# CSCI 4030U: Big Data Analytics

# Labs 4 and 5

Syed Naqvi
100590852

February 27, 2024

## Lab 4

a. Limiting the Apriori algorithm to only find up to frequent pairs:

```
    # Pruning: removing all infrequent item pairs from C2
    # this will generate L2
    pairs = tuple(C2)
    for pair in pairs:
        if C2[pair] < min_support:
            del C2[pair]
    L2 = C2
    # appending list of frequent pairs to frequent_sets
    set_counts.update(L2)

    frequent_sets = []
    for s in set_counts:
        frequent_sets.append(list(s))

    return frequent_sets
[3]  ✓ 0.0s                                                    Python
```

b. Downloading retail dataset for the PCY and Apriori algorithms:

LOADING DATA

```
# Load the retail dataset from the URL, this will be used in lab 4, not lab 3
def load_data_from_url(url):
    response = urllib.request.urlopen(url)
    lines = response.readlines()
    dataset = [list(map(int, line.strip().split())) for line in lines]
    return dataset

dataset = load_data_from_url("http://fimi.uantwerpen.be/data/retail.dat")
[2]  ✓ 1.7s                                                    Python
```

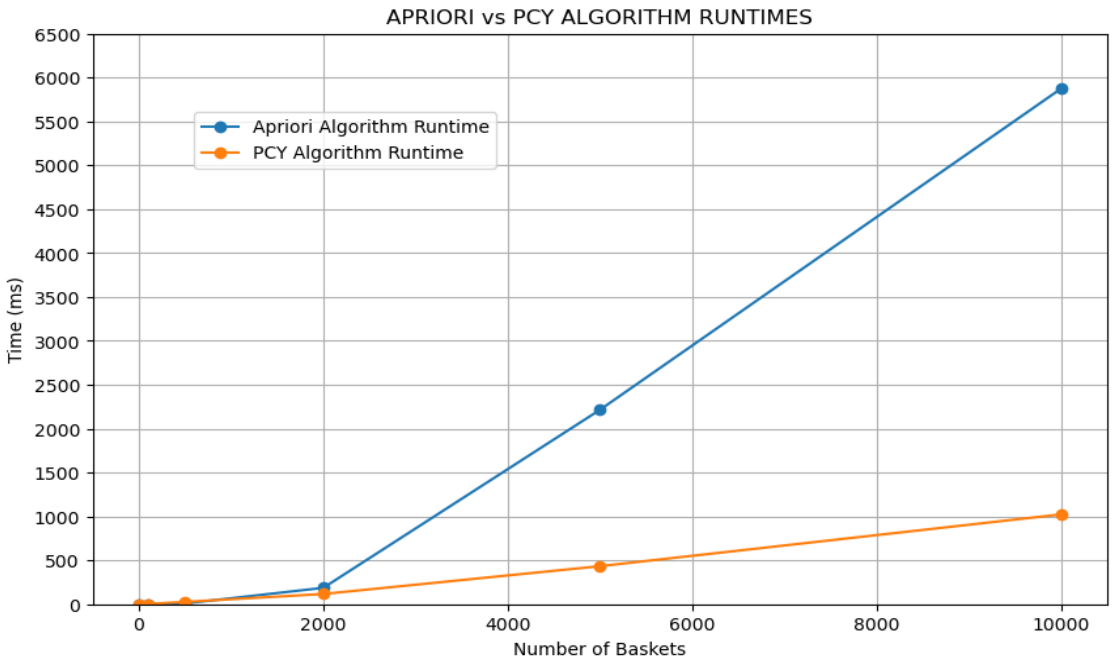
c. Comparing Apriori and PCY algorithms using the provided partitions of the dataset:

