KingMakers

Game Recommender System
Take-Home Assessment Documentation

Compiled by Keelan Govender

29/03/2023

Table of Contents

Introduction

In today's fast-paced gaming industry, personalized game recommendations play a crucial role in enhancing player engagement and retention. Content-based recommender systems have emerged as a powerful tool to suggest games to players based on their individual preferences and playing history. By leveraging the attributes and characteristics of games, these systems can provide tailored recommendations that align with players' interests, leading to increased player satisfaction and loyalty.

The objective of this assessment is to develop a content-based recommender system that recommends games to players based on the games that they have previously played. By analyzing the similarity of game attributes, the system aims to identify and suggest games that closely match the player's preferences, thereby improving the overall gaming experience.

The development of this recommender system involves several key steps. Firstly, a synthetic dataset for a game attribute table is created, containing 1000 unique game entries with various attributes such as game ID, name, genre, features, and more. This dataset serves as the foundation for understanding the characteristics of each game.
Secondly, a synthetic player history table is generated, representing the interaction history of players with different games. Each row in this table includes the player ID and the game ID, indicating which games each player has played. This information is vital for understanding player preferences and building personalized recommendations.
The recommender system utilizes the game attribute table and player history table to find similar games based on the attributes of the games a player has already played. By leveraging the power of machine learning algorithms and similarity metrics, the system can identify games that closely match a player's preferences, providing them with highly relevant and personalized recommendations.

Throughout this assessment, we will delve into the process of data generation, preprocessing, model development, evaluation, and deployment. The chosen technology stack, including PySpark, Databricks, and popular Python libraries like scikit-learn and pandas, will be utilized to build a scalable and efficient recommender system. Additionally, we will explore the application of MLOps practices to ensure the system's reliability, maintainability, and performance in a cloud environment, with a focus on Azure's cloud infrastructure.

By developing a content-based recommender system for game recommendations, we aim to showcase the potential of machine learning in enhancing player engagement and providing personalized gaming experiences. The insights gained from this assessment can be valuable for game developers, publishers, and platforms seeking to optimize their recommendation strategies and drive player satisfaction.

Synthetic Data Generation

(Please refer to Appendix A - Synthetic Data Generation Code)

Game Attribute Table

Columns Provided:

Feature Category: Features which incorporate a game's attributes.
Special Features: Information regarding the population of data for the Feature Category.
Expected Metadata Values: Datatypes and Expected values.

Comments/Assumptions: My thoughts on how each feature is interpreted and assumptions made.
These would normally be taken to business users, data scientists and others involved in the process of generating this model or with domain knowledge to verify assumptions or rework logic.

Feature Category	Special Features	Expected Metadata Values	Comments/ Assumptions
ID		Type: int	ID Value.
Name	Slot Game title “Queen of the Slots, Luck 'o de Oirish”, etc.	Type: string	Name of game.
Wild Symbols	Regular, Expanding, Stick	Type: Regular, Expanding, Sticky	Categorical feature. Assumption that there is no ordinality between features.
Scatter Symbols	Regular Scatter	Presence: Yes/No	Boolean value of Yes/No if scatter is present.
Free Spins	Free Spin Rounds	Count: Numerical value (e.g., 10, 20, 50)	Numerical value representing the number of free spins.
Bonus Rounds	Various Mini-Games	Presence: Yes/No	Boolean value of Yes/No if a bonus round is present.
Multipliers	Win Multipliers	Multiplier Value: Numerical (e.g., 2x, 3x)	Numerical value representing number of multipliers.

Jackpots	Progressive	Type: Fixed, Progressive	Categorical feature. Assumption that there is no ordinality between features. Progressive is specified in the "Special Features" column, so assumption is made that all values in this feature are "Progressive" but could include "Fixed" in the future.
Paylines	Multiple Paylines	Count: Numerical value (e.g., 10, 25, 50)	Numerical Value representing the paylines.
Reel Mechanisms	Cascading Reels, Megaways	Type: Cascading, Megaways	Categorical feature. Non-ordinal representation of reel mechanisms of type Cascading or Megaways. Possible inclusion of others in future?
Gamble Feature	Card Color/Suit Guessing, etc.	Presence: Yes/No	Boolean value of Yes/No indicates the presence of the gamble feature.
RTP	Return to Player Rate	Percentage: Numerical value (e.g., 92%, 96%)	Numerical value representing the return of the player rate.
Volatility	High, Low	Level: High, Medium, Low	Categorical feature with ordinality representing Volatility at Low, Medium, or High. Based on Special Features Column, assumption is that the games at present can be between Low or High with possibility of Medium in the future.

