Data-Mining-Series-2-Important-Topics

1. Web usage mining

Web Usage Mining is the process of extracting useful information from logs of web access, like user click streams (the sequence of clicks a user makes on a website).

Where does the data come from?

- Web server logs
- Application server logs

What does it do?

- Tracks users' browsing history (which pages they visit and in what order).
- Finds patterns in user behavior (like frequently visited pages, search trends, and associations).
- Helps predict what users might be searching for on the Internet.

Example:
Imagine a shopping website tracking what products users view. If many users check **mobile phones** → **reviews** → **price comparison**, the site can recommend reviews whenever someone views a mobile phone.

Why is it useful?

- Helps businesses understand customer preferences.
- Improves website design and user experience.
- Can be used for targeted advertising.

2. Web structure mining

Web Structure Mining is the process of analyzing the structure of web pages and their connections to extract useful information.

Types of Structure Mining:

1. **Intrapage Structure Mining:**
    - Looks inside a single web page.
    - Analyzes **internal links** (links within the page) and **HTML/XML code**.
    - Example: Checking how sections of a Wikipedia page are linked together.
2. **Interpage Structure Mining:**
    - Studies how different web pages are connected through hyperlinks.
    - Helps in understanding the relationship between pages on the web.
    - Example: Google's PageRank algorithm ranks websites based on link connections between them.

Why is it useful?

- Helps in **search engine optimization (SEO)** (Google ranks pages based on links).
- Improves website navigation by organizing links efficiently.
- Assists in detecting **spam websites** (unnatural link networks).

3. Web content mining

Web Content Mining is the process of analyzing the actual content of web pages and search results to extract useful information.

Types of Web Content Mining:

1. **Web Page Content Mining:**
    - Traditional method of searching for information on web pages.
    - Extracts content such as **text, images, videos, audio, and structured records** (like product details on e-commerce sites).
    - Example: Google scanning web pages to show relevant search results.

2. **Search Result Mining:**
   - Analyzing the web pages that appear in search results.
   - Helps refine searches by finding patterns in previously retrieved results.
   - Example: Google's "People also ask" section suggests related topics based on user searches.

4. TF-IDF

Term Frequency (TF)

TF(Term frequency) measures how often a word (term) appears in a document.

Types of Term Frequency:

1. **Binary TF**:
   - If a word appears in a document, TF = 1; otherwise, TF = 0.
   - Example: If the word "data" appears in a document, TF(data) = 1.
2. **Raw Count TF**:
   - Counts how many times a word appears.
   - Example: If "data" appears 3 times in a document, TF(data) = 3.
3. **Relative Term Frequency:**
   - Adjusts for document length by dividing the term frequency by the total number of words in the document.

Inverse Document Frequency (IDF)

IDF measures how **important** a word is by checking how common it is across multiple documents.

- If a word appears in **many** documents, it is **less important** (like "the" or "is").
- If a word appears in **fewer** documents, it is **more important** (like "blockchain" or "machine learning").

5. Text retrieval methods

Text retrieval methods help find relevant documents based on user queries. They are mainly divided into two types:

1. Document Selection Methods

- These methods **select documents** that exactly match the query conditions.
- Uses **Boolean retrieval model**, where documents are represented by **keywords**.
- Users provide a **Boolean expression** (AND, OR, NOT) to filter documents.

Examples of Boolean Queries:

✅ **"car AND repair shops"** → Retrieves documents containing both "car" and "repair shops".
✅ **"tea OR coffee"** → Retrieves documents containing either "tea" or "coffee".
✅ **"database systems BUT NOT Oracle"** → Retrieves documents about database systems but **excludes** those mentioning "Oracle".

🔹 **Limitations:**

- Only returns documents that **fully satisfy** the query.
- Doesn't rank results by relevance.
- Works well for **precise searches** but **not great for exploratory searches**.

