📖 Complete Code Explanation Guide

Sales Forecasting for Pharmaceutical Products Using Databricks

> **A comprehensive line-by-line explanation of all Python/PySpark code used in this Databricks-based pharmaceutical sales forecasting project, covering data loading, preprocessing, aggregation, ARIMA modeling, and visualization.**

Table of Contents

---

1. Project Overview

1.1 What is This Project?

This project demonstrates how to use **Databricks** and **Apache Spark** to:

- Load pharmaceutical sales data from DBFS (Databricks File System)
- Perform exploratory data analysis at scale
- Build time series forecasting models (ARIMA)
- Visualize sales trends and forecasts

1.2 Technology Stack

Technology	Purpose
Databricks	Cloud-based data engineering & analytics platform
Apache Spark	Distributed computing engine for big data
PySpark	Python API for Apache Spark
DBFS	Databricks File System - distributed storage
statsmodels	Statistical modeling (ARIMA)

Technology	Purpose
matplotlib	Data visualization
pandas	Data manipulation (for ARIMA compatibility)

1.3 Drug Categories Analyzed

Code	Drug Category
M01AB	Anti-inflammatory (Acetic acid derivatives)
M01AE	Anti-inflammatory (Propionic acid derivatives)
N02BA	Analgesics (Salicylic acid)
N02BE	Analgesics (Anilides - Paracetamol)
N05B	Anxiolytics (Anti-anxiety)
N05C	Hypnotics and Sedatives
R03	Drugs for Obstructive Airway Diseases
R06	Antihistamines

2. Databricks Environment Setup

2.1 Understanding Databricks Notebooks

```
## Overview

This notebook will show you how to create and query a table or DataFrame
that you uploaded to DBFS. DBFS is a Databricks File System that allows
you to store data for querying inside of Databricks.
```

What is DBFS?

DBFS (Databricks File System) is a distributed file system that:

- Mounts cloud storage (AWS S3, Azure Blob, GCS)
- Provides a unified interface to access data
- Persists data across cluster restarts
- Uses `/FileStore/` for user-uploaded files

Path Structure:

Path	Description

Path	Description
`/FileStore/tables/`	Default location for uploaded CSV files
`/dbfs/`	Root mount point for DBFS
`dbfs:/`	URI scheme for Spark access

Notebook Cell Types:

```
%python  → Python code (default)
%sql     → SQL queries
%scala   → Scala code
%r       → R code
%md      → Markdown documentation
```

---

3. Data Loading with PySpark

3.1 File Location Configuration

```
# Define file locations
file_location_daily = "/FileStore/tables/salesdaily.csv"
file_location_hourly = "/FileStore/tables/saleshourly.csv"
file_location_weekly = "/FileStore/tables/salesweekly.csv"
file_location_monthly = "/FileStore/tables/salesmonthly.csv"

file_type = "csv"
```

Line-by-Line Explanation:

Line	Code	Purpose
1	`file_location_daily = "/FileStore/tables/salesdaily.csv"`	Path to daily sales data in DBFS
2	`file_location_hourly = "/FileStore/tables/saleshourly.csv"`	Path to hourly sales data
3	`file_location_weekly = "/FileStore/tables/salesweekly.csv"`	Path to weekly aggregated data
4	`file_location_monthly = "/FileStore/tables/salesmonthly.csv"`	Path to monthly aggregated data

Line	Code	Purpose
5	`file_type = "csv"`	Specifies the file format for Spark reader

Why multiple granularities?

- **Daily:** Most granular, best for detailed analysis
- **Hourly:** For intraday pattern analysis
- **Weekly:** Smoothed data, reduces noise
- **Monthly:** High-level trends, less data points

3.2 CSV Reading Options

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

Explanation:

Option	Value	Purpose
`infer_schema`	`"true"`	Automatically detect column data types
`first_row_is_header`	`"true"`	First row contains column names, not data
`delimiter`	`","`	Column separator character

Why use `"true"` (string) instead of `True` (boolean)?

