**AYDIN ADNAN MENDERES UNIVERSITY**

**ENGINEERING FACULTY**

**COMPUTER SCIENCE ENGINEERING DEPARTMENT**

**Term Project   
Recommendation System with Spark  
CSE424 Big Data Analysis, Spring 2024/2025 Student’s Number - Student’s Name SURNAME:**

**2118050076 - Mehmet ÖZCAN**

**211805036 - Ahmet Muhammed AYDIN**

**211805008 - Fedai PAÇA**

**Lecturer:**

**Asst. Prof. Dr.Hüseyin ABACI**

### GitHub Repository Url

<https://github.com/fedaipaca/myanimelist-recommender>

### Anime Recommendation System with Apache Spark

A collaborative filtering-based anime recommendation system built using Apache Spark's ALS (Alternating Least Squares) algorithm. This project analyzes anime ratings from MyAnimeList dataset to provide personalized recommendations and discovers similar content through matrix factorization techniques.

#### Installation

To set up the environment and run the project, follow these steps:

#### Prerequisites

**Java 21**Install Java Development Kit (JDK) 21 from [here](https://www.oracle.com/tr/java/technologies/downloads/#java21).  
# Check Java version

java -version

**Python 3.12.10+**# Check Python version

python --version

Project contains a .python-version file to ensure compatibility with Python 3.12.10. You can use tools like pyenv to manage Python versions easily.

#### Required Python Packages

You will find the requirements.txt file in the project root directory. Install the required packages using pip:

pip install -r requirements.txt

#### Main packages and Libraries are used in this project:

* pyspark: For distributed data processing and machine learning
* pandas: For data manipulation and analysis
* numpy: For numerical operations
* matplotlib: For data visualization
* seaborn: For statistical data visualization
* jupyter: For interactive notebooks
* findspark: For Spark session management in Jupyter notebooks
* ipykernel: For Jupyter kernel management
* pyarrow: For saving and loading Parquet files

#### Why no need to install standalone Spark:

* PySpark Includes Spark Jars: When you install PySpark via pip, it comes with all the necessary Spark JAR files bundled together.
* findspark Handles Discovery: The findspark.init() automatically locates your PySpark installation and sets up the SPARK\_HOME environment variable.
* Local Mode (local[\*]): You're running Spark in local mode, which means:
  + No cluster manager needed
  + No separate worker nodes
  + Everything runs in your Python process
  + Uses all your CPU cores (\*)

Even this is enough for local development, using standalone Spark installation comes with benefits, of distributed computing, better memory management, fault tolerance etc.

### Dataset Information

The project uses three main datasets from MyAnimeList.  
Original dataset can be found in [Kaggle MyAnimeList Dataset.](https://www.kaggle.com/datasets/azathoth42/myanimelist)

Original dataset are in csv format, but we converted them to Parquet format for better performance and efficiency in Spark.

About Parquet format:  
Parquet is a columnar storage file format optimized for use with big data processing frameworks like Apache Spark. It provides efficient data compression and encoding schemes, which significantly reduce the amount of disk space required and improve query performance. Parquet files are designed to work well with complex nested data structures, making them ideal for analytical workloads.

Original dataset is around 3gb. But after saving them in Parquet format, the size is reduced to around 800mb.

#### 1. Users Dataset (users.parquet)

**Size**: 108711 users

**Columns**:

* **username**: Display name of the user (nullable = true)
* **user\_id**: Unique identifier for the user (nullable = true)
* **user\_watching**: Number of anime the user is currently watching (nullable = true)
* **user\_completed**: Number of anime the user has completed (nullable = true)
* **user\_onhold**: Number of anime the user has put on hold (nullable = true)
* **user\_dropped**: Number of anime the user has dropped (nullable = true)
* **user\_plantowatch**: Number of anime the user plans to watch (nullable = true)
* **user\_days\_spent\_watching**: Total days the user has spent watching anime (nullable = true)
* **gender**: Gender of the user (nullable = true)
* **location**: Location of the user (nullable = true)
* **birth\_date**: Birth date of the user (nullable = true)
* **access\_rank**: User's access rank on the platform (nullable = true)
* **join\_date**: Date the user joined the platform (nullable = true)
* **last\_online**: Date the user was last online (nullable = true)
* **stats\_mean\_score**: Mean score of all ratings given by the user (nullable = true)
* **stats\_rewatched**: Number of anime the user has rewatched (nullable = true)
* **stats\_episodes**: Total number of episodes watched by the user (nullable = true)

### 2. Anime Dataset (animes.parquet)

**Size**: 6668 anime entries

**Columns**:

* **anime\_id**: Unique identifier for each anime (nullable = true)
* **title**: Main title of the anime (nullable = true)
* **title\_english**: English title of the anime (nullable = true)
* **title\_japanese**: Japanese title of the anime (nullable = true)
* **title\_synonyms**: Alternative titles or synonyms for the anime (nullable = true)
* **image\_url**: URL of the anime's cover image (nullable = true)
* **type**: Type of the anime (e.g., TV, Movie, OVA) (nullable = true)
* **source**: Source material for the anime (e.g., Manga, Light Novel) (nullable = true)
* **episodes**: Total number of episodes (nullable = true)
* **status**: Current status of the anime (e.g., Finished Airing, Currently Airing) (nullable = true)
* **airing**: Boolean indicating if the anime is currently airing (nullable = true)
* **aired\_string**: Human-readable airing period (nullable = true)
* **aired**: Detailed airing period in structured format (nullable = true)
* **duration**: Duration of each episode (nullable = true)
* **rating**: Content rating of the anime (e.g., PG-13, R) (nullable = true)
* **score**: Average user score for the anime (nullable = true)
* **scored\_by**: Number of users who scored the anime (nullable = true)
* **rank**: Rank of the anime based on score (nullable = true)
* **popularity**: Popularity rank of the anime (nullable = true)
* **members**: Number of members who added the anime to their list (nullable = true)
* **favorites**: Number of users who marked the anime as a favorite (nullable = true)
* **background**: Background information or trivia about the anime (nullable = true)
* **premiered**: Season and year the anime premiered (nullable = true)
* **broadcast**: Scheduled broadcast time (nullable = true)
* **related**: Information about related anime (e.g., sequels, spin-offs) (nullable = true)
* **producer**: Production company or entity responsible for the anime (nullable = true)
* **licensor**: Licensing company for the anime (nullable = true)
* **studio**: Studio responsible for the anime's animation (nullable = true)
* **genre**: Genres associated with the anime (nullable = true)
* **opening\_theme**: Opening theme song(s) of the anime (nullable = true)
* **ending\_theme**: Ending theme song(s) of the anime (nullable = true)
* **duration\_min**: Duration of each episode in minutes (nullable = true)
* **aired\_from\_year**: Year the anime started airing (nullable = true)

### 3. User-Anime Ratings Dataset (users\_animes.parquet)

**Size**: 31284030 ratings

**Columns**:

* **username**: Display name of the user (nullable = true)
* **anime\_id**: Unique identifier for the anime (nullable = true)
* **my\_score**: Rating given by the user to the anime (nullable = true)

## Project Structure

* notebooks/ contains the Jupyter notebook with the complete project code and analysis.
* data/ contains the Parquet files for users, anime, and user-anime ratings.
* trained\_models/ contains the saved best performing ALS model.
* results\_df.json contains the final results of the model predictions and recommendations.

