Project Documentation: An End-to-End System for Predicting Customer Purchase Propensity

Author: Himanshu Raj

Date: June 14, 2025

Project GitHub Repository: https://github.com/magnetbrains-bit/end-to-end-ml-pipeline-retail

Live API Documentation: https://propensity-api-himanshu.onrender.com/docs

---

Table of Contents

---

1. Introduction

1.1. Project Motivation

The field of machine learning is rapidly evolving from a purely analytical discipline to an engineering one. The ability to simply build an accurate model is no longer sufficient; the true value lies in operationalizing these models into robust, scalable, and automated systems. This project was conceived as a comprehensive exercise to bridge that gap, demonstrating the full lifecycle of a machine learning product—from raw data exploration to a live, deployed API on the cloud.

1.2. Business Problem & Objectives

In the competitive e-commerce landscape, untargeted marketing is inefficient and costly. This project addresses the core business problem of **optimizing customer engagement and marketing spend**.

The primary objective was to build a system that could predict a user's **propensity to make a purchase within the next 7 days**. This intelligence allows a business to:

- **Target** high-propensity users with personalized offers.
- **Reduce** ad spend by not targeting low-propensity users.
- **Increase** overall conversion rates and revenue.

1.3. Scope and Limitations

- **In Scope:** The project covers data processing, feature engineering, iterative model training, API creation, containerization, and cloud deployment.
- **Out of Scope:** Full CI/CD pipelines with automated testing, real-time feature stores, and advanced model monitoring dashboards are designated as future work. The model is pre-trained and "baked-in" to the deployment rather than being retrained live.

2. Technology Stack & Development Environment

2.1. Core Technologies and Libraries

A curated set of modern, industry-standard tools was chosen for this project.

Category	Technology / Library	Purpose
Data Science & Modeling	Python 3.11, Pandas, NumPy	Core language and libraries for data manipulation and numerical operations.
	Scikit-learn	Used for data splitting and robust model evaluation metrics.
	XGBoost	High-performance gradient boosting algorithm for the predictive model.
API & Backend	FastAPI	A modern, high-speed Python framework for building REST APIs.
	Uvicorn	The ASGI server used to run the FastAPI application.
Infrastructure & Deployment	Docker	Platform for containerizing the application, ensuring portability.
	Render	Cloud platform for deploying the containerized web service.
Development & Version Control	Google Colab, Jupyter, Visual Studio Code, Git, GitHub	Tools for exploration, development, and version control.

2.2. Environment Setup

The project was developed using a hybrid approach:

- **Google Colab:** Initial EDA and model prototyping were performed here to leverage the high-RAM environment for handling the large dataset.
- **Local Development:** A local Python virtual environment was used for developing the API, scripts, and Docker container. All required packages are listed in requirements.txt.

3. Data Sourcing & Exploratory Data Analysis (EDA)

3.1. Dataset Overview

The project uses the Retail Rocket E-commerce Dataset. It contains over 2.7 million timestamped user interaction events (view, addtocart, transaction) from a 4.5-month period, providing a rich source for behavioral analysis.

3.2. Initial Data Loading & Cleaning

The primary events.csv file was loaded using Pandas. The timestamp column was converted from milliseconds to a standard datetime format. To manage memory usage on a local machine, data types were optimized during the CSV read process.

3.3. Key Findings from EDA

- **Severe Class Imbalance:** A value count of the event column revealed that transactions are extremely rare compared to views, making up less than 1% of the data. This confirmed that accuracy would be a poor evaluation metric and that metrics like Precision, Recall, and AUC would be critical.
  [Image: A bar chart showing the distribution of 'view', 'addtocart', and 'transaction' events]
- **User Activity Patterns:** Plotting the number of events over time showed clear weekly seasonality, with activity dipping on weekends. This suggests that time-based features (e.g., day of the week) could be valuable additions in future iterations.
  [Image: A line chart showing total events per day over the 4.5-month period]

4. Feature Engineering: Crafting Predictive Signals

4.1. Rationale for Time-Windowed Features

The core hypothesis was that a user's recent behavior is more predictive of their future actions than their entire history. To capture this, I engineered a set of **time-windowed features**, which calculate user activity over multiple lookback periods.

