DISCOVERY AND EFFECTIVE USE OF FREQUENT ITEM-SET MINING AND ASSOCIATION RULES IN DATASETS

NIMA SHAHBAZI

A DISSERTATION SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTER SCIENCE AND ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO

NOVEMBER 2018

Abstract

The unprecedented rise in digitized data generation has led to the ever-expanding demand for sophisticated storage and analysis methods capable of handling vast amounts of complex data, much of which is stored within many databases. Owing to the large size of such databases, employment of sophisticated analysis methods, such as data mining and machine learning, becomes necessary to extract useful insights regarding a given system under study. Frequent itemset mining and association rules mining represent two key approaches to mining knowledge stored in databases. However, handling of large databases often leads to time-consuming calculations that necessitate large amounts of memory. In this regard, the development of methods capable of enabling faster, less laborious search or pattern discovery remains a central focus in the field of data mining. Incontestably, such methods could aid in faster processing and knowledge extraction, enabling new breakthroughs in how knowledge is acquired from data and applied in real-world applications. However, real-world applications are often hindered by limitations inherent to currently available algorithms. For instance, many itemset mining algorithms are known to first store a given database as a tree structure in memory. However, such algorithms fail to provide a tight upper bound on the number of nodes that will be generated during the tree building process – accordingly, there are no upper bounds governing the amount of memory that is needed to generate such trees. As such, practical implementation of frequent itemset

mining algorithms is often restricted by memory consumption. However, despite the importance of memory consumption in the applicability of itemset mining, this factor has not drawn adequate attention from the data mining community and remains as a key challenge in its application. In addition, the majority of algorithms widely used and studied to date are known to require multiple database scans, a factor which restricts their applicability for incremental mining applications. In this regard, the development of an algorithm capable of dynamically mining frequent patterns "on-the-fly" would open new pathways in data mining, enabling the application of itemset mining methods to new real-world applications, in addition to vastly improving current applications.

In this thesis, different approaches are proposed in relation to the above-mentioned limitations currently hampering further progress in this significant area of data mining. First, an upper bound on the number of nodes of well-known tree structures in frequent itemset mining is presented. Second, aiming to overcome the memory consumption constraint, a memory-efficient method to store data processed by the frequent itemset mining algorithm is proposed, where instead of a tree, data is stored in a compact directed graph whose nodes represent items. Third, an algorithm is proposed to overcome costly databases scans in the form of a novel SPFP-tree (single pass frequent pattern tree) algorithm. Lastly, approaches that allow for frequent itemset and association rules to be practically and effectively used in real world applications are proposed. First, the quality and effectiveness of frequent itemset mining in solving a real world "facility management problem" is examined. Second, with aims of improving the quality of recommendations made to users, as well as to overcome the cold-start problem suffered by new users, a hybrid approach is herein proposed for the application of association rules into recommender systems.

Acknowledgements

I would like to take this opportunity to convey my sincerest gratitude to the people who have lent me their support throughout the duration of this thesis, as well as those who have contributed to set me on this path with their support and guidance throughout all my years.

Firstly, to my advisor, Prof. Jarek Gryz, for agreeing to supervise me, for believing in me, and for his tireless assistance throughout this journey. In addition to constantly motivating me to explore new limits, his guidance was invaluable throughout both the research and writing phases of this thesis. Among his many admirable traits, his patience, motivation, and vast knowledge are an inspiration to current and aspiring researchers in this field.

To Prof. Aijun An, whose data mining class brought me the idea for my first research paper. She also provided me with the opportunity to join the BRAIN team and to make use of their research facilities. Without her precious support I could not have conducted this research.

Prof. Parke Godfrey, for his insightful comments and encouragement, as well as for the questions he posed throughout this process, which allowed me to achieve new breakthroughs by contemplating broached

topics from various perspectives. His extensive knowledge in databases and data mining enabled me to tackle said challenges from various directions.

My lovely parents and my two sisters (Nasim and Ghazal), real-life heroes in my life story. These lovely humans never ever let me settle for less than what I could achieve during my younger years, and have always pushed me to new limits by believing in my abilities. Their love and affection have always strengthened me, as shown through their unwavering spiritual support, their ongoing eager interest in my work, and their readiness to always stand by me throughout difficult times. Thank you for being there for your son and your little brother.

One of my best friends, Masoud Memarzadeh, from whom I learned so much. He has taught me not only math, but so much about life. His sharp solutions to various problems have always motivated me to think outside the box, which has helped me understand and formalize problems in formal forms.

Naresh Bangia and Ajay Agrawal, whom I met during my internship at the NextAI and Creative Destruction lab. Their willingness to take on a new guy and show him the ropes undoubtedly helped me better understand and convert various theories and academic concepts into applicable business ideas. I really cherish my experience in CDL and NextAI.

Last but not least, I would like to thank my lovely wife, Arezoo, for her support during the entire length of this doctorate. She has always motivated me to achieve this dream, and has stood by my side during the entire time I struggled through this process, ensuring I never lost sight of my objectives. Thank you Arezoo, I have reached the finish line thanks to your relentless support and devotion, and I couldnt be more grateful for all you have done to help me get here.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 MOTIVATIONS AND OBJECTIVES

Frequent itemset mining is an important task in data mining and has been deployed in a wide range of applications, including bio-informatics [70], web mining [43], software bug mining [48], system caching [72], among numerous others. Indeed, frequent itemset mining has been proven to be a successful technique for extracting useful information from large datasets, spurring the development of numerous algorithms to date with relevance in various application domains, including market analysis, inventory control, and many others [79].

Association rules mining is a leading data mining technology that is employed in variety of fields to find associations or relations between data items, as well as uncover key attributes of large data sets. Briefly, the process enables the discovery of patterns, associations or correlations, as well as the identification of relationships among patterns themselves with minimal human effort needed, thus easily unearthing valuable

information for use that would otherwise not be easily available due to the large size and complexity of modern day databases. The vast applicability of the technique is undoubtedly confirmed by the number of algorithms and models developed to date for a wide range of application domains, including telecommunication networks, market analysis, risk management, and inventory control, among others.

Given a data set of transactions (each containing a set of items), frequent itemset mining finds all the sets of items that satisfy the *minimum support*, a parameter provided by a user or an application. The value of that parameter determines the number of itemsets discovered by the mining algorithm. If a low minimum support threshold is chosen for a database, then a large number of frequent itemsets may be found. On the other hand, a high minimum support threshold may exclude the presence of interesting, potentially useful itemsets from the search results. Thus, the mining process usually needs to be run multiple times before a satisfactory result can be achieved. In either case, the size of the data structure used by the mining algorithms to store temporary results can vary widely and is difficult to predict in advance.

When dealing with large databases, the representation and storage of the data become critical factors in the processing time of the mining algorithms. Therefore, successful application of a frequent itemset mining algorithm for solving real world problems is often restricted by memory/CPU consumption. However, despite the importance of memory consumption in the applicability of itemset mining, this factor has not drawn adequate attention from the data mining community, remaining as a key challenge in its applications.

The majority of algorithms widely used and studied in frequent itemset mining to date are known to require multiple database scans, a factor which restricts their applicability for incremental mining applications. In this regard, the development of an algorithm capable of dynamically mining frequent patterns "on-the-fly" would open new pathways in data mining, enabling the application of itemset mining methods to new real-world applications, in addition to vastly improving current applications.

In order to mine special patterns in the data, many itemset mining algorithms first store the database as a tree structure in memory. However, there is no tight upper bound for the number of nodes and

the memory requirements for these trees. Many of these algorithms are well suited for *interactive mining* where the database remains unchanged and only the minimum support threshold gets changed. Hence, such algorithms work well in situations that follow the *"build once, mine many"* principle; however, they are inefficient in situations that require incremental mining (where the database is changed frequently) [45].

The mining of association rules represents yet another significant challenge in the field. The development of mining association rules, whereby relations between distinct items in a dataset are revealed, has been historically motivated by the potential benefits associated with analysis of supermarket transaction data, as mining of such data allows for a more in-depth understanding of customer behavior regarding product purchase. Chiefly, association rules mining aims at describing existent relations among distinct items with respect to their selection by customers; particularly, how, why, or how often certain items are selected together. For instance, the association rule bread cheese (40%) describes that two out of five customers who bought bread also purchased cheese. Undoubtedly, such rules aid in revealing existing market needs and interpreting customer behavior, consequently helping inform decisions regarding product pricing, store layout, promotions, purchasing logistics, etc. Given its vast applicability, much of the research in this area has been successfully applied to real world scenarios. However, in order to enhance the usability of the data that is attained, improvements must be made to the technology. New advances in the field of association rule mining would undoubtedly result in the increased applicability of the method, enabling more effective usage of data and enhanced benefits to businesses and costumers alike. Given the ever increasing demand for methods that enable sophisticated insight into patterns of customer behavior as well as the ever growing digitalization of data, the development of novel techniques to discover and mine high quality association rules from datasets demands significant attention.

However, the successful application of extracted rules towards solving real world problems is very often restricted by the quality of the rules. In this respect, research involving the effective application of association rules to real world applications (e.g, recommender systems) has proliferated in recent years. Indeed, since its introduction by Agrawal et al. [2] in 1993, the association rule principle has received a great deal of attention

from the scientific and technological communities, resulting in an abundance of publications focusing on new or enhanced algorithms to solve more efficiently such mining problems.

1.2 RESEARCH CONTRIBUTION

This thesis contributes to the data mining field in the areas of frequent itemset mining and association rules mining for recommender systems. Contributions to the field of itemset mining include the development and subsequent real-world application of an algorithm with demonstrated suitability for incremental mining applications, a function that is made possible through decreased memory consumption and the single database scanning modality of the algorithm. Further, the following work shifts the paradigm in the field of association rules mining for recommender systems by introducing the feasible use of association rules mining in recommender systems in the form of an award-winning hybrid system.

Two main challenges associated with frequent itemset mining algorithms concern *costly databases scans* and *large memory consumption*.

Within the field of frequent itemset mining, most of the algorithms widely used and studied to date require two database scans, and thus cannot be used for streaming data applications. Further, most of these algorithms are designed for static datasets, where input transactions are fixed. Thus, incremental mining algorithms are not easily adoptable for on-the-fly, fast, and memory efficient mining. In order to overcome costly databases scans, a novel algorithm based on *SPFP-tree* (single pass frequent pattern tree) is proposed in this thesis. The proposed algorithm allows for a *single scan* of the database, to mine efficient frequent itemsets by dynamically changing the tree structure to create a highly compact tree on-the-fly.

When dealing with large databases, the representation and storage of the data becomes a critical factor in the processing time of a given mining algorithm. Existing techniques deploy either list-based or tree-based structures to store data. The problem with both structures, however, is that in cases where a large number

of itemsets needs to be processed by the algorithm, the application may run out of memory. In this sense, the development of methods or structures that could lead to more efficient memory consumption have significant value. Here, a *memory-efficient method* to store data processed by the frequent itemset mining algorithm is proposed in this thesis: instead of using a tree, the data is stored in a compact directed graph whose nodes represent items. In this way, the size of the graph is bounded by the number of distinct items present in the database.

Two efficient tree structures, alphabetical- and FP-trees, are used to store itemsets in memory for mining of special patterns. Here, an upper bound is provided for the maximum number of nodes in FP and Alphabetical trees. The *upper bound* on the number of nodes is provided in the context of a *greedy algorithm* for the alphabetical tree structure, while a *closed form* solution for the FP-tree is derived.

Two key methods, namely association rules and recommendation systems, can be scrutinized in relation to their role in data mining applications. Given that both association rules and recommendation systems aim at identifying items that appear together frequently, with the latter particularly targeted at providing recommendations to the public, it should therefore be possible to couple these two techniques together to create a more powerful hybrid system that relies on the strengths of each technique. Therefore, with aims of improving the quality of recommendations made to users, as well as to overcome the cold-start problem suffered by new users, a *hybrid approach* is herein proposed for the application of association rules into recommender systems. The herein described hybrid approach won the ACM WSDM18 Recommendation challenge. The herein presented approach is used to feed association rules extracted from a database into a recommender system. Briefly, the information attained is used to improve the quality of recommendations being made by the system to users, as well as to address the cold-start problem often encountered by recommendation systems when new users are added to the database. The theoretical principles and formulas developed in this work can also be adapted to a wide range of real-word recommender systems, thus extending their usability beyond their theoretical value by enabling practical use of the proposed approaches.