COMPARING RUNTIMES

```
# generating a sequence of partitions of the dataset
# each partition is from the oth to nth basket
partitions = [0, 100, 500, 2000, 5000, 10000]

# with roughly 16000 unique items there are roughly 135 million possible unique item pairs
# each partition could potentially contain all or most unique item pairs
    # a fully populated dictionary with 100 million int:int entries takes up roughly 7GB of memory
    # a fully populated dictionary with 10 million int:int entries takes up roughly 0.83GB of memory
# thus we avoid many bucket collisions even if a high percentage of the potential pairs do exist in the partitions,
NUM_BUCKETS = 10000000

# data points for graphing
x = [] # number of baskets
ap_y = [] # apriori runtime per basket
pcy_y = [] # pcy runtime per basket

for E in partitions:

    # extracting current dataset partition
    d = dataset[:E]
    support = 5

    ### RUNTIME OF APRIORI ###
    ap_start = time.time()
    apriori_freq_items = apriori(d, support)
    ap_end = time.time()

    ### RUNTIME OF PCY ###
    pcy_start = time.time()
    pcy_freq_items = pcy_algorithm(d, support, NUM_BUCKETS)
    pcy_end = time.time()

    x.append(E)
    ap_y.append((ap_end-ap_start)*1000)
    pcy_y.append((pcy_end-pcy_start)*1000)
```

[32] ✓ 10.9s

d. Graphing results. Dataset size is on the x-axis while runtime (ms) is on y-axis:




e. All screenshots have been provided.

f. We can see that the PCY algorithm has a much faster runtime than Apriori which seems to increase exponentially faster as the datasets increase in size. This finding is consistent with expected performance results since Apriori must generate and store all possible candidate pairs based on frequent singletons (which could be quite numerous, especially for low support values) while PCY generates candidate pairs based only on baskets encountered during a pass. Thus, PCY ends up generating far fewer pairs since the amount that actually exists in the dataset is usually much less than what is possible based on frequent singletons.

## Lab 5

a. The Jaccard similarity of two sets is the size of their intersection divided by the union. We have the following sets:

$$C_1 = \{2, 3, 4, 5\}$$
$$C_2 = \{3, 4, 6, 8\}$$
$$C_3 = \{2, 3, 6\}$$

We can now calculate the Jaccard similarity of each pair of the above sets:

$$\begin{aligned} sim(C_1, C_2) &= \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} \\ &= \frac{|\{2, 3, 4, 5\} \cap \{3, 4, 6, 8\}|}{|\{2, 3, 4, 5\} \cup \{3, 4, 6, 8\}|} \\ &= \frac{|\{3, 4\}|}{|\{2, 3, 4, 5, 6, 8\}|} \\ &= \frac{2}{6} \\ &= \frac{1}{3} = 0.3\overline{3} \end{aligned}$$

$$\begin{aligned} sim(C_1, C_3) &= \frac{|C_1 \cap C_3|}{|C_1 \cup C_3|} \\ &= \frac{|\{2, 3, 4, 5\} \cap \{2, 3, 6\}|}{|\{2, 3, 4, 5\} \cup \{2, 3, 6\}|} \\ &= \frac{|\{2, 3\}|}{|\{2, 3, 4, 5, 6\}|} \\ &= \frac{2}{5} = 0.4 \end{aligned}$$

$$\begin{aligned} sim(C_2, C_3) &= \frac{|C_2 \cap C_3|}{|C_2 \cup C_3|} \\ &= \frac{|\{3, 4, 6, 8\} \cap \{2, 3, 6\}|}{|\{3, 4, 6, 8\} \cup \{2, 3, 6\}|} \\ &= \frac{|\{3, 6\}|}{|\{2, 3, 4, 6, 8\}|} \\ &= \frac{2}{5} = 0.4 \end{aligned}$$

b. To find the expected value of the Jaccard similarity between sets $S$ and $T$ where $S, T \subseteq U$, $|S| = |T| = m$ and $|U| = n$ we utilize the definition of expected value of functions of random variables:

$$\mathbb{E}(\phi(K)) = \sum_{k \in \Omega} \phi(k) \cdot P(K = k) \tag{1}$$

The above equation states that the expected value of a function of a random variable is the sum of the function evaluated for all possible inputs multiplied by the probability of said input.

