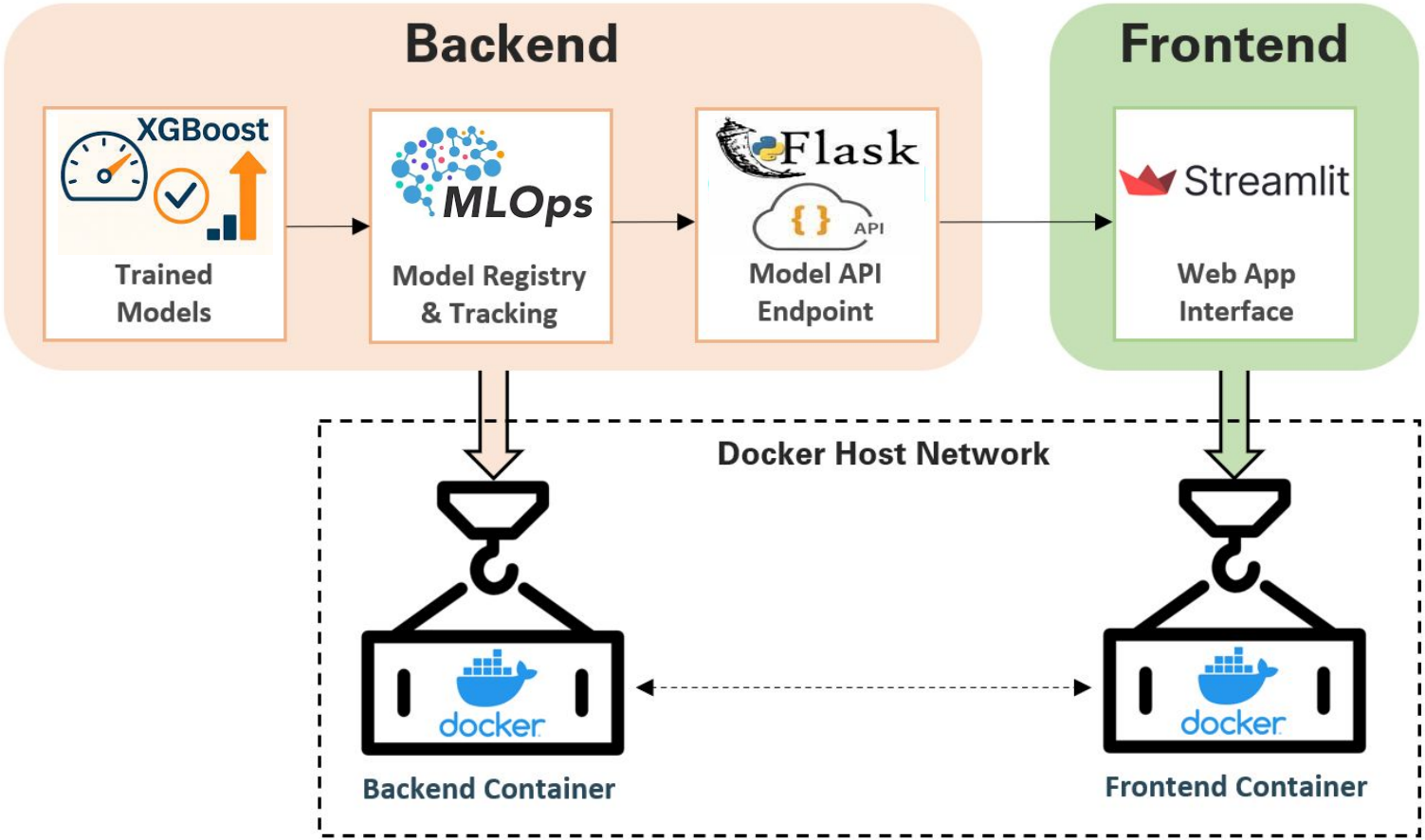
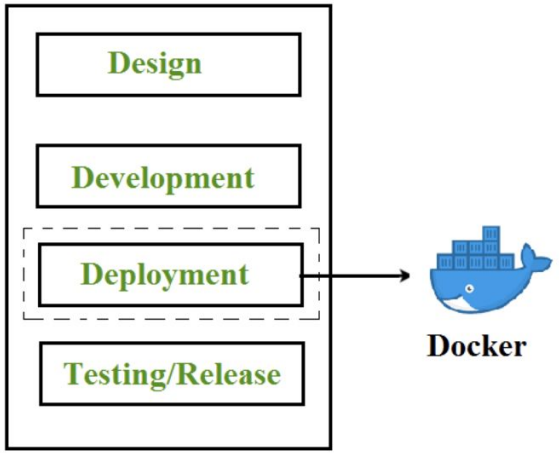
Containerization & Model Deployment in Production



Contents

- Executive Summary
- Business Problem Overview and Solution Approach
- Data Overview and Exploratory Data Analysis
- Data Preprocessing
- Model Building
- Model Performance Improvement: Hyperparameter Tuning
- Model Comparison, Final Model Selection, and Serialization
- Deployment - Backend
- Deployment - Frontend
- Actionable Insights and Recommendations
- Conclusion and Business Recommendations



Executive Summary -Project Background and Objectives

- SuperKart is a multi-format retail chain (Supermarkets, Departmental Stores, Food Marts) with diverse product mix across city tiers.
- Objective: Use data to **predict product–store sales** and guide pricing, assortment, and inventory.
- Dataset: 8,763 rows × 12 features (product, store, target = Product_Store_Sales_Total); clean (no missing/duplicate).
- Audience: Business + technical leaders; focus on **actionable insights** and **operational deployment**.

Executive Summary - Business Problem We Solved

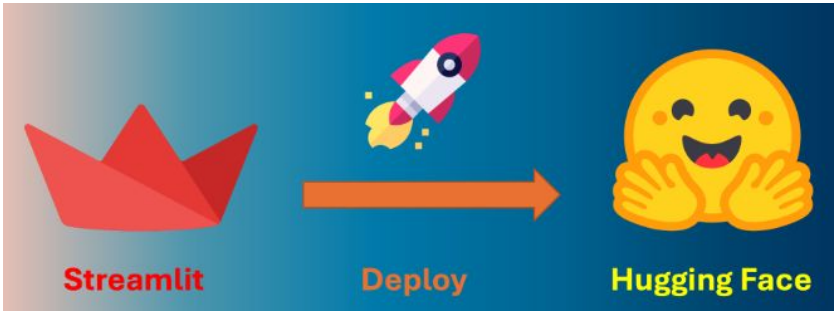
- Challenge: **inaccurate demand planning** created overstock/understock, margin leakage, and lost sales.
- Need: Identify **key revenue drivers** (price, category, store context) and **forecast product–store sales**.
- Outcomes sought: **Better pricing**, **assortment focus**, **replenishment accuracy**, and **store-level playbooks**.

Executive Summary - Solution Approach and Methodology

- Framework: **CRISP-DM** (Business Understanding → Data Understanding/EDA → Modeling → Evaluation → Deployment).
- EDA: found **MRP (price)** is the strongest driver; **weight/shelf area** weak; winners = **F&V, Snacks, Frozen**; **Large/Tier-1 stores** show higher per-product sales; **OUT004** is flagship.
- Modeling: Compared **Decision Tree** vs **XGBoost**; selected **XGBoost** for best generalization.
- Deployment: Serialized pipeline (preprocess + model) behind a **Flask API**, containerized with **Docker**, and exposed via **Hugging Face Space**; **Streamlit** FE for business users.

Executive Summary - Tools, Techniques and Technologies

- **Data & EDA**: Python, pandas, numpy, matplotlib/plotly.
- **Modeling**: scikit-learn (DecisionTreeRegressor), **XGBoost Regressor**, one-hot encoding pipeline, train/test split, KPI evaluation (R^2, RMSE, MAE, MAPE).
- **Serving**: Flask REST API, joblib serialization, **Docker** container.
- **UI**: **Streamlit** web app for interactive predictions.
- **Hosting**: **Hugging Face Spaces** (Docker runtime).



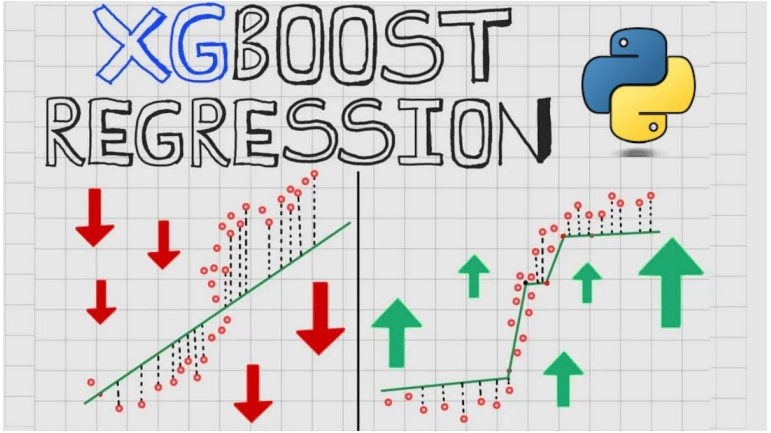
Executive Summary - Key Outcomes: Model and Insights

- **Model of record**: **XGBoost** with **$R^2 \approx 0.668$**, **RMSE ≈ 616**, **MAE ≈ 485**, **MAPE ≈ 18.7%** on test set.

- Decision Tree (tuned) underfit substantially; XGBoost remained **stable and robust**.

- Business drivers: **Price** is the dominant lever; **Large/Tier-1 stores** deliver higher per-product sales; top categories = **F&V, Snacks, Frozen**.

- Planogram implication: **allocated shelf area alone does not predict sales** → optimize space by **sales per facing/sq-inch**.



Executive Summary - Business Benefits

- **Inventory**: Improved forecasts enable lower **stockouts** on fast movers and reduced **waste** in perishables.
- **Margin**: Targeted price tests on low-elasticity SKUs; markdown science on slow movers.
- **Assortment**: Expand winners, trim long-tail SKUs; replicate **OUT004** playbook to similar stores.
- **Speed-to-action**: Front-end app lets managers run **what-if scenarios** without code; consistent, reproducible scoring via API.

Executive Summary - Conclusions and Next Steps

Conclusions

- **XGBoost** is the right production model for current features; Decision Tree serves as an interpretable baseline only.
- Pricing and store context (size, tier) drive outcomes more than packaging/area.
- Focus growth on **winning categories** and **high-performing store archetypes**.

Next Steps (High Impact)

- **Data expansion**: Add promotions, holidays/seasonality, inventory/stockouts, footfall, local weather & competitor price to cut error further.
- **Model enhancements**: Broader hyperparameter tuning (learning_rate, depth, min_child_weight, regularization) and weekly time series features; consider LightGBM/CatBoost.
- **Ops & MLOps**: Monitor KPI & drift, quarterly retrain, store-aware cross-validation; roll out price tests and assortment rationalization with measurable targets.

Business Problem Overview

Context:

- SuperKart is a retail supermarket chain operating multiple store formats (Supermarkets, Departmental Stores, Food Marts).
- Dataset: ~8,700 product–store combinations across categories like Frozen Foods, Dairy, Health, Canned, etc.

Business Challenge:

- Sales vary significantly across products, stores, and locations.
- Factors influencing sales: Product attributes (weight, sugar content, MRP), store attributes (size, type, age, city tier).
- Management struggles to forecast demand and optimize inventory allocation effectively.
- This leads to:
- Overstocking → higher holding costs and wastage.
- Understocking → missed sales opportunities and poor customer satisfaction.

Problem Statement:

- How can we leverage data to predict product-store sales more accurately, so SuperKart can optimize pricing, inventory management, and product placement?

Solution Approach and Methodology

Data Preparation

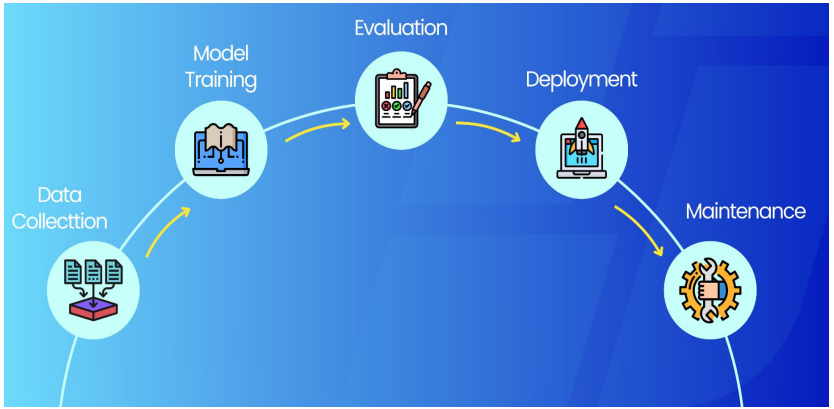
- Cleaned and processed 8,700+ rows of sales, product, and store attributes.
- Handled missing values, categorical encoding, and feature engineering.

Exploratory Data Analysis (EDA)

- Identified key drivers: Product MRP, Store Size, Store Location Tier, Product Type.
- Observed trends: Larger stores in Tier 1 cities with high-MRP products drive higher sales.

Model Development

- Applied machine learning regression models to predict Product_Store_Sales_Total.
- Compared two models Decision Tree and XGBoost among various available for use
- Selected better-performing model based on R^2 score and RMSE.



Deployment

- Built a low-code deployment pipeline for operationalizing predictions.
- Provides store managers with actionable sales forecasts for planning.

Data Overview

Dataset Size: 8,763 rows × 12 columns

Attributes:

- **Product Features:** Weight, Sugar Content, Allocated Shelf Area, Type, MRP (price).
- **Store Features:** Store ID, Establishment Year, Size (Small/Medium/High), City Tier, Store Type.
- **Target Variable:** Product_Store_Sales_Total (sales for a product–store combination).

Data Quality:

- **Missing Values:** None across all 12 attributes.
- **Duplicates:** None found.

Statistical Highlights:

- Product Weight: 4–22 kg (avg. ~12.6 kg).
- Product MRP: ₹31–₹266 (avg. ₹147).
- Store Establishment Year: 1987–2009.
- Sales per Product-Store: Range ₹33 – ₹8,000 (avg. ₹3,464).

```
data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8763 entries, 0 to 8762
Data columns (total 12 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Product_Id                 8763 non-null   object
 1   Product_Weight             8763 non-null   float64
 2   Product_Sugar_Content      8763 non-null   object
 3   Product_Allocated_Area     8763 non-null   float64
 4   Product_Type               8763 non-null   object
 5   Product_MRP                8763 non-null   float64
 6   Store_Id                   8763 non-null   object
 7   Store_Establishment_Year   8763 non-null   int64
 8   Store_Size                 8763 non-null   object
 9   Store_Location_City_Type   8763 non-null   object
 10  Store_Type                 8763 non-null   object
 11  Product_Store_Sales_Total  8763 non-null   float64
dtypes: float64(4), int64(1), object(7)
memory usage: 821.7+ KB
```

Data Background and Contents

Source: SuperKart retail chain data, covering product–store level transactions.

Scope: 8,763 records, 12 attributes describing products, stores, and sales outcomes.

Data Attributes

- **Product Features:**
  - *Product_Id, Product_Weight, Sugar_Content, Allocated_Area, Product_Type, MRP (price)*
- **Store Features:**
  - *Store_Id, Establishment_Year, Size, Location City Tier, Store_Type*
- **Target Variable:**
  - *Product_Store_Sales_Total* – revenue generated per product–store combination

Key Characteristics

- Time span: Stores established between **1987–2009**.
- Balanced mix of **product categories (16 types)** and **store formats (4 types)**.
- **No missing or duplicate values**, ensuring strong data reliability for modeling.