Finally, the quality and effectiveness of frequent itemset mining in solving a real world "facility management problem" are also examined as part of the current work. Here, the main objective of the task is to classify complaints or Work Orders (WOs) that describe issues requesting specific action. Automatic text classification applies machine learning techniques such as Random Forests, Bayesian networks, and support vector machines (SVM) to train an algorithm to extract features from a set of pre-labeled text documents in order to classify new documents based on their contents. Random Forests have demonstrated increased performance due to reduced model variance without increasing bias. It has been shown that randomization can de-correlate the trees in the ensemble, resulting in a highly effective classification method. Given that the important features extracted by these algorithms are mainly words that appear together, it should therefore be possible to address this task via application of frequent itemset algorithms to create a powerful yet simple learning algorithm. The algorithm presented here not only predict classes with more than 90% accuracy, but the running time is twenty times faster than random forest and memory usage is also one third of random forest.

1.3 THESIS ORGANIZATION

In Chapter 2, an in-depth review of the associated literature is presented to include works that broach topics and areas relevant to the herein presented body of work, with particular focus given to frequent itemset and association rule mining algorithms. Given the extensive work carried out in this area to date, the presented review aimed at avoiding duplication of existing works, as well as at providing an overview of the main challenges currently being faced by the frequent itemset mining community.

Chapter 3 (from the published "Building FP-Tree on the Fly: Single-Pass Frequent Itemset Mining." paper), introduces a single pass frequent itemset mining algorithm suitable for data streams and interactive/incremental mining.

Chapter 4 (from the published "Memory Efficient Frequent Itemset Mining." paper), proposes a memory-

efficient algorithm for frequent itemset mining applications. Here, the proposed data structure is described and key factors regarding the efficiency of the algorithm for frequent itemset mining are discussed.

Chapter 5 details the development of an upper bound for the number of nodes in well-known tree structure representations for frequent itemset mining algorithms, where the upper bound is provided in the context of a greed algorithm and a close form solution.

Chapter 6 (from the published "WSDM Cup 2018: Truncated SVD-based Feature Engineering for Music Recommendation" paper), in turn, provides a proof-of-concept practical application of association rules to solve a significant challenge faced by recommender systems, namely the cold-start problem. In this work, association rules extracted from a database are used in conjunction with other well-known methods to enhance the overall quality of recommendations proposed by the recommender system, while simultaneously addressing the cold-start problem typically encountered when new user profiles with insufficient information are introduced into the database.

Chapter 7 (from the "Machine learning and BIM visualization for maintenance issue classification and enhanced data collection." paper) presents a proposal to improve classification accuracy with respect to Work Orders in facility management through employment of the frequent itemset approach FIA.

Finally, Chapter 8 offers conclusions regarding the proposed body of work and recommendations for future research, including possible development activities to further enhance the usability of the presented techniques.

Chapter 2

Background and Literature Review

The purpose of this chapter is to present an in-depth review of the topics, areas and works related to the research presented here. Firstly, definitions and problem statements for frequent itemset and association rule mining are presented. A brief but comprehensive in depth review of frequent itemset and association rule mining algorithms (broad survey on prevalent techniques introduced in the past few years to address this problem) is made.

2.1 PROBLEM STATEMENT

Definition 2.1. *Let $A = \{a_1, a_2, ..., a_n\}$ be a set of literals, called items. Set $P = \{a_{i_1}, \ldots, a_{i_k}\} \subseteq A$, where $i_k \in [1, n]$, is called a* pattern *(or more specifically an itemset or a k-itemset if it contains k items). A transaction $T = (tid, P)$ or $T_{tid} = P$ is a tuple where tid is a transaction-id and P is a* pattern *or itemset. A transaction database $\mathcal{D}$ is a set of transactions T. A transaction $T = (tid, P)$ is said to contain (or support) an itemset X, if $X \subseteq P$.*

We put a total order $<_x$ *on the items in* A*, where* x *represent item ordering. A* pattern $P = \{a_{i_1}, \ldots, a_{i_k}\}$ *is said to be ordered if we change it to sequence* $\mathcal{P} = \langle a_{i_1}, \ldots, a_{i_k} \rangle$*, and* $\forall j \in [1, k-1],\ a_{i_j} <_x a_{i_{j+1}}$*. Given two ordered* patterns*,* $\mathcal{P}_1 = \langle a_{i_1}, \ldots, a_{i_m} \rangle$ *and* $\mathcal{P}_2 = \langle a_{j_1}, \ldots, a_{j_k} \rangle$*,* $m \leq k$*, if* $\forall s \in [1, m], a_{i_s} = a_{j_s}$*, then* $\mathcal{P}_1$ *is called a prefix of* $\mathcal{P}_2$*. An* ordered transaction *is a transaction with an ordered* pattern *or* $\mathcal{T}_{tid} = \langle a_{i_1}, \ldots, a_{i_k} \rangle$*, and* $\forall j \in [1, k-1], a_{i_j} <_x a_{i_{j+1}}, \{a_{i_1}, \ldots, a_{i_k}\} \subseteq A$*. Let* $|\mathcal{T}_{tid}| = |\langle a_{i_1}, \ldots, a_{i_k} \rangle| = k$*.*

Definition 2.2. *The cover of an itemset* X *consists of the set of transaction identifiers of transactions that contain* X*:*

$$cover(X, \mathcal{D}) = \{tid \mid (tid, P) \in \mathcal{D}, X \subseteq P\}.$$

Definition 2.3. *The* support count *(or frequency count) of an itemset* X *in* $\mathcal{D}$ *is the number of transactions in the cover of* X *in* $\mathcal{D}$*:*

$$support\_count(X, \mathcal{D}) = |cover(X, \mathcal{D})|$$

We also divide support count *to the total number of transactions in database to get the* support ratio *or* support*:*

$$support(X, \mathcal{D}) = \frac{support\_count(X, \mathcal{D})}{|\mathcal{D}|}$$

where $|\mathcal{D}|$ *denotes the total number of transactions in database. We also omit* $\mathcal{D}$ *whenever it is clear from context. The minimal support threshold* σ_{abs}*, with* $0 \leq \sigma_{abs} \leq 1$ *is used to determine whether an itemset is frequent by identifying if its support is more than* σ_{abs}*.*

Definition 2.4. *Let* $\mathcal{D}$ *be a transaction database, and* σ *a minimal support threshold. The collection of frequent itemsets in* $\mathcal{D}$ *respective to* σ *can be formulated as:*

$$\mathcal{F}(\mathcal{D}, \sigma) = \{X \subseteq A \mid support(X, \mathcal{D}) \geq \sigma\}$$

Problem 2.1. ***(Itemset Mining)*** *Given* $\mathcal{D}$ *and* σ*, find* $\mathcal{F}(\mathcal{D}, \sigma)$*.*

The set of itemsets $\mathcal{F}$ are not the only items of interest, but rather their actual supports are more useful in real practice.

Definition 2.5. *If X and Y are itemsets, and $X \cap Y = \{\}$, an association rule can be expressed as $X \Rightarrow Y$. This implies that if a transaction contains all items in X, it has to also contain all Y items. X is therefore, known as the* body or antecedent*, and Y is accordingly known as the* head or consequent *of the rule. The support/support_count of association rule $X \Rightarrow Y$, equals $X \cup Y$'s support/support_count. An association rule is called frequent if its support is greater than a given minimal support threshold σ_{abs}. In this thesis only the absolute minimal support threshold for association rules is considered and the subscript abs, unless explicitly stated otherwise, is omitted.*

Definition 2.6. *The association rule confidence of $X \Rightarrow Y$ is defined as the conditional probability of having Y, with an existing X, in the same transaction, or:*

$$confidence(X \Rightarrow Y, \mathcal{D}) = Probability(Y|X) = \frac{support(X \cup Y, \mathcal{D})}{support(X, \mathcal{D})}$$

Note that we will remove database $\mathcal{D}$, when it is clear from context. The rule is deemed confident *if $Probability(Y|X)$ surpasses a given minimal confidence threshold γ, with $0 \leq \gamma \leq 1$.*

Definition 2.7. *Suppose that transaction database $\mathcal{D}$, minimum support threshold σ and minimum confidence threshold γ are given. The collection of frequent and confident association rules with respect to σ and γ can be formulated as:*

$$\mathcal{R}(\mathcal{D}, \sigma, \gamma) = \left\{X \Rightarrow Y \ | X, Y \subseteq A \wedge X \cap Y = \{\} \wedge X \cup Y \in \mathcal{F}(\mathcal{D}, \sigma) \wedge confidence(X \Rightarrow Y, \mathcal{D}) \geq \gamma\right\}$$

or simply $\mathcal{R}$ when $A, \mathcal{D}$, σ and γ are clear from the context.

Problem 2.2. ***(Association Rule Mining)*** *Given $\mathcal{D}$, σ and γ, find $\mathcal{R}(\mathcal{D}, \sigma, \gamma)$.*

Not only the association rules $\mathcal{R}$, but their actual support and confidence of the rules are useful in real practice.

Example 2.1. *Suppose that $A = \{beer, diaper, cheese, chocolate\}$ and consider the database in Table 2.1. Table 2.2 provides frequent itemsets in $\mathcal{D}$ where $\sigma = 25\%$ and Table 2.3 provides confident and frequent*

Table 2.1: *Transaction Database for Example 2.1*

Database	
1000	{beer, diaper, chocolate}
1001	{ beer, diaper }
1002	{ cheese, chocolate }
1003	{diaper,cheese }

association rules where $\sigma = 25\%$ *and a* $\gamma = 50\%$*. Based on Table 2.2 the set of the frequent itemsets, is as follows:*

$\mathcal{F}(\mathcal{D}, \sigma) = \Big\{\{beer\}, \{diaper\}, \{cheese\}, \{chocolate\}, \{beer, diaper\}, \{beer, chocolate\}, \{diaper, pizza\},$
$\{diaper, chocolate\}, \{cheese, chocolate\}, \{beer, diaper, chocolate\}\Big\}$

The first ever algorithm proposed for solving the association rule mining problem was divided into two phases [2]. In the first phase, all frequent itemsets are generated.In the second phase, all frequent and confident association rules are generated. A Majority of association rule mining algorithms follow this two-phased strategy. In the following sections, these two phases are discussed in further detail. In addition to the support and confidence measures, other measures have been proposed for obtaining more interesting association rules. Tan et al. [67] provide an overview of various measures proposed in statistics, machine learning and data mining literature.

2.2 FREQUENT ITEMSET MINING

Discovering all frequent itemsets is a challenging task. The search space is exponential with respect to the number of items occurring in the database. On the other hand, the support threshold, may limit the output to a reasonable subspace. In addition, such databases are usually massive and contain millions of

Table 2.2: *Itemsets and their support in database of Example 2.1*

Itemsets	Cover	Support Count	Support
$\{beer\}$	$\{1000, 1001\}$	2	50%
$\{diaper\}$	$\{1000, 1001, 1003\}$	3	75%
$\{cheese\}$	$\{1002, 1003\}$	2	50%
$\{chocolate\}$	$\{1000, 1002\}$	2	50%
$\{beer, diaper\}$	$\{1000, 1001\}$	2	50%
$\{beer, chocolate\}$	$\{1000\}$	1	25%
$\{diaper, cheese\}$	$\{1003\}$	1	25%
$\{diaper, chocolate\}$	$\{1000\}$	1	25%
$\{cheese, chocolate\}$	$\{1002\}$	1	25%
$\{beer, diaper, chocolate\}$	$\{1000\}$	1	25%

transactions, which makes support counting expensive. In this section, these two aspects are analyzed in further detail.

The search space of all itemsets comprises precisely $2^n - 1$ different itemsets, where n is the total number of items in A. If A is large enough, it is impossible to employ a naive approach for generating and counting the supports of all itemsets over the database in a reasonable time. In many applications, A contains thousands of items leading to the number of itemsets; the power set of this is intractable. Numerous approaches have been proposed to remedy this problem, to perform more directed searches in the search space.

2.2.1 Search Space for Itemset Mining

When deploying such a search, multiple collections of candidate itemsets are generated, and their supports are computed until all frequent itemsets have been generated.