Themes	Movie, TV Show, Characters, etc.	Type: Specific theme (e.g., Adventure, Sci-Fi)	Categorical feature representing themes.
Mystery Symbols	Transforming Symbols	Presence: Yes/No	Boolean value of Yes/No indicating whether a transforming symbol is present.
Random Triggers	Random Events	Presence: Yes/No	Boolean value of Yes/No indicating whether random events are relevant.

Synthetic Dataset Generation

To generate the synthetic data, a set of Python scripts were created. These scripts leverage object-oriented programming principles to ensure code modularity, reusability, and maintainability. The scripts are organized into classes and methods/functions to encapsulate the logic for generating each table.
The population of these tables is based on generating realistic synthetic data using random values, predefined metadata, and web scraping techniques. The generated data is stored in Pandas DataFrames for further analysis and usage in the recommendation system.

The following is a summary of how each table is populated:

1. Game Attributes Table

The game attributes table is populated with synthetic data representing the attributes and metadata of the games. The logic for each column is as follows:

Column	Logic
game_id	Unique identifier for each game, generated as sequential integers.
game_name	Game names are retrieved using web scraping techniques from online slots websites.
wild_symbols	Randomly selected from a list of possible values ("regular", "expanding", "sticky").
scatter_symbols	Randomly selected from a list of possible values ("yes", "no").
free_spins	Randomly generated integer value representing the number of free spins.
bonus_rounds	Randomly selected from a list of possible values ("yes", "no").

multipliers	Randomly generated integer value representing the multiplier value.
jackpots	Randomly selected from a list of possible values ("fixed", "progressive").
paylines	Randomly generated integer value representing the number of paylines.
reel_mechanisms	Randomly selected from a list of possible values ("cascading", "megaways").
gamble_feature	Randomly selected from a list of possible values ("yes", "no").
RTP	Randomly generated float value representing the return to player percentage.
volatility	Randomly selected from a list of possible values ("high", "medium", "low").
themes	Randomly selected from a list of possible game themes.
mystery_symbols	Randomly selected from a list of possible values ("yes", "no").
random_triggers	Randomly selected from a list of possible values ("yes", "no").

The generated game attributes table is stored in a Pandas DataFrame and then outputted to a CSV file for further usage. Pandas is chosen for its efficient data manipulation and easy integration with other Python libraries.

2. *Player Attribute Table*

The player attribute table is populated with synthetic data representing the attributes of the players. The logic for each column is as follows:

Column	Logic
player_id	Unique identifier for each player, generated as sequential integers.
first_name	Randomly generated the first name for each player.
last_name	Randomly generated last name for each player.

The generated player attribute table is stored in a Pandas DataFrame and then outputted to a CSV file for further usage.

3. Player History Table

- The player history table is populated with synthetic data representing the players' game history.
- For each player, a random number of games played is generated, up to a specified maximum.
- Game IDs are randomly selected from the game attribute table for each player's game history.
- The probability of a player using the same machine for the next game is considered when selecting game IDs.

The generated player history data, including player IDs and corresponding game IDs, is stored in a Pandas DataFrame and then outputted to a CSV file for further usage.

Developing the Model

Approach

The approach taken for developing the content-based game recommender system focuses on creating a baseline model that can quickly generate value while providing a solid foundation for future iterations and improvements. The goal is to create a Minimum Viable Product (MVP) that can be easily explained to stakeholders and gain their buy-in before diving into more complex model development.
The recommender system leverages the game attributes and player history data to find similar games based on the attributes of the games a player has already played. By utilizing machine learning algorithms and similarity metrics, the system can identify games that closely match a player's preferences, delivering personalized recommendations.

Steps

(Notebook will be provided alongside documentation)

1. Data Preprocessing:

- The game attributes dataset undergoes text vectorization on the 'game_name' and 'themes' columns using techniques like tokenization, hashing TF, and IDF. These techniques convert text data into numerical representations suitable for machine learning algorithms.
- Numerical columns such as 'free_spins', 'paylines', and 'RTP' are normalized using MinMaxScaler to ensure all features are on the same scale and prevent any feature from dominating the similarity calculations.
- Categorical columns are one-hot encoded to convert them into binary vectors, allowing the model to handle categorical data effectively. Columns with only one unique value are dropped as they do not provide any discriminatory information.
- The preprocessed game attributes are combined into a single feature vector using VectorAssembler, which concatenates all the feature columns into a single vector column.

