utils\_helpers\_model\_context.py

* *import polars as pl* : polars is a fast pandas like library for working with tables of data in python.
* *def rename\_cols(df: pl.DataFrame) -> pl.DataFrame* : This creates a function named rename\_cols that expects you to pass in a Polars table (a DataFrame). It will return you a new table with some column names changed.
* We are just doing long name short name mapping :

A screenshot of a computer

AI-generated content may be incorrect.

Top\_model\_context.py

**Purpose:**  
The class analyzes transactional data (like sales), finds top performers (brands, customers, SKUs, etc.), and generates human-readable text summaries.

**Input data:**

A Polars DataFrame with columns like:

* brand (brand of product)
* sku\_name (product name/code)
* customer\_name (who bought)
* branch (store/region)
* date (transaction date)
* quantity (number of items sold)
* value (money made from the sale)

(But it’s robust: if columns are missing, it’ll say so in the summary!)

WORKING

**1. Initialization (\_\_init\_\_)**

* You create an instance:  
  analyzer = PerformanceAnalyzer(df) (where df is your data; must be a Polars DataFrame).
* It stores the data as self.df.
* It prepares an empty list self.summaries to store summary texts.
* It calls a private method \_generate\_summaries() which does all the heavy work.

2. **Generate Summaries (\_generate\_summaries)**

**Here's what it calculates:**

**a) Top Brands**

* By Revenue (sum of value)
* By Quantity (sum of quantity)

**b) Top Customers**

* By Volume (sum of quantity)

**c) Top Regions**

* By Revenue (sum of value), using the branch column

**d) Top SKUs (Stock Keeping Units, ie, products)**

* By Revenue
* By Quantity

**For each:**

* If any needed column(s) are missing, it writes a warning/notice in summaries.
* Otherwise, it finds the top N (e.g., top 10 or 15) entries, gets their totals, and formats this as a text bullet list.

**3. Finding Top Performers (\_get\_top\_performers)**

This helper method does the DataFrame math:

* Groups the data (e.g., by brand), sums up the relevant column (value or quantity).
* Sorts them (descending, so highest first).
* Limits to the top n (default: 10).
* Returns the result as a list of dictionaries, each with the grouping key (name) and value.

**Optionally**, you can filter by date range (if you pass a date\_range).

Region\_normalizer.py : convert all region names to uppercase

Region\_model\_context.py: The CustomerAnalyzer class is designed to **analyze sales transaction data** and **summarize who the top customers are**, both **globally** and **regionally**, based on sales revenue (value).

**How does it work?**

**1. Initialization (\_\_init\_\_)**

* When you create a CustomerAnalyzer, you pass in a Polars DataFrame (df) with transaction data.
* If the DataFrame has a branch column (which represents the sales region), it **removes any rows where branch is missing or an empty string**.
* The cleaned DataFrame is stored in self.df.
* It then calls \_generate\_summaries() to **create readable summaries** of the top customers.

**2. Generating Summaries (\_generate\_summaries)**

This method does the heavy lifting. It precomputes summary texts and stores them in self.summaries. Here are the main steps:

**A. Checks if there’s data**

* If the filtered DataFrame is empty, adds "No customer data available to generate regional summaries." to summaries and returns immediately.

**B. Global Top Customers**

* Checks if both customer\_name and value columns exist; otherwise, explains what's missing.
* If they exist, calls \_get\_top\_customers() **without a region filter** (region="All Regions") to get the top 10 customers by total revenue globally.
* Formats the output as:
* Global top customers by revenue:
* If none found, adds an appropriate message.

**C. Regional Top Customers**

* Checks if all needed columns (branch, customer\_name, value) exist; otherwise, explains what's missing.
* It finds all unique regions by looking at the branch column.
* For the first 3 unique regions (as a demonstration/sample), it:
  + Calls \_get\_top\_customers() for that region.
  + Summarizes the results similarly, labeled by region, for example:
  + If no results, lets you know for that particular region.

**3. Finding Top Customers (\_get\_top\_customers)**

This helper method does the core data math:

* Checks if **all required columns** are present.
* If a region is specified (not "All Regions"), filters to just that region.
* If filtered data is empty, returns an empty result.
* Groups the data by customer\_code (handles situations where a customer may appear with variations in name but unique codes).
* For each customer, it:
  + Takes the first customer\_name (as the primary display name),
  + Sums up their revenue (value column; missing values are treated as 0),
  + Takes the first region (for reference).
* Sorts the results descending by the total revenue.
* Limits to the top N results (default 10).
* Returns the results as a list of Python dictionaries, which makes it easy to format into readable summaries later.

