CPE 393 Special Topic III: Machine Learning Operations
Dynamic Pricing Optimization for Online Products

Group Members

64070503440	Pichyapa	Khanapattanawong	(Section 31)
64070503451	Sakolkrit	Pengkhum	(Section 31)
64070503473	Ruedhaidham	Soros	(Section 31)
67540460076	Jules	Hoppenot	(Section 31)

This project is created as a partial requirement of
CPE 393 Special Topic III: Machine Learning Operations
Computer Engineering, International Program
King Mongkut's University of Technology Thonburi

Table of Content

Introduction

Idea of Project / Background

In today's highly competitive e-commerce landscape, pricing strategies play a crucial role in maximizing revenue and maintaining market position. Many companies are shifting towards dynamic pricing, where prices are adjusted in real-time based on various factors such as demand, seasonality, competition, and historical sales trends. This approach allows businesses to stay competitive and optimize profitability while responding to market fluctuations efficiently.

With the rise of machine learning and data-driven decision-making, automated pricing optimization has become a key strategy for online retailers. By leveraging historical data and predictive analytics, businesses can set optimal prices that maximize revenue while maintaining a competitive edge.

Project Scope

This project is structured into several key phases, from data collection to model deployment and evaluation. The scope of work includes the following:

1. Data Collection and Preprocessing:
   - Gather and clean data from the Sales and Inventory Snapshot Dataset.
   - Merge relevant datasets, including sales history, product attributes, retail pricing, cost of goods, and seasonal indicators.
   - Perform feature engineering to create relevant variables for the pricing model, such as demand elasticity, price fluctuations, and time-based effects
2. Machine Learning Model Development:
   - Train a base LightGBM model to predict demand at different price levels.
   - Compare alternative models (XGBoost, CatBoost) to determine the most effective approach and find the model for optimal prediction accuracy.
3. Deployment and Monitoring:
   - Containerize the model using Docker for seamless deployment.
   - Set up MLflow for model tracking and versioning.
   - Use Evidently AI to monitor model performance and detect data drift.
   - Automate periodic retraining workflows using Apache Airflow to keep the pricing model up to date.
4. Evaluation and Business Impact Analysis:
   - Conduct testing with simulated pricing scenarios.

Objective

The primary objective of this project is to develop a machine learning-based pricing optimization system that helps businesses dynamically adjust their product prices to maximize revenue and maintain competitiveness. Using historical sales data, pricing trends, and seasonal indicators, we aim to:

1. Train a predictive model that estimates demand at different price points.
2. Implement a pricing optimization engine that selects the most profitable price for each product based on predefined business objectives (e.g., maximum revenue, market share, or price elasticity).

Proposed Methods

1. Preprocess the data from the Sales and Inventory Snapshot dataset; Sales snapshot, Inventory snapshot, Retail price, Cost of goods, and Master Calendar.
2. Perform feature engineering to allow the data to be processed by the model
3. Train a base machine learning model to predict `sold_quantity`, how many products will be sold based on given price, discount, and other factors.
4. Implement a pricing optimization engine which uses the model to predict the sold_quantity at different prices in a price range, then select the best price and discount based on the business objective. (Maximum profit, Maximum sales, etc.)
5. Expose the price recommendation model as API.

Methodology and Design

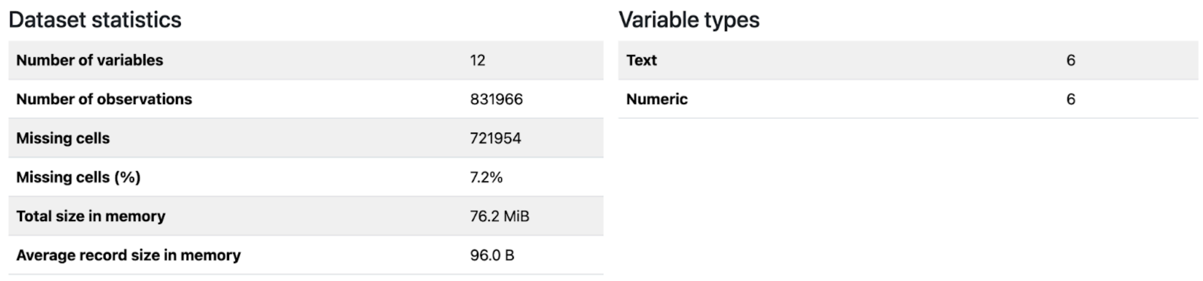
Dataset and Exploratory Data Analysis

For this project, to train the model, we selected the Sales and Inventory Snapshot Dataset, which provides comprehensive historical data from fashion retail operations. The dataset includes detailed records of product-level sales, inventory levels, retail pricing, cost of goods (COGs), product attributes, store information, and calendar-based seasonal indicators.

We chose this dataset as it contains all key components required which are historical data of sale and stock, product details, cost of goods, and seasonal context. This should allow us to create the MVP model that is able to recognize seasonal effects and stock availability in addition to the fundamental information required to predict the most effective pricing.

We will be using 6 datasets which are;

1. Sales Snapshot: Historical sales data for time-series analysing, requires preprocessing to merge different snapshots and format dates.



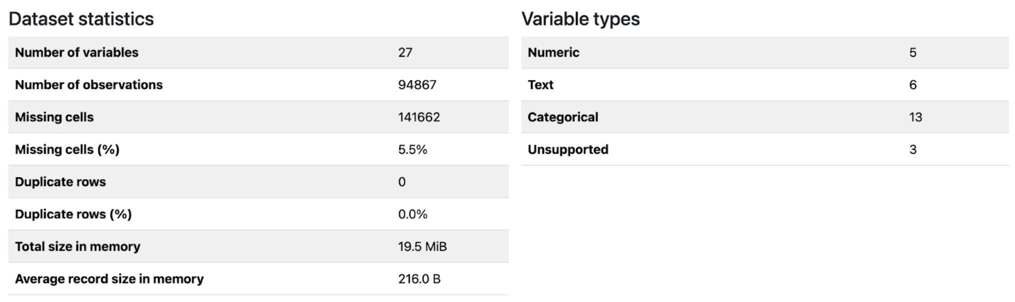
Dataset statistics

Number of variables	12
Number of observations	831966
Missing cells	721954
Missing cells (%)	7.2%
Total size in memory	76.2 MiB
Average record size in memory	96.0 B

Variable types

Text	6
Numeric	6

2. Product Master: Static product attributes, requires preprocessing for universal units and language as the dataset is from Vietnam. Help formulate prices based on different product types.

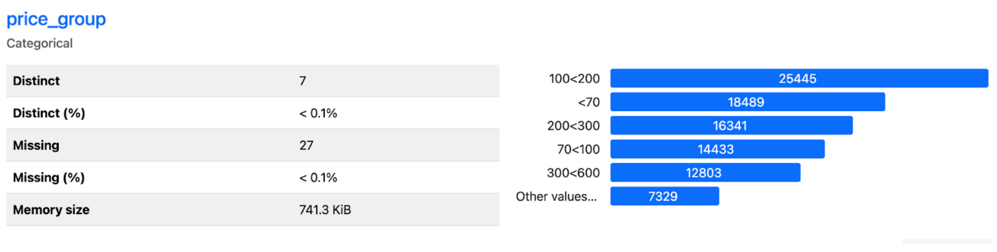


Dataset statistics

Number of variables	27
Number of observations	94867
Missing cells	141662
Missing cells (%)	5.5%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	19.5 MiB
Average record size in memory	216.0 B

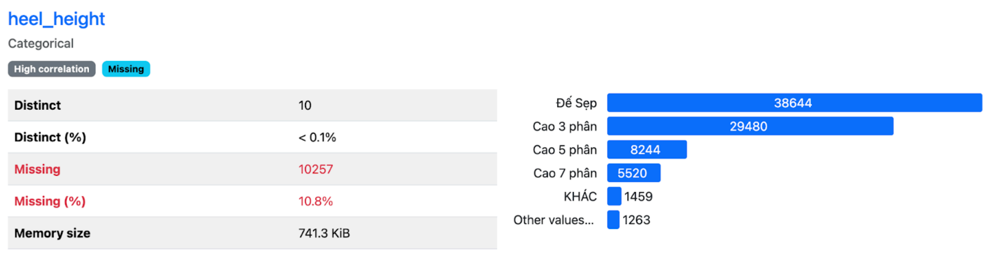
Variable types

Numeric	5
Text	6
Categorical	13
Unsupported	3



price_group
Categorical

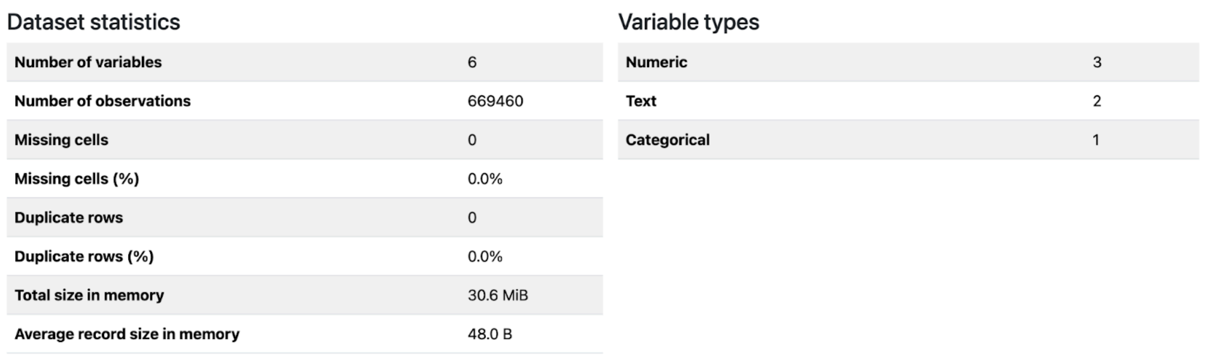
Distinct	7
Distinct (%)	< 0.1%
Missing	27
Missing (%)	< 0.1%
Memory size	741.3 KiB



heel_height
Categorical
High correlation Missing

Distinct	10
Distinct (%)	< 0.1%
Missing	10257
Missing (%)	10.8%
Memory size	741.3 KiB

3. Retail Price: This dataset lists the official retail price (MSRP) for each product_id over time, serves as pricing anchor. Requires preprocessing to format dates.