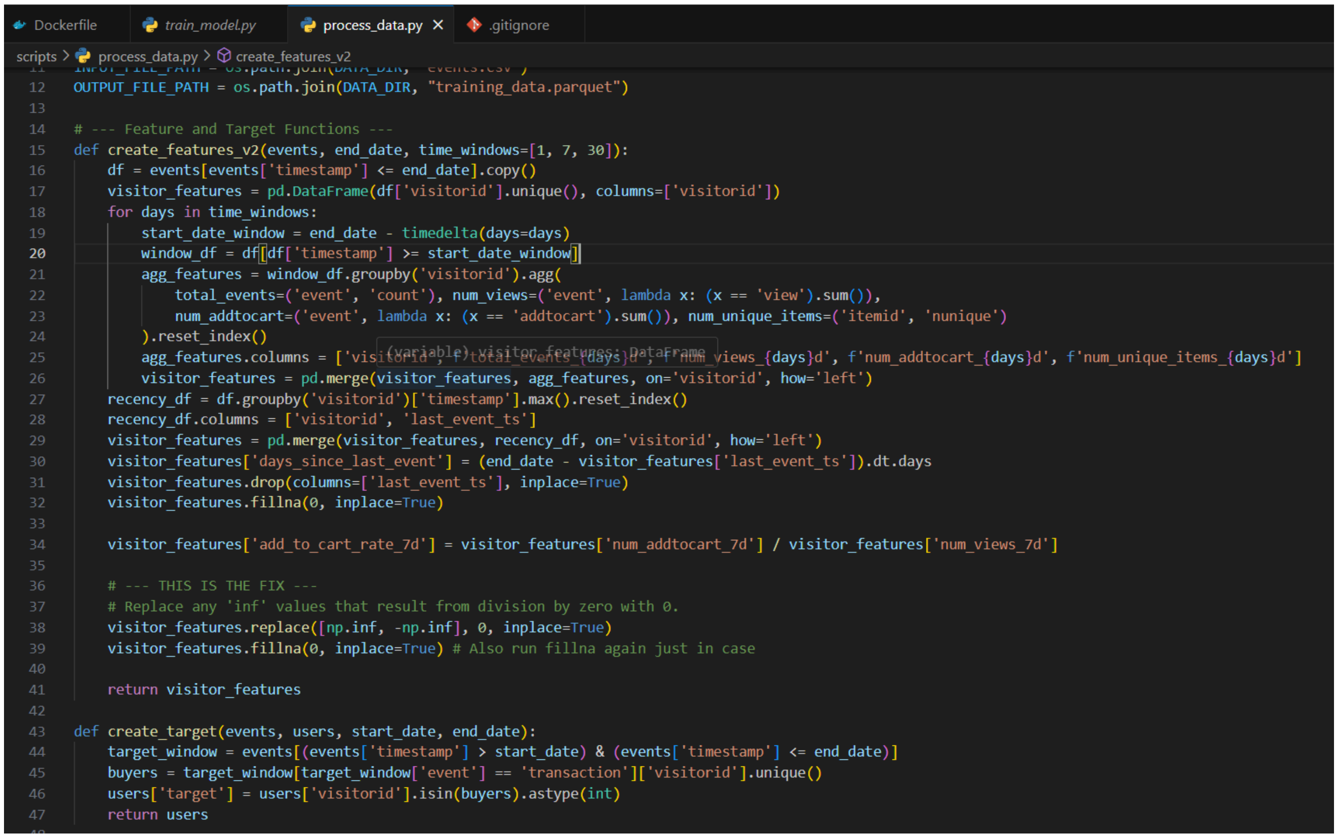
4.2. Detailed Feature Descriptions

For each user, the following features were calculated for the last 1, 7, and 30 days:

- num_views_{n}d: Total items viewed. Measures general interest and browsing activity.
- num_addtocart_{n}d: Total "add to cart" events. A very strong signal of purchase intent.
- num_unique_items_{n}d: The variety of items a user interacts with.
- days_since_last_event: Captures user recency. A user active yesterday is more valuable than one last seen a month ago.
- add_to_cart_rate_7d: The ratio of addtocart to view events. This powerful feature measures the user's conversion from browsing to consideration.

4.3. Implementation in Python

This logic was encapsulated in a reusable function within scripts/process_data.py. A critical part of this function was handling potential inf values that arose from division-by-zero errors.



```python
OUTPUT_FILE_PATH = os.path.join(DATA_DIR, "training_data.parquet")

# --- Feature and Target Functions ---
def create_features_v2(events, end_date, time_windows=[1, 7, 30]):
    df = events[events['timestamp'] <= end_date].copy()
    visitor_features = pd.DataFrame(df['visitorid'].unique(), columns=['visitorid'])
    for days in time_windows:
        start_date_window = end_date - timedelta(days=days)
        window_df = df[df['timestamp'] >= start_date_window]
        agg_features = window_df.groupby('visitorid').agg(
            total_events=('event', 'count'), num_views=('event', lambda x: (x == 'view').sum()),
            num_addtocart=('event', lambda x: (x == 'addtocart').sum()), num_unique_items=('itemid', 'nunique')
        ).reset_index()
        agg_features.columns = ['visitorid', f'total_events_{days}d', f'num_views_{days}d', f'num_addtocart_{days}d', f'num_unique_items_{days}d']
        visitor_features = pd.merge(visitor_features, agg_features, on='visitorid', how='left')
    recency_df = df.groupby('visitorid')['timestamp'].max().reset_index()
    recency_df.columns = ['visitorid', 'last_event_ts']
    visitor_features = pd.merge(visitor_features, recency_df, on='visitorid', how='left')
    visitor_features['days_since_last_event'] = (end_date - visitor_features['last_event_ts']).dt.days
    visitor_features.drop(columns=['last_event_ts'], inplace=True)
    visitor_features.fillna(0, inplace=True)

    visitor_features['add_to_cart_rate_7d'] = visitor_features['num_addtocart_7d'] / visitor_features['num_views_7d']

    # --- THIS IS THE FIX ---
    # Replace any 'inf' values that result from division by zero with 0.
    visitor_features.replace([np.inf, -np.inf], 0, inplace=True)
    visitor_features.fillna(0, inplace=True) # Also run fillna again just in case

    return visitor_features

def create_target(events, users, start_date, end_date):
    target_window = events[(events['timestamp'] > start_date) & (events['timestamp'] <= end_date)]
    buyers = target_window[target_window['event'] == 'transaction']['visitorid'].unique()
    users['target'] = users['visitorid'].isin(buyers).astype(int)
    return users
```

5. Model Development & Iterative Improvement

5.1. Model Selection: Why XGBoost?

The **XGBoost (Extreme Gradient Boosting)** algorithm was chosen for its state-of-the-art performance on tabular data. It is an ensemble method that builds a sequence of decision trees, where each new tree corrects the errors of its predecessors. It is known for its speed, regularization features to prevent overfitting, and its ability to handle missing data.