Table 2.3: *Association rules and their support and confidence of Example 2.1*

Rule	Support Count	Support	Confidence
$\{beer\} \Rightarrow \{diaper\}$	2	50%	50%
$\{beer\} \Rightarrow \{chocolate\}$	1	25%	100%
$\{diaper\} \Rightarrow \{beer\}$	2	50%	66%
$\{cheese\} \Rightarrow \{diaper\}$	1	25%	50%
$\{cheese\} \Rightarrow \{chocolate\}$	1	25%	50%
$\{chocolate\} \Rightarrow \{beer\}$	1	25%	50%
$\{chocolate\} \Rightarrow \{diaper\}$	1	25%	50%
$\{chocolate\} \Rightarrow \{cheese\}$	1	25%	50%
$\{beer, diaper\} \Rightarrow \{chocolate\}$	1	25%	50%
$\{beer, chocolate\} \Rightarrow \{diaper\}$	1	25%	100%
$\{chips, chocolate\} \Rightarrow \{beer\}$	1	25%	100%
$\{beer\} \Rightarrow \{diaper, chocolate\}$	1	25%	50%
$\{chocolate\} \Rightarrow \{beer, diaper\}$	1	25%	50%

If the size of a collection of candidate itemsets surpass the available RAM, the I/O cost could become very expensive. Furthermore, generating as few candidate itemsets as possible is vital, since computing the supports for them can be a tedious task. In the best-case scenario, only the frequent itemsets are generated and their supports are counted (which is not the case in general). The fundamental property used by most algorithms is the support monotonicity property.

Lemma 2.1. ***(Support monotonicity property)*** *Given database $\mathcal{D}$ and two itemsets X and Y, if $X \subseteq Y$, then $support(Y) \leq support(X)$.*

Proof. This follows immediately from that $cover(Y) \subseteq cover(X)$, which is true by construction. ■

Therefore, if an itemset is infrequent, so must be all its supersets. In the literature, this property is referred to as the *downward closure property*. Mannila and Toivonen [53] showed that frequent itemsets can be denoted by the collection of *maximal or minimal* frequent itemsets, and proposed the notion of the *border* of a downward closed collection of itemsets for this purpose.

Definition 2.8. ***(Border)*** *The border $Bd(\mathcal{F})$ contains those itemsets $X \subseteq A$ such that all subsets of X are in $\mathcal{F}$, and no superset of X is in $\mathcal{F}$:*

$$Bd(\mathcal{F}) = \left\{X \subseteq A \mid \forall\ Y \subset X : Y \in \mathcal{F} \wedge \forall\ Z \supset X : Z \notin \mathcal{F}\right\}$$

Those itemsets in $Bd(\mathcal{F})$ that are in $\mathcal{F}$ are noted as the positive *border $Bd^{+}(\mathcal{F})$, and those itemsets in $Bd(\mathcal{F})$ that are not in $\mathcal{F}$ are noted as the* negative *border $Bd^{-}(\mathcal{F})$. Below example better illustrate the definition.*

Example 2.2. *Using the same $\mathcal{F}(\mathcal{D}, \sigma) = \Big\{\{beer\}, \{diaper\}, \{cheese\}, \{chocolate\}, \{beer, diaper\}, \{beer, chocolate\}, \{diaper, pizza\}, \{diaper, chocolate\}, \{cheese, chocolate\}, \{beer, diaper, chocolate\}\Big\}$ of Example 2.1.*

The borders are:

- $Bd(\mathcal{F}) = \big\{\{beer, cheese\}, \{beer, diaper, chocolate\}, \{diaper, cheese\}, \{cheese, chocolate\}, \{diaper, cheese, chocolate\}\big\}$
- $Bd^{+}(\mathcal{F}) = \big\{\{beer, diaper, chocolate\}, \{diaper, cheese\}, \{cheese, chocolate\}\}\big\}$
- $Bd^{-}(\mathcal{F}) = \big\{\{beer, cheese\}, \{diaper, cheese, chocolate\}\big\}$

Theorem 2.2. *[53] Consider database $\mathcal{D}$, and σ are given. Finding the collection $\mathcal{F}(\mathcal{D}, \sigma)$ would require that at least all itemsets in the negative border $Bd^{-}(\mathcal{F})$ to be evaluated.*

It is worthwhile to note that the number of itemsets in the positive or negative border of any given downward closed collection of itemsets of A can still be large, however, it is bounded by $\binom{|A|}{\lfloor \frac{|A|}{2} \rfloor}$ known as Sperner's theorem. It could become infeasible to generate all frequent itemsets for a given database if their number is large. Furthermore, if the transaction database is dense, or σ is set too low, there could exist many frequent itemsets, making the process infeasible to send them all to the output, considering that a frequent itemset of size k contains at least $2^k - 1$ other frequent itemsets [52, 60].

2.2.2 Transaction Database

Access to the database is required to compute the supports of a collection of itemsets. The representation of the transaction database is an important consideration in most algorithms. Theoretically, such databases can be represented using a binary two-dimensional matrix where each row represents an individual transaction and each column represent an item in A. There are several ways of implementing such a matrix. The most commonly employed is the horizontal data layout where each transaction has a unique identifier along with a list of items occurring in that transaction. An alternative layout is the vertical data layout, consisting of a set of items, each followed by its cover [74]. Many algorithms use a tree structure to store the transaction database in the memory for itemset mining problem. The best known is the FP-Growth Algorithm [32]. In

the following sections, we will discuss the well-known techniques and algorithms for frequent itemset and association rule mining.

2.3 APRIORI FREQUENT ITEMSET AND ASSOCIATION RULE MINING

Agrawal et al. [2] developed the first algorithm to generate all frequent itemsets and confident association rules, known as AIS. They also introduce this mining problem. Soon after, the algorithm was enhanced and took a new name, the Apriori Algorithm. For the remainder of this chapter, it is assumed, unless stated otherwise, that the list of all items is denoted as $\mathcal{I}$. Figure 2.1 outlines the itemset mining phase of the Apriori algorithm. Notation $X[i]$ is used to represent the ith item in X. The k-prefix of an itemset X is the k-itemset $\{X[1], ..., X[k]\}$. The algorithm iteratively generates candidate itemsets C_{k+1} of size $k+1$, starting with k = 0 (line 1), by performing a breadth-first search through the search space of all itemsets. The database, is scanned to count the supports of all candidate k-itemsets, and those candidate itemsets are incremented (lines 5-10). All frequent itemsets are then inserted into $\mathcal{F}_k$ (lines 12-14). If all subsets of an itemset are known to be frequent, it is deemed a candidate. For example, if C_1 contains all items in $\mathcal{I}$, and at a certain level k, all itemsets of size $k+1$ in $Bd(\mathcal{F}_k)$ will be generated. This can be done in two stages. In the first stage join, $\mathcal{F}_k$ is joined with itself. Then, the union $X \cup Y$ of itemsets $X, Y \in \mathcal{F}_k$ is generated only if they have the same $k-1$-prefix (lines 14-16). In the second stage, prune, $X \cup Y$ is only put in C_{k+1} if all of its k-subsets appear in $\mathcal{F}_k$ (lines 17-19).

When all frequent itemsets are generated, all frequent and confident association rules can be generated. The algorithm for this purpose (shown in Figure 2.2) is very much like the frequent itemset mining algorithm. The first step is to generate all frequent itemsets using frequent itemset mining algorithm. Next, every frequent itemset I is separated into a candidate consequent Y and a antecedent X, where $X = I \setminus Y$. This procedure starts with $Y = \{\}$, resulting in the rule $I \Rightarrow \{\}$, always holding with 100% confidence (line 4). Afterwards, the algorithm iteratively produces candidate heads C_{k+1} of size $k+1$, starting with $k = 0$ (line

Input: $\mathcal{D}, \sigma$
Output: $\mathcal{F}(\mathcal{D}, \sigma)$

```
1: C_1 := {{i} | i ∈ I}
2: k := 1
3: while C_k ≠ {} do
4:    // Compute the supports of all candidate itemsets
5:    for all transactions (tid, I) ∈ D do
6:       for all candidate itemsets X ∈ C_k do
7:          if X ⊆ I then
8:             X.support++
9:          end if
10:      end for
11:   end for
12:   // Extract all frequent itemsets
13:   F_k := {X | X.support ≥ σ}
14:   // Generate new candidate itemsets
15:   for all X, Y ∈ F_k, X[i] = Y[i] for 1 ≤ i ≤ k − 1, and X[k] < Y[k] do
16:      I = X ∪ {Y[k]}
17:      if ∀J ⊂ I, |J| = k : J ∈ F_k then
18:         C_{k+1} := C_{k+1} ∪ I
19:      end if
20:   end for
21:   k++
22: end while
```

Figure 2.1: *[2] Apriori Itemset Mining*

Input: $\mathcal{D}, \sigma, \gamma$
Output: $\mathcal{R}(\mathcal{D}, \sigma, \gamma)$

1: Compute $\mathcal{F}(\mathcal{D}, \sigma)$
2: $\mathcal{R} := \{\}$
3: **for all** $I \in \mathcal{F}$ **do**
4: $\quad \mathcal{R} := \mathcal{R} \cup I \Rightarrow \{\}$
5: $\quad C_1 := \{\{i\} \mid i \in I\}$;
6: $\quad k := 1$;
7: $\quad$ **while** $C_k \neq \{\}$ **do**
8: $\quad\quad$ // Extract all heads of confident association rules
9: $\quad\quad H_k := \{X \in C_k \mid \mathit{confidence}(I \setminus X \Rightarrow X, \mathcal{D}) \geq \gamma\}$
10: $\quad\quad$ // Generate new candidate heads
11: $\quad\quad$ **for all** $X, Y \in H_k, X[i] = Y[i]$ for $1 \leq i \leq k-1$, and $X[k] < Y[k]$ **do**
12: $\quad\quad\quad I = X \cup \{Y[k]\}$
13: $\quad\quad\quad$ **if** $\forall J \subset I, |J| = k : J \in H_k$ **then**
14: $\quad\quad\quad\quad C_{k+1} := C_{k+1} \cup I$
15: $\quad\quad\quad$ **end if**
16: $\quad\quad$ **end for**
17: $\quad\quad k{+}{+}$
18: $\quad$ **end while**
19: $\quad$ // Cumulate all association rules
20: $\quad \mathcal{R} := \mathcal{R} \cup \{I \setminus X \Rightarrow X \mid X \in H_1 \cup \cdots \cup H_k\}$
21: **end for**

Figure 2.2: *[2] Apriori Association Rule Mining*

5). A antecedent is considered as a candidate if all its subsets represent confident rules. This process is closely identical to the candidate itemset generation in frequent itemset mining Algorithm (lines 11-16). The support of I and X is retrieved from $\mathcal{F}$ for computing the confidence of a candidate head Y. All heads with confident rules are put in H_k (line 9) and eventually inserted into $\mathcal{R}$ (line 20). Fortunately, the time needed to find such rules would be equal to that of finding all frequent sets, if the number of frequent and confident association rules is not too large.

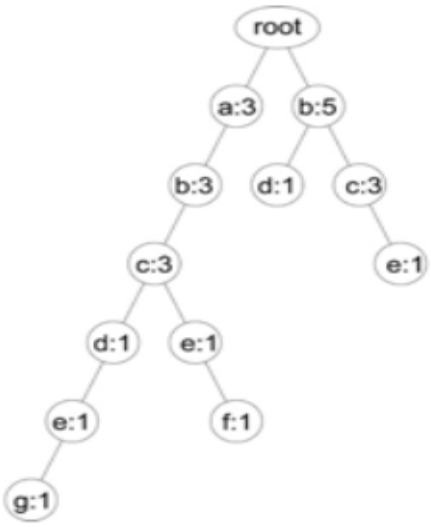


Figure 2.3: *Trie (Prefix-tree) based on Table 2.4*

2.4 TRIE DATA STRUCTURE

Trie (or prefix-tree) is another data structure that is commonly used [7, 45, 49, 31]. Each k-itemset has a node associated with it in a trie, which is also true for its $k-1$-prefix. The root node is the empty itemset. All of the 1-itemsets are branched from the root node, and the item they represent determines their branches' labels. Any other k-itemset is branched from its $k-1$-prefix. All candidate k-itemsets are, at a certain iteration k, stored at depth k in the trie. The path starts at the root node to find the candidate-itemsets that are included in a transaction T. It is easy to use a trie for the join step of the candidate generation procedure. This is due to the fact that all itemsets of size k with the same $k-1$-prefix are denoted by the branches of the same node [14]. Table 2.4 shows a transaction database and Figure 2.3 shows a prefix-tree representation of this transaction database, if transactions are sorted alphabetically first and then added to the tree.