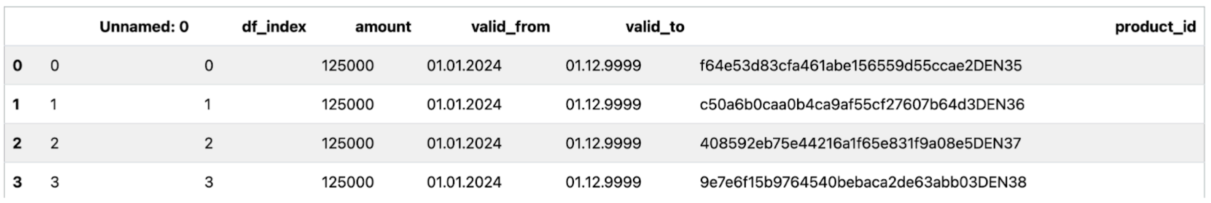


Dataset statistics

Number of variables	6
Number of observations	669460
Missing cells	0
Missing cells (%)	0.0%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	30.6 MiB
Average record size in memory	48.0 B

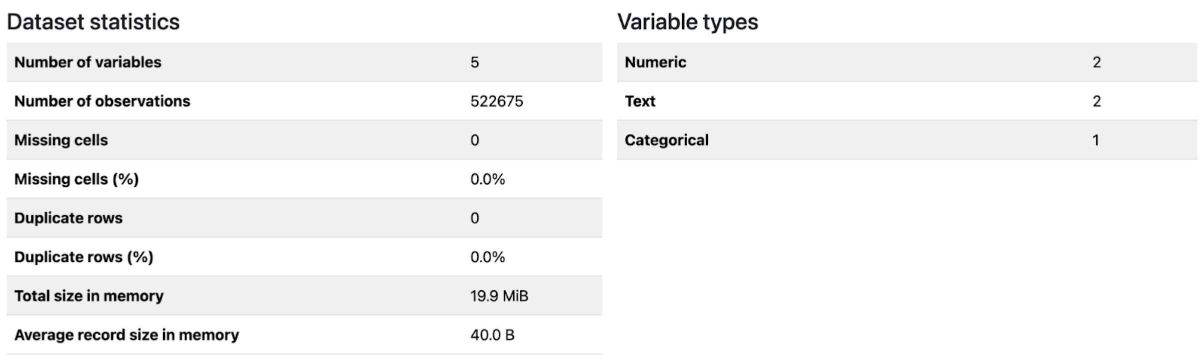
Variable types

Numeric	3
Text	2
Categorical	1



	Unnamed: 0	df_index	amount	valid_from	valid_to	product_id
0	0	0	125000	01.01.2024	01.12.9999	f64e53d83cfa461abe156559d55ccae2DEN35
1	1	1	125000	01.01.2024	01.12.9999	c50a6b0caa0b4ca9af55cf27607b64d3DEN36
2	2	2	125000	01.01.2024	01.12.9999	408592eb75e44216a1f65e831f9a08e5DEN37
3	3	3	125000	01.01.2024	01.12.9999	9e7e6f15b9764540bebaca2de63abb03DEN38

4. COGs (Cost of Goods Sold): Provides the unit cost (amount) of each product, with validity periods, enabling profit calculations.

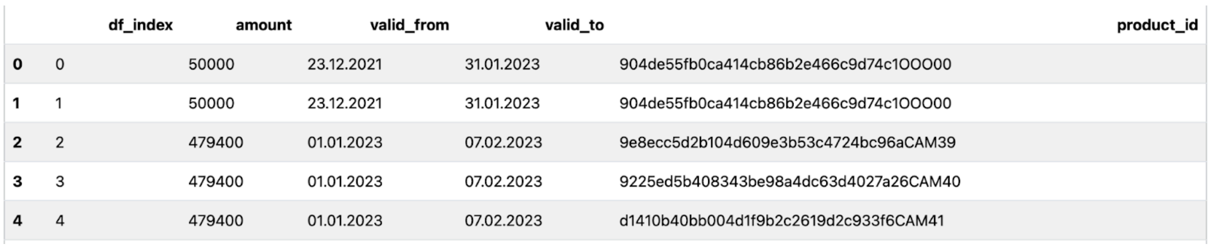


Dataset statistics

Number of variables	5
Number of observations	522675
Missing cells	0
Missing cells (%)	0.0%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	19.9 MiB
Average record size in memory	40.0 B

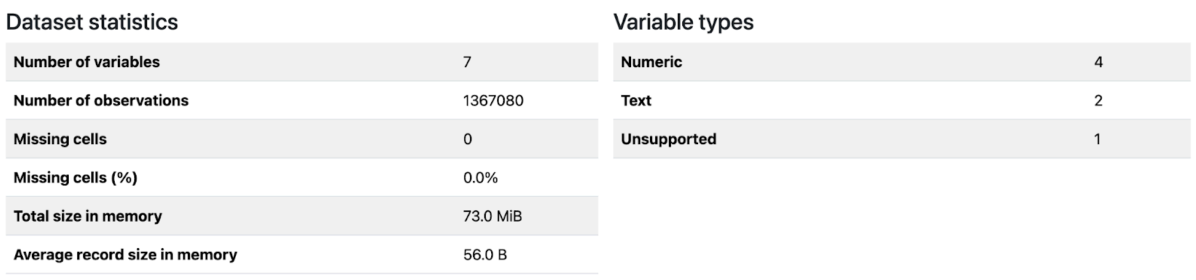
Variable types

Numeric	2
Text	2
Categorical	1



	df_index	amount	valid_from	valid_to	product_id
0	0	50000	23.12.2021	31.01.2023	904de55fb0ca414cb86b2e466c9d74c1OOO00
1	1	50000	23.12.2021	31.01.2023	904de55fb0ca414cb86b2e466c9d74c1OOO00
2	2	479400	01.01.2023	07.02.2023	9e8ecc5d2b104d609e3b53c4724bc96aCAM39
3	3	479400	01.01.2023	07.02.2023	9225ed5b408343be98a4dc63d4027a26CAM40
4	4	479400	01.01.2023	07.02.2023	d1410b40bb004d1f9b2c2619d2c933f6CAM41

5. Inventory Snapshot: Records weekly stock levels (quantity) per product and storage location (sloc, plant). This dataset will be used to model stock-aware pricing and avoid overstock or stockouts, and it requires preprocessing to merge different snapshots and format dates.

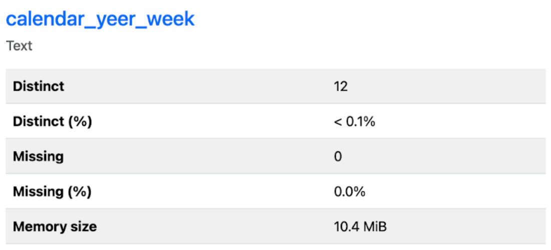


Dataset statistics

Number of variables	7
Number of observations	1367080
Missing cells	0
Missing cells (%)	0.0%
Total size in memory	73.0 MiB
Average record size in memory	56.0 B

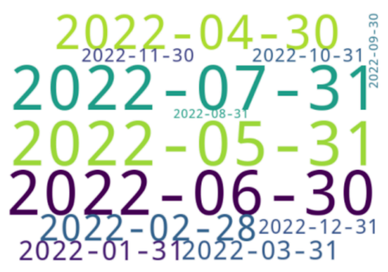
Variable types

Numeric	4
Text	2
Unsupported	1

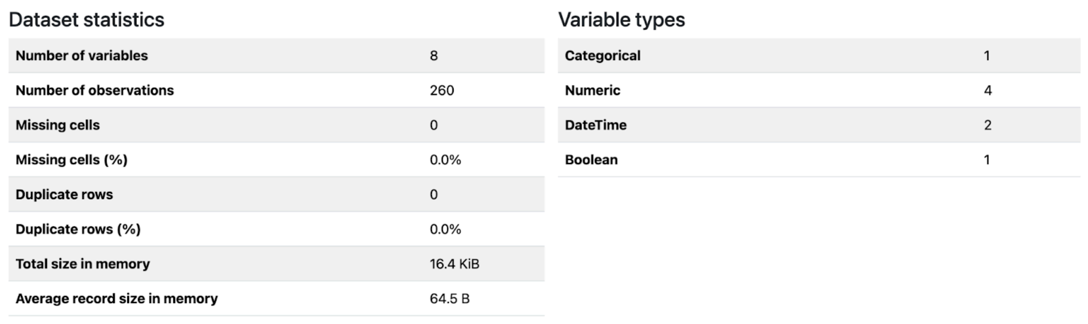


calendar_yeer_week
Text

Distinct	12
Distinct (%)	< 0.1%
Missing	0
Missing (%)	0.0%
Memory size	10.4 MiB



6. Master Calendar: Maps year, month, week, and year_week to actual dates, also includes flags like CNY. This dataset adds time-based features like seasonality, holidays, and demand shifts.