2. Document Ranking Methods

- Instead of strict filtering, these methods **rank documents based on relevance**.
- **Matches keywords** in the query with those in the documents.
- **Assigns a score** to each document based on how well it matches the query.
- **Presents results as a ranked list** (e.g., Google Search).

How Ranking Works?

- **Frequency-based ranking** → More keyword matches = higher score.
- **Statistical methods** (e.g., TF-IDF, probability models) assign weights to words.
- **Machine learning & AI-based ranking** (used in modern search engines).

🔹 **Why is it useful?**

- Helps in **search engines, recommendation systems, and large databases**.
- More user-friendly for **ordinary users** compared to strict Boolean searches.

6. Text Mining-Text Data Analysis and-information Retrieval

Text Mining

Text mining is an interdisciplinary field that applies techniques from data mining, machine learning, statistics, and computational linguistics to extract meaningful information from textual data. It is commonly used in various domains such as digital libraries, web pages, emails, and news articles.

Common tasks in text mining include:

- **Text categorization** – Assigning predefined categories to documents.
- **Text clustering** – Grouping similar documents together.
- **Sentiment analysis** – Identifying opinions or emotions in text.
- **Entity extraction** – Finding names, places, and other structured data.

Text Data Analysis

In text mining, various basic measures are used to analyze and retrieve information from text-based datasets. Two primary methods used in text data analysis include:

1. **Precision** – The percentage of retrieved documents that are actually relevant to the query.
2. **Recall** – The percentage of relevant documents that were retrieved from the dataset.
3. **F-Score** – A metric that balances precision and recall to provide a single evaluation score.

Text Retrieval Methods

Text retrieval methods can be categorized into:

1. **Document Selection Methods:**
   - Uses constraints to filter and retrieve documents based on exact keyword matches.
   - Example: Boolean retrieval model (e.g., "car AND repair shops").
2. **Document Ranking Methods:**
   - Ranks retrieved documents based on their relevance to the query.
   - Scores documents based on keyword frequency and similarity to the query.

TF (Term Frequency) and IDF (Inverse Document Frequency)

The **TF-IDF** measure is commonly used in text retrieval and ranking systems

- **Term Frequency (TF)**: Measures how frequently a term appears in a document.
- **Inverse Document Frequency (IDF)**: Measures the importance of a term. A term that appears in many documents has a lower IDF value.

7. *Apriori algorithm*

The **Apriori Algorithm** is used in **data mining** to find **frequent itemsets** in a **large dataset of transactions**. It is commonly used for **market basket analysis**, where we find items that are often bought together.

Step-by-Step Explanation with Example

Step 1: Understanding the Data

Imagine you own a supermarket, and you have **9 transactions** (customer purchases). Your goal is to find which products are **frequently bought together**.

Transaction ID	Items Bought
1	Milk, Bread, Butter
2	Milk, Bread
3	Bread, Butter
4	Milk, Butter
5	Milk, Bread, Butter
6	Milk
7	Bread
8	Butter
9	Milk, Bread

Step 2: Set a Minimum Support Count

- **Support count** is the number of times an item or itemset appears in transactions.
- Let's say we decide **minimum support count = 2** (meaning an item should appear at least twice to be considered frequent).

Apriori Algorithm Steps

Step 1: Find Frequent 1-Itemsets (L1)

1. Count how many times each item appears in the transactions (from the table above)
2. Keep only those that meet the **minimum support count** (Minimum support count = 2).

Item	Count
Milk	6
Bread	6
Butter	4

Since all have **count ≥ 2**, they are **frequent**.

Step 2: Generate 2-Itemsets (L2)

Now, we pair **frequent** items from **L1**.

Itemset	Count
(Milk, Bread)	4
(Milk, Butter)	3
(Bread, Butter)	3

All have **count ≥ 2**, so they are **frequent**.

Step 3: Generate 3-Itemsets (L3)

Now, we form sets of **3 items** from L2:

Itemset	Count
(Milk, Bread, Butter)	2

Since **count = 2** (which is equal to min support count), it is **frequent**.