Table 2.4: *Transaction Database*

tid	Content	Sort based on Alphabetical Order
1	$\{a, d, b, g, e, c\}$	$\langle a, b, c, d, e, g\rangle$
2	$\{b, f, c, a, e\}$	$a, b, c, e, f\rangle$
3	$\{b\}$	$\langle b\rangle$
4	$\{d, b\}$	$\langle b, d\rangle$
5	$\{a, c, b\}$	$\langle a, b, c\rangle$
6	$\{c, b, e\}$	$\langle b, c, e\rangle$
7	$\{b, c\}$	$\langle b, c\rangle$
8	$\{c, b\}$	$\langle b, c\rangle$

2.5 THE APRIORI-TID AND APRIORI-HYBRID ALGORITHMS

Many other algorithms that were proposed after the introduction of the original Apriori algorithm retain the same general structure. However, by adding new methods, they try to enhance certain steps in the original algorithm. Most research has focused on improving the support counting procedure of the Apriori algorithm, due to the fact that the performance of the Apriori algorithm is primarily determined by the performance of this procedure. It is worthwhile to recall that the performance of this procedure is primarily reliant on the number of candidate itemsets in each transaction. Agrawal et al. proposed two other algorithms, AprioriTid and AprioriHybrid along with the Apriori algorithm [4],[3].

The enhancement in AprioriTid algorithm lies in its ability to reduce the required time for the support counting procedure. This is achieved by replacing each transaction in the database with a set of candidate itemsets in that transaction (the adapted transaction database is denoted by $\bar{C}_k$). The algorithm is shown in Figure 2.4. AprioriTid algorithm performs much faster in later iterations; however, its performance is much slower than the original Apriori in first iterations. This is primarily a result of the additional overhead when

Input: $\mathcal{D}, \sigma$
Output: $\mathcal{F}(\mathcal{D}, \sigma)$

```
1: Compute F_1 of all frequent items
2: C̄_1 := D (with all items not in F_1 removed)
3: k := 2
4: while F_{k-1} ≠ {} do
5:    Compute C_k of all candidate k-itemsets
6:    C̄_k := {}
7:    // Compute the supports of all candidate itemsets
8:    for all transactions (tid, T) ∈ C̄_k do
9:       C_T := {}
10:      for all X ∈ C_k do
11:         if {X[1], ..., X[k-1]} ∈ T ∧ {X[1], ..., X[k-2], X[k]} ∈ T then
12:            C_T := C_T ∪ {X}
13:            X.support++
14:         end if
15:      end for
16:      if C_T ≠ {} then
17:         C̄_k := C̄_k ∪ {(tid, C_T)}
18:      end if
19:   end for
20:   Extract F_k of all frequent k-itemsets
21:   k++
22: end while
```

Figure 2.4: *AprioriTid algorithm*

$\bar{C}_k$ does not fit into the RAM and consequently is written to the disk. More details of the implementation in provided in [4].

To speed up the AprioriTid algorithm, AprioriHybrid [3] combines the Apriori and AprioriTid algorithms in a particular way. The idea behind this hybrid algorithm is to use Apriori for the initial iterations and switch to AprioriTid when $\bar{C}_k$ fits into RAM. A heuristic is used for estimating the size of $\bar{C}_k$ in the current iteration. The algorithm decides to switch to AprioriTid, if this size is small enough. Although AprioriHybrid is not airtight, it performs almost always better than Apriori.

2.6 DHP OPTIMIZATION

Park et al. [59] proposed Direct Hashing and Pruning (DHP) shortly after the Apriori algorithm was proposed to reduce the number of candidate itemsets. The mechanism that differentiates DHP is its ability to gather information about candidate itemsets of size $k+1$ during the kth iteration and the support of all candidate k-itemsets is counted by scanning the database. This is done by hashing all $(k+1)$-subsets of each transaction to a hash table after pruning. Each bucket has a counter in the hash table that keeps record of how many itemsets have been hashed in it. This method decreases the number of candidate itemsets to be counted significantly. However, there is a significant overhead for creating the hash tables and writing the adapted database to disk at every iteration [59].

2.7 THE DIC AND PARTITIONING ALGORITHMS

Brin et al. [17] proposed the DIC algorithm which, by separating the database into intervals of a certain size, attempts to reduce the number of passes over the database. In the first step, all candidate patterns of size 1 are generated. Then, over the first interval of the database, the supports of the candidate sets are counted. If all subsets are known to be frequent in advance, a new candidate pattern of size 2 is generated based on these supports and its support is counted over the database along with the patterns of size 1. After each interval, candidate patterns are generated and counted. The algorithm would stop when no more candidates can be generated and all those generated have been counted throughout the database. While this technique cuts the number of scans on database considerably, its performance is severely reliant on the data distribution. Brin et al. [17] claim that the performance enhancements due to reordering all items in support ascending order is insignificant. However, this is not in general true for Apriori. The supports of the 1-itemsets that computed only in the first interval were the basis for the reordering in DIC. Therefore, success of this heuristic is highly dependent on the data distribution [18].

Savasere et al. [64] proposed a different approach known as the Partition Algorithm. This approach stores the database in the RAM via the vertical database layout and computes support of an itemset using intersection of two of its subsets covers. The algorithm does so by storing the cover of each frequent itemset. If X and Y are subsets of candidate k-itemset I, its support is computed by joining X and Y similar to Apriori algorithm where covers of X and Y are intersected to make the cover of I. The approach described means that covers of all itemsets would be stored in RAM. This would require a lot of space and makes it impossible for large databases to be processed by this algorithm. In order to address this issue, the algorithm divides the database into multiple disjoint partitions. The algorithm then computes frequent covers in each division and puts them back in a unified order as shown in the Algorithm in Figure 2.5. The choice of division size depends on the memory capacity and is determined in such a way that each partition can be stored in RAM by itself. At the end, all different parts along with their frequent sets are merged to create a superset of all frequent itemsets over the complete database. It is evident that if an itemset is frequent over the whole database, it should also be relatively frequent in one of the partitions. Therefore, if the Algorithm in Figure 2.5 is repeated multiple times, the end result would provide the most frequent itemsets over the whole database as shown in Figure 2.6.

2.8 THE ECLAT ALGORITHM

The vertical database layout is used in Eclat and intersection based approach is used to compute the support of an itemset as show in Figure 2.7. In this algorithm, line 6 computes the support while a candidate itemset is represented by each set $I \cup \{i, j\}$. In this approach, the number of candidate itemsets generated is much larger due to the fact that the algorithm does not rely on the monotonicity property. The itemsets required for the prune step are not available in this approach; therefore, Eclat generates itemsets by only employing the join step from Apriori. If all items in the database are ordered in the support ascending order, it can reduce the number of itemsets required to generate at each iteration and consequently will reduce the number of covers computed from these itemsets. This will result in less memory space required to store these covers.

Input: $\mathcal{D}, \sigma$
Output: $\mathcal{F}(\mathcal{D}, \sigma)$

Compute $\mathcal{F}_1$ and store with every frequent item its cover
$k := 2$
while $\mathcal{F}_{k-1} \neq \{\}$ **do**
  $\mathcal{F}_k := \{\}$
  for all $X, Y \in \mathcal{F}_{k-1}, X[i] = Y[i]$ for $1 \leq i \leq k-2$, and $X[k-1] < Y[k-1]$ **do**
    $I = \{X[1], \ldots, X[k-1], Y[k-1]\}$
    if $\forall J \subset I : J \in \mathcal{F}_{k-1}$ **then**
      $I.cover := X.cover \cap Y.cover$
      if $|I.cover| \geq \sigma$ **then**
        $\mathcal{F}_k := \mathcal{F}_k \cup I$
      end if
    end if
  end for
  k++
end while

Figure 2.5: *[64] Local Itemset Mining for Partition algorithm*

Input: $\mathcal{D}, \sigma$
Output: $\mathcal{F}(\mathcal{D}, \sigma)$

Partition $\mathcal{D}$ in $D_1, \ldots, D_n$
// Find all local frequent itemsets
for $1 \leq p \leq n$ **do**
  Compute $C^p := \mathcal{F}(D_p, \lceil \sigma_{rel} \cdot |D_p| \rceil)$
end for
// Merge all local frequent itemsets
$C_{global} := \bigcup_{1 \leq p \leq n} C^p$
// Compute actual support of all itemsets
for $1 \leq p \leq n$ **do**
  Generate cover of each item in D_p
  for all $I \in C_{global}$ **do**
    $I.support := I.support + |I[1].cover \cap \cdots \cap I[|I|].cover|$
  end for
end for
// Extract all global frequent itemsets
$\mathcal{F} := \{I \in C_{global} \mid I.support \geq \sigma\}$

Figure 2.6: *[64] Partition algorithm*

Input: $\mathcal{D}, \sigma, I \subseteq \mathcal{I}$
Output: $\mathcal{F}[I](\mathcal{D}, \sigma)$

```
1:  F[I] := {}
2:  for all i ∈ I occurring in D do
3:     F[I] := F[I] ∪ {I ∪ {i}}
4:     // Create D^i
5:     D^i := {}
6:     for all j ∈ I occurring in D such that j > i do
7:        C := cover({i}) ∩ cover({j})
8:        if |C| ≥ σ then
9:           D^i := D^i ∪ {(j, C)}
10:       end if
11:    end for
12:    // Depth-first recursion
13:    Compute F[I ∪ {i}](D^i, σ)
14:    F[I] := F[I] ∪ F[I ∪ {i}]
15: end for
```

Figure 2.7: *Eclat algorithm*

This reordering can be performed at every iteration between line 6-11. It can be argued that the task of counting support of all itemsets is much more efficient compared to Apriori. In addition, the total size of all covers kept in the RAM is less compared to Partition. This is due to the fact that in this approach, their covers are stored in RAM at iteration k [13].

After Eclat, another approach was proposed by Zaki and Gouda [75, 76]. This approach has the advantage of computing the support of an itemset more efficiently using the vertical database layout. The enhancement is possible due to less storage requirement stemming from storing *diffset*, the difference between the cover of I and the cover of the $k-1$-prefix of I, instead of the cover of a k-itemset I. Subtracting the size of *diffset* from the support of its $k-1$-prefix would result in the support of I. The main advantage of this approach is that there is no need to store the support for each itemset and it can be called as a parameter in the algorithm's recursive function calls. Given the two *diffsets* of its subsets $I \cup i$ and $I \cup j$, the diffset of an

itemset $I \cup \{i, j\}$, with $i < j$ follows recursively as:

$$\text{diffset}(I \cup \{i, j\}) = \text{ diffset}(I \cup \{j\}) \setminus \text{diffset}(I \cup \{i\}).$$

The new designation for this algorithm, known as dEclat [75], has proven to improve considerably the performance of the algorithm. However, the layout for the original database is still in vertical. If we look at an arbitrary recursion path of the algorithm, it starts from $\{i_1\}$, all the way up to $I = \{i_1, ..., i_k\}$.

Hipp et al. [35] proposed a hybrid optimization that combines Apriori and Eclat. The hybrid approach in the algorithm uses Apriori to start generating frequent itemsets in a breadth-first manner, and shifts to Eclat at some point as a depth-first strategy. The user chooses the exact switching point in this scenario. If too many transactions contain candidate itemsets for storing them into RAM, it is impossible to use Eclat, while continuing with Apriori would not be a problem.

2.9 THE FP-GROWTH ALGORITHM

In the rest of this thesis, for simplicity, we assume that that items in transactions (itemsets of length 1) are frequent.

FP-growth employs the horizontal as well as the vertical database layout for storing the database in RAM. The way it works is by storing the actual transactions from the database in a trie structure (prefix-tree) with a linked list of all attached transactions known as a Frequent-Pattern tree (FP-tree) [31], FP-tree construction and the FP-Growth procedure is shown in Figure 2.8. As a pre-processing step, the items in the database are ordered in support ascending order. Next, for the first step of the process, a root node is created and labeled as "null". For each transaction in the database, the items are processed in reverse order. Each node in the FP-tree has a counter to keep track of the transactions sharing it. Moreover, the transaction item nodes following the prefix are generated and associated with their links. In addition, a

Algorithm: **FP_growth.** Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

- D, a transaction database;
- *min_sup*, the minimum support count threshold.