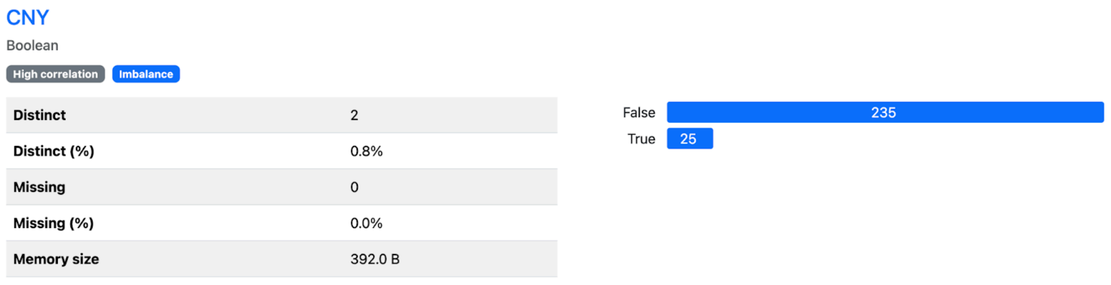


Dataset statistics

Number of variables	8
Number of observations	260
Missing cells	0
Missing cells (%)	0.0%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	16.4 KiB
Average record size in memory	64.5 B

Variable types

Categorical	1
Numeric	4
DateTime	2
Boolean	1



CNY

Boolean

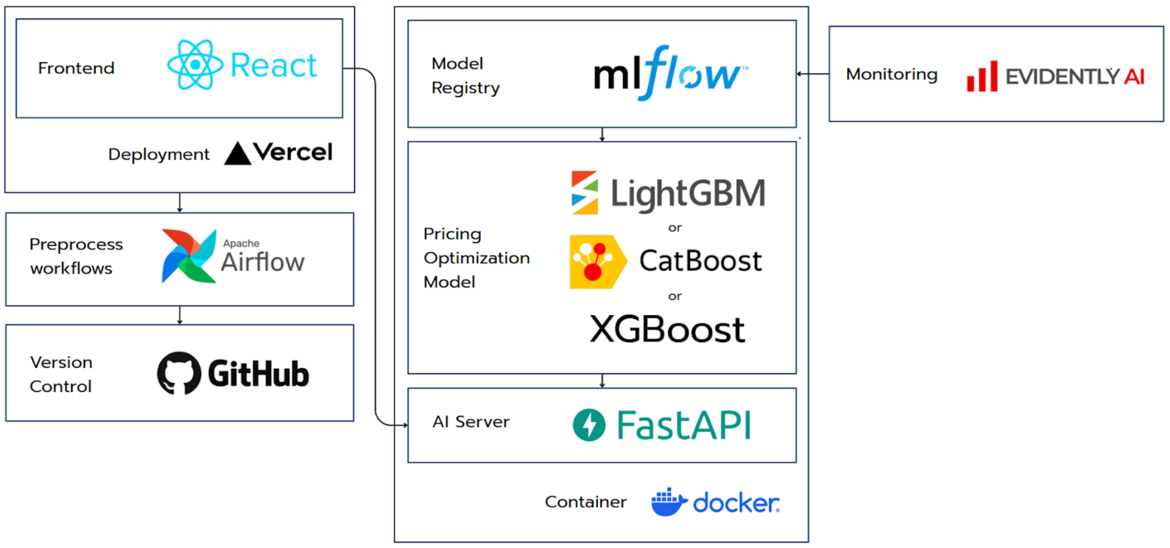
High correlation Imbalance

Distinct	2
Distinct (%)	0.8%
Missing	0
Missing (%)	0.0%
Memory size	392.0 B

False	235
True	25

The 6 datasets will be preprocessed, merged, and selected only necessary features to train the model.

MLOps Tools and Tech Stack

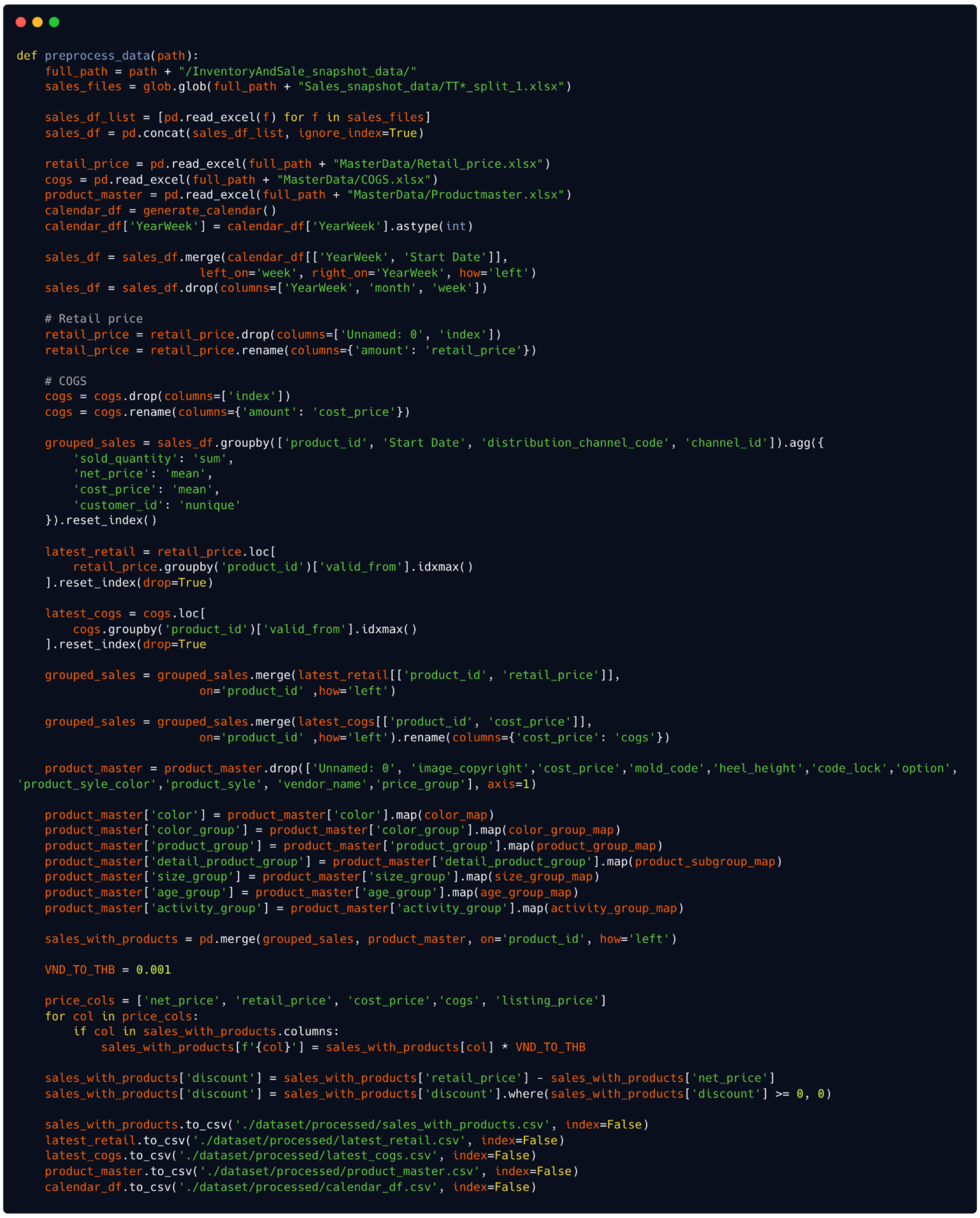


To support the development, deployment, and monitoring of the pricing optimization engine, we designed a modular and scalable MLOps architecture leveraging modern, open-source tools. The frontend is built with React and deployed via Vercel, providing a responsive UI for interacting with the pricing recommendation system. Apache Airflow is used to orchestrate preprocessing workflows such as sales data cleaning, feature generation, and periodic retraining schedules. All code and version control are managed through GitHub, ensuring collaboration, traceability, and enabling CI/CD pipelines.

The core pricing prediction model is built using LightGBM, with CatBoost and XGBoost as alternatives depending on data characteristics. These models are served via a lightweight FastAPI server, containerized using Docker for consistent and scalable deployment. MLflow manages model registry and experiment tracking to ensure reproducibility and model lifecycle governance. For monitoring, we integrate Evidently AI, an open-source tool that tracks data drift, model performance, and key metrics to maintain prediction quality in production. This stack ensures the system is both production-ready and flexible for future iterations or business objectives.

