Department of Computer Science and Engineering (Data Science)

Subject: Machine Learning – IV Laboratory
AY: 2023-24

Experiment 7

Name: Jhanvi Parekh **SAP Id:** 60009210033

Aim: Implement the PCY (Park, Chen, and Yu) Algorithm for efficient Market Basket Analysis to identify frequent itemsets in large transaction datasets.

Theory:

Introduction: The PCY Algorithm is an advanced approach to Market Basket Analysis, specifically designed to handle large-scale transaction datasets. This algorithm introduces hash functions and a hash table to significantly reduce the number of candidate itemsets, thereby improving the overall efficiency of the association rule mining process.

PCY Algorithm Workflow:

1. Calculate Singleton Support:
   - Begin by counting the occurrences of each item in the dataset. Identify singleton items with support greater than or equal to the specified minimum support.
2. Hash Table Construction:
   - Create a hash table and utilize hash functions to count pairs of items in the dataset efficiently.
3. Filtering using Hash Table:
   - Filter pairs based on the hash table, considering only those pairs with counts above the minimum support threshold.
4. Candidate Set Generation:
   - Generate the final candidate set based on the filtered pairs obtained from the hash table.

Lab Exercise:

Implement the PCY Algorithm using Python.

Sample Problem:

Given Data:

```
Threshold value or minimization value = 3
Hash function= (i*j) mod 10

    T1  =   {1, 2, 3}
    T2  =   {2, 3, 4}
    T3  =   {3, 4, 5}
    T4  =   {4, 5, 6}
    T5  =   {1, 3, 5}
    T6  =   {2, 4, 6}
    T7  =   {1, 3, 4}
    T8  =   {2, 4, 5}
    T9  =   {3, 4, 6}
    T10 =   {1, 2, 4}
    T11 =   {2, 3, 5}
    T12=    {3, 4, 6}
```

Solution:
Step 1: Mapping all the elements in order to find their length.

```
Items →    {1, 2, 3, 4, 5, 6}
Key         1  2  3  4  5  6
Value       4  6  8  8  6  4
```

Step 2: Removing all elements having value less than 1.

But here in this example there is no key having value less than 1. Hence, candidate set = {1, 2, 3, 4, 5, 6}

Step 3: Map all the candidate set in pairs and calculate their lengths.

```
T1: {(1, 2) (1, 3) (2, 3)} = (2, 3, 3)
T2: {(2, 4) (3, 4)} = (3 4)
T3: {(3, 5) (4, 5)} = (5, 3)
T4: {(4, 5) (5, 6)} = (3, 2)
T5: {(1, 5)} = 1
T6: {(2, 6)} = 1
T7: {(1, 4)} = 2
T8: {(2, 5)} = 2
T9: {(3, 6)} = 2
T10:______
T11:______
T12:______
```

Step 4: Apply Hash Functions. (It gives us bucket number)

```
Hash Function = ( i * j) mod 10
(1, 3) = (1*3) mod 10 = 3
(2,3) = (2*3) mod 10 = 6
(2,4) = (2*4) mod 10 = 8
(3,4) = (3*4) mod 10 = 2
(3,5) = (3*5) mod 10 = 5
(4,5) = (4*5) mod 10 = 0
(4,6) = (4*6) mod 10 = 4
```

Now, arrange the pairs according to the ascending order of their obtained bucket number.

Bucket no.	**Pair**
0	(4,5)
2	(3,4)
3	(1,3)
4	(4,6)
5	(3,5)
6	(2,3)
8	(2,4)

Step 5: In this final step we will prepare the candidate set.

Bit vector	Bucket no.	Highest Support Count	Pairs	Candidate Set
1	0	3	(4,5)	(4,5)
1	2	4	(3,4)	(3,4)
1	3	3	(1,3)	(1,3)
1	4	3	(4,6)	(4,6)
1	5	5	(3,5)	(3,5)
1	6	3	(2,3)	(2,3)
1	8	3	(2,4)	(2,4)

Hence, the frequent itemsets are (4, 5), (3,4)

Code with output:

Normal PCY Algorithm:

```
from itertools import combinations
from collections import defaultdict
from prettytable import PrettyTable

def hash_function(i, j, num_buckets):
    return (i * j) % num_buckets

def  pcy_algorithm(transactions, min_support):
    item_count = defaultdict(int)

    # Count individual items
    for transaction in transactions:
        for item in transaction:
            item_count[item] += 1

    # Find frequent items
    frequent_items = {item for item, count in item_count.items() if count >= min_support}
    pair_count = defaultdict(int)

    # Count pairs of frequent items
    for transaction in transactions:
        for i, j in combinations(frequent_items.intersection(transaction), 2):
            pair_count[(i, j)] += 1

    num_buckets = 10
    bucket_count = [0] * num_buckets
    bucket_pairs = defaultdict(list)

    # Hash pairs into multiple buckets
    for pair, count in pair_count.items():
        if count >= min_support:
            bucket_index = hash_function(pair[0], pair[1], num_buckets)
            bucket_count[bucket_index] += count
            bucket_pairs[bucket_index].append(pair)

    # Collect candidates based on bucket counts
    candidates = []
    for bucket_index, pairs in bucket_pairs.items():
        if bucket_count[bucket_index] >= min_support:
            candidates.extend(pairs)

    candidate_counts = {pair: pair_count[pair] for pair in candidates}
```

```
output_table = []
for bucket_index, pairs in bucket_pairs.items():
    bit_vector = [1]
    highest_support_count = max(candidate_counts[pair] for pair in pairs)
    output_table.append({
        'Bit Vector': bit_vector,
        'Bucket Number': bucket_index,
        'Highest Support Count': highest_support_count,
        'Pairs': pairs,
        'Candidate Set': candidates
    })

return output_table
```