**First we find $\phi(\mathbf{k})$.**

Given that sets $S$ and $T$ are randomly selected subsets of $U$, we treat the cardinality of their intersection as a random variable and define their Jaccard similarity as a function of this random variable. We thus let $|S \cap T| = k$ and by the principle of inclusion-exclusion, note that $|S \cup T| = |S| + |T| - |S \cap T| = 2m - k$, giving us the following:

$$Sim(S, T) = \frac{|S \cap T|}{|S \cup T|} = \frac{k}{2m - k} = \phi(k) \tag{2}$$

**Next we find $\mathbb{P}(\mathbf{K = k})$.**

Note that there are $\binom{n}{k}$ ways to select $k$ elements for our intersection. The remaining $m - k$ elements of either $S$ or $T$ can be selected in $\binom{n-k}{m-k}$ ways and the remaining $m - k$ elements of the final set can be chosen in $\binom{n-m}{m-k}$ ways. Thus, there are $\binom{n}{k} \cdot \binom{n-k}{m-k} \cdot \binom{n-m}{m-k}$ ways to choose $S$ and $T$ such that $|S \cap T| = k$. The number of ways to select an arbitrary set of length $m$ from $n$ total elements is $\binom{n}{m}$ meaning there are $\binom{n}{m}^2$ ways of selecting two consecutive subsets of $m$ elements from $n$ total. We thus define the probability of having $k$ elements in our intersection:

$$\mathbb{P}(K = k) = \frac{\binom{n}{k}\binom{n-k}{m-k}\binom{n-m}{m-k}}{\binom{n}{m}^2} \tag{3}$$

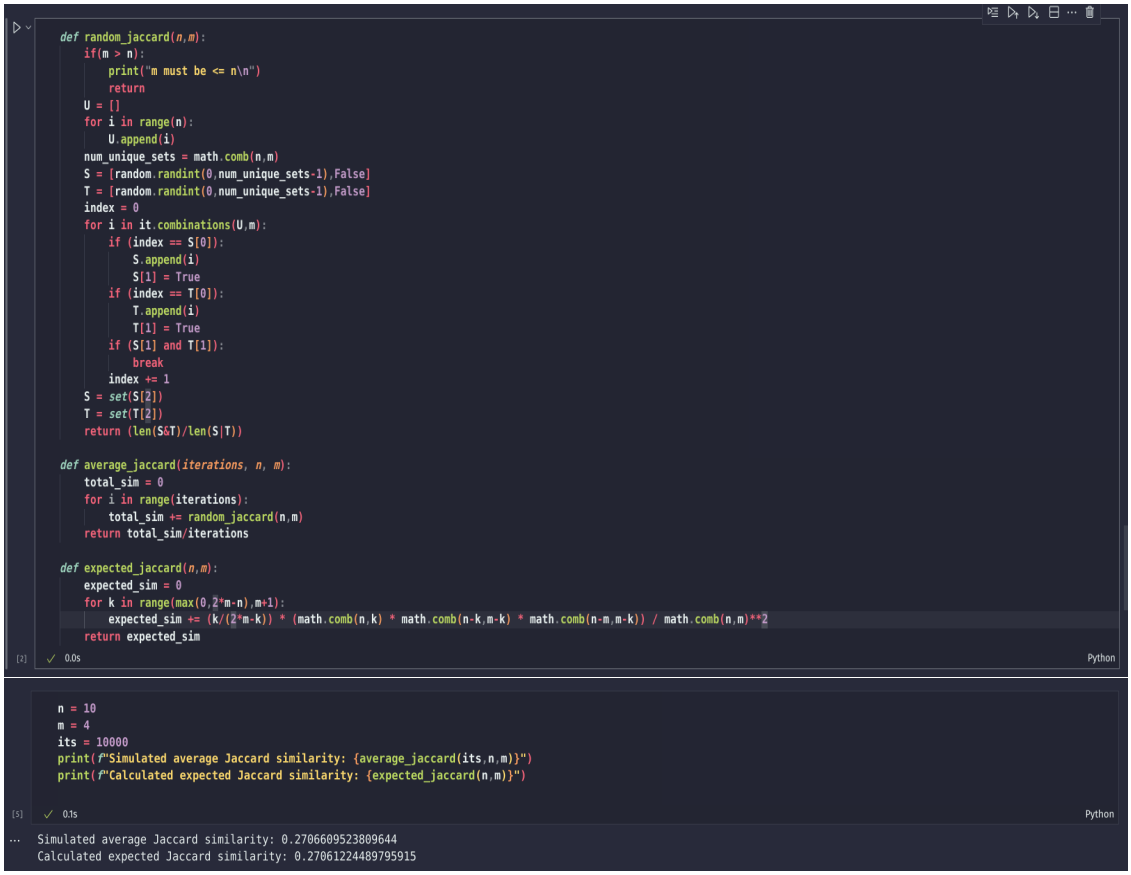
A final observation is regarding our bounds of summation. We can have a maximum intersection of $k = m$ and a minimum intersection of $k = max(0, 2m - n)$ since sufficiently large subsets will make it impossible for them to be disjoint.