Implementation

Data Preprocessing



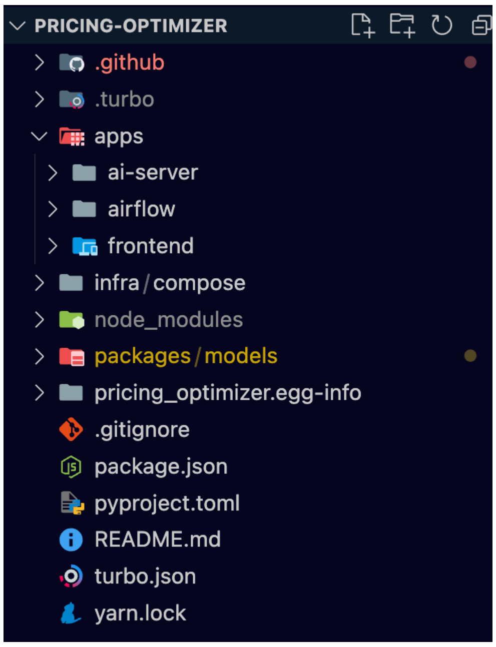
```python
def preprocess_data(path):
    full_path = path + "/InventoryAndSale_snapshot_data/"
    sales_files = glob.glob(full_path + "Sales_snapshot_data/TT*_split_1.xlsx")

    sales_df_list = [pd.read_excel(f) for f in sales_files]
    sales_df = pd.concat(sales_df_list, ignore_index=True)

    retail_price = pd.read_excel(full_path + "MasterData/Retail_price.xlsx")
    cogs = pd.read_excel(full_path + "MasterData/COGS.xlsx")
    product_master = pd.read_excel(full_path + "MasterData/Productmaster.xlsx")
    calendar_df = generate_calendar()
    calendar_df['YearWeek'] = calendar_df['YearWeek'].astype(int)

    sales_df = sales_df.merge(calendar_df[['YearWeek', 'Start Date']],
                      left_on='week', right_on='YearWeek', how='left')
    sales_df = sales_df.drop(columns=['YearWeek', 'month', 'week'])

    # Retail price
    retail_price = retail_price.drop(columns=['Unnamed: 0', 'index'])
    retail_price = retail_price.rename(columns={'amount': 'retail_price'})

    # COGS
    cogs = cogs.drop(columns=['index'])
    cogs = cogs.rename(columns={'amount': 'cost_price'})

    grouped_sales = sales_df.groupby(['product_id', 'Start Date', 'distribution_channel_code', 'channel_id']).agg({
        'sold_quantity': 'sum',
        'net_price': 'mean',
        'cost_price': 'mean',
        'customer_id': 'nunique'
    }).reset_index()

    latest_retail = retail_price.loc[
        retail_price.groupby('product_id')['valid_from'].idxmax()
    ].reset_index(drop=True)

    latest_cogs = cogs.loc[
        cogs.groupby('product_id')['valid_from'].idxmax()
    ].reset_index(drop=True

    grouped_sales = grouped_sales.merge(latest_retail[['product_id', 'retail_price']],
                      on='product_id' ,how='left')

    grouped_sales = grouped_sales.merge(latest_cogs[['product_id', 'cost_price']],
                      on='product_id' ,how='left').rename(columns={'cost_price': 'cogs'})

    product_master = product_master.drop(['Unnamed: 0', 'image_copyright','cost_price','mold_code','heel_height','code_lock','option',
'product_syle_color','product_syle', 'vendor_name','price_group'], axis=1)

    product_master['color'] = product_master['color'].map(color_map)
    product_master['color_group'] = product_master['color_group'].map(color_group_map)
    product_master['product_group'] = product_master['product_group'].map(product_group_map)
    product_master['detail_product_group'] = product_master['detail_product_group'].map(product_subgroup_map)
    product_master['size_group'] = product_master['size_group'].map(size_group_map)
    product_master['age_group'] = product_master['age_group'].map(age_group_map)
    product_master['activity_group'] = product_master['activity_group'].map(activity_group_map)

    sales_with_products = pd.merge(grouped_sales, product_master, on='product_id', how='left')

    VND_TO_THB = 0.001

    price_cols = ['net_price', 'retail_price', 'cost_price','cogs', 'listing_price']
    for col in price_cols:
        if col in sales_with_products.columns:
            sales_with_products[f'{col}'] = sales_with_products[col] * VND_TO_THB

    sales_with_products['discount'] = sales_with_products['retail_price'] - sales_with_products['net_price']
    sales_with_products['discount'] = sales_with_products['discount'].where(sales_with_products['discount'] >= 0, 0)

    sales_with_products.to_csv('./dataset/processed/sales_with_products.csv', index=False)
    latest_retail.to_csv('./dataset/processed/latest_retail.csv', index=False)
    latest_cogs.to_csv('./dataset/processed/latest_cogs.csv', index=False)
    product_master.to_csv('./dataset/processed/product_master.csv', index=False)
    calendar_df.to_csv('./dataset/processed/calendar_df.csv', index=False)
```

The dataset, originally provided in fragmented Excel files, includes transactional sales data, retail prices, cost of goods sold (COGS), and detailed product metadata. To ensure temporal alignment, we programmatically constructed a weekly calendar covering the years 2022 to 2023, where each week is uniquely identified by a `YearWeek` key and mapped to its corresponding start and end dates. Sales data were read from multiple weekly files and concatenated into a unified DataFrame. These records were merged with the custom calendar to replace week identifiers with actual dates and aggregated by product ID, date, distribution channel, and channel ID. Aggregation metrics included total sold quantity, average net price, average cost price, and customer count.

Next, master data tables were cleaned by removing irrelevant fields and renaming columns for clarity. The latest retail prices and COGS per product were extracted by identifying the most recent valid entries based on the `valid_from` field. These were merged into the sales data to ensure that pricing analysis reflected the most recent and relevant values. Additionally, product metadata was enriched by mapping categorical variables—such as color, product group, and activity group—using predefined dictionaries, thus standardizing descriptive attributes for consistent downstream usage.

To prepare the dataset for cross-border economic analysis, all monetary values originally recorded in Vietnamese Dong (VND) were converted to Thai Baht (THB) using a fixed exchange rate of 0.0015. Key pricing columns including net price, retail price, and COGS were transformed accordingly. A derived feature, discount, was introduced to quantify the difference between retail and net price, with negative values clipped to zero to avoid anomalies. The final cleaned and feature-engineered dataset was exported into CSV files, alongside auxiliary files such as the processed retail and COGS tables, product metadata, and the generated calendar. This structured output supports robust analytical workflows and facilitates future work on demand prediction, price optimization, and product performance analysis.

Monorepository Organization



```
PRICING-OPTIMIZER
  .github
  .turbo
  apps
    ai-server
    airflow
    frontend
  infra / compose
  node_modules
  packages / models
  pricing_optimizer.egg-info
  .gitignore
  package.json
  pyproject.toml
  README.md
  turbo.json
  yarn.lock
```

The repository is organized into three primary directories: apps, packages, and infrastructure. The apps directory contains three distinct applications: a React-based frontend for user interaction, an AI server responsible for machine learning operations, and an Apache Airflow instance used for orchestrating data workflows. This separation of concerns enhances modularity while maintaining the benefits of unified version control and dependency management.

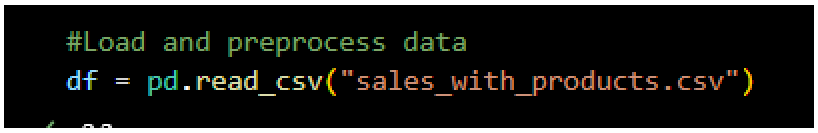
The packages directory serves as a centralized location for shared code, such as data models and utility modules, which are reused across multiple applications. This approach significantly reduces code duplication and ensures consistency in data handling and business logic throughout the system. By enforcing shared type definitions and standardized interfaces, the architecture promotes reliability and ease of maintenance.

To support this multi-application environment, the project leverages modern build tools and package managers. Turborepo, configured via turbo.json, is employed to enable build caching and parallel execution, significantly improving development and deployment performance. The project supports both JavaScript/TypeScript and Python ecosystems, using Yarn for frontend and Node.js packages, and pyproject.toml for managing Python dependencies. This multi-language support is essential, as the system spans web development, machine learning, and workflow automation.

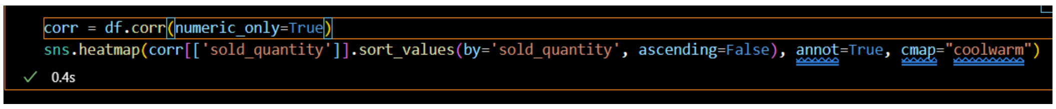
In addition, the architecture reflects best practices in software engineering. Codebases are modular and follow a consistent naming convention, with shared ESLint and TypeScript configurations enforcing coding standards across teams. Documentation is maintained centrally, ensuring that developers have access to unified resources and system overviews, thereby improving onboarding and collaboration.

MLOps Tools and Tech Stack

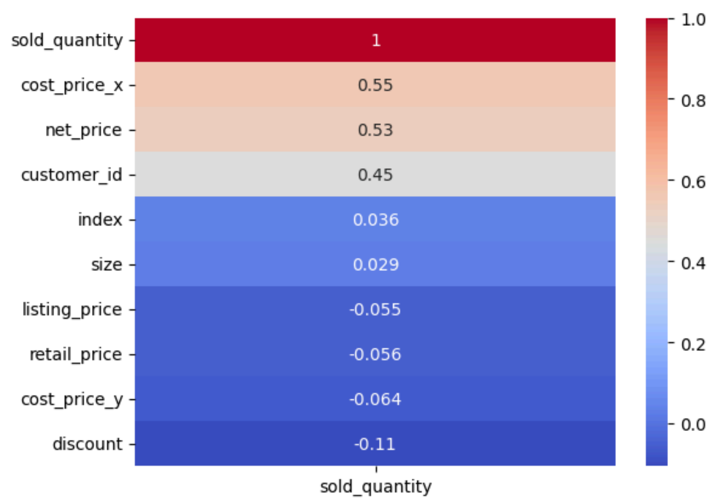
In this project, we aim to compare the performance of three tree-based machine learning models: LightGBM, CatBoost, and XGBoost. To evaluate each model, we use standard regression metrics — MAE (Mean Absolute Error), RMSE (Root Mean Squared Error), and R^2 (Coefficient of Determination). The implementation follows an MLOps workflow that includes preprocessing, feature engineering, hyperparameter tuning using GridSearchCV, model tracking with MLflow, and interpretability with SHAP. Before we're training our model, we need to work on feature engineering beforehand.

