Movie Recommendation with MLlib - Collaborative Filtering (Implementation 2)

Shruti Kavishwar
San Francisco Bay University
Guided By: Prof Henry Chang

Agenda

- Introduction
- Data Loading and Parsing
- Building the Model
- Evaluating the model
- Saving and Loading the Model
- Conclusion

Introduction

- **Overview**
  - This project focuses on developing a movie recommendation system using collaborative filtering techniques.
- **Goal**
  - The main objective is to build a recommendation system that suggests movies to users based on their past ratings and preferences.
- **Tools**:
  - **Apache Spark:** A unified analytics engine for large scale data processing
  - **Pyspark**: Python API for Spark, enabling Python developers to interface with Spark.
  - **MLlib**: Spark machine learning library that provides various algorithms, including collaborative filtering.

Data Loading and Parsing

- **Each Row consists of a user, a product and a rating**
- **Loading Data:**
  - The `**sc.textFile("<filepath>")**` ine loads the data from the specified path
- **Parsing data:**
  - First `**map**` function splits each line of dataset by commas to extract userID, productID and rating
  - Second `**map**` function converts these extracted values into a Rating object which is required by the ALS algorithm in MLlib.

```
data = sc.textFile("ml-100k/u.data")
#data = sc.textFile("https://grouplens.org/datasets/movielens/100k/u.data")
ratings = data.map(lambda l: l.split('\t')).map(lambda l: Rating(int(l[0]),int(l[1]), float(l[2])))
```

Building the Model

- **The recommendation model is build using the Alternating Least Squares (ALS) algorithm, which is a popular collaborative filtering technique in Machine learning**
- **ALS works by approximating the user-item interaction matrix as the product of two lower-rank matrices, thus predicting missing entries (i.e rating)**
- **Key Parameters:**
  - **Rank:** Represents the number of latent factors in the model. Higher rank can capture more nuance in the data but can also increase the complexity. Here we have set it to 10.
  - **Number of iterations:** The no. of iterations to run ALS. More iterations can lead to better convergence but at the cost of increased computational time. Here we have set it to 10

Building the Model

- `ALS.train()` method is used to train the collaborative filtering model.
- It takes `ratings`, RDD, `rank` and `numIterations` as input and outputs a trained ALS model.
- This trained model can then be used to predict user ratings for movies they have not yet rated.

```
rank = 10
numIterations = 10
model = ALS.train(rank=rank, iterations=numIterations, ratings=ratings)
```

Evaluating the Model

- **Once the model is trained it is essential to evaluate its performance to ensure it provides accurate recommendations.**
- **The evaluation is done by comparing the predicted ratings with the actual ratings using Mean Squared Error (MSE) metric**
- **Steps:**
  - Prepare test data by extracting user and product pairs from the ratings data.
  - Use the trained model to predict ratings for these pairs
  - Join the actual ratings with the predicted ratings
  - Compute the Mean Squared Error (MSE) to evaluate the models accuracy.

Evaluating the Model

- **Test Data Preparation**
  - The '**testdata**' RDD is created by extracting user and product pairs from the 'rating' data.
- **Predictions**
  - The `**model.predictAll**` method is used to predict ratings for these user-product pairs.
  - The predictions are transformed to a format where each entry is a tuple of form `**((user, product), predicted_ratings)**`
- **Joining Ratings:**
  - The `**ratings**' RDD is joined with the `**predictions**` RDD to create an RDD where each entry is a tuple of form `**((user, product), (actual_rating, predicted_rating))**`

Evaluating the Model

- **Calculating MSE:**
  - The Mean Squared Error (MSE) is calculated by computing the average of the squared differences between actual ratings and predicted ratings.
  - A lower MSE indicates better model performance.

```
testdata = ratings.map(lambda p: (p[0], p[1]))
predictions = model.predictAll(testdata).map(lambda r: ((r[0], r[1]), r[2]))
ratesAndPreds = ratings.map(lambda r: ((r[0], r[1]), r[2])).join(predictions)
MSE = ratesAndPreds.map(lambda r: ((r[1][0] - r[1][1])**2)).mean()
print("Mean Squared Error = " + str(MSE))
```

Saving and Loading the Model

- **After training and evaluating the recommendation model, it is important to save the model for future use.**
- **Saving the model allows us to reuse it without retraining , saving time and computational resources.**
- **The saved model can be loaded anytime for making predictions or further evaluations.**
- **Steps**
  - Save the trained model to a specified directory
  - Load the saved model from the directory when needed.

Saving and Loading the Model

- **Saving the model**
  - The `**model.save**` method is used to save the trained model to a specified path **("target/tmp/myCollaborativeFilter")**
- **Loading the model**
  - The `**MatrixFactorizationModel.load**` method is used to load the saved model from the specified path.
  - Once loaded, `**sameModel**` can be used to make predictions just like the original model

```
model.save(sc, "target/tmp/myCollaborativeFilter")
sameModel = MatrixFactorizationModel.load(sc, "target/tmp/myCollaborativeFilter")
sc.stop()
```

Output of the Model

- The Mean Squared Error is **0.4844**
- This is relatively low MSE indicating that the model's predictions are quite close to the actual ratings, suggesting good model performance.

