Table of Contents

Table of Figures

Table of Tables

Abstract

This report outlines the comprehensive design, development, and deployment of a machine learning pipeline aimed at predicting global video game sales, utilizing the VGSales dataset. The project employs a modular MLOps methodology, incorporating tools such as Docker, Apache Airflow, MariaDB, Redis, MLflow, and FastAPI. A thorough ELT (Extract, Load, Transform) strategy facilitates adaptable and scalable data transformations post-ingestion. The pipeline encompasses various stages, including data ingestion, validation, preprocessing, feature engineering, model development, evaluation, deployment, and monitoring. Exploratory data analysis is enhanced through extensive visualizations and statistical summaries that yield actionable insights. Critical considerations such as data privacy, security, fairness, and ethical deployment are meticulously addressed to uphold transparency and integrity. This project effectively showcases a technically sound, reproducible, and ethically responsible analytics solution that aligns with modern data engineering and MLOps best practices.

Keywords: MLOps, Data Analytics Pipeline, Video Game Sales, ELT, Machine Learning, Data Visualization, Ethics, Data Privacy, Apache Airflow, Docker, MariaDB, Redis, MLflow, FastAPI, Model Drift, Data Engineering

Introduction

This report contains the design and implementation of an MLOps pipeline for training in a regression model. More specifically, it predicts Global_Sales of a game based on features such as the platform, genre, publisher, release year, and sales in various regions. The importance of correctly forecasting video game sales revolves around the pragmatic implication toward stakeholders of the gaming industry, such as publishers, retailers, marketers, and developers. Reliable sales forecasts affect decisions to invest, how much to spend on promotion efforts, inventory control, and platform-based strategies. Hence, a generic and scalable machine learning pipeline had been constructed using several state-of-the-art tools like Docker, Apache Airflow, MariaDB, Redis, MLflow, FastAPI, and many Python libraries including Scikit-learn, Pandas, and PyArrow. The creation of each stage of the pipeline considered data integrity, reproducibility, modularity, and automation.The pipeline will ingest and preprocess data, develop a model, deploy it, and monitor the application using industry-standard tools such as Docker, Airflow, MLflow, and FastAPI.

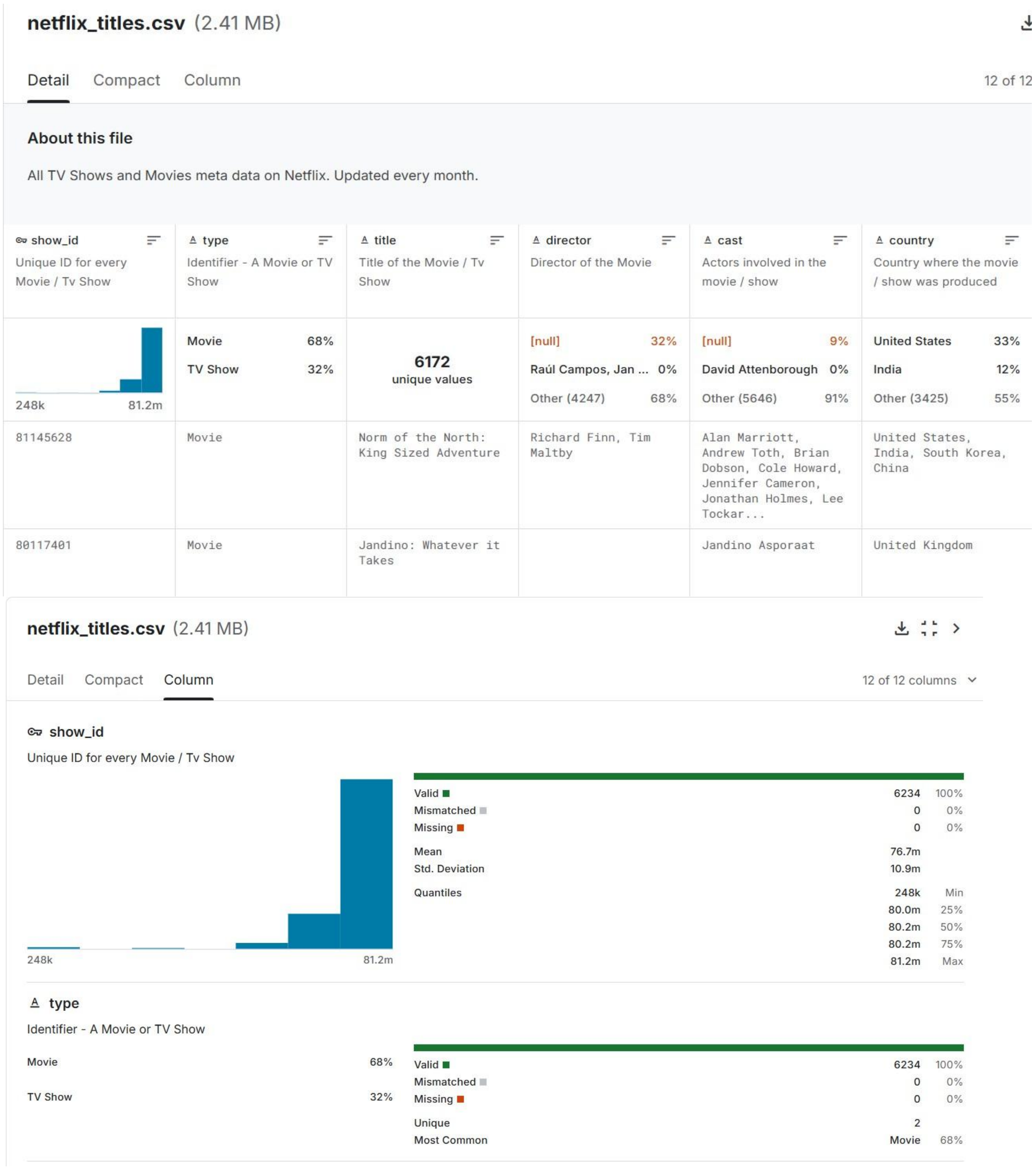
Datasets Comparison and Selection

Initially three datasets were chosen based on their potential to solve a supervised learning problem. The criteria for selection included rich data, a variety of features, a clear and measurable target variable, and relevance to a real-world application. The following is the detailed information on each dataset, and the justification on why the final dataset was chosen.

Dataset 1: Netflix Movies and TV shows

This dataset consists of information about TV shows and movies available on Netflix, with fields such as, title, type, director, cast, country, release_year, and genre. It was first hosted on Kaggle by Shivam Bansal, which was scrapped from Netflix's public catalogue. It represents various entertainment metadata but lacks user interaction data. It was primarily collected for cataloguing and for exploration data analysis of Netflix's streaming library. It is a CSV file which comes under flat file, and it has 12 columns and more than 8000 records (Bansal, 2019). It can be stored in local file, in a general CSV format with semi-structured fields like cast lists and genres.

It contains categorical features such as genre, country and rating, thus making it suitable for genre prediction, recommendation systems or content classification.



netflix_titles.csv (2.41 MB)

Detail Compact Column 12 of 12

About this file

All TV Shows and Movies meta data on Netflix. Updated every month.

show_id	type	title	director	cast	country
Unique ID for every Movie / Tv Show	Identifier - A Movie or TV Show	Title of the Movie / Tv Show	Director of the Movie	Actors involved in the movie / show	Country where the movie / show was produced
248k – 81.2m	Movie 68% TV Show 32%	6172 unique values	[null] 32% Raúl Campos, Jan ... 0% Other (4247) 68%	[null] 9% David Attenborough 0% Other (5646) 91%	United States 33% India 12% Other (3425) 55%
81145628	Movie	Norm of the North: King Sized Adventure	Richard Finn, Tim Maltby	Alan Marriott, Andrew Toth, Brian Dobson, Cole Howard, Jennifer Cameron, Jonathan Holmes, Lee Tockar...	United States, India, South Korea, China
80117401	Movie	Jandino: Whatever it Takes		Jandino Asporaat	United Kingdom

netflix_titles.csv (2.41 MB)

Detail Compact Column 12 of 12 columns

show_id

Unique ID for every Movie / Tv Show

248k 81.2m

Valid 🟩	6234	100%
Mismatched ▨	0	0%
Missing 🟥	0	0%
Mean	76.7m	
Std. Deviation	10.9m	
Quantiles	248k	Min
	80.0m	25%
	80.2m	50%
	80.2m	75%
	81.2m	Max

type

Identifier - A Movie or TV Show

Movie	68%
TV Show	32%

Valid 🟩	6234	100%
Mismatched ▨	0	0%
Missing 🟥	0	0%
Unique	2	
Most Common	Movie	68%

Figure 1: Details of netflix dataset on Kaggle (Bansal, 2019).

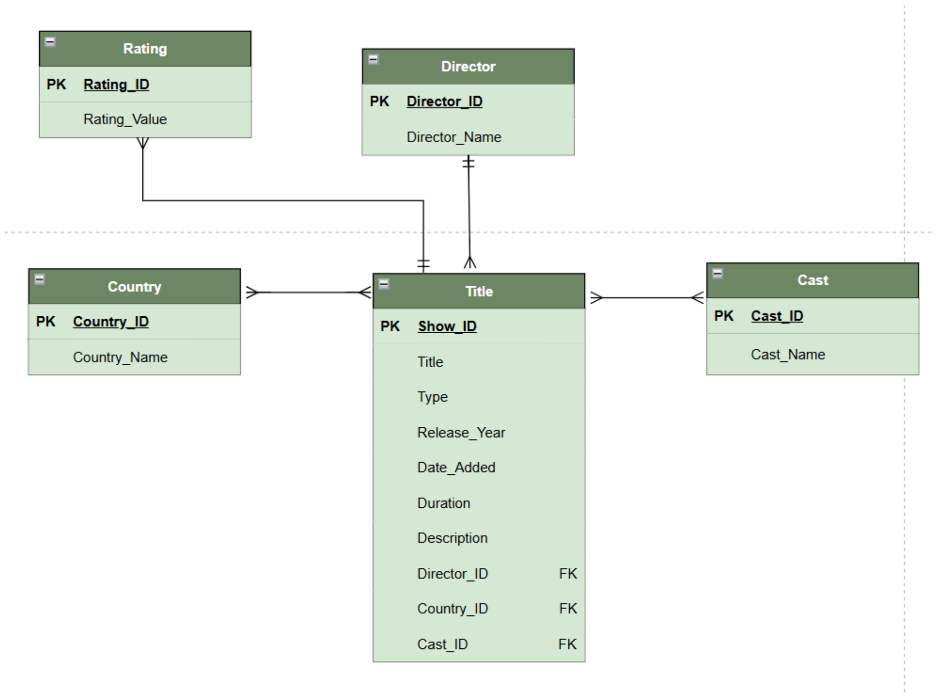
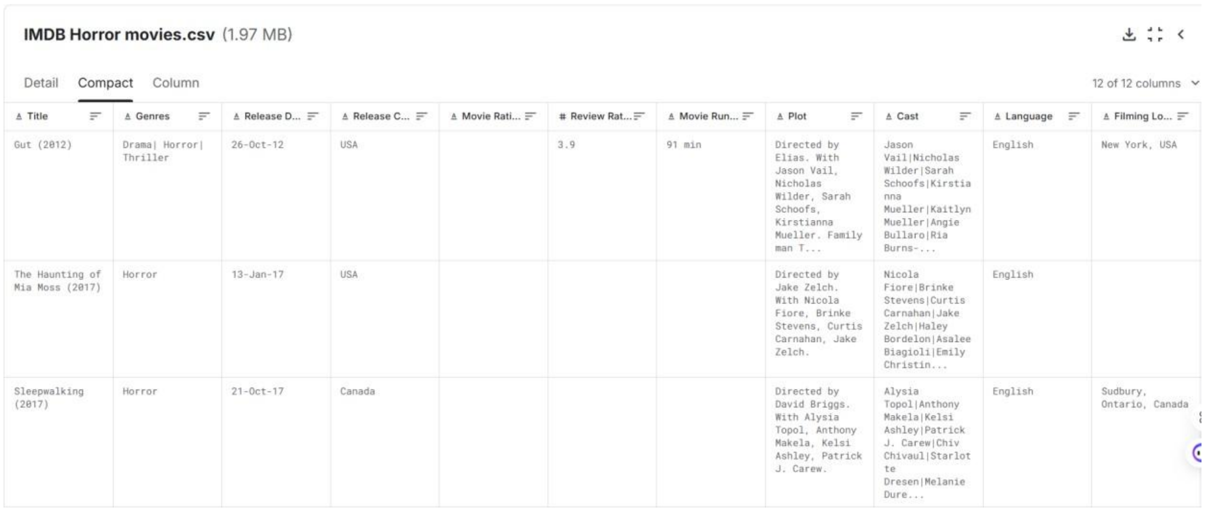


Figure 2: ER diagram of netflix dataset

Dataset 2: IMDb Horror Movie Dataset

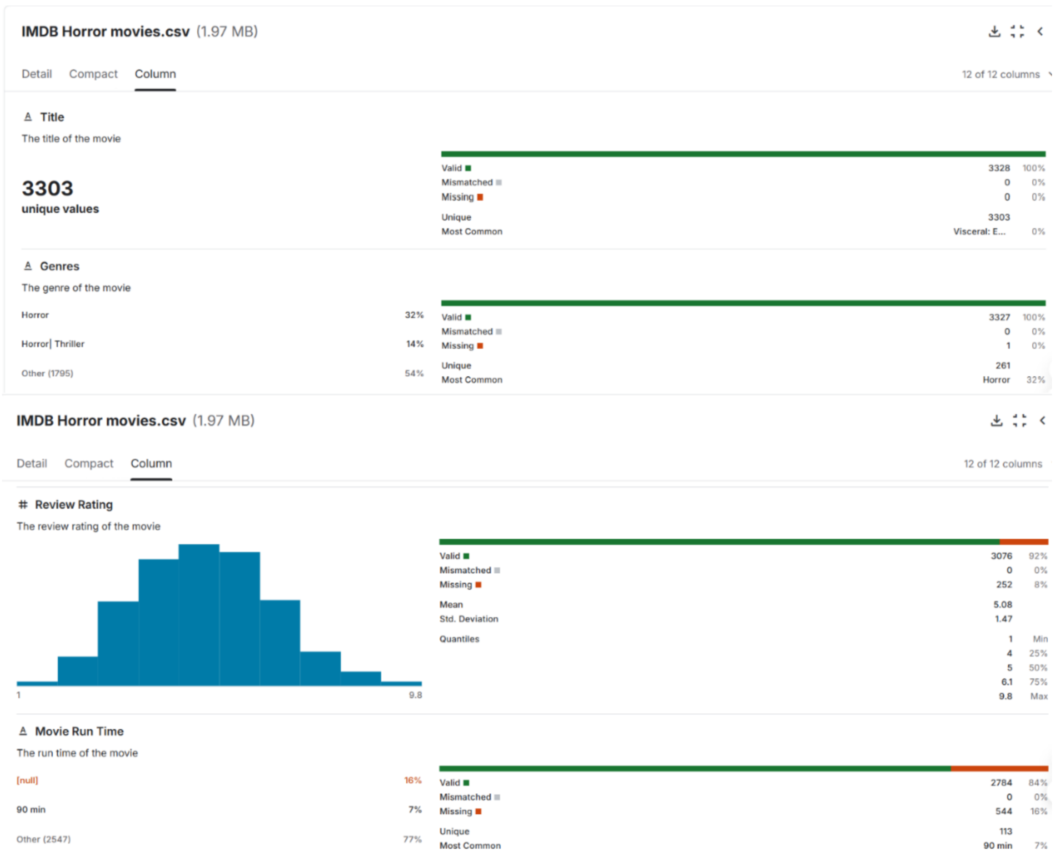
This dataset was originally provided by PromptCloudHQ which was scraped from IMDb and then uploaded on Kaggle, mainly focusing on horror genre films. It includes horror film metadata from IMDb such as title, year, runtime, certificate, genre, director, stars, IMDB rating and votes. This dataset was created for the purpose of sentiment analysis, popularity trend analysis and content profiling. It is a flat file meaning; it's in CSV format consisting of 13 columns and about 3500 rows (PromptCloud, 2017). It includes both the numerical fields and categorical attributes. There may be limitations concerning generalizability and applicability across broader media because it is limited to only one domain, that is horror films only.


IMDB Horror movies.csv (1.97 MB)

Detail Compact Column

12 of 12 columns

Title	Genres	Release D...	Release C...	Movie Rati...	Review Rat...	Movie Run...	Plot	Cast	Language	Filming Lo...
Gut (2012)	Drama| Horror| Thriller	26-Oct-12	USA		3.9	91 min	Directed by Elias. With Jason Vail, Nicholas Wilder, Sarah Schoofs, Kirstianna Mueller. Family man T...	Jason Vail|Nicholas Wilder|Sarah Schoofs|Kirstianna Mueller|Kaitlyn Mueller|Angie Bullaro|Ria Burns-...	English	New York, USA
The Haunting of Mia Moss (2017)	Horror	13-Jan-17	USA				Directed by Jake Zelch. With Nicola Fiore, Brinke Stevens, Curtis Carnahan, Jake Zelch.	Nicola Fiore|Brinke Stevens|Curtis Carnahan|Jake Zelch|Haley Bordelon|Asalee Biagioli|Emily Christin...	English	
Sleepwalking (2017)	Horror	21-Oct-17	Canada				Directed by David Briggs. With Alysia Topol, Anthony Makela, Kelsi Ashley, Patrick J. Carew.	Alysia Topol|Anthony Makela|Kelsi Ashley|Patrick J. Carew|Chiv Chivaul|Starlotte Dresen|Melanie Dure...	English	Sudbury, Ontario, Canada



IMDB Horror movies.csv (1.97 MB)

Detail Compact Column 12 of 12 columns

Title

The title of the movie

3303
unique values

Valid 🟩	3328	100%
Mismatched ▨	0	0%
Missing 🟥	0	0%
Unique	3303	
Most Common	Visceral: E...	0%

Genres

The genre of the movie

Horror	32%
Horror| Thriller	14%
Other (1795)	54%

Valid 🟩	3327	100%
Mismatched ▨	0	0%
Missing 🟥	1	0%
Unique	261	
Most Common	Horror	32%

IMDB Horror movies.csv (1.97 MB)

Detail Compact Column 12 of 12 columns

Review Rating

The review rating of the movie

Valid 🟩	3076	92%
Mismatched ▨	0	0%
Missing 🟥	252	8%
Mean	5.08	
Std. Deviation	1.47	
Quantiles	1	Min
	4	25%
	5	50%
	6.1	75%
	9.8	Max

Movie Run Time

The run time of the movie

[null]	16%
90 min	7%
Other (2547)	77%

Valid 🟩	2784	84%
Mismatched ▨	0	0%
Missing 🟥	544	16%
Unique	113	
Most Common	90 min	7%

Figure 3: IMDB dataset details on Kaggle *(PromptCloud, 2017).*

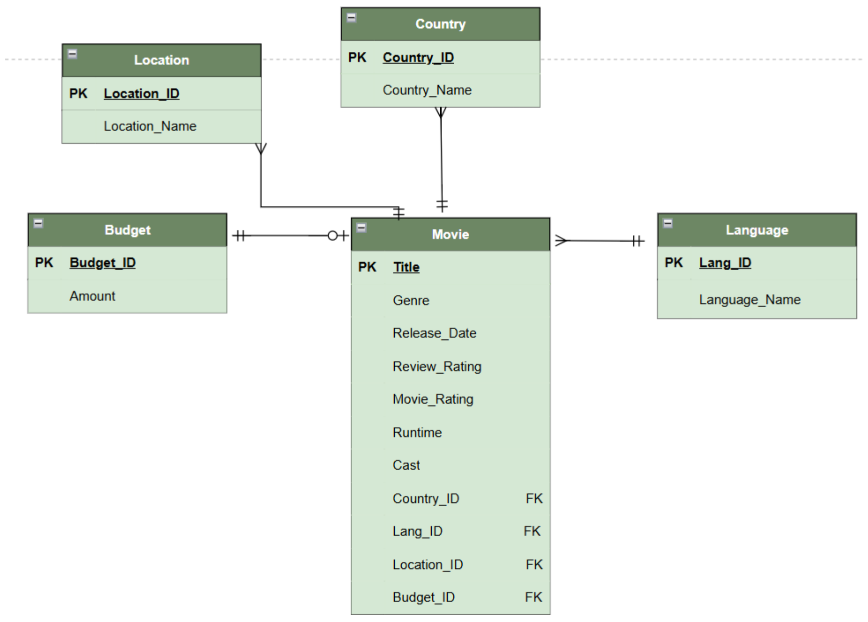
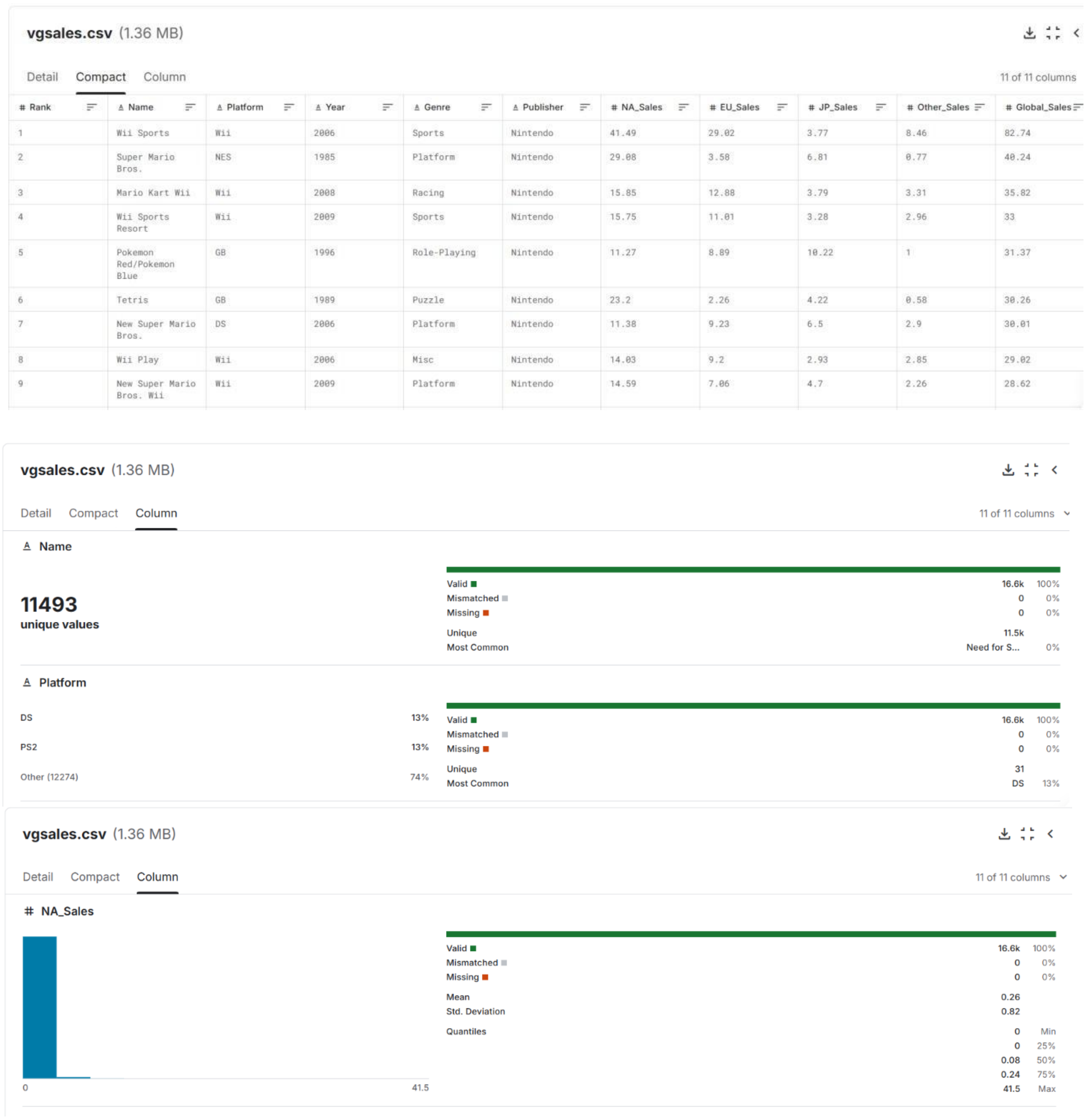


Figure 4: ER diagram of IMDB dataset

Dataset 3: Video Game Sales

The Video Game Sales dataset contains data on over 16,500 video games with fields such as Name, Platform, Year, Genre, Publisher, and sales across different regions, which is why it includes variables such as, NA_Sales, EU_Sales, JP_Sales, Other_Sales and Global_Sales. This dataset was collected and aggregated from various public databases by Gregorout on Kaggle, which supports detailed analytics. It is stored as a structured flat CSV file and includes a balanced mix of categorical and numerical data (Smith, no date). Its diversity allows for extensive feature engineering. For example, categorical variables such as Genre and Platform can be label-encoded or one-hot encoded, while numerical data on sales can be normalized or binned to serve as classification targets.



vgsales.csv (1.36 MB)

Detail | Compact | Column — 11 of 11 columns

# Rank	Name	Platform	Year	Genre	Publisher	# NA_Sales	# EU_Sales	# JP_Sales	# Other_Sales	# Global_Sales
1	Wii Sports	Wii	2006	Sports	Nintendo	41.49	29.02	3.77	8.46	82.74
2	Super Mario Bros.	NES	1985	Platform	Nintendo	29.08	3.58	6.81	0.77	40.24
3	Mario Kart Wii	Wii	2008	Racing	Nintendo	15.85	12.88	3.79	3.31	35.82
4	Wii Sports Resort	Wii	2009	Sports	Nintendo	15.75	11.01	3.28	2.96	33
5	Pokemon Red/Pokemon Blue	GB	1996	Role-Playing	Nintendo	11.27	8.89	10.22	1	31.37
6	Tetris	GB	1989	Puzzle	Nintendo	23.2	2.26	4.22	0.58	30.26
7	New Super Mario Bros.	DS	2006	Platform	Nintendo	11.38	9.23	6.5	2.9	30.01
8	Wii Play	Wii	2006	Misc	Nintendo	14.03	9.2	2.93	2.85	29.02
9	New Super Mario Bros. Wii	Wii	2009	Platform	Nintendo	14.59	7.06	4.7	2.26	28.62

vgsales.csv (1.36 MB)

Detail | Compact | Column — 11 of 11 columns

Name

11493
unique values

Valid 🟩	16.6k	100%
Mismatched ▨	0	0%
Missing 🟥	0	0%
Unique	11.5k	
Most Common	Need for S...	0%

Platform

DS	13%
PS2	13%
Other (12274)	74%

Valid 🟩	16.6k	100%
Mismatched ▨	0	0%
Missing 🟥	0	0%
Unique	31	
Most Common	DS	13%

vgsales.csv (1.36 MB)

Detail | Compact | Column — 11 of 11 columns

# NA_Sales

Valid 🟩	16.6k	100%
Mismatched ▨	0	0%
Missing 🟥	0	0%
Mean	0.26	
Std. Deviation	0.82	
Quantiles	0	Min
	0	25%
	0.08	50%
	0.24	75%
	41.5	Max

Figure 5: Details of Video Game Sales dataset on Kaggle (Smith, no date)

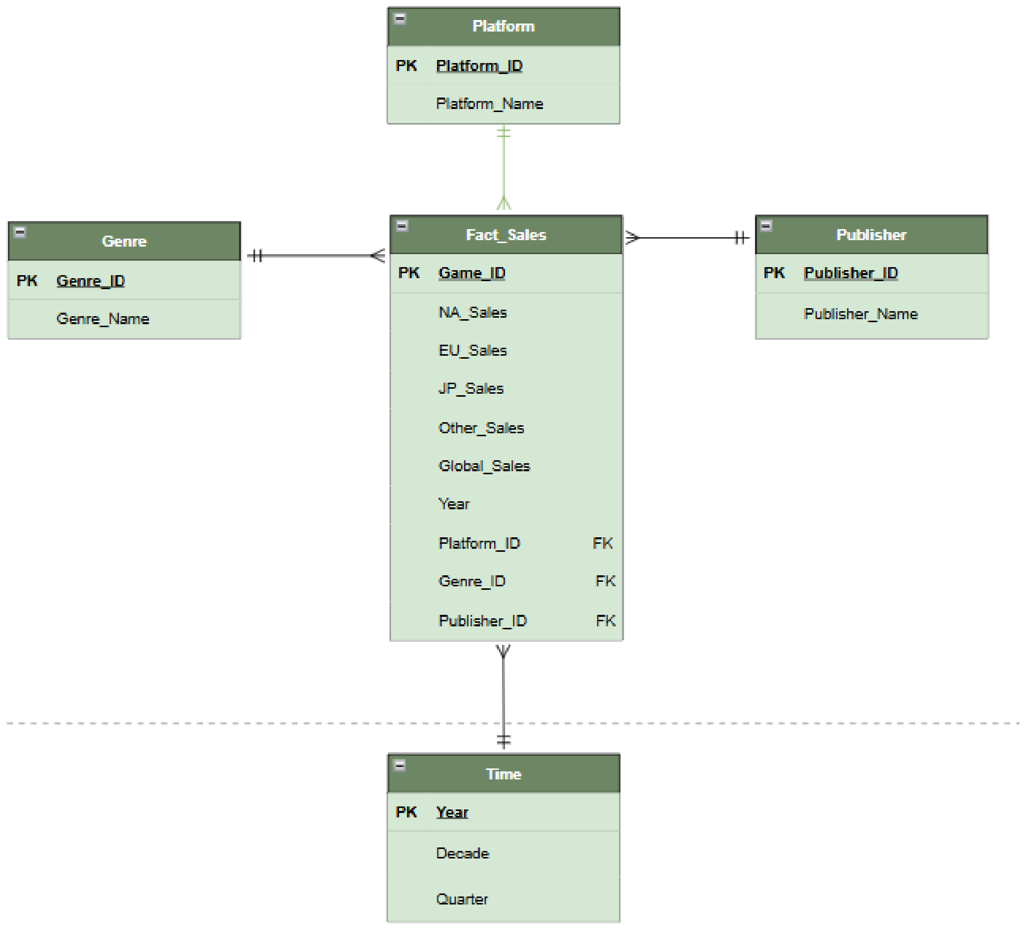


Figure 6: ER diagram of Video Game Sales dataset

Final Dataset Selection

After comparing the three datasets, the Video Game Sales (VGSales) dataset was chosen because it is relevant, well-organized, and has strong potential for business impact. It contains detailed information on video games released on various platforms, genres, and publishers, along with sales data from major markets like North America, Europe, and Japan. The rich metadata makes it possible to develop strong features, which is important for accurate sales predictions. The VGSales dataset is a strong choice because it closely matches real-world market needs. It helps stakeholders understand how platform trends, game genres, and regional sales interact. Unlike the other datasets, VGSales offers clear, well-organized, and historically important data that supports meaningful analysis of market behavior over time.