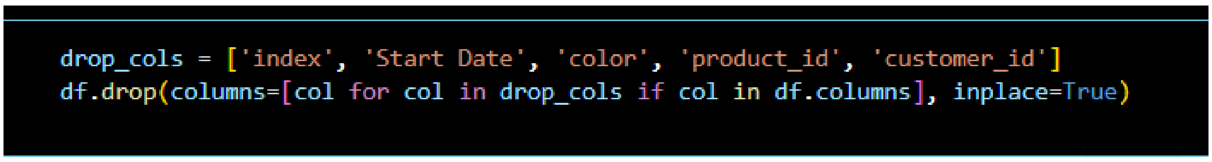


```
#Load and preprocess data
df = pd.read_csv("sales_with_products.csv")
```

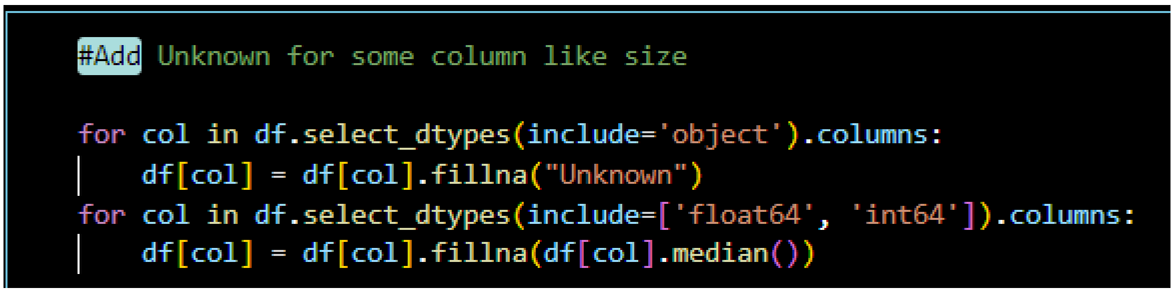


```
corr = df.corr(numeric_only=True)
sns.heatmap(corr[['sold_quantity']].sort_values(by='sold_quantity', ascending=False), annot=True, cmap="coolwarm")
✓ 0.4s
```

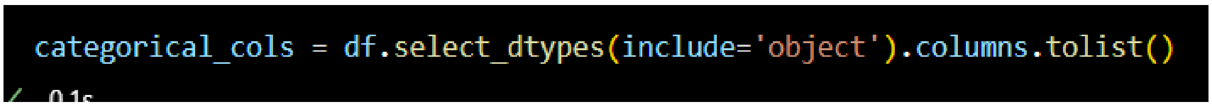




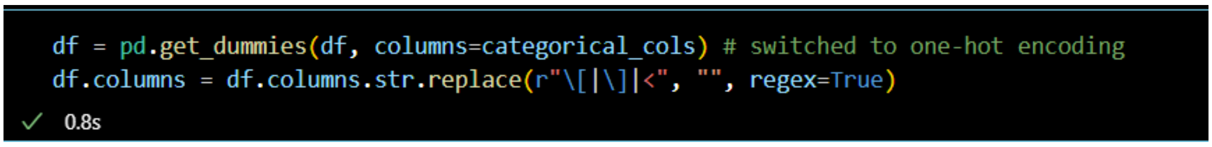
```
drop_cols = ['index', 'Start Date', 'color', 'product_id', 'customer_id']
df.drop(columns=[col for col in drop_cols if col in df.columns], inplace=True)
```



```
#Add Unknown for some column like size

for col in df.select_dtypes(include='object').columns:
    df[col] = df[col].fillna("Unknown")
for col in df.select_dtypes(include=['float64', 'int64']).columns:
    df[col] = df[col].fillna(df[col].median())
```



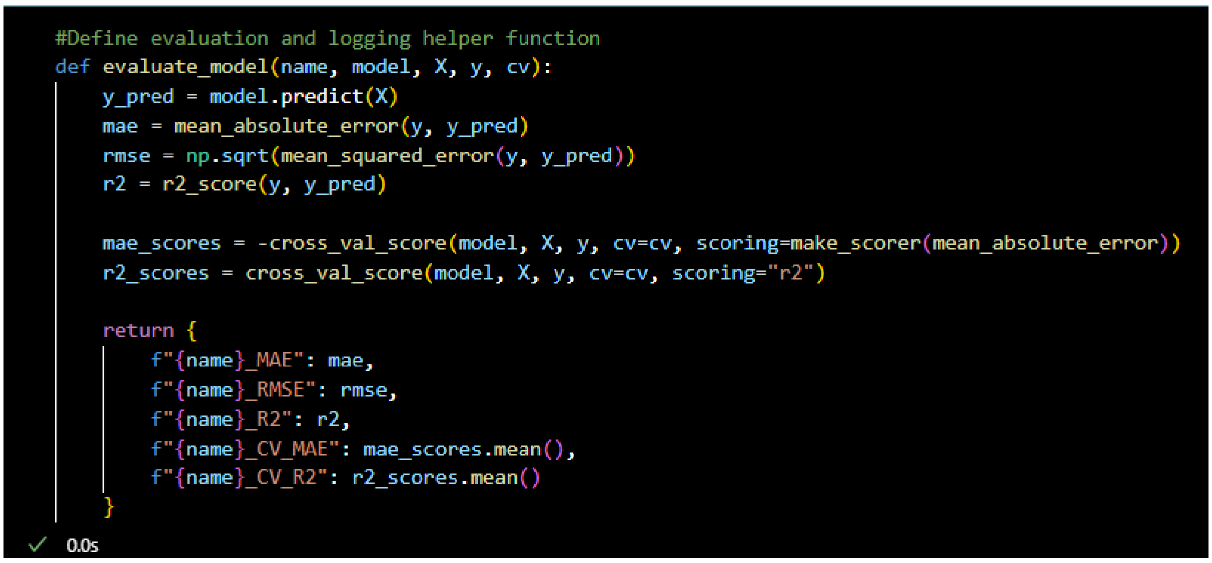
```
categorical_cols = df.select_dtypes(include='object').columns.tolist()
```



```
df = pd.get_dummies(df, columns=categorical_cols) # switched to one-hot encoding
df.columns = df.columns.str.replace(r"\[|\]|<", "", regex=True)
✓ 0.8s
```

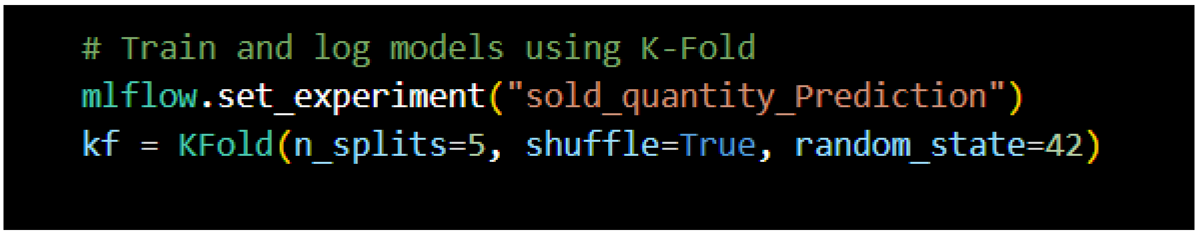
To better understand which features are most relevant to our target variable sold_quantity, we plot a correlation heatmap. This helps us visually identify which numeric variables are strongly or weakly correlated with the target. After loading, we drop irrelevant columns that are either identifiers or carry no predictive value, such as 'index', 'Start Date', 'color', 'product_id', and 'customer_id'. This step ensures the model isn't distracted by noisy or meaningless features. Next, we handle missing values with different criteria. For categorical columns, we fill missing entries with "Unknown" to retain as much data as possible. For numerical columns, we use the median of each column to impute missing values, which is robust against outliers.

We identify categorical columns and apply one-hot encoding to prepare them for tree-based models. After encoding, we clean up the column names to remove characters like [and <, which can cause errors.



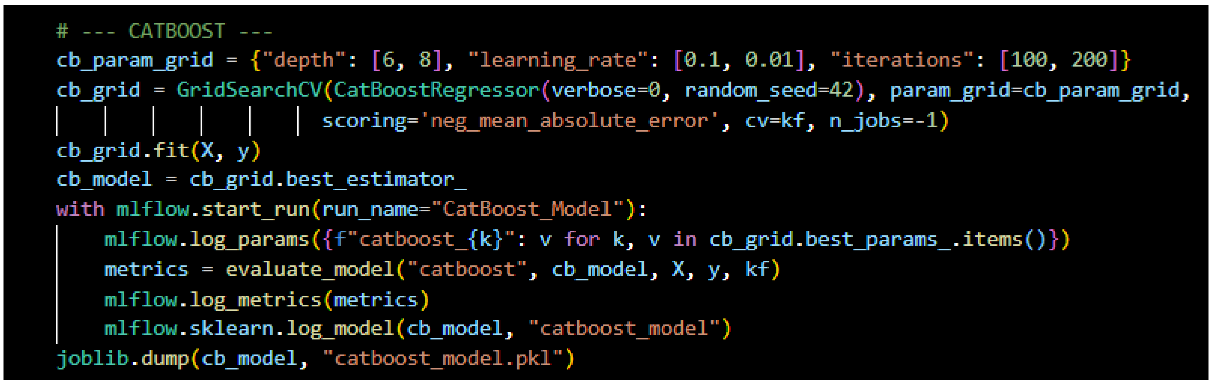
```python
#Define evaluation and logging helper function
def evaluate_model(name, model, X, y, cv):
    y_pred = model.predict(X)
    mae = mean_absolute_error(y, y_pred)
    rmse = np.sqrt(mean_squared_error(y, y_pred))
    r2 = r2_score(y, y_pred)

    mae_scores = -cross_val_score(model, X, y, cv=cv, scoring=make_scorer(mean_absolute_error))
    r2_scores = cross_val_score(model, X, y, cv=cv, scoring="r2")

    return {
        f"{name}_MAE": mae,
        f"{name}_RMSE": rmse,
        f"{name}_R2": r2,
        f"{name}_CV_MAE": mae_scores.mean(),
        f"{name}_CV_R2": r2_scores.mean()
    }
```
```
✓ 0.0s
```

To evaluate models consistently, we implement a utility function that computes both in-sample and cross-validated metrics, including Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R². This function also performs 5-fold cross-validation to ensure robust performance estimates and avoid overfitting to a single train-test split. Metrics from both the full dataset and cross-validation are logged for comparison.



