VIETNAM GENERAL CONFEDERATION OF LABOUR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**

A red and blue logo

Description automatically generated

**TRAN LE GIA BAO**

**NGUYEN TRAN QUANG HUY**

**MINING PROBABILISTIC MAXIMAL HIGH-UTILITY FREQUENT ITEMSETS**

**RESEARCH TOPIC 1**

**MAJOR**

**SOFTWARE ENGINEERING**

HO CHI MINH CITY, YEAR 2024

VIETNAM GENERAL CONFEDERATION OF LABOUR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**

A red and blue logo

Description automatically generated

**TRAN LE GIA BAO**

**NGUYEN TRAN QUANG HUY**

**MINING PROBABILISTIC MAXIMAL HIGH-UTILITY FREQUENT ITEMSETS**

**RESEARCH TOPIC 1**

**MAJOR**

**SOFTWARE ENGINEERING**

Advised by

**Ph. D Nguyen Chi Thien**

HO CHI MINH CITY, YEAR 2024

**Acknowledgment**

We express our profound gratitude to the Faculty of Information Technology for granting us access to the resources necessary for completing our report. Our heartfelt thanks also go to Dr. Nguyen Chi Thien for his invaluable guidance throughout the process.

We acknowledge that our report may have imperfections due to our limited knowledge and experience. We eagerly anticipate your feedback and suggestions, as they are essential for our growth and improvement in skills and expertise.

Once again, we would like to extend our sincere gratitude!

**DECLARATION OF AUTHORSHIP**

This thesis is the result of our collective efforts, undertaken with the guidance and supervision of Dr. **Nguyen Chi Thien**. We affirm that the content and findings presented herein are original and have not been submitted for any other purpose previously. All data and figures included in this thesis were collected and analyzed by us, drawing from a variety of sources. Wherever external information has been used, it has been properly cited in the references section.

Furthermore, we have recognized and explicitly cited comments, critiques, and data from other authors and organizations used in our work.

We will bear full responsibility for any instances of fraud identified in our thesis. Ton Duc Thang University holds no accountability for any copyright violations that may arise from our work.

Ho Chi Minh City, Day 10 Month 03 Year *2024*

*Author*

(signature and full name)

*Nguyễn Trần Quang Huy*

*Trần Lê Gia Bảo*

**Abstract**

Traditional data mining methods struggle under the weight of uncertain data’s numerous potential interpretations, which introduces significant computational complexity and inefficiency. The pursuit of innovative solutions to this complex issue, alongside its practical implications, has naturally led us to dedicate our research efforts to this field.

Our primary aim is to develop the APFI\_MAX\_TopK algorithm, an advanced algorithm designed for the efficient processing and analysis of uncertain data to identify valuable probabilistic maximal frequent itemsets (PMFIs). This algorithm distinguishes itself by introducing a dual-phase strategy: first, generating the top-ranking-k candidate itemsets with the potential to be PMFIs, and second, confirming these candidates and ranking them according to their likelihood. The approach innovates by leveraging approximation methods to significantly reduce computational demands without sacrificing the accuracy of the results.

# Contents

[Acknowledgment i](#_Toc161647197)

[Abstract iii](#_Toc161647198)

[Contents 1](#_Toc161647199)

[Abbreviations 2](#_Toc161647200)

[List of Images 3](#_Toc161647201)

[List of Tables 4](#_Toc161647202)

[Introduction 5](#_Toc161647203)

[Reason for choosing the topic 5](#_Toc161647204)

[Objectives of implementing the topic 6](#_Toc161647205)

[Related works 7](#_Toc161647206)

[Problem Definition 8](#_Toc161647207)

[Algorithm Design 11](#_Toc161647208)

[Top-ranking-k with Candidate Generation 13](#_Toc161647209)

[PMFIs Conforming 22](#_Toc161647210)

[Solution 29](#_Toc161647211)

[Implementation using Java Object 30](#_Toc161647212)

[Implementation using Java Eclipse Primitives: 34](#_Toc161647213)

[Environment and Parameters for Evaluation 29](#_Toc161647214)

[Experiment 39](#_Toc161647215)

[Setup 39](#_Toc161647216)

[Results 41](#_Toc161647217)

[Discussion 45](#_Toc161647218)

[Conclusion 46](#_Toc161647219)

[Data availability 47](#_Toc161647220)

[References 48](#_Toc161647221)

# Abbreviations

UD Uncertain Database

PMFIs Probabilistic Maximal Frequent Items

PFIs Probabilistic Frequent Itemsets

PMF Probability Mass Function

MFIs Maximal frequent Itemsets

# List of Images

[Figure 5‑1 Class diagram using Java Object 32](#_Toc161940459)

[Figure 5‑2 Class diagram using Java Eclipse Primitives: 36](#_Toc161940460)

[Figure 6‑1 Compare time between Java Object and Java Eclipse Primitive 43](#_Toc161940461)

[Figure 6‑2 Compare memory between Java Object and Java Eclipse Primitive 43](#_Toc161940462)

[Figure 6‑3 Compare time between Java Object and Java Eclipse Primitive 44](#_Toc161940463)

[Figure 6‑4 Compare memory between Java Object and Java Eclipse Primitive 44](#_Toc161940464)

# List of Tables

[Table 3‑1 Example: An Example of Uncertain Transaction Database 8](#_Toc161940582)

[Table 3‑2 Probabilistic Outcome Table for Transaction Item Sets 9](#_Toc161940583)

[Table 6‑1 Introduction Dataset 40](#_Toc161940584)

[Table 6‑2 Time and Memory running with a Java Object 41](#_Toc161940585)

[Table 6‑3 Time and Memory running with a Java Eclipse Primitive 42](#_Toc161940586)

# Introduction

In the dynamic field of data analytics, the extraction of frequent itemsets from vast datasets has become a fundamental process for identifying significant patterns and associations. This is particularly important in sensor networks and related areas where data is incessantly generated, necessitating efficient and precise pattern recognition for event detection and decision-making. The intrinsic uncertainty of sensor data, due to variables like noise, measurement discrepancies, and incomplete data, presents considerable challenges to conventional mining algorithms.

Acknowledging this, the intricacy and sheer amount of data produced by modern sensor networks call for innovative techniques capable of handling this uncertainty. Traditional algorithms for mining frequent itemsets are reliable within certain data environments but are inadequate for accommodating the probabilistic nature of real-world data. This shortfall prompts the need for tailored algorithms adept at extracting maximal frequent itemsets (MFIs) amidst the uncertain landscape of sensed data.

Our study introduces an algorithm designed for approximating the mining of probabilistic maximal frequent itemsets (PMFIs) from uncertainly sensed data. Our approach is rooted in a two-pronged process: the initial generation of PMFI candidates, followed by their validation through a robust approximation technique. We apply our method across various datasets, validating the efficiency of our algorithm in sifting meaningful patterns from data replete with uncertainty, thereby contributing to advanced event detection and analysis in sensor networks and other applications.

