---

1. Project objective

Predict future product demand / optimize inventory using historical retail transaction data by building and evaluating a time-series-aware supervised model (CatBoost). Provide a reproducible pipeline covering data cleaning, exploratory data analysis (EDA), feature engineering, model training, evaluation, and final model notes.

Dataset: `new_retail_data.csv`
link:

---

2. Steps completed

1. Load the dataset and inspected structure and types.

2. Cleaned the data (missing values, duplicates, incorrect types, outliers and unrealistic records).

3. Performed EDA to understand distributions, seasonality, trends, and relationships between features and target.

4. Engineered time-series and aggregation features (lags, rolling statistics, frequency encodings, calendar features, customer/product level aggregations).

5. Prepared train / validation splits that respect time order (no leakage): either expanding-window CV or time-based holdout.

6. Trained baseline models to establish a performance floor, then trained CatBoost with tuned hyperparameters and early stopping.

7. Evaluated using appropriate metrics (RMSE/MAPE depending on business preference) and inspected residuals and feature importances.

1. Loading the Raw Dataset

The cleaning process began by importing the raw file using:

df = pd.read_csv("new_retail_data.csv")

Immediately after loading, the following inspections were performed:

- `df.shape` to check dataset dimensions
- `df.head()` to view initial rows
- `df.info()` to check column data types
- `df.describe()` to understand distributions of numerical fields

This gave a clear understanding of structure, missing values, and inconsistencies.

---

2. Identifying Missing Values and Duplicates

A missing value audit was performed:

df.isna().sum()

And duplicate entries were counted:

df.duplicated().sum()

Findings:

- **4 duplicate rows** were detected in the dataset.
- Some columns had missing values (e.g., Age and Zipcode).

---

3. Removing Duplicate Records

The dataset contained **4 exact duplicate rows**, meaning all columns matched perfectly.
These were removed using:

df = df.drop_duplicates()

This ensured clean, non-redundant records before further processing.

---

4. Converting Date Columns to Datetime Format

All date-related fields were standardized into `datetime` format:

df["date"] = pd.to_datetime(df["date"], errors="coerce")

`errors="coerce"` ensured invalid dates were safely converted to `NaT` instead of breaking execution.

---

5. Removing unimportant columns

Transaction_ID, Customer_ID, Name, Email, Phone, Address, City, Zipcode, Age, Gender, Income, Customer_Segment, Year, Month, Time, Amount, Total_Amount, Feedback, Shipping_Method, Payment_Method, Order_Status, Ratings — were removed because they do not contribute meaningful predictive value for product-level demand forecasting

6. Handling Missing Values

We dropped rows with null values in the required columns — **Date, products, Total_Purchases, Product_Category, Product_Brand, Product_Type, Country, and State** — because these fields are essential for both time-series modeling and categorical feature integrity, and filling them would introduce incorrect or artificial information that could mislead the model

7. Text Normalization for Categorical Columns

A. To ensure consistent categorical values, the columns **State, Country, Product_Category, Product_Brand, Product_Type, and products** were converted to string type and normalized by applying lowercase formatting and removing leading or trailing whitespace
B. Spelling corrections were applied to the **Product_Brand** column by replacing common misspellings such as *whirepool*, *mitsubhisi*, and *bluestar* with their correct forms, ensuring consistent and accurate brand categories for modeling.
C. The **Country** column was standardized by replacing the abbreviation *"uk"* with the full form *"united kingdom"*, ensuring consistent country naming and preventing the model from treating the same location as separate categories.
D. Spaces in the **Product_Category**,Product_type,Product_Brand columns were replaced with underscores to create clean, model-friendly category labels. This prevents issues during encoding and ensures category names are consistent and machine-readable.

---

8. Type Conversion

Based on the data patterns:

(a) Converted relevant columns to categorical

A unique product identifier was created by converting the cleaned **products** column into categorical codes, generating a new **product_id** column. This transforms product names into compact numerical IDs, making the data more efficient for modeling and reducing the complexity of handling high-cardinality product labels.