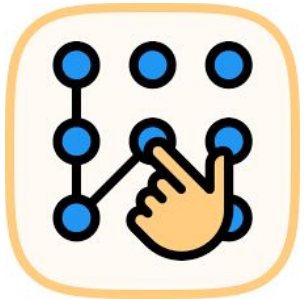
Exploratory Data Analysis- Univariate Analysis Overview

Key Observations:

- **Product_Weight:** Normally distributed around 12–14 kg. Few extreme values but no severe outliers.
- **Product_Allocated_Area:** Skewed toward lower values; most products occupy <0.1 allocation.
- **Product_MRP:** Moderate right-skew; many items priced around ₹140–₹170, but a tail exists toward higher MRPs.
- **Product_Store_Sales_Total (Target):** Roughly normal distribution, centered ~₹3,400, with upper sales spikes near ₹8,000.

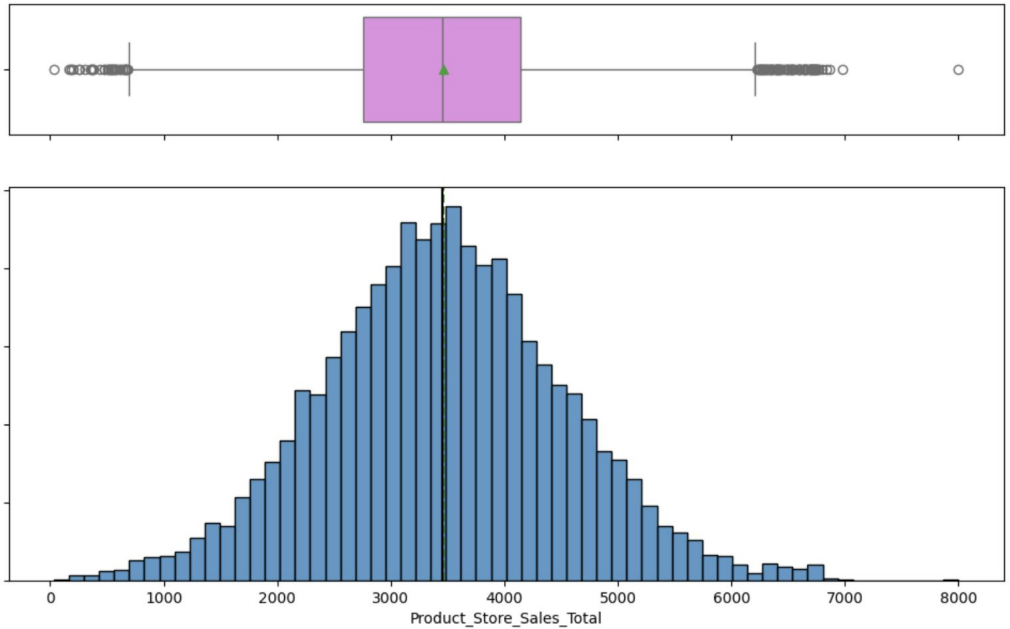
Categorical Variables:

- **Sugar Content:** Mostly “Low Sugar” (~55%) and “Regular” (~30%), with smaller “No Sugar” and “High Sugar” categories.
- **Product Type:** 16 categories; Fruits and Vegetables, Snack Foods, Frozen Foods are most frequent.
- **Store Size:** Majority Medium-sized (~70%).
- **City Tier:** Tier 2 stores dominate (~71% of entries).
- **Store Type:** “Supermarket Type2” makes up over half the records.



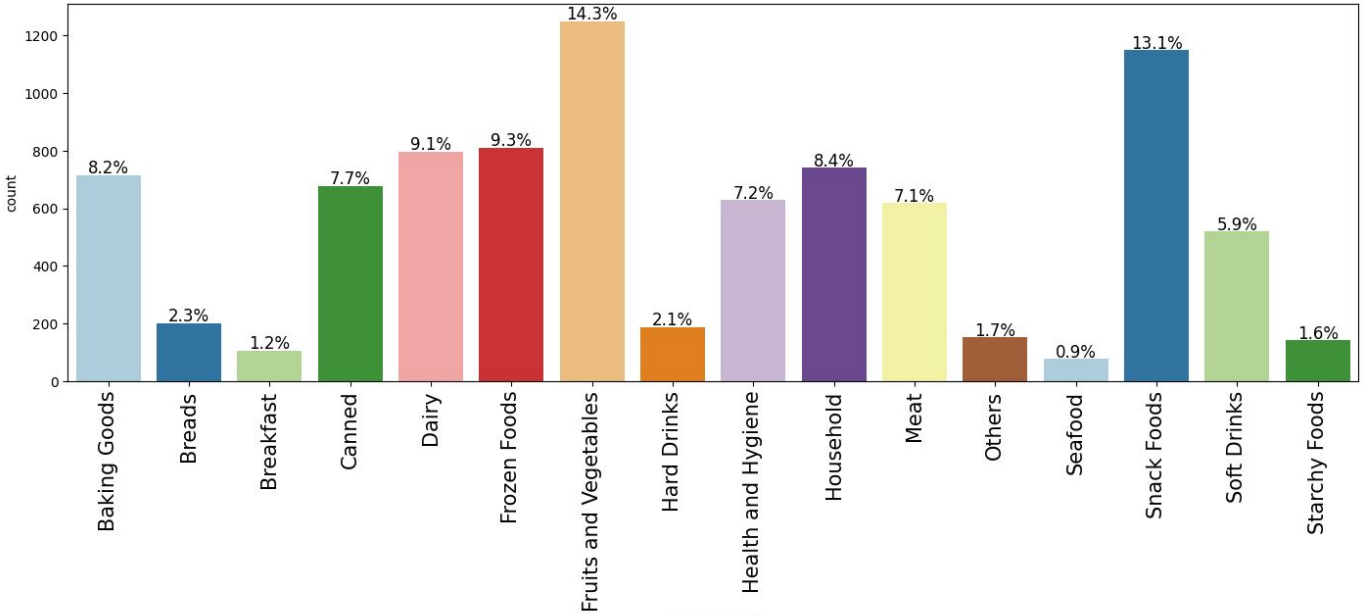
Univariate Analysis Example
Product_Store_Sales_Total (Target Variable)

- Approx. **normal distribution**, mean around ₹3,450.
- Majority of sales fall between **₹2,700–₹4,200**.
- Outliers with sales >₹7,000 show presence of **fast-moving products**.
- Indicates **predictable sales trend** with few extreme performers.



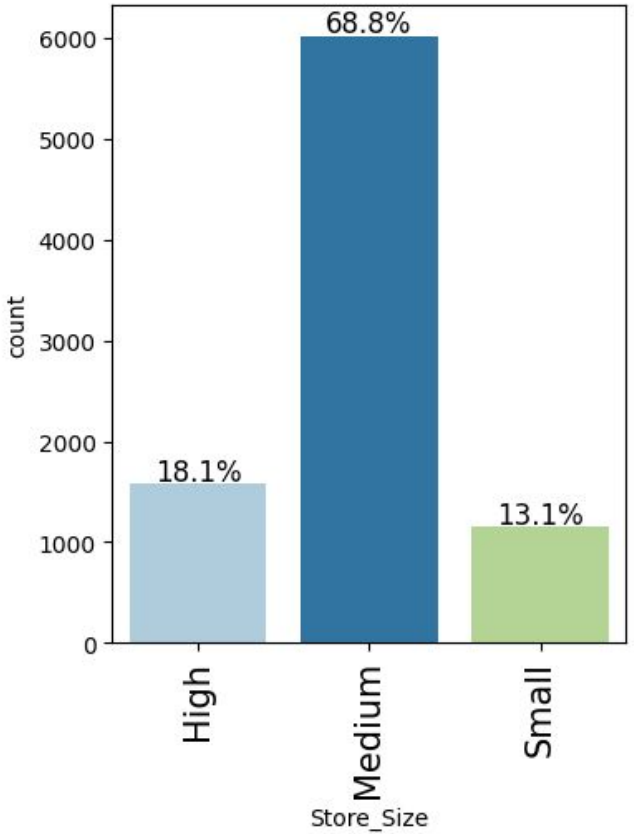
Univariate Analysis Example - Product_Type

- 16 categories; **Fruits and Vegetables, Snack Foods, Frozen Foods** lead in count.
- Smaller categories (like Breakfast, Seafood) exist but are less frequent.
- Implies a **broad product mix**, but sales concentrated in food essentials/snacks.

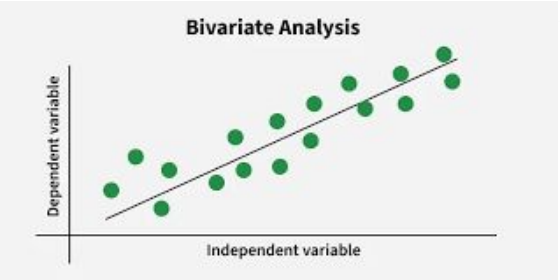


Univariate Analysis Example - Store_Size

- **Medium-sized stores** dominate (~70%).
- Smaller representation from **Small and High** stores.
- Indicates SuperKart's **core footprint is medium-sized outlets**.



Exploratory Data Analysis- Bivariate Analysis Overview



Product Factors vs. Sales:

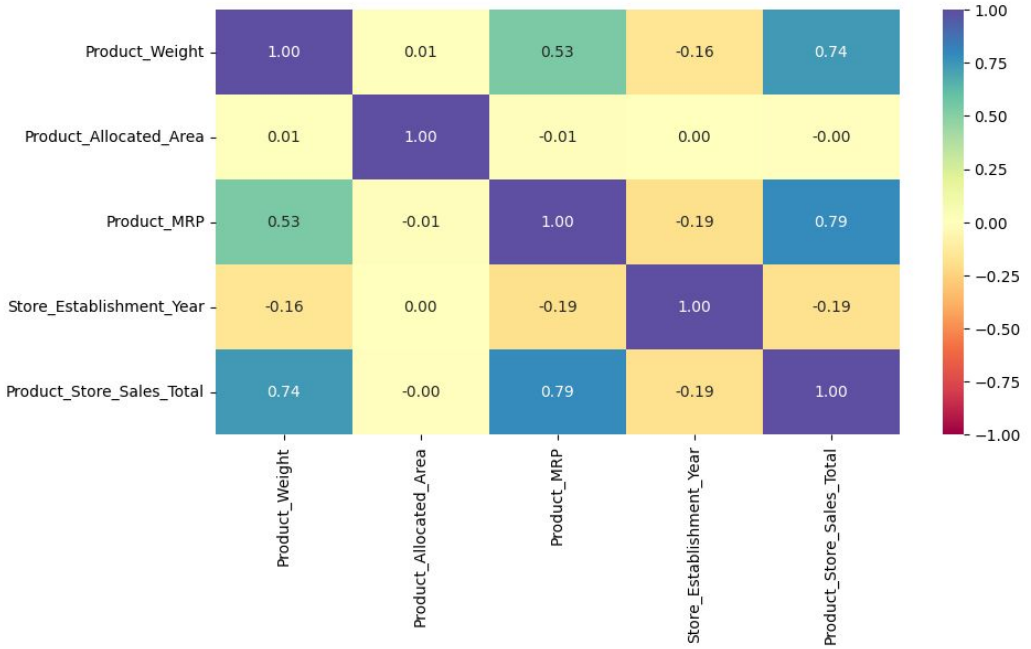
- **Higher MRP** products generally show higher sales values.
- **Sugar Content:** “Regular” and “Low Sugar” products sell more compared to niche (No/High Sugar).
- **Product Type:** Perishables (Fruits, Vegetables, Dairy) dominate sales volumes.

Store Factors vs. Sales:

- **Store Size:** Larger stores (High) drive higher sales per product than Medium or Small.
- **City Tier:** Tier 1 cities yield higher sales per product compared to Tier 2 and Tier 3.
- **Store Type:** Supermarket Type2 and Type1 outperform Food Marts.

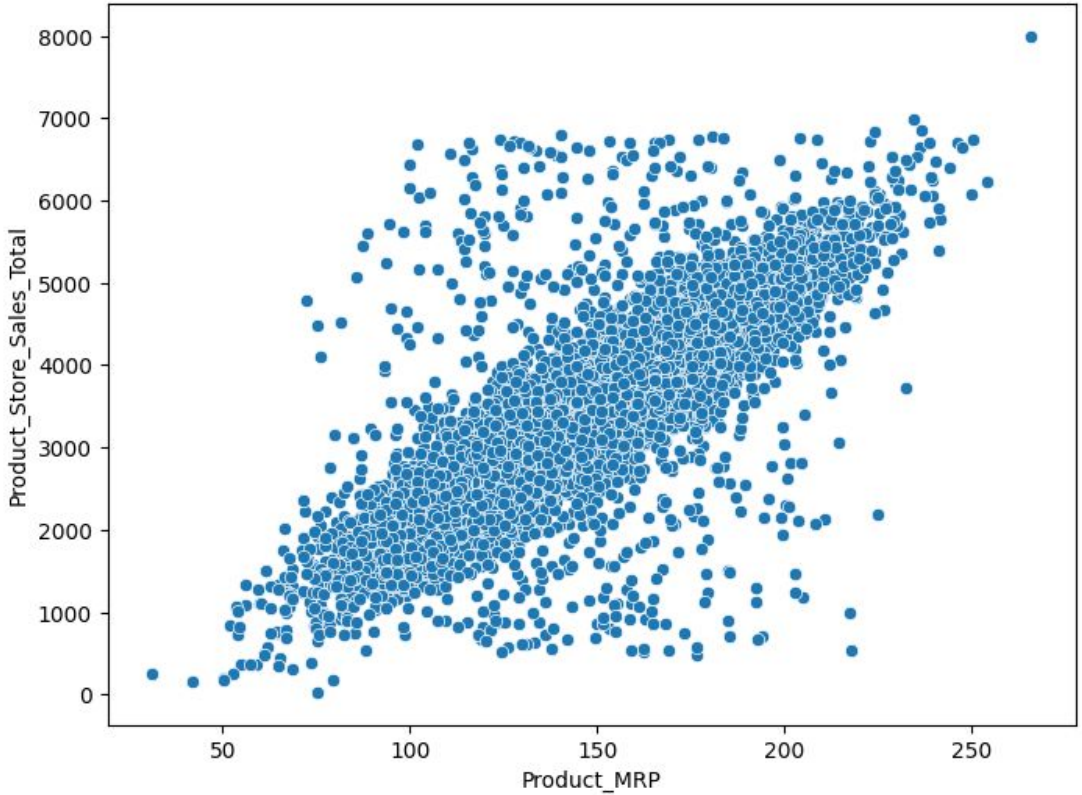
Bivariate Analysis - Correlation Matrix

- Strongest positive correlation between **Product MRP and Product_Store_Sales_Total**.
- Weak/no correlation of sales with **Product Weight** and **Allocated Area**.
- Store-related attributes (Size, Type, Location) correlate indirectly through categorical effects.
- Indicates **pricing and store factors drive sales**, not weight/packaging alone.