## Reason for choosing the topic

In our report, we aim to provide an in-depth analysis of our unanimous choice to explore the research subject "*Mining probabilistic maximal high-utility frequent itemsets*." [1] Our fascination with this subject stems from the growing integration of sensor networks in diverse fields such as environmental surveillance, health care, and industrial operations. These networks accumulate a wealth of data, which is frequently laced with uncertainties due to various factors like noise, sensor malfunctions, or unpredictable environmental changes.

The core challenge that captured our interest is the effective mining of this uncertain data [2] to unearth significant patterns or events crucial for making well-informed decisions. Conventional data mining methods falter when faced with the vast number of potential interpretations of uncertain data [3], leading to substantial computational complexities and inefficiencies. Given its challenging nature, the innovative solutions it proposes, and its relevance to real-world issues, this topic has become the unequivocal choice for our current research project.

## Objectives of implementing the topic

We will assess the performance and scalability of our APFI\_MAX\_TopK algorithm by starting with smaller datasets, initially using the 'chess' dataset, which consists of 3,196 transactions. We will then progress to the 'Mushroom' dataset, containing 8,124 transactions, before finally tackling the comprehensive 'T10I4D100K' dataset, which includes 100,000 transactional records. This incremental approach will allow us to systematically validate the algorithm's efficiency in processing and analyzing vast quantities of data, ensuring it can handle increasingly complex datasets with precision.

Java, renowned for its robustness and platform-independent capabilities, has been selected as the programming language for developing our algorithm. We aim to enhance our development workflow by incorporating Maven, a powerful tool for project management and build automation. Maven's adeptness in managing dependencies will be crucial, especially for integrating the Java Eclipse Primitive Collections. These collections are designed for high-performance and memory-efficient data handling, which is essential for processing the large-scale data involved in our project.

Upon the culmination of the development and rigorous testing, our objective is to graphically depict the performance of the APFI\_MAX\_TopK algorithm, comparing it with existing benchmarks in the literature, thereby demonstrating its effectiveness and utility in the field of data mining under uncertainty.

# Related works

Initially, it points out the efficacy of well-known algorithms like Apriori [4] [5] and FP-growth [6] in exact databases but underscores their limitations in uncertain databases due to the probabilistic nature of itemset support. To navigate these challenges, the document categorizes existing methods into two primary groups based on their distinct definitions of frequent items in uncertain databases [7].

The first group bases its approach on the expected support of itemsets, utilizing algorithms like U-Apriori and UFP-growth, which are essentially extensions of techniques used in exact databases [8]. However, this approach may overlook critical information about item frequencies due to its reliance on expected support alone.

In response to the limitations of expected support-based methods, a new definition of frequent items, termed probabilistic frequent itemsets (PFIs), is introduced [9]. This concept seeks to encapsulate deeper insights by aggregating the frequencies of itemsets across various potential interpretations of the data. This leads to the development of algorithms like DP and DC, designed to mine PFIs more effectively.

Despite these advancements, the document acknowledges the inefficiency of these methods due to the exponential number of possible worlds they need to consider [10]. It highlights recent efforts focused on mining maximal frequent itemsets, which can represent all frequent itemsets efficiently, thus reducing computational costs and memory requirements [11].

Finally, the document also addresses the significant computational challenge posed by the need to compute probability mass functions (pmf), which has a time complexity of O(nlogn), making it time-consuming for large datasets [12].

# Problem Definition

**Uncertain Transaction Database**:

An uncertain transaction database is a type of uncertain database where data uncertainty is represented in transactions. Uncertain data refers to data collected from sensors or similar sources that have inherent uncertainties due to various factors such as noise, errors in measurement, or incompleteness.

Table 3‑1 Example: An Example of Uncertain Transaction Database

|  |  |  |
| --- | --- | --- |
| **ID** | **Attributes** | **Probability** |
| T1 | ABCD | 0.5 |
| T2 | BCD | 0.6 |
| T3 | ABD | 0.7 |

**Probabilistic Outcome for Transaction Item Sets:**

In statistical terms, the table enumerates the likelihood of occurrence for each distinct set of items that may result from a series of transactions. This representation calculates the probabilities under the assumption that each item's presence in a transaction is an independent event.

Table 3‑2 Probabilistic Outcome Table for Transaction Item Sets

|  |  |  |
| --- | --- | --- |
| **Transactions** | **Set of Items** | **Probability** |
| null | null | (1 - 0.5) \* (1 - 0.6) \* (1 - 0.7) = 0.06 |
| T1 | ABCD | 0.5 \* (1 - 0.6) \* (1 - 0.7) = 0.06 |
| T1, T2 | ABCD, BCD | 0.5 \* 0.6 \* (1 – 0.7) = 0.09 |
| T1, T3 | ABCD, ABD | 0.5 \* (1 - 0.6) \* 0.7 = 0.14 |
| T1, T2, T3 | ABCD, BCD, ABD | 0.5 \* 0.6 \* 0.7 = 0.21 |
| T2 | BCD | (1 - 0.5) \* 0.6 \* (1 - 0.7) = 0.09 |
| T2, T3 | BCD, ABD | (1 - 0.5) \* 0.6 \* 0.7 = 0.21 |
| T3 | ABD | (1 - 0.5) \* (1 - 0.6) \* 0.7 = 0.14 |

**Probabilistic Frequent Itemsets**

Probabilistic Frequent Itemsets (PFIs) in the context of uncertain databases are defined by considering the sum of their frequent probabilities across all.

First, the support **x** (Sup(**x**)) of an itemset **x** is the number of times **x** occurs in the dataset, the support of **x** is at least as large or equal as the minimum support threshold (minsup), as stated in formula 3.1:

(3.1)

An itemset x is deemed a Probabilistic Frequent Itemset if the sum of the probabilities (P) that its support exceeds a given minimum support threshold in these possible worlds is greater than or equal to a minimum probabilistic frequent threshold (minpro), as indicated by formula 3.2:

(3.2)

**Probabilistic Maximal Frequent Item**

A Probabilistic Maximal Frequent Itemset (PMFI) is defined as an itemset that is not only frequent but also maximal in a probabilistic sense within an uncertain database. This means that a PMFI is an itemset that meets the minimum support threshold with a probability greater than or equal to a specified probabilistic threshold, and there is no superset of this itemset that also meets these criteria.

If you have an itemset Y that is probabilistic frequent, and X is a subset of Y (denoted X Y), then for X to be a maximal probability frequent itemset, it must be equal to Y - implying that there is no larger itemset than X that has the same probabilistic frequency property. Essentially, X is one of the largest itemsets that satisfy the frequency criterion, making it a maximal element in the set of all frequent itemsets in the database.

# Algorithm Design

**TODIS-MAX**

TODIS-MAX is its top-down approach for the confirmation of PMFIs candidates, starting from the longest itemsets and progressing to shorter ones. This framework significantly improves over traditional methods by quickly yielding potential PMFIs during the candidate generation phase.

However, one of the primary challenges associated with TODIS-MAX is the computation of the probability mass function (pmf) of an itemset, which is essential for confirming its frequency. This computation has at least a time complexity of O(nlogn), making it less efficient when dealing with large-scale data.