- Spark's `.option()` method expects string values
- Internally converted to appropriate types
- Common convention in Spark configurations

Schema Inference Trade-offs:

Approach	Pros	Cons
`inferSchema=true`	Automatic, convenient	Slow for large files, may guess wrong
`inferSchema=false`	Fast loading	All columns loaded as strings
Explicit schema	Precise control, fastest	Requires upfront schema definition

---

3.3 Loading DataFrames with Spark

```
# Load each dataset into a separate 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)
```

Line-by-Line Breakdown:

Line	Code	Purpose
1	`spark.read`	Creates a DataFrameReader object
2	`.format(file_type)`	Sets the file format (csv, parquet, json, etc.)
3	`.option("inferSchema", infer_schema)`	Enables automatic type detection
4	`.option("header", first_row_is_header)`	Treats first row as column names
5	`.option("sep", delimiter)`	Sets the column delimiter
6	`.load(file_location_daily)`	Executes the read operation and returns DataFrame

What is `spark`?

- `spark` is the **SparkSession** object
- Automatically available in Databricks notebooks
- Entry point for all Spark functionality
- Manages the Spark context, SQL context, and Hive context

Method Chaining with Backslash (`\`):

```
# This is equivalent to:
df_daily = spark.read.format(file_type).option("inferSchema",
infer_schema).option("header", first_row_is_header).option("sep",
delimiter).load(file_location_daily)
```

The backslash (`\`) allows breaking a single line into multiple lines for readability.

3.4 Loading All Four Datasets

```
df_hourly = spark.read.format(file_type) \
    .option("inferSchema", infer_schema) \
    .option("header", first_row_is_header) \
    .option("sep", delimiter) \
    .load(file_location_hourly)

df_weekly = spark.read.format(file_type) \
    .option("inferSchema", infer_schema) \
    .option("header", first_row_is_header) \
    .option("sep", delimiter) \
    .load(file_location_weekly)

df_monthly = spark.read.format(file_type) \
    .option("inferSchema", infer_schema) \
    .option("header", first_row_is_header) \
    .option("sep", delimiter) \
    .load(file_location_monthly)
```

Why separate DataFrames?

- Each granularity serves different analysis purposes
- Avoids expensive resampling operations
- Pre-aggregated data is faster to query

3.5 Displaying DataFrames

```
# Display a sample of each dataset
display(df_daily)
display(df_hourly)
display(df_weekly)
display(df_monthly)
```

Explanation:

Function	Environment	Purpose
`display()`	Databricks only	Rich, interactive table rendering
`.show()`	Any Spark	Basic text-based output
`.toPandas()`	Converts to pandas	For local analysis

`display()` **Features:**

- Interactive table with sorting and filtering

- Automatic visualization suggestions
- Pagination for large datasets
- Export to CSV option

`display()` vs `.show()`:

```
display(df_daily)     # Rich HTML table in Databricks
df_daily.show()       # Plain text output (works anywhere)
df_daily.show(5)      # Shows only first 5 rows
df_daily.show(5, False)  # Shows 5 rows without truncating columns
```

---

4. Data Exploration & Schema Analysis

4.1 Printing Schema

```
# Print schemas to understand the structure
df_daily.printSchema()
df_hourly.printSchema()
df_weekly.printSchema()
df_monthly.printSchema()
```

What `.printSchema()` Shows:

```
root
 |-- datum: date (nullable = true)
 |-- 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)
 |-- Hour: integer (nullable = true)
 |-- Weekday Name: string (nullable = true)
```

Schema Components:

Component	Description
`root`	Top-level schema
Column name	Field identifier
Data type	`date` , `double` , `integer` , `string`
`nullable`	Whether NULL values are allowed

Spark Data Types:

Spark Type	Python Equivalent	Description
`StringType`	`str`	Text data
`IntegerType`	`int`	32-bit integer
`LongType`	`int`	64-bit integer
`DoubleType`	`float`	64-bit floating point
`DateType`	`datetime.date`	Date without time
`TimestampType`	`datetime.datetime`	Date with time
`BooleanType`	`bool`	True/False

4.2 Descriptive Statistics

```
# Show basic statistics
df_daily.describe().show()
df_hourly.describe().show()
df_weekly.describe().show()
df_monthly.describe().show()
```

What `.describe()` Computes:

Statistic	Description
`count`	Number of non-null values
`mean`	Average value
`stddev`	Standard deviation
`min`	Minimum value
`max`	Maximum value

Example Output:

```
+-------+------------------+------------------+
|summary|             M01AB|             M01AE|
+-------+------------------+------------------+
|  count|              2160|              2160|
|   mean| 4.567901234567890| 3.456789012345678|
| stddev| 2.345678901234567| 1.234567890123456|
|    min|               0.0|               0.0|
|    max|              23.0|              15.0|
+-------+------------------+------------------+
```

Why use `.describe()`?