On the other hand, the Netflix Movies and TV Shows dataset, while rich in details like cast, director, and genre, doesn't have a useful numerical target for regression. It's better suited for recommendation systems or organizing content. The IMDb Horror Movies dataset includes numbers like budget and runtime but focuses on a narrow genre and has many missing or inconsistent entries. This limits its usefulness and makes it harder to build a reliable, scalable model.

Data Analytics Pipeline

General Overview of Pipeline

A machine learning pipeline is a well-organized system that automates the steps needed to build, deploy, and maintain a machine learning model. It helps data move smoothly from the raw input stage to the final prediction output. The main purpose of a pipeline is to make the process more consistent, reduce mistakes, make results easier to reproduce, and allow the system to grow easily (Google Cloud, no date).

Generally, a machine learning pipeline consists of the following steps:

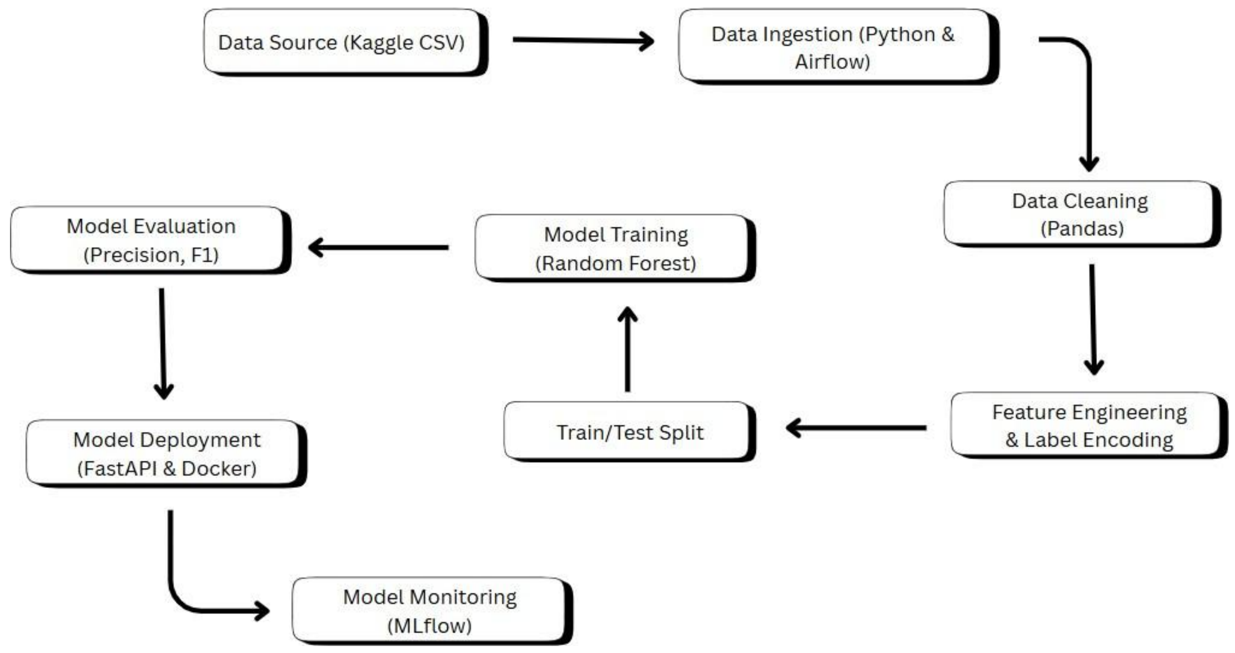

Data Source (Kaggle CSV)
Data Ingestion (Python & Airflow)
Data Cleaning (Pandas)
Feature Engineering & Label Encoding
Train/Test Split
Model Training (Random Forest)
Model Evaluation (Precision, F1)
Model Deployment (FastAPI & Docker)
Model Monitoring (MLflow)

Figure 7: General Pipeline process

1. Data Collection

   This step involves acquiring raw data from one or multiple sources, which may include databases, APIs, files, or web scraping. The quality and structure of the data at this stage greatly influence the overall model performance (IBM, 2021).

2. Data ingestion

   Ingestion is the process of retrieving the raw dataset from its original source (like, CSV format) and importing it into a structured data environment, for example, SQL database. It ensures that there is consistent access to structured data for both analysts and automated systems, as it acts as a checkpoint for data validation before storage. Generally, data engineers configure the ingestion pipelines using airflow to orchestrate it via scheduled DAGs, on which using the python script, tasks such as reading, cleaning, and schema mapping are done before inserting the data into the database (GeeksforGeeks, 2024).

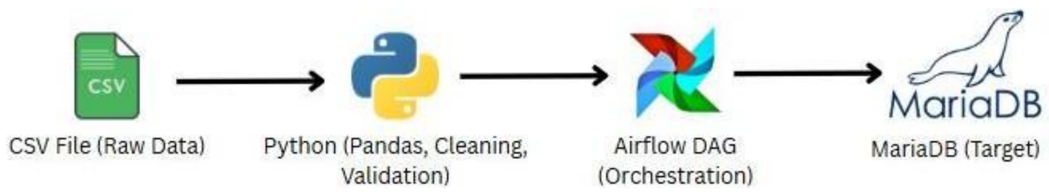

CSV
CSV File (Raw Data)
Python (Pandas, Cleaning, Validation)
Airflow DAG (Orchestration)
MariaDB
MariaDB (Target)

Figure 8: Data ingestion process

3. Data Validation

   Before any transformation or modeling, it's also crucial to validate data for issues like missing values, incorrect formats, or out-of-range values which helps ensure that the dataset is suitable for analysis (Talk Cloud, 2024).

4. Data preprocessing

   Data preprocessing is about preparing the data for modelling by transforming it into a consistent and usable format, with tasks such as missing value imputation, feature engineering, and encoding categorical variables. This step is conducted because raw datasets often contain inconsistencies or non-numeric features that must be resolved before modelling, and preprocessing ensures that the input is compatible while model's accuracy is also improved (D. Jain, 2019).

5. Feature Engineering

   In this stage, new features are derived from existing ones to enhance model performance. This might involve aggregations, transformations, or domain-specific logic to improve the predictive power of the dataset (GeeksforGeeks, 2023).

6. Model development

   This stage involves training and evaluating a supervised classification model to predict a game's sales class. Modelling is done to transform the structured data into a predictive tool that supports various decision making. For this step, a machine learning algorithm is applied to the training dataset to learn the underlying patterns which involves selecting an appropriate algorithm, tuning hyperparameters, and fitting the model (Shubham Sayon, 2020).

7. Model Evaluation

   The trained model needs to be evaluated on unseen data (validation or test sets) using various performance metrics to ensure that the model is generalizing well and is not overfitting the training data.

8. Model Deployment

   Model deployment is the process of exposing a trained model to other systems through an interface such as an API, as it allows the model to serve predictions in real-time, making it accessible for non-technical users or integrated systems. Deployment infrastructures are managed by MLOps engineers which Product teams use to predict through dashboards. The trained model is then serialized using joblib and served using FastAPI.

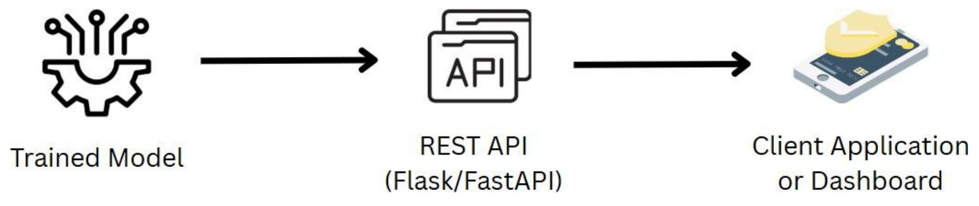


Figure 9: Model deployment process

9. Model Monitoring

   To track a model's performance after deploying it to detect drift and trigger retraining, the model is monitored regularly because without doing so model performance may degrade over time due to data changes or outdated patterns. Monitoring can be done using tools like MLflow to predict and compare the outcomes, also alerts can be configured to retrain the model when accuracy falls below set thresholds (Wang, 2024).

Approach to Pipeline Design, Implementation, and Storage

Data Type and Measurement

The selected dataset **vgsales** includes the following variables with their statistical data types:

Variables	**Statistical data types**
Name (Nominal)	Identifier for video games
Platform (Nominal)	Categories such as PS2, Wii, etc.
Year (Ordinal/ Interval)	Treated as interval data for trend analysis
Genre (Nominal)	Categories like Action, Sports, etc.
Publisher (Nominal)	Non-numeric identifiers of game publishers
NA_Sales, EU_Sales, Other_Sales, Global_Sales (Ratio)	Continuous numeric values that represent regional and global sales figures

Table 1: Variables with their statistical data types

This distinction in variables is then used to inform the types of analysis and visualizations used, which can be categorized into the following:

- Bar charts were utilized for nominal variables such as Platform and Genre to illustrate the frequency of game releases or the average sales within each category, thereby emphasizing the most dominant platforms or genres.
- Histograms were employed to illustrate the distribution of numerical variables, including Global_Sales and regional sales. This analysis uncovered right-skewed patterns, indicating that most games sold fewer than 1 million units, while a selected few bestsellers achieved significantly higher sales figures.
- Box plots facilitated the visual identification of outliers in sales across various genres and platforms, assisting in the recognition of performance variability within categories.

- Heatmaps of correlation matrices were employed for ratio variables to identify multicollinearity, exemplified by the strong correlations observed between NA_Sales and Global_Sales, which proved beneficial in the selection of model features.
- Line plots along with decade-based groupings were employed using Year (considered as an interval) to depict historical trends in game publishing and sales volume.

In summary, these visualizations served not only as exploratory instruments but also played a crucial role in guiding decisions related to feature engineering, model tuning, and the interpretation of business-relevant insights. Nominal data was represented through bar plots and count plots, whereas ratio data was assessed using histograms, boxplots, and correlation matrices. The visualizations are available in Appendix 8.

Research Questions

The exploratory data analysis helped formulate these three key research questions to aid on understanding more:

Firstly, how do regional sales vary by genre and platform over the past decades?	This inquiry seeks to reveal trends in the performance of different game genres and platforms across various markets, including North America, Europe, Japan, and others. By compiling sales data over the decades, it became evident that certain platforms, such as PlayStation 2 or Wii, excel in specific regions and for genres, for instance, Sports games in North America. Visual representations, including grouped bar charts and stacked plots, offered comprehensive insights into these sales dynamics.
Secondly, which features exert the most significant influence on global sales predictions?	An analysis of feature importance utilizing Random Forest Regressor and correlation heatmaps indicated that regional sales, particularly NA_Sales and EU_Sales, were strong predictors of Global_Sales. Other features, such as Platform and Genre, although categorical, impacted performance based on their encoding and were reflected in the importance scores of the model. These insights contributed to refining the selection of predictive features during model development.
Thirdly, are there observable temporal trends in sales volumes across various platforms or genres?	By categorizing the release years into decades and examining sales trends over time, it was noted that certain platforms dominated specific periods, such as PS2 in the early 2000s and Wii in the late 2000s. Likewise, particular genres, including Action and Sports, exhibited peaks in popularity during different timeframes. Line plots and decade-based boxplots facilitated the observation of these long-term trends and provided valuable insights into the evolution of the gaming industry over time.

Distribution and Relationship Insights

The analysis of the dataset's distribution and relationships unveiled significant structural patterns. The Global_Sales data displayed a pronounced right-skewed distribution, suggesting that while most games achieved relatively modest sales, a select few blockbuster titles experienced exceptional success. This understanding was pivotal in the selection of regression techniques that are resilient to skewness and guided the application of log-transformed variables for model optimization. The distributions of Platform and Genre were also markedly uneven, with legacy platforms such as PS2 and Nintendo Wii prevailing in the dataset. This not only reflects historical market share but also introduced biases that required mitigation during model training through the balancing of feature representation. In terms of genre, Action and Sports games were the most prevalent, influencing the segmentation strategies for feature engineering.

Correlation analysis indicated a strong positive relationship between regional sales figures, particularly NA_Sales and EU_Sales, and Global_Sales. These features emerged as essential inputs for the predictive model due to their significant predictive capabilities. Heatmaps visually corroborated the presence of multicollinearity among regional sales, aiding in the avoidance of redundancy during feature selection. Categorical patterns were apparent in the alignment of specific platforms with certain genres or decades, exemplified by the strong association of Role-Playing games with Japanese platforms and Action games with global blockbusters. This contextual understanding not only enriches the analysis but also informs business decisions, such as marketing strategies targeted at specific regions.

Differences between Ingested and Raw data

Basis	**Explanation**
Data Size Reduction	The initial dataset comprised roughly 16,500 rows. This number decreased to about 4,000 rows during the ingestion process to enhance processing speed and minimize computational overhead throughout the iterative model development phase.
Removal of Missing Target Values	All entries lacking value in the Global_Sales field were omitted to maintain data consistency and prevent complications during model training.
Standardization of Data Types	Throughout the processes of ingestion and preprocessing, every field underwent validation and standardization — this included the conversion of strings into categorical types and the adjustment of numerical columns into appropriate float or integer formats.
Encoding and Transformation	Categorical variables, including Platform and Genre, were encoded using label encoding to make them suitable for application in machine learning models.
Schema Integration	The dataset was restructured to conform to a star schema, facilitating efficient storage in MariaDB, which aligns with the requirements of OLAP-style querying and subsequent analytics.

Model Drift and its Impact

Model drift refers to the occurrence where the predictive performance of a machine learning model diminishes over time due to alterations in the underlying data distribution. In practical applications, this can greatly impact the reliability and accuracy of the system if not properly managed. Model drift is mainly divided into two categories: data drift and concept drift.

1. Data Drift	Data drift takes place when the statistical characteristics of the input features change over time, even if the relationship between input and output remains constant. For instance, in the video game sector, the emergence of new platforms or shifting consumer preferences can result in changes in the frequency of specific genres or platforms within the dataset. This drift can lead to the model making less precise predictions because it was trained on outdated data that no longer mirrors current trends.
2. Concept Drift	Concept drift occurs when the relationship between input features and the target variable shifts. In the context of this project, concept drift might emerge if factors affecting game sales (such as marketing strategies, subscription-based gaming, or digital distribution models) evolve in a manner that changes the significance or effect of existing features. This renders previously learned relationships obsolete and requires model retraining.
3. Impact on Deployed Systems	The repercussions of model drift can encompass diminished prediction accuracy, erroneous business insights, and a decline in trust in automated systems. In production settings, this can result in poor decision-making, misallocated resources, and lost opportunities. Thus, recognizing and addressing drift is essential for preserving the model's relevance and effectiveness.

Table 2: Impacts of model drift

Combating Model Drift in the Deployed System

Several approaches can be applied to prevent model drift in the deployed video game sales prediction system so that it stays accurate and in line with new market trends.

Regular Monitoring and Performance Tracking	To begin, set up a pipeline for continuous model monitoring by using tools such as MLflow, since they are already included in the present system. If the performance measures (such as MAE, RMSE or R^2) show a big decline, it suggests that drift is happening. Thresholds or alerts for when metrics start to drop can cause a quick response.
Scheduled Model Retraining	A regular re-training of the model on updated patterns within the gaming industry should happen about once every four months because new developments appear frequently. Thanks to machine learning, the algorithm can follow shifts in data distributions or important features in the data set. The pipeline can still use the usual preprocessing and architecture, though it includes the up-to-date sales data.

Data Collection and Dataset Refresh	It is necessary to frequently update the dataset to solve both data and concept drift. Therefore, one should add new video game sales records from recent years to what is currently in the dataset. As a result, the model can learn about new trends such as digital games selling better, the increasing popularity of subscriptions or changes in the most popular genres on every platform.
Feature re-evaluation and Engineering	When features that once helped in prediction become less relevant, model drift may also take place. Add a step to your retraining by using tools like Random Forests or SHAP values to calculate feature importance now and then. This makes one doubt the value of Platform and Genre in predicting sales or advises the addition of new features related to the mode a game is available in and its release season.

Table 3: Combating measures for model drift

Data Extraction, Loading, and Transformation Process

Stage	**Action**
Source	CSV file downloaded from Kaggle (Video Game Sales dataset)
Extraction	Automated using a Python script with pandas, managed by Airflow DAG
Validation	Schema checks: column names, data types, missing/ duplicate entries
Transformation (ETL)	Pre-cleaning in Python: fill missing Publisher, impute Year, drop null Global_Sales
Alternative (ELT)	Load raw data into staging table then clean using SQL scripts or views
Loading	Data is inserted into MariaDB using a star schema (fact and dimensions
Access	Data us queried via SQL or extracted using Python for analytics or ML

Table 4: Extraction, Loading and Transformation Process

ELT approach instead of ETL

In this data analytics pipeline, the ELT (Extract, Load, Transform) methodology is employed rather than the conventional ETL (Extract, Transform, Load) process. This is evident in the workflow's structure: raw data is initially extracted and subsequently loaded directly into storage systems like Redis and MariaDB, with transformations applied later during model training or prediction. Tasks such as addressing missing values, feature encoding, and data augmentation are conducted after the data has been loaded into memory or storage, rather than prior to that.

Further explanation on why specifically ELT was used is provided in Appendix 9.

Pipeline Design Strategy

The pipeline was crafted to be modular, scalable, and automated mirroring contemporary MLOps methodologies. Each phase of the machine learning lifecycle was segmented into autonomous, script-based elements. These elements were managed through Apache Airflow utilizing a Directed Acyclic Graph (DAG) to establish the sequence of task execution and their interdependencies. Each DAG task was aligned with a significant processing phase: ingestion, validation, cleaning, preprocessing, model training, logging, and deployment.

A basic look into modular workflow design:

- All code was structured within a specific Conda environment (vgsales_environment).
- The project directory vgsales_project/housed CSV files.
- The main Airflow orchestration script, pipeline.py, facilitated the overall execution flow.
- Each task within the pipeline was created as an independent Python script located in the dags/ folder for streamlined integration and management.

1. Tools and Libraries used

For this project, the MLOps pipeline was built using a mix of open-source tools, picking each one based on how well it performs, scales, and how much support and familiarity our team has with it. Following is the list of tools used for this project:

Tool & Libraries	**Purpose**
Docker	Containerization of MariaDB, Redis, and other services
Airflow	Task scheduling and pipeline orchestration
MariaDB	Storage of structured ingested data
Redis	Temporary data exchange between pipeline tasks
MLFlow	Tracking and versioning trained models
FastAPI	Exposing the model through a RESTful API
Scikit-learn	Model training and evaluation
Pandas	Data Loading, manipulation, and preprocessing
numpy	Numerical operations and generating synthetic noise
PyArrow	Efficient binary serialization between panda and Redis
sqlalchemy	ORM (Object-Relational Mapping) and database engine for uploading DataFrames to MariaDB
joblib	Saving and loading ML models and encoders
Matplotlib.pyplot	Plotting feature importance graphs
pydantic	Request data validation in the FastAPI server
Great_expectations	Automate data validation and quality checks

Table 5: Tools and Libraries used

Further explanation on why those tools and libraries was used is provided in Appendix 1.

Pipeline Implementation and Deployment Stages

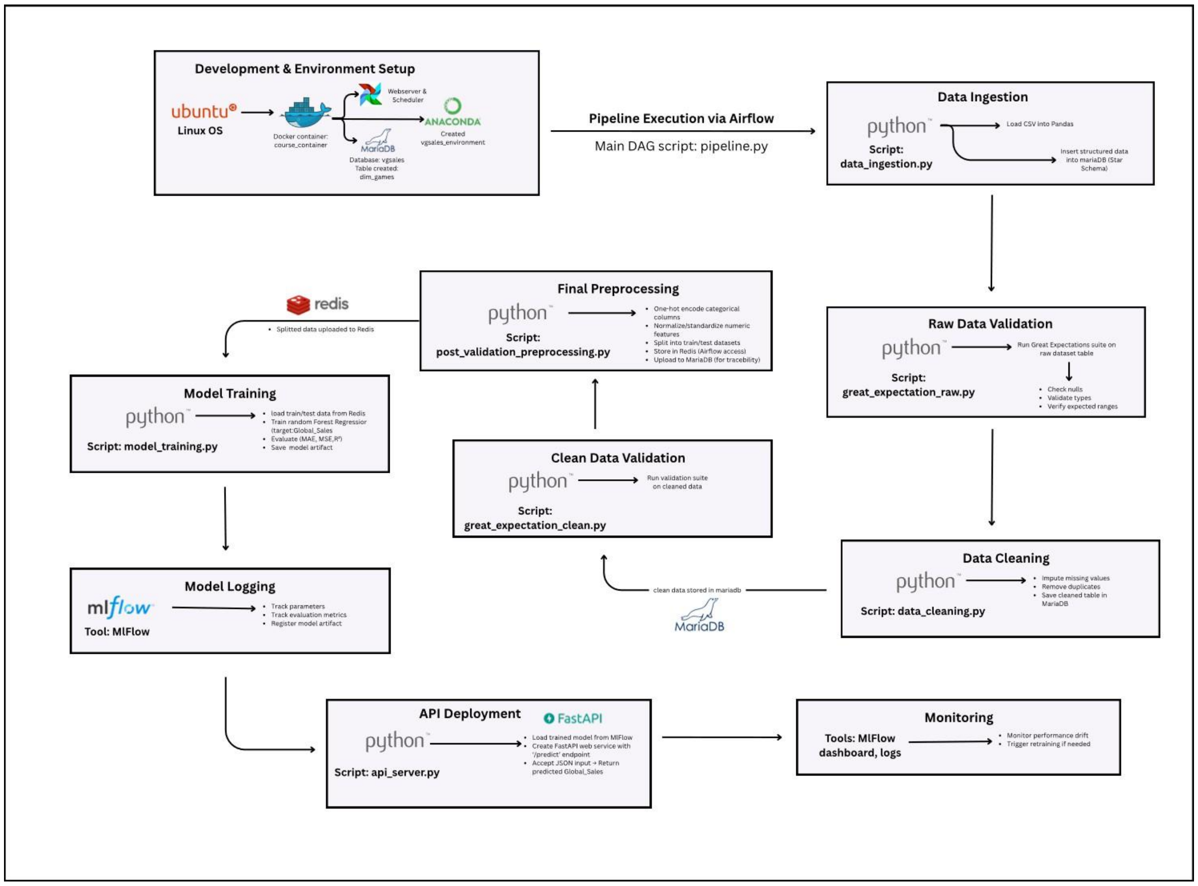
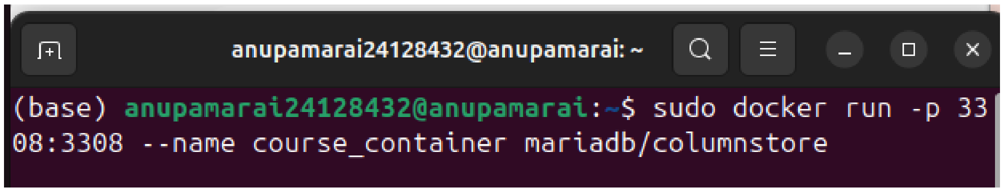

Development & Environment Setup
ubuntu
Linux OS
Docker container: course_container
Webserver & Scheduler
ANACONDA
Created vgsales_environment
MariaDB
Database: vgsales
Table created: dim_games
Pipeline Execution via Airflow
Main DAG script: pipeline.py
Data Ingestion
python
Script: data_ingestion.py
Load CSV into Pandas
Insert structured data into mariaDB (Star Schema)
Raw Data Validation
python
Script: great_expectation_raw.py
Run Great Expectations suite on raw dataset table
• Check nulls
• Validate types
• Verify expected ranges
Data Cleaning
python
Script: data_cleaning.py
• Impute missing values
• Remove duplicates
• Save cleaned table in MariaDB
clean data stored in mariadb
MariaDB
Clean Data Validation
python
Script: great_expectation_clean.py
Run validation suite on cleaned data
Final Preprocessing
python
Script: post_validation_preprocessing.py
• One-hot encode categorical columns
• Normalize/standardize numeric features
• Split into train/test datasets
• Store in Redis (Airflow access)
• Upload to MariaDB (for tracebility)
redis
• Splitted data uploaded to Redis
Model Training
python
Script: model_training.py
• load train/test data from Redis
• Train random Forest Regressior (target:Global_Sales
• Evaluate (MAE, MSE,R²)
• Save model artifact
Model Logging
mlflow
Tool: MlFlow
• Track parameters
• Track evaluation metrics
• Register model artifact
API Deployment
FastAPI
python
Script: api_server.py
• Load trained model from MlFlow
• Create FastAPI web service with '/predict' endpoint
• Accept JSON input → Return predicted Global_Sales
Monitoring
Tools: MlFlow dashboard, logs
• Monitor performance drift
• Trigger retraining if needed

Figure 10: The overall flow and process of implementing and deploying the pipeline

1. System Setup and Development Environment

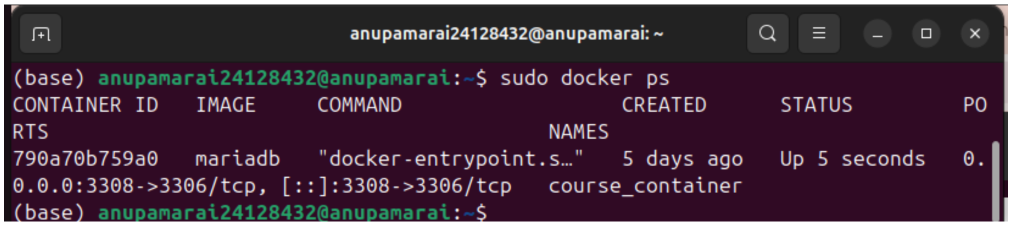
   The pipeline was designed and implemented within a Linux (Ubuntu) environment to guarantee compatibility with open-source software and realistic deployments. A Docker container referred to as **course_container** was utilized to encapsulate critical services including MariaDB and Apache Airflow. This configuration enhanced portability and isolation throughout various development environments. The Python environment was managed via Conda environment (**vgsales_environment**), to ensure that all the version control of dependencies. The codes were executed using DAG, while creating structured logs in airflow server, so the .py files containing codes were saved inside the dags folder in airflow folder.

   First, required packages were installed, for example, docker, miniconda, etc. Then a MariaDB columnStore container was created after pulling the latest one successfully.



```
anupamarai24128432@anupamarai: ~
(base) anupamarai24128432@anupamarai:~$ sudo docker run -p 33
08:3308 --name course_container mariadb/columnstore
```

Figure 11: Creating a MariaDB container named course_container

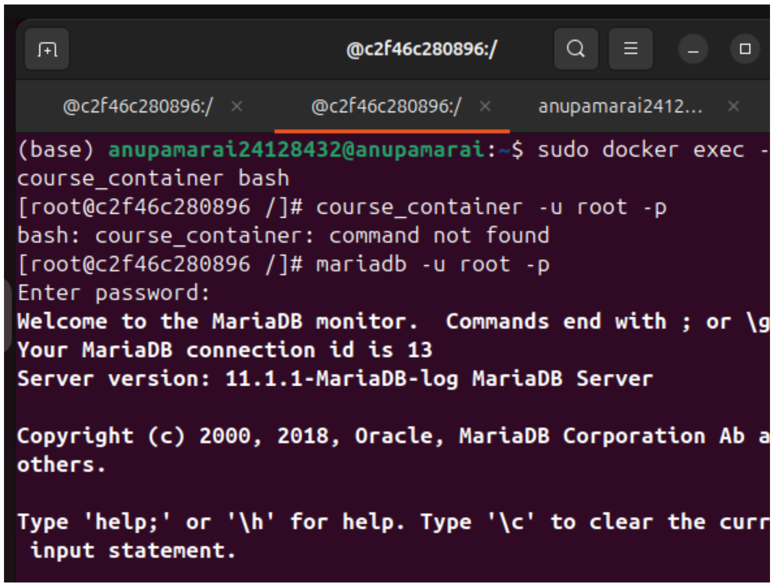


anupamarai24128432@anupamarai: ~

```
(base) anupamarai24128432@anupamarai:~$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED      STATUS         PO
RTS                                        NAMES
790a70b759a0   mariadb   "docker-entrypoint.s…"   5 days ago   Up 5 seconds   0.
0.0.0:3308->3306/tcp, [::]:3308->3306/tcp   course_container
(base) anupamarai24128432@anupamarai:~$
```

Figure 12: Checking if the created container is started or not

Then a user with accessibility privileges is created, which the user can log in to MariaDB CLI to execute sql commands like create database, show tables, etc. Creation of databases and tables, execution of such commands are further shown in Data ingestion stage.