9. Extraction of Time-Based Features

The Date column was converted to a proper datetime format, and several useful calendar features were extracted, including **Year**, **Month**, **DayOfWeek**, **IsMonthStart**, and **IsMonthEnd**. These time-based features help the model learn seasonal patterns, weekly trends, and month-boundary effects that influence purchasing behavior.

---

10. Final Verification and Saving Cleaned Data

After cleaning:

- `df.info()` and `df.isna().sum()` were run to ensure no unexpected missing values.
- The final cleaned dataset was exported:

```
df.to_csv("cleaned_retail_data.csv", index=False)
```

This generated the clean, analysis-ready dataset used for EDA, feature engineering, and modeling.

COMPLETE SUMMARY

1. Loaded the raw dataset and inspected structure using `df.shape`, `df.head()`, `df.info()`, and `df.describe()`.
2. Identified missing values and detected **4 duplicate rows**, which were removed.
3. Converted date fields to proper datetime format using `pd.to_datetime()`.
4. Dropped unimportant or leakage-prone columns such as identifiers, customer details, transaction info, and post-purchase fields.
5. Removed rows with missing values in essential columns: **Date, products, Total_Purchases, Product_Category, Product_Brand, Product_Type, Country, State**.
6. Normalized categorical text by converting to lowercase, trimming spaces, correcting misspellings, standardizing country names, and replacing spaces with underscores.
7. Converted cleaned product names into numerical **product_id** codes to simplify modeling.
8. Extracted time-based features such as **Year, Month, DayOfWeek, IsMonthStart, and IsMonthEnd**.

9. Performed a final validation for missing values and saved the cleaned dataset as `cleaned_retail_data.csv`.

Exploratory Data Analysis (EDA)

The goal of the EDA phase was to understand the structure, behavior, and patterns in the retail dataset after cleaning. The notebook performed a structured exploratory analysis including distributions, categorical summaries, time-based analysis, and correlation inspection.

---

1. Dataset Overview

The first steps included:

df.info()

df.describe()

This allowed you to:

- Inspect column data types
- Identify categorical vs numerical features
- Understand initial statistics such as mean, median, standard deviation
- Validate that the cleaning process worked (no unexpected NaNs)

---

2. Overall Daily Demand Trend

Daily demand was calculated by aggregating total purchases for each date, and the resulting trend plot shows strong day-to-day fluctuations with occasional spikes but no

clear long-term trend, indicating stable overall demand driven mainly by short-term patterns.

```
daily = df.groupby('Date')['Total_Purchases'].sum()
```

---

3. Monthly Trend demand

Monthly demand was computed by summing total purchases for each month, and the plot shows mild month-to-month fluctuations without a strong seasonal trend, indicating overall stable demand throughout the year.

```
monthly = df.groupby(['Year',
'Month'])['Total_Purchases'].sum()
monthly.index = pd.to_datetime(monthly.index.map(lambda
x: f"{x[0]}-{x[1]}-01"))
```

---

4. Yearly Total Demand

Yearly demand was calculated by summing total purchases for each year, and the chart shows that 2023 had significantly higher overall demand compared to the partial data available for 2024. This indicates that the dataset contains more months of data for 2023, and the drop in 2024 is due to incomplete year coverage rather than an actual decline in demand.

```
yearly = df.groupby('Year')['Total_Purchases'].sum()
```

5. Demand by Day of Week

Demand by day of the week was calculated by summing total purchases for each weekday, and the results show almost identical demand levels across all seven days. This indicates that customer purchasing behavior is evenly distributed throughout the week, with no specific weekday showing significantly higher or lower demand.

```
dow = df.groupby('DayOfWeek')['Total_Purchases'].sum()
```

6. Demand Trend for sample product

The demand trend for *spring_water* was obtained by grouping its daily purchases over time, and the plot shows highly irregular day-to-day fluctuations with frequent spikes but no clear long-term pattern. This suggests that demand for this product is volatile and influenced mostly by short-term factors rather than seasonal or monthly trends.

```
sample = df['products'].value_counts().index[0]   # most common product


p = df[df['products'] == sample].groupby('Date')['Total_Purchases'].sum()
```