2. Similarity Calculation:

- Cosine similarity is used to calculate the similarity between game feature vectors. Cosine similarity measures the cosine of the angle between two vectors, providing a measure of their similarity. It is chosen for its effectiveness in high-dimensional spaces and its ability to handle sparse data.
- A similarity matrix is created to store the pairwise similarities between games. The similarity matrix is computed using PySpark's CartesianProduct and cosine similarity functions, leveraging the distributed computing capabilities of PySpark for efficient calculation.
- The similarity matrix is converted to a PySpark DataFrame for efficient lookup and retrieval of similarity scores.

3. Neighborhood Creation:

- The k-nearest neighbours (kNN) approach is used to create item neighbourhoods. kNN is a non-parametric algorithm that finds the k most similar items to a given item based on a similarity measure.
- For each game, the k most similar games are identified based on the cosine similarity scores stored in the similarity matrix.
- The item neighbourhoods are stored in a PySpark DataFrame, where each row represents a game and its corresponding k nearest neighbours. This allows for efficient retrieval of similar games during the recommendation process.

4. Recommendation Generation:

- When a player requests recommendations, their played games are retrieved from the player history table.
- For each played game, the recommender system looks up its neighbours in the item neighbourhoods DataFrame.
- The neighbouring games are ranked based on their similarity scores and filtered to remove games the player has already played. This ensures that the recommendations are personalized and relevant to the player's gaming history.
- The top-ranked games are recommended to the player. The number of recommendations can be customized based on the desired user experience and system requirements.

Model Validation

To validate the recommender system, a subset of the player history data can be held out as a test set. The recommender system generates recommendations for each player in the test set based on their played games. The generated recommendations are then compared against the actual games played by the player in the test set. Evaluation metrics such as precision, recall, and F1 score can be calculated to assess the performance of the recommender system.

- Precision measures the proportion of recommended games that are relevant to the player. It is calculated as the ratio of the number of relevant recommended games to the total number of recommended games.

- Recall measures the proportion of relevant games that are successfully recommended. It is calculated as the ratio of the number of relevant recommended games to the total number of relevant games in the test set.
- F1 score is the harmonic mean of precision and recall, providing a balanced measure of the recommender system's performance.

These evaluation metrics help quantify the effectiveness of the recommender system in providing relevant and personalized game recommendations to players.

Pros

- Quick and explainable: The content-based recommender system using game attributes is relatively straightforward to implement and can be easily explained to stakeholders, facilitating buy-in and understanding. The use of cosine similarity and kNN algorithm provides a transparent and intuitive approach to finding similar games based on their attributes.
- Personalized recommendations: By considering the attributes of the games a player has played, the system can generate personalized recommendations that align with the player's preferences. This personalization enhances the gaming experience and increases player engagement and satisfaction.
- Scalability: The use of PySpark and efficient data structures like the similarity matrix allows the recommender system to handle large-scale datasets and generate recommendations efficiently. PySpark's distributed computing capabilities enable the system to scale horizontally and process massive amounts of data in parallel.

Cons

- Limited to game attributes: The current MVP relies solely on game attributes for generating recommendations, which may not capture all the factors influencing player preferences, such as social interactions or temporal dynamics. Incorporating additional data sources and collaborative filtering techniques could enhance the recommendations and provide a more comprehensive understanding of player preferences.
- Cold-start problem: The recommender system may struggle to provide accurate recommendations for new players with limited playing history or new games with few similar games. Addressing the cold-start problem may require incorporating additional data sources, such as player demographics or game metadata, to provide meaningful recommendations for new entities.
- Lack of diversity: The content-based approach may lead to recommendations that are too similar to the games a player has already played, lacking diversity and potentially missing out on novel game experiences. Incorporating diversity-aware recommendation techniques or combining content-based filtering with other approaches, such as collaborative filtering or contextual information, could help introduce more diversity in the recommendations.