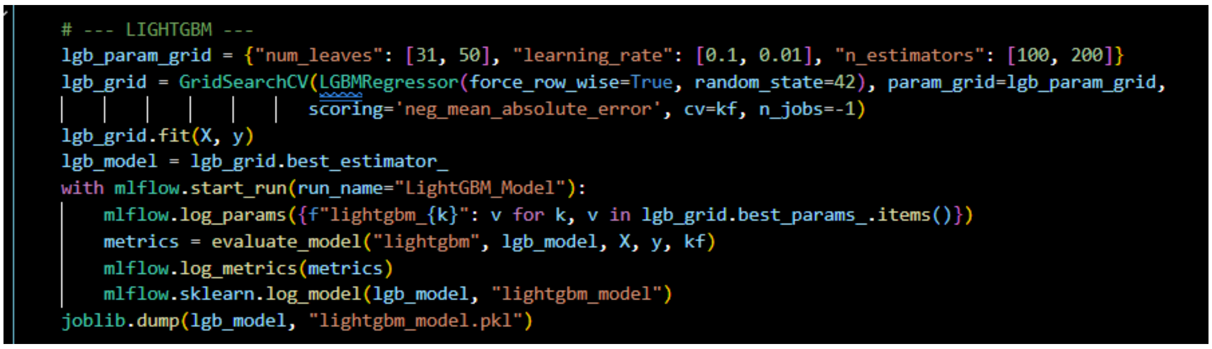
```python
# Train and log models using K-Fold
mlflow.set_experiment("sold_quantity_Prediction")
kf = KFold(n_splits=5, shuffle=True, random_state=42)
```

We use 5-fold cross-validation (KFold) across all models to ensure fair and consistent evaluation. This technique splits the data into five different train-test combinations, reducing the variance of model performance estimates. We configure MLflow to track experiments, allowing us to log and compare hyperparameters, metrics, and trained models across all three boosting methods.



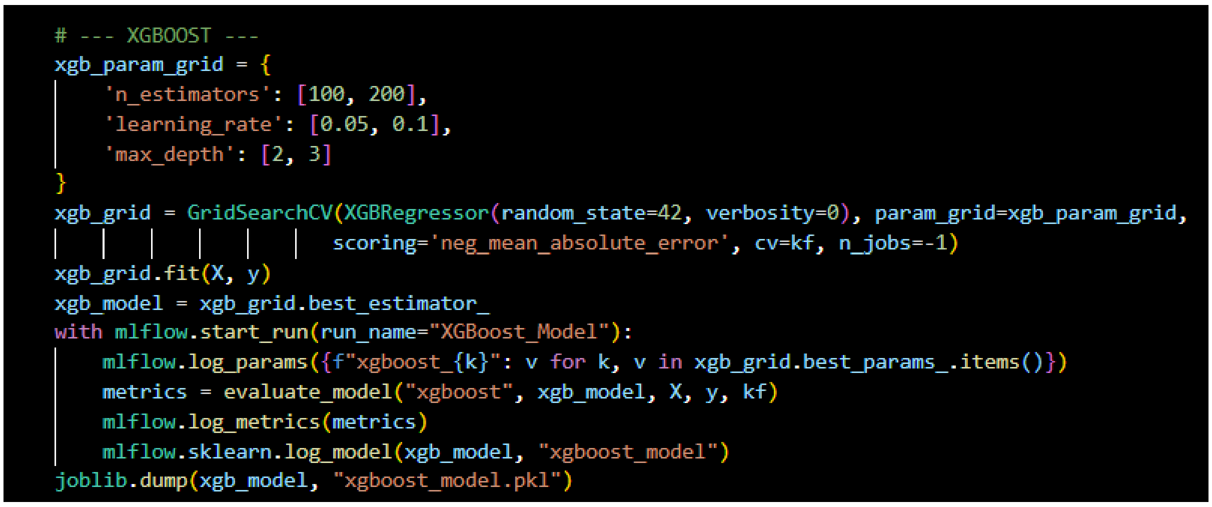
```
# --- CATBOOST ---
cb_param_grid = {"depth": [6, 8], "learning_rate": [0.1, 0.01], "iterations": [100, 200]}
cb_grid = GridSearchCV(CatBoostRegressor(verbose=0, random_seed=42), param_grid=cb_param_grid,
                       scoring='neg_mean_absolute_error', cv=kf, n_jobs=-1)
cb_grid.fit(X, y)
cb_model = cb_grid.best_estimator_
with mlflow.start_run(run_name="CatBoost_Model"):
    mlflow.log_params({f"catboost_{k}": v for k, v in cb_grid.best_params_.items()})
    metrics = evaluate_model("catboost", cb_model, X, y, kf)
    mlflow.log_metrics(metrics)
    mlflow.sklearn.log_model(cb_model, "catboost_model")
joblib.dump(cb_model, "catboost_model.pkl")
```

For model training, we begin with CatBoost. We tune three key hyperparameters: depth (with values 6 and 8), learning_rate (0.1 and 0.01), and iterations (100 and 200). CatBoost is well-suited for categorical data and is robust against overfitting. Despite using one-hot encoding in this case for consistency, CatBoost remains highly performant. We use GridSearchCV to find the best parameter combination and log the results to MLflow. The final model is saved using the joblib library for later use.



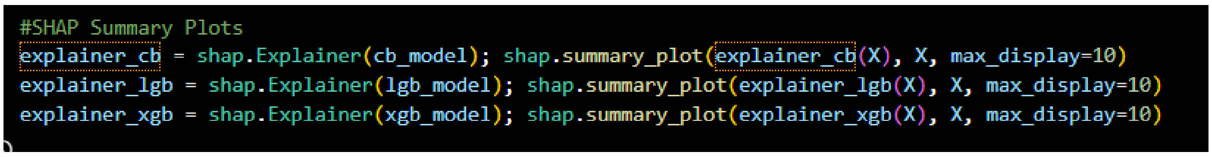
```
# --- LIGHTGBM ---
lgb_param_grid = {"num_leaves": [31, 50], "learning_rate": [0.1, 0.01], "n_estimators": [100, 200]}
lgb_grid = GridSearchCV(LGBMRegressor(force_row_wise=True, random_state=42), param_grid=lgb_param_grid,
                        scoring='neg_mean_absolute_error', cv=kf, n_jobs=-1)
lgb_grid.fit(X, y)
lgb_model = lgb_grid.best_estimator_
with mlflow.start_run(run_name="LightGBM_Model"):
    mlflow.log_params({f"lightgbm_{k}": v for k, v in lgb_grid.best_params_.items()})
    metrics = evaluate_model("lightgbm", lgb_model, X, y, kf)
    mlflow.log_metrics(metrics)
    mlflow.sklearn.log_model(lgb_model, "lightgbm_model")
joblib.dump(lgb_model, "lightgbm_model.pkl")
```

Next, we train the LightGBM model. Here, we tune num_leaves (31 and 50), learning_rate (0.1 and 0.01), and n_estimators (100 and 200). LightGBM is optimized for speed and memory efficiency, making it suitable for large-scale tasks. Just like with CatBoost, we evaluate the model with 5-fold cross-validation and log all relevant artifacts and metrics to MLflow. The best-performing model is then exported using Joblib.

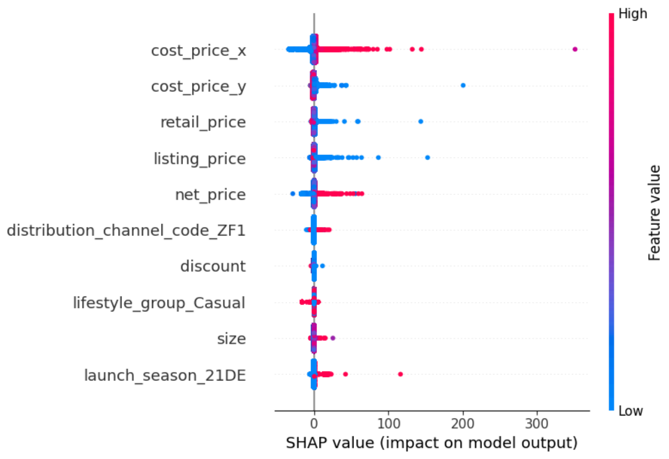


```
# --- XGBOOST ---
xgb_param_grid = {
    'n_estimators': [100, 200],
    'learning_rate': [0.05, 0.1],
    'max_depth': [2, 3]
}
xgb_grid = GridSearchCV(XGBRegressor(random_state=42, verbosity=0), param_grid=xgb_param_grid,
                        scoring='neg_mean_absolute_error', cv=kf, n_jobs=-1)
xgb_grid.fit(X, y)
xgb_model = xgb_grid.best_estimator_
with mlflow.start_run(run_name="XGBoost_Model"):
    mlflow.log_params({f"xgboost_{k}": v for k, v in xgb_grid.best_params_.items()})
    metrics = evaluate_model("xgboost", xgb_model, X, y, kf)
    mlflow.log_metrics(metrics)
    mlflow.sklearn.log_model(xgb_model, "xgboost_model")
joblib.dump(xgb_model, "xgboost_model.pkl")
```

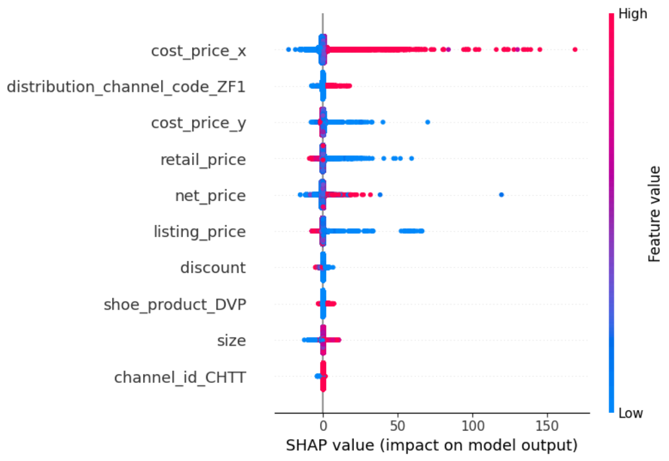
For XGBoost, we use a separate hyperparameter grid with slightly narrower bounds. The tuned parameters include n_estimators (100 and 200), learning_rate (0.05 and 0.1), and max_depth (2 and 3). These were chosen to balance performance and training time. XGBoost requires special handling of feature names, so our earlier column name sanitization step is crucial here. As with the other models, we run GridSearchCV and log all results to MLflow. The trained XGBoost model is also serialized with Joblib.