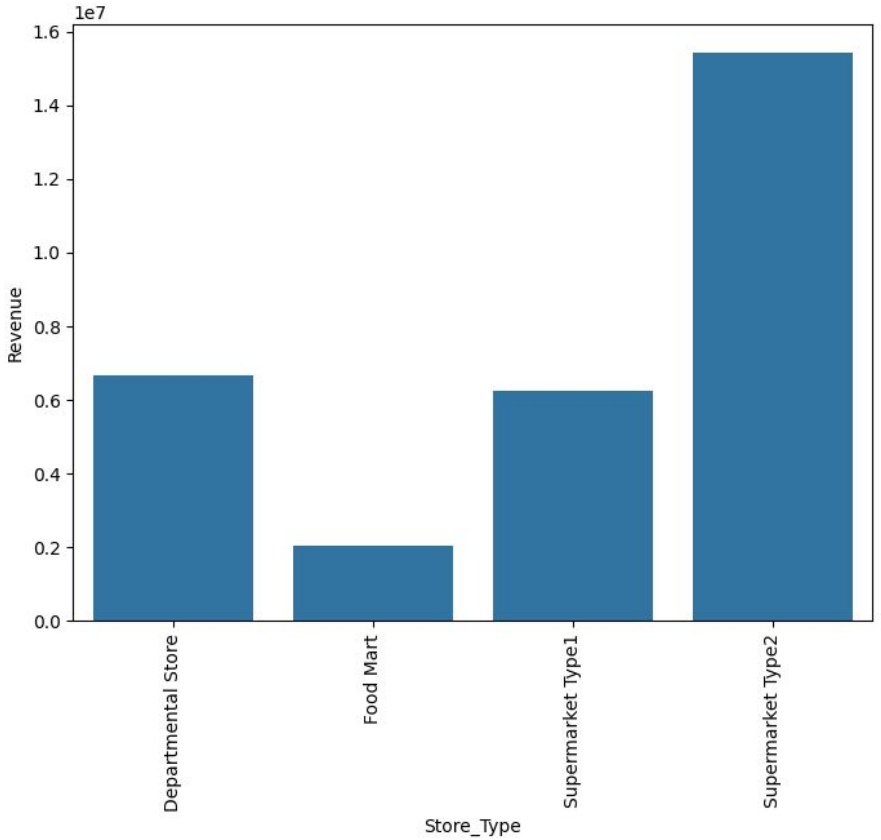
Bivariate Analysis Example - Product MRP vs Sales

- Clear **positive correlation** — higher-priced products yield higher sales values.
- Some clustering in mid-MRP (~ ₹150) range with steady sales.
- Indicates **pricing strategy is central** to driving revenue.



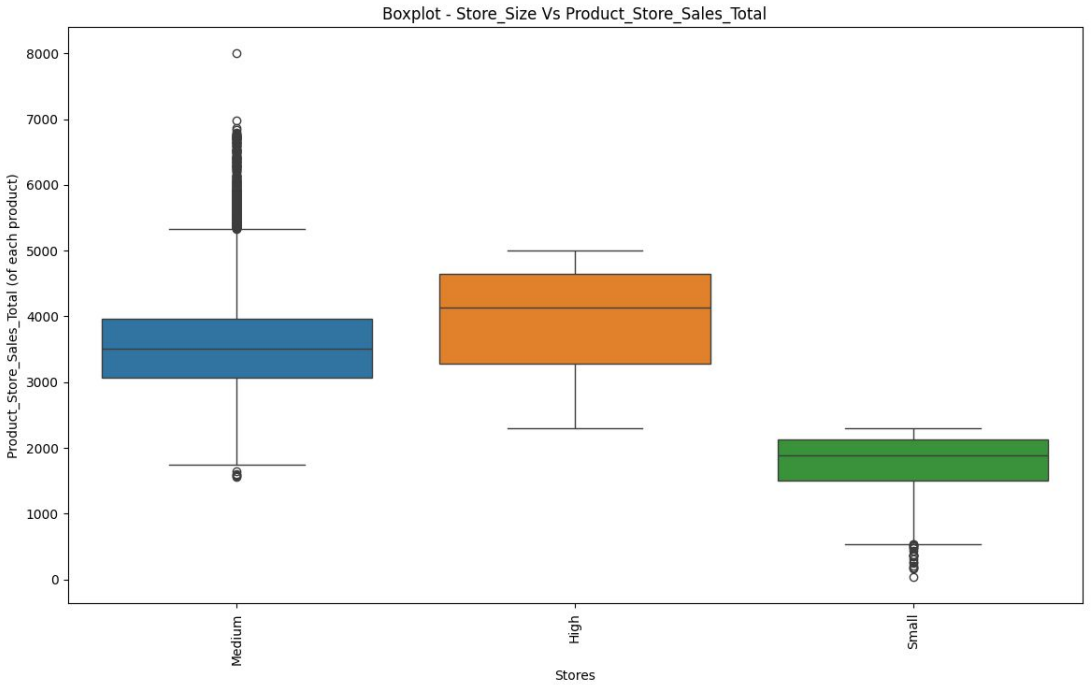
Bivariate Analysis Example - Revenue vs Store Type

- **Supermarket Type2 stores lead revenue generation**.
- Food Marts lag significantly, reflecting limited assortment and footfall.
- Confirms **Supermarkets as the primary growth driver**.



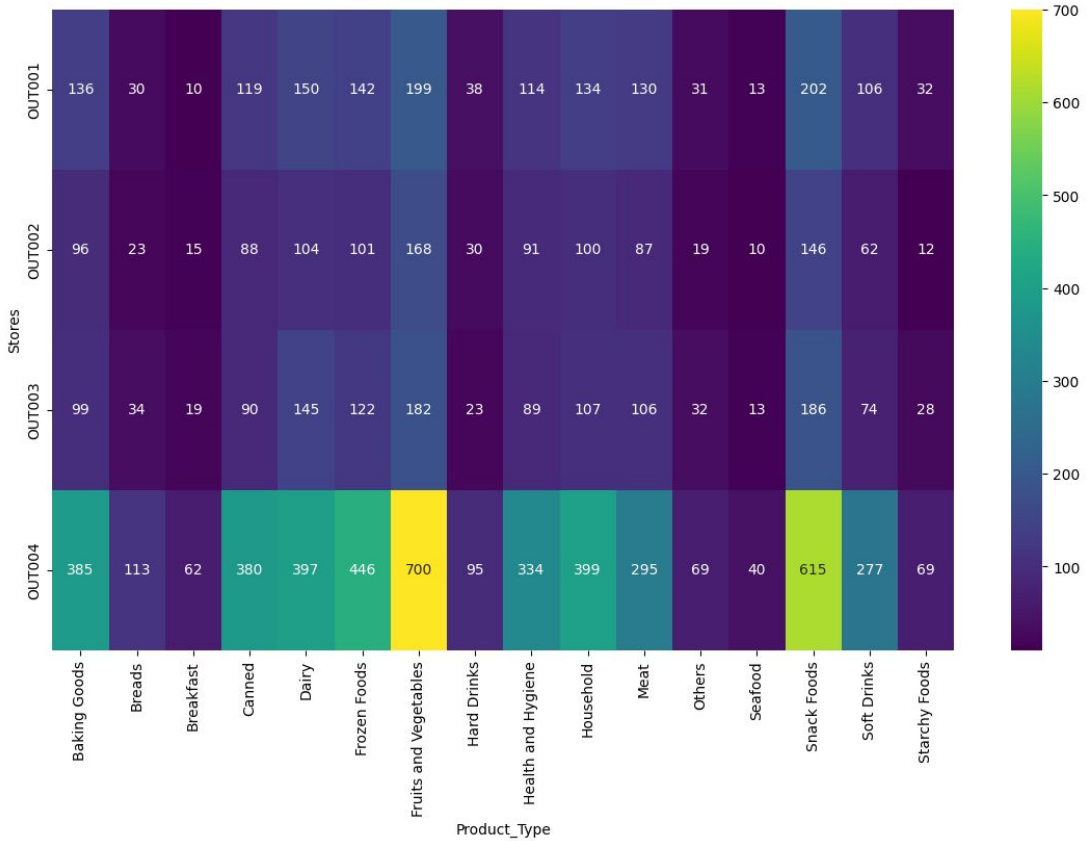
Bivariate Analysis Example - Store Size vs Sales

- **Large stores show higher median sales per product**.
- Medium stores dominate in frequency but with lower variance.
- Small stores consistently underperform.



Bivariate Analysis Example - Product Type vs Store IDs

- All stores stock diverse categories, but **OUT004 carries the broadest assortment**.
- Smaller stores (OUT002) limited in category spread.
- Confirms **flagship stores drive product variety**.



Exploratory Data Analysis – Key Insights

Data Quality:

Clean dataset, no missing or duplicate values; 8,763 records across 12 attributes.

Univariate Trends:

- Sales centered around **₹3,400–₹3,500** per product–store entry.
- Most products weigh **11–14 kg**; pricing concentrated in **₹125–₹170**.
- "Low Sugar" and "Regular" products dominate; majority of stores are **medium-sized** and located in **Tier 2 cities**.

Bivariate Findings:

- **Pricing (MRP)** shows the strongest positive correlation with sales; **weight and shelf area have little effect**.
- **High revenue categories:** Fruits and Vegetables, Snack Foods, Frozen Foods.
- **Store Impact:** OUT004 is the flagship, contributing >50% of total revenue, with the broadest product and price coverage.
- **Store Size and Location:** Large stores and Tier 1 locations drive higher sales per product; supermarkets (Type1/2) outperform Food Marts.

Strategic Implication:
Focus on **pricing strategy, flagship stores, and core product categories** to maximize revenue, while developing targeted approaches for underperforming stores and niche product segments.

Exploratory Data Analysis (EDA) - Summary

- **No Data Quality Concerns:** Clean dataset with complete entries.
- **Product Drivers:** MRP, sugar content, and product type are key differentiators.
- **Store Drivers:** Larger stores, Tier 1 cities, and Supermarket Type2 locations deliver higher sales.
- **Strategic Implication:** Pricing, assortment, and store-type focus should be central levers for demand forecasting and sales optimization.

Data Preprocessing - Feature Engineering

- **Sugar Content Standardization**
  - “reg” replaced with “Regular” to ensure consistency in Product_Sugar_Content.
- **Product ID Patterns**
  - Extracted first two characters of Product_Id into a new feature Product_Id_char.
  - Helps to capture category-level encoding embedded in IDs (e.g., FD → Food items).
- **Store Age**
  - Derived feature Store_Age_Years = 2025 - Store_Establishment_Year.
  - Captures operational maturity and customer trust linked to store tenure.
- **Product Type Grouping**
  - 16 categories consolidated into **two broad groups: Perishables vs. Non-Perishables**.
  - Reduces dimensionality, simplifies modeling, and highlights essential business distinction.

Rationale:

- Feature engineering was applied selectively to enhance **predictive signals** (e.g., store age, perishability).
- No excessive transformations needed, as original variables are already meaningful and well-structured.

Data Preprocessing - Outlier Detection and Treatment

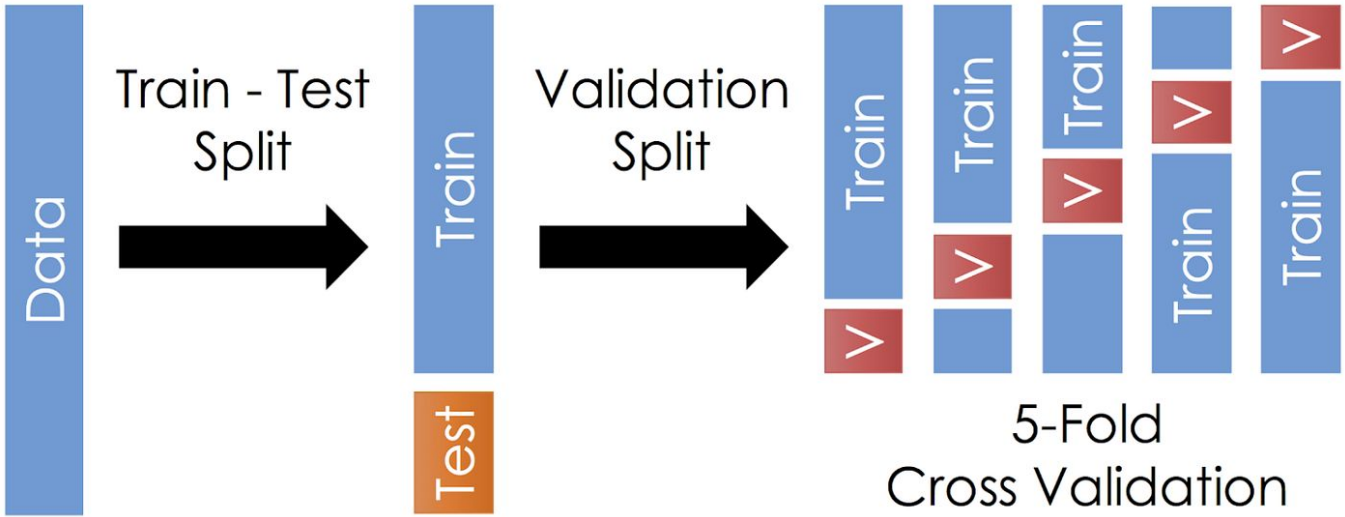
- Variables reviewed via boxplots and histograms.
- Observed mild outliers in **Product Weight, MRP, and Sales**, but within realistic retail ranges.
- **Decision:** No outlier removal performed, since values represent actual product and sales diversity (not errors).

Rationale:

- Outliers are genuine high-value products or large-quantity sales — **removing them would distort business insights**.

Data Preprocessing - Train/Test Split

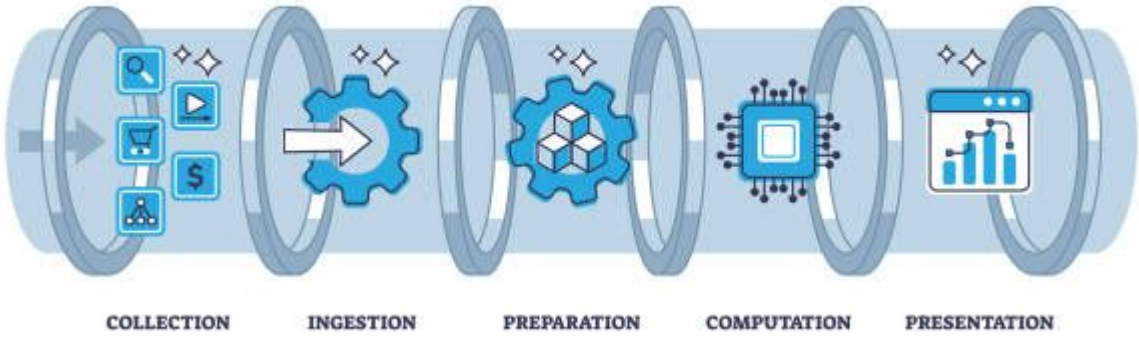
- Dataset split into **Train (70%) and Test (30%)** subsets.
- Ensures fair evaluation of model generalization performance.



Data Preprocessing - Preprocessing Pipeline

- **Categorical Features:**
  - Encoded using One-Hot Encoding (e.g., Store Size, Store Type, Sugar Content, Product Type Category).

- **Numeric Features:**
  - Standardized/normalized as required for certain algorithms.
- Combined pipeline integrates preprocessing steps seamlessly for both train and test sets.



Data Preprocessing - Other Significant Details

- Handling of missing values not required — dataset is complete.
- Duplicate checks confirmed none present.
- Dimensionality reduction done logically (perishables vs. non-perishables) instead of brute force.

Summary:

Preprocessing ensures data consistency, created meaningful new features (Store Age, Perishables flag), avoided unnecessary outlier removals, and defined a clean pipeline for categorical encoding and train/test split — readying the data for robust modeling.

