Big Data HW 2

Parham Gilani – 400101859

سوال ۱) الف) lift(A→B) میزان ارتباط دو آیتم A و B نسبت به رخداد‌های آنها در dataset پیدا می‌کند و میزان نزدیکی مشاهده A و B به میزان مشاهده A و B جدا پیدا می‌کند. اگر lift(A→B) بزرگ‌تر از یک باشد پس به احتمال زیاد A و B باهم رخ می‌دهند و اگر کمتر از ۱ باشد پس رخداد A احتمال رخداد B را کم می‌کند به عبارتی هر دو باهم ظاهر نمی‌شوند.

ب) conv(A→B) احتمال رخداد A بدون B را نشان می‌دهد و نسبت بازه‌ای که هر دو با هم رخ بدهند. و هرچه conv(A→B) بزرگ‌تر است به این معنی است که احتمال رخداد A بدون B کم است و اگر conv(A→B) کم باشد یعنی است که احتمال رخداد A بدون B زیاد است.

ج) بجای confidence احتمال وقوع B را با توجه به A پیدا می‌کند و بدون در نظر گرفتن تعداد دفعات که B رخ داده است و شکل زیادی پیش می‌آید که B در دataset به صورت زیاد رخ داده باشد. برای مثال:

total → 100

80 → B conf(A,B) = 1

10 → A,B

که در این حالت با احتمال زیاد A→B رخ می‌دهد چون تا رخ دادن A خود رخداد B مکرر است.

پس در کل confidence نزدیکی را در نظر نمی‌گیرد در حالی که lift و conv اینها را در نظر می‌گیرند.

سوال ۲) الف)

$number\_freq = \frac{100}{5} \rightarrow number\_freq = 20 \rightarrow I \geq 20$

ب) بازای support threshold برابر با تمام جفت اعدادی که کوچک‌ترین مضرب مشترکشان کمتر از 20 باشد freq هستند.

ج)

1 2 3 --- 100

1 1,2 1,3 $2^2 \times 5^2$

total = 482

د)

$(6,9) \rightarrow 3$: $confidence = \frac{sup(6,9,3)}{sup(6,9)} = \frac{\{\frac{100}{18}\}}{\{\frac{100}{18}\}} = 1$. $(2,4,5) \rightarrow 3$: $confidence = \frac{sup(2,3,4,5)}{sup(2,4,5)} = \frac{\{\frac{100}{60}\}}{\{\frac{100}{20}\}} = \frac{1}{5} \rightarrow$

$interest = confidence - sup(3) = 1 - \frac{1}{3} = \frac{2}{3}$

$interest = confidence - sup(3) = \frac{1}{5} - \frac{1}{3} = \frac{-2}{15}$

سوال 3) الف) جفت آیتم‌ها $= \binom{I}{2} = \frac{I(I-1)}{2} \rightarrow$ میزان حافظه $= \frac{I(I-1)}{2} \times 4$ byte

ب)

maximum pairs $= \binom{P}{2} \times B$

ج)

triangle_matrix_size $= \frac{I(I-1)}{2} \times 4$ byte (i و j) → (i و c)

triple_size $= 3 \times 4$ byte $\times$ num_nonZero_pairs $\leq \frac{I(I-1)}{2} \times 4$ byte $\rightarrow$

$num\_nonZero\_pairs \leq \frac{I(I-1)}{6}$

سوال 4) الف)

x	h_1	h_2	h_3
0	1	3	2
1	4	1	1
2	1	5	0
3	4	3	5
4	1	1	4
5	4	5	3

S_4 : row(2,4,3)
$h_1 \rightarrow 1$
$h_2 \rightarrow 1$
$h_3 \rightarrow 0$
(1,1,0)

S_3 : row(0,1,3,5)
$h_1 \rightarrow 1$
$h_2 \rightarrow 1$
$h_3 \rightarrow 1$
(1,1,1)

S_2 : (1,2,5)
$h_1 \rightarrow 1$
$h_2 \rightarrow 1$
$h_3 \rightarrow 0$
(1,1,0)

S_1 : (0,2,4)
$h_1 \rightarrow 1$
$h_2 \rightarrow 1$
$h_3 \rightarrow 0$
(1,1,0)

ب) چون h_3 مقادیر یکتا بازای x از 5 تا دارد پس معتبر است.

ج) چون minhash آنها تقریبا یکی شده است پس 6 hash متوانستند آنها را از هم جدا کنند و اگر hash های h_1 و h_2 معتبر بودند می‌توانستیم به حالت بهینه‌تری برسیم.

pair	∩	∪	Jacc	minhash ∩	minhash sim
4,3	3	0→5	1/6	2 num	2/3
4,2	2	1→5	1/5	3 num	1
4,1	2,4	0,2,4	1/2	3 num	1
3,2	1,5	0→3,5	2/5	2 num	2/3
3,1	0	0→5	1/6	2num	2/3
2,1	2	0,1,2,4,5	1/5	3num	1

Q1)

A. In this section we have found the most frequent words in first layer and making triplets in second layer. Here is the details:

First Pass: Identifying Frequent Words

- **Tokenization and Counting:**
  - Each abstract is split into individual words.
  - The occurrences of each word across all abstracts are counted using PySpark transformations (`flatMap`, `map`, `reduceByKey`).
- **Filtering Frequent Words:**
  - Words that appear fewer times than the specified `first_pass_support_threshold` (e.g., 300 occurrences) are discarded.
  - A set of frequent words is returned, which is used in the second pass to restrict the scope of analysis.