## Requirements for this project

* Present/show your computer info (with code) IP address, name and configuration of your PC etc.
* Use spark dataframes.
* Make some EDA (Exploratory Data Analysis) on your dataset.
* Use at least one map() and reduce() (derivatives i.e. reduceByKey) functions.
* Use 70% of dataset for training, 30% for testing.
* Use ALS (Alternating Least Squares) for training recommendation model with last 4 digit of your student number as a “seed” value.
* Also change the parameters of ALS re-run the algorithm for parameters “rank” (10, 50, 200), “iteration” (10, 50, 200) and “lambda” (0.01, 0.1). This means 18 different model will be created using specified rank-iteration-lambda values.
* Run ALS with different parameters and visualize the comparative performance result of different parameters with a plot in the program you wrote (i.e. pyspark - Jupiter notebook).
* Find and present MSE (Mean Squared Error), RMSE (Root Mean Squared Error) for performance evaluation of each model and explain them, indicate best model for your dataset, explain why. Plot a graph of all RMSE values according to changing iteration or rank values.
* Make prediction with ALS and compare it with the original values (with real values) side by side.
* Cosine similarity between ALS model and the products should be found then, 10 users who will like product X the most should be determined.

#### Spark Configurations

We used the configurations below for starting the spark session.

appName("AnimeRecommender"): Sets the name of your Spark application that will appear in the Spark UI and logs. This helps identify your application when monitoring.

master("local[\*]"): Specifies the cluster manager. local[\*] means run Spark locally using all available CPU cores on your machine.

spark.driver.memory: Allocates 6GB of RAM to the driver program (your main application). The driver coordinates the execution and stores results.

spark.memory.fraction: Sets 70% of the heap memory for execution and storage. The remaining 30% is reserved for user data structures and internal metadata.

spark.sql.shuffle.partitions: Sets the number of partitions to use when shuffling data for joins or aggregations. Default is 200, but 100 is often better for smaller datasets.

spark.rdd.compress: Enables compression of RDD data when stored in memory, saving space at the cost of some CPU overhead.

spark.network.timeout: Maximum time (800 seconds) to wait for network operations. Prevents timeouts during large data transfers.

spark.executor.heartbeatInterval: How often (30 seconds) executors send heartbeats to the driver to indicate they're still alive.

spark.driver.extraJavaOptions: Sets JVM stack size to 12MB for the driver. Prevents stack overflow errors in complex operations.

spark.executor.extraJavaOptions: Sets JVM stack size to 12MB for executors. Important for deep recursion or complex operations.

spark.serializer: Uses Kryo serializer instead of Java serialization. Kryo is faster and more compact.

spark.kryoserializer.buffer.max: Sets maximum buffer size (1GB) for Kryo serialization. Prevents buffer overflow with large objects.

spark.kryo.unsafe: Enables unsafe operations in Kryo for better performance, but with slightly less safety checks.

spark.kryo.referenceTracking: Disables reference tracking in Kryo, improving performance when you don't have circular references.

spark.cleaner.referenceTracking.cleanCheckpoints: Automatically cleans up checkpoint files when they're no longer needed.

spark.cleaner.periodicGC.interval: Triggers garbage collection every 30 seconds to clean up unused RDDs and shuffle data.

setCheckpointDir: Sets directory for storing checkpoint files (used for fault tolerance and breaking lineage chains).

setLogLevel("ERROR"): Only shows error messages in logs, reducing verbosity during development.



## Methodology

Our recommendation system follows a systematic approach using collaborative filtering:

### 1. Collaborative Filtering with ALS

* Uses matrix factorization to decompose the user-item rating matrix
* Learns latent (hidden, generated) features for both users and anime

**What is Collaborative Filtering?**Collaborative filtering is a technique used in recommendation systems to predict a user's interests by collecting preferences from many users. It assumes that if two users agree on one issue, they are likely to agree on others as well. Collaborative filtering can be user-based (finding similar users) or item-based (finding similar items).  
A basic example is Netflix recommending movies based on what similar users have watched and rated highly.

**What is ALS?**In a user-movie ratings table, some users may have rated some movies, while others may not have rated those movies at all. ALS fills in these missing ratings by learning patterns from the existing ratings. It does this by breaking down the large user-movie matrix into two smaller matrices: one representing users and another representing movies. Each user and movie is represented by a set of latent features (hidden characteristics).

### 2. Cosine Similarity Analysis

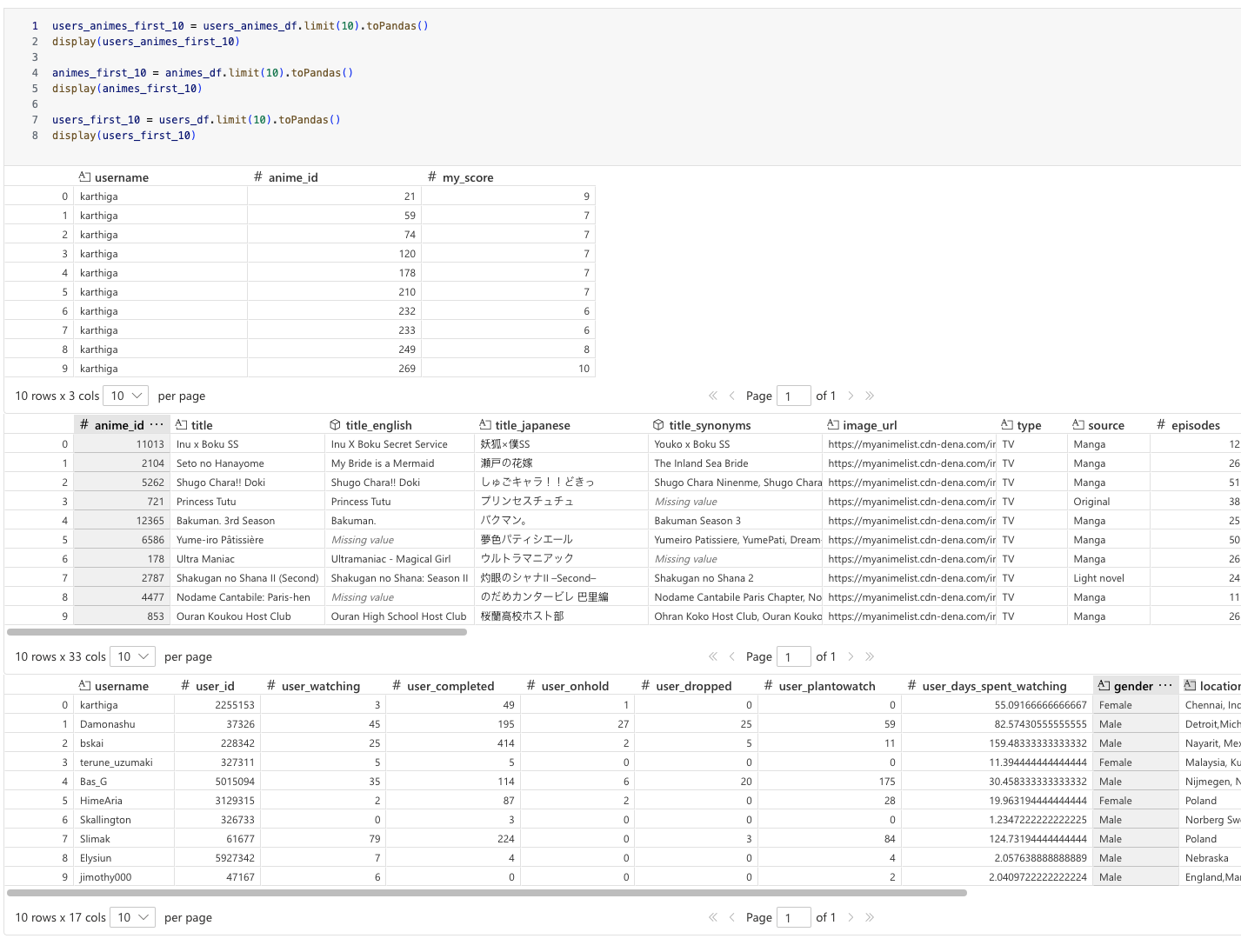
* Calculates similarity between anime based on learned features
* Enables content-based recommendations