Continuous Improvement

It's important to emphasize that this MVP serves as a starting point for the recommender system. As more data becomes available and feedback is gathered from users, the model can be iteratively improved. Future iterations may incorporate collaborative filtering techniques to leverage the collective preferences of players, incorporate additional data sources such as user reviews or social media interactions, or explore more advanced machine learning algorithms like matrix factorization or deep learning to capture complex patterns and latent factors influencing player preferences.
By continuously iterating and refining the recommender system based on user feedback, performance metrics, and emerging techniques in the field of recommender systems, the model can evolve into a more sophisticated and accurate system that better captures player preferences and provides an enhanced gaming experience.

Deployment

From: Rules of Machine Learning: Best Practices for ML Engineering

"To make great products:

do machine learning like the great engineer you are, not like the great machine learning expert you aren't."

Automation (Let's call it Local Deployment):

(Please refer to Appendix B – Current Architecture Diagram (Local))

Code:

The code for each component is provided in the respective files:

- Database: init.sql and Dockerfile
- Recommender Model: recommender.py, model_requirements.txt, and Dockerfile
- API: api.py, api_requirements.txt, and Dockerfile
- Frontend: app.py, app_requirements.txt, and Dockerfile
- Orchestration: docker-compose.yml

The code is organized in a modular and reusable manner, following best practices for software development and containerization. Each component has its own Dockerfile that specifies the necessary dependencies, configurations, and runtime environment. The docker-compose.yml file orchestrates the deployment and interaction between the components.

Approach:

The approach to automating the game recommender system solution is to mimic a production environment by separating the components into their respective functions, namely: the Database, the Recommender model, the API, and the front end. The goal is to modularize and isolate the functionalities of these system components.

Pros:

1. Modularity and maintainability of the system.
2. Scalability and ability to independently scale components.
3. Fault isolation and improved system stability.
4. Flexibility in choosing suitable technologies for each component.
5. Streamlined development and testing processes.
6. Easy orchestration and deployment using Docker Compose.

Cons:

1. Increased complexity in managing multiple components and their interactions.
2. Overhead in terms of resource utilization due to running separate containers for each component.
3. Potential performance impact due to communication between components over the network.
4. Need for careful versioning and compatibility management between components.
5. Increased learning curve for developers to understand and work with the modularized architecture.

Despite the cons, the benefits of modularization, scalability, fault isolation, and technology independence outweigh the drawbacks in most scenarios. The provided code and Docker Compose configuration demonstrate how to structure and orchestrate the game recommender system using this approach.

MLOps Practices

To ensure the reliability, maintainability, and performance of the game recommender system in a production environment, the following MLOps practices can be implemented:

1. Version Control

- Use a version control system, such as Git, to track changes to the codebase, models, and configuration files.
- Implement a branching strategy to manage different stages of development, such as feature branches, development, staging, and production.
- Enforce code reviews and pull request processes to maintain code quality and collaborate effectively.

2. Continuous Integration and Continuous Deployment (CI/CD):

- Set up a CI/CD pipeline using tools like Jenkins, Azure DevOps, or GitLab CI/CD.
- Automate the build, testing, and deployment processes to ensure consistent and reliable deployments.
- Implement automated tests, including unit tests, integration tests, and end-to-end tests, to catch bugs and regressions early in the development cycle.
- Use containerization technologies, such as Docker, to package the application and its dependencies for consistent deployment across different environments.

3. Model Versioning and Tracking:

- Implement a model versioning system to keep track of different versions of the trained models.
- Use tools like MLflow or Azure ML to track model lineage, hyperparameters, and performance metrics.
- Store trained models in a model registry for easy retrieval and deployment.

4. Monitoring and Logging:

- Implement monitoring and logging mechanisms to track the performance and health of the recommender system in production.
- Use tools like Prometheus and Grafana for collecting and visualizing metrics, such as request latency, error rates, and resource utilization.
- Set up alerts and notifications to proactively identify and address issues or anomalies.
- Implement centralized logging using tools like ELK stack (Elasticsearch, Logstash, Kibana) to collect and analyze application logs.

5. Data and Model Validation:

- Establish data validation processes to ensure the quality and integrity of the input data.
- Implement data pipelines that validate and preprocess the data before feeding it into the recommender system.
- Perform regular model validation and testing to assess the performance and accuracy of the deployed models.

- Use techniques like A/B testing or shadow testing to compare the performance of new models against the existing ones before fully deploying them.

6. Infrastructure as Code (IaC):

- Use infrastructure as code practices to define and manage the infrastructure resources required for the recommender system.
- Leverage tools like Terraform or Azure Resource Manager templates to provision and configure resources in a declarative and repeatable manner.
- Version control the infrastructure code alongside the application code to ensure consistency and reproducibility.