**TOP-RANKING-K**

Top-ranking-k [13] of mining uncertain databases refers to the process of identifying the top k frequent patterns from databases where data items have associated probabilities that reflect their likelihood of presence. This approach is especially useful in scenarios where databases are derived from noisy data sources, such as sensor readings, where the exact presence of an item in a transaction might not be certain.

**CHERNOFF BOUND**

The Chernoff Bound is a probabilistic inequality that provides a bound on the tail distribution of sums of independent random variables. It is a sharper bound compared to other bounds like the Markov or Chebyshev inequalities, especially for the tail probabilities of binomial distributions or sums of random variables that can be bounded by binomial distributions.

Let be independent random variables taking values in {0,1}. Let be the sum of these variables and let be the expected value of .

For any the Chernoff Bound [14], expressed by formulas 3.3 and 3.4, play a crucial role in probability theory and statistical analysis. These bounds provide upper and lower tail inequalities that quantify the probability of a random sum X deviating from its mean by a certain factor δ:

**Upper Tail Inequality:**

(4.1)

**Lower Tail Inequality:**

(4.2)

These inequalities show that the probability of the random sum eviating from its mean by a factor of decreases exponentially fast with .

Chernoff bounds are particularly useful for demonstrating that the sum of independent random variables is highly concentrated around its expected value, making them powerful tools in algorithms and combinatorial probability.

**APFI\_MAX\_TopK algorithm**

The APFI\_MAX\_TopK algorithm is specifically designed for identifying Probabilistic Maximal Frequent Itemsets (PMFIs) within uncertain data sources, particularly focusing on the complexities of noisy sensor data. This method incorporates a top-ranking-k mechanism alongside an approximation strategy to enhance its efficiency. The algorithm unfolds through two distinct phases:

**1. Top-ranking-k Candidate Generation**: At the outset, APFI\_MAX\_TopK embarks on generating PMFI candidates by applying a sophisticated approach that employs a bound on the expected support of itemsets. This innovative method efficiently sifts through potential candidates, spotlighting the top-k itemsets based on their expected support. This targeted filtration significantly condenses the candidate set, thereby streamlining subsequent analytical processes and reducing the overall time required for execution.

**2. Confirmation and Ranking of PMFIs**: Once the candidate pool is refined, the algorithm delves into verifying which itemsets genuinely qualify as PMFIs, with a particular emphasis on those within the top-k bracket.

## Top-ranking-k with Candidate Generation

**Input**:

* “***UD UD***”: An instance of the “***UD***” class, representing the uncertain database. This object includes the transactions in the database along with the probabilities associated with each item within these transactions.
* “***int minsup***”: The minimum support threshold. This is the minimum number of transactions in which a pattern must appear to be considered significant.
* “***double minpro***”: The minimum probability threshold. This threshold is used to filter out patterns based on their calculated probability. From this and minsup, the lower bound (***lb***) formula is used to filter out patterns whose expected support is less than the minimum required for them to be considered significant.
* “***int k***”: Specifies the number of top patterns to return. The function will identify the top k patterns that meet the minimum support and probability thresholds.

**Process**:

* Initializes a list of sets ***“F”*** to hold candidate itemsets and a priority queue ***“topKPatterns”*** to maintain the top-k patterns based on their exception value ***“E”***.
* Iterates through each candidate set in ***“F”***, calculating its support, exception value ***“E”***, variance, and probability. If a candidate meets the minimum support and its ***“E”*** value is greater than the lower bound calculated by ***lb(minsup, minpro)***, it's considered a valid pattern.
* Adds valid patterns to ***“topKPatterns”***, ensuring it never holds more than ***“k”*** elements by removing the pattern with the lowest ***“E”*** value if necessary.
* Generates the next level of candidate sets from the current valid patterns and repeats the process until no new candidates can be generated or the set of candidates is empty.
* Converts the priority queue to a list and sorts it in descending order of ***“E”*** before returning.

**Output**:

***“List<C>”***: A list of ***“C”*** objects, where each ***“C”*** object represents a pattern that has met the minimum support and probability thresholds. Each ***“C”*** object includes:

* The set of items in the pattern.
* The calculated exception value ***“E”*** for the pattern.
* The variance of the pattern.
* The count of transactions ***(j)*** in which the pattern appears.
* The probability of the pattern.

The output list is sorted in descending order of the exception value “E”, presenting the top-k significant patterns in the uncertain database.

**Explain the Algorithm 1**

Given Table (Table 2 Example):

* T1: Attributes = ABCD, Pro = 0.5
* T2: Attributes = BCD, Pro = 0.6
* T3: Attributes = ABD, Pro = 0.7

Probabilistic: {A = 0.5 \* 0.7 = 0.35 ; B = 0.5 \* 0.6 \* 0.7 = 0.21 ; C = 0.5 \* 0.6 = 0.3; D = 0.5 \* 0.6 \* 0.7 = 0.21}

1. **Initialization**:

* List L = {A, B, C, D}
* minsup = 1, minpro 0.3, k = 3;
* Calculate lowbound = 0.5604628493881512

1. **Calculate Expectation for Single Items**:

* \*A\*: Exp(A) = 0.35 + 0.35 = 0.7
* Count = 2 (Present in T1 and T3).
* Since Count >= minsup(1) and Exp(A) = 0.7 > lb(1, 0.3) ≈ 0.56, A become a candidate itemset.
* \*B\*: Exp(B) = 0.21 + 0.21 + 0.21 = 0.63.
* Count = 3 (Present in T1,T2 and T3).
* Since Count >= minsup(1) and Exp(B) = 0.63 > lb(1, 0.3) ≈ 0.56, B become a candidate itemset.
* \*C\*: Expc(C) = 0.3 + 0.3 = 0.6.
* Count = 2 (Present in T1 and T3).
* Since Count >= minsup(1) and Expc(C) = 0.6 > lb(1, 0.3) ≈ 0.56, C become a candidate itemset.
* \*D\*: Exp(D) = 0.21 + 0.21 + 0.21 = 0.63.
* Count = 3 (Present in T1,T2 and T3).
* Since Count >= minsup(1) and Exp(D) = 0.63 > lb(1, 0.3) ≈ 0.56, D become a candidate itemset.
* **Single Items: A, B, C and D.**

1. **Generate and Calculate Expectation for Pair Itemsets**:

* Combine items from C1 to form pairs: {AB, AC, AD, BC, BD, CD}.
* \*AB\*: Exp(AB) = Exp(A) \* Exp(B) = 0.7 \* 0.63 = 0.441.
* Count = 2 (Present in T1 and T3).
* Since Count >= minsup(1) and Exp(AB) = 0.441 is NOT greater than lb(1, 0.3) ≈ 0.56.
* \*AC\*: Exp(AC) = Exp(A) \* Exp(C) = 0.7 \* 0.6 = 0.42.
* Count = 1 (Present only in T1).
* Since Count >= minsup (1) and Exp(AC) = 0.42 is NOT greater than lb(1, 0.3) ≈ 0.56.
* \*AD\*: Exp(AD) = Exp(A) \* Exp(D) = 0.7 \* 0.63 = 0.441.
* Count = 2 (Present in T1 and T3).
* Since Count >= minsup(1) and Exp(AD) = 0.441 is NOT greater than lb(1, 0.3) ≈ 0.56.
* \*BC\*: Exp(BC) = Exp(B) \* Exp(C) = 0.63 \* 0.6 = 0.378.
* Count = 2 (Present in T1 and T2).
* Since Count >= minsup(1) and Exp(BC) = 0.378 is NOT greater than lb(1, 0.3) ≈ 0.56.
* \*BD\*: Exp(BD) = Exp(B) \* Exp(D) = 0.63 \* 0.63 = 0.3969.
* Count = 3 (Present in all transactions).
* Since Count >= minsup(1) and Exp(BD) = 0.3969 is NOT greater than lb(1, 0.3) ≈ 0.56.
* \*CD\*: Exp(CD) = Exp(C) \* Exp(D) = 0.6 \* 0.63 = 0.378.
* Count = 2 (Present in T1 and T2).
* Since Count >= minsup(1) and Exp(BD) = 0.378 is NOT greater than lb(1, 0.3) ≈ 0.56.
* **Pair Itemsets: ∅.**

1. **Generate and Calculate Expectation for Triplet Itemsets**:

* Combine items from C1 to form triplets: {ABC, ABD, ACD, BCD}.
* \*ABC\*: Exp(ABC) = Exp(A) \* Exp(B) \* Exp(C) = 0.7 \* 0.63 \* 0.6 = 0.265.
* Count = 1 (Present only in T1).
* Since Count >= minsup(1) and Exp(ABC) = 0. 265 is NOT greater than lb(1, 0.3) ≈ 0.56.
* \*ABD\*: Exp(ABD) = Exp(A) \* Exp(B) \* Exp(D) = 0.7 \* 0.63 \* 0.63 = 0.278.
* Count = 2 (Present in T1 and T3).
* Since Count >= minsup (1) and Exp(ABD) = 0. 278 is NOT greater than lb(1, 0.3) ≈ 0.56.
* \*ACD\*: Exp(ACD) = Exp(A) \* Exp(C) \* Exp(D) = 0.7 \* 0.6 \* 0.63 = 0. 265.
* Count = 1 (Present only in T1).
* Since Count >= minsup(1) and Exp(ACD) = 0. 265 is NOT greater than lb(1, 0.3) ≈ 0.56.
* \*BCD\*: Exp(BCD) = Exp(B) \* Exp(C) \* Exp(D) = 0.63 \* 0.6 \* 0.63 = 0.238.
* Count = 2 (Present in T1 and T2).
* Since Count >= minsup(1) and Exp(BCD) = 0.238 is NOT greater than lb(1, 0.3) ≈ 0.56.
* **Triplet Itemsets: ∅.**

1. **Generate and Calculate Expectation for Quadruplet Itemsets**:

* Since we have all four items A, B, C, and D present in T1, we can form the quadruplet {ABCD}.
* \*ABCD\*: Exp(ABCD) = Exp(A) \* Exp(B) \* Exp(C) \* Exp(D) = 0.7 \* 0.63 \* 0.6 \* 0.63 = 0.167.
* Count = 1 (Present only in T1).
* Since Count >= minsup(1) and Exp(BCD) = 0.167 is NOT greater than lb(1, 0.3) ≈ 0.56.
* **Quadruplet Itemsets: ∅.**

1. **Output**:

The selection of top-K candidates are {A, B, D, C} in ascending order, where only the top 3 are chosen due to a limit of k = 3, resulting in the top-K candidates being {A, B, D}.

The CGEBTopKFunction algorithm, Algorithm [1], is designed to select the top-K candidates based on certain criteria. It takes as input the UD (User Data), minsup (minimum support), minpro (minimum profit), and k (the limit on the number of candidates to be selected). The output is a list of the top-K candidates.

Now let's take a closer look at the pseudocode for the CGEBTopKFunction algorithm:

**Algorithm [1] CGEBTopKFunction**

**INPUT**: UD, minsup, minpro, k

**OUTPUT**: List of top-K Candidates C

**Begin**

Initialize an empty list **F**, empty set **elements** to store unique items

**elementsProbability** = elements probability from UDB

**For** each transaction in UD **do**

**For** each item in transaction **do**

Add item to elements

**End for**

**End for**

**For** each e in elements **do**

Create a new set itemSet containing only e; Add itemSet to **F**

**End for**

Initialize an empty list **results**, empty set **varList**

**While F** is not empty **do**

Initialize an empty list L

**For** each set f in **F** **do**

Initialize count = 0, Exp = 0, Var = 0, Prob = 1

**For** each item ff in f **do**

Multiply prob by the probability of ff from **elementsProbability**

**End for**

**Fo**r each transaction in D **do**

If transaction contains all items in f then

Exp += Prob; Var += Prob multiplied by (1 - prob); count += 1

**End if**

**If** count >= minsup && Exp >= lb(Exp, minsup, minpro) then

**newPattern** = new C(f, E, Var, count, Prob); Add **newPattern** to **topKPatterns**

**If** **topKPatterns** exceeds size k **then**

Remove the pattern with the lowest E from **topKPatterns**

**End if**

Add f to L; Break

**End if**

**End for**

**End for**

**F** = generateSet(L, elements)

**If** L is empty **then**

Break

**End if**

**End while**

List result = new ArrayList from topKPatterns

Sort result in descending order of E

**Return result**

**End**

**Input**:

* “***List<Set<String>> A”:*** This list contains sets of strings, where each set represents an itemset. These are most likely the current itemsets that have been found to be frequent.
* “***Set<String> B***”: This set contains unique items, possibly the individual elements that are considered for combining with the itemsets in ***“A”*** to form new candidate itemsets for the next iteration.

**Process**:

* For each itemset in ***A*** (let’s call it ***a***), the function would find items in B that can be appended to a to create a new itemset that is one item larger.
* If ***B*** contains the items ***{b1, b2, b3,...}*** and a is ***{a1, a2}***, the new generated sets might be ***{a1, a2, b1}***, ***{a1, a2, b2}***, ***{a1, a2, b3}***, and so on, provided that ***b1, b2, b3***, etc., are not already in ***a***.
* These new sets are added to the result list if they meet certain criteria, such as not being duplicates and potentially meeting other algorithm-specific conditions for candidate generation.

**Output**:

* ***“List<Set<String>> result”***: The returned list is a collection of new itemsets generated by adding each element from set **“*B”*** to the existing itemsets from list ***A*.** This list represents the next level of candidate itemsets that will be checked for frequency in the subsequent step of the mining algorithm.
* The output list ***“result”*** is then used in the next iteration of the algorithm to check which itemsets are actually frequent in the dataset.

The generateSet function is a helper function used within the CGEBTopKFunction algorithm. It takes as input a list of sets (A) and a set of elements (B), and it outputs a list of new sets.