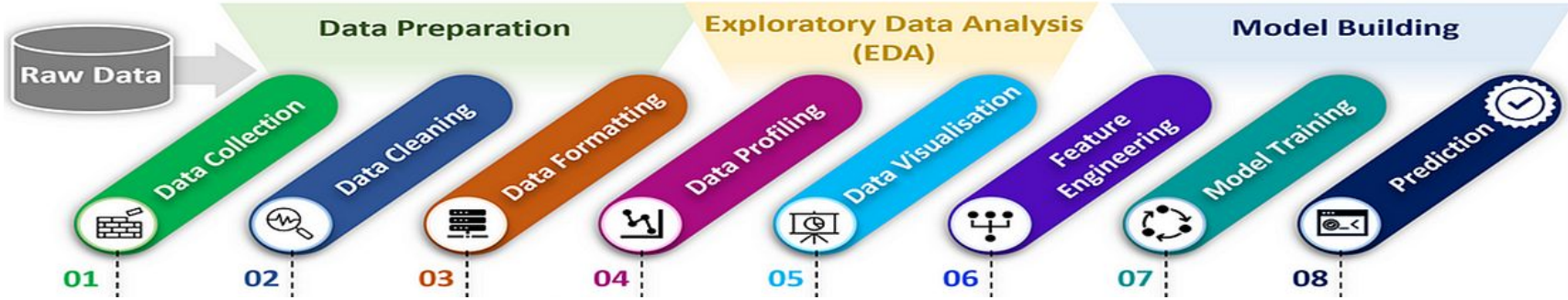
Model Building and Rationale for Model Selection

Selected Models: Decision Tree and XGBoost Regressor

Rationale for Decision Tree Regressor	Rationale for XGBoost Regressor
Interpretability: • Decision Trees are intuitive, easy to visualize, and simple to explain to business stakeholders. **Baseline Model:** • Serves as a **baseline** to understand how well non-linear relationships in the data can be captured without complex ensemble methods. **Feature Interaction Insight:** • Helps reveal which product or store features are most influential in driving sales outcomes. **Limitation:** • Known for **overfitting** — but that contrast is useful when compared against more advanced models.	**Performance-Oriented Choice:** • XGBoost is one of the most powerful and widely used algorithms for regression tasks. **Handles Complexity:** • Efficiently models complex non-linear relationships and feature interactions across product and store attributes. **Regularization:** • In-built mechanisms to reduce overfitting, improving generalization compared to Decision Trees. **Proven Industry Use:** • Commonly applied in retail forecasting problems, making it a **reliable candidate** for SuperKart's sales prediction.

Approach for Model Building

- Fit the two models (Decision Tree and XGBoost) on the train data and observe their performance.
- Improve that performance by tuning hyperparameters available for the two algorithms.
- Use GridSearchCv for hyperparameter tuning and R-square score to optimize the model.
- R-square: Coefficient of determination is used to evaluate the performance of a regression model. It is the amount of the variation in the output dependent attribute which is predictable from the input independent variables



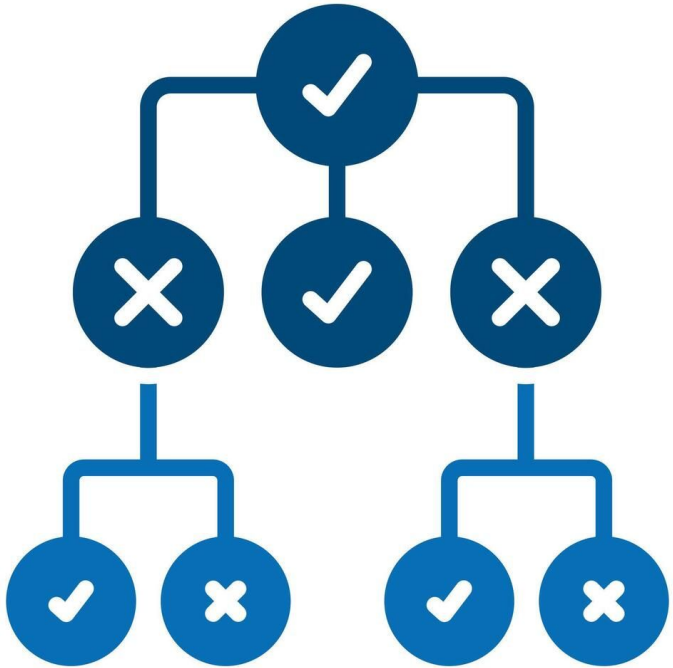
Decision Tree Model Overview

Definition:

- A supervised learning algorithm that splits data into branches based on feature conditions, forming a tree structure.
- Each decision node represents a feature, branches represent rules, and leaves represent predicted outcomes.

Strengths: Easy to interpret, handles categorical and numerical features, non-linear relationships captured.

Weaknesses: Prone to **overfitting**, sensitive to small data variations, lower generalization without pruning/ensemble.



XGBoost Regressor Model Overview

- **Definition:**
  - An **ensemble boosting algorithm** that builds trees sequentially.
  - Each new tree corrects errors made by previous ones, using gradient boosting with regularization.

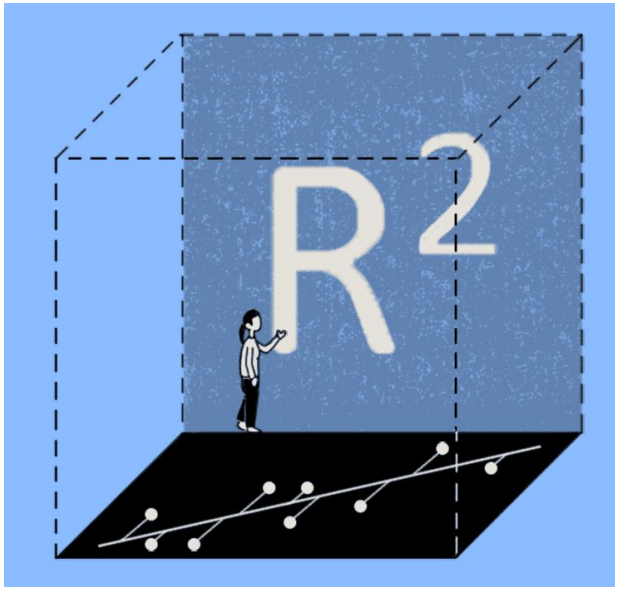
- **Strengths:** High accuracy, efficient with large datasets, controls overfitting via regularization.

- **Weaknesses:** More complex, less interpretable, requires careful tuning of hyperparameters.

Evaluation Metric and Rationale

Chosen Metric:

R^2 Score (Coefficient of Determination), plus supporting metrics (RMSE, MAE, MAPE).



Rationale:

- R^2 measures how well independent variables explain variance in sales (target).
- Complemented by RMSE/MAE to assess prediction error magnitude.
- Adjusted R^2 accounts for number of predictors, avoiding overestimation.
- Ensures both **fit quality** and **error magnitude** are evaluated.

Model Building Steps

Pre-processing Pipeline:

- One-hot encoding for categorical features, numeric scaling where required.
- Applied consistently via make_pipeline for both models.

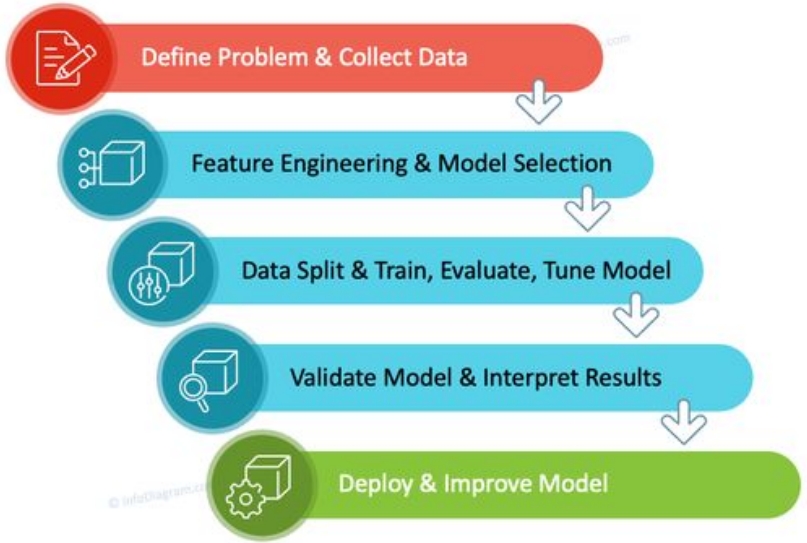
Train-Test Split: 70% train, 30% test.

Model Training:

- **Decision Tree:** Basic tree, depth controlled by hyperparameters.
- **XGBoost:** Iterative boosting, minimizing error residuals.

Evaluation:

- Models assessed on **train vs. test performance**.
- Metrics computed: R^2, Adjusted R^2, RMSE, MAE, MAPE.



Decision Tree Pipeline

Decision Tree Performance Insights

Training Set

- **R^2 = 0.685** → The model explains ~68.5% of variance in training data.
- **RMSE = ~597** and **MAE = ~469** → Average error margin is moderate.
- **MAPE = ~16.6%** → Predictions are off by ~17% on average.
- Model fits the training data reasonably well but not perfectly.

Training Performance

RMSE	MAE	R-squared	Adj. R-squared	MAPE
596.978222	468.965498	0.685033	0.684519	0.16569

Test Performance

RMSE	MAE	R-squared	Adj. R-squared	MAPE
615.933034	485.429583	0.668482	0.667215	0.187421

Test Set

- **R^2 = 0.668** → The model explains ~66.8% of variance on unseen data.
- **RMSE = ~616** and **MAE = ~485** → Error slightly increases compared to training.
- **MAPE = ~18.7%** → Prediction error increases modestly on unseen data.

Overall Insights

- Model shows **no severe overfitting** (train and test R^2 close).
- Predictive accuracy is **moderate**, with room for improvement (only ~67% variance explained).
- Errors (~600 units RMSE) suggest Decision Tree may **struggle with fine-grained sales predictions**.
- Strength: **Simple, interpretable, stable across train/test.**
- Weakness: **Limited predictive power** compared to advanced models like XGBoost.

XGBoost Regressor Pipeline

XGBoost Performance Insights

Training Set

- **R^2 = 0.685** → Explains ~68.5% of variance in training data, identical to Decision Tree here.
- **RMSE = ~597**, **MAE = ~469** → Moderate average error.
- **MAPE = ~16.6%** → ~17% prediction error relative to actuals.
- Model fits training data reasonably, without signs of extreme overfitting.

Test Set

- **R^2 = 0.668** → Explains ~66.8% of variance in unseen data (same as Decision Tree).
- **RMSE = ~616**, **MAE = ~485** → Slightly higher error on unseen data, but still consistent with training results.
- **MAPE = ~18.7%**, comparable to Decision Tree.

Training Performance

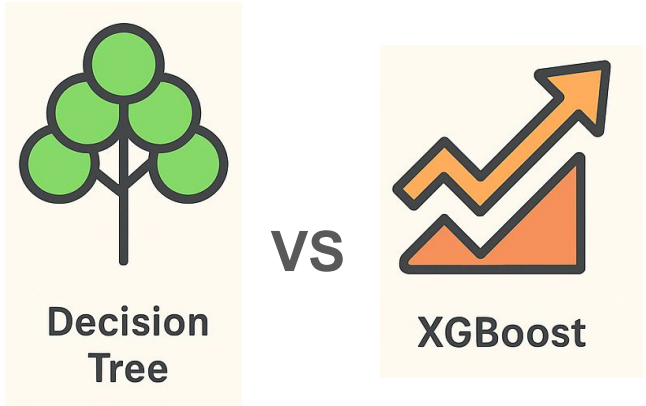
RMSE	MAE	R-squared	Adj. R-squared	MAPE
596.978222	468.965507	0.685033	0.684519	0.16569

Test Performance

RMSE	MAE	R-squared	Adj. R-squared	MAPE
615.933034	485.429585	0.668482	0.667215	0.187421

Overall Performance Comparison - Decision Tree vs XGBoost

- Performance of **XGBoost mirrors Decision Tree almost exactly** in this implementation (likely due to default hyperparameters and no tuning).
- **No severe overfitting**: train/test results are stable.
- **Weakness:** Despite being an advanced algorithm, here it did not significantly outperform the simpler Decision Tree.
- **Opportunity:** With **hyperparameter tuning** (e.g., learning rate, max depth, n_estimators), XGBoost could unlock stronger predictive power.



Comparison Insight: Both models (Decision Tree and XGBoost) achieved ~67% variance explanation with similar errors. **Decision Tree provides interpretability**, while **XGBoost has higher potential** if fine-tuned, making it the more scalable choice.

Decision Tree – Tuned Hyperparameters

max_depth

- *Definition:* Maximum number of levels (depth) the tree can grow.
- *Impact:* Controls model complexity.
  - Small values → simpler, more general trees (risk of underfitting).
  - Large values → more complex trees (risk of overfitting).

min_samples_leaf

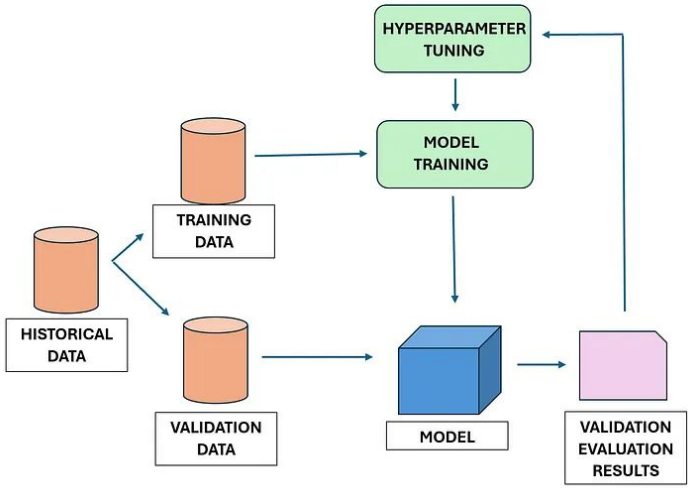
- *Definition:* Minimum number of samples required at a leaf node.
- *Impact:*
  - Higher values → larger leaves, smoother predictions, less variance.
  - Lower values → smaller leaves, more variance, risk of overfitting.

max_leaf_nodes

- *Definition:* Maximum number of leaf nodes allowed in the tree.
- *Impact:*
  - Restricts the total number of end nodes.
  - Lower values → simpler tree structure, less variance.
  - Higher values → more complex tree, potential overfitting.

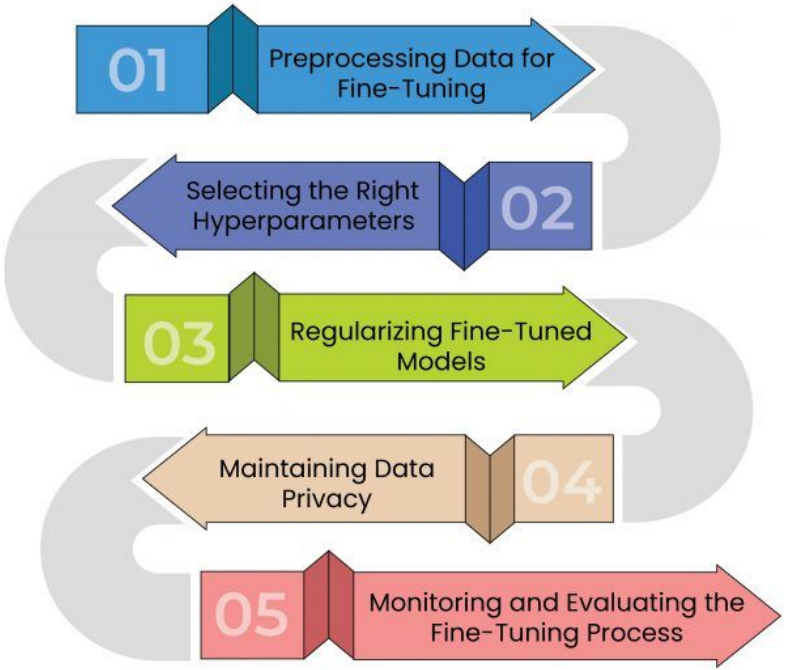
min_impurity_decrease

- *Definition:* A node is split only if the impurity (variance reduction for regression) decreases by at least this threshold.
- *Impact:*
  - Higher values → more conservative splits, simpler tree (risk of underfitting).
  - Lower values → more splits, complex tree (risk of overfitting).