7. Security and Compliance:

- Implement security best practices, such as encryption of sensitive data, secure communication channels, and access controls.
- Ensure compliance with relevant regulations and standards, such as GDPR or HIPAA, depending on the industry and data being processed.
- Regularly perform security audits and vulnerability assessments to identify and mitigate potential risks.

8. Continuous Learning and Improvement

- Foster a culture of continuous learning and improvement within the team.
- Encourage experimentation and iteration to refine the recommender system based on user feedback, new data, and emerging technologies.
- Regularly review and update the MLOps practices to align with industry best practices and evolving requirements.

By adopting these MLOps practices, the game recommender system can be deployed, monitored, and maintained effectively in a production environment. It ensures the system's reliability, scalability, and performance while enabling continuous improvement and adaptation to changing business needs.

The Way Forward:

1. Gathering Business Insights:

- Engage with stakeholders, analysts, and data scientists to understand the core problem the recommender system aims to solve and the desired outcomes. Conduct workshops and interviews to gather domain knowledge and identify potential features that can enhance the recommender system.
- Collaborate with business experts to gain insights into user behaviour, game characteristics, and any other relevant factors that can influence recommendations. Document and incorporate the gathered insights into the feature engineering process and model development.

2. Model Optimization and Evaluation:

- Clearly define the objective function and metrics that the recommender system should optimize for, such as user engagement, retention, or revenue. Develop and implement evaluation methods to assess the model's performance against these metrics.
- Utilize business understanding to design measures that can detect model shortcomings and identify areas for improvement. Continuously monitor and evaluate the model's performance in production to ensure it aligns with business goals and user expectations.

3. Production System Research and Analysis:

- Conduct thorough research and analysis of production recommender systems to identify best practices, architectures, and technologies used in the industry. Explore scalability considerations, such as handling large-scale data, real-time recommendations, and model updates.
- Investigate techniques for model versioning, A/B testing, and incremental model updates in a production environment. Document the findings and incorporate them into the system design and implementation plan.

4. Cost Exploration and Optimization:

- Analyze the cost implications of running the recommender system in Azure's cloud environment. Identify the major cost drivers, such as compute resources, storage, and data transfer.
- Explore cost optimization strategies.

Proposed Solutions Architecture

(Please refer to Appendix C - Proposed Architecture Diagram)

Overview

The proposed solutions architecture is designed to deploy a scalable and efficient game recommender system based on content filtering. Leveraging isolated containers for system components and edge caching for recommendations, the architecture aims to minimize latency and optimize resource utilization. The system operates with static files for game attributes, periodically updating recommendations with new game additions.

Components

1. **Database**: Hosts game attribute data and player history records. Utilizes an autoscaling group to manage potential high loads efficiently.
2. **Recommender Model**: Trains and generates recommendations based on game attributes. Updates recommendations periodically and pushes them to edge locations for caching.
3. **API**: Handles interactions with players, populates player history records, and supports potential feature enrichment through feedback mechanisms.
4. **Frontend**: Provides a user interface for players to receive and interact with recommendations.

Workflow

1. **Data Population and Training**: Game attribute data is periodically updated in the database. The recommender model retrains based on this updated data, generating new recommendations.
2. **Edge Caching**: New recommendations are pushed to edge locations/content delivery networks for caching, reducing latency for player requests.
3. **API Interaction**: Players interact with the system through the API, which populates player history records and may facilitate feature enrichment through feedback mechanisms.
4. **Monitoring and Metrics**: All components are monitored for usage, cost, and impact. Metrics are collected to evaluate recommendation effectiveness and system performance.

Conclusion

The proposed solutions architecture prioritizes simplicity, cost-effectiveness, and scalability while aiming to provide valuable game recommendations to players. By leveraging static files for game attributes, periodic retraining, edge caching, and potential feature enrichment, the architecture sets a foundation for iterative improvement and adaptation based on user feedback and emerging research. Monitoring and metrics play a crucial role in gauging recommendation effectiveness and ensuring system reliability. As the system evolves, further iterations can explore advanced features, models, and research to enhance recommendation quality and player satisfaction.

Through this architecture, the game recommender system aims to continuously deliver personalized gaming experiences, driving player engagement and loyalty in the fast-paced gaming industry.