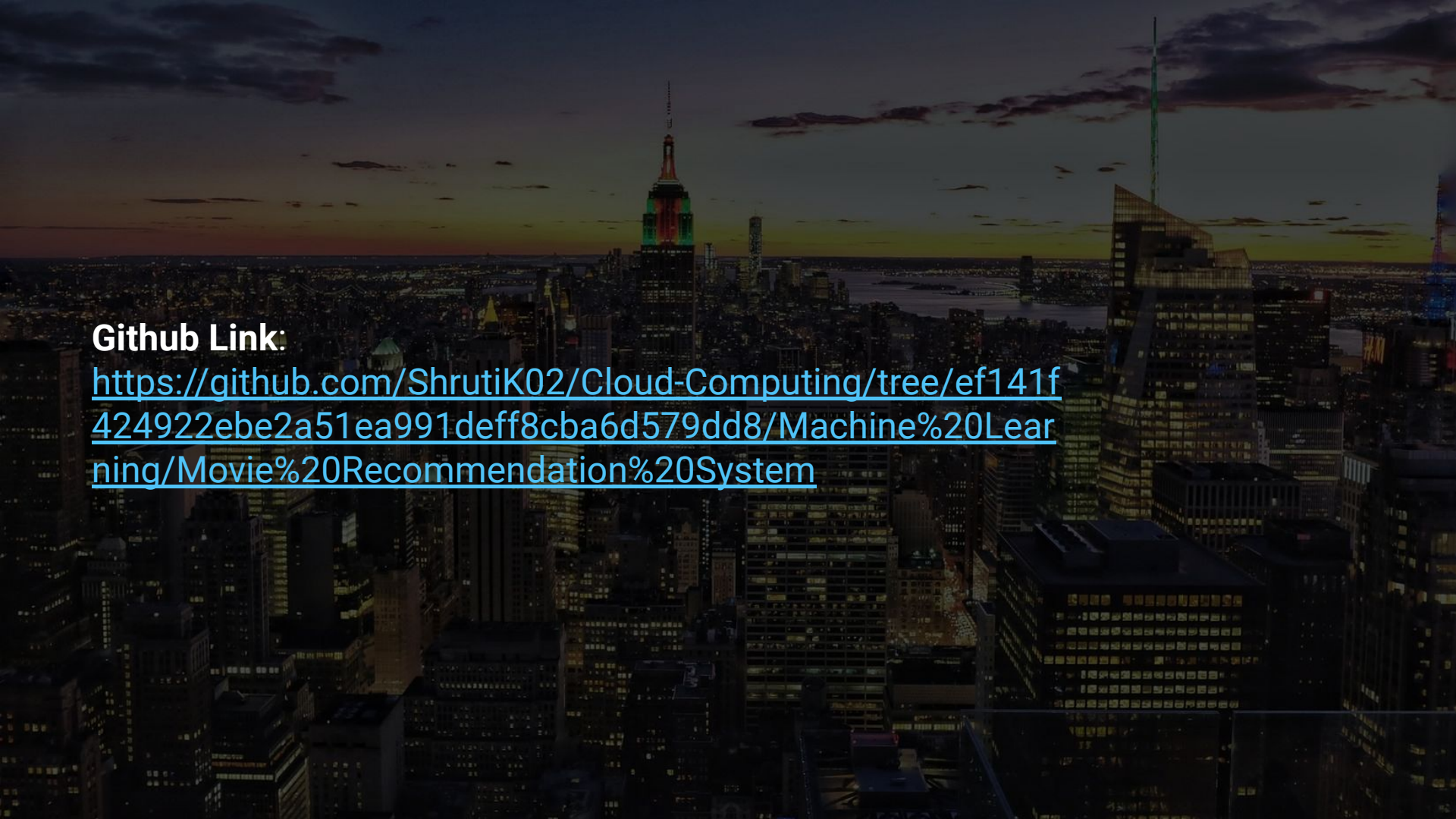
```
ml-100k.zip         100%[===================>]   4.70M  3.45MB/s    in 1.4s

2024-07-17 02:54:06 (3.45 MB/s) - 'ml-100k.zip' saved [4924029/4924029]

Archive:  ml-100k.zip
   creating: ml-100k/
  inflating: ml-100k/allbut.pl
  inflating: ml-100k/mku.sh
  inflating: ml-100k/README
  inflating: ml-100k/u.data
  inflating: ml-100k/u.genre
  inflating: ml-100k/u.info
  inflating: ml-100k/u.item
  inflating: ml-100k/u.occupation
  inflating: ml-100k/u.user
  inflating: ml-100k/u1.base
  inflating: ml-100k/u1.test
  inflating: ml-100k/u2.base
  inflating: ml-100k/u2.test
  inflating: ml-100k/u3.base
  inflating: ml-100k/u3.test
  inflating: ml-100k/u4.base
  inflating: ml-100k/u4.test
  inflating: ml-100k/u5.base
  inflating: ml-100k/u5.test
  inflating: ml-100k/ua.base
  inflating: ml-100k/ua.test
  inflating: ml-100k/ub.base
  inflating: ml-100k/ub.test
Mean Squared Error = 0.4844291448471857
```

Future Work

- **Improve Model Accuracy**
  - Experiment with different ranks and iterations to fine-tune the model
  - Consider additional data processing and feature engineering techniques
- **Exploring Different Algorithms**
  - Investigate other recommendation algorithms like content-based filtering or hybrid models to improve recommendations.
- **Scalability and Real time Predictions:**
  - Implement the recommendation system in a real-time environment to provide instant movie suggestion.



Github Link:
https://github.com/ShrutiK02/Cloud-Computing/tree/ef141f424922ebe2a51ea991deff8cba6d579dd8/Machine%20Learning/Movie%20Recommendation%20System