5.2. The V1 Baseline Model: An Experiment in Failure

The initial model was trained on simple, globally aggregated features. This served as a baseline to measure future improvements against.

- **Result:** The model was completely unusable. While it had a decent Recall, its **Precision was only 0.03%**. This meant that for every 10,000 users it flagged as "likely buyers," only 3 would actually make a purchase.

5.3. The V2 High-Performance Model: An Iterative Success

After diagnosing V1's failure, I engineered the V2 solution:

1. **Advanced Features:** Implemented the time-windowed features described in Chapter 4.
2. **Optimal Thresholding:** Used the Precision-Recall curve to identify the prediction threshold that maximized the F1-score, correctly handling the class imbalance.

5.4. Model Evaluation Strategy

A crucial part of the evaluation strategy was the **chronological train-test split**. Data was split strictly by time to simulate a real-world scenario where a model trained on past data is used to predict future outcomes. This prevents data leakage and provides a much more realistic estimate of the model's performance than a random split.

Given the imbalanced dataset, the following metrics were used:

- **ROC AUC:** To measure the model's overall ability to discriminate between the positive and negative classes.
- **Precision:** To measure how accurate the model's positive predictions are (True Positives / (True Positives + False Positives)). Crucial for business cost.
- **Recall:** To measure how many of the actual positive cases the model found (True Positives / (True Positives + False Negatives)).
- **F1-Score:** The harmonic mean of Precision and Recall, providing a single score that balances both.

5.5. Final Performance Results & Analysis

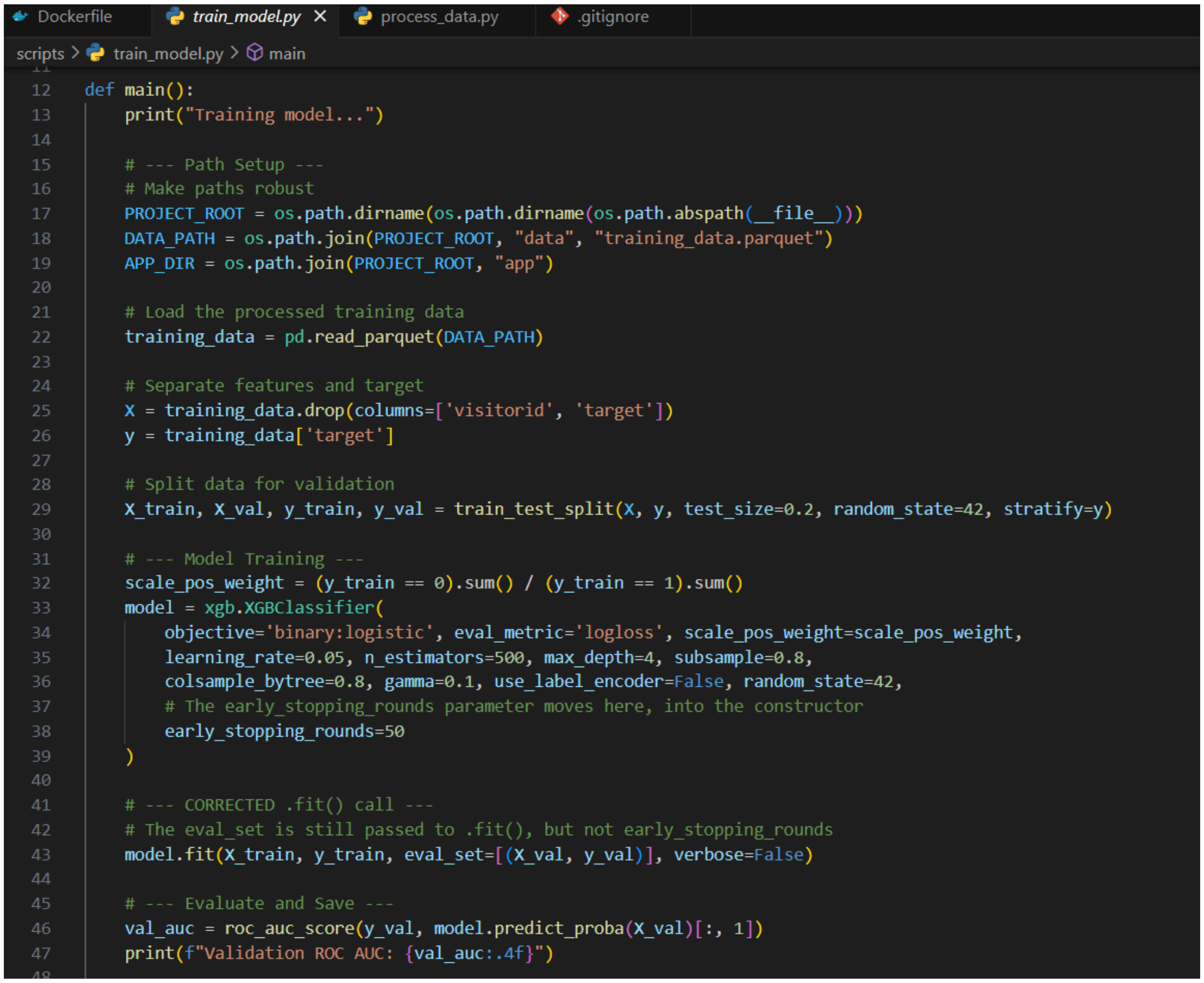
The iterative process yielded a dramatic and meaningful improvement.