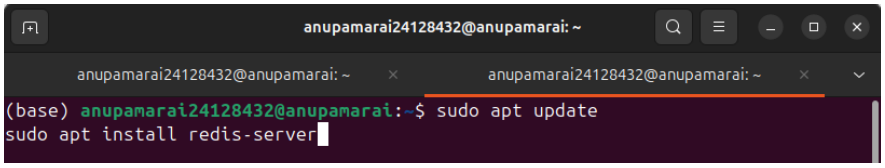
@c2f46c280896:/

@c2f46c280896:/ | @c2f46c280896:/ | anupamarai2412...

```
(base) anupamarai24128432@anupamarai:~$ sudo docker exec -
course_container bash
[root@c2f46c280896 /]# course_container -u root -p
bash: course_container: command not found
[root@c2f46c280896 /]# mariadb -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g
Your MariaDB connection id is 13
Server version: 11.1.1-MariaDB-log MariaDB Server

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab a
others.

Type 'help;' or '\h' for help. Type '\c' to clear the curr
 input statement.
```

Figure 13: Accessing container's CLI (Command Line Interface)

Redis is installed and ran inside the docker container.



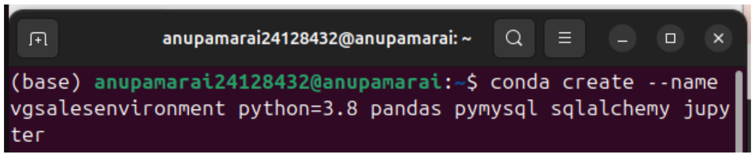
anupamarai24128432@anupamarai: ~

anupamarai24128432@anupamarai: ~ | anupamarai24128432@anupamarai: ~

```
(base) anupamarai24128432@anupamarai:~$ sudo apt update
sudo apt install redis-server
```

Figure 14: Installing redis

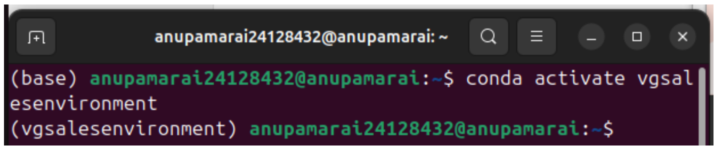
A Conda environment is created using the command "conda create --name <env_name> python=<version>", which sets up an isolated workspace with a specific Python version and dependencies. This helps avoid conflicts between packages across different projects.



anupamarai24128432@anupamarai: ~

```
(base) anupamarai24128432@anupamarai:~$ conda create --name
vgsalesenvironment python=3.8 pandas pymysql sqlalchemy jupy
ter
```

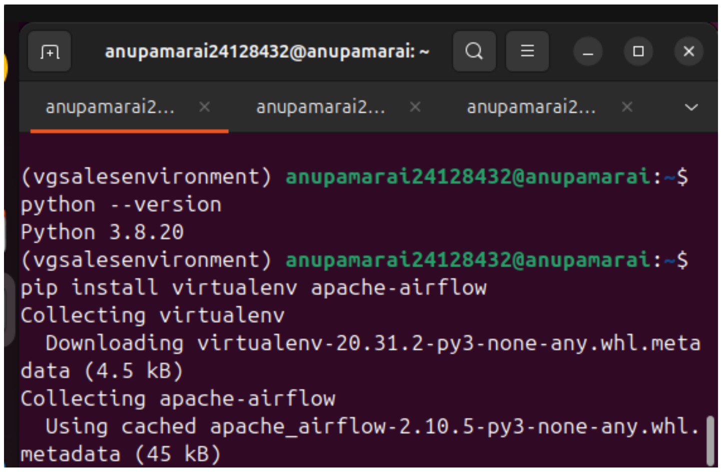
Figure 15: Creation of a conda environment named vgsalesenvironment



```
(base) anupamarai24128432@anupamarai:~$ conda activate vgsal
esenvironment
(vgsalesenvironment) anupamarai24128432@anupamarai:~$
```

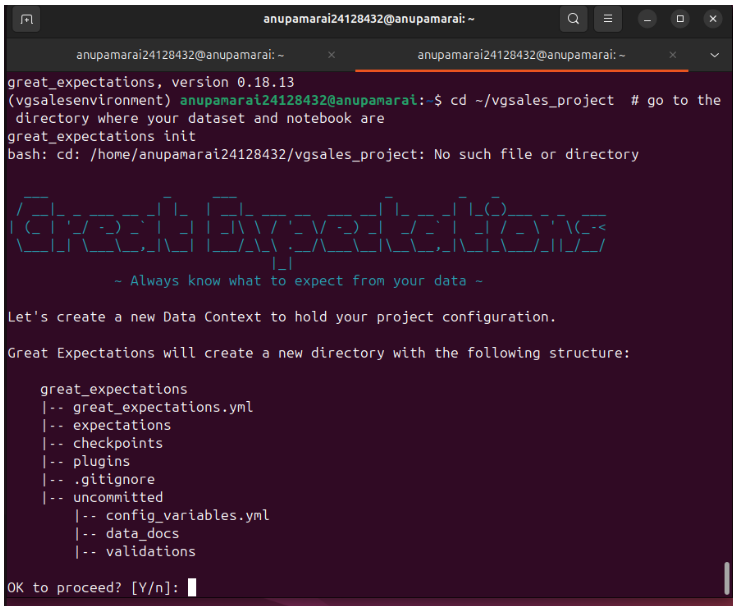
Figure 15: Activating the conda environment

After activating the Conda environment, required packages like airflow, mlflow, python libraries, etc. were installed inside the environment.



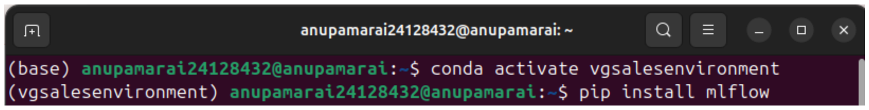
```
(vgsalesenvironment) anupamarai24128432@anupamarai:~$
python --version
Python 3.8.20
(vgsalesenvironment) anupamarai24128432@anupamarai:~$
pip install virtualenv apache-airflow
Collecting virtualenv
  Downloading virtualenv-20.31.2-py3-none-any.whl.meta
data (4.5 kB)
Collecting apache-airflow
  Using cached apache_airflow-2.10.5-py3-none-any.whl.
metadata (45 kB)
```

Figure 16: Installing python and airflow inside the conda environment



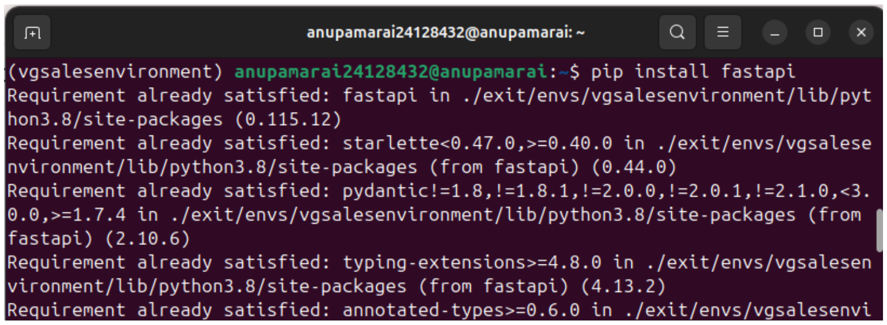
```
great_expectations, version 0.18.13
(vgsalesenvironment) anupamarai24128432@anupamarai:~$ cd ~/vgsales_project  # go to the
 directory where your dataset and notebook are
great_expectations init
bash: cd: /home/anupamarai24128432/vgsales_project: No such file or directory

  ___              _     ___                  _        _   _
 / __|_ _ ___ __ _| |_  | __|_ ___ __  ___ __| |_ __ _| |_(_)___ _ _  ___
| (_ | '_/ -_) _` |  _| | _|\ \ / '_ \/ -_) _|  _/ _` |  _| / _ \ ' \(_-<
 \___|_| \___\__,_|\__| |___/_\_\ .__/\___\__|\__\__,_|\__|_\___/_||_/__/
                                |_|
             ~ Always know what to expect from your data ~

Let's create a new Data Context to hold your project configuration.

Great Expectations will create a new directory with the following structure:

    great_expectations
    |-- great_expectations.yml
    |-- expectations
    |-- checkpoints
    |-- plugins
    |-- .gitignore
    |-- uncommitted
        |-- config_variables.yml
        |-- data_docs
        |-- validations

OK to proceed? [Y/n]:
```

Figure 17: Installing Great Expectations



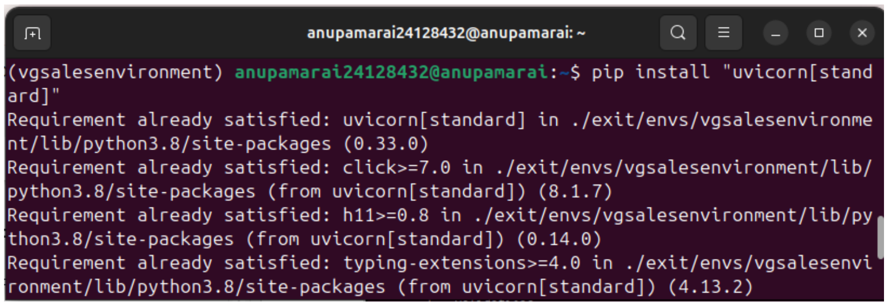
```
(base) anupamarai24128432@anupamarai:~$ conda activate vgsalesenvironment
(vgsalesenvironment) anupamarai24128432@anupamarai:~$ pip install mlflow
```

Figure 18: Installing mlflow



```
(vgsalesenvironment) anupamarai24128432@anupamarai:~$ pip install fastapi
Requirement already satisfied: fastapi in ./exit/envs/vgsalesenvironment/lib/pyt
hon3.8/site-packages (0.115.12)
Requirement already satisfied: starlette<0.47.0,>=0.40.0 in ./exit/envs/vgsalese
nvironment/lib/python3.8/site-packages (from fastapi) (0.44.0)
Requirement already satisfied: pydantic!=1.8,!=1.8.1,!=2.0.0,!=2.0.1,!=2.1.0,<3.
0.0,>=1.7.4 in ./exit/envs/vgsalesenvironment/lib/python3.8/site-packages (from
fastapi) (2.10.6)
Requirement already satisfied: typing-extensions>=4.8.0 in ./exit/envs/vgsalesen
vironment/lib/python3.8/site-packages (from fastapi) (4.13.2)
Requirement already satisfied: annotated-types>=0.6.0 in ./exit/envs/vgsalesenvi
```

Figure 19: Installing FastAPI



```
(vgsalesenvironment) anupamarai24128432@anupamarai:~$ pip install "uvicorn[stand
ard]"
Requirement already satisfied: uvicorn[standard] in ./exit/envs/vgsalesenvironme
nt/lib/python3.8/site-packages (0.33.0)
Requirement already satisfied: click>=7.0 in ./exit/envs/vgsalesenvironment/lib/
python3.8/site-packages (from uvicorn[standard]) (8.1.7)
Requirement already satisfied: h11>=0.8 in ./exit/envs/vgsalesenvironment/lib/py
thon3.8/site-packages (from uvicorn[standard]) (0.14.0)
Requirement already satisfied: typing-extensions>=4.0 in ./exit/envs/vgsalesenvi
ronment/lib/python3.8/site-packages (from uvicorn[standard]) (4.13.2)
```

Figure 20: Installing uvicorn

2. Data Ingestion

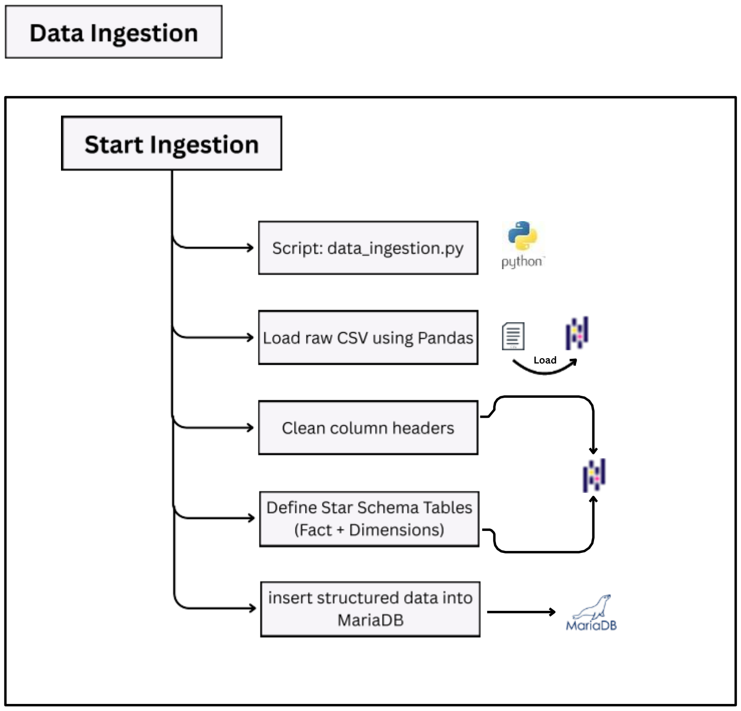
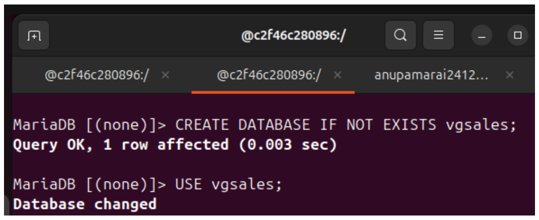

Data Ingestion
Start Ingestion
Script: data_ingestion.py
python
Load raw CSV using Pandas
Load
Clean column headers
Define Star Schema Tables (Fact + Dimensions)
insert structured data into MariaDB
MariaDB

Figure 21: Data ingestion process

After pulling the raw dataset from Kaggle and stored locally, a MariaDB database was then set up to load and store the clean and structured version of the video game sales dataset. First, as referred to Figure 11, a docker container named course_conatiner was created to host the MariaDB service, ensuring consistency across environments.

Then inside this container a database named vgsales was created and inside the database, a table named dim_game was created.



```
MariaDB [(none)]> CREATE DATABASE IF NOT EXISTS vgsales;
Query OK, 1 row affected (0.003 sec)

MariaDB [(none)]> USE vgsales;
Database changed
```

Figure 22: Creation of vgsales database

Then using Pandas, the CSV file containing the raw data was read into memory, this is done to transition data from an unstructured file-based format (CSV) into a structured in-memory DataFrame.

```
    print("📥 Reading CSV file...")
    df = pd.read_csv(csv_path)
    print("✅ CSV read successfully.")
```

Figure 23: Reading raw datasets into memory

Then a connection to the MariaDB database was established using SQLAlchemy.

```
engine = create_engine(
    f"mysql+pymysql://{db_user}:{db_password}@{db_host}:{db_port}/{db_name}"
)
```

Figure 24: Establishing a database connection

The ingested DataFrame was then written into the previously created dim_game table using the to_sql method.

```
print(f"🗃 Ingesting data into `{table_name}` table...")
df.to_sql(table_name, con=engine, if_exists='replace', index=False)
print("✅ Data ingestion completed.")
return df
```

Figure 25: Ingesting Data into the Database

The if_exists=’replace’ argument ensures that if the table already exists, it is replaced with the new data. The full snippet of the code and the output of data ingestion process is uploaded in Appendix 2.

3. Data Validation

Data validation was executed in two phases to guarantee the overall quality and dependability of the dataset. The initial validation focused on the raw data to detect problems such as missing values, duplicates, incorrect data types, and outliers. Following the cleaning process, a second validation round was carried out to verify that these issues had been adequately resolved and that the dataset conformed to the necessary structure for model training. This two-step methodology contributed to ensuring that the data was both precise and prepared for subsequent analysis.

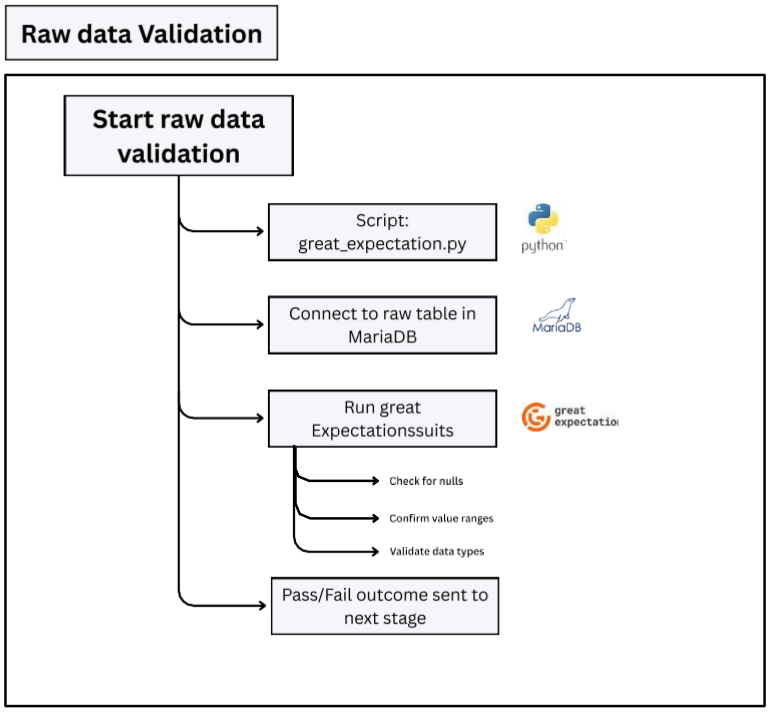
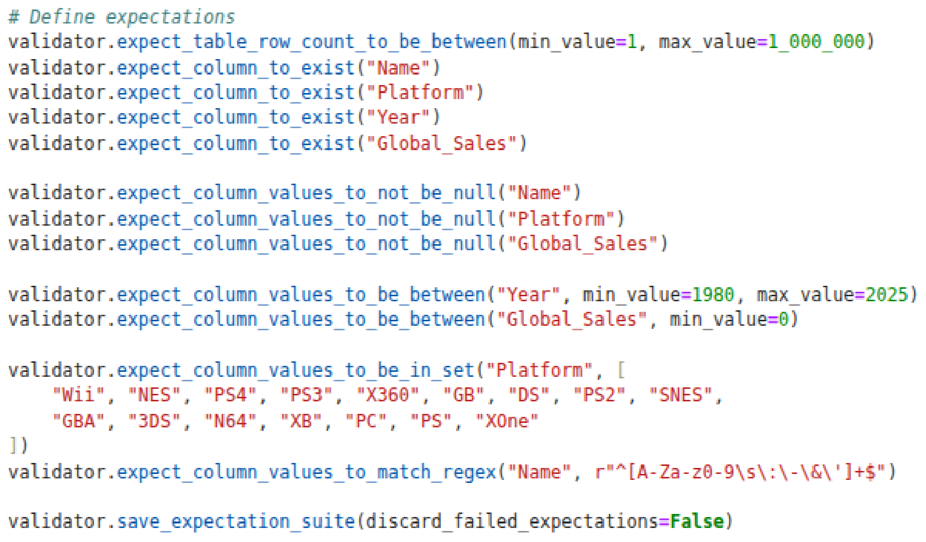

Raw data Validation
Start raw data validation
Script: great_expectation.py
python
Connect to raw table in MariaDB
MariaDB
Run great Expectationssuits
great expectatio
Check for nulls
Confirm value ranges
Validate data types
Pass/Fail outcome sent to next stage

Figure 26: Raw data validation process

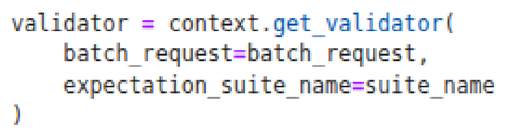
The expectations were defined to check for the presence of key columns (Name, Platform, Year, Global_Sales), non-null constraints, valid ranges and matching formats.



```
# Define expectations
validator.expect_table_row_count_to_be_between(min_value=1, max_value=1_000_000)
validator.expect_column_to_exist("Name")
validator.expect_column_to_exist("Platform")
validator.expect_column_to_exist("Year")
validator.expect_column_to_exist("Global_Sales")

validator.expect_column_values_to_not_be_null("Name")
validator.expect_column_values_to_not_be_null("Platform")
validator.expect_column_values_to_not_be_null("Global_Sales")

validator.expect_column_values_to_be_between("Year", min_value=1980, max_value=2025)
validator.expect_column_values_to_be_between("Global_Sales", min_value=0)

validator.expect_column_values_to_be_in_set("Platform", [
    "Wii", "NES", "PS4", "PS3", "X360", "GB", "DS", "PS2", "SNES",
    "GBA", "3DS", "N64", "XB", "PC", "PS", "XOne"
])
validator.expect_column_values_to_match_regex("Name", r"^[A-Za-z0-9\s\:\-\&\']+$")

validator.save_expectation_suite(discard_failed_expectations=False)
```

Figure 27: Raw data validation code

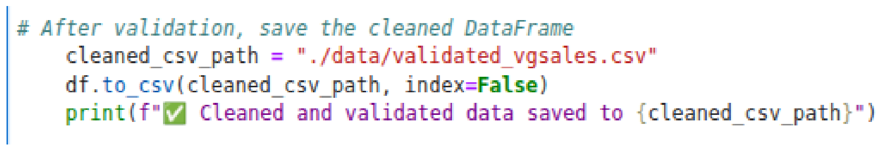
After applying the necessary cleaning steps (handling missing values, filtering records, correcting formats), a second validation was conducted on the cleaned dataset. The same expectations were re-used to confirm that all issues had been resolved, and that the dataset now complied with the quality standards.



```
validator = context.get_validator(
    batch_request=batch_request,
    expectation_suite_name=suite_name
)
```

Figure 28: Clean data validation code

After successful validation, the cleaned and validated dataaset was saved for further use.



```
# After validation, save the cleaned DataFrame
    cleaned_csv_path = "./data/validated_vgsales.csv"
    df.to_csv(cleaned_csv_path, index=False)
    print(f"✅ Cleaned and validated data saved to {cleaned_csv_path}")
```

Figure 29: Clean data saved after validation

The full code and outputs are in Appendix 3.

4. Data Preprocessing

The preprocessing stage took place in two main parts: initial cleaning to verify the correctness of the raw data and later steps of post-validation preprocessing to ensure the good quality of the data. Such a stage was to transform the dataset in such a way that it would be eligible for analysis and the performance of machine learning algorithms.

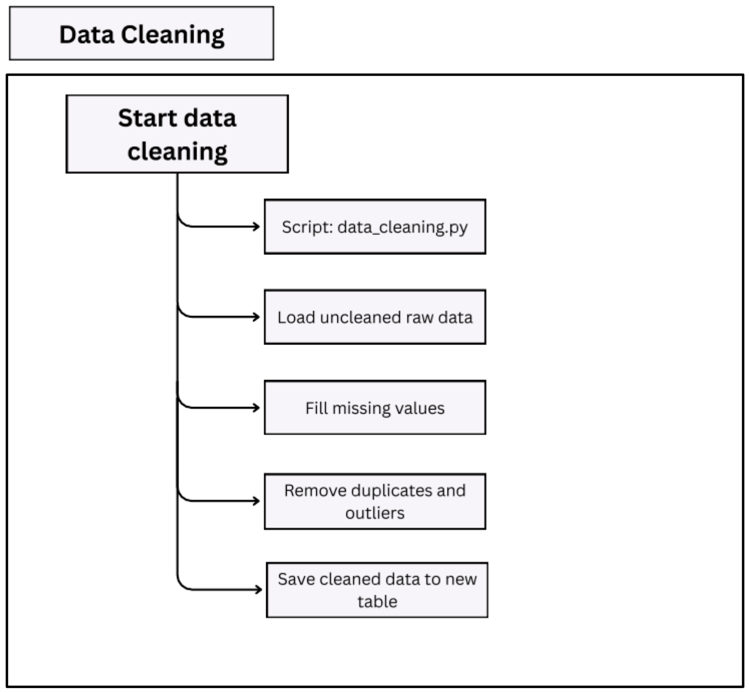
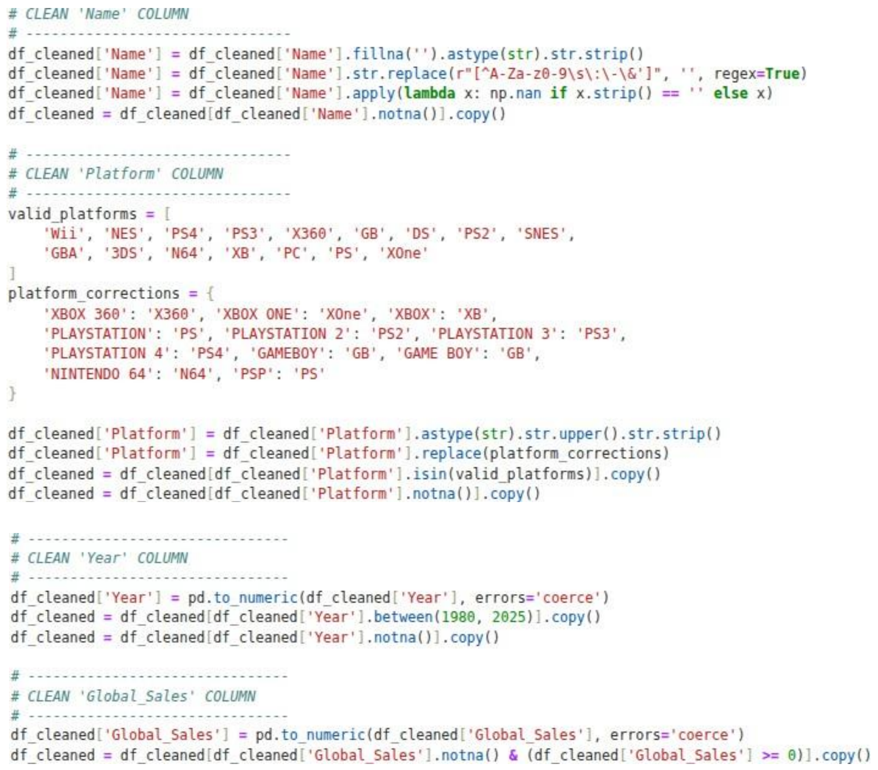


Figure 30: Data cleaning process

After identifying issues in the raw data during initial validation, the process consisting of changing the Name column, checking the numeric values of the year and global sales to be in the correct range, and adjusting the regional sales cells where the sales data was missing, executed.



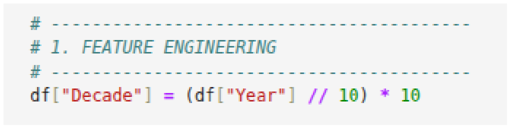
```
# CLEAN 'Name' COLUMN
# -------------------------------
df_cleaned['Name'] = df_cleaned['Name'].fillna('').astype(str).str.strip()
df_cleaned['Name'] = df_cleaned['Name'].str.replace(r"[^A-Za-z0-9\s\:\-\&']", '', regex=True)
df_cleaned['Name'] = df_cleaned['Name'].apply(lambda x: np.nan if x.strip() == '' else x)
df_cleaned = df_cleaned[df_cleaned['Name'].notna()].copy()

# -------------------------------
# CLEAN 'Platform' COLUMN
# -------------------------------
valid_platforms = [
    'Wii', 'NES', 'PS4', 'PS3', 'X360', 'GB', 'DS', 'PS2', 'SNES',
    'GBA', '3DS', 'N64', 'XB', 'PC', 'PS', 'XOne'
]
platform_corrections = {
    'XBOX 360': 'X360', 'XBOX ONE': 'XOne', 'XBOX': 'XB',
    'PLAYSTATION': 'PS', 'PLAYSTATION 2': 'PS2', 'PLAYSTATION 3': 'PS3',
    'PLAYSTATION 4': 'PS4', 'GAMEBOY': 'GB', 'GAME BOY': 'GB',
    'NINTENDO 64': 'N64', 'PSP': 'PS'
}

df_cleaned['Platform'] = df_cleaned['Platform'].astype(str).str.upper().str.strip()
df_cleaned['Platform'] = df_cleaned['Platform'].replace(platform_corrections)
df_cleaned = df_cleaned[df_cleaned['Platform'].isin(valid_platforms)].copy()
df_cleaned = df_cleaned[df_cleaned['Platform'].notna()].copy()

# ------------------------------
# CLEAN 'Year' COLUMN
# ------------------------------
df_cleaned['Year'] = pd.to_numeric(df_cleaned['Year'], errors='coerce')
df_cleaned = df_cleaned[df_cleaned['Year'].between(1980, 2025)].copy()
df_cleaned = df_cleaned[df_cleaned['Year'].notna()].copy()

# ------------------------------
# CLEAN 'Global_Sales' COLUMN
# ------------------------------
df_cleaned['Global_Sales'] = pd.to_numeric(df_cleaned['Global_Sales'], errors='coerce')
df_cleaned = df_cleaned[df_cleaned['Global_Sales'].notna() & (df_cleaned['Global_Sales'] >= 0)].copy()
```

Figure 31: Data cleaning

This cleaned dataset was then passed into the second round of Great Expectations validation to confirm the conformity with the defined expectations.

Once the data passed the final validation stage, further preprocessing was applied to prepare the dataset.

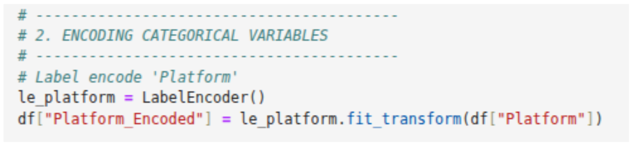
1. Feature Engineering: A new Decade feature was created to represent the release decade of each game, aiding in temporal analysis.