The purpose of this function is to generate new sets by combining each set in A with each element in B. It does so by iterating over each set in A, finding the index of the last item in the set within the list of elements (bList), and then creating new sets by adding elements from bList to the existing sets.

Next, we will examine the pseudocode for the generateSet function in more detail.

**Function generateSet**

**INPUT**: List of Sets A, Set of elements B

**OUTPUT**: List of new Sets

**Begin**

Initialize an empty list result

Convert B to list bList

**For** each set a in **A** **do**

Convert a to list aList

Find index of the last item of aList in bList

**For** j from (index + 1) to size of bList **do**

Create new set newSet from a

Add element at j of bList to newSet

Add newSet to result

**End for**

**End for**

**Return result**

**End**

## PMFIs Conforming

**Input**:

* “***UD UD***”: An instance of the “***UD***” class, representing the uncertain database. This object includes the transactions in the database along with the probabilities associated with each item within these transactions.
* “***int minsup***”: The minimum support threshold. This is the minimum number of transactions in which a pattern must appear to be considered significant.
* “***double minpro***”: The minimum probability threshold. This threshold is used to filter out patterns based on their calculated probability. From this and minsup, This upper bound is likely used to filter patterns by determining a threshold value of expected support for an itemset, above which the itemset can be considered significant.
* “***int k***”: Specifies the number of top patterns to return. The function will identify the top k patterns that meet the minimum support and probability thresholds.

**Process:**

* The function ***“APFI\_MAX\_TopK”*** is an advanced version of the itemset mining function that not only looks for frequent itemsets but also aims to find the most significant ones within the parameter of ***“k”***.
* It probably incorporates both the concepts of support (how often an itemset occurs in the dataset) and the probability (how likely it is that the itemset truly is frequent in the presence of uncertain data).
* The function likely uses these inputs to generate a list of itemset candidates, evaluate their frequency and probability against ***“minsup”*** and ***“minpro”***, and then determine which ones are maximal — meaning no superset of these itemsets is equally or more frequent.

**Output**:

***“List<C>”***: A list of ***“C”*** objects, where each ***“C”*** object represents a pattern that has met the minimum support and probability thresholds. Each ***“C”*** object includes:

* The set of items in the pattern.
* The calculated exception value ***“E”*** for the pattern.
* The variance of the pattern.
* The count of transactions ***(j)*** in which the pattern appears.
* The probability of the pattern.

These ***“C”*** objects are sorted in a way that the most significant itemsets according to the algorithm's criteria are presented, up to the number ***“k”***.

**Explain Algorithm 2 (APFI MAX)**

1. **Initialization**:
   * **Fre\_Pre** is initially set to null because there are no previously found frequent itemsets.
   * **Fre\_Cur** is also set to null at the start.
2. **Processing Candidates**:
   * Since **Fre\_Pre** is null initially, all candidates from {A, D, B} can be added directly to **Fre\_Cur**.
3. **Frequency Measurement**:
   * For each itemset in **Fre\_Cur** (which now contains B, D, A), call Algorithm 3 to measure its frequency.

Algorithm 3 will check if the expected support E(X) for each itemset X is greater than or equal to the upper bound (ub) calculated.

**Input**:

* ***“int minsup”***: The minimum support threshold, which is a count of the number of transactions in which an itemset should appear to be considered frequent.
* ***“double minpro”***: The minimum probability threshold. This threshold is used to filter out patterns based on their calculated probability.
* ***“double E”***: The expected support or expectation of the itemset, which could have been calculated previously in the **algorithm 1** .

**Process**:

* The function checks if the expected support ***“E”*** is greater than or equal to the upper bound ***“ub”*** calculated from ***“minsup’*** and ***“minpro”***.
* If E is not greater than or equal to ***“ub”***, it checks whether the lower bound ***“lb”***, also calculated from ***“minsup”*** and ***“minpro”***, is less than ***“minsup”***.
* The purpose of the bounds check is to ensure that the itemset is frequent and probabilistically significant within the context of the given thresholds.

**Output**:

The function returns a boolean value:

* ***“true”*** if the itemset's expected support is greater than or equal to the upper bound or if the lower bound is less than minsup.
* ***“false”*** otherwise.

This means that the ***“FM”*** function is used as a filter to determine whether patterns meet the set thresholds and should be included in the result set for further processing or for the final output.

**Explain Algorithm 3 (FM- Frequent Measurement)**

* **Input**: Each itemset X, **minsup = 1**, **minpro = 0.3**, and the calculated upper bound of 1.94.
* **Output**: The boolean value of the frequency for X.

For each itemset X (B, D, BD):

1. Calculate E(X) based on their occurrence probabilities from the transactions (as per the details from Algorithm 1).
2. Compare E(X) to the calculated ub (1.94):
   * If E(X) ≥ ub, then the itemset X is considered frequent. No candidates generation satisfied.
   * Otherwise, refer to Eq.6 (lb < minsup) to determine the frequency of X. So It is true.

**Output:**

PMFIs Conforming: D, B, A.

The APFI\_MAX\_TopKFunction algorithm, Algorithm [2], is designed to identify the top-K candidate patterns based on certain criteria. It takes as input the UD (User Data), minsup (minimum support), minpro (minimum profit), and k (the limit on the number of candidate patterns to be selected). The output is a list of the top-K candidate patterns.

**Algorithm [2] APFI\_MAX\_TopKFunction**

**INPUT**: UD, minsup, minpro, k

**OUTPUT**: List of top-K Candidate Patterns res

**Begin**

allPatterns = CGEBFucntion(UD, minsup, minpro, k)

Initialize an empty list **Fre\_Cur**, empty list **Fre\_Pre**, empty list **res**

**For** each pattern in allPatterns **do**

Get the itemset X from pattern; isSuperset = False

**For** each itemset pre in **Fre\_Pre** **do**

**If** X is a superset of pre **then**

isSuperset = True; break

**End if**

**End for**

**If** isSuperset == True **then**

Add X to **Fre\_Cur**

**End if**

**If** FM(minsup, minpro, pattern.getE()) == True **then**

Add pattern to **res**

**If** size(**res**) > k **then**

sort(res) descending order by getE

Remove the last item from res

**End if**

Add X to **Fre\_Cur**

**End if**

**Fre\_Pre =** new list containing items from **Fre\_Cur;** **Fre\_Cur** = NULL

**End for**

Sort res in ascending order by getE

**If** size(**res**) > k **then**

Set res to the first k items of res

**End if**

Return **res**

**End**

The Frequency Measurement algorithm, Algorithm [3], is designed to determine whether a given pattern meets certain criteria based on its support and profit values. It takes as input the minimum support (minsup), minimum profit (minpro), and the frequency of the pattern (Exp). The output is a boolean value indicating whether the pattern meets the criteria.