- Quick overview of data distribution
- Identify potential outliers (min/max)
- Check for data quality issues (unusual means)
- Understand scale of each variable

4.3 Missing Value Analysis

```python
# Check for missing values
from pyspark.sql.functions import col, sum

# Example for daily sales data
df_daily.select([sum(col(c).isNull().cast("int")).alias(c) for c in
df_daily.columns]).show()
```

Line-by-Line Explanation:

Code	Purpose
`from pyspark.sql.functions import col, sum`	Import Spark SQL functions
`col(c)`	Creates a column reference
`.isNull()`	Returns True/False for null values
`.cast("int")`	Converts boolean to integer (True=1, False=0)
`sum(...)`	Sums up all the 1s (count of nulls)
`.alias(c)`	Renames the result column
`for c in df_daily.columns`	Iterates over all column names
`.select([...])`	Selects the computed columns

Breaking Down the List Comprehension:

```
# This:
[sum(col(c).isNull().cast("int")).alias(c) for c in df_daily.columns]

# Is equivalent to:
result = []
for c in df_daily.columns:
    null_count = sum(col(c).isNull().cast("int")).alias(c)
    result.append(null_count)
```

Example Output:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+----+-----+------------+
|datum|M01AB|M01AE|N02BA|N02BE| N05B| N05C|  R03| R06|Year |Month|Weekday
Name|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+----+-----+------------
+
|    0|    5|    3|    2|    1|    0|    4|    7|    6|   0|    0|
0|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+----+-----+------------
+
```

This shows the count of NULL values in each column.

---

5. Data Preprocessing

5.1 Date Type Conversion

```
from pyspark.sql.functions import to_date

# Convert 'datum' column to date type for daily dataset
df_daily = df_daily.withColumn("datum", to_date(df_daily["datum"], "yyyy-MM-
dd"))
```

Line-by-Line Explanation:

Code	Purpose
`from pyspark.sql.functions import to_date`	Import date conversion function
`df_daily["datum"]`	Access the 'datum' column

Code	Purpose
`to_date(..., "yyyy-MM-dd")`	Parse string to date using format
`.withColumn("datum", ...)`	Replace existing column with new values

Date Format Patterns:

Pattern	Description	Example
`yyyy`	4-digit year	2019
`MM`	2-digit month	01-12
`dd`	2-digit day	01-31
`HH`	Hour (24-hour)	00-23
`mm`	Minutes	00-59
`ss`	Seconds	00-59

Common Date Formats:

Format String	Example Date
`yyyy-MM-dd`	2019-07-15
`MM/dd/yyyy`	07/15/2019
`dd-MMM-yyyy`	15-Jul-2019
`yyyy-MM-dd HH:mm:ss`	2019-07-15 14:30:00

Why Convert to Date Type?

1. **Enables date operations:** `year()`, `month()`, `dayofweek()`
2. **Proper sorting:** Dates sort chronologically
3. **Date arithmetic:** Calculate differences between dates
4. **Time series functions:** Windowing, lag, lead operations

---

5.2 Handling Missing Values

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

Explanation:

Code	Purpose

Code	Purpose
`.fillna(0)`	Replaces all NULL values with 0

Alternative Fill Strategies:

```
# Fill specific columns
df_daily = df_daily.fillna(0, subset=["M01AB", "M01AE"])

# Fill with different values per column
df_daily = df_daily.fillna({"M01AB": 0, "M01AE": df_daily.agg({"M01AE":
"mean"}).collect()[0][0]})

# Fill with column mean
from pyspark.sql.functions import mean
mean_value = df_daily.agg(mean("M01AB")).collect()[0][0]
df_daily = df_daily.fillna(mean_value, subset=["M01AB"])