Cross\_sell\_model\_Context.py:

We are finding "what products are often bought together" patterns (that’s “market basket analysis”), using the **Apriori algorithm**, which can help drive bundling, recommendations, and targeted marketing.It does:

* + Creates a mapping of product IDs to product names (if available)
  + Runs the Apriori algorithm and gets rules
  + Formats rules into readable plain English
  + Generates general summaries/stats

Chatbot\_model\_context3.py:

This script helps the chatbot to answer a wide range of **sales data questions using an advanced LLM and data science methods**.

What it doe :

Analyzes sales data (from a Parquet file)

Cleans, normalizes, and standardizes the data behind the scenes

Uses analytics models to find **top brands, customers, cross-selling rules…**

Can plot timeseries, multi-brand charts, forecast weekly sales, and much more

Understands your query by running it through a large language model (LLM, via Groq)

Outputs **clear answers, relevant charts, or summary insights**

Is designed to be expandable and robust, handling missing or messy columns gracefully

**SalesDataChatbot: THE MAIN CLASS**

**Init – Setting up your data environment:**

* Reads the Parquet file as a Polars DataFrame (pl\_df)
* **Cleans column names** (standardizes, possibly gets rid of accidental spaces).
* **Strips whitespace** from brand and branch columns, standardizes regions to UPPERCASE using normalize\_region\_name.
* **Casts** the date column to Date type for proper time filtering.
* **Maps alternative column names** to “value” (sales) and “quantity” (units sold).
* **Finds available brands and regions**—important for query **autocorrect** later!
* **Initializes model helpers** for:
  + **Performance analysis** (best brands, SKUs, regions, etc.)
  + **Customer analysis** (top customers, by region and globally)
  + **Cross-selling analysis** (what products are bought together)
* **Aggregates all context text** so the LLM has sales summaries to “refer to”.

**LLM (Language Model) Setup:**

* Reads the Groq API key from your .env file (using dotenv)
* Sets up the connection to Groq (the LLM provider)

**C. Asking the Chatbot (How Queries Are Processed)**

**1. User asks a question.**

**2. The chatbot uses the LLM to deeply understand “what is the user asking for?”**

**3. It categorizes the query into these intents:**

* Simple sales calculation (total sales for X, Y, or a date range)
* Weekly forecast plot for a brand
* Plotting (single brand/region, multi-brand, or multi-region comparisons)
* General data questions

**4. LLM is prompted with comprehensive instructions, the names of valid brands/regions, and a current user query to extract:**

* **Intent**
* **Brand(s)**
* **Region(s)**
* **Date window**
* **Metric to plot**

**5. Depending on detected intent, the chatbot:**

* Calculates sales
* Plots data
* Forecasts weekly sales (with advanced Holt-Winters smoothing, including outlier removal)
* Plots multiple brands or regions for comparison
* (Or, for general/complex questions, defers to the LLM with a context summary)

**The results (text or plot) are returned for the user.**

**D. Plotting Functions**

**Plotting is highly flexible:**

* **Single Brand/Region Plot:** Timeseries of sales or quantity for a brand or region
* **Multi-Brand Plot:** Compare several brands on one chart (optionally in a region)
* **Multi-Region Plot:** Compare a brand across several regions in one chart
* **Weekly Brand Forecast Plot:** Holt-Winters exponential smoothing (with outlier-cleaning and auto-parameter search)

**E. Helper/Support Functions**

* **Date Parsing:** Robustly understands many date formats.
* **Clean Columns:** Autocorrect and standardize messy or alternative column names.
* **Region and Brand Normalization:** To deal with spelling/case inconsistencies.
* **Sales Calculation:** Filters data by given brand, region, and/or date, and sums sales value.

**F. Integration with the Modeling Tools**

We import and use our own classes/modules for **PerformanceAnalyzer, CustomerAnalyzer, CrossSellingAnalyzer,** and helpers for normalization.  
**This means** that any insights, summaries, or cross-sell rules these models generate feed right into the chatbot’s natural language answers (giving it data-aware “intelligence”).

App\_model\_context2.py:

Workflow of chatbot:

* **On App Start:** It loads and preprocesses data once (from Parquet file, e.g. "sales\_rt.parquet").
* **User asks question…**  
  → Chatbot **analyzes the question** (finds intent, brands/regions/dates mentioned)  
  → **Performs requested analytics/plot**  
  → **Returns the answer** as text or plot
* **The chat is interactive and remembers prior Q&A history.**