**Algorithm [3] Frequency Measurement**

**INPUT**: minsup, minpro, Exp

**OUTPUT**: Boolean indicating whether the pattern meets the criteria

**Begin**

**If** E >= ub(minsup, minpro) **then**

Return True

**Else** **if** lb(minsup, minpro) < minsup **then**

Return True

**Else**

Return False

**End if**

**End**

# Solution

## Environment and Data description

**Programming language**: Java

**Approaches:**

* 1. Java Object
  2. Using Java Collection Framework (Java Eclipse Primitive Collections)

**Input-Dataset:**

The input file is a text file with a .txt extension. The elements within the file are separated by tabs. Each line of the file contains a series of items followed by an element that determines the probability. The final element on each line can fall into one of two cases:

* If it is a number, it is used directly as the probability (prob) associated with the preceding items in the line.
* If it is a text string, each item within that line is assigned a probability that is randomly generated.

This file format might typically be used in applications such as probabilistic databases, where transactions are not certain, and therefore, items are associated with probabilities to represent the likelihood of their occurrence.

**Performance evaluation**

Time: In performance evaluation, time refers to the amount of time it takes for a program or process to execute or complete a task. It is typically measured in seconds, milliseconds, or other units of time. The time metric helps evaluate the speed or efficiency of the program, with lower time values indicating better performance.

Memory: Memory, in performance evaluation, refers to the amount of computer memory or RAM (Random Access Memory) that a program or process requires to run. It is typically measured in kilobytes (KB) or megabytes (MB). The memory metric helps evaluate the resource usage and efficiency of the program, with lower memory values indicating more efficient memory utilization.

When evaluating the performance of different approaches, such as the Java Object approach and the Maven (Java Eclipse primitive collections) approach, time and memory are essential metrics. By measuring the time and memory consumption of these approaches, you can compare their efficiency and resource requirements.

Based on the main ideas presented above, combined with 1 algorithm (pseudocode), we will have 2 approaches to implement the algorithms as follows: 1. Using Java Object, 2. Using Java Collection Framework (called Java Eclipse Primitives). The details of the implementation and the reasons for using each approach will be further elaborated below.

## Implementation using Java Object

In this approach, we will implement the algorithms using Java objects, leveraging the object-oriented programming paradigm. Each entity in the problem domain will be represented as an object, and the algorithms will manipulate these objects to achieve the desired outcomes.

We used object data types (Integer, Double, Boolean, etc.). Using these data types would require boxing and unboxing operations (boxing is the process of converting a primitive type (int, double, boolean) to its corresponding object wrapper class (Integer, Double, Boolean), and unboxing is the reverse process). This leads to additional runtime costs due to object creation and destruction during each operation. While the overhead may be negligible for small amounts of data, it can become substantial for large data sets that are frequently updated. Therefore, by using primitive types where possible to avoid boxing and unboxing, the application runtime performance can be significantly improved.

**Reasons for using.**

Clear abstraction: Objects encapsulate both data and behavior, providing a clear and intuitive representation of entities in the problem domain.

Code organization: Object-oriented design promotes modular and organized code, making it easier to maintain and extend the implementation.

Reusability: Objects can be reused across different parts of the codebase, promoting code reuse and reducing redundancy.

Flexibility: Object-oriented design allows for flexible and scalable solutions, accommodating changes and additions to the codebase with minimal impact.

**Class diagram**

Next, let us move on to classdiagram figure 5.1 for the first approach using Java objects. Figure 5.1 shows the class diagram for the object-oriented approach.

**Class Item:**

Function: Stores a set of elements and the corresponding probability.

Methods:

* Item(item): Initializes an Item object with a list of elements and a random probability.
* getItem(): Returns the set of elements in the Item object.

**Class UD:**

Function: Stores the uncertain database and handles related methods.

Methods:

* getProbability(): Returns a list of probabilities of elements in the uncertain database.
* removeProbFromUD(): Removes probability from the uncertain database and returns a list of unique data sets.

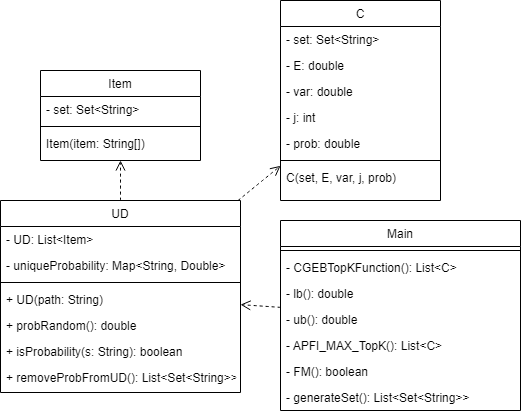


Figure 5‑1 Class diagram using Java Object

**Class C:**

Function: Stores information about a candidate and related attributes.

Attributes:

* set: Set of elements in the candidate.
* E: Expected value.
* Var: Variance.
* j: Threshold value.
* prob: Probability.

Methods:

* getSet(): Returns the set of elements in the candidate.
* getE(): Returns the expected value.
* getVar(): Returns the variance.
* getJ(): Returns the threshold value.
* getProb(): Returns the probability.

**Class Main:**

Function: Contains the main() method to start the program execution.

Methods:

* CGEBTopKFunction: It returns a list of C objects representing the top-k patterns. ***(Using Algorithm [1])***
* APFI\_MAX\_TopK: It returns a list of C objects representing the filtered top-k patterns. ***(Using Algorithm [2])***
* lb: This method calculates the lower bound value based on the given minsup and minpro parameters. It returns a double value.
* ub: This method calculates the upper bound value based on the given minsup and minpro parameters. It returns a double value.
* generateSet: Set Union Operation.
* *FM:* ***(Using Algorithm [3])***

## Implementation using Java Eclipse Primitives

In contrast to the object-oriented approach, this approach involves implementing the algorithms using Java Eclipse Primitives, which are low-level data types provided by the Eclipse IDE (Java Collection Framework). These primitives offer a more direct and efficient way of handling data compared to objects.

Primitive types allow operations to be performed directly on the underlying values without implicit object instantiation during boxing/unboxing. For throughput-sensitive systems handling vast amounts of numeric data, primitive collections can provide a notable performance boost compared to their object-oriented equivalents. This optimization is especially important for applications that perform frequent insertions, deletions or updates on large datasets.

**Reasons for using**

The main reason: For tasks requiring high performance or dealing with large datasets, using primitives can lead to faster execution times and reduced memory overhead. By bypassing the overhead associated with objects, primitives allow for more streamlined and optimized code.

Performance: Primitives offer better performance compared to objects, especially for tasks involving intensive computation or large datasets.

Memory efficiency: Primitives consume less memory compared to objects, making them suitable for memory-constrained environments or applications dealing with large volumes of data.

Low-level control: Using primitives provides direct control over memory allocation and manipulation, allowing for fine-tuning and optimization of code.

Compatibility: Eclipse Primitives are compatible with other low-level libraries and frameworks, facilitating integration with existing codebases or libraries optimized for primitive data types.

**Class diagram**

Continuing with the classdiagram section, we will now move to classdiagram figure 5.2 which uses the remaining approach with the Java Eclipse framework. Figure 5.2 displays the class diagram for the framework-based approach.