   ```
   # ----------------------------------------
   # 1. FEATURE ENGINEERING
   # ----------------------------------------
   df["Decade"] = (df["Year"] // 10) * 10
   ```

   Figure 32: Feature Engineering

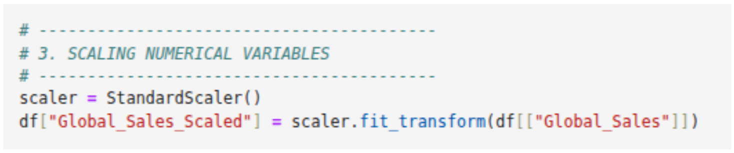
2. Encoding: Categorical data such as Platform was label encoded to convert it into numerical format suitable for machine learning models.

   ```
   # ----------------------------------------
   # 2. ENCODING CATEGORICAL VARIABLES
   # ----------------------------------------
   # Label encode 'Platform'
   le_platform = LabelEncoder()
   df["Platform_Encoded"] = le_platform.fit_transform(df["Platform"])
   ```

   Figure 33: Label Encoding

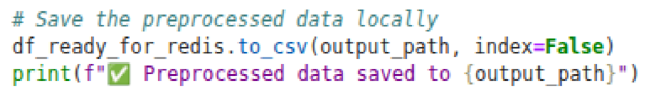
3. Scaling: The Global_Sales values were normalized using StandardScaler to ensure uniform feature scaling.



```
# ----------------------------------------
# 3. SCALING NUMERICAL VARIABLES
# ----------------------------------------
scaler = StandardScaler()
df["Global_Sales_Scaled"] = scaler.fit_transform(df[["Global_Sales"]])
```

Figure 34: Scaling

Then the fully preprocessed dataset was saved for later uses in training and deployment.

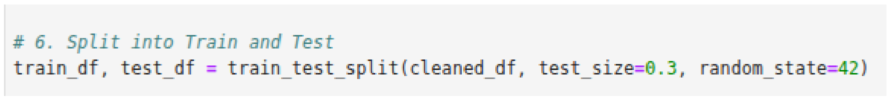


```
# Save the preprocessed data locally
df_ready_for_redis.to_csv(output_path, index=False)
print(f"✅ Preprocessed data saved to {output_path}")
```

Figure 35: Preprocessed data saved

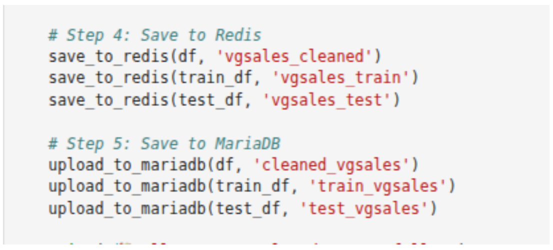
The full code is made available in Appendix 4.

Then after preprocessing, the dataset was split into training and test subsets using train_test_split. Each subset, along with the full dataset, was serialized using APache Arrow and stored in Redis for retreival during the model development and inference stages, and the dataset was also uploaded to mariadb.



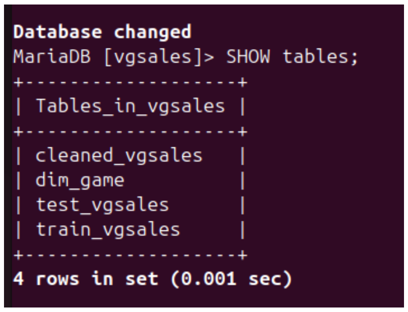
```
# 6. Split into Train and Test
train_df, test_df = train_test_split(cleaned_df, test_size=0.3, random_state=42)
```

Figure 36: Data splitted into train and test set



```
    # Step 4: Save to Redis
    save_to_redis(df, 'vgsales_cleaned')
    save_to_redis(train_df, 'vgsales_train')
    save_to_redis(test_df, 'vgsales_test')

    # Step 5: Save to MariaDB
    upload_to_mariadb(df, 'cleaned_vgsales')
    upload_to_mariadb(train_df, 'train_vgsales')
    upload_to_mariadb(test_df, 'test_vgsales')
```

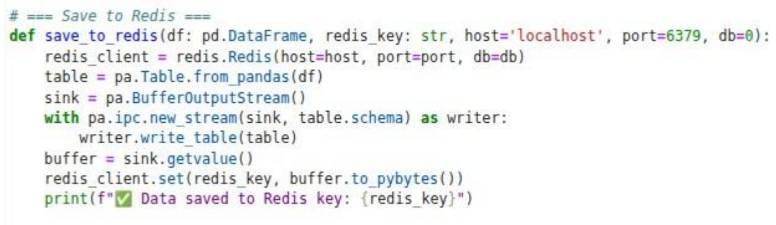
Figure 37: Saved and uploaded to redis and mariadb respectively



```
Database changed
MariaDB [vgsales]> SHOW tables;
+-------------------+
| Tables_in_vgsales |
+-------------------+
| cleaned_vgsales   |
| dim_game          |
| test_vgsales      |
| train_vgsales     |
+-------------------+
4 rows in set (0.001 sec)
```

Figure 38: Cleaned and splitted datas uploaded in MariaDB

The save_to_redis function converts each DatFrame to Arrow IPC format and stores it using redis.set().

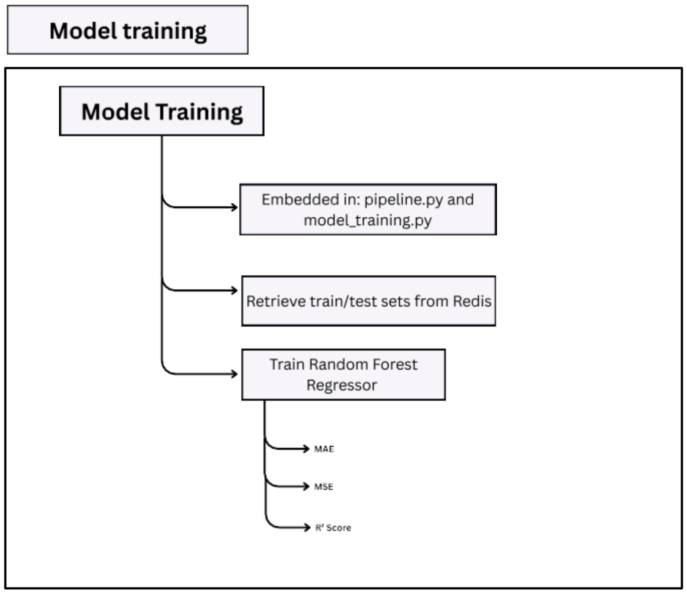


```
# === Save to Redis ===
def save_to_redis(df: pd.DataFrame, redis_key: str, host='localhost', port=6379, db=0):
    redis_client = redis.Redis(host=host, port=port, db=db)
    table = pa.Table.from_pandas(df)
    sink = pa.BufferOutputStream()
    with pa.ipc.new_stream(sink, table.schema) as writer:
        writer.write_table(table)
    buffer = sink.getvalue()
    redis_client.set(redis_key, buffer.to_pybytes())
    print(f"✅ Data saved to Redis key: {redis_key}")
```

Figure 39: save_to_redis function

The full code showing this uploading in Redis process is in Appendix 5.

5. Model Development



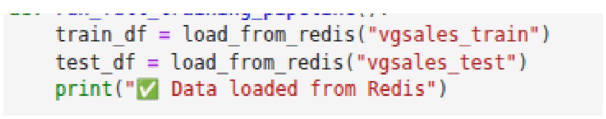
Model training
Model Training
Embedded in: pipeline.py and model_training.py
Retrieve train/test sets from Redis
Train Random Forest Regressor
MAE
MSE
R² Score

The Evaluation metrics used to evaluate the trained model are:

Metric	**Purpose**
MSE (Mean Squared Error)	Measures the average squared difference between predicted and actual values
RMSE (Root Mean Squared Error)	Provides error in the same units as the target (Global_Sales), penalizes large errors
MAE (Mean Absolute Error)	Measures the average absolute error between predictions and actual values
R^2	Indicates how well the model explains variance in the target variable
Explained Variance	Measures the proportion of variance captured by the model

Table 6: Evaluation Metrics used

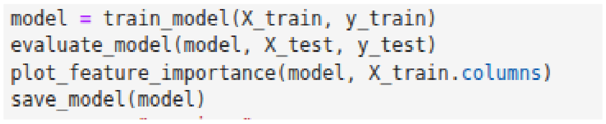
The training and test sets were retrieved directly from Redis, then a RandomForestRegressor model was trained, evaluated and logged using MlFlow.



```
    train_df = load_from_redis("vgsales_train")
    test_df = load_from_redis("vgsales_test")
    print("✅ Data loaded from Redis")
```

Figure 40: Loading data from redis

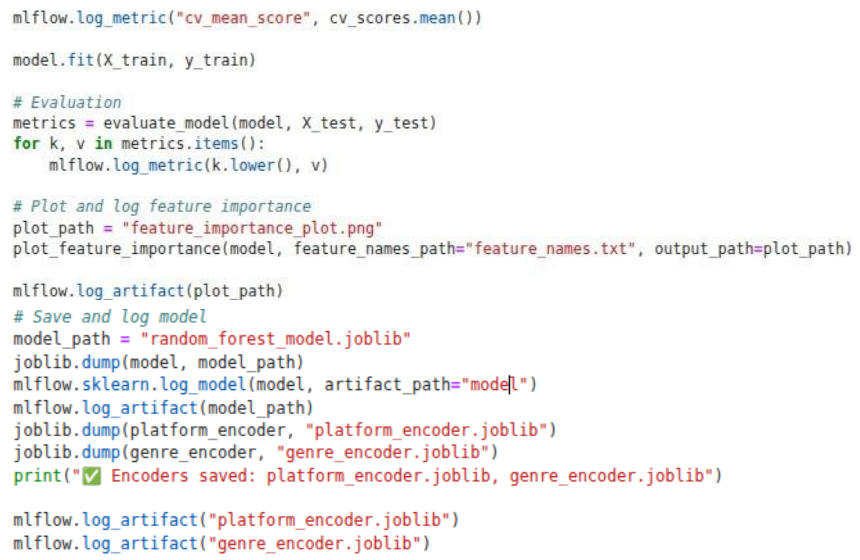
A RandomForestRegressor was trained using 5-fold cross-validation to assess stability, and the trained model was evaluated using metrics such as MAE, MSE and R^2. Feature importance was visualized and saved.



```
model = train_model(X_train, y_train)
evaluate_model(model, X_test, y_test)
plot_feature_importance(model, X_train.columns)
save_model(model)
```

Figure 41: Model training

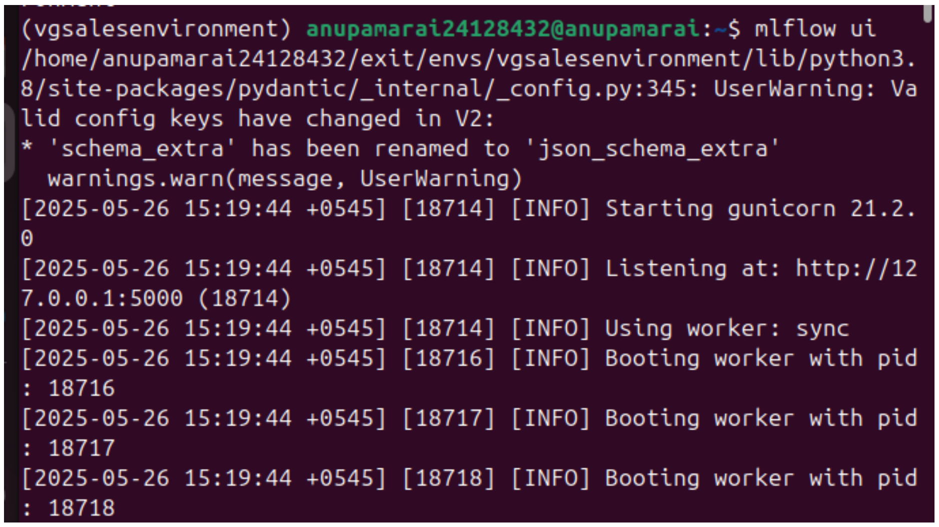
All metrics, artifacts (model file, plot, encoders) and parameters were tracked using MLflow for versioning and reproducibility.



```
mlflow.log_metric("cv_mean_score", cv_scores.mean())

model.fit(X_train, y_train)

# Evaluation
metrics = evaluate_model(model, X_test, y_test)
for k, v in metrics.items():
    mlflow.log_metric(k.lower(), v)

# Plot and log feature importance
plot_path = "feature_importance_plot.png"
plot_feature_importance(model, feature_names_path="feature_names.txt", output_path=plot_path)

mlflow.log_artifact(plot_path)
# Save and log model
model_path = "random_forest_model.joblib"
joblib.dump(model, model_path)
mlflow.sklearn.log_model(model, artifact_path="model")
mlflow.log_artifact(model_path)
joblib.dump(platform_encoder, "platform_encoder.joblib")
joblib.dump(genre_encoder, "genre_encoder.joblib")
print("✅ Encoders saved: platform_encoder.joblib, genre_encoder.joblib")

mlflow.log_artifact("platform_encoder.joblib")
mlflow.log_artifact("genre_encoder.joblib")
```

Figure 42: Logging in Mlflow

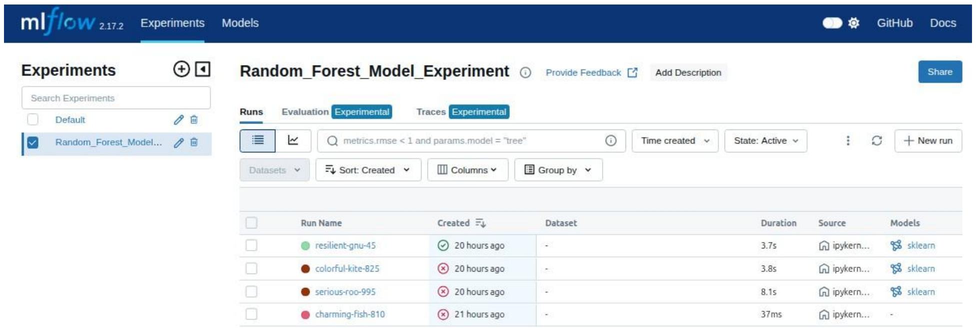
Then the loaded log can be observed through MLflow's UI, to start the UI the following command is executed in the terminal.



```
(vgsalesenvironment) anupamarai24128432@anupamarai:~$ mlflow ui
/home/anupamarai24128432/exit/envs/vgsalesenvironment/lib/python3.
8/site-packages/pydantic/_internal/_config.py:345: UserWarning: Va
lid config keys have changed in V2:
* 'schema_extra' has been renamed to 'json_schema_extra'
  warnings.warn(message, UserWarning)
[2025-05-26 15:19:44 +0545] [18714] [INFO] Starting gunicorn 21.2.
0
[2025-05-26 15:19:44 +0545] [18714] [INFO] Listening at: http://12
7.0.0.1:5000 (18714)
[2025-05-26 15:19:44 +0545] [18714] [INFO] Using worker: sync
[2025-05-26 15:19:44 +0545] [18716] [INFO] Booting worker with pid
: 18716
[2025-05-26 15:19:44 +0545] [18717] [INFO] Booting worker with pid
: 18717
[2025-05-26 15:19:44 +0545] [18718] [INFO] Booting worker with pid
: 18718
```

Figure 43: MlFlow UI is started

Then MlFLow is hosted in the localhost:5001.



Run Name	Created	Dataset	Duration	Source	Models
resilient-gnu-45	20 hours ago	-	3.7s	ipykern...	sklearn
colorful-kite-825	20 hours ago	-	3.8s	ipykern...	sklearn
serious-roo-995	20 hours ago	-	8.1s	ipykern...	sklearn
charming-fish-810	21 hours ago	-	37ms	ipykern...	-

Figure 44: Mlflow log in webserver

Full code and the explanation on why the evaluation metrics were used is available in Appendix 6.

6. Model Deployment and Monitoring

   After training and evaluating the Random Forest regression model, the deployment process ensures that the model is logged, saved, and made available for real-time predictions via FastAPI.

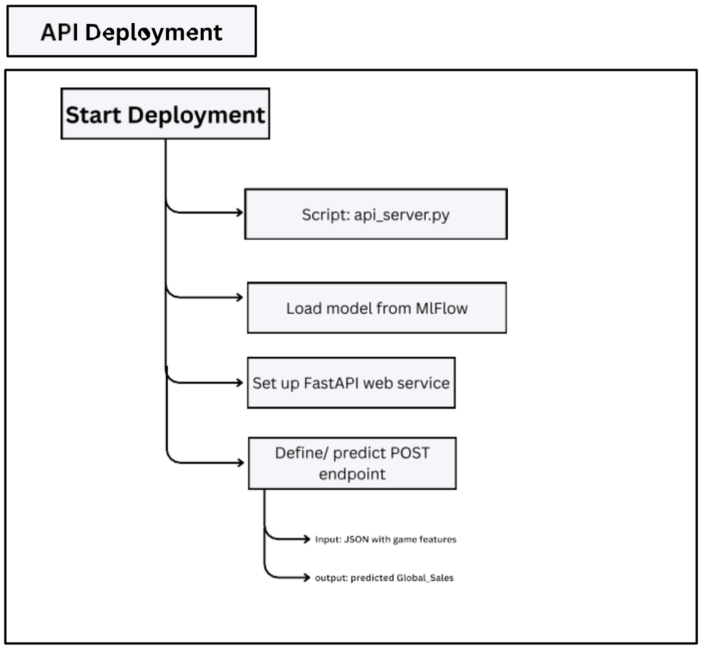
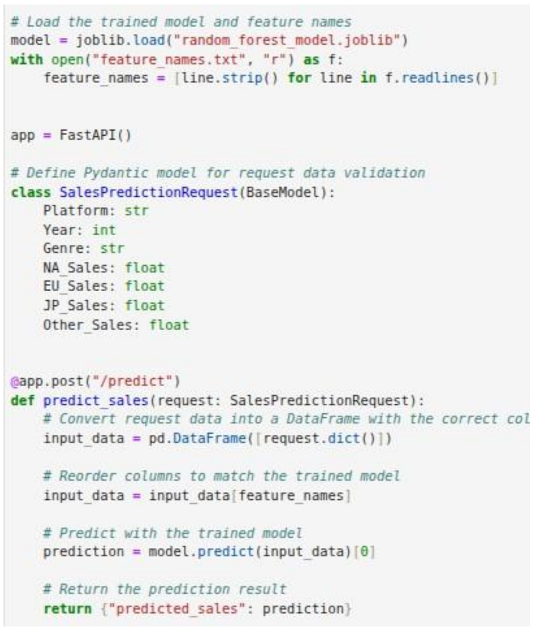

API Deployment
Start Deployment
Script: api_server.py
Load model from MlFlow
Set up FastAPI web service
Define/ predict POST endpoint
Input: JSON with game features
output: predicted Global_Sales

Figure 45: API deployment process

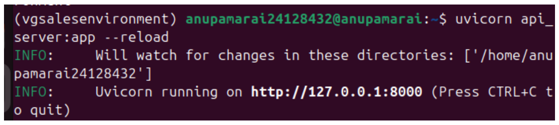
To expose the trained model as a REST API, we use FastAPI a modern Python web framework. It loads the model and accepts POST requests for predictions.



```python
# Load the trained model and feature names
model = joblib.load("random_forest_model.joblib")
with open("feature_names.txt", "r") as f:
    feature_names = [line.strip() for line in f.readlines()]


app = FastAPI()

# Define Pydantic model for request data validation
class SalesPredictionRequest(BaseModel):
    Platform: str
    Year: int
    Genre: str
    NA_Sales: float
    EU_Sales: float
    JP_Sales: float
    Other_Sales: float


@app.post("/predict")
def predict_sales(request: SalesPredictionRequest):
    # Convert request data into a DataFrame with the correct col
    input_data = pd.DataFrame([request.dict()])

    # Reorder columns to match the trained model
    input_data = input_data[feature_names]

    # Predict with the trained model
    prediction = model.predict(input_data)[0]

    # Return the prediction result
    return {"predicted_sales": prediction}
```

Figure 46: API code

Then the API is started locally.



```
(vgsalesenvironment) anupamarai24128432@anupamarai:~$ uvicorn api_
server:app --reload
INFO:     Will watch for changes in these directories: ['/home/anu
pamarai24128432']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C t
o quit)
```

Figure 47: Starting API server

The output and codes for FastAPI predictions is in Appendix 7.

The deployment steps are also typically orchestrated as tasks in an Airflow DAG. Each task can be monitored through the Airflow webserver logs.

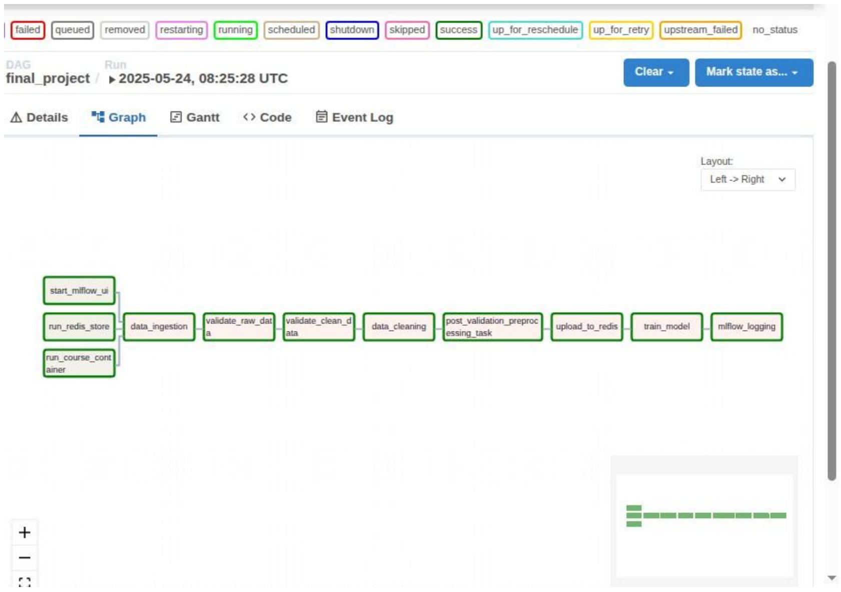
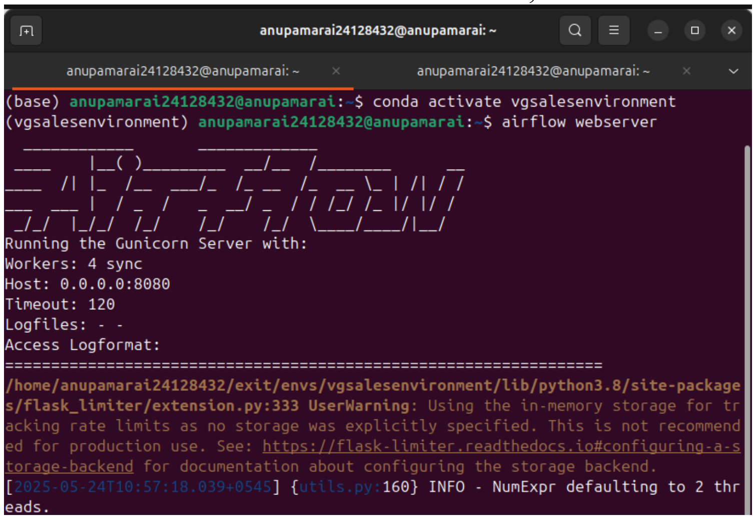


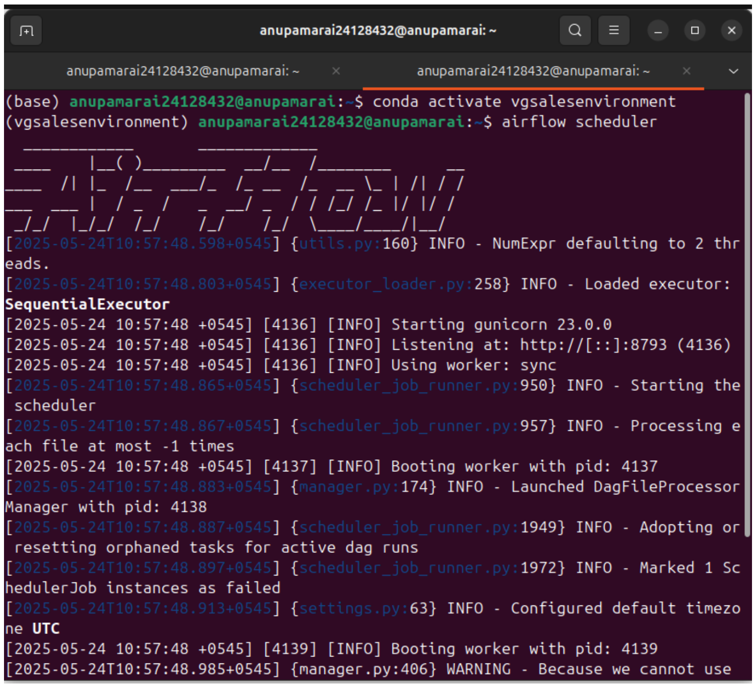
Figure 48: Airflow DAG logging the overall pipeline tasks

But before this to host it in web browsers, airflow webserver and scheduler needs to be set up.



```
(base) anupamarai24128432@anupamarai:~$ conda activate vgsalesenvironment
(vgsalesenvironment) anupamarai24128432@anupamarai:~$ airflow webserver
  ____________       _____________
 ____    |__( )_________  __/__  /________      __
____  /| |_  /__  ___/_  /_ __  /_  __ \_ | /| / /
___  ___ |  / _  /   _  __/ _  / / /_/ /_ |/ |/ /
 _/_/  |_/_/  /_/    /_/    /_/  \____/____/|__/
Running the Gunicorn Server with:
Workers: 4 sync
Host: 0.0.0.0:8080
Timeout: 120
Logfiles: - -
Access Logformat:
=================================================================
/home/anupamarai24128432/exit/envs/vgsalesenvironment/lib/python3.8/site-package
s/flask_limiter/extension.py:333 UserWarning: Using the in-memory storage for tr
acking rate limits as no storage was explicitly specified. This is not recommend
ed for production use. See: https://flask-limiter.readthedocs.io#configuring-a-s
torage-backend for documentation about configuring the storage backend.
[2025-05-24T10:57:18.039+0545] {utils.py:160} INFO - NumExpr defaulting to 2 thr
eads.
```

Figure 49: Starting airflow webserver



```
(base) anupamarai24128432@anupamarai:~$ conda activate vgsalesenvironment
(vgsalesenvironment) anupamarai24128432@anupamarai:~$ airflow scheduler
  ____________       _____________
 ____    |__( )_________  __/__  /________      __
____  /| |_  /__  ___/_  /_ __  /_  __ \_ | /| / /
___  ___ |  / _  /   _  __/ _  / / /_/ /_ |/ |/ /
 _/_/  |_/_/  /_/    /_/    /_/  \____/____/|__/
[2025-05-24T10:57:48.598+0545] {utils.py:160} INFO - NumExpr defaulting to 2 thr
eads.
[2025-05-24T10:57:48.803+0545] {executor_loader.py:258} INFO - Loaded executor:
SequentialExecutor
[2025-05-24 10:57:48 +0545] [4136] [INFO] Starting gunicorn 23.0.0
[2025-05-24 10:57:48 +0545] [4136] [INFO] Listening at: http://[::]:8793 (4136)
[2025-05-24 10:57:48 +0545] [4136] [INFO] Using worker: sync
[2025-05-24T10:57:48.865+0545] {scheduler_job_runner.py:950} INFO - Starting the
 scheduler
[2025-05-24T10:57:48.867+0545] {scheduler_job_runner.py:957} INFO - Processing e
ach file at most -1 times
[2025-05-24 10:57:48 +0545] [4137] [INFO] Booting worker with pid: 4137
[2025-05-24T10:57:48.883+0545] {manager.py:174} INFO - Launched DagFileProcessor
Manager with pid: 4138
[2025-05-24T10:57:48.887+0545] {scheduler_job_runner.py:1949} INFO - Adopting or
 resetting orphaned tasks for active dag runs
[2025-05-24T10:57:48.897+0545] {scheduler_job_runner.py:1972} INFO - Marked 1 Sc
hedulerJob instances as failed
[2025-05-24T10:57:48.913+0545] {settings.py:63} INFO - Configured default timezo
ne UTC
[2025-05-24 10:57:48 +0545] [4139] [INFO] Booting worker with pid: 4139
[2025-05-24T10:57:48.985+0545] {manager.py:406} WARNING - Because we cannot use
```

Figure 50: Starting airflow scheduler

Data Storage Plan

1. Physical and Logical Storage Structure

   The data storage system utilized for this project was established with MariaDB operating within a Docker container (course_container) on Ubuntu Linux platform. The original data was sourced from various CSV files located in the vgsales_project directory within a Conda environment (vgsales_environment). After ingestion, these CSV files were analyzed and organized into a relational format that is appropriate for analytical tasks.

2. Rationale for Star Schema Adoption

   To enable efficient querying, feature engineering, and integration with subsequent ML processes, the ingested data was structured into a star schema within MariaDB. This scheme comprises a central fact table along with multiple dimension tables. It was crafted to minimize complexity in analytics queries while accommodating scalable data operations. It was specifically implemented because of its following features:

Query Simplicity	As Star schemas are made up of a central fact table, with many dimension tables linking to it. This organization allows SQL queries to slice and dice, filter, group and aggregate data elements (for example, by platform, genre or year), promotes rapid slicing, aggregation, and filtering of data for both business intelligence and machine learning.
Optimized for Read Performance	Well-suited for a project that is often read from the database for model training and validation as star schema is built for read-heavy tasks. It also performs well in data analytics environments where the main work is analyzing data rather than editing it. Fewer queries are needed to get data because everything is denormalized which improves speed when handling larger amounts of data. It supports OLAP tasks (like summing up, digging down, comparing) which are vital for model creation, displaying dashboards and handling instant analysis. Looking up Genre_ID, Platform_ID and Year as indexed keys cut down query processing time, most noticeably with the Video Game Sales dataset.
ML Integration	Star schema is efficient during the feature engineering and preprocessing steps when used with machine learning. The way data is arranged supports easy identification of things like the average sales in each genre, trends over decades or popularity grouped by publishers. Because every categorical variable (Genre or Publisher) is in its own dimension table, you can use one-hot, label or frequency encoding for them without adding data to the core table. Having features and data clearly separated in ML pipelines reduces data copying and makes it simpler to do feature selection, train models and evaluate them.
Entity Separation	An important point is that star schemas logically divide business entities which enhances how modular, easy-to-understand and easy-to-maintain the database is. All the main categories (genres, publishers, platforms) are placed in dimension tables which makes it easier to manage specific details in the data. Doing it this way ensures there's no need to update every data record for a genre or platform, as changes happen only in one place in the advanced table.