Appendix A: Synthetic Data Generation Code

This code generates a synthetic dataset for a gaming application. The dataset consists of three main tables:

1. Game Attributes Table: Contains attributes and metadata about the games.
2. Player Attribute Table: Contains attributes about the players.
3. Player History Table: Represents the players' game history, including the games they've played and the associated metadata.

DataGenerator Class

Initialization

The DataGenerator class is initialized with the following attributes:

- game_attributes_metadata: A dictionary containing metadata about the game attributes. It specifies the expected values or data types for each attribute. The attributes include:
  - game_id: Integer
  - game_name: String
  - wild_symbols: List of possible values ("regular", "expanding", "sticky")
  - scatter_symbols: List of possible values ("yes", "no")
  - free_spins: Integer
  - bonus_rounds: List of possible values ("yes", "no")
  - multipliers: Integer
  - jackpots: List of possible values ("fixed", "progressive")
  - paylines: Integer
  - reel_mechanisms: List of possible values ("cascading", "megaways")
  - gamble_feature: List of possible values ("yes", "no")
  - RTP: Float
  - volatility: List of possible values ("high", "medium", "low")
  - themes: String
  - mystery_symbols: List of possible values ("yes", "no")
  - random_triggers: List of possible values ("yes", "no")

- game_attributes_table: Initially set to None, it will hold the generated game attributes table.
- player_attribute_table: Initially set to None, it will hold the generated player attribute table.
- player_history_table: Initially set to None, it will hold the generated player history table.

Methods

- start_synthetic_data_generation(num_rows=1000, num_players=200, max_games_played=30, override=None):
  - This method kicks off the synthetic data generation process.
  - It takes optional parameters:
    - num_rows: Number of rows/games in the game attribute table (default: 1000).
    - num_players: Number of players in the dataset (default: 200).
    - max_games_played: Maximum number of games played by each player (default: 30).

  - override: If set to "player_attribute" or "game_attribute", it skips the corresponding table population step (not implemented yet).
  - It calls the populate_game_attributes_table(), populate_player_attribute_table(), and populate_player_history_table() methods to generate the respective tables.
  - Finally, it saves the generated tables as CSV files in the "backend/data" directory.

- populate_game_attributes_table(num_rows):
  - This method populates the game attributes table with synthetic data.
  - It retrieves game names using the get_game_names() method.
  - It generates random values for other game attributes based on the metadata defined in game_attributes_metadata.
  - It creates a DataFrame with the generated game attribute data and returns it.

- populate_player_attribute_table(num_players=200):
  - This method populates the player attribute table with synthetic data.
  - It generates player IDs, first names, and last names for the specified number of players.
  - It creates a DataFrame with the generated player attribute data and returns it.

- populate_player_history_table(max_games_played, same_machine_probability=0.7):
  - This method populates the player history table with synthetic data.
  - For each player, it generates a random number of games played (up to max_games_played).
  - It selects game IDs randomly from the game attribute table, considering the probability of using the same machine for the next game (same_machine_probability).
  - It creates a DataFrame with the generated player history data and returns it.

- get_game_names():
  - This method retrieves game names for populating the game attribute table.
  - It checks if a cached file ("backend/data/web_scraping_cache.pkl") exists. If it does, it loads the game names from the cache.
  - If the cache file doesn't exist, it calls the web scraping functions (scrape_first_site(), scrape_second_site(), scrape_third_site()) to retrieve game names from online slots websites.
  - It combines the scraped game names, removes duplicates, and shuffles them randomly.
  - It saves the game names to the cache file for future use and returns the list of game names.

Web Scraping Module

The web scraping module (web_scraping.py) contains three functions to retrieve game names from online slots websites:

- scrape_first_site(url):
  - This function scrapes game names from the website "rwnewyork.com".
  - It sends a GET request to the specified URL and parses the HTML content using the lxml library.
  - It extracts the game names from the parsed HTML using XPath and returns them as a list.

- scrape_second_site(url):
  - This function scrapes game names from the website "ybrcasinoandsportsbook.com".
  - Similar to scrape_first_site(), it sends a GET request, parses the HTML content, and extracts the game names using XPath.
  - It returns the game names as a list.

- scrape_third_site(url):
  - This function scrapes game names from the website "casino.guru".
  - It sends GET requests to multiple pages of the website by iterating over a range of page numbers.
  - For each page, it parses the HTML content, finds the game names using XPath, and appends them to a list.
  - It returns the combined list of game names.