Output: The complete set of frequent patterns.

Method:

1. The FP-tree is constructed in the following steps:
   (a) Scan the transaction database D once. Collect F, the set of frequent items, and their support counts. Sort F in support count descending order as L, the *list* of frequent items.
   (b) Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in D do the following.
   Select and sort the frequent items in *Trans* according to the order of L. Let the sorted frequent item list in *Trans* be $[p|P]$, where p is the first element and P is the remaining list. Call insert_tree($[p|P]$, T), which is performed as follows. If T has a child N such that $N.item\text{-}name = p.item\text{-}name$, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call insert_tree(P, N) recursively.
2. The FP-tree is mined by calling FP_growth(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, α)
(1)  if Tree contains a single path P then
(2)     for each combination (denoted as β) of the nodes in the path P
(3)        generate pattern β ∪ α with support_count = minimum support count of nodes in β;
(4)  else for each a_i in the header of Tree {
(5)     generate pattern β = a_i ∪ α with support_count = a_i.support_count;
(6)     construct β's conditional pattern base and then β's conditional FP_tree Tree_β;
(7)     if Tree_β ≠ ∅ then
(8)        call FP_growth(Tree_β, β); }
```

Figure 2.8: *[31] FP-Tree and FP-Growth Procedure*

header table is created to track occurrences of each item in the tree through node-links while storing its support. The rationale behind storing transactions in the FP-tree in support descending fashion comes from the fact that arranging the more frequently occurring items closer to the root would result in a higher likelihood of them being shared and consequently keeping the overall representation size of the database rather small. Example 2.3 shows the FP-tree with a header table of a sample transaction database.

Example 2.3. *Suppose that all transactions are sorted in the support descending order as shown in Table 2.5. The corresponding FP-tree (with the header table) illustrated in Figure 2.9.*

Table 2.5: *Transaction Database*

tid	sorted transactions
1	$\langle a, b, c, d, e, f \rangle$
2	$a, b, c, d, e \rangle$
3	$\langle a, d \rangle$
4	$\langle b, d \rangle$
5	$\langle b, d, f \rangle$
6	$\langle a, b, c, e, f \rangle$

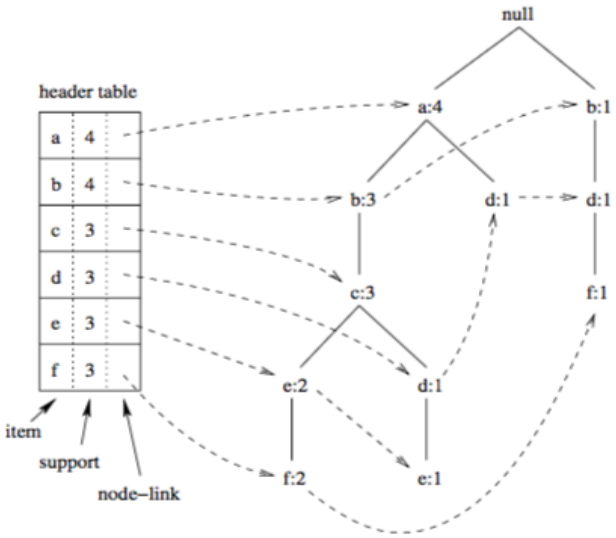


Figure 2.9: *FP-tree with its header table*

The supports of all frequent items for such FP-tree can be found in the header table. Clearly, the FP-tree outlines a lossless representation of the complete transaction database. In fact, the linked list for an item in the header table represents that item's cover in a compressed format. In the meantime, each branch stemming from root node represents a set of transactions in a compressed format. The FP-growth algorithm behaves like Eclat, excluding the FP-tree. However, it requires additional steps for maintaining the FP-tree structure while performing the recursion steps, in contrast to Eclat, which only requires maintaining the covers of all itemsets generated. For every $i \in \mathcal{I}$ to generate all frequent itemsets in $\mathcal{F}$, the i-projected database of $\mathcal{D}$ is generated as described in algorithm shown in Figure 2.10.

FP-growth is mainly different from Eclat in representation and maintenance of the i-projected database. This means that lines 5-10 of the Eclat algorithm need to be modified for FP-growth. To do so, the first task is to compute all frequent items for $\mathcal{D}^i$ (lines 6-10). Following the linked list that starts from the entry of i in the header table can accomplish this. In the next step, it follows its path up to the root node and adds its count to the support of each item it goes through. Next, for those transactions in which i occurs, the FP-tree for the i-projected database is built and intersected with the set of all frequent items in $\mathcal{D}$ greater than i (lines 11-13). Note that at every recursive step, item j occurring in $\mathcal{D}^i$ represents the itemset $I \cup \{i, j\}$. This means that for each frequent item i in $\mathcal{D}$, all frequent 1-itemsets in the i-projected database $\mathcal{D}^i$ are found by the algorithm. Further optimization on this technique can be achieved as follows. Consider an FP-tree consisting of a single path. It is evident that for such FP-tree, no recursion is required. Therefore, for single paths of the tree the recursion part can be skipped. It is also worth to mention that one of the FP-growth advantage is its ability to store dataset in a compressed format.

2.10 THE FELINE ALGORITHM

Most proposed algorithms for frequent itemset mining were for static datasets, where the number of transactions and support threshold are fixed during the mining process. What happens when the number of

Input: $\mathcal{D}, \sigma, I \subseteq \mathcal{I}$
Output: $\mathcal{F}[I](\mathcal{D}, \sigma)$

$\mathcal{F}[I] := \{\}$
for all $i \in \mathcal{I}$ occurring in $\mathcal{D}$ **do**
  $\mathcal{F}[I] := \mathcal{F}[I] \cup \{I \cup \{i\}\}$
  // Create $\mathcal{D}^i$
  $\mathcal{D}^i := \{\}$
  $H := \{\}$
  for all $j \in \mathcal{I}$ occurring in $\mathcal{D}$ such that $j > i$ **do**
    if $support(I \cup \{i, j\}) \geq \sigma$ **then**
      $H := H \cup \{j\}$
    end if
  end for
  for all $(tid, X) \in \mathcal{D}$ with $i \in X$ **do**
    $\mathcal{D}^i := \mathcal{D}^i \cup \{(tid, X \cap H)\}$
  end for
  // Depth-first recursion
  Compute $\mathcal{F}[I \cup \{i\}](\mathcal{D}^i, \sigma)$
  $\mathcal{F}[I] := \mathcal{F}[I] \cup \mathcal{F}[I \cup \{i\}]$
end for

Figure 2.10: *rewriting FP-Growth Algorithm based on Eclat*

transactions and support threshold varies during the mining process? Variations based on Apriori and the FP-Growth framework have been developed to address this problem, such as the FUP [21] and UWEP [8] algorithms based on the Apiori framework, and AFPIM [38], FELINE with CATS tree [22], and CanTree [45] Algorithms based on the FP-Growth framework.

In another effort, Cheung and Zaiane [22] used a CATS tree toward an interactive mining problem. A CATS tree uses the idea of FP-tree for storage compression, and does not generate candidate itemsets for finding frequent itemsets.

The tree construction process starts with a database scan. New transactions are added at the root level and compared with the children nodes. This comparison helps avoiding duplicates. Hence, if the new items have the same contents as the children, the transaction is merged with the node that has the highest frequency level. This process continues recursively until all common items are found and duplicates are eliminated. Any remaining items become an addition to the last merged node. To keep the hierarchy of the tree, if a node has a frequency higher than its ancestors, it should be swapped in such a way that the frequency always flows down the tree with higher frequencies at the higher nodes. To illustrate this process, let us look at an example.

Example 2.4. *For the database in Table 2.6, CATS tree is shown step by step after adding each transaction in Figure 2.11.*

The important steps in this process to be noted of are as follows. At first, the CATS tree is empty. In the next step, Transaction $\{a, d, b, g, e, c\}$ *is added to the root, with no modifications. After adding transaction* $\{d, f, b, a, e\}$*, common items i.e.,* (a, d, b, e) *are merged. Therefore, node* e *is swapped with node* g*. As there is not other common item, the remaining item from the new transaction i.e.,* (f) *is added to a new branch to* e*. Transactions* $\{a\}$ *and* $\{d, a, b\}$ *are added in a similar way. When transaction* $\{a, c, b\}$ *is added, common items a and b are merged. Nodes b and d are swapped and move up. However, there is an issue with common item c, which cannot be swapped and merged due to violation of the frequency law. Therefore, item c is added*

Table 2.6: *Transaction Database for CATS tree [45]*

	Tid	Contents
Original Database (DB)	1	$\{a, d, b, g, e, c\}$
	2	$\{d, f, b, a, e\}$
	3	$\{a\}$
	4	$\{d, a, b\}$
First Group of insertion (db1)	5	$\{a, c, b\}$
	6	$\{c, b, a, e\}$
Second Group of insertion (db2)	7	$\{a, b, c\}$
	8	$\{a, b, c\}$

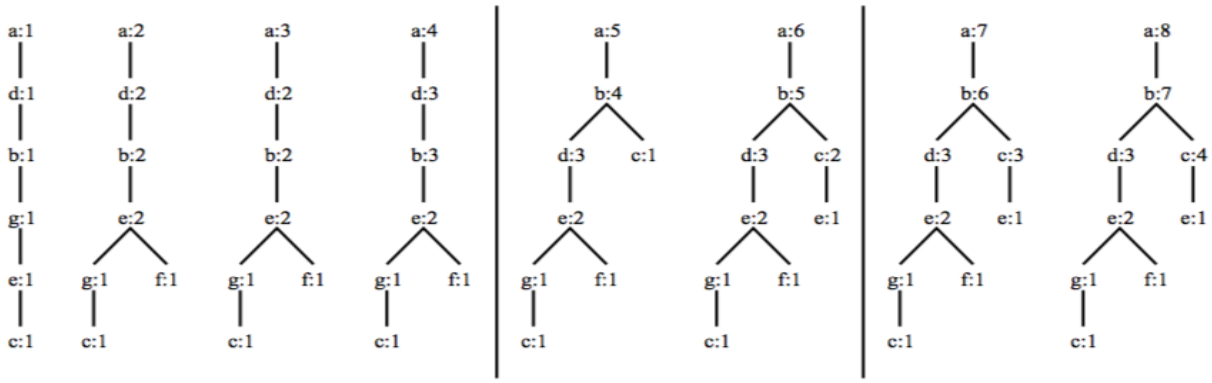


Figure 2.11: *CATS tree in addition of each transaction of Table 2.6 (from left to right) [45]*

as a new branch to b to address that issue.

It is important to make a few remarks. First, CATS trees have the property of keeping all items in every transaction, whereas FP-trees only keep the frequent items. Second, CATS trees use local frequency to order their nodes, whereas FP-trees employ global frequencies. For example, after transaction with $Tid = 6$ is added, item e is above c on the left branch while item c is above e on the right branch. This leads to some concerns regarding CATS tress.

1. Since the tree construction in CATS only used a single data scan (without prior knowledge of data), its maximum compression is not guaranteed.

2. The tree compression is sensitive to (a) database transaction order and (b) transaction item order.

2.11 THE AFPIM ALGORITHM

Koh and Shieh [38] were the first researchers to introduce the Adjusting FP-tree for Incremental Mining (AFPIM) algorithm in 2004. The main difference between this algorithm and its ancestor FP-tree is that it uses another notion for the frequent item. In this algorithm, frequent item is an item, for which its frequency is equal to or greater than a threshold known as *preMinsup*. This threshold is usually lower than that of FP-tree user-support threshold *minsup*. In the same way as FP-tree, all frequent items are arranged in a descending order based on their global frequency. When operations (i.e, insertions, modifications, and/or deletions) are done on transactions, frequency of items change, and this implies that the order in the tree needs to be adjusted. The AFPIM addresses this issue by swapping items in the tree using a bubble sort, which recursively exchanges adjacent items.

Insertion may lead to new items in the tree, and may introduce new items by making non-frequent items in the previous iteration to frequent items in the current database. Since in this situation, the existing