Table 7: Rationale for star schema adoption

3. Potential Drawbacks and Alternatives

Despite its advantages, the star schema introduces some data redundancy due to denormalization. So, for larger and more volatile datasets, this may impact storage efficiency and update operations. Two alternative models that were considered:

Normalized Relational Model	Its scalability and faithful upholding of data integrity make Normalized Relational Model well-known. Putting data into many linked tables prevents repeated information and makes certain everything is correct throughout the database. Even so, using this model can make queries more complex, mainly when used for analysis, as you may have to join several different tables. A machine learning pipeline relies on rapid access and transformation of data may not perform as desired when complex joins are involved.
One Big Table	One Big Table is also an alternative to star schema, where facts and dimensions are all put into a large, flat table. A simple structure helps when just looking around or making basic visualizations and it can be beneficial in the early phases of prototyping or with minimal data. But its lack of scalability makes it unsuitable for managing big data analytics. Because of this, there is often extra data stored, storage becomes inefficient and it becomes difficult to do quick changes or separate different things for targeted study.

Table 8: Potential Drawbacks and Alternatives

4. Star Schema Structure

Fact Table:

Column	**Description**
Game_ID(PK)	Unique identifier for the game
NA_Sales	Sales in North America
EU_Sales	Sales in Europe
JP_Sales	Sales in Japan
Other_Sales	Sales in other regions
Global_Sales	Total sales globally
Year	Year of release (FK to Time)
Platform_ID	Foreign key to Platform table
Genre_ID	Foreign key to Genre table
Publisher_ID	Foreign key to Publisher table

Table 9: Fact table

Dimension Tables:

1. Platform (Dimension)	
Column	Description
Platform_ID(PK)	Unique ID for each platform
Platform_Name	For example, PS2, Xbox, Switch
2. Genre (Dimension)	
Column	Description
Genre_ID(PK)	Unique ID for each genre
Genre_Name	For example, Action, RPG

3. Publisher (Dimension)	
Column	Description
Publisher_ID(PK)	Unique ID for each publisher
Publisher_Name	For example, Nintendo, Ubisoft
4. Time (Dimension)	
Column	Description
Year (PK)	Year of release
Decade	Grouping (For example, 2000s.)
Quarter	For example, Q1-Q4

Table 10: Dimension table

5. Comparison of OLTP and OLAP Roles in the Data Analytics Pipeline

Aspect	**OLTP (Online Transaction Processing)**	**OLAP (Online Analytical Processing)**
Purpose	Manages day-to-day transactional tasks with frequent read and write operations.	Handles complex analytical queries for decision-making and data exploration.
Role in Pipeline	Used to store and manage preprocessed, structured data.	Used to perform analysis, modeling, and evaluation on large volumes of data.
Technology Used	'MariaDB' is used for consistent relational data storage.	Pandas, Numpy, Scikit-learn are used for in-memory data processing and machine learning.
Typical Operations	Inserting clean records, updating tables, querying small data subjects.	Aggregations, comparisons, statistical transformation, model training and cross validation.
Data Access Pattern	Transactional (meaning row-level operations)	Analytical (meaning columnar or batch-level operations)
Performance Optimization	Focused on speed and reliability of insert/update operations.	Focused on high throughput read performance and analytical depth.
Additional Enhancement	Redis is used for temporary data caching to optimize data access speed without hitting the database.	Redis supports fast access to pre-split datasets, improving pipeline responsiveness and analytical efficiency.
Integration Purpose	Supports consistent ingestion and storage of pipeline-ready data.	Enables feature engineering, evaluation, visualization, and advanced modeling logic.
Design Strategy	Ensures transactional consistency for operational processes.	Enables analytical scalability by separating heavy computing tasks from operational systems.

Table 11: Comparison of OLTP and OLAP

Considerations regarding Ethic, Privacy and Legality

While conducting all the above processes, be it raw data collection, or model deployment, there is a concern regarding ethics and privacy, which should be addressed properly beforehand. This section consists of the following areas of concern:

1. Data Privacy and Legal Compliance
   The data set utilized in this project was sourced from Kaggle and is accessible to the public for academic and non-commercial purposes. This guarantees adherence to open data usage regulations. Notably, the dataset does not include any personally identifiable information (PII), which greatly mitigates privacy concerns. Although the dataset does not directly pertain to user data, ethical data management practices were maintained throughout the process to avert misuse or unintended reidentification. Furthermore, complete attribution to data sources is included in the References section to uphold transparency and honor intellectual property rights.

2. Security Measures
   All elements of the pipeline are encapsulated using Docker to provide a secure and isolated execution environment. Sensitive tasks such as database access are safeguarded through role-based access controls, secured environment variables, and Docker network isolation. Redis, which is employed for intermediate data transfer, is set up not to retain data on disk, thereby reducing exposure risk. Logging, error tracking, and Airflow DAG audit trails further enhance secure and traceable workflow execution.

3. Bias and Fairness
   Although the dataset offers a broad spectrum of historical video game data, it displays an inherent bias towards older and more established platforms like the PlayStation 2 and Wii, while newer consoles and niche genres are underrepresented. This can distort predictive results in favor of the overrepresented categories. To counter this, data preprocessing measures were implemented, including techniques to diminish sampling bias, such as excluding exceedingly rare categories, balancing class distributions when feasible, and monitoring model performance across various segments. These actions contribute to ensuring that the model remains fair, generalizable, and relevant for contemporary gaming contexts.

4. Ethics in Model Deployment
   The ethical deployment of machine learning models necessitates that their outcomes facilitate equitable and informed decision-making. In this project, the predictive model

Essay: Understanding Differential Privacy and Its Real-World Applications

1. Differential Privacy

   Differential Privacy lets organizations learn from datasets with personal information, but it reduces the chances that people can be singled out. Cynthia Dwork and her colleagues described differential privacy in a formal paper in 2006 which ensures that a data analysis output will not change much regardless of how much data from any one person is used (Dwork *et al.*, 2006). To sum up, it makes it impossible for anyone's data to be recovered from the grouped information, so individuals' privacy is defended.

The principle is to add a specific level of noise to information or outcomes, making it hard to notice if anyone was there or not. Because it both serves a purpose and maintains privacy, differential privacy has a big role in modern data science as we deal with more data being collected and concerns about being watched and misused.

2. Key Principles of Differential Privacy

   - Epsilon (ε) is a major factor that determines the level of privacy in any system. Having smaller values protects privacy but the results might not be as accurate.

   - One technique seen in differential privacy is to add random noise taken from Laplace or Gaussian distributions to the outputs of queries or models.

   - Once the data has differential privacy applied, any other processing done does not compromise privacy.

3. Real-World Applications

   Differential privacy has transitioned from theory to actual practical use by some large companies and government bodies:

   - Apple: Starting with iOS 10, any user behavior data (e.g., emoji usage, Safari crashes) collected by Apple through differential privacy remains anonymous. Through end-to-end encryption, Apple can update how its services work without letting anyone else access private information (Apple, 2020).

   - Chrome web browser and the analytics Google uses internally are protected by differential privacy. Using RAPPOR, an early implementation, technology developers could track people's use of data but did not expose their identities (Erlingsson, Pihur and Korolova, no date).

   - The 2020 U.S. Census was the first census where respondent privacy was protected through differential privacy by the Census Bureau. To stop individuals from being identified from the published results, the bureau mixed noise into the data in areas with few people (Abowd, 2018). It was a breakthrough on how governments handle the relationship between transparency and safeguarding data.

   - Microsoft dahlver use differential privacy in reaching products like SmartNoise for scientists to work with private data without putting it at risk.

4. Benefits and Challenges

   Differential privacy stands out mainly because it provides a formal privacy promise. It helps ensure the right use of data and obeys data protection rules such as GDPR. Such an approach is useful for machine learning because using real data for training could expose people to re-identification. But still, there are obstacles. More noise added to the output of a differentially private algorithm can decrease its accuracy. In addition, determining the correct ε-value means finding a fair balance between someone's utility and their need for privacy which can be tough with real-time systems.

Ethical and Legal Reflections on the Data Pipeline

Based on my research, what ethical and legal concerns are most relevant to how I am currently handling and processing data within my pipeline?	Fairness and representation of data are the biggest ethical issues in the current pipeline. Due to the fact that old consoles like PS2 and Wii are the main ones in the dataset, Action and Sports genres are also the ones that receive the most focus. Because of this, results from these models might not represent new games or genres very well. Even though no PII is included in the dataset, it is still necessary to follow the rules for using the data, because the dataset came from Kaggle. Citation and transparency about the data were important and using the data solely for academic purposes were required. Furthermore, the obligation to prevent bias or false results in the predictive outputs made by the data pipeline is a major ethical responsibility.
Within my prototype system, what steps could I take to enhance the security and privacy of my data analytics pipeline?	There are many extra things that can be done along the pipeline to improve data security. It would be beneficial if Docker's infrastructure included, for example, the secure storage of environment variables and regular scanning for vulnerabilities in images. By adding more secure authentication and encrypted traffic routes between Redis and MariaDB, the chances of losing or leaking sensitive data could be reduced. Even if the current dataset does not have any sensitive data, producing synthetic data or using differential privacy approaches in further versions will support using real user data in the future. Also, RBAC in Airflow and Docker with audit logging allows you to follow model activities and lower exposure risks when models are used.

Appendices

Appendix 1: (Justification on usage of tools and architectures)

Docker: Docker helps put services into containers, allowing for the same environment settings during development and production. This is mostly helpful when trying to imitate real-life microservice designs where various parts (like databases, APIs, etc.) work in different containers.

Apache Airflow: Apache Airflow was installed to orchestrate the pipeline tasks by using Directed Acyclic Graphs (DAGs). This tool allows for the automation and scheduling of the various stages in the pipeline, checking dependencies as well as providing very robust error handling and logging capabilities through its user interface.

MariaDB: MariaDB ColumnStore was used as the analytics database because it performs well with analytical queries and supports SQL standards. It stores the ingested and transformed data, making querying during preprocessing and analysis efficient.

Redis: Redis served as a fast in-memory data store to temporarily hold intermediate datasets between tasks. This helped overcome Airflow's limitation of not being able to pass large data directly between tasks. DataFrames were serialized with PyArrow before being stored in Redis.

MLflow: MLflow tracked machine learning experiments, models, and their related parameters and metrics. It offered version-controlled storage for trained models and ensured experiments could be reproduced.

FastAPI: FastAPI was chosen to deploy the trained model as a RESTful web service. Its asynchronous features and automatic documentation through Swagger UI made it ideal for lightweight and scalable APIs.

Pandas: It is a python library used for data manipulation and analysis. It handles CSV input and output, Dataframe transformations, filtering and integration with other tools like pyarrow, sqlalchemy, and sklearn.

Numpy: Numpy is a fundamental package for numerical computing in Python, which is used to fill missing values, generate Gaussian noise and numerical computations during preprocessing and model evaluation.

PyArrow: It is Apache Arrow's Python interface for high-speed, columnar data serialization, which enables fast serialization of pandas DataFrames for storing and retrieving from Redis.

SqlAlchemy: It is a SQL toolkit and ORM for Python which is used to connect pandas DataFrames to MariaDB to upload cleaned datasets into structured SQL tables.

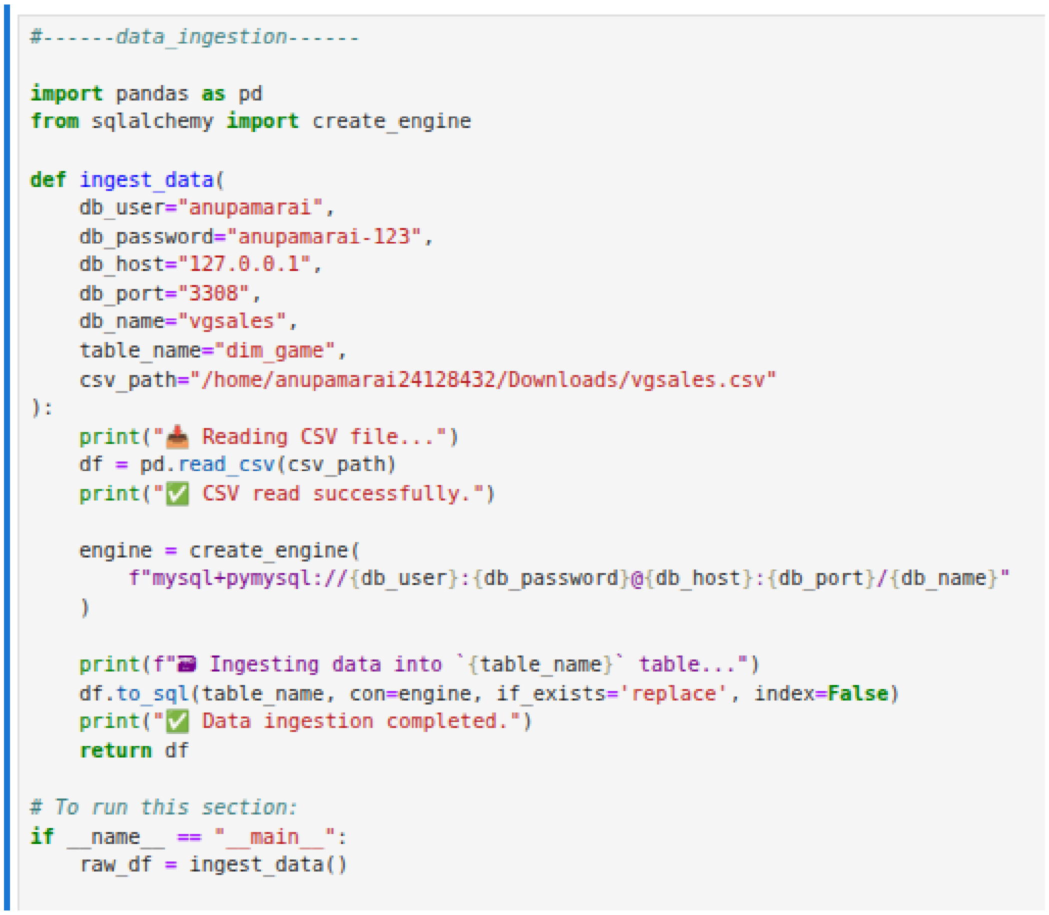
Joblib: Joblib is a lightweight library for saving and loading Python objects efficiently which is used to persist the trained model and encoders to disk for later use in deployment.

Matplotlib.pyplot: It is a plotting library for creating static visualizations, which generate and save feature important graphs for model interpretability.

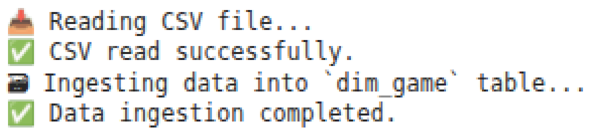
Pydantic: Pydantic is a data validation library that uses Python type annotations, to ensure incoming prediction requests in the FastAPI server are type-checked and validated before inference.

Great_Expectations: It is an open-source tool for data quality validation, profiling, and documentation to automate data validation checks, ensuring clean data.

Appendix 2: (Data ingestion process's code and output)

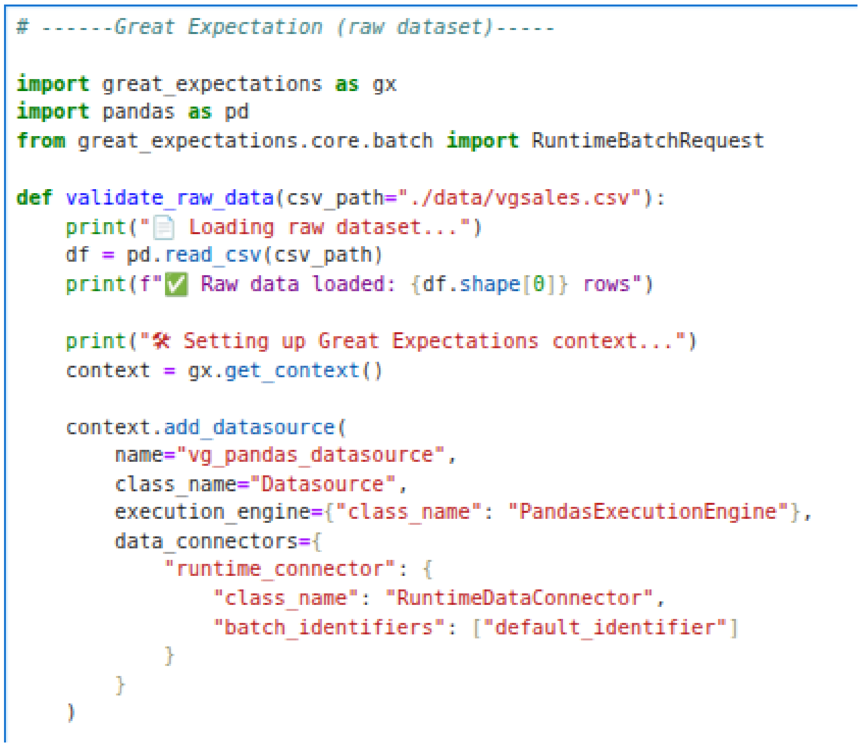


```python
#------data_ingestion------

import pandas as pd
from sqlalchemy import create_engine

def ingest_data(
    db_user="anupamarai",
    db_password="anupamarai-123",
    db_host="127.0.0.1",
    db_port="3308",
    db_name="vgsales",
    table_name="dim_game",
    csv_path="/home/anupamarai24128432/Downloads/vgsales.csv"
):
    print("📥 Reading CSV file...")
    df = pd.read_csv(csv_path)
    print("✅ CSV read successfully.")

    engine = create_engine(
        f"mysql+pymysql://{db_user}:{db_password}@{db_host}:{db_port}/{db_name}"
    )

    print(f"🗃 Ingesting data into `{table_name}` table...")
    df.to_sql(table_name, con=engine, if_exists='replace', index=False)
    print("✅ Data ingestion completed.")
    return df

# To run this section:
if __name__ == "__main__":
    raw_df = ingest_data()
```

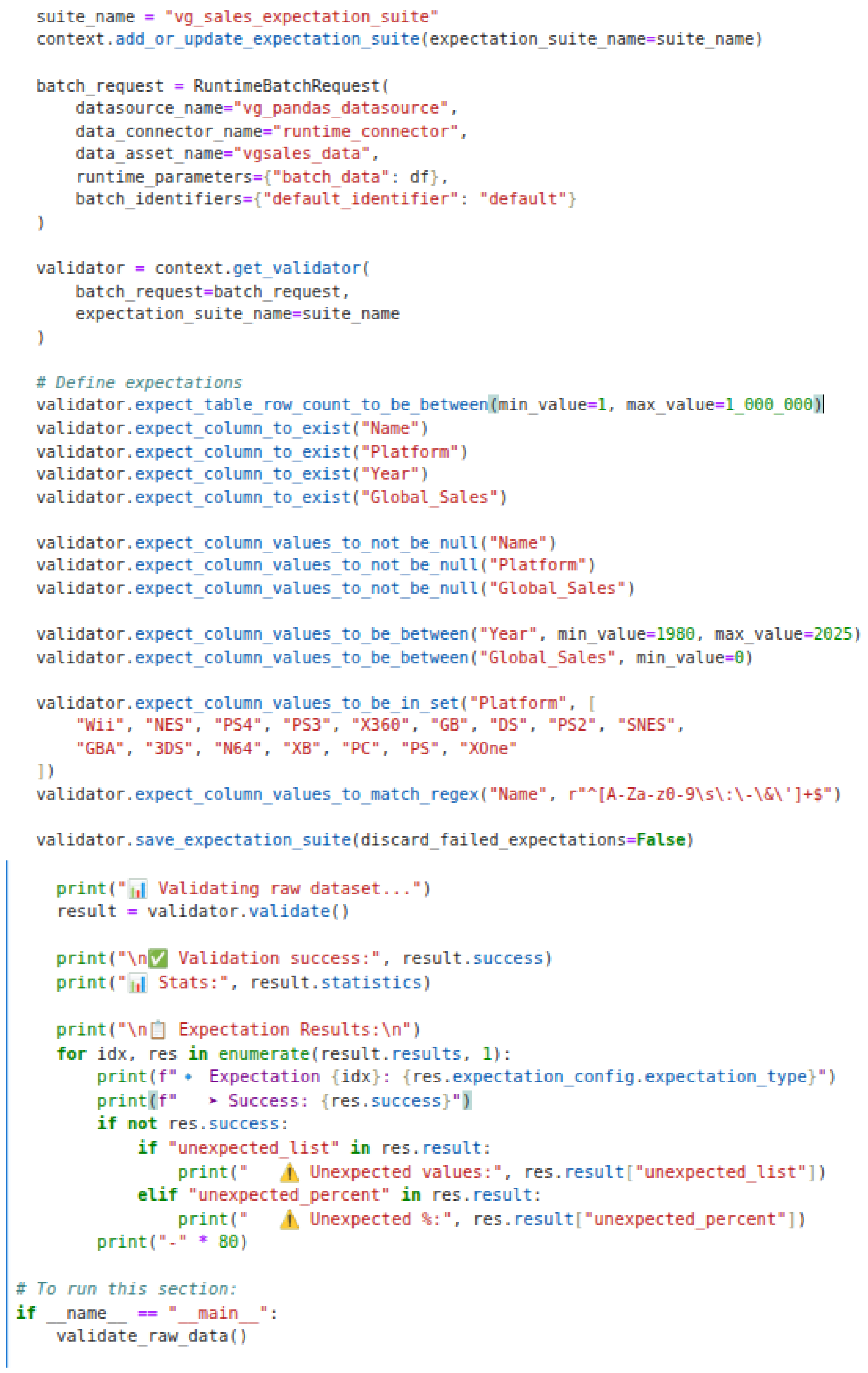


```
📥 Reading CSV file...
✅ CSV read successfully.
🗃 Ingesting data into `dim_game` table...
✅ Data ingestion completed.
```

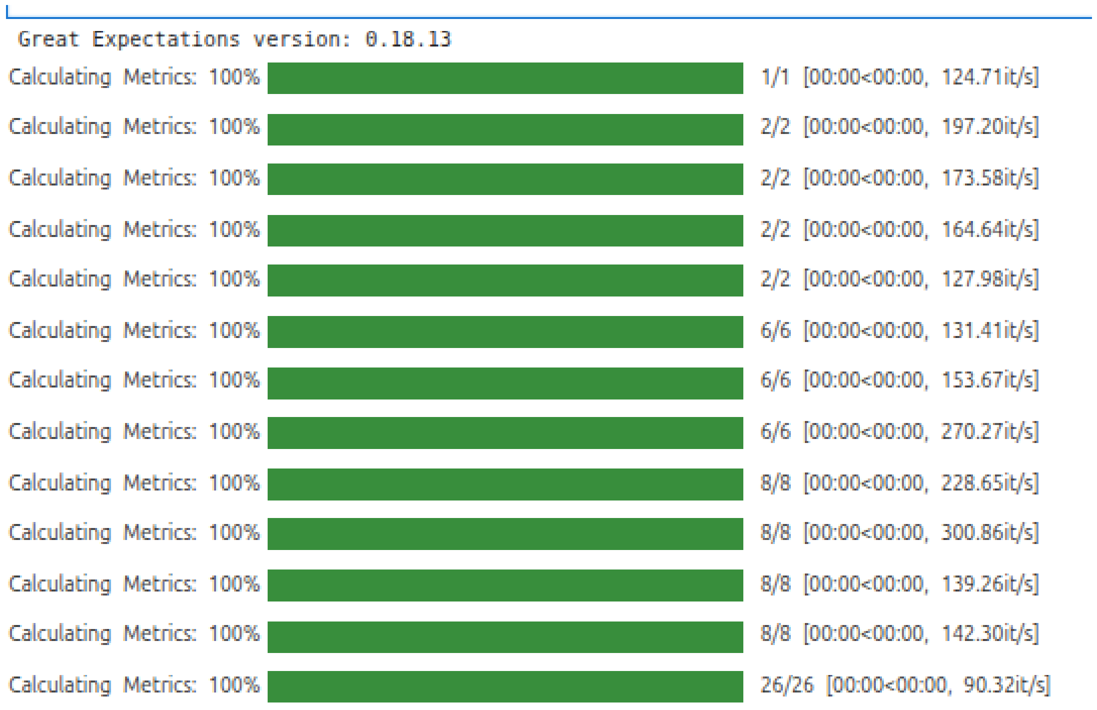
Appendix 3: (Data validation process's code and output)



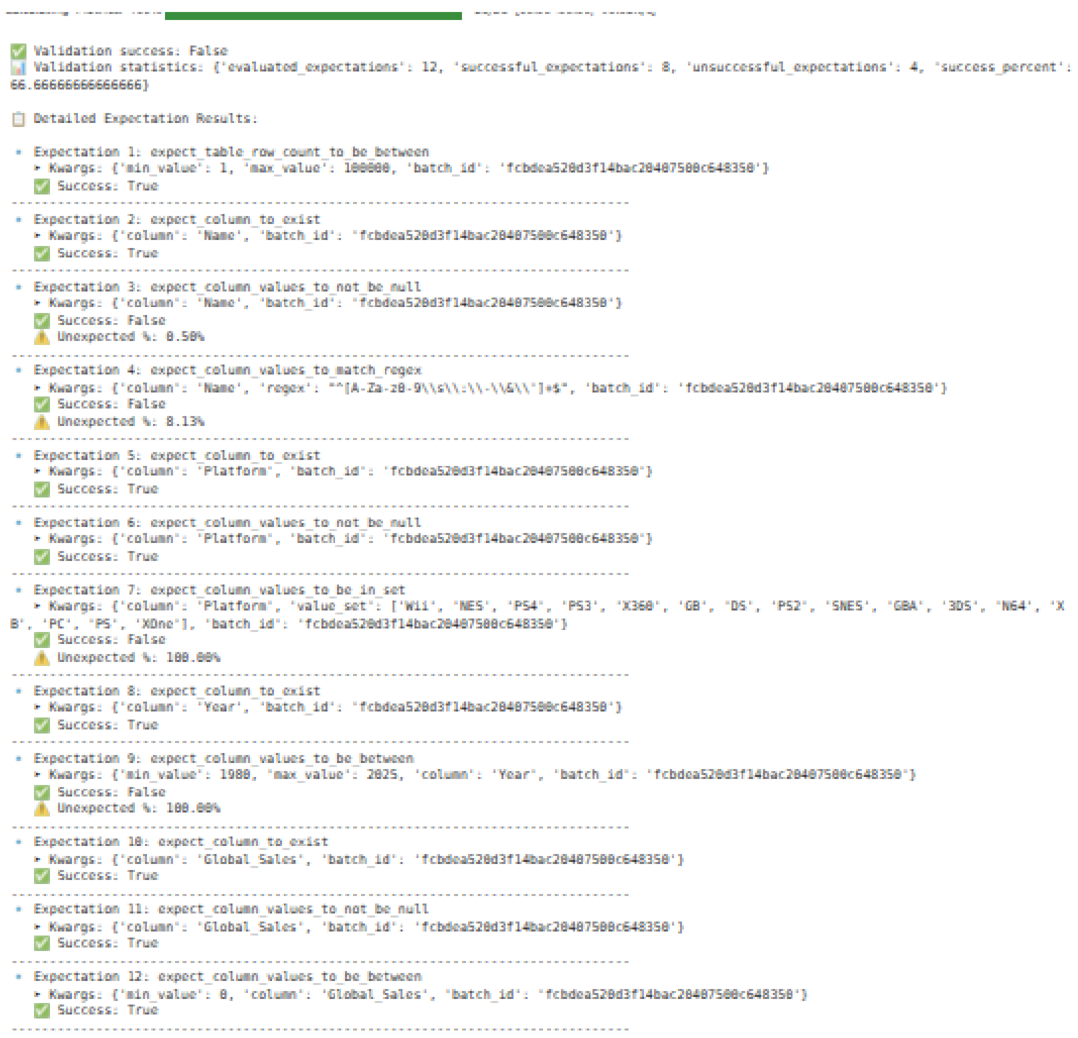
```
# ------Great Expectation (raw dataset)-----

import great_expectations as gx
import pandas as pd
from great_expectations.core.batch import RuntimeBatchRequest

def validate_raw_data(csv_path="./data/vgsales.csv"):
    print("📄 Loading raw dataset...")
    df = pd.read_csv(csv_path)
    print(f"✅ Raw data loaded: {df.shape[0]} rows")

    print("🛠 Setting up Great Expectations context...")
    context = gx.get_context()

    context.add_datasource(
        name="vg_pandas_datasource",
        class_name="Datasource",
        execution_engine={"class_name": "PandasExecutionEngine"},
        data_connectors={
            "runtime_connector": {
                "class_name": "RuntimeDataConnector",
                "batch_identifiers": ["default_identifier"]
            }
        }
    )
```