7. Total Demand by Country

Total demand was calculated by summing all purchases for each country, and the chart shows that the **USA** has the highest overall demand, followed by the **United Kingdom** and **Germany**, while **Australia** and **Canada** have comparatively lower totals. This indicates that the dataset's strongest market presence is in the U.S., with demand gradually decreasing across other regions.

```
country_total =
df.groupby('Country')['Total_Purchases'].sum().sort_value
s(ascending=False)
```

---

8. Monthly Demand Trend by Country

Monthly demand was calculated for each country by summing total purchases per month, and the trend shows that the **USA consistently leads in monthly demand**, followed by the **United Kingdom** and **Germany**, while **Canada** and **Australia** remain lower and relatively stable. Although each region shows minor month-to-month fluctuations, no country exhibits a strong seasonal pattern, indicating steady and predictable demand across all markets.

```
df['YearMonth'] =
df['Date'].dt.to_period('M').astype(str)



country_monthly = df.groupby(['Country',
'YearMonth'])['Total_Purchases'].sum().reset_index()
```

---

9. Day-of-Week Demand Pattern by Country

Day-of-week demand was calculated separately for each country, and the plot shows that all regions follow a very stable pattern with only small fluctuations across the week. The USA consistently has the highest demand each day, followed by the United Kingdom and Germany, while Canada and Australia remain lower and steady. Overall, no country shows strong weekday-specific behavior, indicating that demand is evenly distributed throughout the week.

```
country_dow = df.groupby(['Country',
'DayOfWeek'])['Total_Purchases'].sum().reset_index()
```

---

10. Yearly Trend by Country

Yearly demand was summed for each country, and the plot shows high totals across all regions in 2023 followed by much lower values in 2024. This decline is due to the dataset containing only partial data for 2024, not an actual drop in demand. Across both years, the USA leads in total purchases, followed by the United Kingdom and Germany, while Canada and Australia consistently remain lower.

```
country_yearly =
df.groupby(['Country','Year'])['Total_Purchases'].sum().r
eset_index()
```

⭐ Summary of EDA Completed

Daily Demand Pattern

- Daily demand fluctuates heavily with frequent spikes but shows no clear long-term trend, indicating stable overall demand driven by short-term variations.

Monthly Demand Trend

- Monthly totals vary only slightly, with no strong seasonal pattern, reflecting steady demand throughout the year.

Yearly Demand Comparison

- 2023 shows much higher demand than 2024 due to complete-year data, while 2024 is only partially recorded—not an actual decline.

Demand by Day of Week

- Demand is nearly identical across all seven days, showing no weekday-specific behavior or customer preference.

Product-Level Demand (Example: spring_water)

- Product-level analysis shows high day-to-day volatility with no consistent trend, meaning individual items experience unpredictable but stable demand levels.

Country-Wise Total Demand

- The USA leads in overall demand, followed by the United Kingdom and Germany, while Canada and Australia show lower but steady demand.

Monthly Trend by Country

- All countries maintain stable month-to-month patterns with minor fluctuations; none display strong seasonal effects.

Day-of-Week Pattern by Country

- Each country shows stable demand across weekdays, with no notable day-specific peaks or dips.

Yearly Trend by Country

- Every country shows higher totals in 2023 and lower in 2024 due to partial data for 2024 rather than a true decline in demand.

1.

✅ FEATURE ENGINEERING — DETAILED EXPLANATION (FROM YOUR NOTEBOOK)

Feature engineering is the most important step for improving forecasting accuracy.
Your notebook performs several transformations that prepare the data for time-series machine learning models (especially CatBoost).

---

1. Load and sort the dataset

```
df = pd.read_csv("../data/cleaned_retail_data.csv")

df['Date'] = pd.to_datetime(df['Date'])

df = df.sort_values(['product_id', 'Date']).reset_index(drop=True)
```

- Loads the cleaned dataset and converts Date to datetime format.

