing internally                              |
|    → d=1 makes non-stationary data stationary                   |
|                                                                 |
| 7. Simple, interpretable model                                  |
|    → Coefficients have clear meaning                            |
|                                                                 |
| 8. Fast training on pandas Series                               |
|    → Seconds to fit on ~2,160 observations                      |
|                                                                 |
| 9. Reliable short-term forecasts                                |
|    → 30-day horizon appropriate for inventory planning          |
└─────────────────────────────────────────────────────────────────┘
```

12. Conclusion

12.1 Summary of Preprocessing-Model Logic

Preprocessing Step	Technique	ARIMA Enablement
Data Loading	PySpark from DBFS	Validates data availability
Schema Check	`printSchema()`, `describe()`	Confirms numeric columns
Type Conversion	`to_date()`	Creates temporal index
Missing Values	`fillna(0)`	Complete series required
Aggregation	`groupBy().sum()`	Business-level analysis
Format Conversion	`toPandas()`	statsmodels compatibility
Index Setting	`set_index('datum')`	Time series structure

12.2 Why ARIMA is the Right Choice

1. **Univariate forecasting:** One drug category at a time
2. **Handles non-stationarity:** `d=1` differencing built-in
3. **Industry standard:** Proven for demand forecasting
4. **Simple implementation:** Fits in single line of code
5. **Interpretable:** Coefficients explain model behavior
6. **Fast:** Trains in seconds on 2,160 observations

12.3 The Complete Value Chain

```
DBFS Storage (Enterprise Data Lake)
        │
        ▼
[PySpark] → Distributed loading & preprocessing
        │
        ▼
[Schema Validation] → Confirm data types
        │
        ▼
[Date Conversion] → Enable temporal operations
        │
        ▼
[Zero Fill] → Complete time series
        │
        ▼
[Spark → Pandas] → Model-compatible format
        │
        ▼
[ARIMA(1,1,1)] → Learn temporal patterns
        │
        ▼
[30-Day Forecast] → Future sales predictions
        │
        ▼
Inventory & Budget Planning Decisions
```

12.4 When to Consider Alternatives

Scenario	Recommended Model
Baseline univariate forecasting	**ARIMA(1,1,1)** ✓
Strong yearly seasonality	SARIMA or Prophet
Multiple drug categories together	VAR (Vector AutoRegression)
Complex non-linear patterns	LSTM or XGBoost
Holiday effects important	Prophet

📊 This document explains the complete rationale from Databricks preprocessing to ARIMA model selection

Understanding the enterprise pipeline ensures reproducible pharmaceutical demand forecasting