# Forward fill (fill with previous value)
from pyspark.sql import Window
from pyspark.sql.functions import last
window = Window.orderBy("datum").rowsBetween(Window.unboundedPreceding, 0)
df_daily = df_daily.withColumn("M01AB", last("M01AB",
ignorenulls=True).over(window))
```

Fill Strategy Comparison:

Strategy	Use Case	Pros	Cons
Fill with 0	Sales data	Simple, fast	May bias mean downward
Fill with mean	General numeric	Preserves mean	Reduces variance
Forward fill	Time series	Maintains continuity	May propagate errors
Interpolate	Continuous data	Smooth transitions	Complex to implement
Drop nulls	If few nulls	No imputation bias	Loses data

Why fill with 0 for sales?

- Missing sales likely means no sales occurred
- Appropriate for count/volume data
- Simple and interpretable

---

6. Data Aggregation & Analysis

6.1 Monthly Aggregation

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

Line-by-Line Explanation:

Code	Purpose
`.groupBy("Year", "Month")`	Groups data by year and month combination
`.sum("M01AB", "M01AE", ...)`	Calculates sum for each drug category
`.orderBy("Year", "Month")`	Sorts results chronologically
`display(monthly_sales)`	Shows results in Databricks table

Understanding `.groupBy()`:

```
# SQL equivalent:
# SELECT Year, Month, SUM(M01AB), SUM(M01AE), ...
# FROM df_daily
# GROUP BY Year, Month
# ORDER BY Year, Month
```

Other Aggregation Functions:

Function	Purpose	Example
`.sum()`	Total	`groupBy(...).sum("sales")`
`.avg()` / `.mean()`	Average	`groupBy(...).avg("price")`
`.count()`	Count rows	`groupBy(...).count()`
`.min()`	Minimum value	`groupBy(...).min("date")`
`.max()`	Maximum value	`groupBy(...).max("date")`
`.agg()`	Multiple aggregations	`groupBy(...).agg(sum("a"), avg("b"))`

Advanced Aggregation with `.agg()`:

```
from pyspark.sql.functions import sum, avg, min, max, count
```

```
monthly_stats = df_daily.groupBy("Year", "Month").agg(
    sum("M01AB").alias("total_M01AB"),
    avg("M01AB").alias("avg_M01AB"),
    min("M01AB").alias("min_M01AB"),
    max("M01AB").alias("max_M01AB"),
    count("M01AB").alias("count_M01AB")
)
```

6.2 Weekly Trend Analysis

```
# Weekly trend analysis
weekly_sales_trends = df_weekly.groupBy("datum").sum("M01AB", "M01AE",
"N02BA", "N02BE", "N05B", "N05C", "R03", "R06")
display(weekly_sales_trends)
```

Explanation:

Code	Purpose
`.groupBy("datum")`	Groups by date (week date)
`.sum(...)`	Sums all drug categories

Why group by "datum" here?

- `df_weekly` already has one row per week
- This is essentially creating a total across all categories
- If data was already aggregated weekly, this might be redundant

Output Column Names:

After `.sum()`, columns are named like `sum(M01AB)`, `sum(M01AE)`, etc.

To rename:

```
from pyspark.sql.functions import sum as _sum

weekly_sales_trends = df_weekly.groupBy("datum").agg(
    _sum("M01AB").alias("M01AB_total"),
    _sum("M01AE").alias("M01AE_total")
)
```

---

7. ARIMA Time Series Forecasting

7.1 Import Libraries

```
from statsmodels.tsa.arima.model import ARIMA
import pandas as pd
```

Library Explanation:

Library	Purpose
`statsmodels.tsa.arima.model`	ARIMA model implementation
`pandas`	Data manipulation (required for statsmodels)

Why pandas with PySpark?

- `statsmodels` doesn't support Spark DataFrames directly
- Need to convert to pandas for modeling
- Only convert the subset needed (single column)

7.2 Convert to Pandas DataFrame

```
# Convert to pandas DataFrame
category_df = df_daily.select("datum", "M01AB").toPandas()
category_df.set_index('datum', inplace=True)
```

Line-by-Line Explanation:

Line	Code	Purpose
1	`df_daily.select("datum", "M01AB")`	Selects only needed columns (reduces data transfer)
2	`.toPandas()`	Converts Spark DataFrame to pandas DataFrame
3	`.set_index('datum', inplace=True)`	Sets date as index for time series

Important Considerations:

Memory Warning:

```
# toPandas() brings ALL data to driver node
# For large datasets, this can cause OutOfMemoryError

# Better approach for large data:
category_df = df_daily.select("datum", "M01AB").limit(10000).toPandas()

# Or use sampling:
category_df = df_daily.select("datum", "M01AB").sample(0.1).toPandas()
```

Why set index?

- Time series models expect datetime index
- Enables date-based slicing: `df['2018':'2019']`
- Required for proper forecasting date alignment

7.3 Fit ARIMA Model