### 3. Parameters We Used

* **Rank**: Number of latent factors (10, 50, 200)  
  What is Rank?  
  Rank in the context of ALS (Alternating Least Squares) refers to the number of latent factors or dimensions used to represent users and items (anime in this case). A higher rank allows the model to capture more complex relationships between users and items, but it also increases the risk of overfitting if the dataset is sparse. In our project, we tested ranks of 10, 50, and 200 to find the optimal balance between model complexity and performance.
* **Iterations**: Number of training iterations (10, 50, 200)  
  What is Iteration?  
  Iteration in the context of ALS (Alternating Least Squares) refers to the number of times the algorithm updates the user and item latent factor matrices during training. Each iteration refines the model's predictions by minimizing the error between predicted and actual ratings. More iterations can lead to better convergence and improved model performance, but they also increase computational time. In our project, we tested iterations of 10, 50, and 200 to evaluate their impact on model accuracy.
* **Regularization (Lambda)**: Controls overfitting (0.01, 0.1)  
  What is Regularization (or Lambda)?  
  Regularization in the context of ALS (Alternating Least Squares) is a technique used to prevent overfitting by adding a penalty term to the loss function. It discourages the model from fitting too closely to the training data, which can lead to poor generalization on unseen data. The regularization parameter (lambda) controls the strength of this penalty. A higher lambda value increases the penalty, while a lower value allows more flexibility in fitting the data. In our project, we tested regularization values of 0.01 and 0.1 to find the optimal balance between model complexity and performance.

## Data Loading

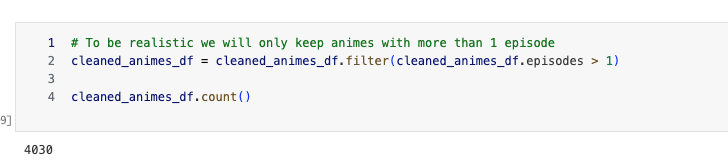
We used spark instance to load parquet files. A visual representation of each data file can be found below.



## Data Preprocessing

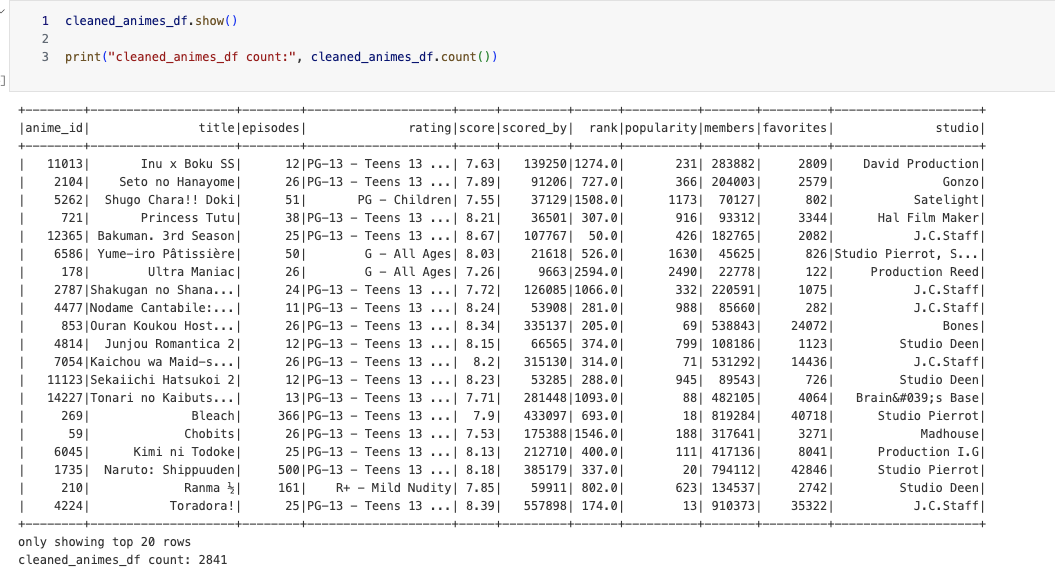
### Cleaning Steps

1. **Anime Filtering**:
   * Removed anime with ≤1 episode (movies excluded for simplicity).



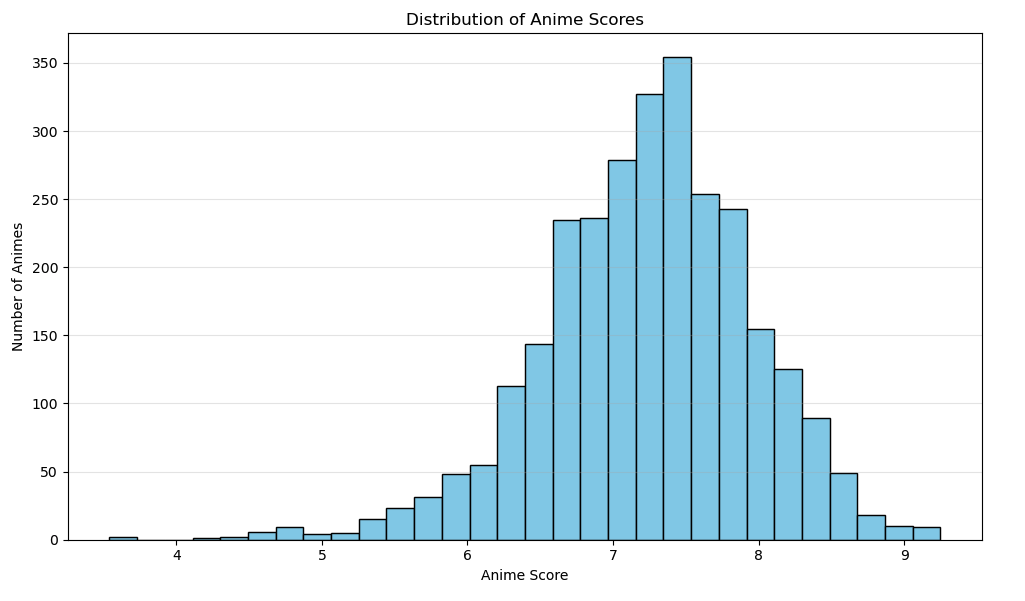
* + Filtered anime with <5,000 members (popularity threshold)
  + Removed anime with score <6.0 (quality threshold)
  + Filtered anime with <1,000 ratings (reliability threshold)

1. **User Filtering**:
   * Removed users with <5 completed anime (experience threshold)

