BDTT Lecture 8 Handout

Recommender Systems: An Introduction

1. Introduction to Recommender Systems

Recommender systems are a fundamental part of many modern digital services. They provide personalised suggestions to users based on their preferences, past interactions, or the behaviour of similar users. For example:

- Streaming services like Netflix and Spotify recommend shows or music to the user
- E-commerce platforms like Amazon recommend products relating to what you have previously bought or viewed

Recommender systems enhance user experience and drive engagement and revenue by helping users discover content or products which are relevant to them.

Why Are Recommender Systems 'Big Data' Problems?

Recommender systems deal with vast amounts of data, making them an essential application within the domain of Big Data. Several factors contribute to their Big Data nature:

- **High Volume:** Millions of users and products generate massive datasets.
- **High Velocity:** New data is constantly generated as users interact with systems.
- **High Variety:** Data comes from multiple sources (explicit ratings, purchase history, browsing behaviour, implicit interactions).
- **Sparsity:** User-item interaction matrices are typically sparse, as users engage with only a small subset of available items.
- **Scalability Needs:** Efficient algorithms and distributed computing frameworks, such as Apache Spark, are required to process large-scale recommender system datasets.

Let's take an example. Netflix has:

- 300 million subscribers
- Approx. 17,000 titles

If an average user has watched 20 titles, as an example, then we're talking about 6 billion user interactions!

2. Types of Recommender Systems

Recommender systems primarily fall into two categories:

Content-Based Filtering

Content-based filtering makes recommendations by analysing the characteristics of items a user has previously interacted with and recommending similar items.

Example: Netflix

- If a user watches *Harry Potter* and *The Chronicles of Narnia*, Netflix may recommend *The Lord of the Rings* based on genre similarity.
- The system builds a profile for each user based on their past preferences and uses similarity measures to find related items.

Advantages:

- Personalised recommendations without needing other users' data.
- Useful for new users or businesses without large datasets.

Disadvantages:

- Limited exploration beyond a user's existing interests (i.e., it may recommend too similar content, leading to a filter bubble).
- Requires accurate and rich metadata to work effectively (for example, film genre classifications, so it recommends films of the same genre, etc)

Collaborative Filtering

Collaborative filtering recommends items based on user behaviour and interactions. It assumes that users with similar past behaviours will have similar future preferences.

Example: Amazon

- If two users have purchased similar books, Amazon might recommend books one user has bought but the other hasn't yet.

Collaborative filtering typically works by considering users and items simultaneously.

Advantages:

- Does not require metadata about items.
- Helps users discover new items they wouldn't have found through content-based filtering

alone (for example, this may include complementary products to the one the user has bought, rather than just products which are similar).

Disadvantages:

- The "cold start" problem: New users and items without enough data cannot receive good recommendations.
- Computationally expensive, requiring efficient algorithms to handle large-scale data.

To train a collaborative filtering recommender system we need some form of user rating. We can distinguish between explicit ratings and implicit ratings.

Explicit Ratings are where users **directly provide** feedback on items.

Examples:

- Star ratings (e.g., giving a movie **4 out of 5 stars** on Netflix).
- Thumbs up/down (e.g., YouTube's like/dislike button).
- Written reviews (e.g., product reviews on Amazon).

Pros: More accurate and reliable since users express their true preferences.

Cons: Often sparse, as many users do not provide explicit feedback.

2. Implicit Ratings are inferred from **user behaviour** rather than direct feedback.

Examples:

- Time spent watching a video (e.g., if a user watches 90% of a movie, it's likely they enjoyed it).
- Clicks on products (e.g., frequently clicking on a product page suggests interest).
- Purchase history (e.g., buying an item may indicate preference).

Pros: More abundant since user behaviour is always being tracked.

Cons: Less reliable—just because someone clicked on something doesn't mean they liked it.

Which one is used in recommender systems in the real-world?

- **Explicit ratings** are preferred when available, but they are often **sparse**.
- **Implicit ratings** are widely used in practice because they provide **rich behavioural data** even if they are noisier.
- Many modern recommender systems **combine both explicit and implicit feedback** for better recommendations.