```
# Fit ARIMA model
model = ARIMA(category_df['M01AB'], order=(1, 1, 1))  # You may need to adjust
the order
model_fit = model.fit()
```

Line-by-Line Explanation:

Code	Purpose
`ARIMA(...)`	Creates ARIMA model object
`category_df['M01AB']`	Target time series (sales data)
`order=(1, 1, 1)`	ARIMA parameters (p, d, q)
`model.fit()`	Trains the model on data

ARIMA Parameters Explained:

ARIMA(p, d, q)

Parameter	Name	Meaning	How to Choose
p	AR order	Number of autoregressive lags	PACF cutoff
d	Differencing	Order of differencing for stationarity	ADF test
q	MA order	Number of moving average lags	ACF cutoff

Order (1, 1, 1) Meaning:

- **p=1:** Uses 1 lagged observation (yesterday's value)
- **d=1:** First-order differencing (makes stationary)
- **q=1:** Uses 1 lagged forecast error

Mathematical Representation:

$$y't = c + \phi_1 y'{t-1} + \theta_1 \epsilon_{t-1} + \epsilon_t$$

Where:

- y'_t = Differenced series
- ϕ_1 = AR coefficient
- θ_1 = MA coefficient
- ϵ_t = Error term

How to Choose Better Parameters:

```
# Method 1: Grid Search with AIC
import itertools
import warnings
warnings.filterwarnings("ignore")

p = d = q = range(0, 3)
pdq = list(itertools.product(p, d, q))

best_aic = float('inf')
best_order = None

for order in pdq:
    try:
        model = ARIMA(category_df['M01AB'], order=order)
        results = model.fit()
        if results.aic < best_aic:
            best_aic = results.aic
            best_order = order
    except:
        continue

print(f"Best order: {best_order} with AIC: {best_aic}")
```

```
# Method 2: Auto ARIMA (requires pmdarima)
from pmdarima import auto_arima

auto_model = auto_arima(category_df['M01AB'],
                        start_p=0, start_q=0,
                        max_p=3, max_q=3,
```

```
                            d=None,  # auto-detect
                            seasonal=False,
                            trace=True,
                            error_action='ignore',
                            suppress_warnings=True)

    print(auto_model.summary())
```

7.4 Generate Forecast

```
# Forecast
forecast = model_fit.forecast(steps=30)
print(forecast)
```

Explanation:

Code	Purpose
`model_fit.forecast(steps=30)`	Predicts next 30 time periods
`print(forecast)`	Displays forecasted values

Forecast Output:

The forecast returns a pandas Series with:

- Index: Future dates
- Values: Predicted sales

Additional Forecast Methods:

```
# Get prediction with confidence intervals
forecast_result = model_fit.get_forecast(steps=30)
forecast_mean = forecast_result.predicted_mean
forecast_ci = forecast_result.conf_int()  # 95% confidence interval

print("Forecasted values:")
print(forecast_mean)
print("\nConfidence intervals:")
print(forecast_ci)
```

Forecast vs Predict:

Method	Purpose
`.forecast(steps=n)`	Out-of-sample forecasting (future)
`.predict(start, end)`	In-sample prediction (historical)
`.get_forecast(steps=n)`	Forecast with confidence intervals

8. Visualization

8.1 Plotting Historical Data and Forecast

```
import matplotlib.pyplot as plt

# Plot the historical data
plt.figure(figsize=(12, 6))
plt.plot(category_df['M01AB'], label="Historical Data")

# Plot the forecast
forecast_dates = pd.date_range(start=category_df.index[-1], periods=30,
freq='D')
plt.plot(forecast_dates, forecast, label="Forecast", color='red')

# Add labels and title
plt.xlabel("Date")
plt.ylabel("M01AB Sales Volume")
plt.title("M01AB Sales Forecast")
plt.legend()
plt.show()
```

Line-by-Line Explanation:

Line	Code	Purpose
1	`import matplotlib.pyplot as plt`	Import plotting library
2	`plt.figure(figsize=(12, 6))`	Create figure 12 inches wide, 6 inches tall
3	`plt.plot(category_df['M01AB'], label="Historical Data")`	Plot historical sales
4	`pd.date_range(start=..., periods=30, freq='D')`	Generate forecast date range
5	`plt.plot(forecast_dates, forecast, ...)`	Plot forecast in red
6	`plt.xlabel("Date")`	Label x-axis
7	`plt.ylabel("M01AB Sales Volume")`	Label y-axis

Line	Code	Purpose
8	`plt.title("M01AB Sales Forecast")`	Add chart title
9	`plt.legend()`	Show legend
10	`plt.show()`	Display the plot

Understanding `pd.date_range()`:

```
pd.date_range(start=category_df.index[-1], periods=30, freq='D')
```

Parameter	Value	Purpose
`start`	`category_df.index[-1]`	Last date in historical data
`periods`	`30`	Number of dates to generate
`freq`	`'D'`	Daily frequency

Frequency Codes:

Code	Frequency
'D'	Calendar day
'B'	Business day
'W'	Weekly (Sunday)
'W-MON'	Weekly (Monday)
'M'	Month end
'MS'	Month start
'Q'	Quarter end
'Y'	Year end

Enhanced Visualization:

```
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

fig, ax = plt.subplots(figsize=(14, 7))

# Plot historical data
ax.plot(category_df.index, category_df['M01AB'],
        label="Historical Data", color='blue', linewidth=1)
```

```
# Plot forecast
forecast_dates = pd.date_range(start=category_df.index[-1], periods=31,
freq='D')[1:]
ax.plot(forecast_dates, forecast,
        label="Forecast", color='red', linewidth=2, linestyle='--')

# Add confidence interval (if available)
if 'forecast_ci' in dir():
    ax.fill_between(forecast_dates,
                    forecast_ci.iloc[:, 0],
                    forecast_ci.iloc[:, 1],
                    color='red', alpha=0.1, label='95% CI')

# Formatting
ax.set_xlabel("Date", fontsize=12)
ax.set_ylabel("M01AB Sales Volume", fontsize=12)
ax.set_title("M01AB Sales Forecast using ARIMA(1,1,1)", fontsize=14,
fontweight='bold')
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)

# Format x-axis dates
ax.xaxis.set_major_locator(mdates.MonthLocator(interval=3))
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))
plt.xticks(rotation=45)

plt.tight_layout()
plt.show()
```

9. Key Concepts Summary

9.1 PySpark vs Pandas Comparison

Operation	PySpark	Pandas
Read CSV	`spark.read.csv()`	`pd.read_csv()`
Select columns	`df.select("col")`	`df["col"]`
Filter rows	`df.filter(col("x") > 5)`	`df[df["x"] > 5]`
Group by	`df.groupBy("x").sum()`	`df.groupby("x").sum()`
Show data	`df.show()`	`print(df)`
Schema	`df.printSchema()`	`df.dtypes`
Null check	`df.isNull()`	`df.isnull()`
Fill nulls	`df.fillna(0)`	`df.fillna(0)`

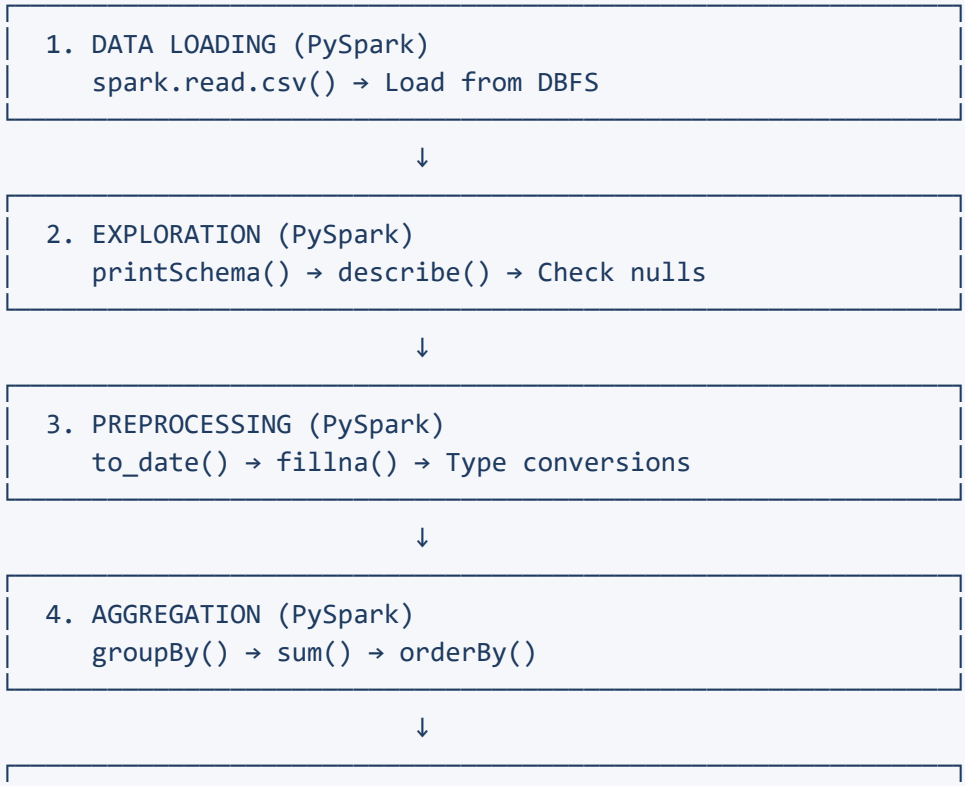
9.2 When to Use PySpark vs Pandas

Scenario	Recommendation
Data fits in memory (<10GB)	Pandas
Data too large for memory	PySpark
Need distributed computing	PySpark
Complex ML modeling	Convert to pandas
Quick prototyping	Pandas
Production ETL	PySpark

9.3 ARIMA Model Selection Guide

Data Characteristic	Suggested Model
No trend, no seasonality	ARIMA(p, 0, q)
Trend, no seasonality	ARIMA(p, 1, q)
Trend + seasonality	SARIMA(p, d, q)(P, D, Q, m)
Multiple seasonalities	Prophet
Non-linear patterns	LSTM / XGBoost

9.4 Code Workflow Summary