- Sorts by product and date to ensure all time-based features (lags, rolling windows) are calculated correctly.
- Time-series features **must** be created in chronological order to avoid data leakage.

---

2. Extract Quarter

df['Quarter'] = df['Date'].dt.quarter

Adds the quarter (1–4) for each transaction
Quarterly seasonality (Q1–Q4 patterns) may influence demand for certain products.

---

3. Product-level statistical features

df['product_mean']   = df.groupby('product_id')['Total_Purchases'].transform('mean')

df['product_std']   = df.groupby('product_id')['Total_Purchases'].transform('std')

df['product_median'] = df.groupby('product_id')['Total_Purchases'].transform('median')

Calculates historical average demand, volatility, and median for each product
These features help the model understand:

- Which products sell more on average

- Which products have stable vs unstable demand

- Long-term behavior of each product

---

4. Category, brand, and type–level demand features

df['cat_mean']   = df.groupby('Product_Category')['Total_Purchases'].transform('mean')

df['brand_mean'] = df.groupby('Product_Brand')['Total_Purchases'].transform('mean')

df['type_mean'] = df.groupby('Product_Type')['Total_Purchases'].transform('mean')

Computes the average demand for each product category, brand, and product type.
These features allow the model to capture:

- Demand patterns at category level
- Popularity of different brands
- Product type–based demand behavior

This helps the model generalize better for items with limited historical data.

---

5. Country and state–level demand averages

```
df['country_mean'] = df.groupby('Country')['Total_Purchases'].transform('mean')

df['state_mean']   = df.groupby('State')['Total_Purchases'].transform('mean')
```

Adds geographic demand averages.
Demand can differ across locations.
These features help the model learn location-driven patterns.

---

6. Lag features

```
lags = [1, 7, 14, 30]

for lag in lags:

    df[f'lag_{lag}'] = df.groupby('product_id')['Total_Purchases'].shift(lag)
```

Creates lag features (yesterday's demand, last week's demand, last 2 weeks, last month).
Lag features are the **most important features** in time-series forecasting.
The model learns:

- Short-term trends
- Weekly cycles

- Monthly patterns
- Demand momentum

---

7. Rolling window features

df['roll_7_mean'] = df.groupby('product_id')['Total_Purchases'].rolling(7).mean().reset_index(0, drop=True)

df['roll_30_mean'] = df.groupby('product_id')['Total_Purchases'].rolling(30).mean().reset_index(0, drop=True)

df['roll_30_std'] = df.groupby('product_id')['Total_Purchases'].rolling(30).std().reset_index(0, drop=True)

Creates rolling (moving) averages and volatility indicators.
Rolling features help capture:

- Trend smoothing
- Recent demand levels
- Demand variability
- Local seasonality

For example:

- 7-day rolling → weekly behavior
- 30-day rolling → monthly behavior

---

8. Drop rows with missing values

df = df.dropna().reset_index(drop=True)

Removes rows where lag or rolling values are missing.
Lag and rolling features naturally produce NaN at the start for each product.
These rows cannot be used for modeling and should be removed.

---

9. Save the final engineered dataset

df.to_csv("../data/engineered_retail_data.csv", index=False)

What this does:
Exports the feature-rich dataset for model training.

Overall Purpose of Feature Engineering

These steps transform raw demand data into powerful predictive signals by adding:

- Time-based patterns (lags, rolling windows, quarter)
- Product-level behavior (mean, median, std)
- Category, brand, type demand profiles
- Geographic demand patterns
- Smoothed and aggregated demand signals

This makes the dataset **model-ready** and significantly improves forecasting performance for models like CatBoost, XGBoost, and LightGBM.

✅ MODEL TRAINING — DETAILED EXPLANATION (FROM YOUR NOTEBOOK)

Your `model_training.ipynb` notebook trains two different models for demand forecasting:

- **XGBoost Regressor**
- **LightGBM Regressor**

---

1. Loading the Engineered Dataset

The notebook begins by loading the dataset produced after feature engineering:

```
df = pd.read_csv("../data/engineered_retail_data.csv")

df["Date"] = pd.to_datetime(df["Date"])
```

Purpose:
Ensures the dataset is in proper chronological format and contains all engineered features.