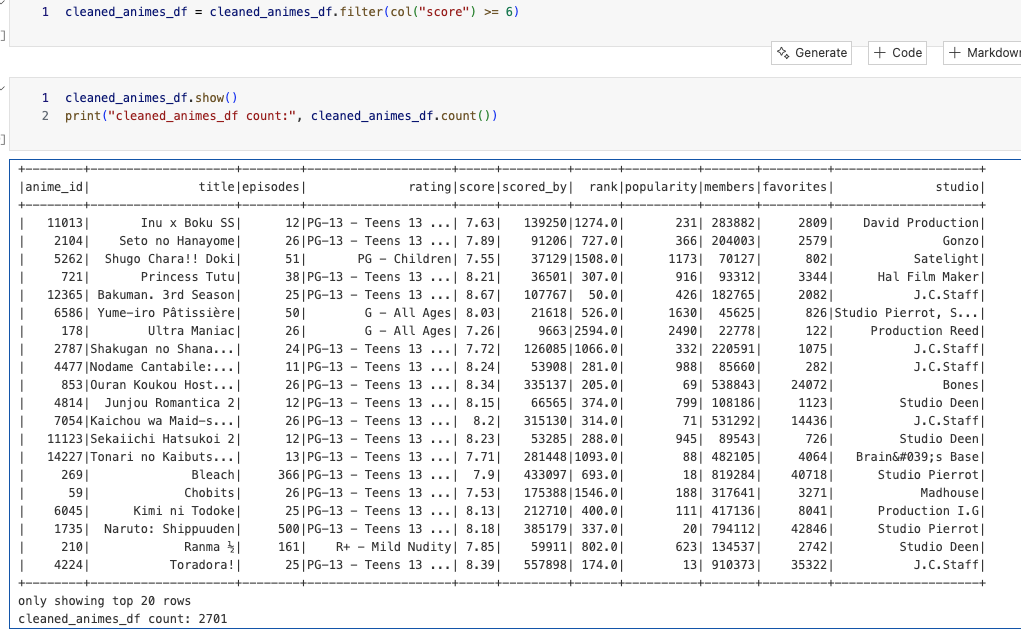


* + Ensured users exist in the ratings dataset

1. **Rating Filtering**:
   * Score distribution



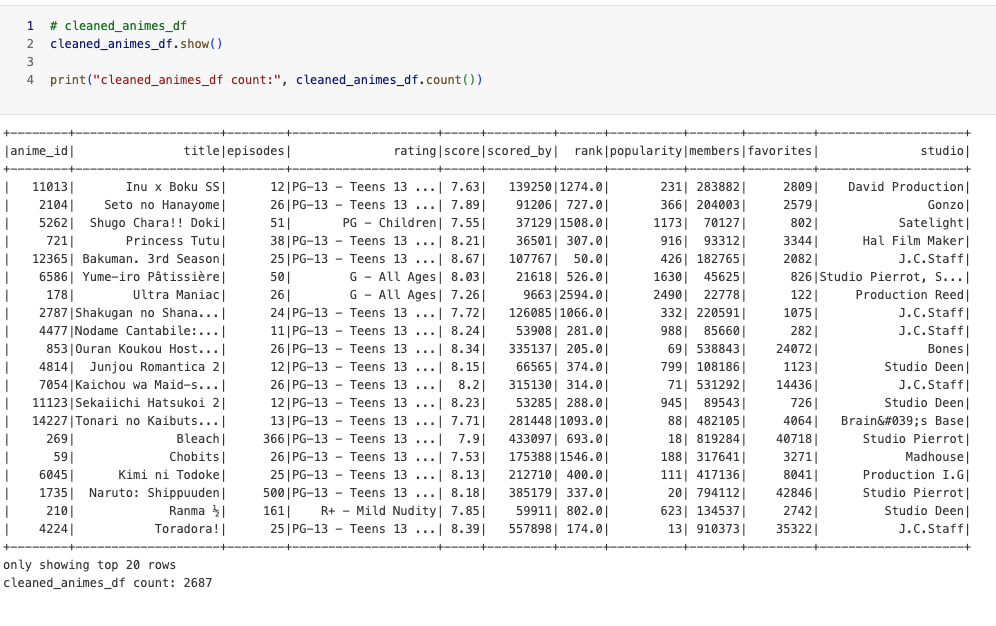
* + Removed ratings <5 (focusing on positive recommendations)



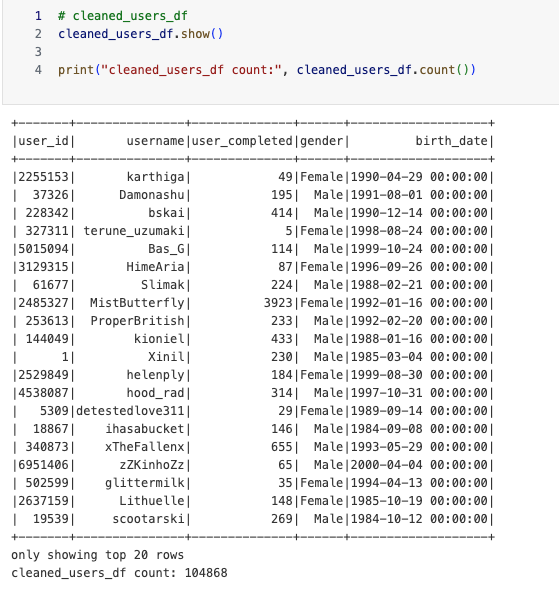
* + Filtered null values and inconsistent data

### Datasets after clean up

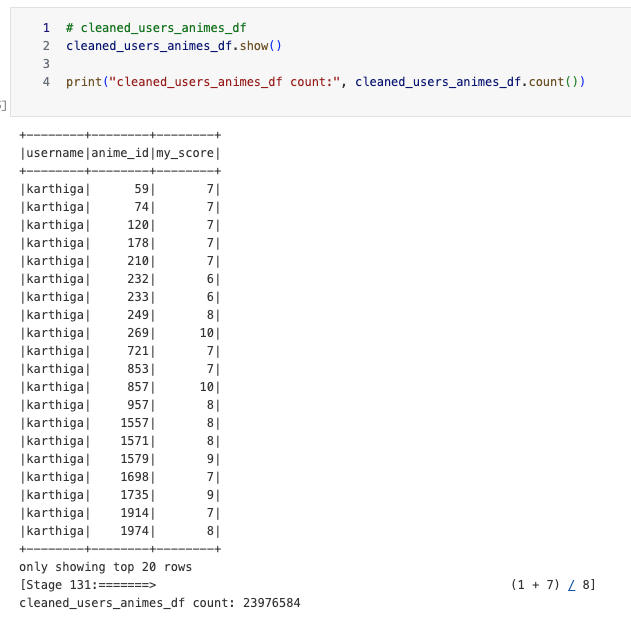
Animes



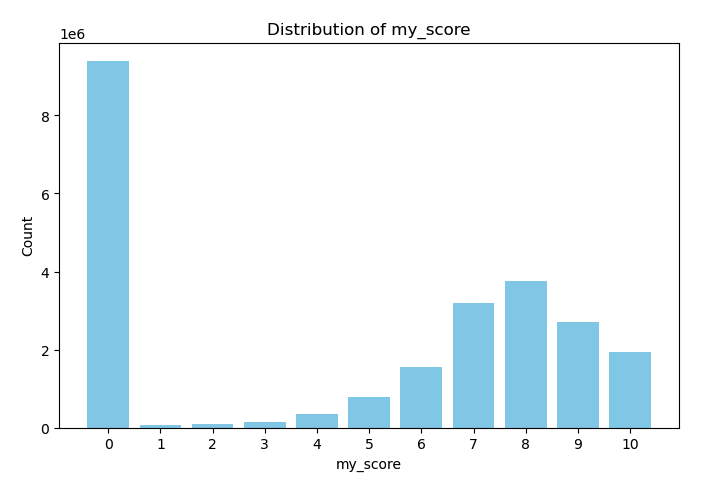
Users



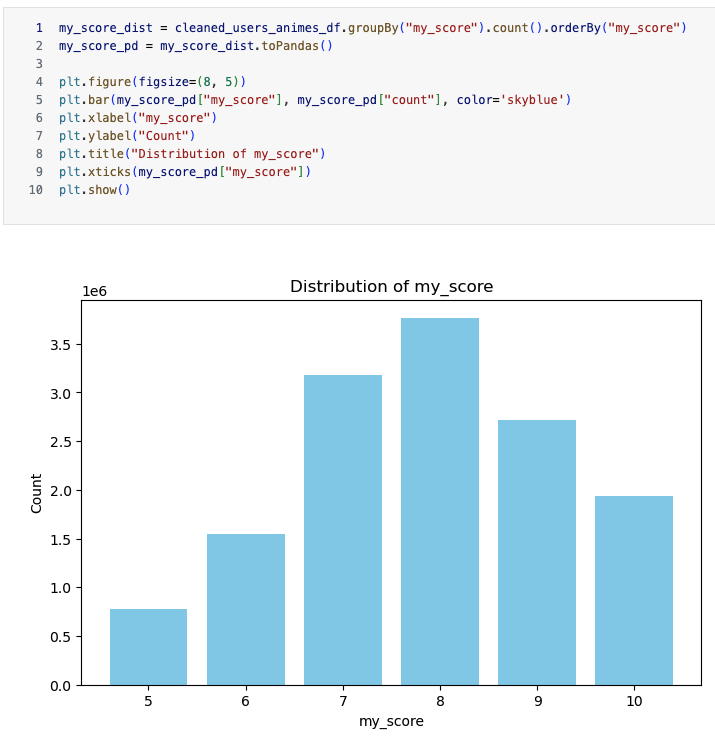
Animes-Users (aka Ratings)