Second Pass: Identifying Frequent Trigrams

- **Trigram Generation:**
  - For each abstract, all possible trigrams (combinations of three unique words) are generated using Python's `itertools.combinations`.
  - Only trigrams where all three words are in the frequent word set (from the first pass) are retained, ensuring efficiency.

- **Counting Trigram Occurrences:**
  - The occurrences of each trigram across all abstracts are counted using similar transformations.

- **Filtering Frequent Trigrams:**
  - Trigrams with occurrences below the `second_pass_support_threshold` (e.g., 1000) are discarded.
  - The remaining frequent trigrams are collected as the final result.

B. In this section the PCY algorithm is a little bit different that A-priori algorithm. Which means in the first layer after filtering the words to find the frequent words we have to use a bitmap on them and in the second layer we should make tripletes and find if they are frequent or not. I must indicate that we have a parameter named hash_table_size which means the hash table isn't unlimited and if it exceeded it will be set by the remaining of that number into the hash_table_size. Here is further details:

Steps in the Algorithm

- **Parameters**
  - **support_threshold_first_pass**: A lower threshold used in the first pass to identify potential candidates.
  - **support_threshold_second_pass**: A higher threshold used in the second pass to filter the final frequent trigrams.
  - **hash_table_size**: Size of the hash table to reduce collisions when hashing trigrams.

- **Utility Functions**
  - **hash_trigram**: Hashes a trigram using SHA-256 and maps it to a bucket in the hash table.
  - **normalize_trigram**: Sorts the words in a trigram to ensure consistency, treating `(A, B, C)` and `(C, B, A)` as the same.

First Pass: Generating Frequent Words and Populating the Hash Table

- **Word Counting**:
  - Tokenizes abstracts into words and counts their occurrences using PySpark transformations.
  - Filters words with counts below `support_threshold_first_pass` and stores the frequent words in a set.

- **Trigram Generation and Hashing**:
  - Generates all trigrams from abstracts.
  - Filters trigrams to include only those composed of frequent words from the first pass.
  - Hashes each trigram into a bucket using `hash_trigram` and populates the hash table with bucket counts.

- **Bitmap Creation**:
  - Converts the hash table into a bitmap where each bucket is marked as frequent (`True`) if its count meets the `support_threshold_first_pass`. This bitmap reduces memory usage and helps prune infrequent trigrams early.

Second Pass: Filtering and Counting Candidate Trigrams

- **Candidate Filtering Using Bitmap**:
  - Generates trigrams from abstracts, ensuring:
  - All words are frequent (from the first pass).
  - The trigram maps to a bucket marked as frequent in the bitmap.

- **Trigram Counting**:
  - Counts occurrences of the filtered trigrams across all abstracts.

- **Final Filtering**
  - Retains trigrams that meet the support_threshold_second_pass.

C. In this section I have simply used jaccard similarity to find If both algorithms are doing a single task and it returns 1 which means the answers are the same.

Q2) In this question I have implemented an LSH that finds similar papers according to their abstracts. At first, we must generate MinHash signatures in order to make Shingles, then generate multiple hash functions and use it to make some Signatures and then for each item in the Signatures we must use LSH to get the answers and then find the similar articles using them and the Jaccard similarity. Here is the further code detail:

1. **Preprocessing**
    - **Tokenization**:
        - Splits the text of each paper (e.g., abstract) into tokens (words).
        - Converts all text to lowercase for case-insensitive comparisons.
    - **Shingles Generation**:
        - Creates overlapping substrings (shingles) of length (k) from the tokens.
        - Example: For tokens ["`this`", "`is`", "`a`", "`test`"] and (k=3), shingles are: {"`thisisatest`"}

2. **MinHashing**
    - **Generate Random Hash Functions**:
        - $h(x) = (a.x + b) \% p$
    - **Create MinHash Signatures**:
        - For each paper's shingles, compute the minimum hash value under each of the hash functions.
        - A signature is an array of these minimum hash values.

MinHashing ensures that the similarity between two sets of shingles is preserved.

3. **Locality-Sensitive Hashing (LSH)**
    - **Purpose**:
        - Efficiently group and retrieve similar items by hashing similar MinHash signatures into the same buckets.
    - **Steps**:
        - **Split MinHash Signatures into Bands**:
            - Divide the signature into (b) bands, each containing (r) rows.
        - **Hash Each Band**:
            - For each band (subarray of the signature), compute a hash.
            - Papers with the same hash for a band are likely to be similar.
4. **Query for Similar Papers**
    - **Jaccard Similarity**:
        - Measures the similarity between two sets of shingles: J(A, B) = Intersection(A,B) / Union(A,B)
        - If the similarity exceeds a threshold (e.g., 0.5), the papers are considered similar.
    - **Steps**:
        - Identify "candidate" papers using LSH:
            - For the query paper, look up its bands in the LSH index to find potential matches.
        - Compute Jaccard Similarity for these candidates.
5. **Main Workflow**
    - **Data Preparation**:
        - Tokenize and shingle all papers in the dataset.
        - Compute MinHash signatures and build the LSH index.
    - **Query Similarity**:
        - For each paper, retrieve candidate matches using LSH.
        - Calculate and rank similar papers based on Jaccard Similarity.