Step 4: Stop When No More Frequent Itemsets

- If we try to generate a **4-itemset**, it won’t have enough support.
- The algorithm **stops** here.

Step 5: Generate Association Rules

Now that we have found **frequent itemsets**, we can generate **association rules** to understand how items are related.

1. What are Association Rules?

An **association rule** is of the form:
$A \Rightarrow B$
which means **"If A is bought, then B is likely to be bought too."**

Each rule has two main parts:

- **Antecedent (Left-hand side - LHS):** The item(s) that appear first (e.g., **Milk, Bread**).
- **Consequent (Right-hand side - RHS):** The item(s) that might appear next (e.g., **Butter**).

Example from our dataset:

- Rule: **(Milk, Bread) ⇒ Butter**
  → Meaning: "If a customer buys Milk and Bread, they might also buy Butter."
- Rule: **Milk ⇒ Bread**
  → Meaning: "If a customer buys Milk, they might also buy Bread."

2. How are these rules selected?

Rules are generated from **frequent itemsets** found in previous steps.
We consider **every subset** of a frequent itemset as a possible **antecedent (LHS)**, and the remaining items as the **consequent (RHS)**.

For example, from the **frequent 3-itemset (Milk, Bread, Butter)**:

- We can make rules like:
  - **(Milk, Bread) ⇒ Butter**
  - **(Milk, Butter) ⇒ Bread**
  - **(Bread, Butter) ⇒ Milk**
  - **Milk ⇒ (Bread, Butter)**
  - **Bread ⇒ (Milk, Butter)**
  - **Butter ⇒ (Milk, Bread)**

Each rule is evaluated using **confidence**.

3. Confidence Calculation

Confidence measures how often the rule is correct. It is calculated as:

$$Confidence(A \Rightarrow B) = \frac{Support(A \cup B)}{Support(A)}$$

This means:

$$\text{Confidence} = \frac{\text{Number of transactions containing both A and B}}{\text{Number of transactions containing A}}$$

Example 1: (Milk, Bread) ⇒ Butter

We already know:

- **Support(Milk, Bread, Butter) = 2** (appears in 2 transactions).
- **Support(Milk, Bread) = 4** (appears in 4 transactions).

$$Confidence((Milk, Bread) \Rightarrow Butter) = \frac{2}{4} = 50\%$$

So, if a customer buys **Milk and Bread together**, there is a **50% chance** they will also buy **Butter**.

Example 2: Milk ⇒ Bread

We already know:

- **Support(Milk, Bread) = 4** (appears in 4 transactions).
- **Support(Milk) = 6** (appears in 6 transactions).

$$Confidence(Milk \Rightarrow Bread) = \frac{4}{6} = 66.7\%$$

So, if a customer buys **Milk**, there is a **66.7% chance** they will also buy **Bread**.

Now, we create **rules** like:

- **If a customer buys Milk and Bread, they might also buy Butter.**
  $(Milk, Bread) \Rightarrow Butter$
  Confidence = 50%
- **If a customer buys Milk, they might also buy Bread.**
  $Milk \Rightarrow Bread$
  Confidence = 66.7%.

Summary of Apriori Algorithm

1. **Find frequent 1-itemsets (L1)** by counting occurrences.
2. **Generate larger itemsets (L2, L3, etc.)** by combining frequent itemsets.
3. **Stop when no more frequent itemsets can be formed.**
4. **Generate association rules** from frequent itemsets.

---

8. FP tree growth algm and problem

The **FP-Growth Algorithm** is used for finding **frequent itemsets** in a dataset, similar to the **Apriori Algorithm**, but it is **faster and more efficient** because it doesn't generate candidate sets multiple times. Instead, it builds a **tree structure** called an **FP-Tree** and mines patterns directly from it.

Why Use FP-Growth Instead of Apriori?

- **Apriori scans the database multiple times**, which makes it slow for large datasets.
- **FP-Growth scans the database only twice** and stores the data in a compact tree format, making it much faster.
- **No need to generate large candidate itemsets**, reducing computational effort.