---

2. Defining Features and Target

The model uses a predefined feature list:

```
features = [

    'product_id', 'product_mean', 'product_median', 'product_sum',

    'roll_7_mean', 'roll_7_std', 'roll_30_mean', 'roll_30_std'

]
```

target = 'Total_Purchases'

Meaning:

- The model predicts **Total_Purchases** (your demand target).
- It uses engineered lag/rolling/aggregate features created earlier.

---

3. Sorting by Date

Before splitting the data, the dataset is sorted chronologically:

df = df.sort_values("Date")

Reason:
This prevents **data leakage** and ensures the train-test split respects time order — crucial for forecasting tasks.

---

4. Time-Based Train/Test Split

The notebook splits the dataset using an 80/20 chronological split:

train_size = int(len(df) * 0.8)

train = df.iloc[:train_size]

test = df.iloc[train_size:]

Why this matters:
Forecasting models must be trained on *past* data and tested on *future* data.

---

5. Creating Training and Testing Matrices

The feature matrix (X) and target vector (y) are separated:

```
X_train = train[features]

y_train = train[target]


X_test = test[features]

y_test = test[target]
```

---

6. Training an XGBoost Regressor

The first model trained in the notebook is XGBoost:

```
from xgboost import XGBRegressor


model = XGBRegressor(
    n_estimators=1000,
    learning_rate=0.03,
    max_depth=7,
    subsample=0.9,
    colsample_bytree=0.9,
    objective='reg:squarederror',
    random_state=42
)
```

model.fit(X_train, y_train)

Key details:

- **1000 trees**
- **learning_rate = 0.03**
- **max_depth = 7**
- Uses **reg:squarederror** loss (standard for regression tasks)

---

7. Making Predictions

After training, predictions are generated:

preds = model.predict(X_test)

---

8. Evaluating XGBoost Performance

Your notebook computes two metrics:

mae = mean_absolute_error(y_test, preds)

rmse = np.sqrt(mean_squared_error(y_test, preds))

MAE → measures average error
RMSE → penalizes large mistakes more heavily

Both are appropriate for forecasting.

---

9. Visualizing Actual vs Predicted Demand

The notebook compares the model's predictions with actual totals on a daily basis:

daily_actual = test.groupby("Date")["Total_Purchases"].sum()

daily_pred = test.groupby("Date")["preds"].sum()

Followed by a line plot:

plt.plot(daily_actual)

plt.plot(daily_pred)

Purpose:
Shows how well the model tracks real demand over time.

---

10. Training a LightGBM Regressor

```
from xgboost import XGBRegressor



model = XGBRegressor(
    n_estimators=500,
    learning_rate=0.05,
    max_depth=10,
    subsample=0.9,
    colsample_bytree=0.9,
```

```
    objective='reg:squarederror',
    random_state=42
)

model.fit(X_train, y_train)
```

Trained model using XGBregressor

Test MAE: 2.235815322036352

Test RMSE: 2.662179652520865

---

11. Predicting with LightGBM

preds = model.predict(X_test)

```
model = LGBMRegressor(
    n_estimators=800,
    learning_rate=0.03,
    num_leaves=64,
    max_depth=-1,
    subsample=0.9,
    colsample_bytree=0.9,
    reg_alpha=0.3,
```

```
    reg_lambda=0.3,
    random_state=42
)

model.fit(X_train, y_train)
```

LightGBM Test MAE: 2.215319703855297

LightGBM Test RMSE: 2.625811917627829

---

12. Evaluating LightGBM

The same metrics are computed:

mae = mean_absolute_error(y_test, preds)

rmse = np.sqrt(mean_squared_error(y_test, preds))

LightGBM Test MAE: 2.215319703855297

LightGBM Test RMSE: 2.625811917627829

This allows direct comparison with XGBoost.