tree cannot be used to represent the database, the AFPIM algorithm needs to rescan the whole database and build a new FP-tree, which can be computationally heavy. To illustrate the workflow of the AFPIM algorithm, consider the following example. In the database shown in Table 2.6, let us set the threshold *preMinsup* to 35% and the minimum support threshold *minsup* to 55%). The original FP-tree along with the trees after the first and second insertions are shown in Figure 3.2. It is worthwhile to make some observations. The AFPIM algorithm scans the entire database to determine the global frequency of each item (i.e., $(a:4, b:3, d:3, e:2)$). In another scan, the algorithm only keeps the frequent items. In this case items having frequency equal or greater than *minsup* must be frequent since $minsup \geq preMinsup$; the reverse does not hold.

For database (DB), FP-tree contains items a, b, d, and e. After transactions with Tid 5 and 6 are inserted, item c (which had a frequency of 1 becomes frequent (because of preMinsup) with a frequency of 3 in $DB \cup db1$. Since not all frequent items in $DB \cup db1$ are in the FP-tree for DB, the AFPIM algorithm needs to rescan the entire database (i.e., $DB \cup db1$) again to build a new FP-tree. This will lead to lot of I/Os, especially when the database is large.

As discussed earlier, one of the issues with AFPIM is its necessity to update items in the tree other than the ones inserted/deleted due to affect of such items on other items. This usually occurs when ordering of items needs to be changed since frequency of some items are affect by others. This brings another major challenge with AFPIM compared to FELINE, as AFPIM employs bubble sort for exchanging adjacent tree nodes recursively. Furthermore, bubble sort has $O(h^2)$ computational order, where h is the number of nodes in the FP-tree. Hence, AFPIM can be computationally inferior compared to its competitors.

Another issue with the AFPIM is determining its parameter preMinsup. This additional parameter ensures only items with a frequency equal to or higher than this threshold are kept in the tree. However, finding a value for minsup itself is known to be challenging and preMinsup is an additional step to defining that parameter. Therefore, finding two parameters makes it an even more challenging task to complete.

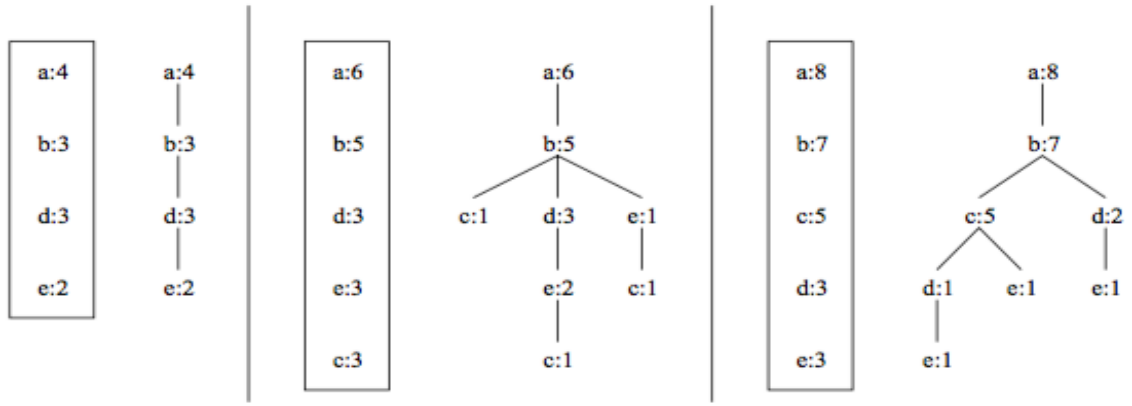


Figure 2.12: *FP-trees for DB, DB ∪ db1 and DB ∪ db1 ∪ db2 (from left to right) [45]*

2.12 THE CP-TREE AND CAN-TREE CONSTRUCTION

The Canonical-order Tree (CanTree) is a representation for a transaction database which does not need to rescan the whole database when it is updated due to transactions being inserted or deleted. This is due to the structure of this tree where items can be arranged in alphabetical or lexicographic order. The CanTree algorithm works in the same way as the FP-tree; however, the main differences are that CanTree builds the prefix tree based on a canonical order and mines the tree in the same fashion as FP-Growth algorithm [45].

This structure is meant to be used for incremental mining. One of the main advantages of this structure is that it only requires one scan of the database compared to that of FP-tree where one scan is required to determine frequency of the items and another to sort them numerically while keeping only the most frequent ones. This feature brings some important properties to this algorithm and tree structure as follows.

- Property 1. Items are ordered in a fixed global canonical ordering.
- Property 2. Items' orders are not affected by the frequency changes due to incremental updates.
- Property 3. A parent node always has the frequency that is at least more than the sum of all its

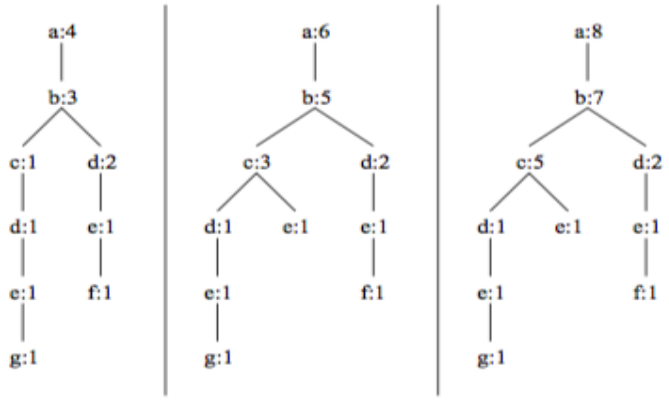


Figure 2.13: *CanTree for DB, $DB \cup db1$ and $DB \cup db1 \cup db2$ (from left to right) [45]*

children.

Because of these unique properties, transaction can be easily added the CanTree structure without having to rescan the database or searching for mergeable paths. Despite delivering the fastest tree construction among all FP-growth based algorithms, CanTree has a major drawback, which is its mining performance due to large branches in its trees. Figure 3.2 illustrates CanTree from Table 2.6 database.

A comparison between FELINE, AFPIM, and CanTree is provided below:

- FELINE

    1. To maintain the CATS tree, single scan of the incremental database db would suffice.
    2. Local frequency of the items in their path determines their descending order.
    3. Updates in the original database DB can incur swapping or merging of tree nodes.

- AFPIM

    1. The AFPIM algorithm would require two scans of the updated databased in the worst case.
    2. Items global frequency positions them in a descending order in the tree.

3. Updates to the databased can cause swapping, splitting, and merging of the nodes.

- CAN-Tree

  1. A single scan of the incremental portion of the database db would be sufficient.
  2. The frequency changes brought in by updates to the database do not affect the canonical order in which the items are arranged.
  3. Updating items in the original database would not require nodes in the tree to be swapped.

The CP-tree, builds an FP-tree structure with a single pass of a transaction database. The item order of a CP-tree is maintained by a list, called an I-list. After adding number of the transactions, if the item order of the I-list differs from the frequency-descending item order (I-list contains the current frequency value of the items), the CP-tree is restructured by the current item order and the I-list is updated. CP-tree construction can be divided into two phases:

- Insertion phase: a) Transactions are read one at a time. b) Inserting them into the tree defined by I-list. c) Updating the frequency of affected items in the I-list.
- Restructuring phase: a) Re-ordering the I-list with respect to frequency-descending order of items. b) Reconstructing the tree to maintain the frequency descending order.

  In order to keep the prefix tree the same as FP-tree, this algorithm adjusts the tree using bubble sort. The algorithm needs to check all the branches in the tree to make sure the nodes in a branch are sorted, and rearrange the nodes in the branches to keep them in the sorted order. This will make the overall mining time reasonably long when a large fraction of branches in the tree require reconstruction. More detailed implementation is available in [68]. Construction of CP-Tree and comparison with CanTree is shown in Figure 2.14.

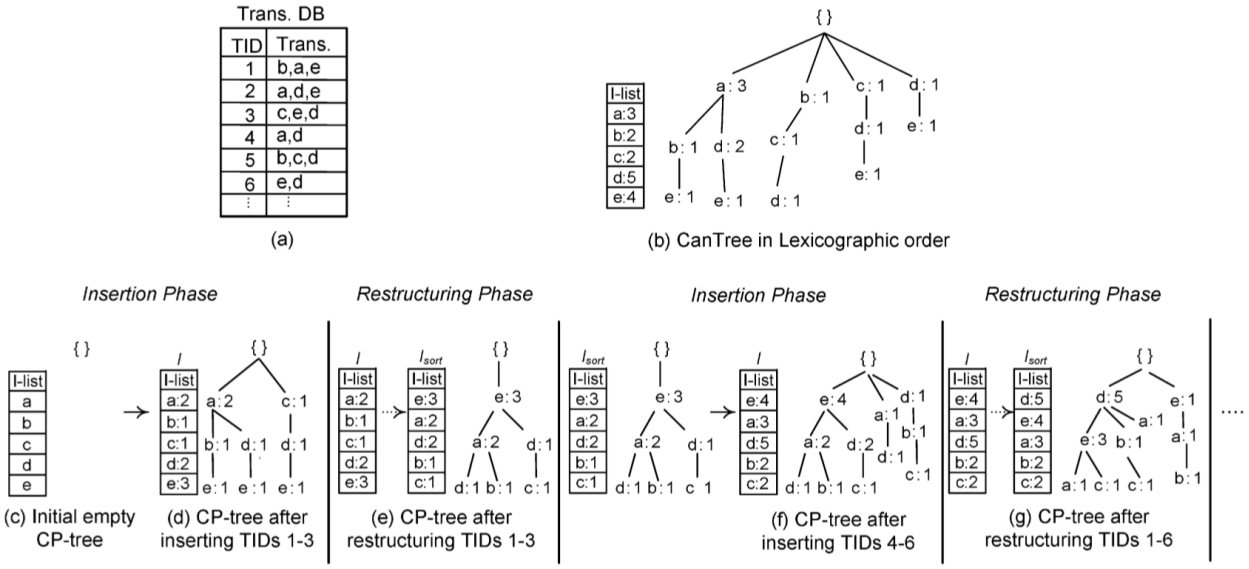


Figure 2.14: *[68] Construction of CP-Tree and comparison with CanTree*

2.13 SUMMARY

Mining frequent itemsets can be broadly divided into two categories; static mining and incremental/interactive mining. The algorithms in static mining assume that the data set does not change during the mining process. These algorithms can be further divided into two main subgroups: Apriori based algorithms and FP-Growth based algorithms. The main weakness of Apriori-based algorithms is the use of multiple database scans and numerous candidate generations. The FP-Growth algorithm eliminates the process of generating candidates and reduces the number of database scans to two. Within the FP-Growth based framework, the algorithms use a tree structure that captures all the necessary database information and mines the frequent itemsets in two or one database scans. Algorithms in this group include AFPIM [39], FELINE with CATS tree [23], Can-Tree [46] and CP-tree [68]. The FP-Growth based incremental algorithms perform mining by incrementally updating a compact data structure, usually an FP-tree structure. APFIM and FUFP-tree algorithms require two database scans in order to build the corresponding FP-tree. Upon updating the database, if any infrequent patterns become frequent, the APFIM algorithm may require rescanning the up-

dated database and FUFP-tree may require rescanning the original database. Many other algorithms were also proposed to improve the performance of FP-Growth. However, these algorithms only work on static datasets and cannot be used for incremental or interactive mining. CanTree was designed for incremental mining purposes. A user defined canonical order is used to arrange items and create the prefix tree which encapsulates all database content. This method requires only one database scan. In the CanTree, items are arranged according to a certain canonical order, which is unaffected by frequency changes. This method does not require merging, splitting, or swapping of nodes. CanTree deploys a similar mining method to FP-Growth by using a divide and conquer strategy. Conditional trees [31] are created for frequent items by traversing the tree path upwards only. Only frequent items are included in the traversal step and they can be easily looked up through a simple header table.

FELINE with the CATS tree is well suited for interactive mining but its efficiency for incremental mining (when the database changes frequently) is unclear, due to its complex tree construction process. Can-Tree captures all the database information in a prefix tree which is based on lexicographic-order and uses the same mining technique as FP-Growth. The creation of a Can-Tree is faster than that of an FP-tree because the use of the lexicographic order requires no swapping or reconstruction of the tree. But it suffers from poor mining performance in comparison to FP-Growth due to the potentially large size of the resulting tree. CP-tree does not maintain the FP-tree structure at all times, but intermittently (e.g, after an addition of every five transactions) does so. In order to keep the prefix tree the same as FP-tree, this algorithm adjusts the tree using bubble sort. The algorithm needs to check all the branches in the tree to make sure the nodes in a branch are sorted, and rearrange the nodes in the branches to keep them in the sorted order. This will make the overall mining time reasonably long when a large fraction of branches in the tree require reconstruction.

Chapter 3

Building FP-tree on the Fly: Single-Pass Frequent Itemset Mining

3.1 INTRODUCTION