Step 1: Scan the Database & Find Frequent Items

- Just like Apriori, first, we **scan the database** to find the **frequency (support count)** of individual items.
- Only keep items that meet the **minimum support count**.

- **Sort items in descending order of frequency**.

Example Transaction Database

Transaction ID	Items Bought
1	Milk, Bread, Butter
2	Milk, Bread
3	Bread, Butter
4	Milk, Butter
5	Milk, Bread, Butter
6	Milk
7	Bread
8	Butter
9	Milk, Bread

Find the Frequency of Each Item

- **Milk**: 6 times
- **Bread**: 6 times
- **Butter**: 4 times

Now, sort them in descending order:

L = {Milk: 6, Bread: 6, Butter: 4}

Step 2: Build the FP-Tree (Frequent Pattern Tree)

1. **Create a root node** (labeled "null").
2. **Scan the transactions again**, inserting them into the FP-Tree:
    - Each transaction follows the **L order (sorted order of frequent items)**.
    - If an item appears in an existing branch, increase its count. Otherwise, create a new branch.

FP-Tree Construction Example

Let's process the transactions in **L order**.

Transaction	Sorted (L Order)
(Milk, Bread, Butter)	(Milk → Bread → Butter)
(Milk, Bread)	(Milk → Bread)
(Bread, Butter)	(Bread → Butter)
(Milk, Butter)	(Milk → Butter)
(Milk, Bread, Butter)	(Milk → Bread → Butter)
(Milk)	(Milk)
(Bread)	(Bread)
(Butter)	(Butter)
(Milk, Bread)	(Milk → Bread)

From this, we build the **FP-Tree**:

```
null
 ├── Milk (6)
 │    ├── Bread (4)
 │    │    ├── Butter (2)
 │    ├── Butter (2)
 ├── Bread (2)
 │    ├── Butter (2)
 ├── Butter (1)
```

- **Each node stores an item and its frequency count**.
- **Repeated items share a path, and their count increases**.

Step 3: Mine the FP-Tree for Frequent Itemsets

- Start from the **least frequent item** (Butter) and move upward.
- **Construct conditional pattern bases** (paths leading to an item).
- **Generate frequent patterns from these paths**.

Example: Finding Frequent Patterns for "Butter"

- **Find all paths leading to "Butter"**:
  - (Milk, Bread → Butter) appears **2 times**.

    - (Bread → Butter) appears **2 times**.
    - (Milk → Butter) appears **2 times**.
- **Find frequent itemsets**:
    - {Milk, Butter}: 2
    - {Bread, Butter}: 2
    - {Milk, Bread, Butter}: 2
- Repeat the same for **Bread and Milk** to get all frequent itemsets.

Final Output - Frequent Itemsets

After mining all patterns from the FP-Tree, we get:

Frequent Itemset	Support Count
{Milk, Bread}	4
{Milk, Butter}	2
{Bread, Butter}	2
{Milk, Bread, Butter}	2

Summary

1. **Scan the database** and count item frequencies.
2. **Sort items in descending order** and construct the **FP-Tree**.
3. **Mine the tree** using conditional pattern bases.
4. **Generate frequent itemsets** without candidate generation.

9. Dynamic Itemset counting

Problem with Apriori Algorithm

Imagine you own a supermarket and you want to find which products are frequently bought together (like "Milk & Bread").

With the **Apriori Algorithm**, you:

1. **Scan the entire transaction database once**.

2. Generate **new item combinations** (like "Milk & Bread" or "Bread & Butter").
3. **Scan the entire database again** to check how often these combinations appear.
4. Repeat this process **multiple times**, which is **slow**.

It would be better if we could **add new item combinations while scanning** instead of waiting for a full database scan

How Dynamic Itemset Counting (DIC) Works

DIC **solves this problem** by allowing new item combinations to be **added while scanning** instead of waiting for a full pass.