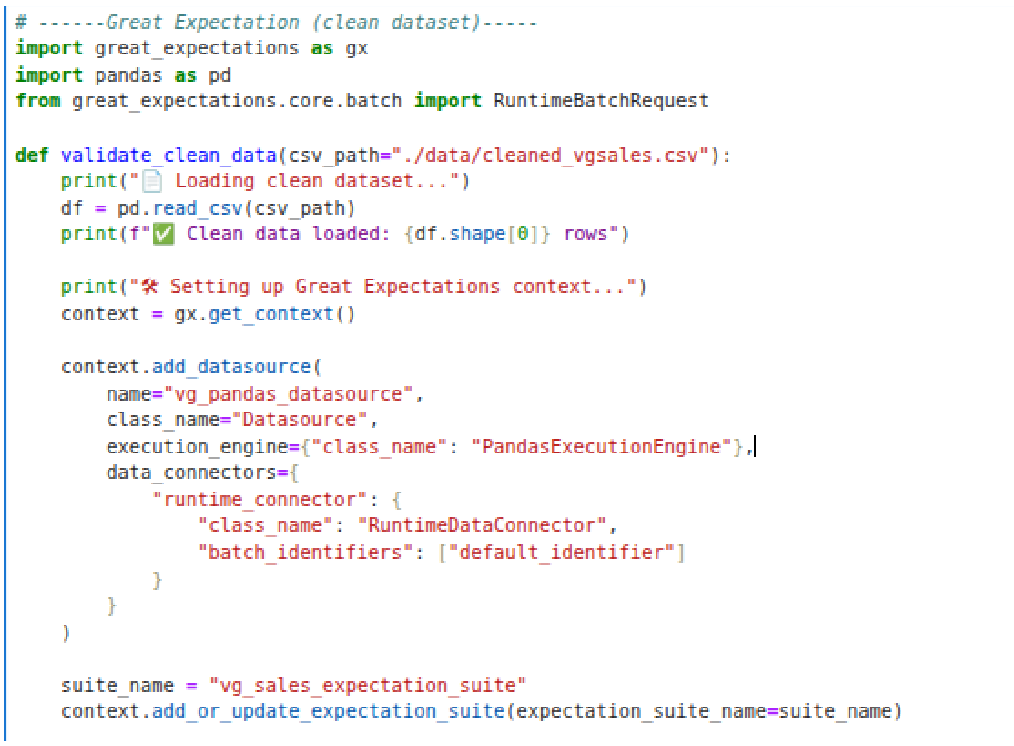
```python
    suite_name = "vg_sales_expectation_suite"
    context.add_or_update_expectation_suite(expectation_suite_name=suite_name)

    batch_request = RuntimeBatchRequest(
        datasource_name="vg_pandas_datasource",
        data_connector_name="runtime_connector",
        data_asset_name="vgsales_data",
        runtime_parameters={"batch_data": df},
        batch_identifiers={"default_identifier": "default"}
    )

    validator = context.get_validator(
        batch_request=batch_request,
        expectation_suite_name=suite_name
    )

    # Define expectations
    validator.expect_table_row_count_to_be_between(min_value=1, max_value=1_000_000)
    validator.expect_column_to_exist("Name")
    validator.expect_column_to_exist("Platform")
    validator.expect_column_to_exist("Year")
    validator.expect_column_to_exist("Global_Sales")

    validator.expect_column_values_to_not_be_null("Name")
    validator.expect_column_values_to_not_be_null("Platform")
    validator.expect_column_values_to_not_be_null("Global_Sales")

    validator.expect_column_values_to_be_between("Year", min_value=1980, max_value=2025)
    validator.expect_column_values_to_be_between("Global_Sales", min_value=0)

    validator.expect_column_values_to_be_in_set("Platform", [
        "Wii", "NES", "PS4", "PS3", "X360", "GB", "DS", "PS2", "SNES",
        "GBA", "3DS", "N64", "XB", "PC", "PS", "XOne"
    ])
    validator.expect_column_values_to_match_regex("Name", r"^[A-Za-z0-9\s\:\-\&\']+$")

    validator.save_expectation_suite(discard_failed_expectations=False)

    print("📊 Validating raw dataset...")
    result = validator.validate()

    print("\n✅ Validation success:", result.success)
    print("📊 Stats:", result.statistics)

    print("\n📋 Expectation Results:\n")
    for idx, res in enumerate(result.results, 1):
        print(f"• Expectation {idx}: {res.expectation_config.expectation_type}")
        print(f"   ➤ Success: {res.success}")
        if not res.success:
            if "unexpected_list" in res.result:
                print("   ⚠️ Unexpected values:", res.result["unexpected_list"])
            elif "unexpected_percent" in res.result:
                print("   ⚠️ Unexpected %:", res.result["unexpected_percent"])
        print("-" * 80)

# To run this section:
if __name__ == "__main__":
    validate_raw_data()
```



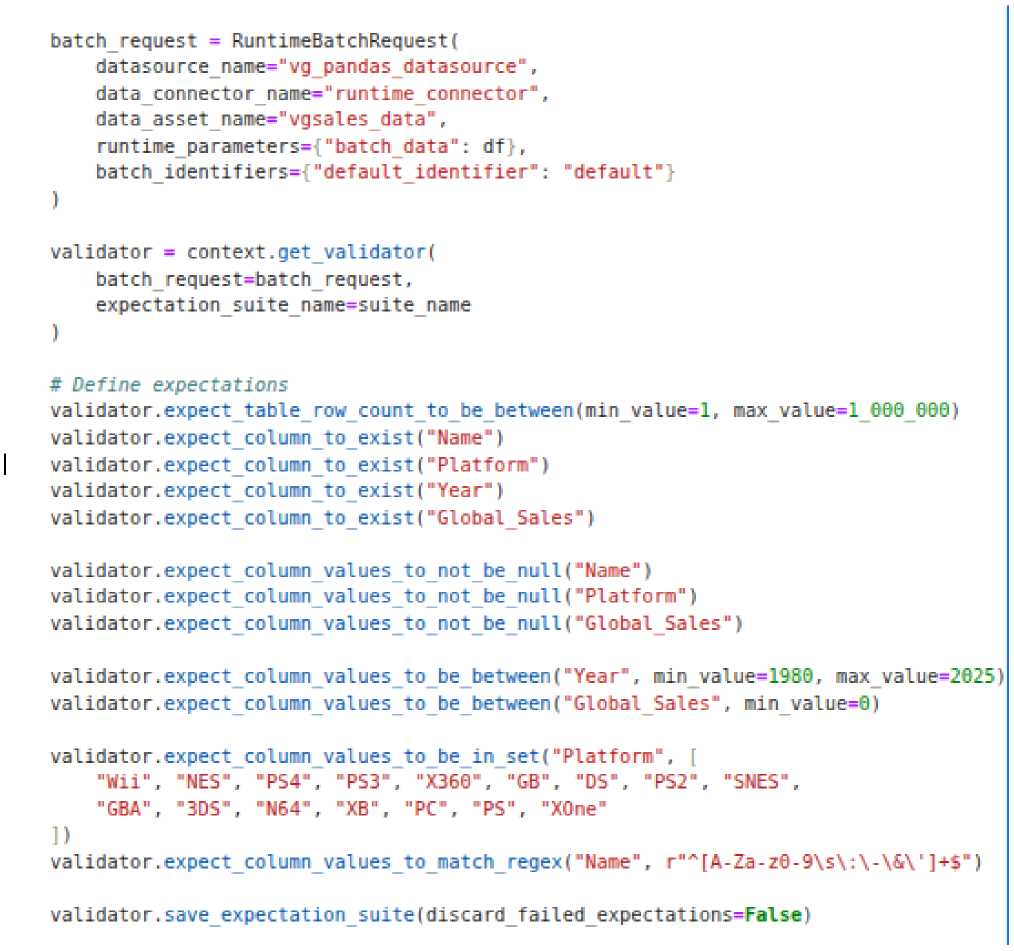
```
Great Expectations version: 0.18.13
Calculating Metrics: 100%  1/1 [00:00<00:00, 124.71it/s]
Calculating Metrics: 100%  2/2 [00:00<00:00, 197.20it/s]
Calculating Metrics: 100%  2/2 [00:00<00:00, 173.58it/s]
Calculating Metrics: 100%  2/2 [00:00<00:00, 164.64it/s]
Calculating Metrics: 100%  2/2 [00:00<00:00, 127.98it/s]
Calculating Metrics: 100%  6/6 [00:00<00:00, 131.41it/s]
Calculating Metrics: 100%  6/6 [00:00<00:00, 153.67it/s]
Calculating Metrics: 100%  6/6 [00:00<00:00, 270.27it/s]
Calculating Metrics: 100%  8/8 [00:00<00:00, 228.65it/s]
Calculating Metrics: 100%  8/8 [00:00<00:00, 300.86it/s]
Calculating Metrics: 100%  8/8 [00:00<00:00, 139.26it/s]
Calculating Metrics: 100%  8/8 [00:00<00:00, 142.30it/s]
Calculating Metrics: 100%  26/26 [00:00<00:00, 90.32it/s]
```



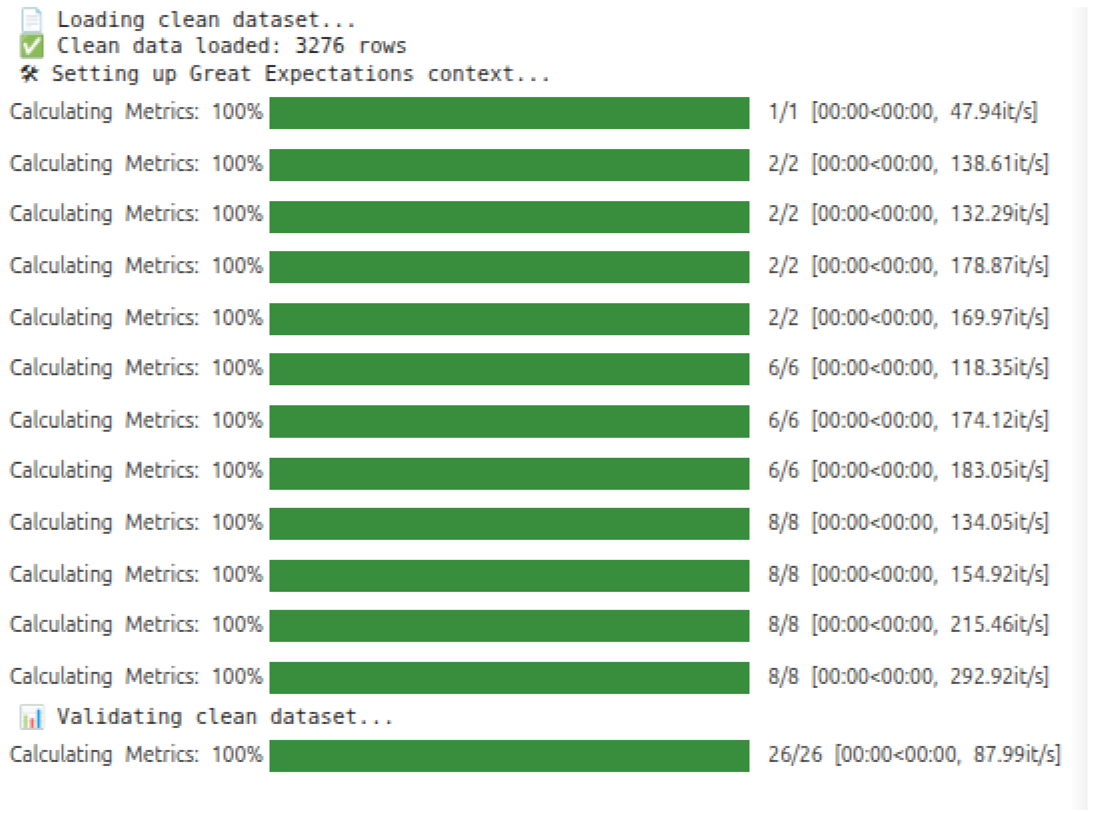
```
[illegible]

✅ Validation success: False
📊 Validation statistics: {'evaluated_expectations': 12, 'successful_expectations': 8, 'unsuccessful_expectations': 4, 'success_percent':
66.66666666666666}

📋 Detailed Expectation Results:

 • Expectation 1: expect_table_row_count_to_be_between
   ▸ Kwargs: {'min_value': 1, 'max_value': 100000, 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: True
--------------------------------------------------------------------------------
 • Expectation 2: expect_column_to_exist
   ▸ Kwargs: {'column': 'Name', 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: True
--------------------------------------------------------------------------------
 • Expectation 3: expect_column_values_to_not_be_null
   ▸ Kwargs: {'column': 'Name', 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: False
   ⚠️ Unexpected %: 0.50%
--------------------------------------------------------------------------------
 • Expectation 4: expect_column_values_to_match_regex
   ▸ Kwargs: {'column': 'Name', 'regex': "^[A-Za-z0-9\\s\\:\\-\\&\\']+$", 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: False
   ⚠️ Unexpected %: 8.13%
--------------------------------------------------------------------------------
 • Expectation 5: expect_column_to_exist
   ▸ Kwargs: {'column': 'Platform', 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: True
--------------------------------------------------------------------------------
 • Expectation 6: expect_column_values_to_not_be_null
   ▸ Kwargs: {'column': 'Platform', 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: True
--------------------------------------------------------------------------------
 • Expectation 7: expect_column_values_to_be_in_set
   ▸ Kwargs: {'column': 'Platform', 'value_set': ['Wii', 'NES', 'PS4', 'PS3', 'X360', 'GB', 'DS', 'PS2', 'SNES', 'GBA', '3DS', 'N64', 'X
B', 'PC', 'PS', 'XOne'], 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: False
   ⚠️ Unexpected %: 100.00%
--------------------------------------------------------------------------------
 • Expectation 8: expect_column_to_exist
   ▸ Kwargs: {'column': 'Year', 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: True
--------------------------------------------------------------------------------
 • Expectation 9: expect_column_values_to_be_between
   ▸ Kwargs: {'min_value': 1980, 'max_value': 2025, 'column': 'Year', 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: False
   ⚠️ Unexpected %: 100.00%
--------------------------------------------------------------------------------
 • Expectation 10: expect_column_to_exist
   ▸ Kwargs: {'column': 'Global_Sales', 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: True
--------------------------------------------------------------------------------
 • Expectation 11: expect_column_values_to_not_be_null
   ▸ Kwargs: {'column': 'Global_Sales', 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: True
--------------------------------------------------------------------------------
 • Expectation 12: expect_column_values_to_be_between
   ▸ Kwargs: {'min_value': 0, 'column': 'Global_Sales', 'batch_id': 'fcbdea520d3f14bac20407500c648350'}
   ✅ Success: True
--------------------------------------------------------------------------------
```



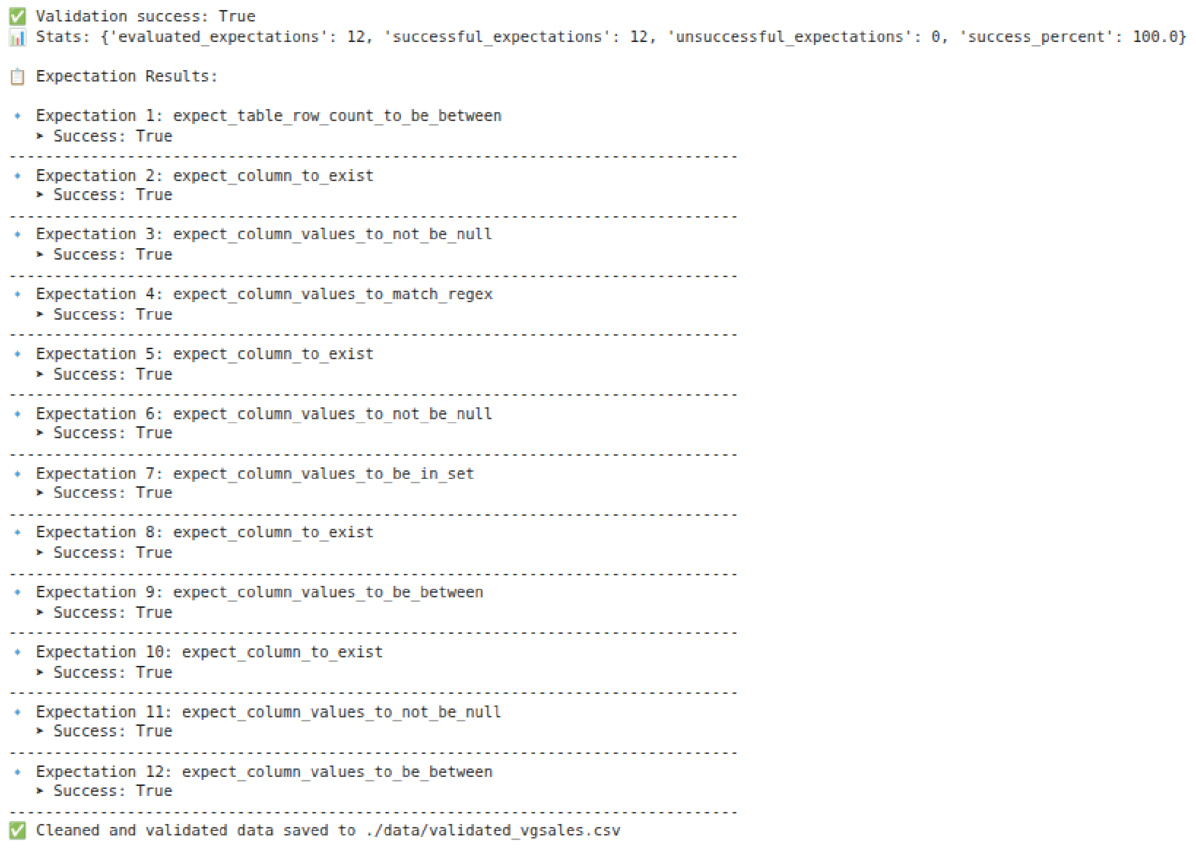
```python
# ------Great Expectation (clean dataset)-----
import great_expectations as gx
import pandas as pd
from great_expectations.core.batch import RuntimeBatchRequest

def validate_clean_data(csv_path="./data/cleaned_vgsales.csv"):
    print("📄 Loading clean dataset...")
    df = pd.read_csv(csv_path)
    print(f"✅ Clean data loaded: {df.shape[0]} rows")

    print("🛠 Setting up Great Expectations context...")
    context = gx.get_context()

    context.add_datasource(
        name="vg_pandas_datasource",
        class_name="Datasource",
        execution_engine={"class_name": "PandasExecutionEngine"},
        data_connectors={
            "runtime_connector": {
                "class_name": "RuntimeDataConnector",
                "batch_identifiers": ["default_identifier"]
            }
        }
    )

    suite_name = "vg_sales_expectation_suite"
    context.add_or_update_expectation_suite(expectation_suite_name=suite_name)
```



```python
    batch_request = RuntimeBatchRequest(
        datasource_name="vg_pandas_datasource",
        data_connector_name="runtime_connector",
        data_asset_name="vgsales_data",
        runtime_parameters={"batch_data": df},
        batch_identifiers={"default_identifier": "default"}
    )

    validator = context.get_validator(
        batch_request=batch_request,
        expectation_suite_name=suite_name
    )

    # Define expectations
    validator.expect_table_row_count_to_be_between(min_value=1, max_value=1_000_000)
    validator.expect_column_to_exist("Name")
    validator.expect_column_to_exist("Platform")
    validator.expect_column_to_exist("Year")
    validator.expect_column_to_exist("Global_Sales")

    validator.expect_column_values_to_not_be_null("Name")
    validator.expect_column_values_to_not_be_null("Platform")
    validator.expect_column_values_to_not_be_null("Global_Sales")

    validator.expect_column_values_to_be_between("Year", min_value=1980, max_value=2025)
    validator.expect_column_values_to_be_between("Global_Sales", min_value=0)

    validator.expect_column_values_to_be_in_set("Platform", [
        "Wii", "NES", "PS4", "PS3", "X360", "GB", "DS", "PS2", "SNES",
        "GBA", "3DS", "N64", "XB", "PC", "PS", "XOne"
    ])
    validator.expect_column_values_to_match_regex("Name", r"^[A-Za-z0-9\s\:\-\&\']+$")

    validator.save_expectation_suite(discard_failed_expectations=False)
```

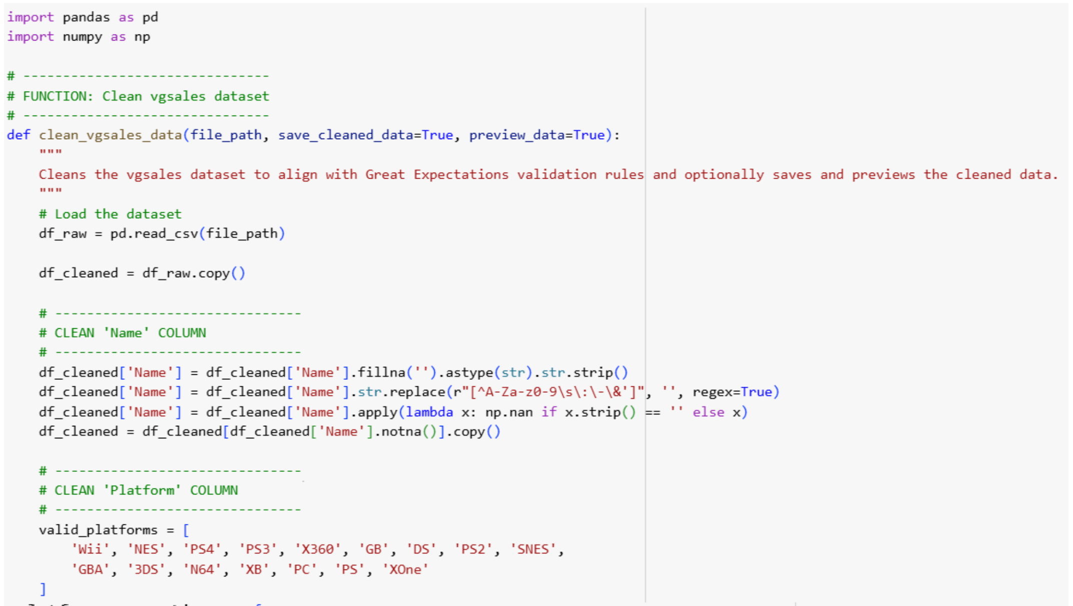


```
📄 Loading clean dataset...
✅ Clean data loaded: 3276 rows
🛠 Setting up Great Expectations context...
Calculating Metrics: 100% ██████████ 1/1 [00:00<00:00, 47.94it/s]
Calculating Metrics: 100% ██████████ 2/2 [00:00<00:00, 138.61it/s]
Calculating Metrics: 100% ██████████ 2/2 [00:00<00:00, 132.29it/s]
Calculating Metrics: 100% ██████████ 2/2 [00:00<00:00, 178.87it/s]
Calculating Metrics: 100% ██████████ 2/2 [00:00<00:00, 169.97it/s]
Calculating Metrics: 100% ██████████ 6/6 [00:00<00:00, 118.35it/s]
Calculating Metrics: 100% ██████████ 6/6 [00:00<00:00, 174.12it/s]
Calculating Metrics: 100% ██████████ 6/6 [00:00<00:00, 183.05it/s]
Calculating Metrics: 100% ██████████ 8/8 [00:00<00:00, 134.05it/s]
Calculating Metrics: 100% ██████████ 8/8 [00:00<00:00, 154.92it/s]
Calculating Metrics: 100% ██████████ 8/8 [00:00<00:00, 215.46it/s]
Calculating Metrics: 100% ██████████ 8/8 [00:00<00:00, 292.92it/s]
📊 Validating clean dataset...
Calculating Metrics: 100% ██████████ 26/26 [00:00<00:00, 87.99it/s]
```

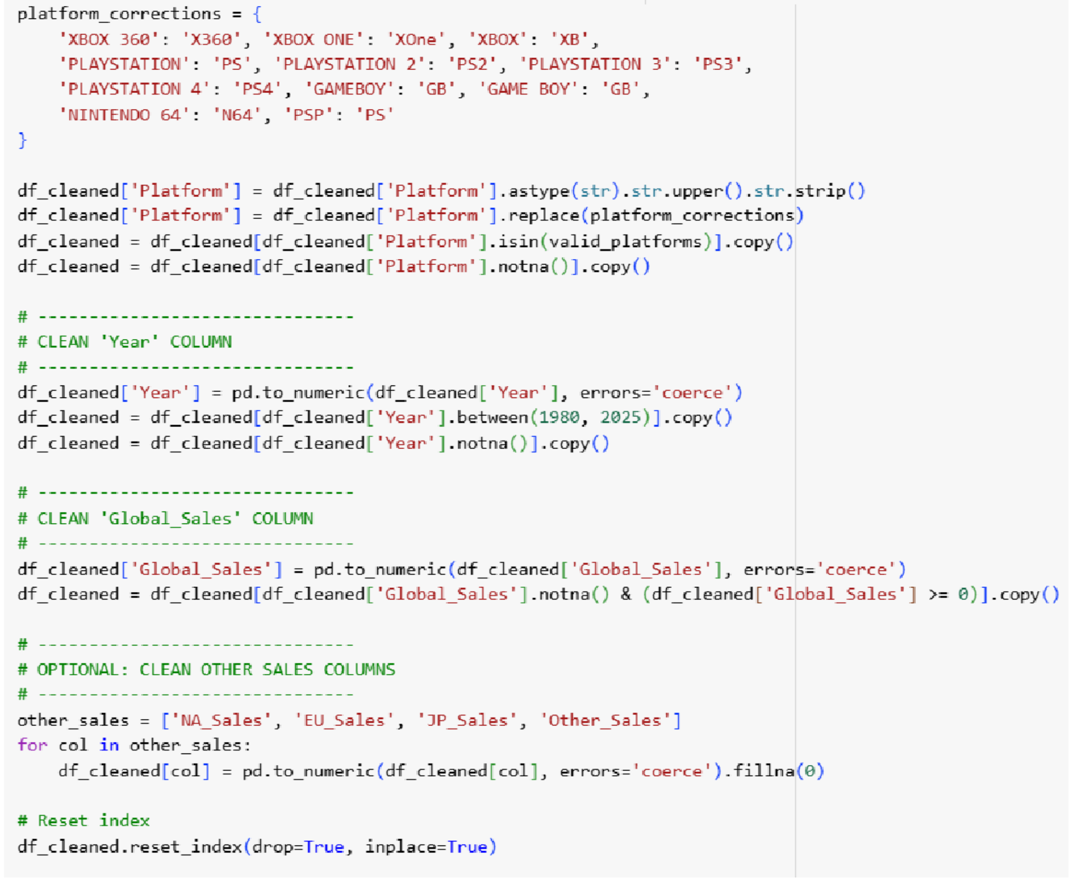


```
✅ Validation success: True
📊 Stats: {'evaluated_expectations': 12, 'successful_expectations': 12, 'unsuccessful_expectations': 0, 'success_percent': 100.0}

📋 Expectation Results:

🔹 Expectation 1: expect_table_row_count_to_be_between
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 2: expect_column_to_exist
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 3: expect_column_values_to_not_be_null
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 4: expect_column_values_to_match_regex
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 5: expect_column_to_exist
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 6: expect_column_values_to_not_be_null
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 7: expect_column_values_to_be_in_set
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 8: expect_column_to_exist
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 9: expect_column_values_to_be_between
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 10: expect_column_to_exist
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 11: expect_column_values_to_not_be_null
  ➤ Success: True
--------------------------------------------------------------------------------
🔹 Expectation 12: expect_column_values_to_be_between
  ➤ Success: True
--------------------------------------------------------------------------------
✅ Cleaned and validated data saved to ./data/validated_vgsales.csv
```

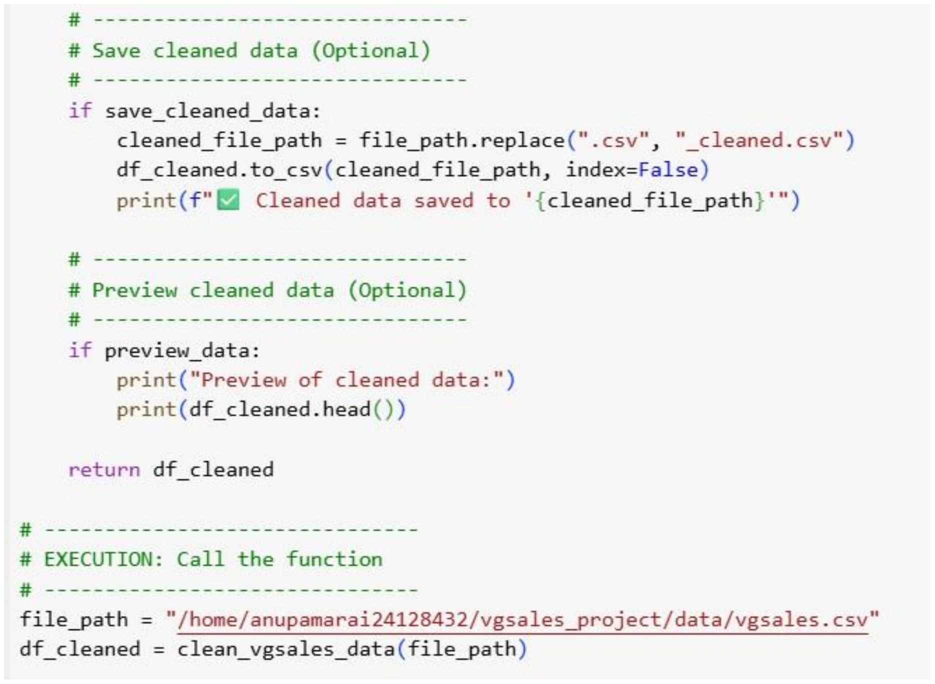
Appendix 4: (Data Preprocessing process's code)