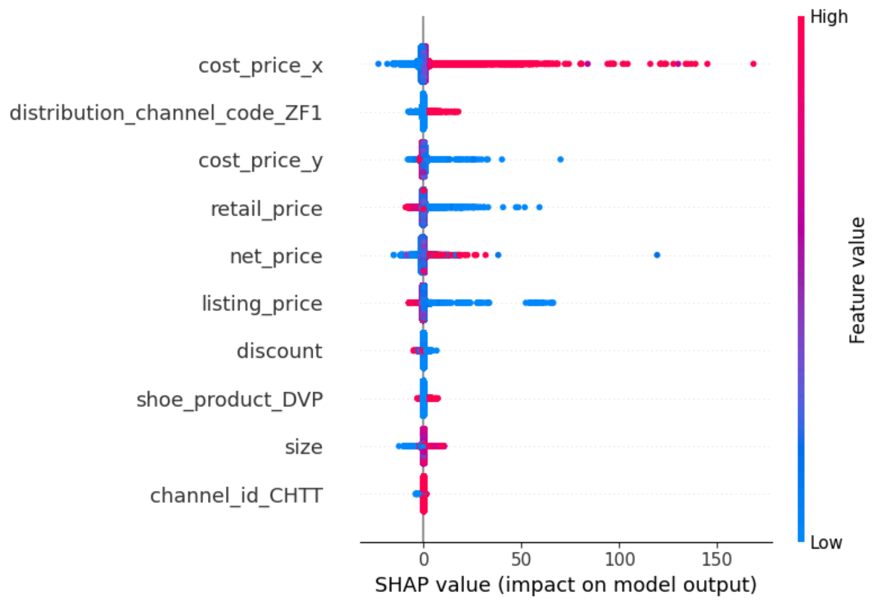
```
#SHAP Summary Plots
explainer_cb = shap.Explainer(cb_model); shap.summary_plot(explainer_cb(X), X, max_display=10)
explainer_lgb = shap.Explainer(lgb_model); shap.summary_plot(explainer_lgb(X), X, max_display=10)
explainer_xgb = shap.Explainer(xgb_model); shap.summary_plot(explainer_xgb(X), X, max_display=10)
```



CatBoost Model

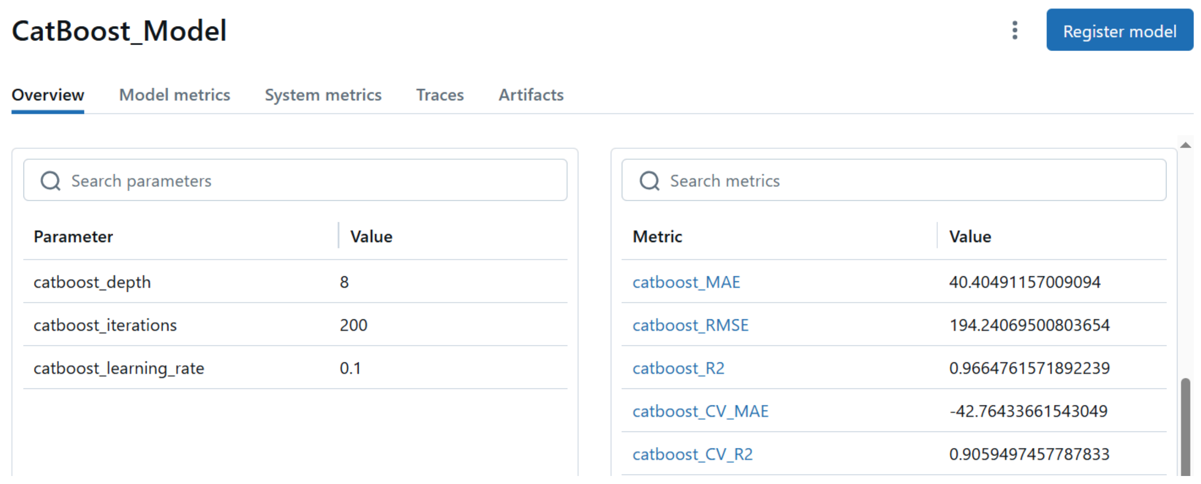


LightGBM



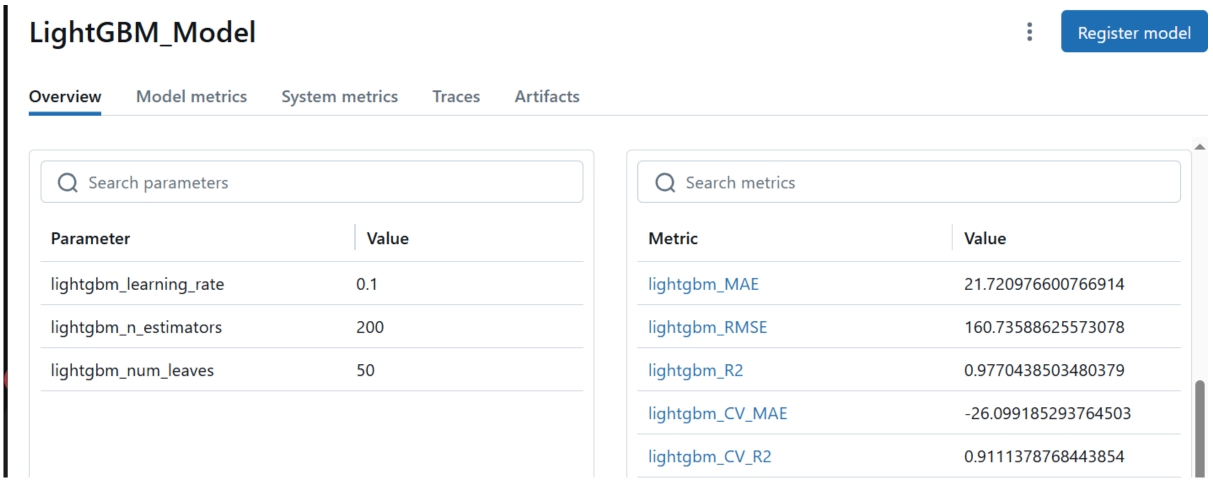
XGBoost

After training, we interpret the models using SHAP (SHapley Additive exPlanations). SHAP values help explain the contribution of each feature to individual predictions. This is particularly valuable in a business setting, where stakeholders need to understand not just predictions, but also why the model made them. We generate summary plots for all three models, which highlight that cost_price, distribution_channel_code_ZF1, retail_price, net_price, and listing_price are among the most impactful features. While all models emphasize similar variables, the exact ranking and influence vary, offering additional insight into each model's decision process.



Parameter	Value
catboost_depth	8
catboost_iterations	200
catboost_learning_rate	0.1

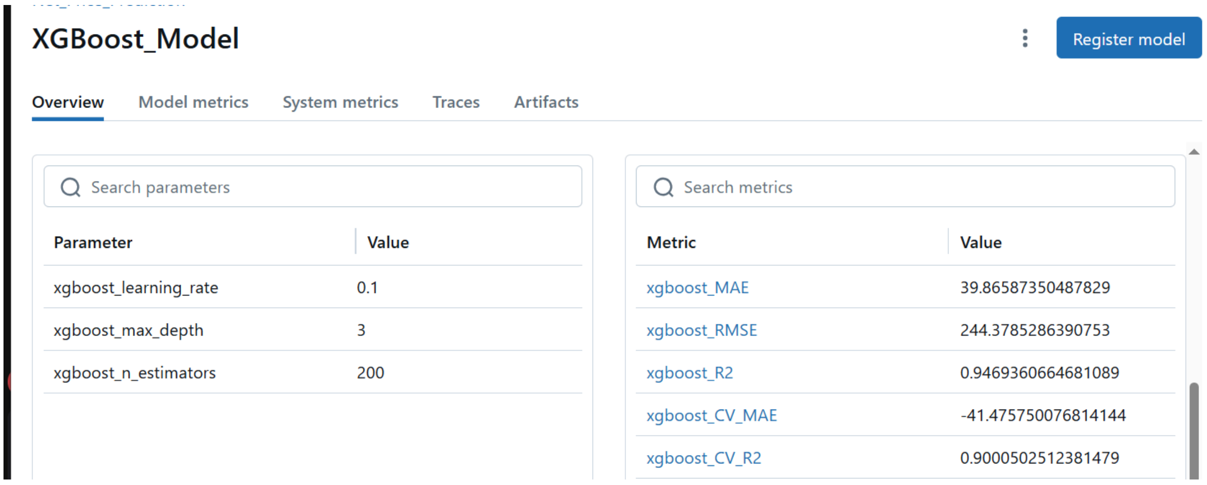
Metric	Value
catboost_MAE	40.40491157009094
catboost_RMSE	194.24069500803654
catboost_R2	0.9664761571892239
catboost_CV_MAE	-42.76433661543049
catboost_CV_R2	0.90594974577787833