Step-by-Step Explanation

1. **Divide the transaction database into smaller blocks.**
    - Instead of scanning everything at once, we scan in smaller sections (e.g., 100 transactions at a time).
2. **Start scanning and counting item occurrences.**
    - If "Milk" appears in many transactions, we note it.
3. **At specific points (start of each block), we introduce new item combinations.**
    - Example: If "Milk" and "Bread" are common in the first few blocks, we start tracking "Milk & Bread" before finishing the full scan.
4. **Continue scanning and updating frequencies dynamically.**
    - This means we don’t have to wait for a full scan before finding new patterns.

Why is DIC Better?

✅ **Faster** – Fewer full scans of the database.
✅ **Smarter** – Detects frequent patterns earlier.
✅ **More efficient** – Works well for large datasets.

Simple Example

Imagine a **school library** tracking book borrowings:

Transaction ID	Books Borrowed
1	Math, Science
2	Science, English

Transaction ID	Books Borrowed
3	Math, English
4	Math, Science, English

Apriori waits until it scans **all 4 transactions** before checking which book combinations are common.

DIC starts adding "Math & Science" as a pattern earlier, maybe after just 2 transactions, making it much faster.

10. efficiency comparison of partition algm with apriori

The **Partitioning Algorithm** and **Apriori Algorithm** are both used for frequent itemset mining, but they differ in efficiency and the way they process data.

1. Apriori Algorithm - How It Works

The **Apriori Algorithm** follows these steps:

1. **Scans the database multiple times** to find frequent itemsets.
2. **Generates candidate itemsets** at each step.
3. **Eliminates infrequent itemsets** and repeats the process until no more frequent itemsets can be found.

✅ **Advantage**: Simple and widely used.
❌ **Disadvantage**: Multiple database scans make it **slow** and **inefficient** for large datasets.

2. Partitioning Algorithm - How It Works

The **Partitioning Algorithm** improves efficiency by:

1. **Dividing the database into smaller partitions** and processing them separately.
2. **Finding frequent itemsets locally in each partition**.
3. **Merging the results** to find global frequent itemsets in just **two database scans**.

✅ **Advantage**: **Faster** than Apriori because it **only scans the database twice**.
❌ **Disadvantage**: Requires **extra memory** for partition management.

3. Efficiency Comparison

Feature	Apriori Algorithm	Partitioning Algorithm
Database Scans	Multiple	Only 2
Processing Method	Scans full database repeatedly	Works on smaller partitions
Speed	Slow for large datasets	Faster and scalable
Memory Usage	Moderate	Higher (needs to store partitions)
Candidate Generation	Yes (inefficient)	No (works on frequent patterns)

✅ **Partitioning Algorithm is more efficient** than Apriori because it reduces **database scans** and processes **smaller partitions instead of the full dataset**.

- **Apriori is simple but slow** because it scans the database multiple times.
- **Partitioning Algorithm is faster and more efficient** because it scans the database only twice.

11. DBSCAN

DBSCAN (**Density-Based Spatial Clustering of Applications with Noise**) is a **smart way** to group similar data points together **without needing to know how many clusters** there are beforehand! 🎯

Imagine you are looking at a **map of a city** and trying to group areas based on **crowd density**

1. **A busy shopping mall** → Many people are close together → **Forms a cluster**.
2. **A small shop in the countryside** → Very few people around → **Considered noise (outlier)**.
3. **A market with multiple stalls** → Another crowded area → **Forms another cluster**.

DBSCAN does this automatically with **real data points**

How Does DBSCAN Work? 🛠️

It groups data points based on two simple ideas:
✅ **Points that are close together belong to the same cluster.**
❌ **Points that are far apart are considered noise.**

To do this, DBSCAN uses **two settings**:

- **ε (epsilon):** A small **circle** around a point (like a small area on the map).
- **MinPts (minimum points):** The **minimum number of points** needed to form a cluster.

Step-by-Step Explanation 📌

1️⃣ Start with a Random Point

- If there are **enough nearby points (MinPts)**, this becomes a **core point** and a **new cluster starts**.
- If not enough points are nearby, the point is **marked as noise** (outlier).