The FP-Growth offers the advantage of avoiding costly database scans in comparison with Apriori-based algorithms. However, since it still requires two database scans, it cannot be used on streaming data. Also, the algorithm is designed for static datasets, where the input transactions are fixed and thus cannot be used for incremental or interactive mining. Existing incremental mining algorithms are not easily adoptable for on-the-fly, fast, and memory efficient FP-tree mining. In this chapter we propose a novel SPFP-tree (single pass frequent pattern tree) algorithm that scans the database only once and provides the same tree as FP-Growth. Our algorithm changes the tree structure dynamically to create a highly compact frequency-ordered tree on the fly. With the insertion of each new transaction our algorithm dynamically maintains a tree identical to an FP-tree. Experimental results show the efficiency of the SPFP-tree algorithm in both

incremental and interactive mining of frequent patterns.

In general, if we can scan the database only once to construct the FP-tree, we are able to mine patterns incrementally. Several algorithms such as Can-Tree [46] and CP-tree [68] have been proposed to capture all the necessary information in a database in one scan. The Can-Tree construction is based on the lexicographic order (i.e. alphabetical order) while CP-tree is based on the frequency descending order. The key contribution of this work is an algorithm that efficiently constructs an FP-tree on the fly, which is thus suitable for incremental and interactive mining. The algorithm constructs an SPFP-tree (single pass frequent pattern) by scanning the database only once and changes dynamically the structure of the tree to maintain the FP-tree structure. Our experimental results show that frequent itemsets mining with the SPFP-tree algorithm is more efficient than existing algorithms for single pass incremental mining.

The rest of this chapter is organized as follows. Section 3.2 presents the SPFP-tree construction for incremental and interactive mining. Experimental results are described in Section 3.3 and the summary is given in Section 3.4.

3.2 SPFP-TREE ALGORITHM

3.2.1 Tree construction/reconstruction

The construction of FP-tree in FP-Growth [33] consists of two major steps. First, the algorithm scans the database and finds the total frequency count of each item. Second, in the second database scan, items in a transaction are sorted in descending order of their frequency and added to the tree with prefix merging. The key feature of our algorithm is to maintain the structure of the FP-tree at all times so it can be mined efficiently due to its compact structure. The novelty of our approach lies in minimizing the number of branch comparisons whenever a new transaction is added.

The proposed algorithm consists of three steps: tree construction, tree reconstruction, and mining. Transactions are read, one at a time, and inserted into the tree. Then, the reconstruction phase modifies the tree structure to maintain its FP-tree structure. Finally, the mining algorithm finds the frequent itemsets from the tree. The major advantage of the method is the fact that the first two steps require only one database scan. Since the third step is identical as the tree mining part of the FP-Growth algorithm, we only describe the first two steps.

We maintain two hash tables for our algorithm. $Hash_1$ stores pairs (item, frequency count) where frequency count is the number of transactions containing the item, and $Hash_2$ stores pairs (frequency count, list of items). Thus, pairs in $Hash_2$ are reversed pairs from $Hash_1$ with items with identical counts collapsed to a single entry (the use of both tables will be explained below). Pairs in both hash tables are ordered with respect to the value of frequency count[1]. Consider a database with transactions: $T_1 : \{A, B\}, T_2 : \{A, C\}$ and $T_3 : \{A, B, C\}$. The resulting hash tables are: $Hash_1 = \{A : 3, B : 2, C : 2\}$ and $Hash_2 = \{3 : (A), 2 : (B, C)\}$. The frequency count order of the items is: $A > B \geq C$.

Let us walk through the details of the algorithm. We maintain a proper FP-tree at all times updating it on the fly. Whenever a transaction is processed, the following steps are taken.

- Sort items in the transaction based on $Hash_1$ frequency count.
- Add the transaction to the tree in a prefix-merging manner.
- Use $Hash_2$ to determine if any item violate the frequency count order (we describe below what exactly this involves) and reconstruct the tree if necessary.
- Update hash tables

As an example, consider the transactions and corresponding FP-tree in Figure 3.1(a) and Figure 3.1(b)

[1]A second hash map is used to store the order information. The full implementation can be find here: https://gist.github.com/dgrant/6332309

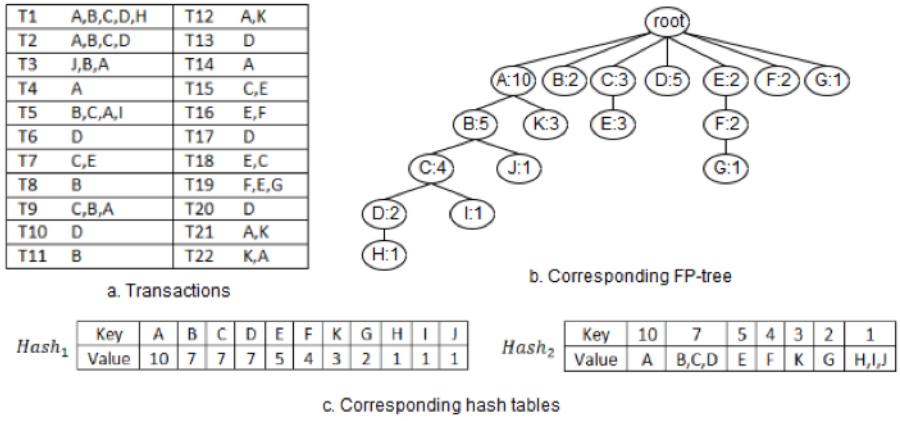


Figure 3.1: *Hash tables and corresponding FP-tree*

respectively (the equivalent hash tables are shown in Figure 3.1(c)). The frequency count order of the items is: $A > B \geq C \geq D > E > F > K > G > H \geq I \geq J$. We use $Hash_1$ table for ordering items in each transaction based on frequency count, and $Hash_2$ to find out if the frequency count order has changed.

Now suppose transaction $T_{23} : \{F, E\}$ is to be processed (so far, the frequency count for E and F is 5 and 4, respectively). The items in the transactions are first sorted with respect to the $Hash_1$ table to E, F and added to the tree in a prefix-merging manner as shown in Figure 3.2 (only the affected branch of the tree is shown). After items E and F are added, we need to verify that there are no items in the $Hash_2$ table with frequency counts equal to the frequency counts of E or F (key 5 has only E as its value and key 4 has only F as its value). This test guarantees that no tree reconstruction is required because adding items F and E does not change the frequency count order of the items (still $A > B \geq C \geq D > E > F > K > G > H \geq I \geq J$). A change in order is possible only when the added item has its current frequency count equal to that of another item. In that case, the addition of a new transaction with that item increases its count above the other one. Finally, the hash tables are updated as shown in Figure 3.2.

To see when the tree reconstruction is necessary, consider transaction $T_{24} : \{D, G\}$. Again, the items are sorted first within the transaction according to their order in $Hash_1$ table (D, G is the correct order) and added to the tree as shown in Figure 3.3. $Hash_2$ table (as shown in Figure 3.2, before it is updated for

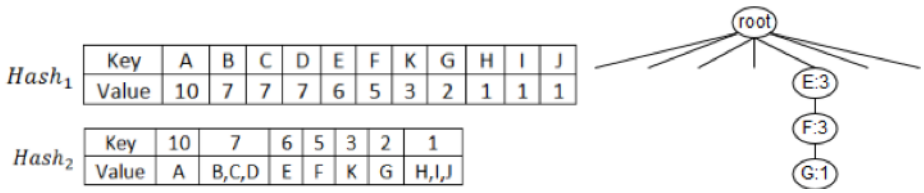


Figure 3.2: *Transaction* $\{F, E\}$ *added*

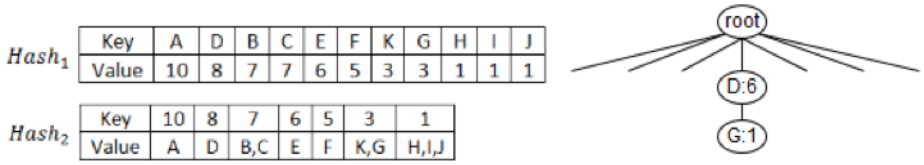


Figure 3.3: *Transaction* $\{D, G\}$ *added*

T_{24}) shows that there are two other items, B and C, whose frequency counts are equal to item D's frequency count (key 7 has values B, C, and D). In this case, we call node D a winner and nodes B and C losers. If there is a branch in the tree where node D's parent is B or C, it will lead to reconstruction of the tree (this is necessary to retain the FP-tree properties). Finally, hash tables are updated as shown in Figure 3.3. Note that the frequency order counts has now changed to: $A > D > BC > E > F > K > G > HIJ$.

Since there is a branch in the FP-tree where Ds parent is C (leftmost branch in Figure 3.1(b)), the reconstruction phase is called. The winner node and all of its children are removed from its parent node and all of its children are kept in memory to be reassigned as children to other nodes. This is illustrated in Figure 3.4(a). Then the loser nodes local counts in the branch they share with the winner node are decreased by the local count of the winner node as shown in Figure 3.4(b). If the local count of the loser node is equal to the local count of the winner node, the loser node is removed as its count becomes 0. The winner node is then added as a child to the immediate parent of the loser nodes, node A in our case, as shown in Figure 3.4(b). If node As children already contain the winner node, then these two nodes can be merged (changing the local count of the two nodes to the sum of the local counts). This is the only case in which two nodes are allowed to be merged. Then the final phase of reconstruction commences as illustrated in Figure 3.4(c). In this phase, we first iterate through the list of loser node items (nodes B and C), a copy

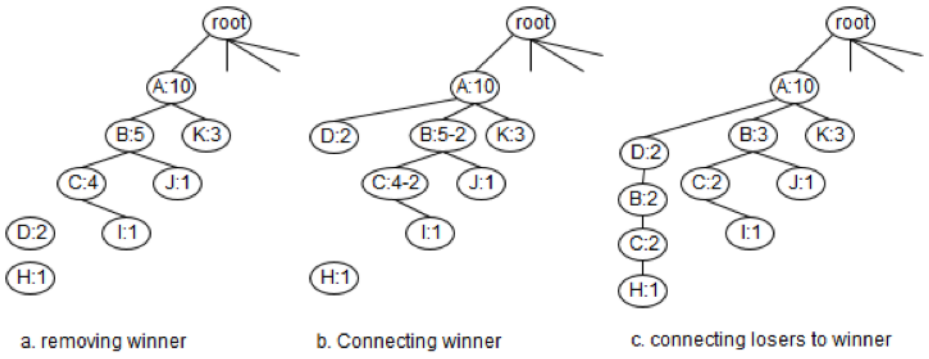


Figure 3.4: *Reconstruction*

of the loser node items will be created and added to the winner node as children. The counts of these items however will be set to the winner node count (Node D with count 2). Finally, the child of the winner node (node H) is added as a child to the bottom-most loser node.

The resulting tree is the same as the FP-tree, and all items are in frequency descending order ($A > D > B \geq C > E > F > K > G > H \geq I \geq J$). Once the FP-tree is built, frequent item sets can be mined similar to the mining part of FP-Growth algorithm for different support thresholds.

Figure 3.5 shows the corresponding pseudo code of the algorithm. Lines 1 to 17 read each transaction from the database, and update the $Hash_1$ and $Hash_2$ tables respectively. If items with the same frequency counts are found (loser items) they are removed from the $Hash_2$ table (line 6). Line 10 is in charge of the reconstruction procedure given the winner and loser items and then the winner item with a new frequency count (previous frequency count plus one) will be added to the $Hash_2$ table.

In the reconstruction procedure, we find all parents of winner nodes which are in the losers list, and decrease their local counts by winners local count (if the local count for losers node becomes zero, the node will be removed). Then we add the winner node to the immediate parent of the loser nodes. Finally, starting from the winner node, iterating through the list of loser nodes' items, a copy of loser nodes will be created and their local counts will be set the same as winners local count, and added to the winners children. Also, the children of the winner node will be added to the children of bottom-most loser node (line 23 to 33).