In terms of basic structure, it is nearly identical to the class diagram of the Java object method; however, the data types used have been modified to suit this approach. And the functionality of the methods remains unchanged.

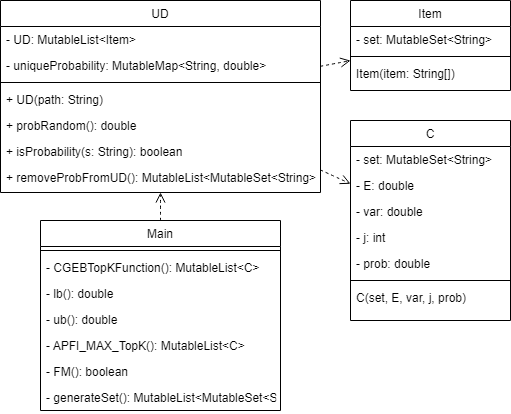


Figure 5‑2 Class diagram using Java Eclipse Primitives:

## User manual

**1. Using Java Object**

Requirement: *Java version 17*

**Step 1:** Compile the Java code. Run the command: **javac Main.java**

**Step 2:** Run the program with parameters. Run the command: **java Main**

**The options are:**

**-f** or **--file** **<filename>**

*Required. Specify the input database filename.*

**-ms** or **--minsup <value>**

*Specify the minimum support threshold. An integer is required.*

**-mp** or **--minpro <value>**

*Specify the minimum probability threshold. A double is required.*

**-k** or **--top-k <value>**

*Specify the number of top patterns to return. An integer is required.*

**-h** or **--help**

*Display help information.*

**-v** or **--version**

*Display version information.*

*Example:* **java Main -f transactions.txt -ms 1000 -mp 0.6 -k 20**

**Step 3:** The program will validate all the options. If any option is invalid or missing a required value, an error will be printed.

**Step 4:** If all options are valid, the results will also be saved to a file named **result\_filename\_timestamp.txt** in the same directory as the java code.

**2. Using Java Eclipse Primitive**

Requirement: *Java version 17, Maven 3.2.3*

**Step 1:** Navigate to the project directory containing the **pom.xml** file.

**Step 2:** Build the project by running **mvn clean install** will compile the code.

**Step 3:** Run the application passing in the required arguments.

Run **mvn compile exec:java -Dexec.mainClass="mining.frequentitemsets.Main" -Dexec.args="-f file\_name.txt -ms 1000 -mp 0.6 -k 100"**

*This will:* Compile the code, Run the Main class, Pass in the database file name, min support, min prob, and top k as arguments.

**Step 4:** Check the console output or results file. The top k patterns will be output to the console Results will also be saved to **result\_file\_name\_timestamp.txt.**

# Experiment

## Setup

**Datasets**:

For the Chess dataset, setting the minsup at 3,000 and minpro (minimum confidence) at 0.6 is quite aggressive given the total transaction count of 3,196. Such high thresholds are likely to significantly reduce the number of frequent itemsets and association rules generated, leading to potentially lower memory usage and faster execution times. This setup will test the Java environment's ability to handle high-threshold scenarios efficiently, where the focus is on extracting only the most significant patterns.

Moving to the Mushroom dataset, the minsup is adjusted to 4,000, with the minpro remaining at 0.6. Despite the Mushroom dataset having more transactions (8,124) than the Chess dataset, the increased minsup value ensures that only the most prevalent patterns are considered. This threshold will likely challenge the Java environment's performance, particularly in terms of memory management, as the dataset's larger size and complexity combined with high minsup could lead to more significant memory allocation and usage.

For the T10I4D100K dataset, the evaluation becomes even more rigorous with a minsup of 6,000 and a minpro of 0.6. Given the incremental approach to testing this dataset—from 20,000 to 100,000 transactions—this high minsup value will test the scalability of the Java environment in Visual Studio Code. It's crucial to observe how memory usage and execution time evolve as the transaction count increases. With such a high minsup, the environment's ability to manage memory efficiently and process transactions quickly is put to the test, especially as the dataset size approaches the upper limit of 100,000 transactions.

Table 6‑1 Introduction Dataset

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset name** | **Number  of Transaction** | **minsup** | **minpro** |
| Chess | 3,196 | 3000 | 0,6 |
| Mushroom | 8,124 | 4000 | 0,6 |
| T10I4D20K | 20,000 | 6000 | 0,6 |
| T10I4D40K | 40,000 | 6000 | 0,6 |
| T10I4D60K | 60,000 | 6000 | 0,6 |
| T10I4D80K | 80,000 | 6000 | 0,6 |
| T10I4D100K | 100,000 | 6000 | 0,6 |

***Note:*** The T10I4D100K.txt file contains a dataset consisting of a list of Trans, each Trans containing a sequence of non-negative integers separated by spaces. The file consists of a total of 100,000 Trans. Each Trans represents a set of non-negative integers, with each number representing an element in the set. The numbers on the same line are sorted in ascending order. This file serves as a sample dataset for data mining, algorithm research, or tasks related to handling large numerical data.

**Computer configuration used for experiments**:

* + CPU: Intel Core i5-5200U
  + Memory: 8 GB RAM
  + Operating system: Windows 10 Home
  + Storage: 512 GB SSD

## Results

**Performance evaluation**

The experimental results obtained from this study reveal the performance differences between running a Java object and a Java Eclipse primitive on different datasets. Table 4 presents the time and memory usage for running a Java object, while Table 5 displays the corresponding results for running a Java Eclipse primitive. These results provide valuable insights into the efficiency and resource consumption of these approaches on various datasets.

Table 6‑2 Time and Memory running with a Java Object

|  |  |  |
| --- | --- | --- |
| **Dataset name** | **Time** | **Memory** |
| Chess | 0,26s | 47,404 kb |
| Mushroom | 0,53s | 29,939 kb |
| T10I4D20K | 3.00s | 39,247 kb |
| T10I4D40K | 8.00s | 21,1043 kb |
| T10I4D60K | 9.00s | 103,021 kb |
| T10I4D80K | 11.00s | 122,981 kb |
| T10I4D100K | 17.00s | 170,425 kb |

Table 6‑3 Time and Memory running with a Java Eclipse Primitive

|  |  |  |
| --- | --- | --- |
| **Dataset name** | **Time** | **Memory** |
| Chess | 1,29s | 33,895 kb |
| Mushroom | 2,10s | 26,698 kb |
| T10I4D20K | 3,00s | 17,304 kb |
| T10I4D40K | 7,00s | 65,263 kb |
| T10I4D60K | 6,00s | 93,051 kb |
| T10I4D80K | 11,00s | 103,287 kb |
| T10I4D100K | 10,00s | 138,618 kb |

Figure 6-1 compares the running times of operations using Java Object and Java Eclipse Primitive with two different datasets: chess and mushroom. For the chess dataset, the running time using Java Object is lower than using Java Eclipse Primitive. However, for the mushroom dataset, both Java Object and Java Eclipse Primitive have significantly higher running times, with Java Eclipse Primitive showing a particularly large increase.