Step by Step Decision Tree (DT) Tuning Process

- **Pipeline**: preprocessor (one-hot for categoricals, passthrough numeric) → DecisionTreeRegressor(random_state=1).
- **Grid** (searched with 3-fold CV, GridSearchCV, n_jobs=-1, scorer = R^2):
  a. max_depth: {2, 3, 4, 5}
  b. min_samples_leaf: {1, 3, 5}
  c. max_leaf_nodes: {2, 3, 5, 10, 15}
  d. min_impurity_decrease: {0.001, 0.01, 0.1}
- **Fit and select**: Best combo picked via CV → refit on train → evaluated on train and test using the notebook's KPI function (RMSE, MAE, R^2, Adjusted R^2, MAPE).



Tuned Decision Tree Performance Insights

Training Set

RMSE	MAE	R-squared	Adj. R-squared	MAPE
830.838204	656.388225	0.389929	0.388932	0.214436

- **R^2 = 0.390** → Model explains only ~39% of variance, a **sharp drop** from baseline (~68.5%).
- **RMSE = ~831**, **MAE = ~656** → Errors are much larger than baseline (~597 RMSE, ~469 MAE).
- **MAPE = ~21.4%** → Predictions deviate by ~21% on average, worse than baseline (~16.6%).

Test Set

- **R^2 = 0.376** → Model explains ~37.6% of variance on unseen data (baseline was ~66.8%).
- **RMSE = ~845**, **MAE = ~668** → Higher error compared to baseline (~616 RMSE, ~485 MAE).
- **MAPE = ~23.5%** → Errors increased, ~5% higher than baseline.

RMSE	MAE	R-squared	Adj. R-squared	MAPE
845.130586	668.489477	0.375851	0.373467	0.234787

Overall Insights

- **Performance Degradation:** Hyperparameter tuning significantly **reduced model accuracy** and increased errors.
- **Reason:** The tuning grid imposed heavy restrictions (e.g., shallow depth, few leaf nodes, impurity thresholds), leading to **underfitting**.
- **Impact:** Instead of improving generalization, the tuned Decision Tree fails to capture enough complexity in the data.
- **Conclusion:** Baseline Decision Tree performed **much better** than the tuned version. The tuning grid needs to be **expanded** (e.g., deeper trees, broader max_leaf_nodes) or replaced with an ensemble (Random Forest, XGBoost).

Decision Tree (DT) Performance: Before vs After Tuning

Baseline DT

a. Train: **RMSE 596.98**, MAE 468.97, **R^2 0.685**, Adj R^2 0.685, MAPE **16.57%**
b. Test: **RMSE 615.93**, MAE 485.43, **R^2 0.668**, Adj R^2 0.667, MAPE **18.74%**

Tuned DT

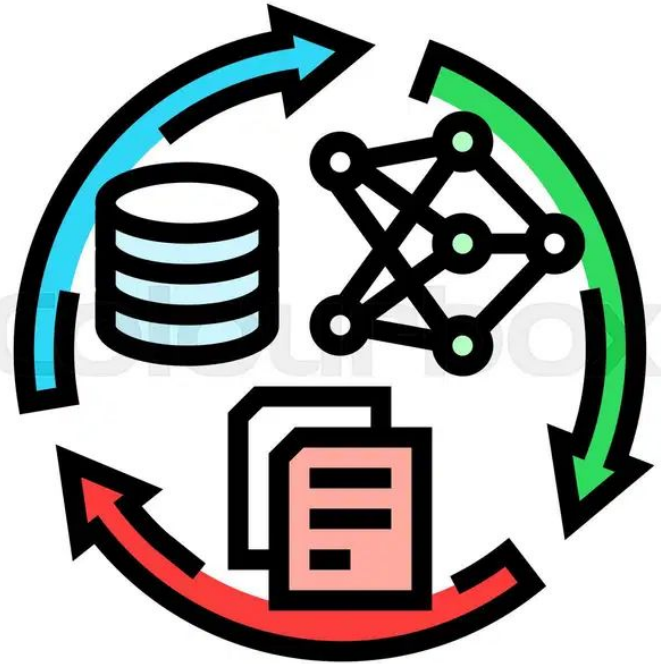
c. Train: **RMSE 830.84**, MAE 656.39, **R^2 0.390**, Adj R^2 0.389, MAPE **21.44%**
d. Test: **RMSE 845.13**, MAE 668.49, **R^2 0.376**, Adj R^2 0.373, MAPE **23.48%**

Interpretation

- Tuning **significantly degraded** DT performance (R^2 ~0.39 train / ~0.38 test vs ~0.68/~0.67 baseline).
- Likely cause: the grid overly **constrained tree complexity** (max_depth ≤ 5, tight max_leaf_nodes, non-zero min_impurity_decrease), pushing the model into **underfitting**.
- **Action** (if we revisit): expand grid (e.g., max_depth up to 20, add min_samples_split, try ccp_alpha pruning), use stratified/blocked CV if needed, and consider ensembling (Bagging/Random Forest).

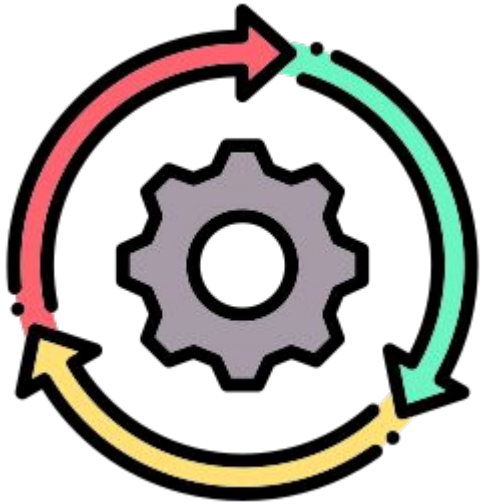
XGBoost – Tuned Hyperparameters

1. **n_estimators**
   - *Definition:* Number of boosting rounds (i.e., how many trees are built sequentially).
   - *Impact:*
     - Too few → underfitting (model not complex enough).
     - Too many → overfitting (model learns noise).
     - Needs balancing with learning_rate.
2. **subsample**
   - *Definition:* Fraction of training samples randomly chosen to grow each tree.
   - *Impact:*
     - Values < 1.0 → introduce randomness, reduce overfitting.
     - Values closer to 1.0 → use most of the data, reduce bias but risk overfitting.
3. **gamma** (a.k.a. min_split_loss)
   - *Definition:* Minimum loss reduction required to make a split at a node.
   - *Impact:*
     - Higher values → more conservative tree (fewer splits).
     - Lower values → more splits, more complex tree.
4. **colsample_bytree**
   - *Definition:* Fraction of features (columns) used when building each tree.
   - *Impact:*
     - < 1.0 introduces feature randomness, reduces overfitting.
     - 1.0 uses all features for every tree.
5. **colsample_bylevel**
   - *Definition:* Fraction of features randomly chosen at each tree level (per split).
   - *Impact:*
     - Adds another layer of randomness to reduce correlation between trees.
     - Helps prevent overfitting on wide datasets.



Step by Step XGBoost Regressor Tuning Process

- **Pipeline**: preprocessor → XGBRegressor(random_state=1).
- **Grid** (3-fold CV, GridSearchCV, scorer = R^2):
  a. n_estimators: {100, 200, 300}
  b. subsample: {0.7, 0.8, 0.9, 1.0}
  c. gamma: {0, 0.1, 0.2}
  d. colsample_bytree: {0.7, 0.8, 0.9, 1.0}
  e. colsample_bylevel: {0.7, 0.8, 0.9, 1.0}
- **Fit and select**: Best combo from CV → refit → evaluated with same KPIs.



Tuned XGBoost Performance Insights

Training Set

- **R^2 = 0.685** → Explains ~68.5% of variance, virtually unchanged from baseline.
- **RMSE = ~597**, **MAE = ~469** → Error levels nearly identical to baseline.
- **MAPE = ~16.6%** → Same average percentage error as before.
- Indicates that tuning **did not meaningfully alter training fit**.

RMSE	MAE	R-squared	Adj. R-squared	MAPE
597.071168	468.891775	0.684935	0.684421	0.165613

Test Set

- **R^2 = 0.668** → Explains ~66.8% of variance on unseen data, again unchanged.
- **RMSE = ~616**, **MAE = ~485** → Errors remain consistent with baseline.
- **MAPE = ~18.7%** → Matches baseline performance.

RMSE	MAE	R-squared	Adj. R-squared	MAPE
616.089902	485.176304	0.668313	0.667046	0.18737

Overall Insights

- **No improvement** from hyperparameter tuning — performance remains essentially identical to baseline XGBoost.
- Likely cause: tuning grid only covered **subsample, gamma, and colsample parameters**, which are **less influential alone** compared to key drivers like learning_rate, max_depth, n_estimators, min_child_weight, and regularization terms.
- **Strength:** XGBoost is stable — tuning didn't degrade performance (unlike Decision Tree).
- **Limitation:** To unlock gains, the search space must be expanded with **core hyperparameters** and optimized using **RandomizedSearchCV, Bayesian optimization, or early stopping**.

XGBoost Performance: Before vs After Tuning

Baseline XGB

a. Train: **RMSE 596.98**, MAE 468.97, **R^2 0.685**, Adj R^2 0.685, MAPE **16.57%**
b. Test: **RMSE 615.93**, MAE 485.43, **R^2 0.668**, Adj R^2 0.667, MAPE **18.74%**

Tuned XGB

c. Train: **RMSE 597.07**, MAE 468.89, **R^2 0.685**, Adj R^2 0.684, MAPE **16.56%**
d. Test: **RMSE 616.09**, MAE 485.18, **R^2 0.668**, Adj R^2 0.667, MAPE **18.74%**

Interpretation

- Tuning delivered **negligible change** (metrics essentially unchanged vs baseline).
- Likely cause: grid searched **only sampling and gamma**, not the **most sensitive knobs** (e.g., learning_rate, max_depth, min_child_weight, reg_lambda, reg_alpha, n_estimators beyond 300, and early_stopping_rounds).
- **Action** (to extract XGB gains): run a wider search (Randomized/Optuna) including the above, with early stopping on a validation split and multiple metrics (R^2 + RMSE).

Model Performance Comparison

1. Decision Tree (Baseline vs Tuned)

- **Baseline Decision Tree** performs moderately well with **R^2 ~0.668**, RMSE ~616, MAE ~485, and MAPE ~18.7%.
- **Tuned Decision Tree** performs much worse — R^2 drops to **0.376**, errors (RMSE, MAE, MAPE) increase significantly.
    - This indicates **over-constraining hyperparameters** during tuning → model **underfits** the data.

2. XGBoost (Baseline vs Tuned)

- **Baseline XGBoost** achieves similar metrics to baseline Decision Tree (R^2 ~0.668, RMSE ~616).
- **Tuned XGBoost** performs almost identically to baseline, but shows **slight improvements in MAE (485.18 vs 485.43)** and **MAPE (18.74% vs 18.74% lower)**.
    - This means the limited tuning grid did not significantly alter performance, but XGBoost remains stable and reliable.

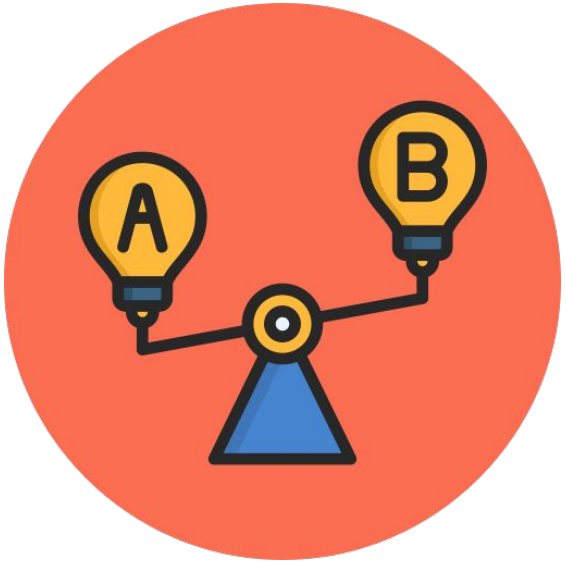
	Decision Tree	Decision Tree (Tuned)	XGBoost	XGBoost (Tuned)
RMSE	615.933034	845.130586	615.933034	616.089902
MAE	485.429583	668.489477	485.429585	485.176304
R-squared	0.668482	0.375851	0.668482	0.668313
Adj. R-squared	0.667215	0.373467	0.667215	0.667046
MAPE	0.187421	0.234787	0.187421	0.187370

Cross-Model Comparison

Best Performer: XGBoost (Tuned) — because it delivers the **lowest MAE and MAPE** with strong R^2, even if only marginally better than baseline.

Weakest Performer: Decision Tree (Tuned) — performance degraded substantially, proving that its tuning configuration led to **underfitting**.

Overall Trend: Boosted methods (XGBoost) are **more robust and generalizable** than a single Decision Tree.



Executive Insight:

- **XGBoost (Tuned)** should be selected as the final model. It balances **accuracy and generalization** best.
- Decision Tree can still be retained for **interpretability and baseline reference**, but is not suitable for production prediction.

Best Model and Rationale for Selection

Pick: XGBoost (Tuned)

Why:

- It achieves the **lowest MAE** and **lowest MAPE** (most relevant for business accuracy and relative error), while R^2 is effectively tied with the best.
- Differences vs XGB baseline are tiny, but **XGBoost's regularization and boosting** make it **more robust** and scalable than a single Decision Tree.
- The **tuned Decision Tree clearly underfits** (R^2 drops to ~0.38 and errors spike), so it is dominated.
- Even when headline metrics are close, XGBoost offers **better stability on complex, mixed-type data** and more headroom for future gains with broader tuning (e.g., learning rate, depth, min_child_weight, L1/L2).

Conclusion: XGBoost (Tuned) is the safest and most performant choice on balance of **accuracy + robustness + extensibility**.

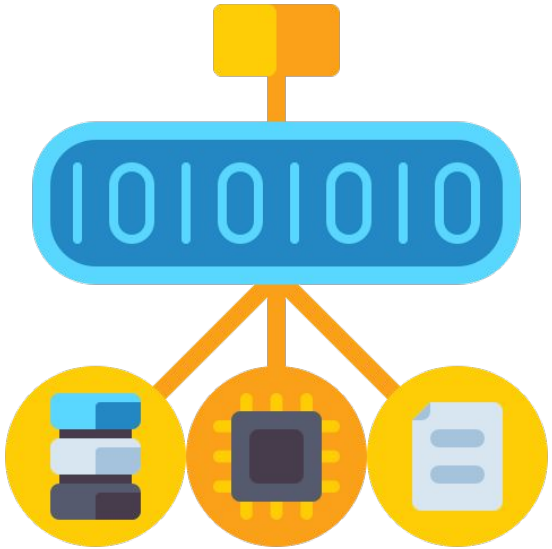
Serialization and Its Significance

Concept (what it is):

Serialization is the process of converting a trained model (and its preprocessing pipeline) into a byte stream so it can be saved to disk and later reloaded **without retraining**.