Output:

```
+------------+---------------+----------------------+----------+-------------------------+
| Bit Vector | Bucket Number | Highest Support Count |  Pairs   |      Candidate Set      |
+------------+---------------+----------------------+----------+-------------------------+
|    [1]     |       8       |          5           | [(2, 4)] | [(2, 4), (3, 4), (4, 6)] |
|    [1]     |       2       |          4           | [(3, 4)] | [(2, 4), (3, 4), (4, 6)] |
|    [1]     |       4       |          4           | [(4, 6)] | [(2, 4), (3, 4), (4, 6)] |
+------------+---------------+----------------------+----------+-------------------------+
Time taken: 0.000000 seconds
```

MultiStage PCY Algorithm:

```
def hash_function(i, j, num_buckets):
    return (i * j) % num_buckets

def pcy_multistage(transactions, min_support, num_stages):
    item_count = defaultdict(int)

    for transaction in transactions:
        for item in transaction:
            item_count[item] += 1

    frequent_items = {item for item, count in item_count.items() if count >=
min_support}

    candidates = frequent_items
    output_table = []

    for stage in range(num_stages):
        pair_count = defaultdict(int)
        num_buckets = 10
        bucket_count = [0] * num_buckets
        bucket_pairs = defaultdict(list)
        for transaction in transactions:
            current_items = candidates.intersection(transaction)
            for i, j in combinations(current_items, 2):
                pair_count[(i, j)] += 1

        for pair, count in pair_count.items():
            if count >= min_support:
                bucket_index = hash_function(pair[0], pair[1], num_buckets)
                bucket_count[bucket_index] += count
```

```
                    bucket_pairs[bucket_index].append(pair)

            new_candidates = set()
            for bucket_index, pairs in bucket_pairs.items():
                if bucket_count[bucket_index] >= min_support:
                    new_candidates.update(pairs)

            if new_candidates:
                output_table.append({
                    'Stage': stage + 1,
                    'Candidates': new_candidates,
                    'Bucket Counts': bucket_count,
                    'Pair Counts': {pair: pair_count[pair] for pair in new_candidates}
                })
            candidates = new_candidates
            if not candidates:
                break

    return output_table
```

Output:

```
+-------+------------+--------------------------------+-------------+
| Stage | Candidates |         Bucket Counts          | Pair Counts |
+-------+------------+--------------------------------+-------------+
|   1   |  {(2, 4)}  | [0, 0, 0, 0, 0, 0, 0, 0, 5, 0] | {(2, 4): 5} |
+-------+------------+--------------------------------+-------------+
Time taken: 0.000000 seconds
```

MultiHash PCY:

```
from itertools import combinations
from collections import defaultdict
from prettytable import PrettyTable

def hash_function1(pair):
    return (pair[0] + pair[1]) % 10

def hash_function2(value):
    return (value * 3 + 5) % 10

def hash_function3(value):
    return (value * 7 + 2) % 10

def multi_hash(pair):
    first_hash = hash_function1(pair)
    second_hash = hash_function2(first_hash)
    third_hash = hash_function3(second_hash)

    return first_hash, second_hash, third_hash

def pcy_algorithm_multihash(transactions, min_support):
    item_count = defaultdict(int)
```

```
    for transaction in transactions:
        for item in transaction:
            item_count[item] += 1

    frequent_items = {item for item, count in item_count.items() if count >= min_support}
    pair_count = defaultdict(int)

    for transaction in transactions:
        for i, j in combinations(frequent_items.intersection(transaction), 2):
            pair_count[(i, j)] += 1

    num_buckets = 10
    bucket_count = [0] * num_buckets
    bucket_pairs = defaultdict(list)

    for pair, count in pair_count.items():
        if count >= min_support:
            for bucket_index in multi_hash(pair):
                bucket_count[bucket_index] += count
                bucket_pairs[bucket_index].append(pair)

    candidates = []

    for bucket_index, pairs in bucket_pairs.items():
        if bucket_count[bucket_index] >= min_support:
            candidates.extend(pairs)

    candidate_counts = {pair: pair_count[pair] for pair in candidates}

    output_table = []
    for bucket_index, pairs in bucket_pairs.items():
        bit_vector = [1]
        highest_support_count = max(candidate_counts.get(pair, 0) for pair in pairs)
        output_table.append({
            'Bit Vector': bit_vector,
            'Bucket Number': bucket_index,
            'Highest Support Count': highest_support_count,
            'Pairs': pairs,
            'Candidate Set': candidates
        })

return output_table
```

Output:

```
+------------+---------------+-----------------------+------------------+--------------------------+
| Bit Vector | Bucket Number | Highest Support Count |      Pairs       |      Candidate Set       |
+------------+---------------+-----------------------+------------------+--------------------------+
|    [1]     |       6       |           5           |     [(2, 4)]     | [(2, 4), (2, 4), (2, 4)] |
|    [1]     |       3       |           5           | [(2, 4), (2, 4)] | [(2, 4), (2, 4), (2, 4)] |
+------------+---------------+-----------------------+------------------+--------------------------+
Time taken: 0.000000 seconds
```

Conclusion:

The PCY Algorithm is a powerful tool for optimizing Market Basket Analysis, particularly when dealing with extensive transaction datasets. This experiment provides hands-on experience in implementing and understanding the efficiency gains achieved by incorporating hash functions and hash tables into the association rule mining process. The PCY Algorithm is crucial in scenarios where computational efficiency is paramount, making it an asset in the realm of data mining.