Amazon Sales Analytics – Data Processing & Modeling Synopsis

1. Data Exploration & Preparation (Pandas & NumPy)

We use pandas and NumPy to load, inspect, and clean the Amazon sales dataset (`Amazon.csv`).

- `pd.read_csv("Amazon.csv")`: Loads the raw sales data into a DataFrame.
- `df.isnull().sum()`: Counts missing values in each column to assess data quality.
- `df.dropna(subset=["OrderDate", "TotalAmount"])`: Removes rows where critical fields (date, target amount) are missing, ensuring reliable analysis.
- `df.nunique()`, `df["Category"].unique()`: Checks the number of unique values and lists categories, brands, and cities to understand categorical features.
- `df["OrderDate"] = pd.to_datetime(df["OrderDate"])`: Converts the order date from string to proper datetime for time-series analysis.

We also strip column names, remove duplicates, and convert numeric columns (Quantity, UnitPrice, Discount, Tax, ShippingCost, TotalAmount) to numeric types to avoid type errors during modeling.

2. Feature Engineering

Several new features are created to make patterns more learnable for models:

- `OrderYear`, `OrderMonth`: Extracted from `OrderDate` for seasonal and trend analysis.
- `OrderIssue`: Binary target flag indicating whether an order had an issue (Returned / Cancelled = 1, otherwise 0).
- Aggregated customer features (later used for clustering) such as:
  - `NumOrders` (number of orders per customer)
  - `TotalSpend` (sum of `TotalAmount`)
  - `TotalQuantity`, `AvgDiscount`, `IssueRate`, `NumCategories`

These engineered features help the classifier and clustering algorithms capture customer behavior and temporal patterns.

3. Feature Transformation & Modeling Setup (Scikit-learn)

We use scikit-learn pipelines to handle preprocessing and modeling in a clean, reproducible way.

3.1 Preprocessing

- Numeric features (e.g., Quantity, UnitPrice, Discount, Tax, ShippingCost, TotalAmount, OrderYear, OrderMonth)
  - `SimpleImputer(strategy="mean")`: Fills any remaining numeric missing values.
  - `StandardScaler()`: Normalizes features to zero mean and unit variance so no single variable dominates.
- Categorical features (e.g., Category, Brand, PaymentMethod, City, State, Country)
  - `SimpleImputer(strategy="most_frequent")`: Replaces missing categories with the most common value.
  - `OneHotEncoder(handle_unknown="ignore")`: Converts categories into one-hot encoded vectors suitable for the model.

These transformers are combined in a `ColumnTransformer`, then wrapped in a `Pipeline` with the final estimator.

3.2 Core Scikit-learn Flow

- `.fit()`: Learns parameters (means, most frequent categories, encodings) from the training data.
- `.transform()`: Applies these learned parameters to new data.
- `.fit_transform()`: Convenience method used only on training data when fitting and transforming in one step.

4. Classification Model – XGBoost

To predict whether an order will have an issue (`OrderIssue`), we train an XGBoost classifier inside the pipeline:

- Model: `XGBClassifier` with tuned hyperparameters (number of estimators, learning rate, max depth, etc.).
- Data split: `train_test_split` with 80% training, 20% test, stratified on `OrderIssue`.
- Metrics: accuracy, precision, recall, F1-score, and full classification report.

This model helps identify risk factors (products, locations, payment types) associated with returns and cancellations.

5. Customer Segmentation – KMeans Clustering

On the aggregated customer table:

- Standardize features with `StandardScaler`.
- Try different cluster counts (e.g., k = 2–4) and compute the silhouette score to choose the best k.
- Assign each customer to a segment (`Cluster`) and summarize profiles:
    - Average spend, number of orders, issue rate, categories purchased.

These clusters reveal groups like "high-value low-issue customers" vs "low-spend high-issue customers," useful for marketing and support strategies.

6. Time-Series Forecasting – ARIMA

Using monthly total sales:

- Build a monthly series: `ts = df.set_index("OrderDate").resample("M")["TotalAmount"].sum()`.
- Split into train and test (last few months as test).
- Fit an ARIMA model on the training period.
- Forecast over the test horizon and compute RMSE to evaluate accuracy.
- Generate future forecasts (next N months) to estimate upcoming revenue trends.

This shows seasonality and growth patterns in sales.

7. NLP-Based Similar Product Recommender

We construct a content-based recommender using product text:

- Combine `ProductName`, `Category`, and `Brand` into a single text field.
- Clean text (lowercasing, removing special characters) for consistency.
- Use `TfidfVectorizer` (uni-grams and bi-grams) to create a TF-IDF matrix.
- Compute cosine similarity between products to find the most similar items.

A helper function `similar_items(product_name, top_n)` returns top-N recommended products based on this similarity, enabling "customers also viewed" style suggestions.

8. Saving Artifacts & Deployment Interfaces

To make the project reusable and deployable:

- Save trained models and artifacts with `pickle`:
    - `orderissue_model.pkl` (pipeline with XGBoost)
    - `customer_clusters.pkl` (cluster assignments / centroids)
    - `arima_sales_model.pkl` (time-series model)
    - `recommender.pkl` (products, index mapping, similarity matrix)

Desktop GUI (Tkinter)

A simple Tkinter-based dashboard provides:

- Buttons to show:
    - EDA charts (payment method distribution, sales by category).
    - Time-series forecast plot (train, test, forecast).
- An input box to type a product name and get similar item recommendations in the UI.

Flask API

A lightweight Flask API exposes:

- `/predict_order_issue` (POST): JSON input of order features → predicted issue flag + probability.
- `/recommend` (GET/POST): product name → list of similar products.

These interfaces demonstrate how the analytics can be integrated into applications or dashboards.