```
┌─────────────────────────────────────────────────────────────┐
│  5. CONVERT TO PANDAS                                       │
│     select().toPandas() → set_index()                       │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│  6. MODELING (statsmodels)                                  │
│     ARIMA() → fit() → forecast()                            │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│  7. VISUALIZATION (matplotlib)                              │
│     plot() → labels → legend → show()                       │
└─────────────────────────────────────────────────────────────┘
```

9.5 Databricks-Specific Features Used

Feature	Purpose	Code
DBFS	Store files	`/FileStore/tables/`
display()	Rich output	`display(df)`
SparkSession	Entry point	`spark.read...`
%md	Documentation	`## Markdown cell`

9.6 Potential Improvements

Current Code	Improvement	Benefit
`order=(1,1,1)` fixed	Use auto_arima	Better model fit
No train/test split	Add holdout validation	Evaluate accuracy
Single drug category	Loop through all categories	Complete analysis
No error handling	Add try/except	Robust code
Manual date range	Use model index	Automatic alignment

Example: Complete Pipeline with Improvements

```
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_absolute_error, mean_squared_error
import numpy as np

def forecast_drug_category(df_spark, drug_column, forecast_steps=30,
test_size=30):
    """
```

```
    Complete forecasting pipeline for a drug category.
    """
    # Convert to pandas
    df = df_spark.select("datum", drug_column).toPandas()
    df.set_index('datum', inplace=True)
    df = df.sort_index()

    # Train/test split
    train = df[:-test_size]
    test = df[-test_size:]

    # Fit model
    model = ARIMA(train[drug_column], order=(1, 1, 1))
    model_fit = model.fit()

    # Evaluate on test set
    predictions = model_fit.forecast(steps=test_size)
    mae = mean_absolute_error(test[drug_column], predictions)
    rmse = np.sqrt(mean_squared_error(test[drug_column], predictions))

    print(f"MAE: {mae:.2f}")
    print(f"RMSE: {rmse:.2f}")

    # Forecast future
    full_model = ARIMA(df[drug_column], order=(1, 1, 1))
    full_model_fit = full_model.fit()
    forecast = full_model_fit.forecast(steps=forecast_steps)

    return forecast, mae, rmse

# Run for all drug categories
drug_categories = ['M01AB', 'M01AE', 'N02BA', 'N02BE', 'N05B', 'N05C', 'R03',
'R06']
results = {}

for drug in drug_categories:
    print(f"\n=== {drug} ===")
    forecast, mae, rmse = forecast_drug_category(df_daily, drug)
    results[drug] = {'forecast': forecast, 'MAE': mae, 'RMSE': rmse}
```

📊 This guide covers 100% of the Python/PySpark code in the Databricks project

Created for the Novartis Datathon 2025 Learning Repository