[Image: The V1 vs. V2 results table graphic we created]

Metric	Model V1 (Baseline)	Model V2 (Final)	Business Impact
Precision	0.03%	**59.3%**	Predictions are now highly reliable and actionable.
F1 Score	0.0006	**0.203**	The overall model quality is orders of magnitude better.

ROC AUC	0.606	**0.785**	Much better at distinguishing buyers from non-buyers.

5.6. Code Implementation

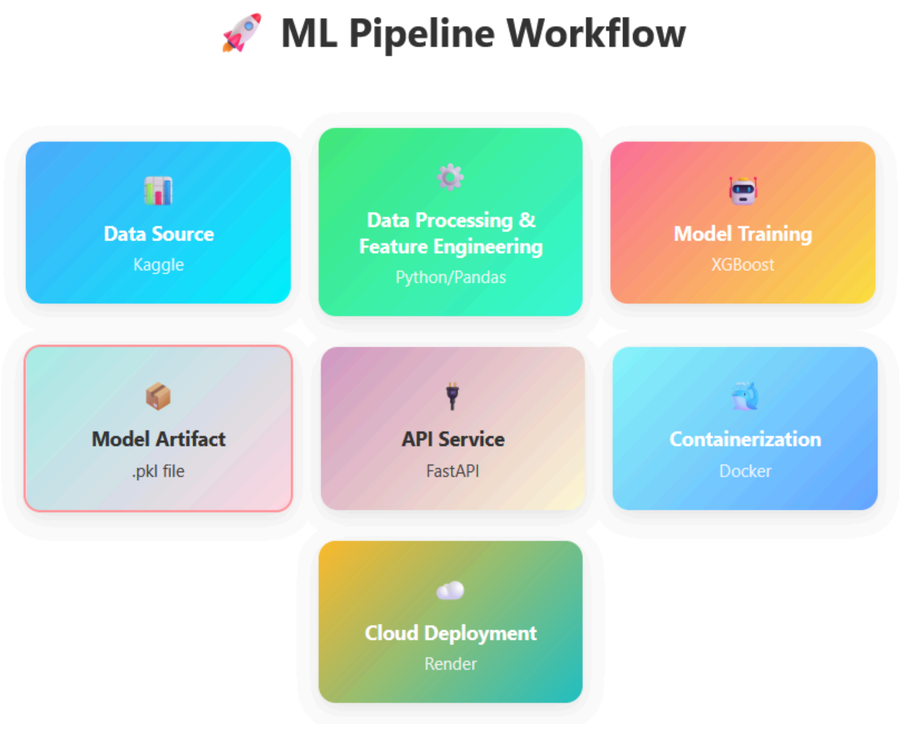


```python
def main():
    print("Training model...")

    # --- Path Setup ---
    # Make paths robust
    PROJECT_ROOT = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    DATA_PATH = os.path.join(PROJECT_ROOT, "data", "training_data.parquet")
    APP_DIR = os.path.join(PROJECT_ROOT, "app")

    # Load the processed training data
    training_data = pd.read_parquet(DATA_PATH)

    # Separate features and target
    X = training_data.drop(columns=['visitorid', 'target'])
    y = training_data['target']

    # Split data for validation
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

    # --- Model Training ---
    scale_pos_weight = (y_train == 0).sum() / (y_train == 1).sum()
    model = xgb.XGBClassifier(
        objective='binary:logistic', eval_metric='logloss', scale_pos_weight=scale_pos_weight,
        learning_rate=0.05, n_estimators=500, max_depth=4, subsample=0.8,
        colsample_bytree=0.8, gamma=0.1, use_label_encoder=False, random_state=42,
        # The early_stopping_rounds parameter moves here, into the constructor
        early_stopping_rounds=50
    )

    # --- CORRECTED .fit() call ---
    # The eval_set is still passed to .fit(), but not early_stopping_rounds
    model.fit(X_train, y_train, eval_set=[(X_val, y_val)], verbose=False)

    # --- Evaluate and Save ---
    val_auc = roc_auc_score(y_val, model.predict_proba(X_val)[:, 1])
    print(f"Validation ROC AUC: {val_auc:.4f}")
```