**Defining $\mathbb{E}(\mathbf{Sim(S, T)})$.**

We thus arrive at our final definition for expected Jaccard similarity:

$$\mathbb{E}(Sim(S, T)) = \mathbb{E}(\phi(K)) = \sum_{k=max(0,2m-n)}^{m} \frac{k}{2m - k} \cdot \frac{\binom{n}{k}\binom{n-k}{m-k}\binom{n-m}{m-k}}{\binom{n}{m}^2} \tag{4}$$

Using simulations, we can demonstrate that the average Jaccard similarity over repeated trials converges to our calculated expected Jaccard similarity:



```python
def random_jaccard(n,m):
    if(m > n):
        print("m must be <= n\n")
        return
    U = []
    for i in range(n):
        U.append(i)
    num_unique_sets = math.comb(n,m)
    S = [random.randint(0,num_unique_sets-1),False]
    T = [random.randint(0,num_unique_sets-1),False]
    index = 0
    for i in it.combinations(U,m):
        if (index == S[0]):
            S.append(i)
            S[1] = True
        if (index == T[0]):
            T.append(i)
            T[1] = True
        if (S[1] and T[1]):
            break
        index += 1
    S = set(S[2])
    T = set(T[2])
    return (len(S&T)/len(S|T))

def average_jaccard(iterations, n, m):
    total_sim = 0
    for i in range(iterations):
        total_sim += random_jaccard(n,m)
    return total_sim/iterations

def expected_jaccard(n,m):
    expected_sim = 0
    for k in range(max(0,2*m-n),m+1):
        expected_sim += (k/(2*m-k)) * (math.comb(n,k) * math.comb(n-k,m-k) * math.comb(n-m,m-k)) / math.comb(n,m)**2
    return expected_sim
```

[2] ✓ 0.0s Python

```python
n = 10
m = 4
its = 10000
print(f"Simulated average Jaccard similarity: {average_jaccard(its,n,m)}")
print(f"Calculated expected Jaccard similarity: {expected_jaccard(n,m)}")
```

[5] ✓ 0.1s Python

```
Simulated average Jaccard similarity: 0.2706609523809644
Calculated expected Jaccard similarity: 0.27061224489795915
```

c. If we assume the number of possible strings of length $k$ is at least $n$, and we let $k = 1$, then there must be $n$ unique 1-strings which implies there must be $n$ unique characters in our document and so all k-shingles must be unique. If we now imagine a k-shingle with $1 \leq k \leq n$ positioned such that the first character of the shingle is the first character of our document, we will be able to shift this shingle $n - k$ positions until the last character of the k-shingle is the last character of the document. Since every translation corresponds to a unique k-shingle we have shown there are at least $n - k$ total k-shingles. Finally, we add our initially positioned k-shingle to the count and conclude that there can be a maximum of $n - k + 1$ k-shingles in a document of $n$ bytes.