**Distribution of Scores**



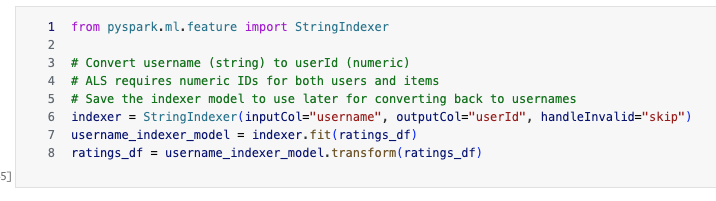
After dropping some unrealistic scores



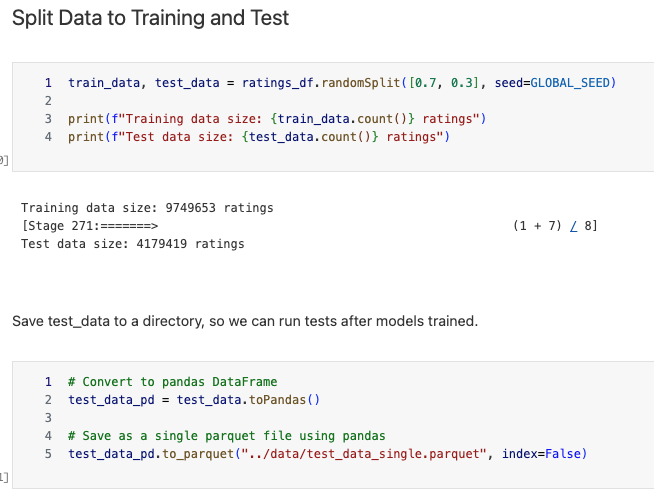
### Training and Testing Split

* **70% Training Set**: Used for model training  
  Training data size: 9749653 ratings
* **30% Testing Set**: Used for model evaluation  
  Testing data size: 4179419 ratings

Using StringIndexer to convert username to numeric values



Splitting dataset to training and test and caching for faster usage in spark



## Models Training with different parameters

### Training Process

Core logic of training with different parameters

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.recommendation import ALS

import time

# We'll try different combinations of based on requirements:

# - rank: number of latent (secret) factors (hidden features) in the model

# - iterations: how many times to optimize the model

# - lambda: regularization parameter to prevent overfitting

ranks = [10, 50, 200]

iterations = [10, 50, 200]

lambdas = [0.01, 0.1]

# will save the results of each model

results = []

best\_model = None

# create a RegressionEvaluator to evaluate the model performance

evaluator\_rmse = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")

evaluator\_mse = RegressionEvaluator(metricName="mse", labelCol="rating", predictionCol="prediction")

print("Starting ALS hyperparameter tuning loop...")

print(f"Will train {len(ranks) \* len(iterations) \* len(lambdas)} models...")

# Loop through all parameter combinations (18 total models)

for rank in ranks:

for num\_iterations in iterations:

for reg\_param in lambdas:

# Create the ALS model with current parameters

# Parameters explained:

# - maxIter: maximum iterations for optimization

# - rank: number of latent factors (model complexity)

# - regParam: regularization parameter (prevents overfitting)

# - coldStartStrategy="drop": ignore users/items with no training data

# - nonnegative=True: ensures predictions are non-negative

als = ALS(

maxIter=num\_iterations,

rank=rank,

regParam=reg\_param,

userCol="userId",

itemCol="itemId",

ratingCol="rating",

coldStartStrategy="drop",

nonnegative=True,

)

print(f"Training model with: rank={rank}, iterations={num\_iterations}, lambda={reg\_param}")

start\_time = time.time()

model = als.fit(train\_data)

end\_time = time.time()

training\_time = end\_time - start\_time

print(f"Model trained in {training\_time:.2f} seconds")

print("Making predictions on test data...")

predictions = model.transform(test\_data)

print("Predictions made.")

# Note: `coldStartStrategy="drop"` handles users/items in test set but not in training.

# However, predictions can still be NaN if the model couldn't generate one for a valid pair.

# RegressionEvaluator will error out on NaNs, so we must drop them.

predictions\_clean = predictions.na.drop(subset=["prediction"])

# Evaluate the model

rmse = evaluator\_rmse.evaluate(predictions\_clean)

mse = evaluator\_mse.evaluate(predictions\_clean)

print(f"RMSE = {rmse:.4f} (lower is better)")

print(f"MSE = {mse:.4f} (lower is better)")

# Check if this model is the best so far

# if best\_model is None then set it to the current model (for first iteration)

if best\_model is None:

best\_model = model

save\_model(best\_model)

else:

# get the previous best RMSE from results

prev\_best\_rmse = min([result["rmse"] for result in results]) if results else float("inf")

# if current rmse is lower than the previous best RMSE, then this is the new best model

if rmse < prev\_best\_rmse:

best\_model = model

save\_model(best\_model)

print("Found a new best model!")

# Store the results for this iteration

results.append(

{

"rank": rank,

"iterations": num\_iterations,

"lambda": reg\_param,

"rmse": rmse,

"mse": mse,

"training\_time": training\_time,

}

)

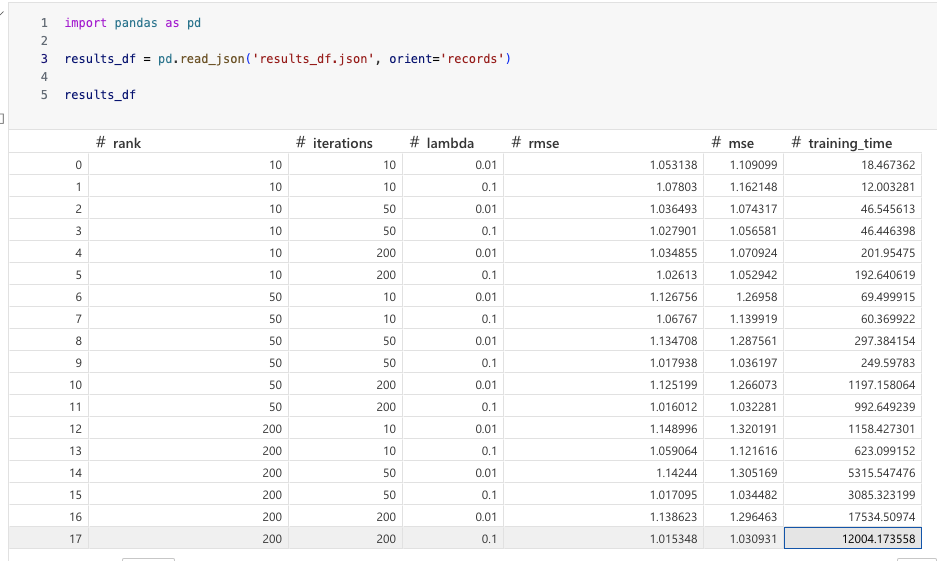
print("-" \* 50)

print("\nHyperparameter tuning finished.")

After each training iteration we saved the training and performance results of model and also saved the best model so far in directory for further usage.