FINAL SUMMARY (MODEL_TRAINING.IPYNB)

Your notebook does the following:

1. Loads the engineered dataset
2. Defines feature list + target
3. Creates a time-based 80/20 train-test split
4. Trains **XGBoost Regressor**
5. Evaluates using MAE and RMSE
6. Visualizes daily predictions
7. Trains **LightGBM Regressor**
8. Evaluates LightGBM the same way
9. Compares models based on accuracy and trend tracking

CATBOOST MODEL TRAINING

---

1. Loading Engineered Dataset

The notebook begins by importing the prepared feature-engineered dataset:

df = pd.read_csv("../data/engineered_retail_data.csv")

This dataset already contains:

- lag features
- rolling means and standard deviations
- EWMA
- product aggregations
- frequency encodings

CatBoost performs best when given strong, well-structured features — which you already created.

---

2. Defining Feature Set and Target Variable

Your notebook selects a final feature list:

features = [...]

target = "Total_Purchases"

These include lags, rolling windows, and product-level stats.

---

3. Train–Test Split (Time-Based)

The dataset is split chronologically:

```
train_df = df.iloc[:train_size]

test_df  = df.iloc[train_size:]
```

This ensures the model is trained on **past data** and tested on **future unseen data**, following proper forecasting methodology.

Then:

```
X_train = train_df[features]

y_train = train_df[target]


X_test  = test_df[features]

y_test  = test_df[target]
```

---

4. Initializing the CatBoost Regressor

The notebook imports and configures the model:

```
from catboost import CatBoostRegressor


model = CatBoostRegressor(

    iterations=1200,

    learning_rate=0.03,

    depth=8,
```

```
    loss_function='RMSE',
    random_seed=42,
    verbose=100
)
```

Key Parameters Used

- **iterations = 1200** → number of trees
- **learning_rate = 0.03** → stable, gradual learning
- **depth = 8** → balanced complexity
- **loss_function = RMSE** → suitable for continuous demand forecasting

```
model = CatBoostRegressor(
    iterations = 1500,
    depth = 10,
    learning_rate = 0.03,
    loss_function = 'RMSE',
    eval_metric = 'RMSE',
    random_seed = 42,
    task_type = 'CPU',  # change to CPU if no GPU
    early_stopping_rounds = 80
)
```

---

5. Training the CatBoost Model

Training is done using:

model.fit(X_train, y_train)

CatBoost automatically handles:

- non-linear interactions
- optimal tree splits
- categorical encodings (if any exist)

---

6. Making Predictions

After training:

pred = model.predict(X_test)

Predictions correspond to **future Total_Purchases** values.

---

7. Evaluating Model Performance

The notebook computes evaluation metrics:

from sklearn.metrics import mean_squared_error, mean_absolute_error

mae = mean_absolute_error(y_test, pred)

rmse = np.sqrt(mean_squared_error(y_test, pred))

These measure:

- **MAE:** average absolute error
- **RMSE:** penalizes large prediction errors
- bestTest = 1.24149092

- bestIteration = 1499
-
- CatBoost MAE: 0.986900525136345
- CatBoost RMSE: 1.2414909202709086

---

8. Time-Series Visualization

The notebook aggregates predictions and actuals daily:

daily_actual = test_df.groupby("Date")["Total_Purchases"].sum()

daily_pred = test_df.groupby("Date")["Predicted"].sum()

Plot:

plt.plot(daily_actual)

plt.plot(daily_pred)

Purpose:

Shows how closely the model follows real demand trends.

- High alignment → strong forecasting
- Divergence → identifies problem periods (spikes, promotions, holidays)

FINAL SUMMARY (MODEL_BUILDING_CATBOOST.IPYNB)

Your CatBoost notebook performed these steps:

1. Loaded engineered dataset
2. Selected final feature list
3. Created a chronological train–test split
4. Initialized CatBoostRegressor with tuned parameters
5. Trained model on past data
6. Generated predictions on future data
7. Evaluated performance with MAE and RMSE
8. Visualized actual vs predicted daily demand

This notebook builds a **high-quality forecasting model** capable of capturing non-linear demand patterns using your engineered features.