LightGBM_Model

Parameter	Value
lightgbm_learning_rate	0.1
lightgbm_n_estimators	200
lightgbm_num_leaves	50

Metric	Value
lightgbm_MAE	21.720976600766914
lightgbm_RMSE	160.73588625573078
lightgbm_R2	0.9770438503480379
lightgbm_CV_MAE	-26.099185293764503
lightgbm_CV_R2	0.9111378768443854



XGBoost_Model

Parameter	Value
xgboost_learning_rate	0.1
xgboost_max_depth	3
xgboost_n_estimators	200

Metric	Value
xgboost_MAE	39.86587350487829
xgboost_RMSE	244.3785286390753
xgboost_R2	0.9469360664681089
xgboost_CV_MAE	-41.475750076814144
xgboost_CV_R2	0.9000502512381479

Finally, we consolidate model performance results using MLflow's UI. The dashboard shows that LightGBM achieves the best generalization performance, with the lowest MAE and highest cross-validated R². CatBoost and XGBoost follow closely, each exhibiting strong performance but slightly higher error margins. These results guide model selection and justify our configuration choices throughout the pipeline.

Deployment

AI Server Deployment

The ai-server, built using Python and FastAPI, is deployed as a Docker-based web service on Render. A multi-stage Dockerfile is used to handle dependency installation and application packaging. The first stage sets up the environment and installs packages specified in requirements.txt. The second stage copies the application code and installed dependencies into a lightweight runtime container and uses uvicorn as the entry point to serve the API.

Render automatically builds and deploys the container when changes are pushed to the designated branch. It exposes the server on port 8000, with health checks and secure environment variable management handled through the platform. Render's support for build caching, automatic rollbacks, and live logging ensures that the deployment remains stable, observable, and easily maintainable in production environments.

https://pelicanos.onrender.com/docs

Frontend Deployment

The frontend application, developed using React and TypeScript, is deployed via Vercel, a modern deployment platform optimized for frontend frameworks and static site hosting. Vercel is tightly integrated with the project's GitHub repository and supports automatic deployments triggered by changes to the main branch. On each deployment, Vercel runs the yarn build command to generate a production-ready build of the application, which is then served globally through Vercel's edge network for optimal performance and low latency.

Importantly, Vercel offers native compatibility with Turborepo, the build system used in this monorepo architecture. As both tools are developed by Vercel Inc., this integration is seamless. The frontend application resides in /apps/frontend and is part of a larger monorepo coordinated via a turbo.json configuration file. Vercel intelligently identifies affected projects within the monorepo and runs only the necessary builds, using Turborepo's incremental caching and dependency graph awareness to avoid redundant computation. Shared code under /packages is reused efficiently, and only modified parts of the codebase are rebuilt and deployed. This results in significantly reduced build times and improved deployment speed—particularly valuable in active development environments where frequent iterations are common.

Vercel further enhances the development workflow by supporting preview deployments for every pull request, automatic HTTPS, custom domains, and out-of-the-box support for client-side routing. These features ensure that the frontend remains performant, secure, and continuously deployable, while aligning with modern DevOps and frontend engineering practices.

CI/CD Pipeline

Frontend Pipeline

The frontend of the Pricing Optimizer project, built with React + Vite, is deployed via Vercel and utilizes a CI/CD pipeline based on Node.js 18 with Yarn 4.6.0, managed through Corepack. The pipeline is automatically triggered by changes to frontend code, shared packages, or workflow configurations. It follows a two-stage process: build and test, followed by deployment. During the build stage, the pipeline installs dependencies, runs ESLint for linting, performs TypeScript type checks, and builds the application. The deployment stage handles production builds and artifact storage.

To ensure quality and reliability, the pipeline integrates code linting, type checking, and build verification. Dependency security scanning is planned for future iterations. The system also stores build artifacts and test results for 7 days and integrates with Codecov for test coverage reporting, laying the groundwork for upcoming enhancements such as integration tests, staging deployment previews, and performance monitoring.

AI Server Pipeline

The backend, implemented with FastAPI (Python 3.10), is deployed via Render and includes a parallel CI/CD pipeline tailored for Python projects. This pipeline is triggered by any changes to the backend codebase, shared packages, or CI workflows. It focuses on automated linting, unit testing, and test coverage reporting to uphold code quality standards. Tools such as Black (code formatting), isort (import sorting), and pytest (with Codecov integration for coverage metrics) are employed to maintain consistency and enforce coding best practices.

The backend pipeline locks dependencies through requirements.txt, ensuring reproducibility and minimizing version-related errors. It is configured to run on commits to both main and dev branches, as well as on pull requests, providing comprehensive validation for all code changes. Future enhancements include performance benchmarks, automated staging deployments, security checks, and documentation generation. This setup ensures a robust backend delivery pipeline aligned with modern DevOps practices, enabling faster iterations and more reliable releases.

Preprocessing with Airflow and DAG

The preprocessing workflow in Airflow is a daily automated pipeline designed to process sales data for pricing optimization. The workflow consists of two sequential tasks that run every day at midnight, starting from January 1, 2024. The first task, check_new_data, is responsible for importing fresh sales data from Kaggle using the kagglehub library. It downloads the dataset and returns the path to the downloaded files. If any errors occur during this process, they are logged and the task fails, triggering a retry after a 5-minute delay.

Once the data is successfully imported, the second task, run_preprocessing, takes over. This task performs comprehensive data transformation and enrichment. It begins by reading sales data from Excel files and merges it with retail price data and Cost of Goods Sold (COGS) information. The pipeline then processes the product master data, applying various mappings for colors, product groups, size groups, and other attributes. A crucial step in the process is the conversion of prices from Vietnamese Dong (VND) to Thai Baht (THB) using a conversion rate of 0.0015. The workflow also calculates discounts by comparing retail prices with net prices, ensuring they are non-negative.

The processed data is then saved into five separate CSV files in the dataset/processed/ directory: sales_with_products.csv containing the main processed sales data, latest_retail.csv with the most recent retail prices, latest_cogs.csv with current cost of goods sold information, product_master.csv containing the processed product master data, and calendar_df.csv with time-based analysis data. The workflow uses Airflow's XCom feature to pass the dataset path between tasks and includes comprehensive error handling and logging throughout the process. This automated pipeline ensures that sales data is consistently processed and made available for pricing optimization analysis, with the entire process being monitored and logged for reliability and traceability.

Conclusion

Result and Discussions

This project successfully delivered a fully operational machine learning-based dynamic pricing optimization system, demonstrating the application of end-to-end MLOps practices in a real-world scenario. The team developed a robust LightGBM model capable of predicting product demand based on pricing, seasonal trends, and product attributes. After comparing multiple models, including CatBoost and XGBoost, LightGBM was selected for its superior performance in cross-validation and generalization metrics. The model was deployed as a containerized FastAPI service on Render, exposing a production-ready API capable of returning price recommendations aligned with specific business goals such as revenue maximization or sales volume. In parallel, an automated data preprocessing pipeline was implemented using Apache Airflow, ensuring that incoming sales data is cleaned, transformed, and made model-ready on a daily basis. A user-facing web application was also developed using React and TypeScript, deployed via Vercel, providing an intuitive interface for inputting product data and receiving optimized pricing suggestions. All services were integrated under a CI/CD workflow managed through GitHub Actions, supporting continuous testing, deployment, and artifact tracking. Model experiments and performance metrics were systematically logged via MLflow, and SHAP was used to generate feature importance plots for interpretability.

Problems and Solutions

During the development of the project, the team encountered two primary challenges. The first was the lack of localized and sufficient training data, which posed difficulties in ensuring the model's relevance to real-world scenarios in the target market. The dataset, originally sourced from Vietnamese retail operations, required significant preprocessing, including language normalization, currency conversion (VND to THB), and extensive feature engineering to make it applicable to a Thai business context. To address this, the team introduced a robust preprocessing pipeline and leveraged domain knowledge to enrich the dataset with meaningful features such as price ratios, discounts, and seasonal indicators.

The second major challenge involved task management and coordination across a multi-component system, which included data preprocessing, model training, API deployment, and frontend development. Initially, this led to inconsistencies and duplication of work. The team resolved this by adopting a modular monorepo structure supported by modern DevOps practices, including centralized version control, shared utility packages, and CI/CD pipelines. This structure ensured that each team member could work independently while maintaining system-wide consistency, significantly improving collaboration and deployment reliability.

Future Improvements

Looking ahead, several enhancements are planned to further increase the system's functionality, usability, and long-term performance. First, the team intends to implement advanced visualizations within the frontend interface to help users better understand pricing trends, demand forecasts, and the factors influencing price recommendations. This will support more informed decision-making and improve transparency. Second, the system will be extended to allow users to upload their own product data, enabling personalized pricing predictions tailored to specific inventory and business needs. This feature would make the platform more versatile and adaptable for various use cases. Lastly, to maintain model accuracy over time, a retraining pipeline will be implemented to automatically retrain the prediction model based on newly ingested sales data. This will ensure that the system adapts to market shifts and changing consumer behavior, keeping the recommendations relevant and effective in dynamic retail environments.