## Results and Analysis

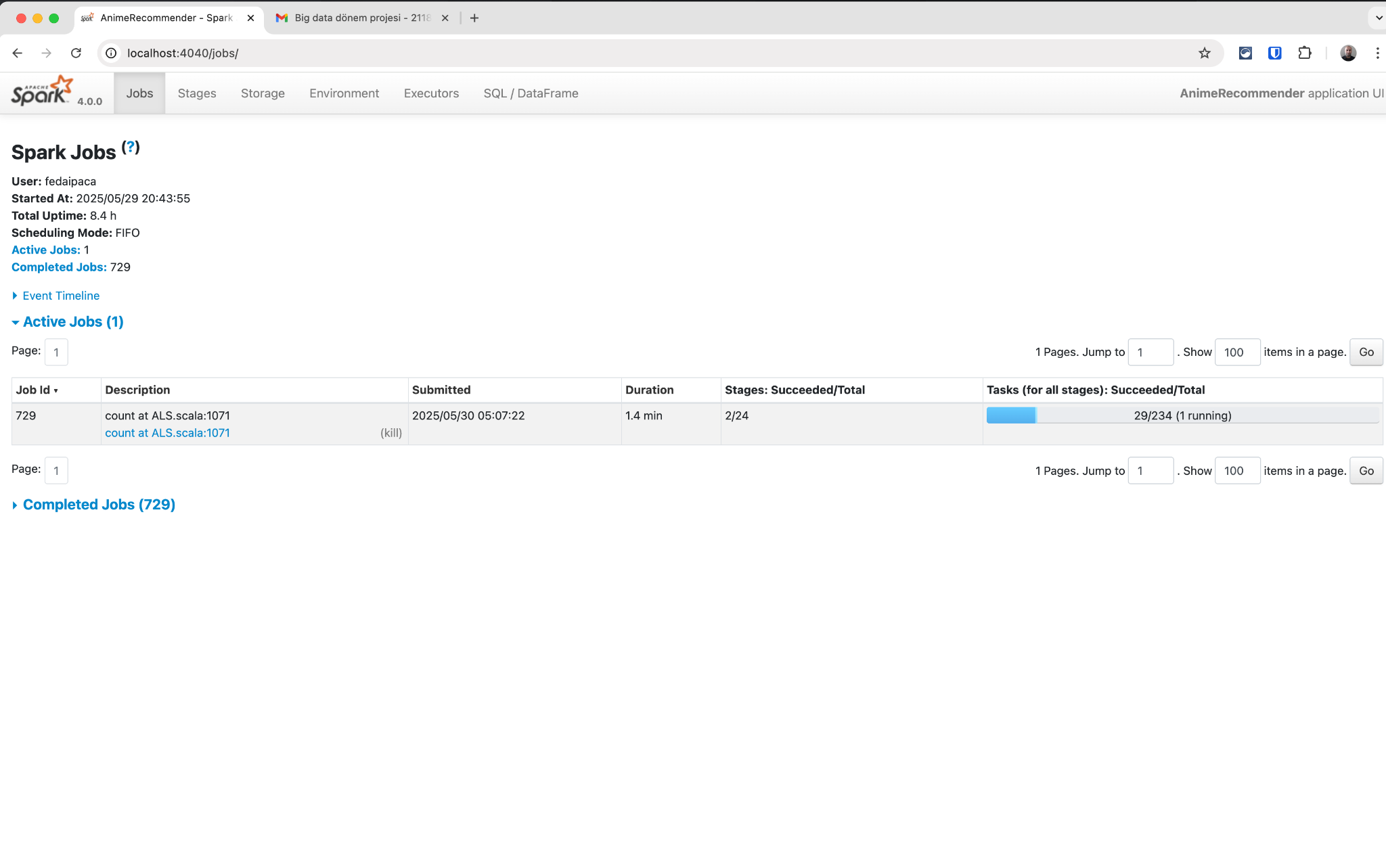
Metrics of models results



Total Training Time: ~11.2 hours (40,391 seconds)

The longest single model (rank=200, iterations=200, lambda=0.01) took 4.9 hours alone!

Spark did 730 jobs in total.



### 

### 

#### Best Model

* Best RMSE: 1.0153
* Best MSE: 1.0309
* Parameters: rank=200, iterations=200, lambda=0.10

This means, on average, the model's predictions are off by about 1.02 rating points, which is quite reasonable for a 1-10 rating scale.

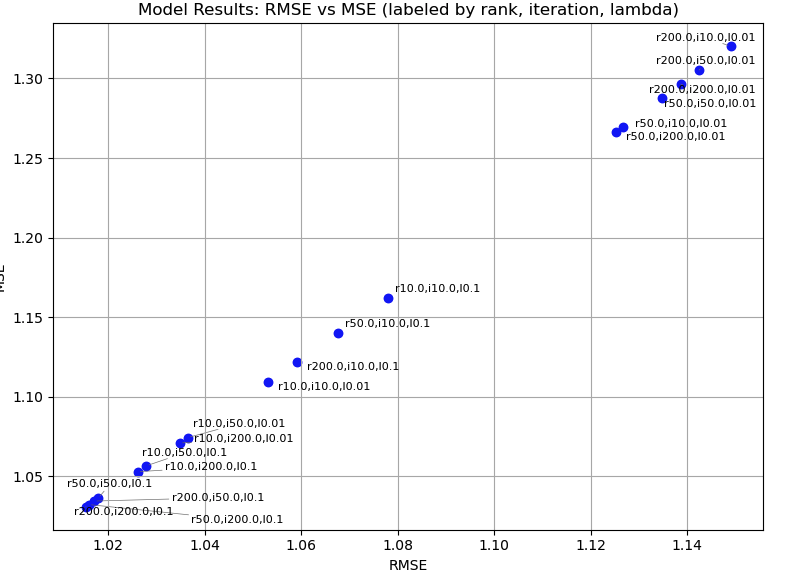
**Why is this the best model?**

* The best model has the lowest RMSE and MSE among all tested configurations. Increasing both the rank and iterations allows the model to better capture the latent structure in the data, while a higher lambda (0.10) helps prevent overfitting.

**RMSE Trends**

* - Increasing iterations generally improves RMSE, especially for higher rank values.
* Higher rank (more latent factors) also tends to improve RMSE, but with diminishing returns and increased computation time.
* Lambda: A value of 0.10 consistently outperforms 0.01, indicating that stronger regularization is beneficial for this dataset.

### Plotting the models with different parameters

**Analysis of Anime Recommendation System Model Performance**

Based on the plot (inferred from the description of the analysis and results):

### Model Metrics

* **RMSE (Root Mean Squared Error)**: Indicates the standard deviation of the prediction errors. Lower RMSE values indicate better model performance.
* **MSE (Mean Squared Error)**: Represents the average of the squares of the errors. Similar to RMSE, lower MSE values indicate better model performance.

### Best Model Identification

* The best model is identified by the lowest RMSE and MSE. In this case, the best model has an RMSE of 1.0153 and an MSE of 1.0309.
* The parameters for this best model are:
  + Rank: 200
  + Iterations: 200
  + Lambda (Regularization): 0.10

### Observations from Parameter Tuning

* **Rank:** Higher rank values (e.g., 200) generally lead to better performance (lower RMSE and MSE). This suggests that a more complex model with more latent factors can capture the underlying patterns in the data more effectively.
* **Iterations:** Increasing the number of iterations also tends to improve model performance. This indicates that the model benefits from more training time to converge to a better solution. However, there might be diminishing returns after a certain point, where increasing iterations does not significantly improve performance.
* **Lambda (Regularization):** A lambda value of 0.10 appears to outperform 0.01. This suggests that stronger regularization is beneficial in preventing overfitting in this dataset. Overfitting occurs when the model learns the training data too well, but fails to generalize to unseen data.