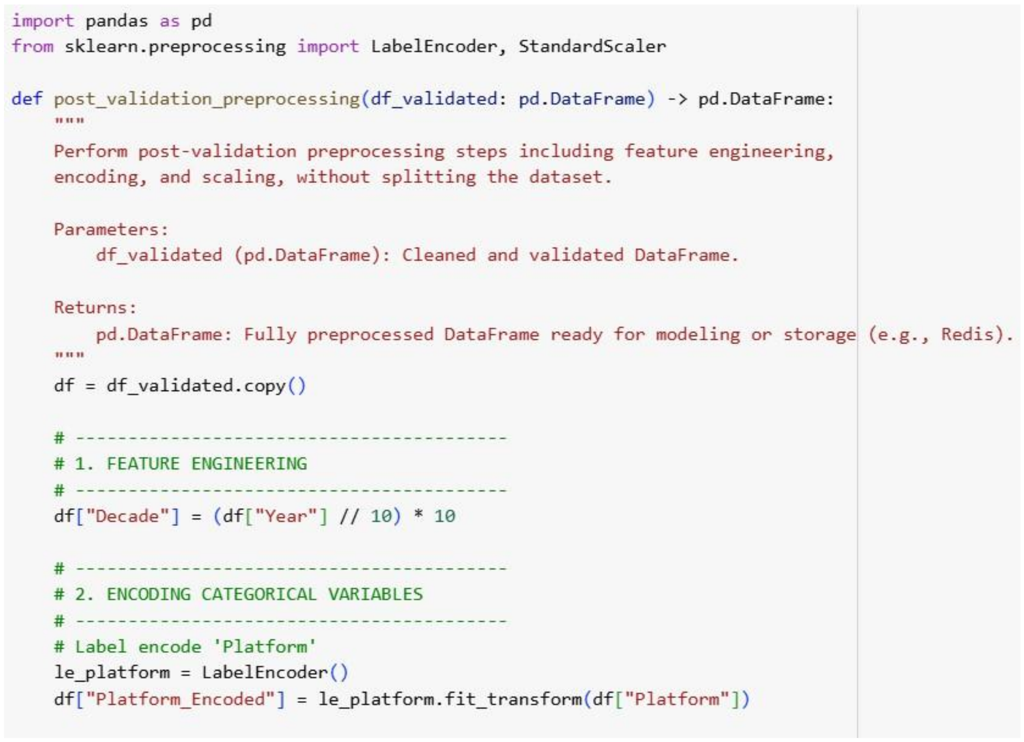
```
import pandas as pd
import numpy as np

# ------------------------------
# FUNCTION: Clean vgsales dataset
# ------------------------------
def clean_vgsales_data(file_path, save_cleaned_data=True, preview_data=True):
    """
    Cleans the vgsales dataset to align with Great Expectations validation rules and optionally saves and previews the cleaned data.
    """
    # Load the dataset
    df_raw = pd.read_csv(file_path)

    df_cleaned = df_raw.copy()

    # ------------------------------
    # CLEAN 'Name' COLUMN
    # ------------------------------
    df_cleaned['Name'] = df_cleaned['Name'].fillna('').astype(str).str.strip()
    df_cleaned['Name'] = df_cleaned['Name'].str.replace(r"[^A-Za-z0-9\s\:\-\&']", '', regex=True)
    df_cleaned['Name'] = df_cleaned['Name'].apply(lambda x: np.nan if x.strip() == '' else x)
    df_cleaned = df_cleaned[df_cleaned['Name'].notna()].copy()

    # ------------------------------
    # CLEAN 'Platform' COLUMN
    # ------------------------------
    valid_platforms = [
        'Wii', 'NES', 'PS4', 'PS3', 'X360', 'GB', 'DS', 'PS2', 'SNES',
        'GBA', '3DS', 'N64', 'XB', 'PC', 'PS', 'XOne'
    ]
```



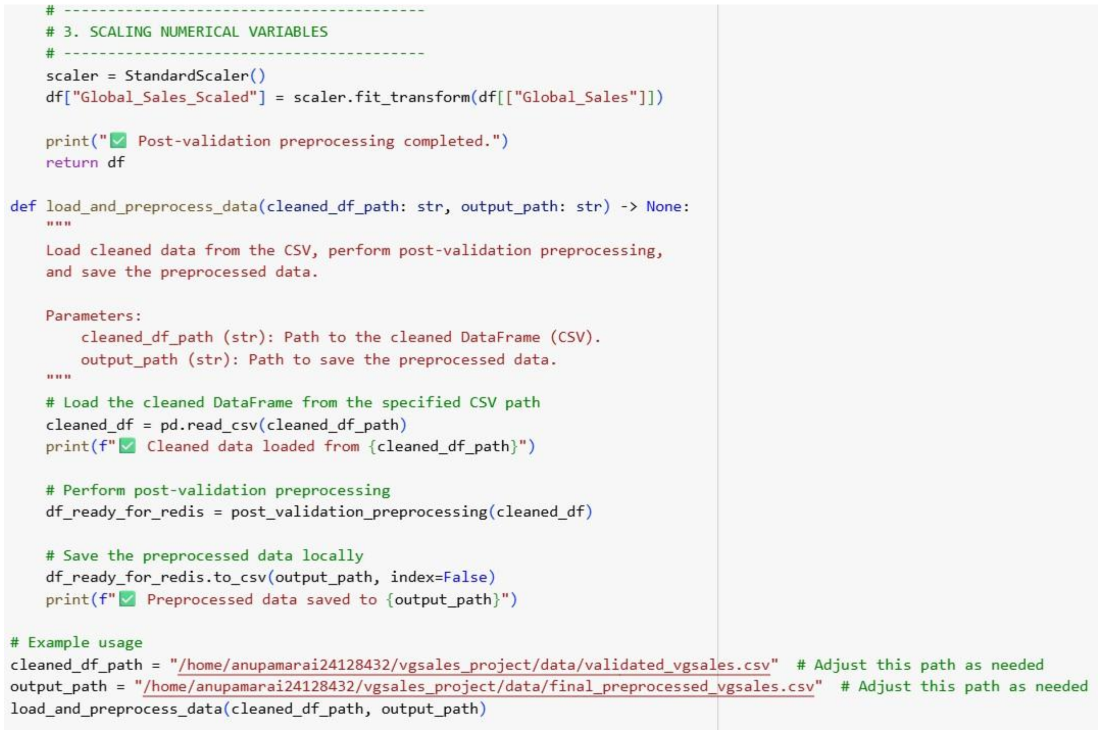
```
platform_corrections = {
     'XBOX 360': 'X360', 'XBOX ONE': 'XOne', 'XBOX': 'XB',
     'PLAYSTATION': 'PS', 'PLAYSTATION 2': 'PS2', 'PLAYSTATION 3': 'PS3',
     'PLAYSTATION 4': 'PS4', 'GAMEBOY': 'GB', 'GAME BOY': 'GB',
     'NINTENDO 64': 'N64', 'PSP': 'PS'
}

df_cleaned['Platform'] = df_cleaned['Platform'].astype(str).str.upper().str.strip()
df_cleaned['Platform'] = df_cleaned['Platform'].replace(platform_corrections)
df_cleaned = df_cleaned[df_cleaned['Platform'].isin(valid_platforms)].copy()
df_cleaned = df_cleaned[df_cleaned['Platform'].notna()].copy()

# ------------------------------
# CLEAN 'Year' COLUMN
# ------------------------------
df_cleaned['Year'] = pd.to_numeric(df_cleaned['Year'], errors='coerce')
df_cleaned = df_cleaned[df_cleaned['Year'].between(1980, 2025)].copy()
df_cleaned = df_cleaned[df_cleaned['Year'].notna()].copy()

# ------------------------------
# CLEAN 'Global_Sales' COLUMN
# ------------------------------
df_cleaned['Global_Sales'] = pd.to_numeric(df_cleaned['Global_Sales'], errors='coerce')
df_cleaned = df_cleaned[df_cleaned['Global_Sales'].notna() & (df_cleaned['Global_Sales'] >= 0)].copy()

# ------------------------------
# OPTIONAL: CLEAN OTHER SALES COLUMNS
# ------------------------------
other_sales = ['NA_Sales', 'EU_Sales', 'JP_Sales', 'Other_Sales']
for col in other_sales:
    df_cleaned[col] = pd.to_numeric(df_cleaned[col], errors='coerce').fillna(0)

# Reset index
df_cleaned.reset_index(drop=True, inplace=True)
```



```python
    # ------------------------------
    # Save cleaned data (Optional)
    # ------------------------------
    if save_cleaned_data:
        cleaned_file_path = file_path.replace(".csv", "_cleaned.csv")
        df_cleaned.to_csv(cleaned_file_path, index=False)
        print(f"✅ Cleaned data saved to '{cleaned_file_path}'")

    # ------------------------------
    # Preview cleaned data (Optional)
    # ------------------------------
    if preview_data:
        print("Preview of cleaned data:")
        print(df_cleaned.head())

    return df_cleaned

# ------------------------------
# EXECUTION: Call the function
# ------------------------------
file_path = "/home/anupamarai24128432/vgsales_project/data/vgsales.csv"
df_cleaned = clean_vgsales_data(file_path)
```

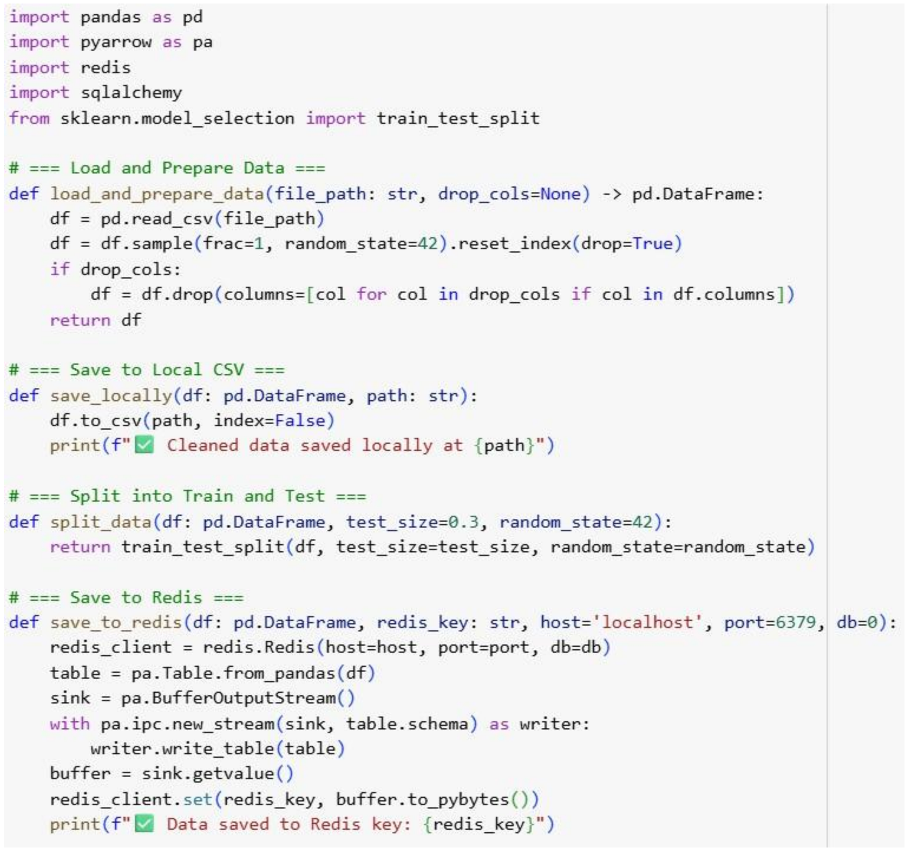


```python
import pandas as pd
from sklearn.preprocessing import LabelEncoder, StandardScaler

def post_validation_preprocessing(df_validated: pd.DataFrame) -> pd.DataFrame:
    """
    Perform post-validation preprocessing steps including feature engineering,
    encoding, and scaling, without splitting the dataset.

    Parameters:
        df_validated (pd.DataFrame): Cleaned and validated DataFrame.

    Returns:
        pd.DataFrame: Fully preprocessed DataFrame ready for modeling or storage (e.g., Redis).
    """
    df = df_validated.copy()

    # ----------------------------------------
    # 1. FEATURE ENGINEERING
    # ----------------------------------------
    df["Decade"] = (df["Year"] // 10) * 10

    # ----------------------------------------
    # 2. ENCODING CATEGORICAL VARIABLES
    # ----------------------------------------
    # Label encode 'Platform'
    le_platform = LabelEncoder()
    df["Platform_Encoded"] = le_platform.fit_transform(df["Platform"])
```

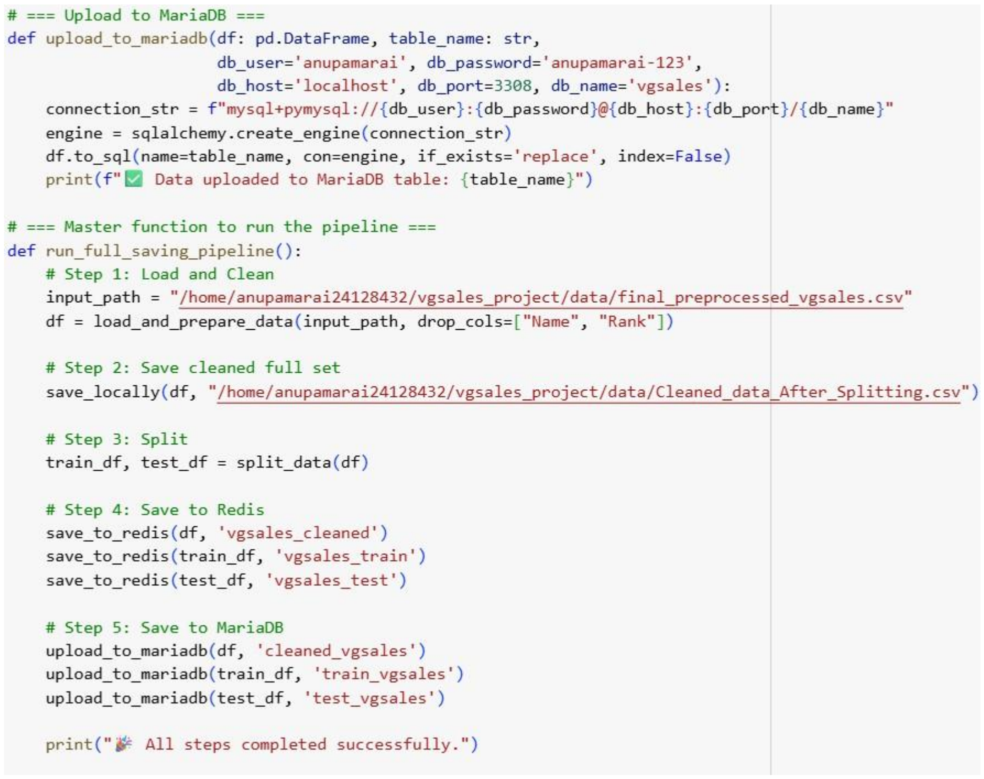


```
    # ----------------------------------------
    # 3. SCALING NUMERICAL VARIABLES
    # ----------------------------------------
    scaler = StandardScaler()
    df["Global_Sales_Scaled"] = scaler.fit_transform(df[["Global_Sales"]])

    print("✅ Post-validation preprocessing completed.")
    return df

def load_and_preprocess_data(cleaned_df_path: str, output_path: str) -> None:
    """
    Load cleaned data from the CSV, perform post-validation preprocessing,
    and save the preprocessed data.

    Parameters:
        cleaned_df_path (str): Path to the cleaned DataFrame (CSV).
        output_path (str): Path to save the preprocessed data.
    """
    # Load the cleaned DataFrame from the specified CSV path
    cleaned_df = pd.read_csv(cleaned_df_path)
    print(f"✅ Cleaned data loaded from {cleaned_df_path}")

    # Perform post-validation preprocessing
    df_ready_for_redis = post_validation_preprocessing(cleaned_df)

    # Save the preprocessed data locally
    df_ready_for_redis.to_csv(output_path, index=False)
    print(f"✅ Preprocessed data saved to {output_path}")

# Example usage
cleaned_df_path = "/home/anupamarai24128432/vgsales_project/data/validated_vgsales.csv"  # Adjust this path as needed
output_path = "/home/anupamarai24128432/vgsales_project/data/final_preprocessed_vgsales.csv"  # Adjust this path as needed
load_and_preprocess_data(cleaned_df_path, output_path)
```

Appendix 5: (Uploading in Redis code)

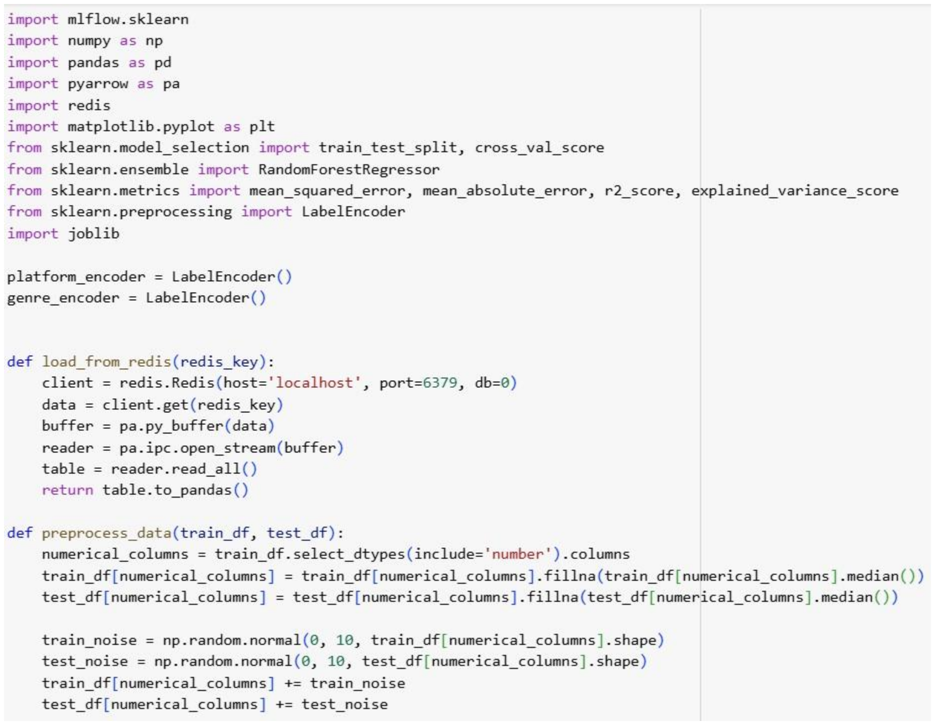


```
import pandas as pd
import pyarrow as pa
import redis
import sqlalchemy
from sklearn.model_selection import train_test_split

# === Load and Prepare Data ===
def load_and_prepare_data(file_path: str, drop_cols=None) -> pd.DataFrame:
    df = pd.read_csv(file_path)
    df = df.sample(frac=1, random_state=42).reset_index(drop=True)
    if drop_cols:
        df = df.drop(columns=[col for col in drop_cols if col in df.columns])
    return df

# === Save to Local CSV ===
def save_locally(df: pd.DataFrame, path: str):
    df.to_csv(path, index=False)
    print(f"✅ Cleaned data saved locally at {path}")

# === Split into Train and Test ===
def split_data(df: pd.DataFrame, test_size=0.3, random_state=42):
    return train_test_split(df, test_size=test_size, random_state=random_state)

# === Save to Redis ===
def save_to_redis(df: pd.DataFrame, redis_key: str, host='localhost', port=6379, db=0):
    redis_client = redis.Redis(host=host, port=port, db=db)
    table = pa.Table.from_pandas(df)
    sink = pa.BufferOutputStream()
    with pa.ipc.new_stream(sink, table.schema) as writer:
        writer.write_table(table)
    buffer = sink.getvalue()
    redis_client.set(redis_key, buffer.to_pybytes())
    print(f"✅ Data saved to Redis key: {redis_key}")
```

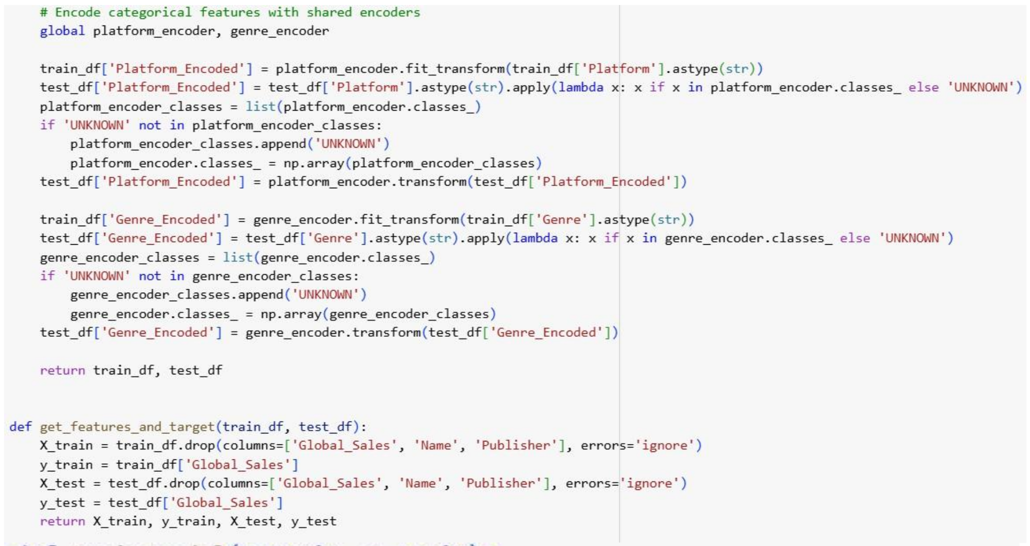


```python
# === Upload to MariaDB ===
def upload_to_mariadb(df: pd.DataFrame, table_name: str,
                      db_user='anupamarai', db_password='anupamarai-123',
                      db_host='localhost', db_port=3308, db_name='vgsales'):
    connection_str = f"mysql+pymysql://{db_user}:{db_password}@{db_host}:{db_port}/{db_name}"
    engine = sqlalchemy.create_engine(connection_str)
    df.to_sql(name=table_name, con=engine, if_exists='replace', index=False)
    print(f"✅ Data uploaded to MariaDB table: {table_name}")

# === Master function to run the pipeline ===
def run_full_saving_pipeline():
    # Step 1: Load and Clean
    input_path = "/home/anupamarai24128432/vgsales_project/data/final_preprocessed_vgsales.csv"
    df = load_and_prepare_data(input_path, drop_cols=["Name", "Rank"])

    # Step 2: Save cleaned full set
    save_locally(df, "/home/anupamarai24128432/vgsales_project/data/Cleaned_data_After_Splitting.csv")

    # Step 3: Split
    train_df, test_df = split_data(df)

    # Step 4: Save to Redis
    save_to_redis(df, 'vgsales_cleaned')
    save_to_redis(train_df, 'vgsales_train')
    save_to_redis(test_df, 'vgsales_test')

    # Step 5: Save to MariaDB
    upload_to_mariadb(df, 'cleaned_vgsales')
    upload_to_mariadb(train_df, 'train_vgsales')
    upload_to_mariadb(test_df, 'test_vgsales')

    print("🎉 All steps completed successfully.")
```

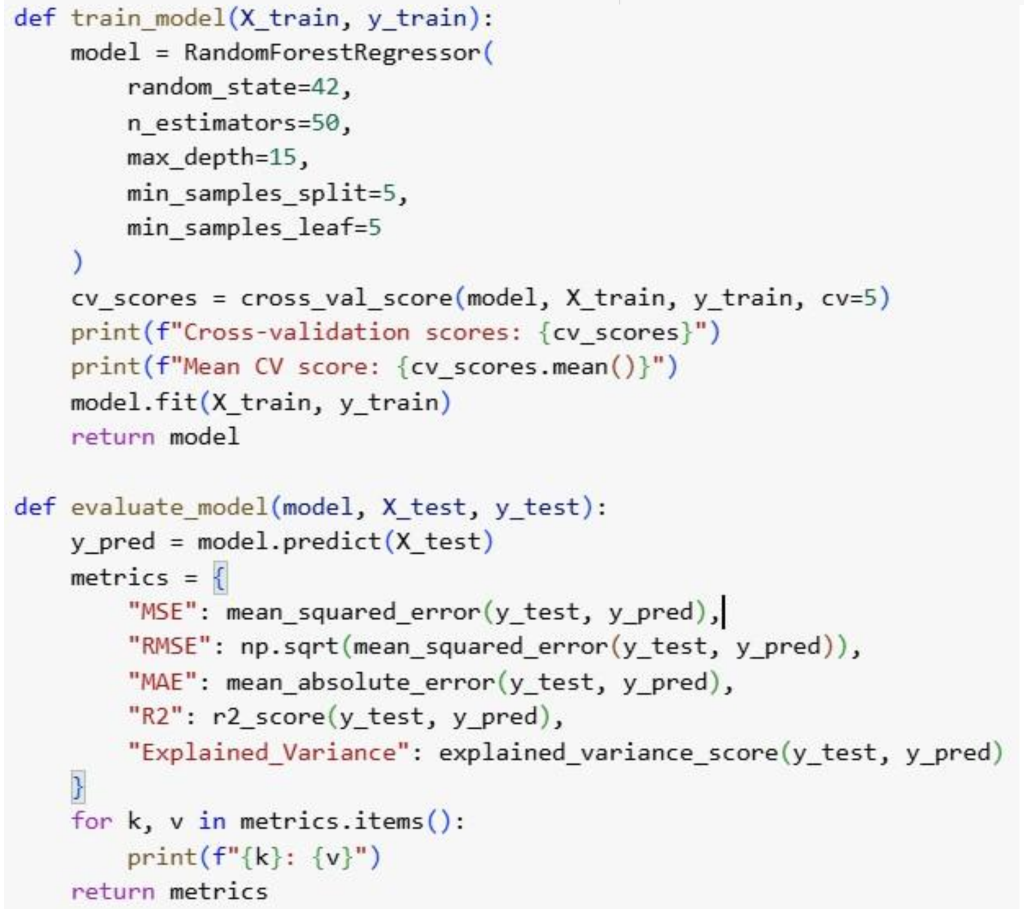
Appendix 6: (Model Development code and Evaluation Metrics)



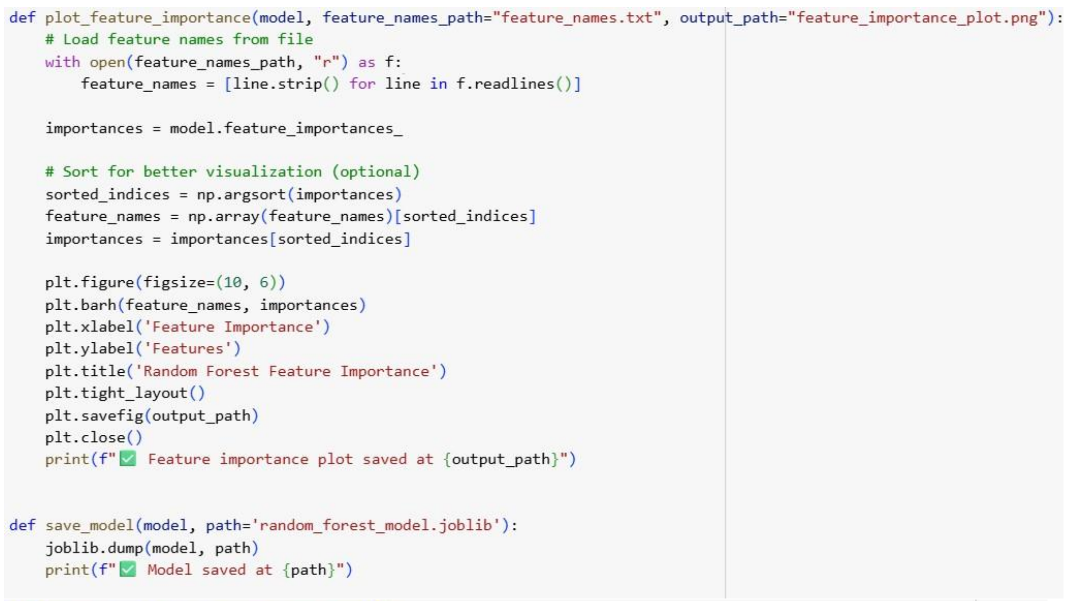
```python
import mlflow.sklearn
import numpy as np
import pandas as pd
import pyarrow as pa
import redis
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score, explained_variance_score
from sklearn.preprocessing import LabelEncoder
import joblib

platform_encoder = LabelEncoder()
genre_encoder = LabelEncoder()


def load_from_redis(redis_key):
    client = redis.Redis(host='localhost', port=6379, db=0)
    data = client.get(redis_key)
    buffer = pa.py_buffer(data)
    reader = pa.ipc.open_stream(buffer)
    table = reader.read_all()
    return table.to_pandas()

def preprocess_data(train_df, test_df):
    numerical_columns = train_df.select_dtypes(include='number').columns
    train_df[numerical_columns] = train_df[numerical_columns].fillna(train_df[numerical_columns].median())
    test_df[numerical_columns] = test_df[numerical_columns].fillna(test_df[numerical_columns].median())

    train_noise = np.random.normal(0, 10, train_df[numerical_columns].shape)
    test_noise = np.random.normal(0, 10, test_df[numerical_columns].shape)
    train_df[numerical_columns] += train_noise
    test_df[numerical_columns] += test_noise
```