6. System Architecture & Deployment (MLOps)

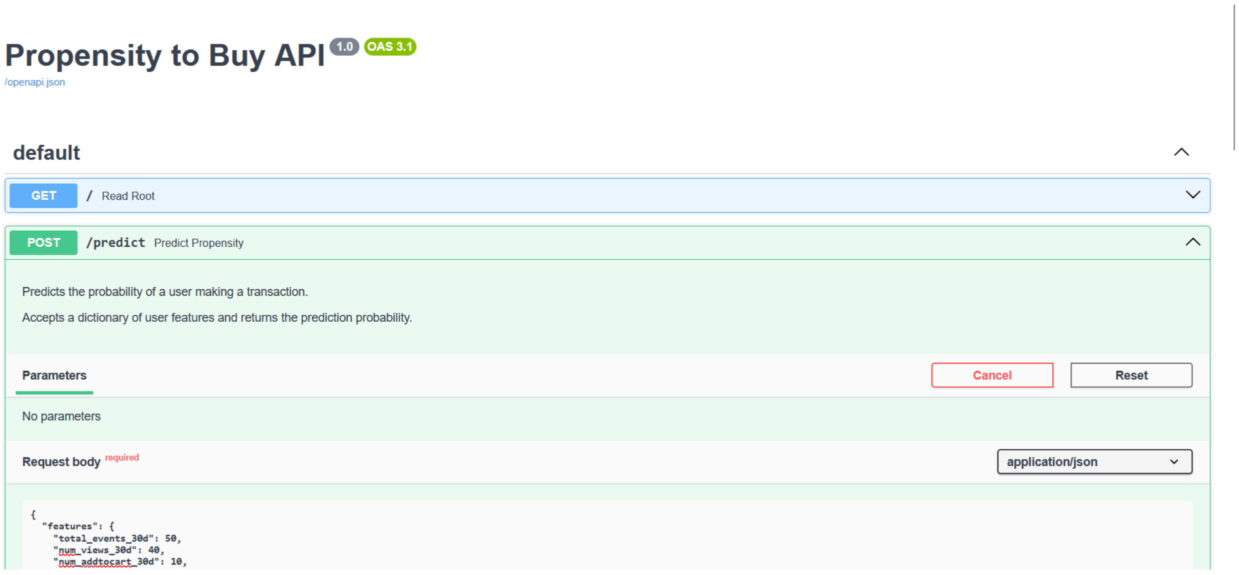
6.1. System Overview Diagram

The final system follows a modern MLOps architecture, taking a model from a static artifact to a live, queryable service.



6.2. The REST API Service (FastAPI)

The trained V2 model is wrapped in a REST API using FastAPI. It exposes a /predict endpoint that accepts a JSON payload of user features and returns a real-time propensity score.



Propensity to Buy API 1.0 OAS 3.1

/openapi.json

default

GET / Read Root

POST /predict Predict Propensity

Predicts the probability of a user making a transaction.

Accepts a dictionary of user features and returns the prediction probability.

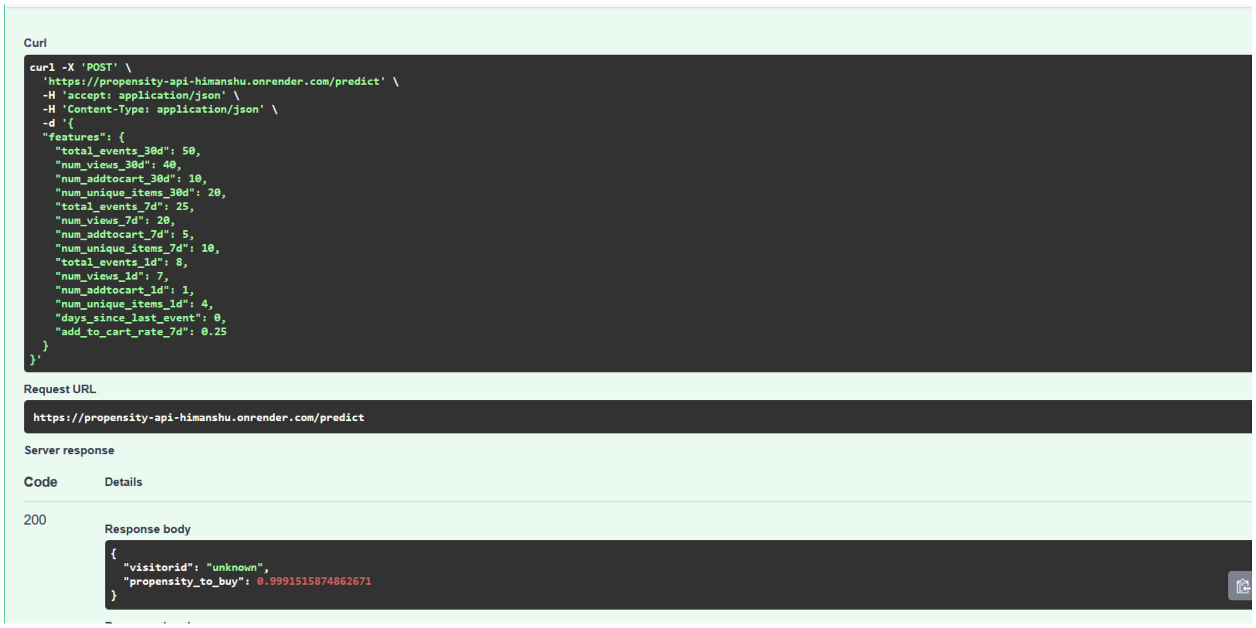
Parameters

Cancel Reset

No parameters

Request body required application/json

```
{
  "features": {
    "total_events_30d": 50,
    "num_views_30d": 40,
    "num_addtocart_30d": 10,
```