Figure 6-2 compares the memory of operations using Java Object and Java Eclipse Primitive with two different datasets: chess and mushroom. For the 'chess' dataset, the memory usage is higher with the Java Object method than with the Java Eclipse Primitive method. For the 'mushroom' dataset, the memory usage is very similar for both Java Object and Java Eclipse Primitive, with Java Object consuming slightly more.

Figure 6‑1 Compare time between Java Object and Java Eclipse Primitive

Figure 6‑2 Compare memory between Java Object and Java Eclipse Primitive

Figure 6‑3 Compare time between Java Object and Java Eclipse Primitive

Figure 6‑4 Compare memory between Java Object and Java Eclipse Primitive

1. Figure 6-3 also represents "Topk APFI-MAX" but splits the data into two series: "Java Object" and "Java Eclipse Primitive”.
2. In the graph, both "Java Object" and "Java Eclipse Primitive" show an increase in running time with data size, but the scale is different, with a maximum running time of about 17,000 ms for the "Java Object" at 100k size. While “Java Eclipse Primitive” only takes about 10,000 ms at 100k size
3. "Java Eclipse Primitive" is more memory efficient as it has lower memory usage compared to "Java Object."

## Discussion

In theoretical terms, when utilizing Java Eclipse primitives, we expect to achieve faster processing speeds compared to using Java objects. Following experimental verification, it can be concluded that as the dataset size increases, the processing speed (application runtime) significantly decreases. However, for smaller datasets (~1000 - 3000 transactions), the observed variance in processing speed between primitive and object approaches is negligible.

# Conclusion

The paper presents a method called Topk APFI-MAX for mining frequent itemsets from uncertain databases to ultimately minimize computational complexity. Specifically, Topk APFI-MAX utilizes a lower bound constraint on the expected support of candidate PFMIs to reduce the candidate set generation.

Then, an estimation method is proposed to measure the frequencies of candidates instead of accurately computing the probability mass function, thus reducing the computational complexity from O(nlogn) to O(n).

In summary, the paper proposes a method for mining frequent itemsets that is optimized for time complexity and suitable for large databases. This method promises applications in many areas such as environmental monitoring, healthcare, e-commerce, etc. where efficient analysis of large uncertain datasets is needed. By reducing the candidate generation and support estimation steps, Topk APFI-MAX can extract frequent patterns from massive uncertain data in a scalable manner. The lower bound pruning and frequency [15, 16, 17, 12] estimation techniques work together to improve runtime performance for uncertain frequent pattern mining.

# Data availability

The dataset and code used in this study are publicly available for replication and further research.

**Dataset**

The dataset utilized in this study comprises 4 datasets: T10I4D100K, Chess, Mushroom, Connect. The dataset can be accessed from the following repository: [<https://github.com/nguyenhuy158/mining-probabilistic-maximal-high-utility-frequent-itemsets>].

**Code Demo**

The code used for analysis and modeling in this research is available in a public GitHub repository: [<https://github.com/nguyenhuy158/mining-probabilistic-maximal-high-utility-frequent-itemsets>]. This repository includes scripts for data preprocessing, algorithms, and evaluation. Additionally, detailed instructions for running the code are provided in the repository's README file.

# References

|  |  |
| --- | --- |
| [1] | S. Chen, L. Nie, X. Tao, Z. Li and L. Zhao, "Approximation of Probabilistic Maximal Frequent Itemset Mining Over Uncertain Sensed Data," IEEE Access, 2020. |
| [2] | Z. Zhao, D. Yan and W. Ng, "Mining Probabilistically Frequent Sequential Patterns in Large Uncertain Databases," 2014. |
| [3] | Y. Li, J. Bailey, L. Kulik and J. Pei, "Mining Probabilistic Frequent Spatio-Temporal Sequential Patterns with Gap Constraints from Uncertain Databases," 2013. |
| [4] | C.-K. Chui, B. Kao and E. Hung, "Mining frequent itemsets from uncertain data," 2007. |
| [5] | L. Wang, R. Cheng, S. D. Lee and D. W. Cheung, "Accelerating probabilistic frequent itemset mining: A model-based approach," 2010. |
| [6] | L. K.-S. Carson, M. Mark Anthony F. and B. Dale A., "A tree-based approach for frequent pattern mining from uncertain data," 2008. |
| [7] | O. Pivert and H. Prade, "A Certainty-Based Model for Uncertain Databases," 2015. |
| [8] | C. K.-S. Leung and S. K. Tanbeer, "Fast tree-based mining of frequent itemsets from uncertain data," 2012. |
| [9] | J. M.-T. Wu, S. Liu and J. C.-W. Lin, "Mining of High-Utility Sequence Patterns in Large-Scale Uncertain Databases," 2022. |
| [10] | L. Nie, Z. Li, H. Qi, W. Liu and W. Qu, "Probabilistic Frequent Itemsets Mining Based on Expectation Bound over Uncertain Database," 2017. |
| [11] | W. Xue, Q. Luo, L. Chen and Y. Liu, "Contour map matching for event detection in sensor networks," 2006. |
| [12] | H. Li, Y. Wang, N. Zhang and Y. Zhang, "Fuzzy maximal frequent itemset mining over quantitative databases," 2017. |
| [13] | L. Tuong, V. Bay, N. H. Van, T. N. Ngoc and W. B. Sung, "Mining top-k frequent patterns from uncertain databases," 2020. |
| [14] | Y. K. F. Andrew, P. B. Wessel and R. B. David, "A Note on the Chernoff Bound for Random Variables in the Unit Interval," 2022. |
| [15] | C. K.-S. Leung and R. K. MacKinnon, "Blimp: A compact tree structure for uncertain frequent pattern mining," 2014. |
| [16] | K.-S. L. Carson and A. B. Dale, "Efficient algorithms for the mining of constrained frequent patterns from uncertain data," 2010. |
| [17] | W. Gan, J. C.-W. Lin, P. Fournier-Viger, H.-C. Chao and V. S. Tseng, "Mining high-utility itemsets with both positive and negative unit profits from uncertain databases," 2017. |
| [18] | A. Asif, T. Shahnawaz and N. Sanam, "Detecting Faulty Sensors by Analyzing the Uncertain Data Using Probabilistic Database," 2020. |
| [19] | S. Chen, L. Nie, X. Tao and Z. Li, "Frequent itemset mining-based spatial subclustering algorithm," 2015. |
| [20] | W. Qian, H. Li, H. Huang, M. Yuan, Y. Xu and G. Sun, "Fast rare itemset mining in uncertain database," 2023. |
| [21] | J. Quan, Z. Liu, D. Chen and H. Zhao, "High-Efficiency Algorithm for Mining Maximal Frequent Item Sets Based on Matrix," 2012. |
| [22] | M. Alharbi, P. Periaswamy and S. Rajasekaran, "Disjunctive rules mining from uncertain databases," 2014. |
| [23] | Y. Lai and J. Xie, "Frequent itemset based event detection in uncertain sensor networks," 2013. |