### General Trends

* Increasing both iterations and rank generally improves RMSE, especially when combined.
* Higher lambda values reduce overfitting and contribute to better performance.
* The model performance metrics indicate a reasonable prediction error (RMSE of around 1.02 on a 1-10 scale), suggesting that the recommendation system can provide useful recommendations.

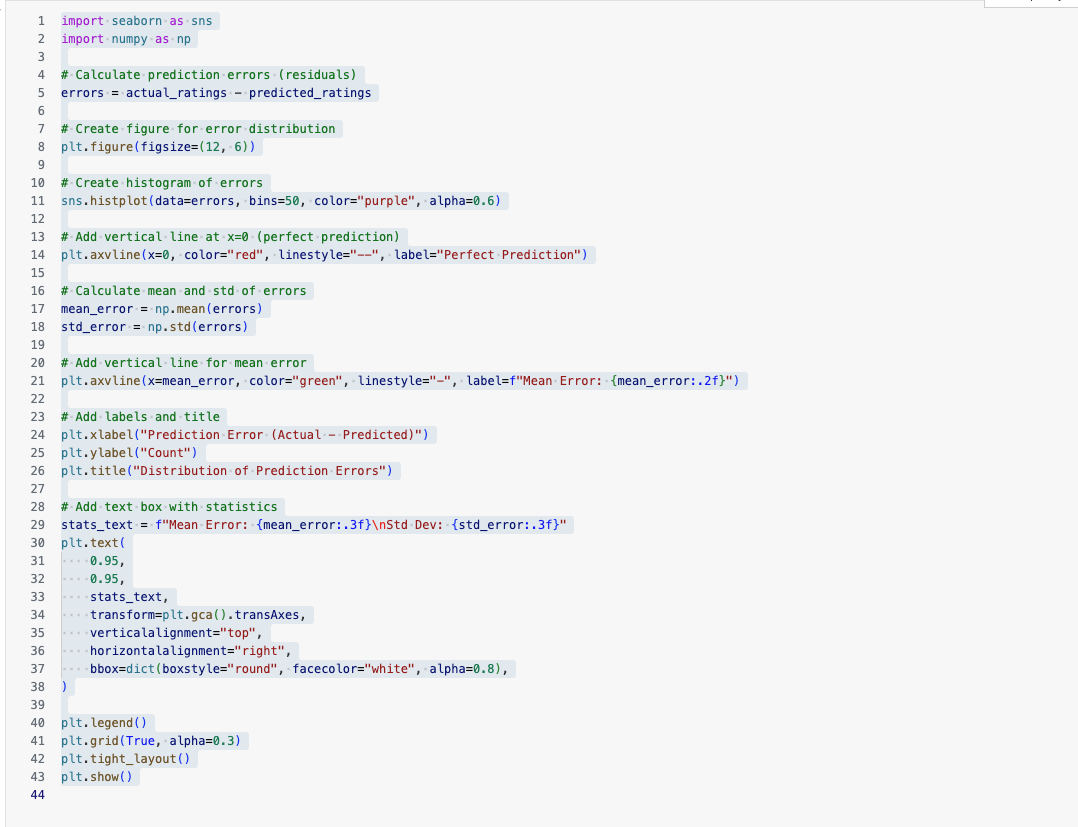
### Key Takeaways

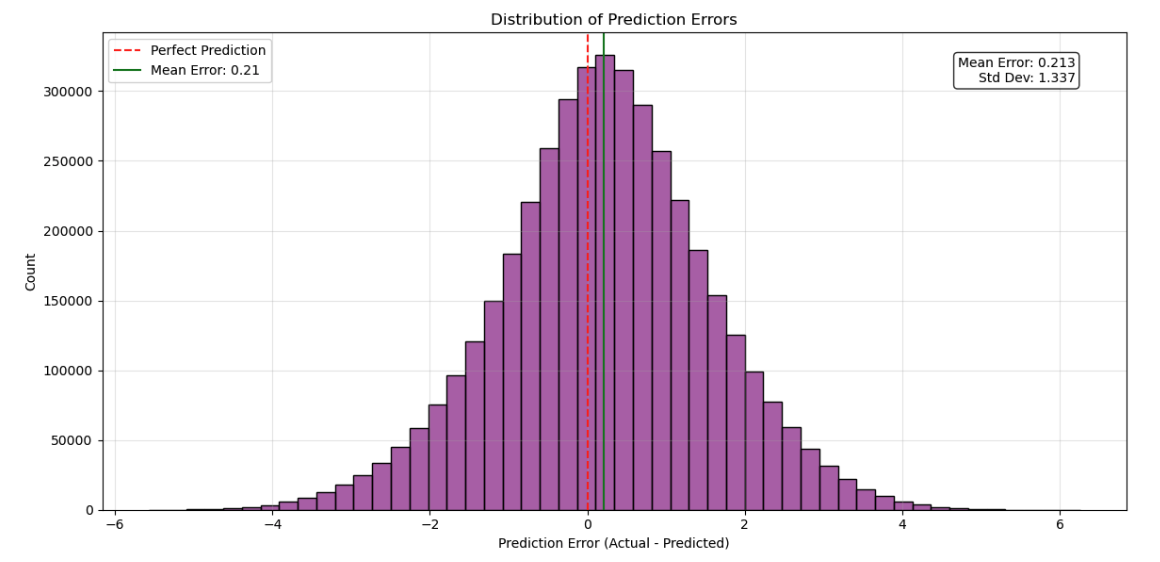
* The hyperparameter tuning process was successful in finding an optimal combination of rank, iterations, and lambda values for the ALS model.
* Model complexity (rank) and training duration (iterations) have a positive impact on performance up to a point.
* Regularization (lambda) is crucial for preventing overfitting and improving generalization.
* The best model achieved an acceptable level of prediction error.

#### Comparing Predictions and Actual values

The recommendation system effectively predicts anime ratings, as shown by the close alignment between predicted and actual values.

### Distribution of Prediction Errors





**What distribution chart shows:**

1. X-axis: Prediction Error (Actual - Predicted)
   * This represents the difference between the actual ratings users gave to animes and what our best model predicted
   * Negative values mean the model over-predicted (predicted higher than actual)
   * Positive values mean the model under-predicted (predicted lower than actual)
   * Zero would be a perfect prediction
2. Y-axis: Count
   * Shows how many predictions fall into each error range
   * The height of each bar indicates the frequency of that particular error magnitude

**Key observations from the chart**

1. Distribution Shape: The histogram shows a normal (bell-curve) distribution centered very close to zero, which is excellent news for our model's performance.
2. Mean Error: 0.213 (shown in green line)
   * This means our model has a very slight tendency to under-predict ratings by about 0.2 points on average
   * This is remarkably close to zero, indicating minimal systematic bias
3. Standard Deviation: 1.337
   * Shows the spread of errors around the mean
   * Most predictions fall within about ±2.67 rating points (2 standard deviations)
4. Concentration Around Zero: The majority of predictions cluster around the red dashed line (perfect prediction), indicating our model is quite accurate for most cases.