Curl

```
curl -X 'POST' \
  'https://propensity-api-himanshu.onrender.com/predict' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "features": {
    "total_events_30d": 50,
    "num_views_30d": 40,
    "num_addtocart_30d": 10,
    "num_unique_items_30d": 20,
    "total_events_7d": 25,
    "num_views_7d": 20,
    "num_addtocart_7d": 5,
    "num_unique_items_7d": 10,
    "total_events_1d": 8,
    "num_views_1d": 7,
    "num_addtocart_1d": 1,
    "num_unique_items_1d": 4,
    "days_since_last_event": 0,
    "add_to_cart_rate_7d": 0.25
  }
}'
```

Request URL

```
https://propensity-api-himanshu.onrender.com/predict
```

Server response

Code Details

200

Response body

```
{
  "visitorid": "unknown",
  "propensity_to_buy": 0.9991515874862671
}
```

6.3. Containerization (Docker)

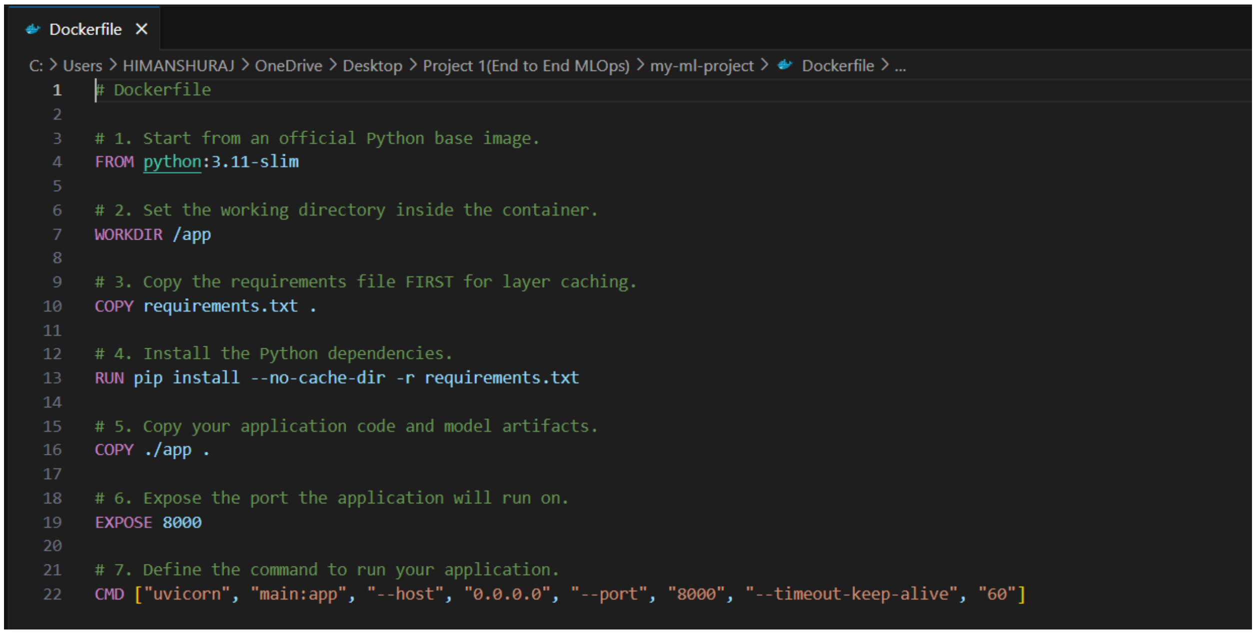
The entire application is containerized using Docker. The Dockerfile below defines the reproducible environment.

# Dockerfile

# 1. Start from an official Python base image.
FROM python:3.11-slim

# 2. Set the working directory inside the container.

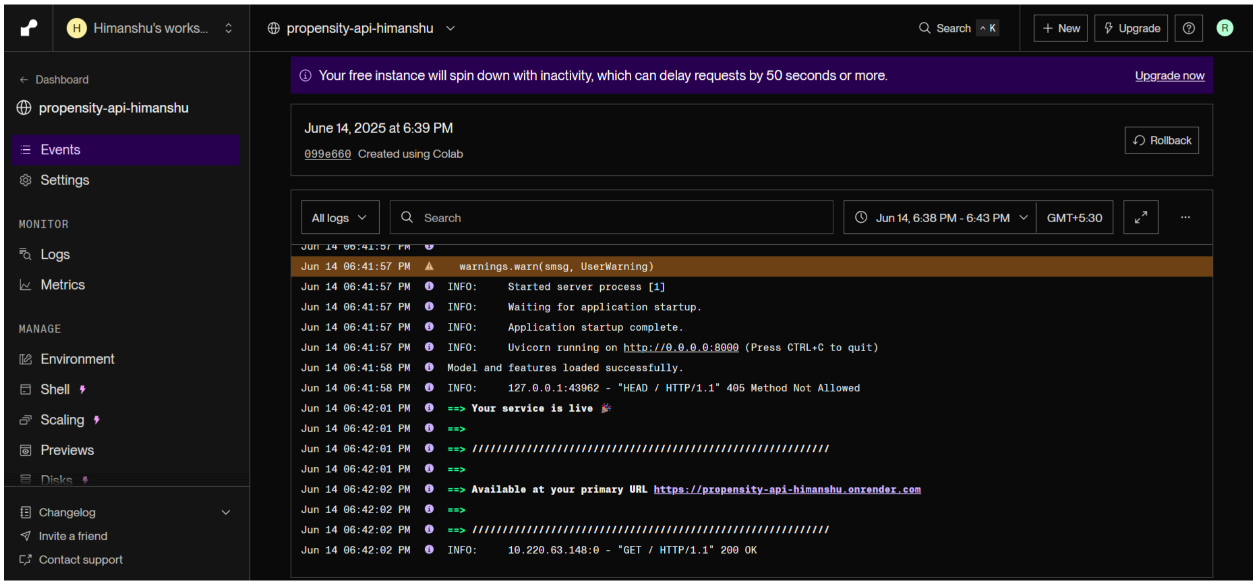
```
WORKDIR /app

# 3. Copy the requirements file FIRST for layer caching.
COPY requirements.txt .

# 4. Install the Python dependencies.
RUN pip install --no-cache-dir -r requirements.txt

# 5. Copy your application code and model artifacts.
COPY ./app .

# 6. Expose the port the application will run on.
EXPOSE 8000

# 7. Define the command to run your application.
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```


Dockerfile
C: > Users > HIMANSHURAJ > OneDrive > Desktop > Project 1(End to End MLOps) > my-ml-project > Dockerfile > ...