The scraped game names are used by the get_game_names() method in the DataGenerator class to populate the game attribute table.

Main Script

The main script (main.py) demonstrates how to use the DataGenerator class to generate the synthetic dataset:

1. It creates an instance of the DataGenerator class.
2. It calls the start_synthetic_data_generation() method with default parameters to start the synthetic data generation process.
3. It accesses the generated tables (game_attributes_table, player_attribute_table, player_history_table) from the DataGenerator instance.
4. It prints the first few rows of each generated table using the head() method.

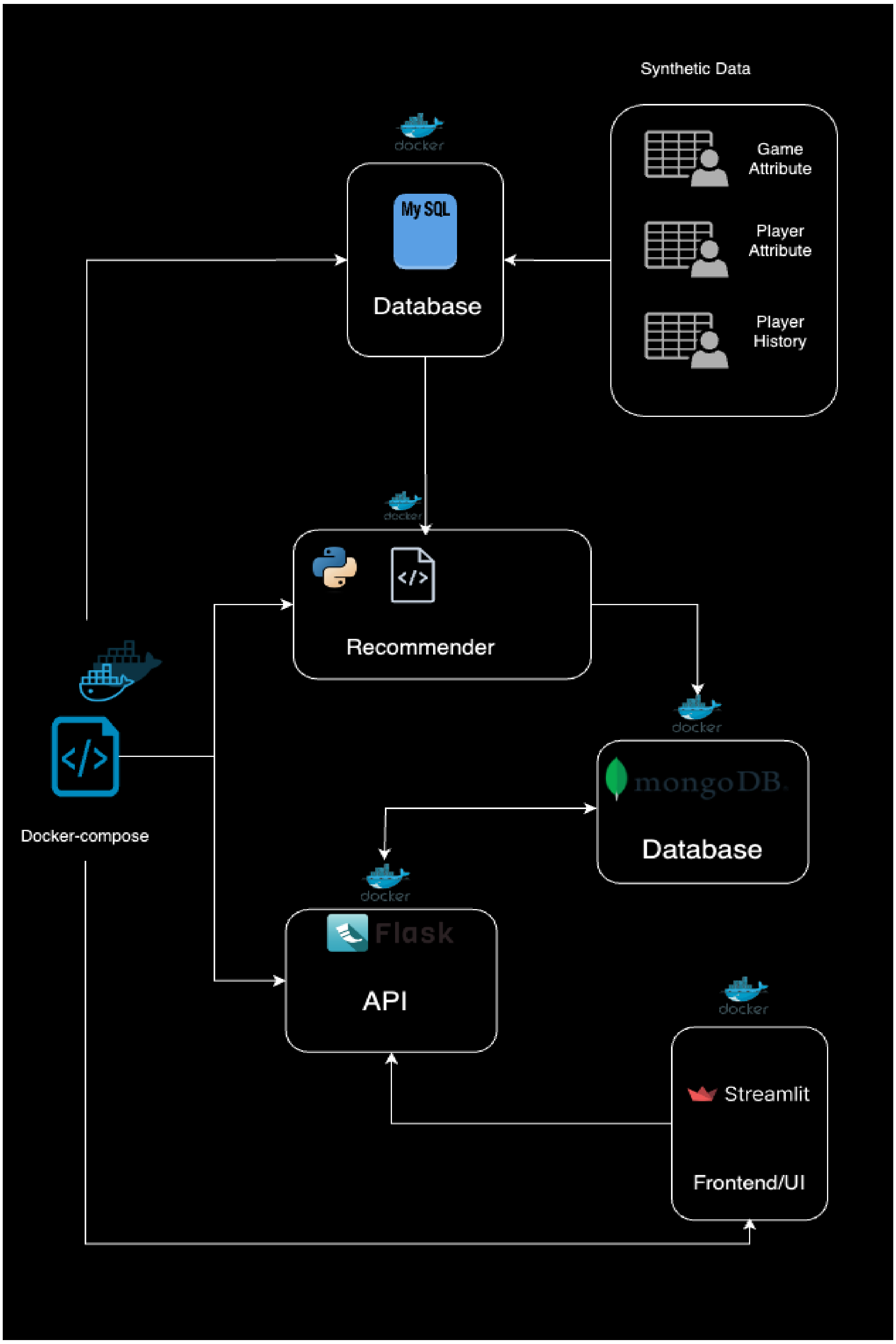
Usage

To generate the synthetic dataset, run the main.py script. It will create an instance of the DataGenerator class, start the data generation process, and print the generated tables.