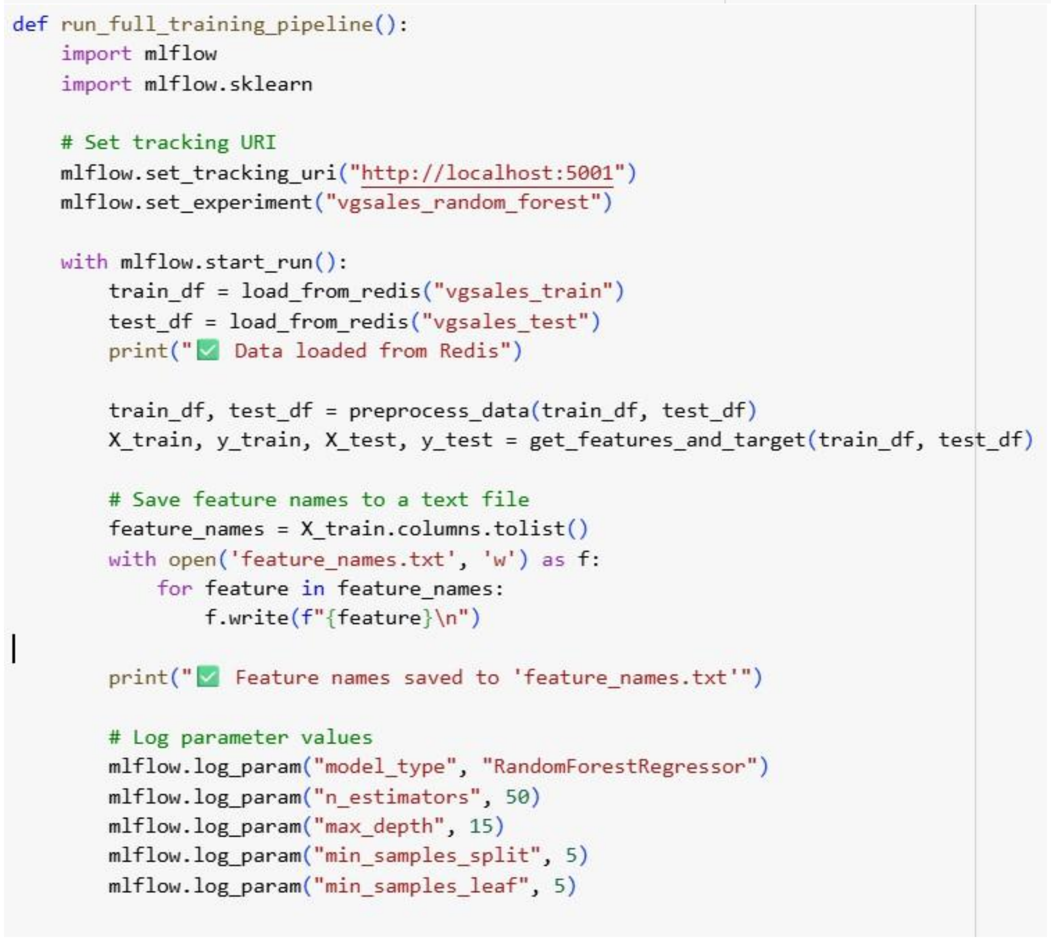
```
    # Encode categorical features with shared encoders
    global platform_encoder, genre_encoder

    train_df['Platform_Encoded'] = platform_encoder.fit_transform(train_df['Platform'].astype(str))
    test_df['Platform_Encoded'] = test_df['Platform'].astype(str).apply(lambda x: x if x in platform_encoder.classes_ else 'UNKNOWN')
    platform_encoder_classes = list(platform_encoder.classes_)
    if 'UNKNOWN' not in platform_encoder_classes:
        platform_encoder_classes.append('UNKNOWN')
        platform_encoder.classes_ = np.array(platform_encoder_classes)
    test_df['Platform_Encoded'] = platform_encoder.transform(test_df['Platform_Encoded'])

    train_df['Genre_Encoded'] = genre_encoder.fit_transform(train_df['Genre'].astype(str))
    test_df['Genre_Encoded'] = test_df['Genre'].astype(str).apply(lambda x: x if x in genre_encoder.classes_ else 'UNKNOWN')
    genre_encoder_classes = list(genre_encoder.classes_)
    if 'UNKNOWN' not in genre_encoder_classes:
        genre_encoder_classes.append('UNKNOWN')
        genre_encoder.classes_ = np.array(genre_encoder_classes)
    test_df['Genre_Encoded'] = genre_encoder.transform(test_df['Genre_Encoded'])

    return train_df, test_df


def get_features_and_target(train_df, test_df):
    X_train = train_df.drop(columns=['Global_Sales', 'Name', 'Publisher'], errors='ignore')
    y_train = train_df['Global_Sales']
    X_test = test_df.drop(columns=['Global_Sales', 'Name', 'Publisher'], errors='ignore')
    y_test = test_df['Global_Sales']
    return X_train, y_train, X_test, y_test
```



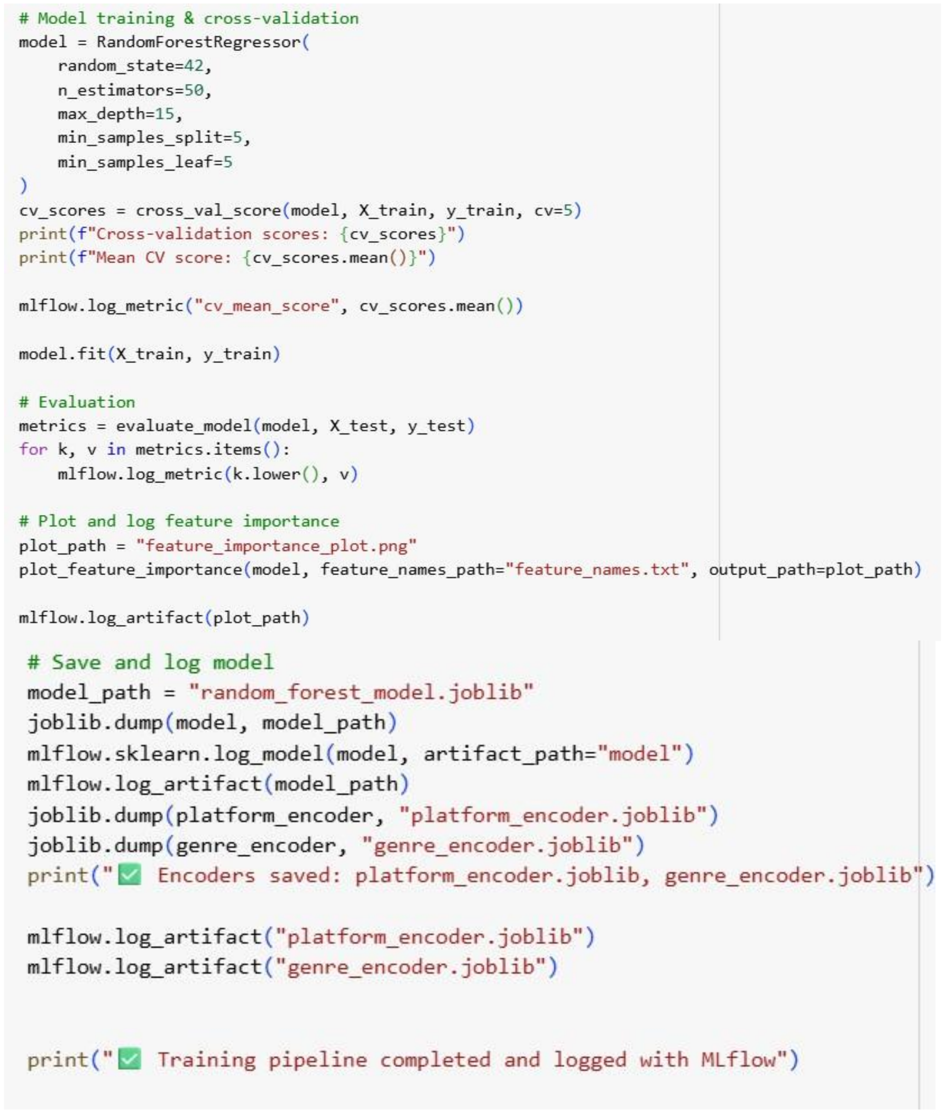
```
def train_model(X_train, y_train):
    model = RandomForestRegressor(
        random_state=42,
        n_estimators=50,
        max_depth=15,
        min_samples_split=5,
        min_samples_leaf=5
    )
    cv_scores = cross_val_score(model, X_train, y_train, cv=5)
    print(f"Cross-validation scores: {cv_scores}")
    print(f"Mean CV score: {cv_scores.mean()}")
    model.fit(X_train, y_train)
    return model

def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    metrics = {
        "MSE": mean_squared_error(y_test, y_pred),
        "RMSE": np.sqrt(mean_squared_error(y_test, y_pred)),
        "MAE": mean_absolute_error(y_test, y_pred),
        "R2": r2_score(y_test, y_pred),
        "Explained_Variance": explained_variance_score(y_test, y_pred)
    }
    for k, v in metrics.items():
        print(f"{k}: {v}")
    return metrics
```



```python
def plot_feature_importance(model, feature_names_path="feature_names.txt", output_path="feature_importance_plot.png"):
    # Load feature names from file
    with open(feature_names_path, "r") as f:
        feature_names = [line.strip() for line in f.readlines()]

    importances = model.feature_importances_

    # Sort for better visualization (optional)
    sorted_indices = np.argsort(importances)
    feature_names = np.array(feature_names)[sorted_indices]
    importances = importances[sorted_indices]

    plt.figure(figsize=(10, 6))
    plt.barh(feature_names, importances)
    plt.xlabel('Feature Importance')
    plt.ylabel('Features')
    plt.title('Random Forest Feature Importance')
    plt.tight_layout()
    plt.savefig(output_path)
    plt.close()
    print(f"✅ Feature importance plot saved at {output_path}")


def save_model(model, path='random_forest_model.joblib'):
    joblib.dump(model, path)
    print(f"✅ Model saved at {path}")
```



```python
def run_full_training_pipeline():
    import mlflow
    import mlflow.sklearn

    # Set tracking URI
    mlflow.set_tracking_uri("http://localhost:5001")
    mlflow.set_experiment("vgsales_random_forest")

    with mlflow.start_run():
        train_df = load_from_redis("vgsales_train")
        test_df = load_from_redis("vgsales_test")
        print("✅ Data loaded from Redis")

        train_df, test_df = preprocess_data(train_df, test_df)
        X_train, y_train, X_test, y_test = get_features_and_target(train_df, test_df)

        # Save feature names to a text file
        feature_names = X_train.columns.tolist()
        with open('feature_names.txt', 'w') as f:
            for feature in feature_names:
                f.write(f"{feature}\n")

        print("✅ Feature names saved to 'feature_names.txt'")

        # Log parameter values
        mlflow.log_param("model_type", "RandomForestRegressor")
        mlflow.log_param("n_estimators", 50)
        mlflow.log_param("max_depth", 15)
        mlflow.log_param("min_samples_split", 5)
        mlflow.log_param("min_samples_leaf", 5)
```



```
# Model training & cross-validation
model = RandomForestRegressor(
    random_state=42,
    n_estimators=50,
    max_depth=15,
    min_samples_split=5,
    min_samples_leaf=5
)
cv_scores = cross_val_score(model, X_train, y_train, cv=5)
print(f"Cross-validation scores: {cv_scores}")
print(f"Mean CV score: {cv_scores.mean()}")

mlflow.log_metric("cv_mean_score", cv_scores.mean())

model.fit(X_train, y_train)

# Evaluation
metrics = evaluate_model(model, X_test, y_test)
for k, v in metrics.items():
    mlflow.log_metric(k.lower(), v)

# Plot and log feature importance
plot_path = "feature_importance_plot.png"
plot_feature_importance(model, feature_names_path="feature_names.txt", output_path=plot_path)

mlflow.log_artifact(plot_path)
# Save and log model
model_path = "random_forest_model.joblib"
joblib.dump(model, model_path)
mlflow.sklearn.log_model(model, artifact_path="model")
mlflow.log_artifact(model_path)
joblib.dump(platform_encoder, "platform_encoder.joblib")
joblib.dump(genre_encoder, "genre_encoder.joblib")
print("✅ Encoders saved: platform_encoder.joblib, genre_encoder.joblib")

mlflow.log_artifact("platform_encoder.joblib")
mlflow.log_artifact("genre_encoder.joblib")


print("✅ Training pipeline completed and logged with MLflow")
```

Mean Squared Error (MSE): MSE is the average of squared differences between predicted and actual values which is used to penalize larger errors more than smaller ones, so it is suitable when large erros are particularly undesirable.

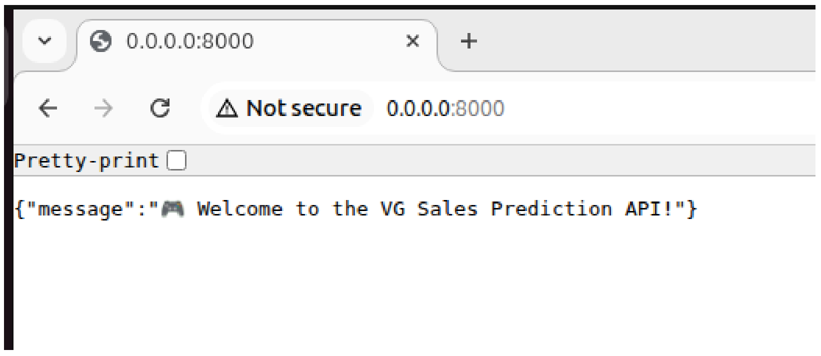
Root Mean Squared Error (RMSE): It is the squared root of MSE, which is use to bring the error back to the same scale as the target variable (Global_Sales), making it more interpretable.

Mean Absolute Error (MAE): MAE is the average of the absolute differences between predicted and actual values, and it is easy to use for interpretation. It is also less sensitive to outliers than MSE.
R-squared (R^2 Score): It is the proportion of variance in the dependent variable that is predictable from the independent variable, which is used to give an overall sense of model accuracy. It ranges from 0 to 1 (closer to 1 is a better outcome).

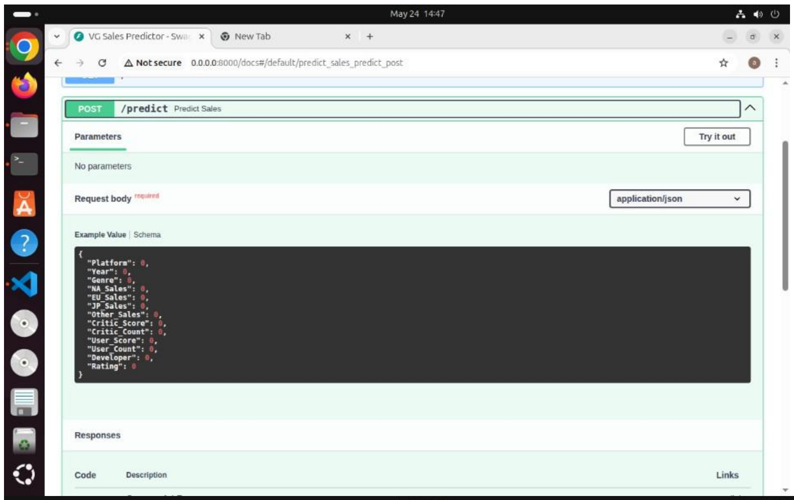
Explained Variance: It measures the proportion of the target variance explained by the model, which is similar to R^2. It is used to confirm how well the model is capturing variability in the data.

Appendix 7: (FastAPI code and output)

```python
import joblib
import pandas as pd
from fastapi import FastAPI
from pydantic import BaseModel
from sklearn.ensemble import RandomForestRegressor

# Load the trained model and feature names
model = joblib.load("random_forest_model.joblib")
with open("feature_names.txt", "r") as f:
    feature_names = [line.strip() for line in f.readlines()]


app = FastAPI()

# Define Pydantic model for request data validation
class SalesPredictionRequest(BaseModel):
    Platform: str
    Year: int
    Genre: str
    NA_Sales: float
    EU_Sales: float
    JP_Sales: float
    Other_Sales: float
@app.post("/predict")
def predict_sales(request: SalesPredictionRequest):
    # Convert request data into a DataFrame with the correct columns
    input_data = pd.DataFrame([request.dict()])

    # Reorder columns to match the trained model
    input_data = input_data[feature_names]

    # Predict with the trained model
    prediction = model.predict(input_data)[0]

    # Return the prediction result
    return {"predicted_sales": prediction}
```



```
{"message":"🎮 Welcome to the VG Sales Prediction API!"}
```


May 24 14:47

VG Sales Predictor - Swa | New Tab

Not secure 0.0.0.0:8000/docs#/default/predict_sales_predict_post

POST /predict Predict Sales

Parameters — Try it out

No parameters

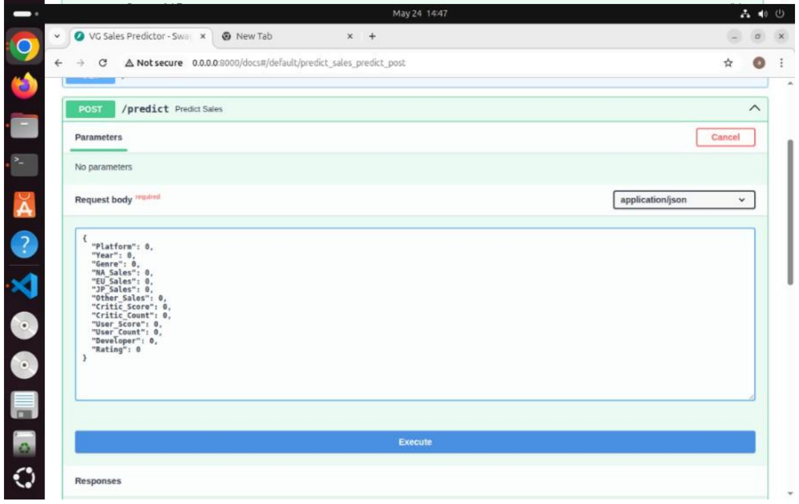
Request body required — application/json

Example Value | Schema

```
{
  "Platform": 0,
  "Year": 0,
  "Genre": 0,
  "NA_Sales": 0,
  "EU_Sales": 0,
  "JP_Sales": 0,
  "Other_Sales": 0,
  "Critic_Score": 0,
  "Critic_Count": 0,
  "User_Score": 0,
  "User_Count": 0,
  "Developer": 0,
  "Rating": 0
}
```

Responses

Code | Description | Links


May 24 14:47

VG Sales Predictor - Swa | New Tab

Not secure 0.0.0.0:8000/docs#/default/predict_sales_predict_post

POST /predict Predict Sales

Parameters — Cancel

No parameters

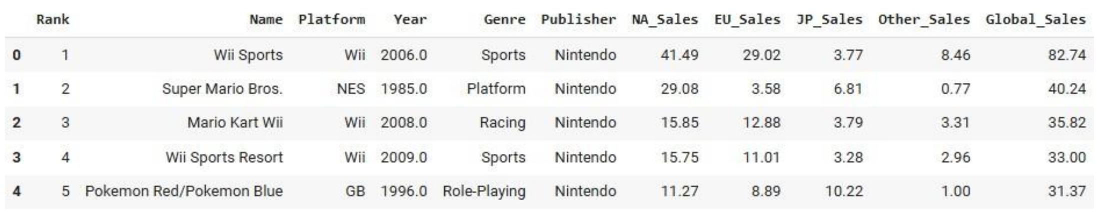
Request body required — application/json

```
{
  "Platform": 0,
  "Year": 0,
  "Genre": 0,
  "NA_Sales": 0,
  "EU_Sales": 0,
  "JP_Sales": 0,
  "Other_Sales": 0,
  "Critic_Score": 0,
  "Critic_Count": 0,
  "User_Score": 0,
  "User_Count": 0,
  "Developer": 0,
  "Rating": 0
}
```

Execute

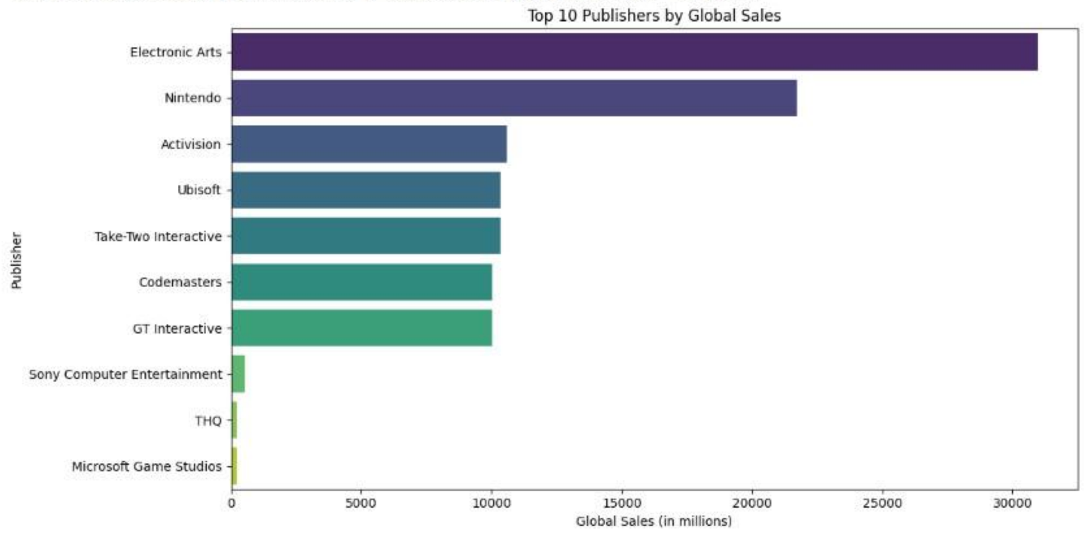
Responses

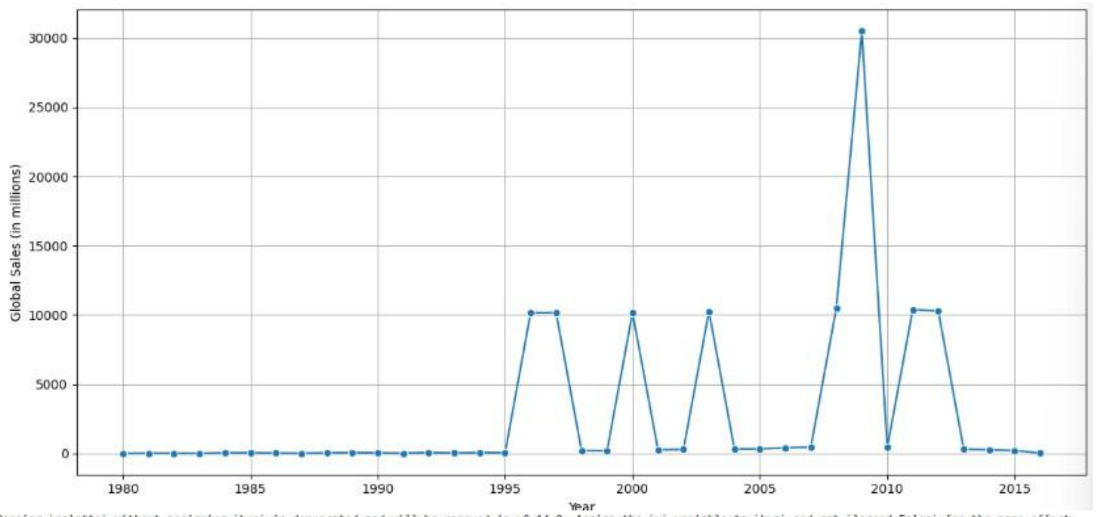
Appendix 8: (Data Visualization)



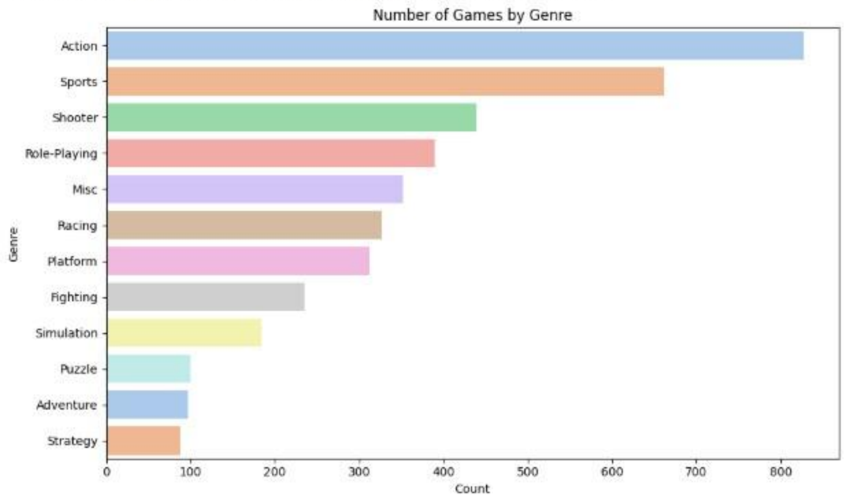
	Rank	Name	Platform	Year	Genre	Publisher	NA_Sales	EU_Sales	JP_Sales	Other_Sales	Global_Sales
0	1	Wii Sports	Wii	2006.0	Sports	Nintendo	41.49	29.02	3.77	8.46	82.74
1	2	Super Mario Bros.	NES	1985.0	Platform	Nintendo	29.08	3.58	6.81	0.77	40.24
2	3	Mario Kart Wii	Wii	2008.0	Racing	Nintendo	15.85	12.88	3.79	3.31	35.82
3	4	Wii Sports Resort	Wii	2009.0	Sports	Nintendo	15.75	11.01	3.28	2.96	33.00
4	5	Pokemon Red/Pokemon Blue	GB	1996.0	Role-Playing	Nintendo	11.27	8.89	10.22	1.00	31.37

```
Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `

  sns.barplot(x=top_publishers.values, y=top_publishers.index, palette='viridis')
```





```
Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same effect.

  sns.countplot(y='Genre', data=df, order=df['Genre'].value_counts().index, palette='pastel')
```



Appendix 9: (Justification for ELT Approach)

The ELT (Extract, Load, Transform) methodology is more appropriate for this project than traditional ETL for several reasons:

Flexibility in experimentation: By loading raw data into a centralized storage layer (MariaDB), data scientists and engineers can iteratively apply various transformation strategies. This method prevents premature commitment to a single transformation logic during extraction, thereby facilitating flexible exploratory workflows and accelerating prototyping.

Storage-first architecture: ELT separates transformation from extraction, taking advantage of the high performance offered by relational databases like MariaDB. These databases are adept at managing and indexing substantial volumes of raw input, which enables more scalable querying and batch processing compared to in-memory transformation pipelines.

MLOps compatibility: ELT is well-aligned with MLOps principles, where data transformations frequently adapt to model requirements. For example, feature engineering may vary based on the algorithm employed. ELT guarantees that raw data remains unaltered in storage, thereby supporting traceability and reproducibility of experiments across various model iterations.

Modularity and maintainability: The transformation logic—such as data cleaning, encoding, or normalization—is organized into separate pipeline components and scripts. This structure allows for the independent updating, testing, and reuse of individual parts of the data pipeline without modifying the raw data or upstream extraction logic. It fosters code reusability and simplifies debugging.

Version control and auditability: By loading raw datasets into databases, teams can monitor data versions, apply timestamps, and utilize audit logs for changes in transformation logic—ensuring compliance and historical traceability.

References

- Abowd, J.M. (2018) ‘The U.S. Census Bureau Adopts Differential Privacy’, *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* [Preprint]. Available at: https://doi.org/10.1145/3219819.3226070.

- Apple (2020) *Overview*, *Apple Machine Learning Research*. Available at: https://machinelearning.apple.com/.

- Bansal, S. (2019). *Netflix Movies and TV Shows*. Kaggle.com.

  https://www.kaggle.com/datasets/shivamb/netflix-shows/versions/3

- Dwork, C. *et al.* (2006) ‘Calibrating Noise to Sensitivity in Private Data Analysis’, *Theory of Cryptography*, pp. 265–284. Available at: https://doi.org/10.1007/11681878_14.

- Erlingsson, Ú., Pihur, V. and Korolova, A. (no date) ‘RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response’. Available at: https://doi.org/10.1145/2660267.2660348.

- GeeksforGeeks (2023) *What is Feature Engineering?*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/what-is-feature-engineering/.

- GeeksforGeeks (2024) *What is Data Ingestion?*, *GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/what-is-data-ingestion/.

- Google Cloud (no date) *Introduction to Vertex AI Pipelines*, *Google Cloud*. Available at: https://cloud.google.com/vertex-ai/docs/pipelines/introduction.

- IBM (2021) *What Is Machine learning?*, *Ibm.com*. IBM. Available at: https://www.ibm.com/think/topics/machine-learning.

- Jain, D. (2019) *Data Preprocessing in Data Mining*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/data-preprocessing-in-data-mining/.

- PromptCloud. (2017). *IMDB Horror Movie Dataset [2012 Onwards]*. Kaggle.com. https://www.kaggle.com/datasets/PromptCloudHQ/imdb-horror-movie-dataset
- Shubham Sayon (2020) *Supervised And Unsupervised Machine Learning – Be on the Right Side of Change*, *Finxter.com*. Available at: https://blog.finxter.com/supervised-and-unsupervised-machine-learning/ (Accessed: 19 May 2025).
- Smith, G. (no date) *Video Game Sales*, *www.kaggle.com*. Available at: https://www.kaggle.com/datasets/gregorut/videogamesales/data.
- Talk Cloud (2024) *Data Validation in Data Engineering: Ensuring Data Quality and Integrity*, *Medium*. Available at: https://medium.com/@talk-cloud/data-validation-in-data-engineering-ensuring-data-quality-and-integrity-178858cac564 (Accessed: 19 May 2025).
- Wang, J. (2024) *ML model monitoring and re-training: When and How? - Julian Wang - Medium*, *Medium*. Available at: https://medium.com/@j.wang.mlds/ml-model-monitoring-and-re-training-when-and-how-7fbfe3b64433 (Accessed: 19 May 2025).