2️⃣ Expand the Cluster

- Look at all points inside the **ε radius**.
- If those points also have **enough neighbors**, they **join the cluster**.
- Keep repeating this **until there are no more points to add**.

3️⃣ Move to the Next Point

- If it's another **core point**, start a new cluster.
- If it's not, ignore it and move on.

This continues **until all points are visited**.

Example: Imagine a Classroom 🏫

👦 **Scenario 1:** Students sitting in groups → DBSCAN finds them as clusters.
👤 **Scenario 2:** A student sitting alone in a corner → DBSCAN marks them as noise (outlier).

Just like in a classroom, DBSCAN **naturally identifies** groups of similar things **without needing you to specify the number of groups**.

---

12. support , confidence, accuracy, precision, recall

Support (Used in Association Rule Mining - Apriori & FP-Growth)

Support measures **how often** an item or itemset appears in the dataset.

Support(A) = (Transactions containing A) / (Total transactions)

🔹 **Example:**

A supermarket has **100 transactions**, and "Milk & Bread" appear together in **20 transactions**.

Support(Milk, Bread) = 20 / 100 = 20%

💡 **Meaning:** 20% of customers bought Milk & Bread together.

Confidence (Used in Association Rule Mining - Apriori & FP-Growth)

Confidence tells us **how often a rule is correct** when a certain item is present.

Confidence(A ⇒ B) = Support(A, B) / Support(A)

🔹 **Example:**

- "Milk & Bread" appear in **20 transactions**.
- "Milk" appears in **40 transactions**.

Confidence(Milk ⇒ Bread) = 20 / 40 = 50%

💡 **Meaning:** If a customer buys Milk, there is a **50% chance** they will also buy Bread.

Accuracy (Used in Classification & Prediction Models)

Accuracy measures **how many predictions were correct** out of all predictions.

Accuracy = (Correct Predictions) / (Total Predictions)

🔹 **Example:**

A spam filter checks **100 emails**:

- **80 emails** were correctly classified as spam/non-spam.
- **20 emails** were classified incorrectly.

Accuracy = 80 / 100 = 80%

💡 **Meaning:** The spam filter correctly classified **80% of emails**.

Precision (Used in Classification & Information Retrieval)

Precision tells us **how many predicted positives are actually correct**.
Precision = TP / (TP + FP)

🔹 **Example:**
A medical test identifies **10 people as having a disease**:

- **8 actually have the disease** (**TP = 8**).
- **2 were wrongly classified** (**FP = 2**).

Precision = 8 / (8 + 2) = 8 / 10 = 80%

💡 **Meaning:** When the test says a person has the disease, it is **80% correct**.

Recall (Sensitivity) (Used in Classification & Information Retrieval)

Recall tells us **how many actual positives were correctly predicted**.
🔹 **Example:**
There are **12 actual patients with a disease**:

- The test correctly finds **8 of them** (**TP = 8**).
- It **misses 4** patients (**FN = 4**).

Recall = 8 / (8 + 4) = 8 / 12 = 66.7%

💡 **Meaning:** The test detects **66.7% of actual cases**.

Term	Meaning	Formula	Example
Support	How often an itemset appears	`Support = Transactions(A) / Total`	"Milk & Bread" in 20/100 transactions → **20% Support**
Confidence	How often a rule is correct	`Confidence = Support(A, B) / Support(A)`	If Milk appears in 40 transactions & Milk & Bread in 20, then **Confidence = 50%**
Accuracy	Overall correctness	`Accuracy = Correct Predictions / Total`	Spam filter is correct 80/100 times → **80% Accuracy**
Precision	Correctness of positive predictions	`Precision = TP / (TP + FP)`	8 people correctly diagnosed out of 10 → **80% Precision**

Term	Meaning	Formula	Example
Recall	How many actual positives were found	`Recall = TP / (TP + FN)`	8 cases correctly found out of 12 → **66.7% Recall**