```
# Dockerfile

# 1. Start from an official Python base image.
FROM python:3.11-slim

# 2. Set the working directory inside the container.
WORKDIR /app

# 3. Copy the requirements file FIRST for layer caching.
COPY requirements.txt .

# 4. Install the Python dependencies.
RUN pip install --no-cache-dir -r requirements.txt

# 5. Copy your application code and model artifacts.
COPY ./app .

# 6. Expose the port the application will run on.
EXPOSE 8000

# 7. Define the command to run your application.
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000", "--timeout-keep-alive", "60"]
```

6.4. Cloud Deployment (Render)

The Docker image is deployed as a live web service on Render. The deployment is linked to the project's GitHub repository for a Continuous Deployment workflow.



```
Jun 14 06:41:57 PM     warnings.warn(smsg, UserWarning)
Jun 14 06:41:57 PM   INFO:     Started server process [1]
Jun 14 06:41:57 PM   INFO:     Waiting for application startup.
Jun 14 06:41:57 PM   INFO:     Application startup complete.
Jun 14 06:41:57 PM   INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
Jun 14 06:41:58 PM   Model and features loaded successfully.
Jun 14 06:41:58 PM   INFO:     127.0.0.1:43962 - "HEAD / HTTP/1.1" 405 Method Not Allowed
Jun 14 06:42:01 PM   ==> Your service is live 🎉
Jun 14 06:42:01 PM   ==>
Jun 14 06:42:01 PM   ==> ///////////////////////////////////////////////////////////
Jun 14 06:42:01 PM   ==>
Jun 14 06:42:02 PM   ==> Available at your primary URL https://propensity-api-himanshu.onrender.com
Jun 14 06:42:02 PM   ==>
Jun 14 06:42:02 PM   ==> ///////////////////////////////////////////////////////////
Jun 14 06:42:02 PM   INFO:     10.220.63.148:0 - "GET / HTTP/1.1" 200 OK
```

6.5. Data and Model Artifact Flow

- **Training Time:** "During the training workflow, the process_data.py script reads the raw events.csv from the /data directory and writes its output, training_data.parquet, back to the same directory. The train_model.py script then reads this parquet file and writes the final model artifacts (.pkl, .json) to the /app directory, ready to be included in the Docker image."
- **Inference Time (Live API):** "When the Docker container starts on Render, the FastAPI application loads the model artifacts from its local /app directory into memory. When a request hits the /predict endpoint, the input JSON data is transformed into a pandas DataFrame *in memory*, scored by the model, and the prediction is returned. No disk I/O is required for a single prediction, ensuring low latency."

6.6. Scalability Considerations

- **Stateless API:** "The prediction API is **stateless**, meaning it does not store any information about past requests. Each prediction is independent. This is a critical design principle that allows for easy horizontal scaling. If traffic increases, we can simply spin up multiple instances of the Docker container behind a load balancer, and any container can handle any request."
- **Render's Autoscaling:** "The deployment platform, Render, offers autoscaling features. For a production workload, the service could be configured to automatically add more container instances as CPU or memory usage crosses a certain threshold, and scale back down when traffic subsides, ensuring both performance and cost-efficiency."

7. Guide to Reproducing the Project

Instructions to replicate the project environment and run the API locally.

1. **Prerequisites:** Git, Python 3.11+, Docker Desktop.
2. **Clone & Setup:**

```
git clone https://github.com/magnetbrains-bit/end-to-end-ml-pipeline-retail.git
cd end-to-end-ml-pipeline-retail
python -m venv venv
venv\Scripts\activate
pip install -r requirements.txt
```

3. **Download Data:** Place events.csv from the Kaggle dataset into the data/ folder.
4. **Run Training:**

```
python scripts/process_data.py
python scripts/train_model.py
```

5. **Build & Run Docker Container:**