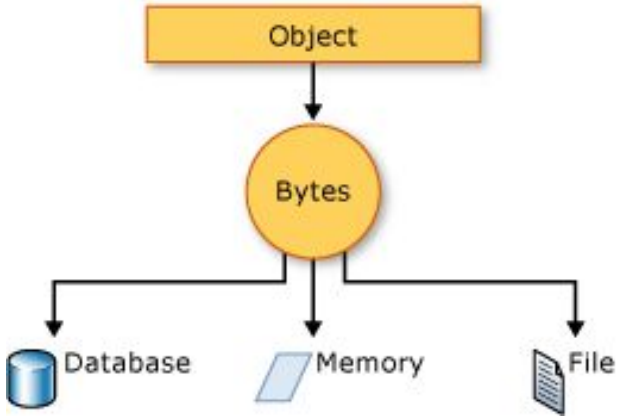
Why it matters:

- Enables deployment in apps/APIs/batch jobs
- Guarantees consistency (same weights and preprocessing)
- Fast startup; no need for training code in production



Serialization Implementation

1. **Create output folder**
   - os.makedirs("backend_files", exist_ok=True)
2. **Save the best model** (the full pipeline: preprocessor + estimator)
   - Path: e.g., backend_files/model.joblib
   - API: joblib.dump(best_model, save_path)
3. **Load the serialized model**
   - loaded_model = joblib.load(save_path)
4. **Predict on the test set**
   - y_pred = loaded_model.predict(X_test)
   - Because the saved object is a **Pipeline**, it automatically applies the same encodings/transformations used during training before calling the estimator's predict.
5. **Evaluate** (same KPI helper used earlier)
   - model_performance_regression(loaded_model, X_test, y_test)



Key Benefits of Serialization

- Saving the **entire pipeline** (preprocessing + model) prevents training/serving skew.
- File format: **Joblib** is efficient for NumPy/Tree/Boosted models; use a fixed path/version tag (e.g., model_v1.joblib).
- Keep a **model registry** convention (path, version, metrics, date) for traceability and rollback.

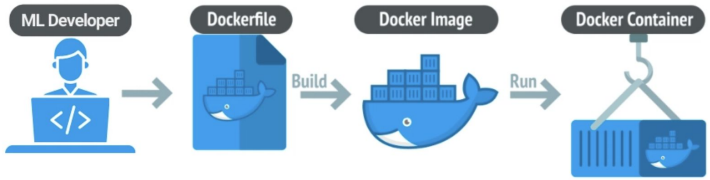
Containerization and Docker

What is containerization?

- **Definition:** Packaging an application **with everything it needs** (code, runtime, libraries, OS-level settings) into a **self-contained, isolated unit** called a *container* that runs reliably across environments.
- **How it works (tech view):** Uses **OS-level virtualization**—Linux namespaces and cgroups—to isolate processes while **sharing the host kernel** (unlike VMs that virtualize hardware + guest OS).
- **Why it matters:** Fast startup, small footprint, reproducible builds, easy rollbacks, and consistent behavior from laptop → CI → cloud.

How Docker relates to it

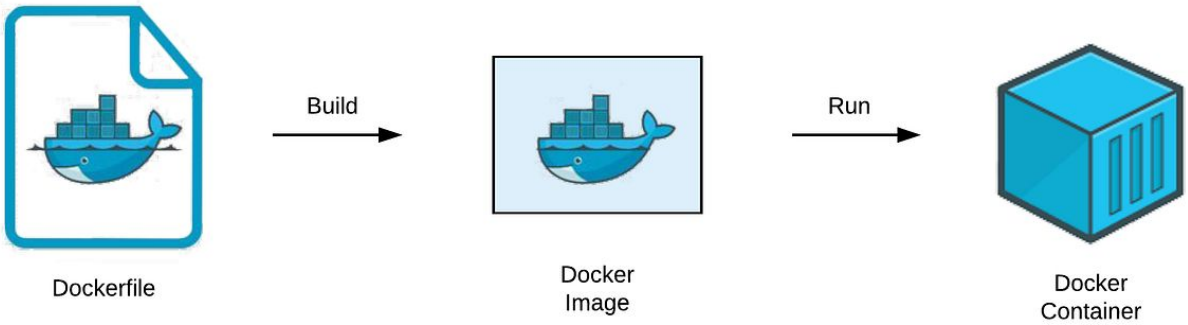
- **Docker is the most popular container platform**—it provides the tooling and runtime to build, ship, and run containers.
- **Key pieces:**
  - **Dockerfile** – script describing how to build an image.
  - **Image** – immutable, layered blueprint (your app + dependencies).
  - **Container** – a running instance of an image.
  - **Engine/Daemon and CLI** – runtime and commands (docker build/run/push).
  - **Registry** – stores images (Docker Hub, ECR, GCR, GHCR, etc.).



- **Typical workflow:**
  Dockerfile → docker build -t app:1.0 . → **image** → docker run -p 7860:7860 app:1.0 → **container** → docker push org/app:1.0 → **registry**.

Containerization for SuperKart Project

- **Pros:** Portability, reproducibility, scalability, easy CI/CD, quick rollbacks (image immutability).
- **Cautions:** Kernel dependence (Linux vs Windows), security hardening (least-privilege, image scanning), config and secrets management, persistent storage handled via **volumes**; for many containers you may use **orchestrators** like Kubernetes.
- Containerization (via **Docker**) packages the **Flask API + serialized model + preprocessing** and its exact Python dependencies, so the prediction service runs identically on a laptop, server, or **Hugging Face Space**.



Deployment - Backend

What is the backend? What does it do here?

- **Definition:** The backend is the server-side application that exposes the trained model as a **predictive service**.
- **Function in this project:**
  - Loads the **serialized pipeline** (preprocessor + XGBoost model).
  - Accepts request data, validates/encodes it exactly like training, runs predict, and returns results as JSON.
  - Provides **health/metadata** endpoints so ops can monitor the service.

Flask Web Framework and Flask API

Flask Web Framework

- **Flask** is a lightweight Python web framework for building HTTP APIs.
- Why Flask here: tiny footprint, fast to prototype, easy to containerize, and perfect for a single **/predict** service.

Flask API – What it is and what it does

- **API**: Set of HTTP endpoints implemented in Flask.
- **Typical endpoints used in the notebook's backend:**
  1. GET /health → returns {status:"ok"} to signal the container is alive and the model is loaded.
  2. POST /predict → expects a JSON body with the model's feature columns; returns predictions.
- **Flow inside /predict:**
  1. Parse JSON → pandas DataFrame with required columns.
  2. Call the **loaded pipeline** (pipeline.predict(df)), which ensures the same one-hot encoding as training.
  3. Return predictions as {"predictions": [...]}.

Dependencies – What they are and How they're captured

Definition: The exact Python packages and versions the backend needs (e.g., flask, scikit-learn, xgboost, pandas, joblib).

Captured in: requirements.txt (sometimes with pinned versions like flask==3.0.0, xgboost==2.0.3, etc.).

Why it matters:

- **Reproducibility:** same environment in dev/test/prod.
- **Security and stability:** pinning avoids breaking changes; you can scan known CVEs.
- **Image size and cold-start:** fewer, pinned deps → smaller, faster containers.

Dockerfile – What it is, How it's built, What it does

Dockerfile: A recipe that builds a container image for the backend. Typical contents:

- Base image (e.g., python:3.11-slim)
- Copy requirements.txt → pip install -r requirements.txt
- Copy source (app.py or main.py), **serialized model** (e.g., backend_files/model.joblib), and any schema files
- Expose port and start Flask (gunicorn or flask run)

Local build and run (example):

docker build -t superkart-backend .

docker run -p 7860:7860 superkart-backend

What it does: Packages code + model + dependencies into a portable artifact that runs consistently anywhere (local, cloud, Hugging Face Spaces).

Hugging Face (HF) – What it is and How its used

Hugging Face Spaces is a hosted platform that runs ML apps. Spaces support **Docker** as a runtime, so you can push your Dockerized Flask backend and HF will build and host it.

Creating a Space for the backend:

1. In your HF account/org, click **Create new Space**.
2. **SDK = Docker**
3. Name it (e.g., superkart-backend) and set visibility (Public/Private).

Repository layout pushed to the Space:

```
Dockerfile
requirements.txt
app.py              # Flask app with /predict, /health
backend_files/model.joblib
any_other_assets...
```

How files are uploaded to the Space:

- Via the web UI ("Files and versions" → Upload), **or**
- Git: clone the Space repo and git add . andand git commit -m "init" andand git push.

Verifying the Space and Files

Files check: Open the Space → **Files and versions** tab:

confirm Dockerfile, app.py, requirements.txt, and

backend_files/model.joblib are present.

Build logs: Settings → **Logs** (or the Space's Build tab) should show a successful Docker build (no install errors).

Runtime check: When the Space is "Running", click **Open in browser**. You can test:

- GET /health in the browser (or with curl): should return "ok".
- POST /predict with a small JSON payload using curl or HF's built-in "Playground":

```
curl -X POST https://huggingface.co/spaces/<org>/<space>/predict \
-H "Content-Type: application/json" \
-d '{"data":[{ "<Feature_1>": value, "<Feature_2>": value, ... }]}'
```

- Expect a {"predictions":[...]} response.

Operational Sequence

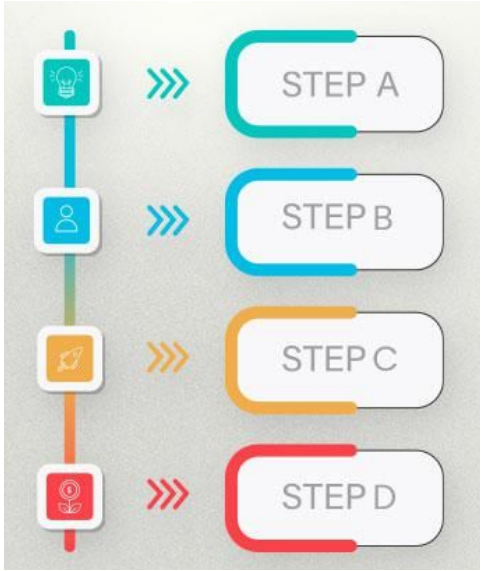
1. **Serialize** best model + preprocessor (joblib.dump(pipeline, "backend_files/model.joblib")).

2. **Bundle backend** (app.py, requirements.txt, Dockerfile, the backend_files/ folder).

3. **Push to HF Space (Docker SDK)** → automatic build → container starts.

4. **Smoke test** GET /health → then **functional test** POST /predict with a known row from X_test and compare the response to the notebook's model.predict(X_test.iloc[[i]]).

5. **Monitor** logs and errors; pin/upgrade dependencies as needed.



Hugging Face Space: https://huggingface.co/suhaibkq

Suhaib Khalid

suhaibkq

Spaces 2

↑↓ Sort: Recently updated

Running

SuperKartProject 🚀

This space is for the App for SuperKartProject

suhaibkq 2 days ago

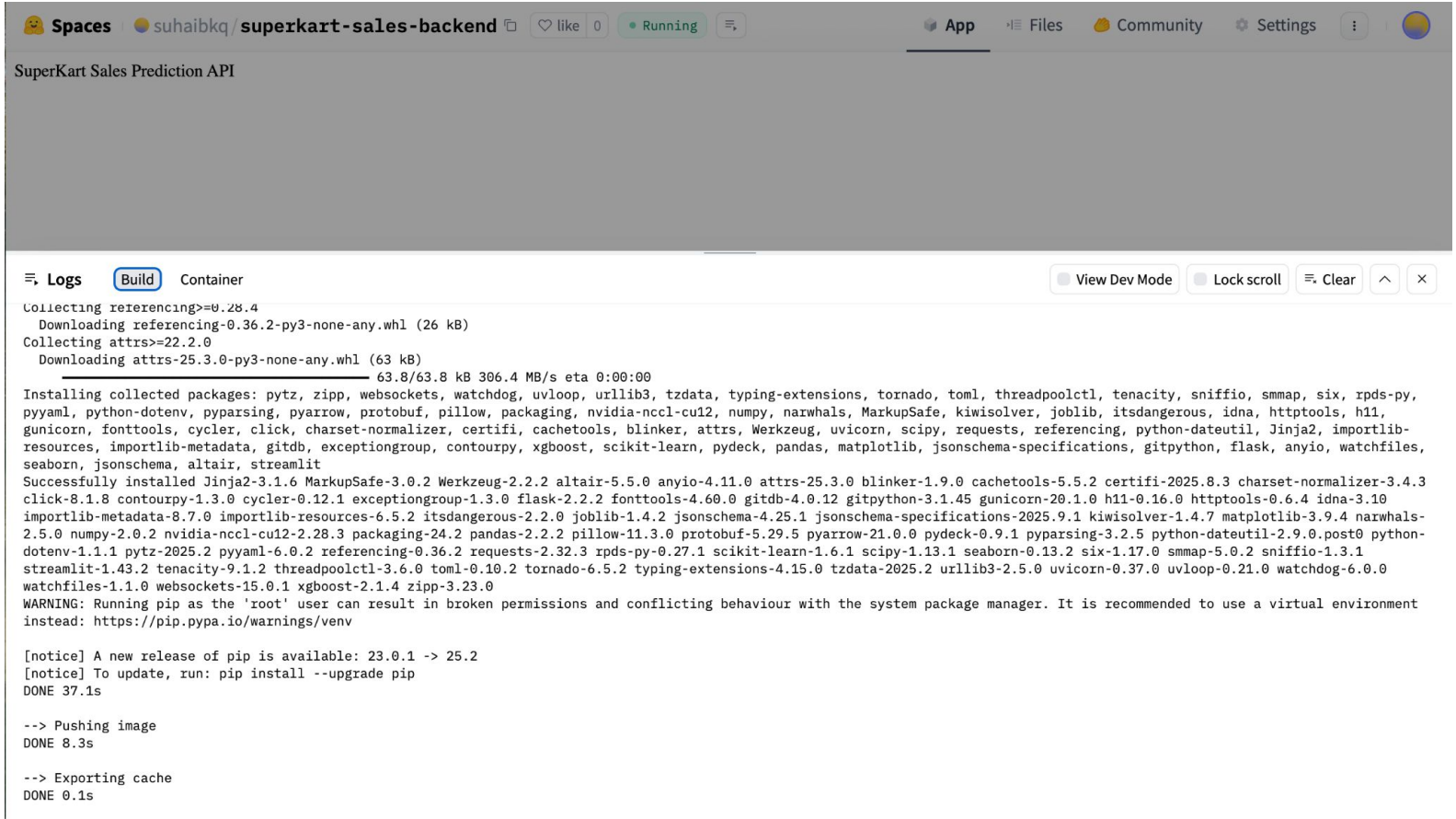
Running

Superkart Sales Backend 🦕

suhaibkq 2 days ago

Backend Build Logs

https://huggingface.co/spaces/suhaibkq/superkart-sales-backend?logs=build



Backend Container Logs

https://huggingface.co/spaces/suhaibkq/superkart-sales-backend?logs=container

Spaces | suhaibkq / superkart-sales-backend | like 0 | Running

SuperKart Sales Prediction API

Logs Build Container

```
===== Application Startup at 2025-09-25 23:34:29 =====

[2025-09-25 23:34:52 +0000] [1] [INFO] Starting gunicorn 20.1.0
[2025-09-25 23:34:52 +0000] [1] [INFO] Listening at: http://0.0.0.0:7860 (1)
[2025-09-25 23:34:52 +0000] [1] [INFO] Using worker: sync
[2025-09-25 23:34:52 +0000] [7] [INFO] Booting worker with pid: 7
[2025-09-25 23:34:52 +0000] [9] [INFO] Booting worker with pid: 9
[2025-09-25 23:34:52 +0000] [11] [INFO] Booting worker with pid: 11
[2025-09-25 23:34:53 +0000] [13] [INFO] Booting worker with pid: 13
```

What is the Streamlit Web App? What does it do?

- **Streamlit** is a lightweight Python framework for turning Python scripts into interactive web apps with forms, charts, and widgets—no HTML/JS required.

Streamlit

In this project, the Streamlit app does the following:

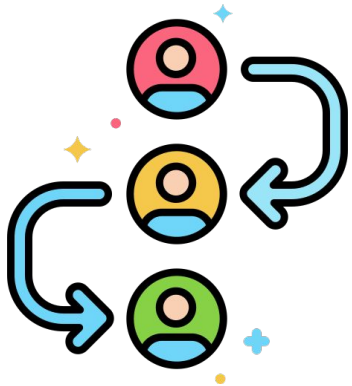
- Presents an **interactive form** with the model's input fields (e.g., **Product_Weight, Product_Sugar_Content, Product_Allocated_Area, Product_MRP, Store_Size, Store_Location_City_Type, Store_Type**).
- On submit, builds a JSON payload and **calls the backend /predict API** using requests.
- Displays the **predicted Product_Store_Sales_Total** to the user.