In other words:

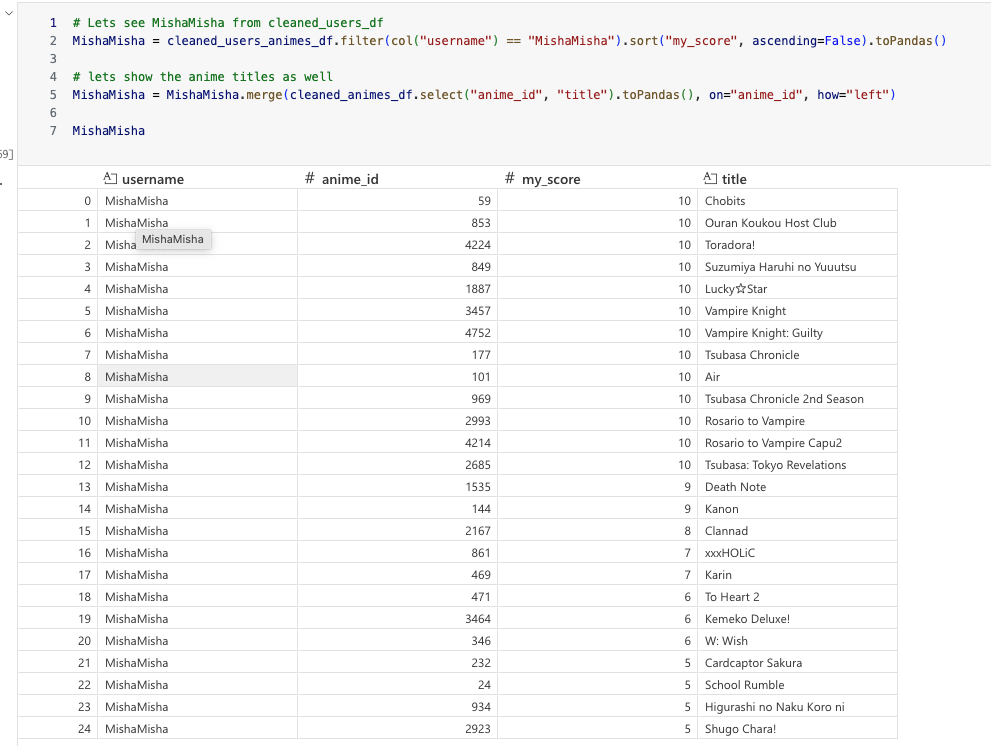
The tight distribution around zero shows our ALS model with parameters (rank=200, iterations=200, lambda=0.10) is performing very well.

The small mean error (0.213) indicates no significant systematic bias.

Most errors fall within a reasonable range for a 5-10 rating scale.

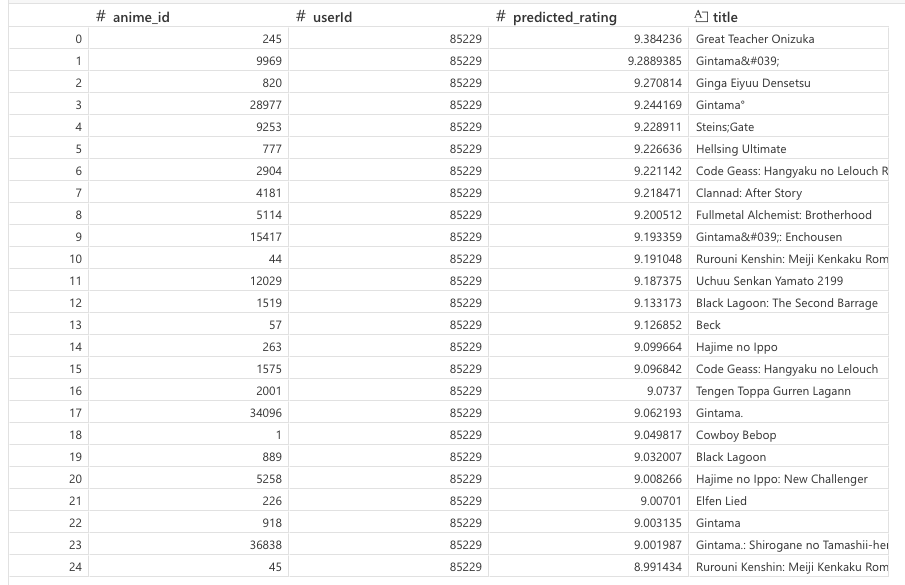
#### Top N Recommendations for a Specific User

Lets observe the favorite animes of username == MishaMisha.



We made the predictions like below:  


Result is:

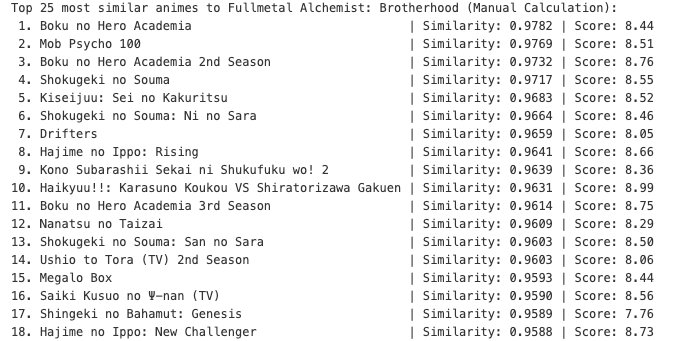


Based on recommended anime genres and popularities, it is clear that the system is recommending similar anime that MishaMisha loved before.

### Finding similar animes to One Punch Man using Cosine Similarity



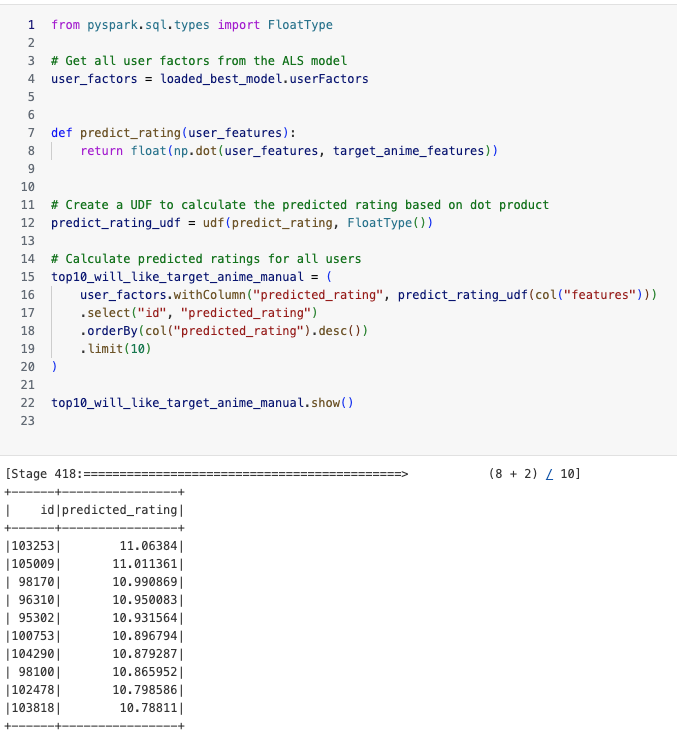
Result is



The recommended animes are indeed similar to One Punch Man.

### Top 10 users who will like One Punch Man

##### Using userFactors



##### Using with built in recommendation system



And it is clear that the users are the same.

## Conclusions

### Limitations and Future Work

**Performance Issues with Local Execution**

Our current setup has several performance bottlenecks:

**1. Single Machine Limitations**

* Running on local[\*] uses only your local CPU cores
* Limited to your machine's RAM (you allocated 6GB to driver)
* No distributed computation benefits
* We can use cluster in real world scenario to distribute computing

**2. Inefficient Hyperparameter Search**

* Sequential training of 18 models (should be parallel)
* No early stopping mechanisms (we can stop for certain threshold to continue with it in production)

**3. Cloud Based solutions**

Cloud based solutions which are specifically created for best performance in big data analysis can reduce the time and improve the overall process and experience.

Some to consider:

AWS EMR: Managed Spark clusters that auto-scale

Databricks: Optimized Spark with MLflow for experiment tracking

Google Dataproc: Google's managed Spark service

Azure HDInsight: Microsoft's big data platform

**REFERENCES**

<https://spark.apache.org/docs/4.0.0/>

<https://spark.apache.org/docs/latest/api/python/index.html>

https://www.kaggle.com/datasets/azathoth42/myanimelist