```
docker build -t propensity-api .
docker run -p 8000:8000 propensity-api
```

6. **Access API: Navigate to http://127.0.0.1:8000/docs in a web browser**

8. Challenges Faced & Key Learnings

This project provided significant practical experience in debugging real-world engineering issues:

- **Environment Pathing:** Solved ModuleNotFoundError in subprocesses by using an explicit path to the virtual environment's Python executable.
- **Data Quality:** Addressed model crashes caused by infinity values in the feature set by implementing a robust data cleaning step.
- **Library Versioning:** Debugged an XGBoostError related to API changes between library versions (early_stopping_rounds).
- **Docker File System:** Troubleshooted a Dockerfile: is a directory error, reinforcing the importance of file system fundamentals.

9. Conclusion & Future Work

This project successfully demonstrates the creation of an end-to-end machine learning service. The final V2 model is highly predictive and provides actionable business intelligence, served via a robust, deployed API.

Potential future improvements include:

1. Full-Scale Automation & Orchestration

The current training process is manual, requiring a developer to run local scripts. The next critical step is to automate this entirely.

- **What:** Implement a workflow orchestration tool like **Prefect** or use **GitHub Actions** to create a scheduled retraining pipeline.
- **Why:** This is the core of MLOps. An automated pipeline ensures that the model can be retrained on new data (e.g., every week) without any human intervention. This keeps the model from becoming stale and its predictions inaccurate as user behavior changes over time.
- **Implementation Details:**
    - The pipeline would be triggered on a schedule (e.g., every Sunday at 2 AM).
    - It would automatically run scripts/process_data.py to generate features from the latest data batch.
    - It would then run scripts/train_model.py to produce a new "candidate" model.
    - **Conditional Deployment:** The pipeline would compare the new model's validation AUC (from metrics.json) against the metric of the currently deployed model. The new model would only be "promoted" (e.g., saved to a specific "production" location) if its performance is significantly better, preventing the deployment of a weaker model.

---

2. Comprehensive Model & Data Monitoring

A deployed model operating in the wild is a "black box" unless it's being monitored. Implementing a monitoring system is crucial for reliability and trust.

- **What:** Build a monitoring dashboard to track model performance, data drift, and concept drift.
- **Why:** To answer critical questions like: "Is the model's accuracy degrading over time?" (concept drift), "Are the incoming user features different from what the model was trained on?" (data drift), and "Is the API service healthy?"
- **Implementation Details:**

    - **Logging:** Modify the FastAPI application to log every incoming prediction request and the model's output to a database (e.g., a simple SQLite DB or a more robust solution like PostgreSQL).
    - **Dashboard:** Create a dashboard using **Streamlit** or **Dash**. This dashboard would read from the log database and visualize:
        - *Operational Metrics:* API latency, request volume, error rates.
        - *Data Drift:* Distributions of key input features over time. A sudden change in days_since_last_event could signal a problem.
        - *Model Performance:* Periodically, once actual outcomes are known, calculate and plot the model's Precision, Recall, and AUC over time to detect degradation.
    - **Alerting:** Set up automated alerts (e.g., email or Slack notifications) if a key metric drops below a certain threshold.

---

3. CI/CD and Professional Software Practices

Continuous Integration (CI) and Continuous Deployment (CD) are practices that enforce code quality and automate the deployment process safely.

- **What:** Implement a CI/CD pipeline using **GitHub Actions**.
- **Why:** To ensure that any new code pushed to the repository is high-quality, bug-free, and doesn't break the existing application. This de-risks the development process.
- **Implementation Details:**
    - **Continuous Integration (CI):** Create a GitHub Actions workflow (.github/workflows/ci.yml) that triggers on every pull request. This workflow would automatically:
        1. **Run a Linter (flake8):** To check for Python style guide violations.
        2. **Run Unit Tests (pytest):** Write tests for critical functions (e.g., does create_features_v2 handle edge cases correctly?). The pull request would be blocked from merging if any tests fail.
    - **Continuous Deployment (CD):** Enhance the existing Render deployment. The workflow would be configured so that only code that is successfully merged into the main branch (after passing all CI checks) will trigger an automatic redeployment on Render.

---

4. Advanced Modeling & Feature Engineering

While the V2 model performs well, there are always opportunities for further accuracy gains.

- **What:** Incorporate more data sources and explore more complex modeling techniques.
- **Why:** To potentially capture more nuanced user behavior and squeeze out extra performance, leading to better business outcomes.

- **Implementation Details:**
    - **Incorporate Item Features:** Fully utilize the item_properties.csv and category_tree.csv files. This would involve complex joins and creating features like "user's preferred category" or "number of items viewed with property X."
    - **Advanced Hyperparameter Tuning:** Instead of pre-selected parameters, use a sophisticated optimization library like **Optuna** to systematically search for the absolute best set of hyperparameters for the XGBoost model.
    - **Explore Sequential Models:** For a more advanced approach, use a deep learning model like an **LSTM (Long Short-Term Memory network)**. An LSTM can understand the *sequence* of user events (view -> addtocart -> view), which might capture intent better than the current "bag-of-events" features. This would represent a significant leap in modeling complexity and capability.

10. References

- Retail Rocket E-commerce Dataset
- XGBoost Documentation
- FastAPI Documentation
- Docker Documentation
- Render Documentation
- A Guide to the MLOps Lifecycle
- Handling Imbalanced Datasets in Machine Learning
- Building and Serving a Scikit-learn Model with FastAPI
- Introduction to Docker for Data Scientists