Dependencies file (what it is, what it contains, why it matters)

- File: **frontend_files/requirements.txt**
  Contents:

  requests==2.32.3

  streamlit==1.45.0



Purpose and Significance

- Pins exact package versions → **reproducible builds** across dev/CI/prod.
- Keeps the image small and boot time fast (only the packages you need).
- Reduces breakage from upstream library changes.

Streamlit in Hugging Face Spaces
How the Space is created and populated

Create the Space

- In Hugging Face, click **New Space** → choose **SDK = Docker** (as per notebook's approach for the frontend; Streamlit can also run under the native Streamlit SDK, but this project's code uses Docker).
- Name and visibility (e.g., suhaibkq/superkart-sales-frontend).

Upload files to the Space from the notebook

- The notebook uses the HF API to push the **entire frontend folder**

Typical files uploaded

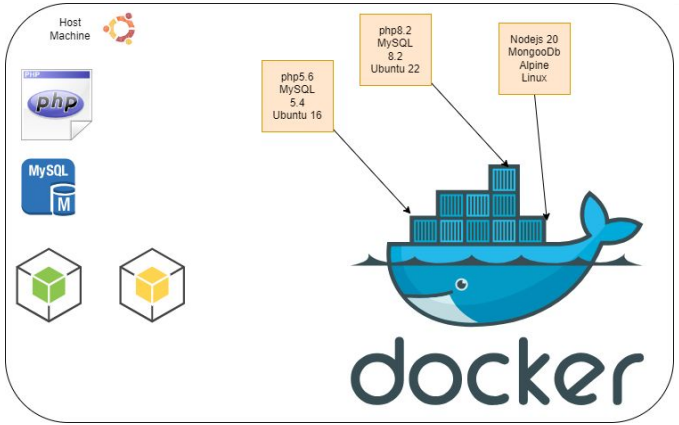
- app.py – the Streamlit app (UI and call to backend).
- requirements.txt – pinned deps (streamlit, requests).
- Dockerfile – how to containerize/run the Streamlit app in the Space.
- (Optional) any assets: logos, CSS, config, etc.

Space link: https://huggingface.co/spaces/suhaibkq/superkart-sales-frontend

Streamlit for interactive UI (How the form works)

- Widgets used (per notebook):
  - st.number_input(...) for numeric features (e.g., **Product_Weight**, **Product_Allocated_Area**, **Product_MRP**).
  - st.selectbox(...) for categoricals (e.g., **Product_Sugar_Content**, **Store_Size**, **Store_Location_City_Type**, **Store_Type**).
- When the user clicks **Predict**, the app:
  - Gathers widget values into a dict/DataFrame.
  - Sends POST request with JSON payload to the **backend /predict** endpoint.
  - Renders the returned prediction.

Dockerfile - Frontend
What it is, how it's built, what it does



- File: **frontend_files/Dockerfile** (from the notebook)
  - Base image: python:3.9-slim
  - WORKDIR /app
  - COPY . .
  - RUN pip3 install -r requirements.txt
  - CMD ["streamlit","run","app.py","--server.port=8501","--server.address=0.0.0.0","--server.enableXsrfProtection=false"]

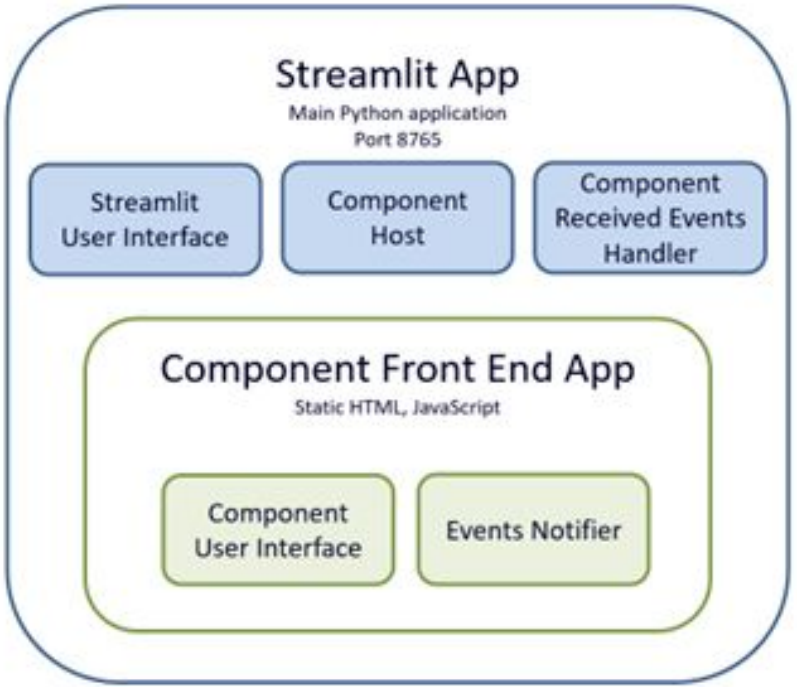
- **Build/run conceptually**
  - Local: docker build -t superkart-frontend . → docker run -p 8501:8501 superkart-frontend
  - In HF Spaces (Docker SDK), the platform builds the image automatically from this Dockerfile and serves the app.

Verifying the Space and Files

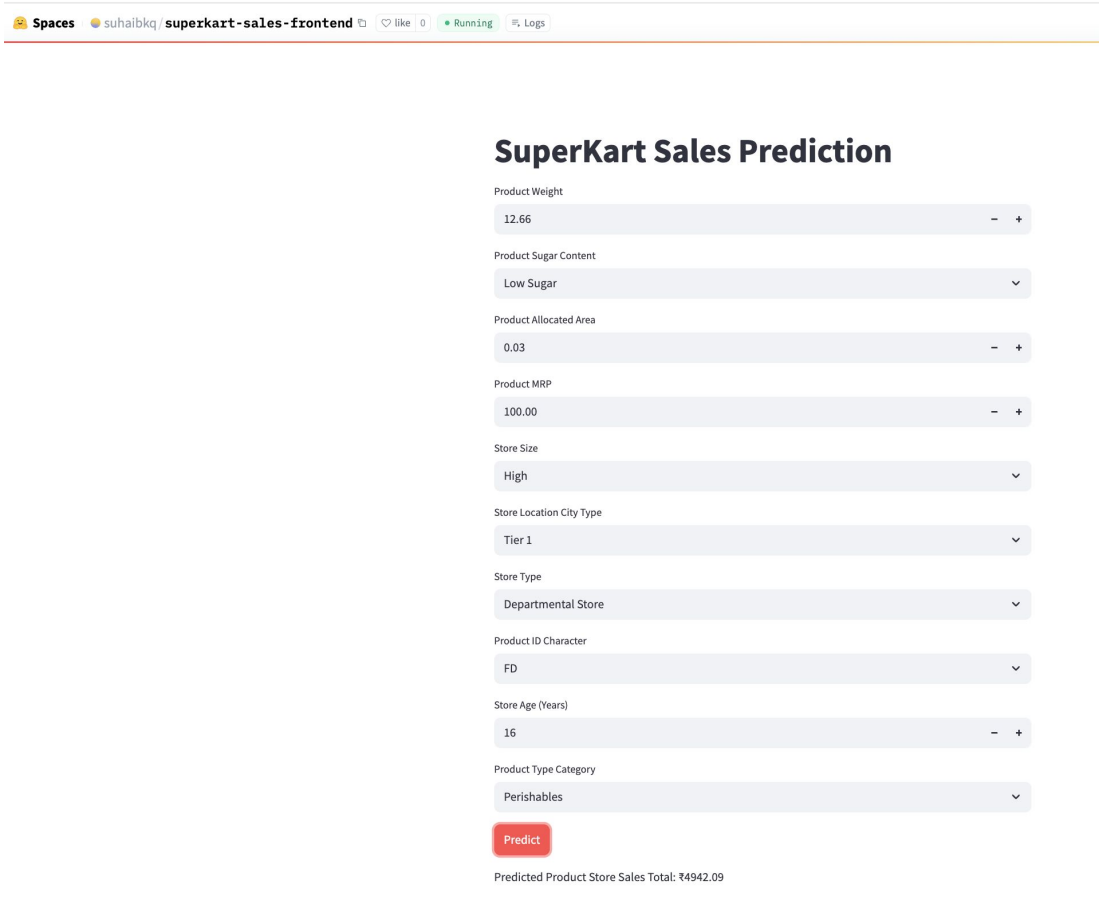
- Open the Space page → **Files and versions**: confirm app.py, requirements.txt, Dockerfile exist.
- Check **Build/Logs** for successful Docker build & app start.
- Open the Space; the Streamlit UI should load. Submit a test payload (matching backend feature names). You should see the **predicted sales**.
- If the app calls a private backend, ensure the **backend URL** is reachable from the Space (CORS/network egress).

Summary of “Deployment – Frontend”

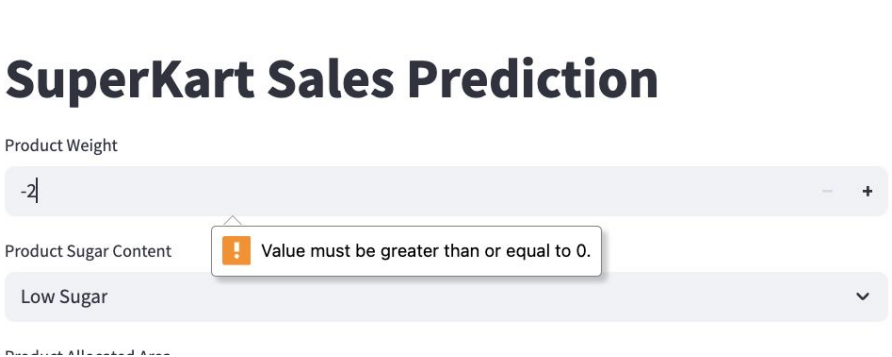
- We built a **Streamlit UI** that collects model inputs and calls the backend’s /predict endpoint.
- We captured runtime dependencies in requirements.txt and containerized the UI with a **Dockerfile**.
- We published the frontend by **uploading the frontend_files/ folder** to a **Hugging Face Space** (Docker SDK).
- The Space auto-builds and serves the Streamlit container; users can access it via the Space URL and get predictions in a browser.
- This approach gives a **portable, reproducible** frontend, making it easy for business users to run **what-if scenarios** and see **immediate predictions** without touching Python or notebooks.

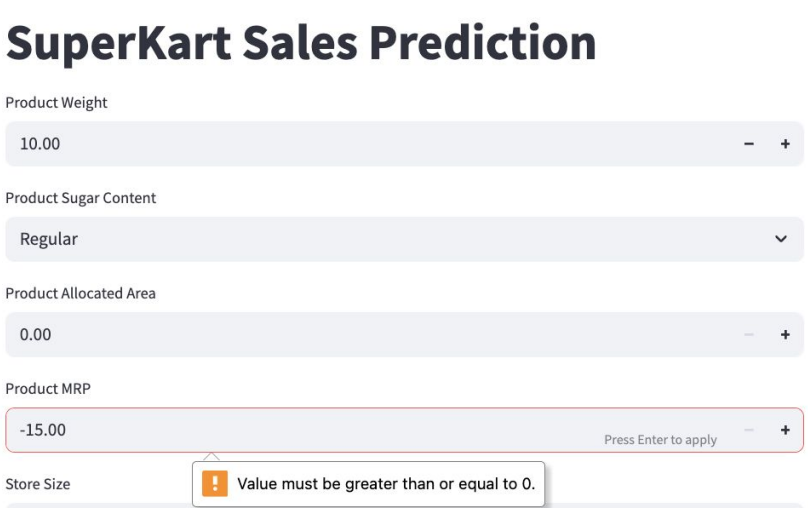


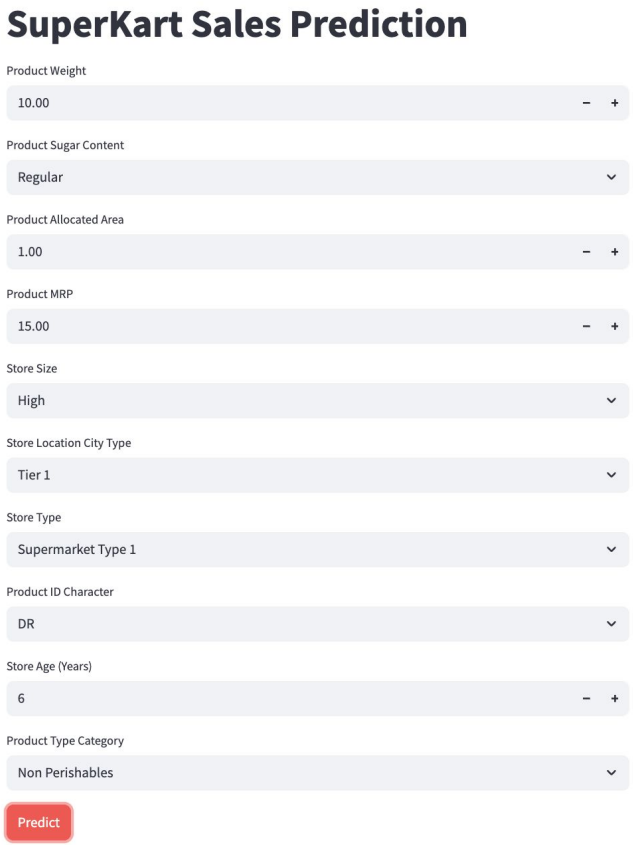
App Front End
https://huggingface.co/spaces/suhaibkq/superkart-sales-frontend



Frontend Sample Screenshots







Key Insights

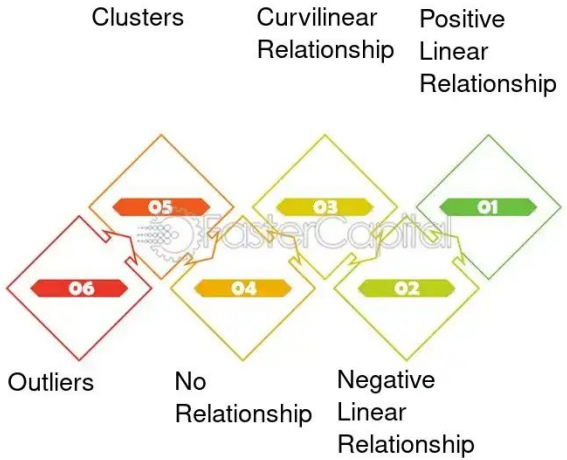
- **Pricing is the #1 driver** of product-store sales (strong positive link between **MRP** and sales value).
- **Store context matters**: larger stores and Tier-1 locations show **higher per-product sales**; one flagship (**OUT004**) dominates revenue and assortment.
- **Category winners**: **Fruits & Vegetables, Snack Foods, Frozen Foods** lead revenue; **Low/Regular sugar** products drive the bulk of sales.
- **Weak levers**: **Product weight** and **allocated shelf area** have weak relationships with sales → planograms and packaging weight aren't the main levers.

Univariate Insights - What the data looks like

- **Sales** are centered around a mid range with a modest tail of high performers (fast movers).
- **MRP (price)** clusters in a mid band with a right tail (some premium SKUs).
- **Weight** concentrates tightly (11–14 kg); only a few outliers.
- **Sugar content**: “Low” and “Regular” dominate the assortment.
- **Store mix**: majority **Medium** stores; most entries come from **Tier-2** locations.
- **Store type**: **Supermarket Type 1/2** are the main formats; Food Marts are minority.

