**EMR Alert System for Drug A Eligibility**

**Overview**

This project develops a machine learning solution to identify patients eligible for "Drug A," an oral antiviral for Disease X, but who are unlikely to be treated. The final product is a REST API that receives patient data and returns a prediction, designed to power an alert system within an Electronic Medical Record (EMR) environment.

The primary goal is to help clinicians make timely treatment decisions by flagging at-risk patients who might otherwise be overlooked.

**Project Architecture**

The solution is structured as a modular Python application designed for clarity, scalability, and easy deployment.

* **/data/**: (Optional) Intended for storing raw input data like .xlsx or .csv files.
* **/notebooks/**: Contains the 1-data-preparation-and-modeling.ipynb Jupyter Notebook, which documents the entire exploratory data analysis, feature engineering, and model training process.
* **/models/**: Stores the final, trained model artifacts.
  + drug\_a\_predictor.joblib: The serialized, trained LightGBM classifier.
  + model\_columns.json: A list of the exact feature columns the model was trained on, ensuring consistency during prediction.
* **/src/**: The source code for the prediction API.
  + main.py: The FastAPI application that loads the model and exposes the /predict endpoint.
* ReadData.py: The master Python script that contains the entire pipeline, from data loading to model evaluation and saving.
* test\_api.py: A client script to test the running API programmatically.
* requirements.txt: A list of all Python dependencies required to run the project.
* Dockerfile: A definition file to containerize the application for easy deployment.

**Development Journey**

This solution was built incrementally, following a standard machine learning project lifecycle.

1. **Data Ingestion & Cleaning**: We started by loading three data sources (fact\_txn, dim\_patient, dim\_physician) from an Excel file. To ensure data quality, we performed several cleaning steps:
   * Standardized all column names and categorical data to lowercase to prevent case-sensitivity errors.
   * Converted date columns to the proper datetime format.
   * Merged the three tables into a single, unified DataFrame.
2. **Feature Engineering**: This was the most critical phase. We transformed the transaction-level data into a patient-level dataset, where each row represents one unique patient. Key features created include:
   * diseasex\_dt: The patient's diagnosis date, assumed to be their earliest transaction date.
   * patient\_age: The patient's age at the time of diagnosis.
   * num\_conditions: A count of the unique high-risk conditions for each patient.
   * **Custom Feature 1 (num\_contraindications)**: A count of medications or conditions that might prevent a patient from receiving Drug A.
   * **Custom Feature 2 (patient\_is\_high\_risk)**: A binary flag that directly encodes the business rule for high-risk patients (age >= 65 or num\_conditions > 0).
   * target: The binary target variable, set to 1 if the patient received Drug A and 0 otherwise.
3. **Model Training & Evaluation**:
   * **Model Selection**: We chose a **LightGBM Classifier**, as it is a powerful and efficient gradient-boosting model well-suited for tabular data.
   * **Filtering**: Based on the project requirements, we filtered the dataset to include only patients **aged 12 and above**.
   * **Preprocessing**: Categorical features were one-hot encoded to be used in the model. We also implemented a step to sanitize feature names to prevent errors during training.
   * **Evaluation**: The model achieved **96% recall** for the "Not Treated" class. This was our primary success metric, as it proves the model is highly effective at identifying the target patient population for the EMR alert.
4. **API Implementation**:
   * We used **FastAPI** to build a robust and fast API.
   * A /predict endpoint was created that accepts patient data via a JSON request.
   * **Pydantic** was used to define a strict data model for the input, ensuring data validation.
   * The API loads the saved model and column list at startup, processes the input data to match the model's expected format, and returns a prediction.

**Getting Started**

Follow these instructions to set up and run the project locally.

**Prerequisites**

* Python 3.8+
* pip (Python package installer)

**Installation & Setup**

1. **Clone the repository:**
2. Bash
3. git clone <your-repository-url>
4. cd drug-a-predictor
5. **Install dependencies:**

Bash

pip install -r requirements.txt

1. **Place your data:** Ensure your Inputdata.xlsx file is in the root project directory.

**Usage**

**1. Data Preparation & Model Training**

To run the entire data processing and model training pipeline from scratch, execute the main script:

Bash

python ReadData.py

This will read the data, perform all cleaning and feature engineering, train the model, and save the artifacts to the /models directory.

**2. Running the API Server**

To start the prediction server, run the following command from the root directory:

Bash

uvicorn src.main:app --reload

The API will now be running at http://127.0.0.1:8000.

**3. Testing the API**

You can test the API in two ways:

* **Interactive Docs (Recommended)**: Open your browser and navigate to http://127.0.0.1:8000/docs. You can execute predictions directly from this user interface.
* **Client Script**: In a new terminal (while the server is running), execute the test script:

Bash

python test\_api.py

**Containerization with Docker**

To build and run this application as a Docker container for consistent deployment:

1. **Build the Docker image:**

Bash

docker build -t drug-predictor-api .

1. **Run the container:**

Bash

docker run -d -p 8000:8000 --name drug-api drug-predictor-api

The API will be accessible at http://localhost:8000.