3. Collaborative Filtering

Introduction to Latent Features

When building recommender systems, especially those using **collaborative filtering**, we often deal with a large **user-item interaction matrix**. However, this matrix is usually **very sparse**, meaning most users have interacted with only a small fraction of available items. This is where **latent features** become useful.

What are Latent Features? Latent features are **hidden patterns** in the data that we do not explicitly define but which help explain user preferences. They capture characteristics that influence user behaviour and item properties without us needing to manually specify them.

Example of Latent Features in Movies

Imagine you have a dataset of movies and user ratings. Instead of manually defining categories like "genre" or "lead actor," a recommender system might learn **latent features** such as:

- **Fantasy vs. Non-Fantasy:** Capturing whether a user prefers fantasy films.
- **High vs. Low Action Content:** Identifying users who like action-packed movies.
- **Artistic vs. Commercial Appeal:** Distinguishing users who prefer niche indie films vs. blockbuster hits.

Each movie and user can then be represented as a combination of these latent features. However, the key point is that we don't decide these latent features ourselves, instead we use machine learning to learn these relationships **automatically** by analysing patterns in user ratings.

Matrix Multiplication and Factorisation: The Foundation of Recommender Systems

To understand how latent features are used in recommendations, we first need to explore **matrix multiplication, factorisation, and matrix factorisation.**

What is Matrix Multiplication?

A **matrix** is simply a grid of numbers. **Matrix multiplication** is a mathematical operation where we multiply two matrices together to produce a new matrix.

Example:

Consider a multiplication of two matrices:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} (1 \times 7) + (2 \times 9) + (3 \times 11) & (1 \times 8) + (2 \times 10) + (3 \times 12) \\ (4 \times 7) + (5 \times 9) + (6 \times 11) & (4 \times 8) + (5 \times 10) + (6 \times 12) \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

Each element in the resulting matrix is calculated by taking the **dot product** of the corresponding **row** from the first matrix and the **column** from the second matrix. So for the first element on the top left, we multiply the elements in the first row of the first matrix with the elements in the first column of the second matrix. If we have a ***N*** x ***K*** matrix (where M is the number of rows and K is the number of columns) and we multiply it by a ***K*** x ***M*** matrix this will give us an ***N*** x ***M***. So in the example above, we multiply a 2 x 3 matrix with a 3 x 2 matrix and get a 2 x 2 matrix. Matrix multiplication is only possible where the number of columns in the first matrix is equal to the number of rows in the second matrix.

What is Matrix Factorisation?

Factorisation is general is the process of finding **factors** – for example, factorising 15 gives us 5 and 3 because 5 x 3 = 15.

Matrix factorisation is the **reverse** of matrix multiplication. Instead of multiplying two matrices together, we **break down a large matrix into two smaller matrices** that, when multiplied, approximate the original matrix.

Example:

Imagine we have a large **user-item interaction matrix** where each row represents a user, each column represents an item (such as a movie), and each cell contains a rating (or is empty if no rating exists). Instead of working with this massive, sparse matrix, we want to approximate it using two smaller matrices:

- One matrix represents **users and their preferences across latent features**.
- The other matrix represents **items and their characteristics across the same latent features**.

Formally, if we have an **M x N** matrix (users x items), we factorise it into:

- A **user matrix** U (size **M x K**) where each user is represented in terms of **K** latent features.
- An **item matrix** V (size **K x N**) where each item is represented in terms of **K** latent features.

Thus, our original matrix is **approximated** as:

$$R \approx U \times V^T$$

Why Do We Factorise the Matrix?

- It helps to **fill in missing values** (i.e., predict ratings for items a user hasn't interacted with

yet).

- It **compresses the data**, making it easier to work with.
- It automatically learns **hidden patterns** (latent features) without requiring manual feature engineering.

How Does Matrix Factorisation Relate to Recommendations?

When we apply **Alternating Least Squares (ALS)** or another factorisation algorithm, we are essentially:

1. Breaking the user-item interaction matrix into two lower-dimensional matrices.
2. Learning latent features that represent user preferences and item characteristics.
3. Using these learned representations to **predict missing values**, thereby generating recommendations.

For example, if a user has watched and rated several action movies highly, their **latent feature vector** will reflect a preference for action films. The system will then **recommend other movies** that have high scores in the same latent features.