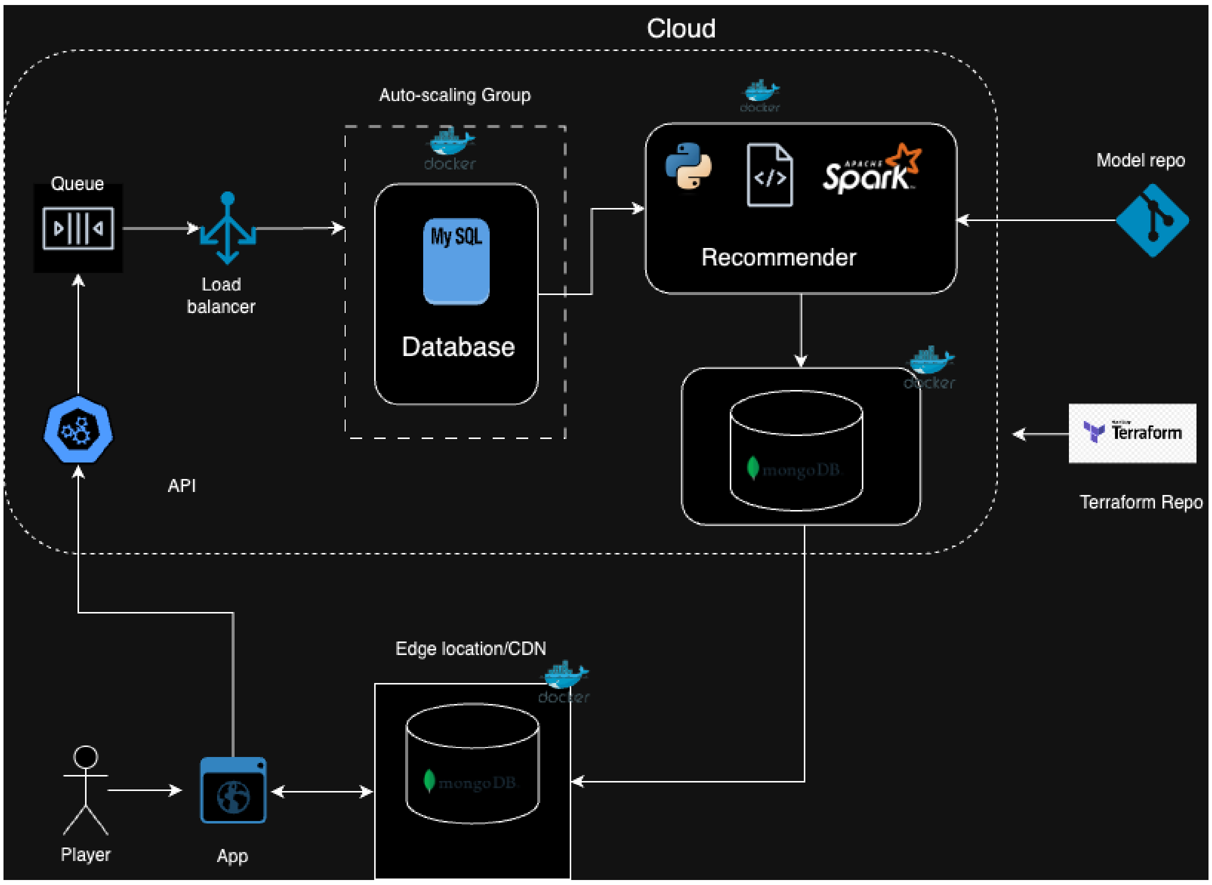
The generated tables will be saved as CSV files in the "backend/data" directory:
- game_attribute.csv: Contains the generated game attribute data.
- player_attribute.csv: Contains the generated player attribute data.
- player_history.csv: Contains the generated player history data.

These CSV files can be used for further analysis, modelling, or any other purposes related to the gaming application.

Appendix B: Recommender - Current Architecture (Local)



Appendix C: Recommender – Proposed Architecture


Cloud
Auto-scaling Group
docker
Queue
Load balancer
My SQL
Database
docker
Recommender
Model repo
docker
mongoDB
Terraform
Terraform Repo
API
Edge location/CDN
docker
mongoDB
Player
App