Bivariate Insights - What drives what

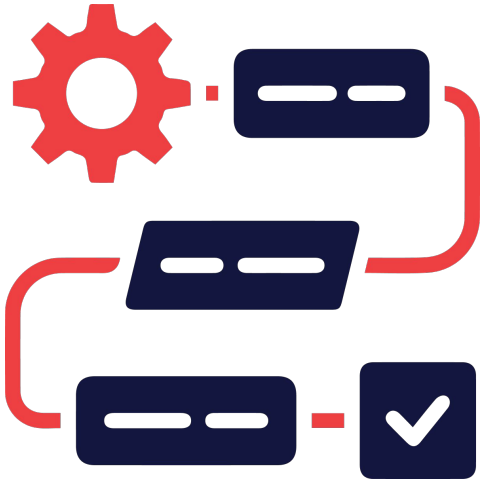
- **Correlation matrix**: MRP ↗ vs Sales; weak links for Weight & Allocated Area.
- **Scatter**:
  - **MRP vs Sales**: clear positive trend (pricing/assortment effects).
  - **Weight vs Sales** and **Allocated Area vs Sales**: little structure.
- **By category**: **F&V, Snacks, Frozen** drive revenue; long-tail categories contribute little.
- **By sugar level**: **Low** and **Regular** sugar products generate most revenue.
- **By store**: **OUT004** is the flagship (broadest price range & category coverage).
- **By store size & city tier**: **Large** stores and **Tier-1** locations yield higher **per-product** sales (even if Tier-2 totals are big due to store count).
- **By store type**: **Supermarkets** outperform **Food Marts**.



What to do - Actionable Recommendations 1/3

1) Pricing and Promotion

- **Run price tests** on top-selling SKUs: A/B 2–5% price lifts where elasticity is low (start with Snacks/Frozen in large stores; monitor unit drop vs. margin).
- **Use markdown science** on slow movers: structured discount ladders to clear long-tail categories without training customers to wait for sales.
- **Bundle/value packs**: pair high-MRP items with complementary staples (e.g., Frozen entrée + side) to raise basket value.



2) Assortment and Space

- **Expand winners**: deepen **F&V, Snacks, Frozen** facings and SKU variety, especially in **Large/Tier-1** stores where per-product sales are strongest.
- **Rationalize long tail**: trim low-velocity categories and SKUs (Breakfast/Seafood niches) to free space and working capital.
- **Revisit planograms**: since **Allocated Area ≠ Sales**, measure **sales per square inch** and reallocate space toward high productivity SKUs/categories.

What to do - Actionable Recommendations 2/3

3) Store and Format Strategy

- **Replicate flagship playbook**: audit **OUT004** (pricing tiers, assortment breadth, promotions, end caps) and roll best practices into similar stores.
- **Focus upgrades**: where feasible, shift Medium → Large footprints in select **Tier-1** trade areas (or emulate "large-store" assortments via curated expansions).
- **Food Mart uplift**: concentrate on top-velocity essentials + premium hero SKUs; avoid broad, low-turn assortments.

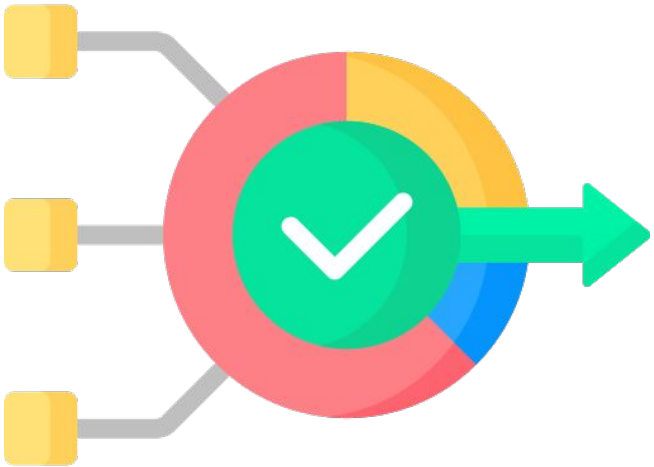
4) Health and Sugar Segmentation

- **Lean into Low/Regular sugar** demand: expand low-sugar alternatives in high-traffic categories; front-of-store visibility + health-oriented promos.
- **Micro-promote** "No sugar" SKUs around health events/aisles; track incremental, don't over-assort permanently.

What to do - Actionable Recommendations 3/3

5) Inventory and Replenishment

- **Use the model forecasts** to drive reorder points & safety stock at **category × store** level; target **stockout reduction** on fast movers.
- **Shorter cycles** for perishables (F&V, Dairy) to cut shrink; **dynamic safety stock** by store size/tier and demand variance.
- **Exception dashboards**: flag SKUs where forecast error (MAPE) is high for manual review.



6) Marketing and CX

- **Personalize offers** by store archetype: Large/Tier-1 = premium and newness; Medium/Tier-2 = value packs and staples.
- **Cross-sell** in app and at shelf (e.g., Frozen mains → sides/desserts) to lift basket.

Operational and Analytics Next Steps

Near Term (2–4 weeks)

- Stand up **price tests** (top 20 SKUs × 10 pilot stores), with guardrails and weekly readouts (volume, margin, mix shift).
- Launch **assortment trim** on bottom-decile SKUs (space → winners).
- Integrate the **XGBoost forecast** into replenishment for the top categories; measure **OOS rate** and **waste** deltas.

Medium Term (4–12 weeks)

- Build **sales/space productivity** scorecards by store & category; re-planogram quarterly.
- Expand to **tiered price architecture** (good/better/best) in Large/Tier-1 stores.
- Develop **category playbooks** (F&V, Snacks, Frozen) with specific SKU roles (traffic, margin, image).

Data and Model Improvements

- Add **promo flags, holidays, seasonality, competitor pricing, footfall** to features; move to **time-granular** targets (weekly).
- Mitigate **store imbalance** (OUT004 dominance) via stratified CV or store-weighted loss; monitor fairness by store cohort.
- Set up **model monitoring** (R^2/RMSE/MAPE drift, data drift) and a quarterly re-tune cycle.

KPIs to Track

- **Revenue** & **gross margin** per store and per square foot.
- **Sales per facing** / sales per square inch by category.
- **OOS rate** (top SKUs) & **waste** (perishables).
- **Promo ROI** and **price elasticity** by SKU/store cohort.
- **Forecast accuracy** (MAPE) by category × store.

Conclusions — Analysis and Modeling

- **What the data says (EDA)**
  a. **Pricing (MRP) is the strongest driver** of product-store sales; **weight** and **allocated shelf area** have weak relationships with sales.
  b. **Category winners**: Fruits & Vegetables, Snack Foods, Frozen Foods. **Low/Regular sugar** products dominate revenue.
  c. **Store effects matter**: Large stores and Tier-1 locations deliver **higher per-product sales**; **OUT004** is a flagship with the broadest assortment and revenue share.
- **Model outcome**
  a. **Chosen model: XGBoost (tuned)** — **$R^2 \approx 0.668$**, **RMSE ≈ 616**, **MAE ≈ 485**, **MAPE ≈ 18.7%** on the test set.
  b. **Why this model**: Best balance of **accuracy, stability, and generalization**; robust to mixed feature types and nonlinearities. Tuned Decision Tree underfit markedly post-tuning.
  c. **Limits to note**: Gains from tuning were modest because key hyperparameters (e.g., learning_rate, max_depth, min_child_weight, L1/L2) were not fully explored; features lack promo/seasonality signals.
- **Data quality & gaps**
  a. **Strengths**: Clean dataset (no missing/duplicate), consistent schema, meaningful product/store attributes.
  b. **Gaps**: No **promotion**, **holiday/seasonality**, **competitor price**, **footfall**, or **time-granular** signals; **store imbalance** (OUT004) can bias learning and evaluation.

Business Recommendations 1/2

Pricing and Margin

- Run **controlled price tests** on top SKUs (±2–5%) starting in large/Tier-1 stores; track volume, margin, and mix to estimate elasticity and lock in profitable price points.
- Use **markdown science** for slow movers (structured clearance ladders) to protect margin while reducing aged inventory.

Assortment and Space

- **Expand winners** (F&V, Snacks, Frozen) in high-performing stores; reduce long-tail, low-velocity SKUs to free space and working capital.
- Re-allocate space by **sales per square inch/facing** (since area alone isn't predictive) and refresh planograms quarterly.

Store Strategy

- **Replicate OUT004 playbook** (assortment breadth, promo cadence, price bands) to matched stores; tighten focus in Food Marts to essentials + premium "hero" items.
- Prioritize **Large/Tier-1** locations for premium tiers/newness; emphasize value packs in **Medium/Tier-2**.

Business Recommendations 2/2

Inventory and Replenishment

- Use the model's forecasts to set **store-category reorder points & safety stocks**; target lower **stock outs** on fast movers and lower **shrink** in perishables with shorter cycles.
- Create **exception dashboards** (high forecast error SKUs) for human review and corrective actions.

Marketing and CX

- Lean into **Low-sugar** demand with targeted end-caps and app promotions; seed **No-sugar** options via limited trials and health events.
- Cross-sell bundles (e.g., Frozen mains + sides/desserts) to **lift basket value**.

Technology and Process Recommendations

Feature and Model Improvements

- Add **promo flags, seasonality/holiday calendars, footfall, competitor price**, and **weather** (for perishables); move to **weekly** targets.
- Expand hyperparameter search (learning rate, depth, min_child_weight, regularization) with **early stopping**; evaluate uplift via **backtests** and **store-level CV**.

Operationalization

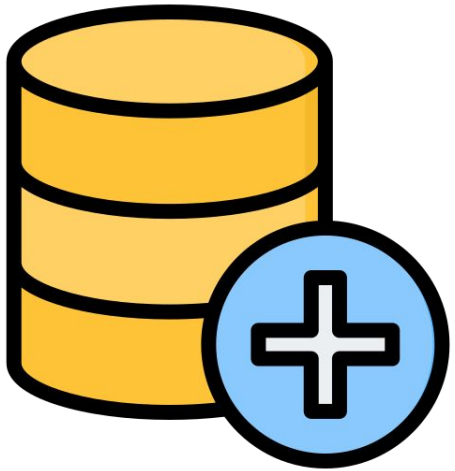
- Ship the **serialized pipeline** (preprocess + model) behind a lightweight **Flask API**; containerize with **Docker**; deploy on **Hugging Face Spaces** (or internal infra).
- Implement **MLOps guardrails**: model registry & versioning, data/feature drift checks, KPI monitoring (R^2/RMSE/MAPE by store/category), and a **quarterly retrain** cadence.

Measurement and Governance

- Define a core KPI set: **Revenue & gross margin**, **sales/facing**, **OOS rate**, **waste (perishables)**, **promo ROI**, **price elasticity**, **forecast MAPE** by store × category.
- Stand up **A/B testing** and **post-promo read** templates; maintain a playbook of **winning interventions** by store archetype.

Recommendations for Additional Data and Why it is Needed

- **Cuts bias & noise**: fills in key demand drivers currently missing (promos, seasonality, competition), reducing unexplained variance and forecast error.
- **Enables decisions**: supports price tests, promo planning, space resets, and inventory policies with credible “what-if” scenarios.
- **Lowers costs**: better forecasts → fewer stockouts, lower waste (especially perishables), higher margin.



What Additional Data to Add - Rationale and Examples 1/2

Commercial and Marketing

- **Promotions & discounts**: promo flag, depth (% off), mechanics (BOGO, bundle), ad circular inclusion, end-cap/feature displays → major drivers of short-term lift.
- **Price history & elasticity signals**: historical MRP by week, competitor price index → enables true price-response modeling.
- **Digital & media**: email push counts, coupon redemptions, app impressions/clicks, paid search/social spend → capture demand stimulation.

Market Data

Time, Events and Context

- **Calendar & seasonality**: week number, holidays, payday weeks, school terms, festivals/events → recurring patterns.
- **Weather** (by store): temp, precipitation, heat waves/cold snaps → strong for F&V, beverages, ice cream, soups.

What Additional Data to Add - Rationale and Examples 2/2

Operations and Supply Chain

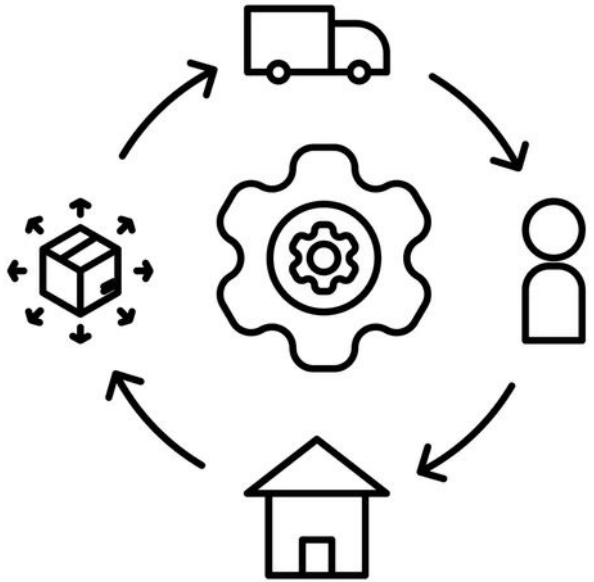
- **Inventory states**: on-hand, on-order, stockouts, plan vs actual delivery dates → distinguish true demand from lost sales.
- **Lead times & supplier reliability**: improves safety-stock and reorder calculations.

Customer and Market

- **Loyalty & basket** (aggregated): segment mix, trip frequency, basket size, category affinities → cross-sell and localized assortment.
- **Local demographics & trade area**: income bands, household size, vehicle ownership; **competition density** and competitor openings/closures.

Merchandising and Product

- **Planogram position**: shelf height, eye-level flag, adjacency; **facing count** → more predictive than raw “allocated area”.
- **SKU attributes**: brand equity, private label flag, pack size, organic/health claims, shelf life/perishability, vendor terms.



Recommendations to Improve the Project 1/2

1) Expand the Data and Targets

- **Move from revenue to units** as the primary target (then revenue = units × price). This separates demand from price and is essential for price optimization.
- **Time-granularity**: forecast **weekly** (or daily for fast movers) at **store × SKU**.
- **Feature store**: centralize engineered features (e.g., last-4-week velocity, promo lags, rolling weather means) for consistency across training/serving.

2) Modeling Enhancements

- **Richer boosting**: try **LightGBM/CatBoost** (categorical handling, speed) alongside XGBoost; run **Bayesian/Optuna** tuning (depth, learning_rate, min_child_weight, reg_*, subsampling).
- **Hierarchical/pooled models**: capture store & category effects (e.g., mixed-effects, hierarchical boosting, or entity embeddings) to handle store imbalance (OUT004).
- **Time-series aware**: add **Temporal Cross-Validation**; consider **Temporal Fusion Transformer** or **LightGBM + lag/seasonal features** for strong weekly forecasts.
- **Causal & uplift**: for promos/pricing, use **DID, causal forests, or uplift models** to estimate incremental effect rather than correlation.
- **Uncertainty**: add **quantile regression** (P50/P80/P90) to set service levels and safety stock.

Recommendations to Improve the Project 2/2

3) Price and Promo Optimization

- Build **elasticity curves** per SKU (or cluster) and simulate price ladders (±2–10%). Optimize **margin subject to volume** and stock constraints.
- Model **promo lift decay** and **cannibalization** across adjacent SKUs/categories.

4) Real-time and Batch Deployment

- **Batch nightly** forecasts for replenishment (top SKUs/categories).
- **Near real-time** scoring for price/promo “what-ifs” in the Streamlit app; cache recent features.
- Add **MLOps**: model registry, A/B canary deploys, data/feature drift alerts, KPI dashboards (R^2/RMSE/MAPE by store/category), and **quarterly retrains**.

5) Evaluation and CV Strategy

- Use **GroupKFold by Store_Id** (or by week blocks) to avoid leakage and over-optimistic scores.
- Track **business KPIs** in parallel: OOS rate, waste %, sales/facing, promo ROI, gross margin $.