Summary:

- **Latent features** capture hidden patterns in user preferences and item characteristics.
- **Matrix multiplication** allows us to combine these latent features to estimate ratings and generate recommendations.
- **Matrix factorisation** helps break down the large user-item matrix into more manageable components that reveal underlying structures in the data.
- **ALS in Spark** efficiently performs this factorisation, making large-scale recommender systems practical.

The Alternating Least Squares (ALS) Algorithm

The **ALS algorithm** is a popular method for matrix Factorisation, particularly in Apache Spark's MLlib library.

How ALS Works:

1. Assigns initial values to user and item matrices.
2. Alternates between fixing one matrix while optimising the other to minimise prediction error.

3. Repeats until the algorithm converges on a stable set of matrices that best approximate the user-item matrix.

Evaluation:

- The accuracy of a recommender system can be assessed using **Root Mean Square Error (RMSE)**, which measures how close predicted ratings are to actual user ratings.
- Regularisation techniques are applied to prevent overfitting.

4. Challenges and Considerations in Recommender Systems

Despite their advantages, recommender systems face several challenges:

- **Cold Start Problem:** New users and items lack sufficient interaction data.
- **Scalability:** Large-scale systems require efficient distributed computing solutions.
- **Bias and Fairness:** Recommendations can reinforce biases if not carefully designed.
- **Privacy Concerns:** User data must be handled securely and ethically.

5. Putting it all together

Modern recommender systems require a robust data pipeline that integrates multiple Big Data technologies. Apache Spark provides the necessary tools for handling large-scale data processing, real-time updates, and storage.

Pipeline Overview

An example pipeline may consist of the following stages:

1. **Data Collection**: Capture user interactions (clickstream data).
2. **Feature Engineering**: Extract relevant features (e.g., time spent on content) as implicit ratings.
3. **Batch Processing**: Use Spark SQL to aggregate data and train an ALS model.
4. **Batch Predictions**: Store recommendations in a NoSQL database for fast retrieval.
5. **Streaming Processing**: Use Spark Streaming to process new user interactions which will then be used for periodic re-training

Step 1: Data Collection of Clickstream and Interaction Data

A key source of user engagement data in many real-world applications is **clickstream data**, which logs user actions on a platform. This includes:

- Page views
- Video start/stop events
- Purchase interactions
- Clicks on recommended items

Example: Inferring Implicit Ratings from Interaction Data

Instead of explicit ratings (e.g., a 5-star rating system), we can use **implicit feedback** derived from user interactions. A practical way to estimate implicit ratings:

- **Start Time and End Time of a Video/Article:**
    - If a user watches 90% of a movie, it likely indicates strong interest (high implicit rating).
    - If a user watches only 10%, they might not be interested (low implicit rating).
- **Number of Clicks on a Product Page:**
    - More interactions suggest stronger engagement.

Step 2: Feature Engineering Using Spark SQL

Once data is collected, we can use **Spark SQL** to clean and transform the data.

This transformed dataset is then used to create a user-item interaction matrix for training the ALS model.

Step 3: Model Training with ALS (Batch Processing in Spark MLlib)

After feature extraction, we train an **ALS model** using Spark's MLlib.

Step 4: Batch Predictions and Storage in NoSQL Database (we will cover NoSQL next week)

Once the ALS model is trained, we generate predictions for all users and store them in a NoSQL database (e.g., MongoDB, DynamoDB, or Cassandra) for efficient retrieval.

Step 5: Process On-going User Interaction Data using Spark Streaming

To ensure that recommendations stay fresh, we incorporate **Spark Streaming** to handle new interactions in real time.

Streaming Pipeline

1. **Consume clickstream data** from Kafka (or another real-time source).
2. **Process new events** (e.g., user watching a movie, clicking a product).

3. **Update ALS model periodically** to reflect new user preferences.
4. **Generate new recommendations for active users.**

This pipeline provides a **realistic** architecture for a recommender system using Apache Spark:

- **Clickstream data** provides implicit feedback for recommendations.
- **Spark SQL** processes raw user interactions.
- **Spark MLlib's ALS** generates recommendations using latent factor models.
- **NoSQL storage** enables fast retrieval of recommendations.
- **Spark Streaming** allows continuous updates of user interactions.