```
Input: Transactional DB
Output:  SPFP-Tree
Begin
 1.   For each transaction T in DB
 2.    For each item X in T
 3.      If X is in hash1
 4.          HithertoCountX := Hash1[X]
 5.          Increment value of  X in Hash1
 6.          Remove X from Hash2[HithertoCountX]
 7.          If Hash2[HithertoCountX] is not empty
 8.            WinnerItem=X
 9.            LosersList=Hash2[HithertoCountX]
 10.           reconstruction(WinnerItem, LosersList)
 11.         EndIf
 12.         insert X into Hash2[HithertoCountX+1]
 13.     else
 14.        insert X into Hash1 with count 1
 15.        insert X into Hash2[1]
 16.    EndIf
 17.  EndFor
 18.   sort T based on frequent descending based on Hash1
 19.   insert T into SPFP-Tree
 20. End For
 21. End
 22.
 23. reconstruction (WinnerItem ,LosersList):
 24.  For each node W with value equal to WinnerItem
 25.       Find the farthest ancestor A from W, which is
             in LosersList.
 26.    If A is not null
 27.      Reduce the local  count of all nodes  from W to A by
             the local count of W
 28.      Insert W as a child of A's parent
 29.      Add path from A to W's parent,  with local count of W
             To the new inserted W.
 30.      Assign all children of W to the end of the new inserted
             path.
 31.      Delete all node having zero local count
 32.    EnIf
 33.  EndFor
```

Figure 3.5: *SPFP-tree Algorithm*

3.2.2 Correctness of the SPFP-tree algorithm

We prove the correctness of the SPFP-tree algorithm. The objective of the proof is to show that tree maintained by the algorithm is the FP-tree. The key procedure is the tree reconstruction: we show that

after the reconstruction the resulting tree is an FP-tree.

Definition 3.1. *Recall Definition 2.1 in chapter 2.*

Definition 3.2. *A batch B is a finite sequence of transactions. Let B_k be the first k transactions $\langle T_1, T_2, ..., T_k \rangle$ based on their appearance in $\mathcal{D}$, $1 \leq k \leq N$.*

We define two operators: $||_k$ as count operator and $\leq_k$ as ordered relation operator on B_k respectively:

$$|a_i|_k = \text{ count of } a_i \text{ in } B_k \quad 1 \leq i \leq n, 1 \leq k \leq N \tag{3.1}$$

$$a_i \leq_k a_j \iff (|a_i|_k < |a_j|_k \text{ or } (|a_i|_k = |a_j|_k \text{ and } i < j)) \quad 1 \leq i, j \leq n, 1 \leq k \leq N \tag{3.2}$$

By Eq. 3.2 items are ordered in a batch by their frequency count, and if they have equal frequency count, they ordered by their index. Consider a batch $B_k, 1 \leq k \leq N$. We define the following trees:

- An unpacked tree $T_k^U (1 \leq k \leq N)$ created from B_k, is defined as follows.
  - All the transactions in B_k are attached to the root of the tree, R, without prefix merging.
- Packed tree $T_k^P (1 \leq k \leq N)$ created from B_k, is defined as follows.
  - All the transactions in B_k attached to R with prefix merging (prefix-tree).
- Level k for each tree consists of all nodes that have distance k from the root (the root is at level 0). The depth of the tree is equal to the largest level of the tree.

In order to create the FP-tree, we need to create the $\leq_N$ order first and then the T_N^P tree (prefix merging). Our algorithm works in N steps. In step i $(1 \leq i \leq N)$ an $\leq_i$ order is created from the $\leq_{i-1}$ order and T_i^P tree from T_{i-1}^P tree. In order to show that our tree is the same as FP-tree, it is required to show that the proposed method for reconstructing T_i^P from T_{i-1}^P tree is correct (which means that Equation 3.3 and Equation 3.4 produce the same tree).

$$T_i^P \qquad 1 \leq i \leq N \tag{3.3}$$

$$T_{i-1}^P \text{ and adding } T_i \qquad 1 \leq i \leq N \tag{3.4}$$

If adding T_i does not require tree reconstruction, then Eq. 3.3 and Eq. 3.4 are obviously the same. But suppose that, there are two items a and b with equal frequency counts in T_{i-1}^P, and T_i consist of one item a. By adding $T_i = \{a\}$ the order of items a and b will change (that is, $a \leq_{i-1} b$ and $b \leq_i a$). Now a branch in T_{i-1}^P needs reconstruction if it has items a and b, as shown in the left branch of Figure 3.6(a). In this case for T_i^P node a must have lower level than node b (because, $b \leq_i a$). The algorithm will change the tree structure as follows. First, $T_i = \{a\}$ added to the tree in a prefix-merging manner (changing node a's local count in l_1 from w to $w + 1$). See the right branch in Figure 3.6(b). Then node a in left branch will go to level l_1 in that branch and its local count k_1 will add up with a local count $w + 1$ in that level. The total local count for a is now $k_1 + w + 1$. Then the k_1 local count of b will be a's child in the left branch and all a's children in the left branch will be $b : k_1$'s child now. New local count for b in l_1 will be $(k_2 - k_1)$. See the right branch in Figure 3.6(c).

To validate the above reconstruction, consider the unpacked tree (the unpacked version o T_{i-1}^P is shown in Figure 3.7, top left). Due to $b \leq_i a$, item a must have lower level than item b. So for two items in the same path we only change the position of those items one by one (now $k_2 - k_1$ local count of b will remain in l_1 level and the remaining k_1 local count of b will be a's child in left branch). In order to get back to T_i^P tree is to pack everything again, after that an exact replica of an FP-tree has been created, see Figure 3.7 (the steps in the figure shows that: first by considering unpack tree, and then changing the node position one by one followed by packing back the result, that the same T_i^P in Figure 3.6(c) is created).

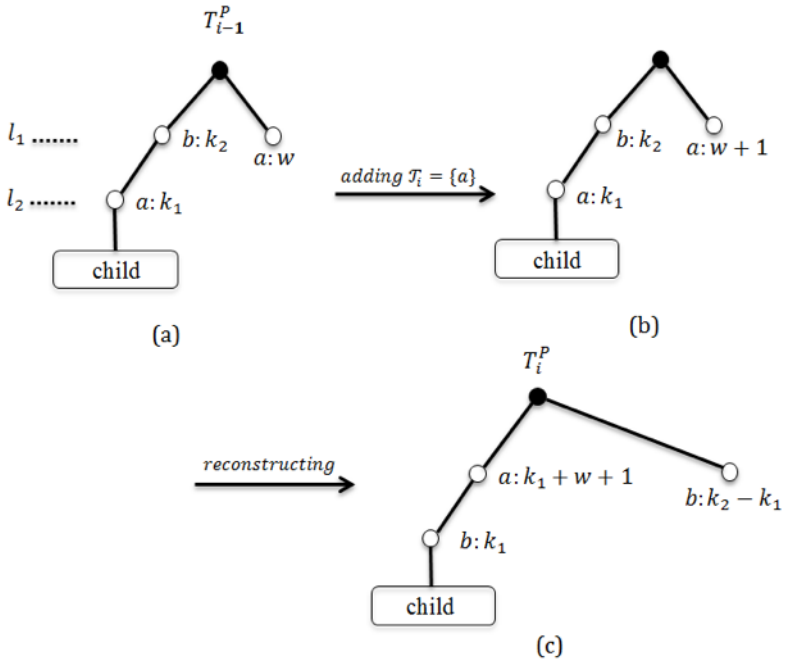


Figure 3.6: *Constructing* T_i^P *from* T_{i-1}^P

3.2.3 Incremental mining with the SPFP-tree

Items can be stored in a compact FP-tree structure regardless of whether they are frequent or not. It is then straightforward to add or delete transactions from the tree. In order to delete a transaction it needs to be sorted first based on the $Hash_1$ table. Then the tree is traversed downwards in order to find the corresponding nodes and their local counts are decreased by one. Afterwards, the $Hash_1$ and $Hash_2$ tables are updated and the reconstruction phase procedure of the tree will be called. To add a transaction one needs to follow the routine described in the SPFP-tree algorithm.

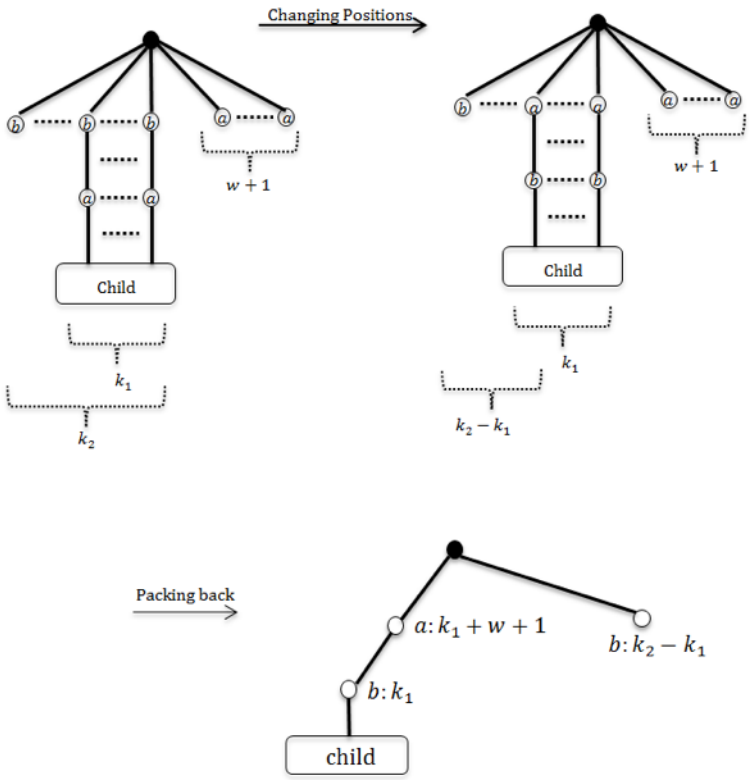


Figure 3.7: *Changing position in* ***unpacked*** *tree and then packing back the tree.*

3.2.4 Interactive mining with SPFP-tree

Since the SPFP-tree algorithm builds a tree from all the items in transactions, it supports mining the tree with different support thresholds. Given a minimum support threshold (min_sup), the algorithm traverses the tree upward from nodes that have a min_sup count in $Hash_1$ table, and builds the corresponding FP-tree. The tree is in frequency-descending order and nodes placed below the traversed nodes are not frequent. Similar to other interactive mining methods if the new min_sup is greater the min_sup in the previous round, it is possible to cache the frequent patterns in previous round and reuse them in the next round. This allows for further reducing of the total mining time.

3.3 PERFORMANCE STUDY

In this section, the performance of the SPFP-tree algorithm is evaluated in comparison with Can-Tree and CP-tree. These two were chosen for comparison since they show good performance among other incremental frequent item set mining algorithms. All programs are implemented in Java and run on Linux centos 6 with Intel core 2 duo 3.00 GHz CPU and 4 GB memory. The reported figures are based on the average execution time over multiple runs. The experiments were performed on sever-al datasets from the UCI Machine Learning Repository [11] specifically Chess, Connect, Mushroom and Accidents.

3.3.1 Performance study of execution time for different threshold levels

In the first experiment we measure how the minimum support threshold affects the runtime of the algorithms. Figure 3.8 shows the runtime for SPFP-tree versus CP-tree and CanTree on different support thresholds for four datasets. The total execution time includes time for the prefix tree construction, and frequent item set mining steps. *Total Time=Construction Time+Mining Time.*

CanTree has smaller construction time in comparison to CP-tree and SPFP-tree, because it uses alphabetical order to add transactions, and no reconstruction is needed. But the tree generated by the CanTree algorithm is larger than CP-tree or SPFP-tree (see Figure 3.10) resulting in larger mining time. The mining time is highly correlated to the number of nodes in the pruned prefix-tree (pruned based on the minimum support threshold). Therefore, when a low min_sup is used the pruned prefix-tree (either for FP-tree or alphabetical tree) has more nodes than when a higher min_sup is used. This causes a dominance in mining time, whereas when we have min_sup (either for the FP-tree or alphabetical tree), the construction time will be dominant.

Mining times for SPFP-tree and CP-tree are the same, since both algorithms have the same final FP-tree structure. Therefore, the difference between them depends on the construction time. The construction time

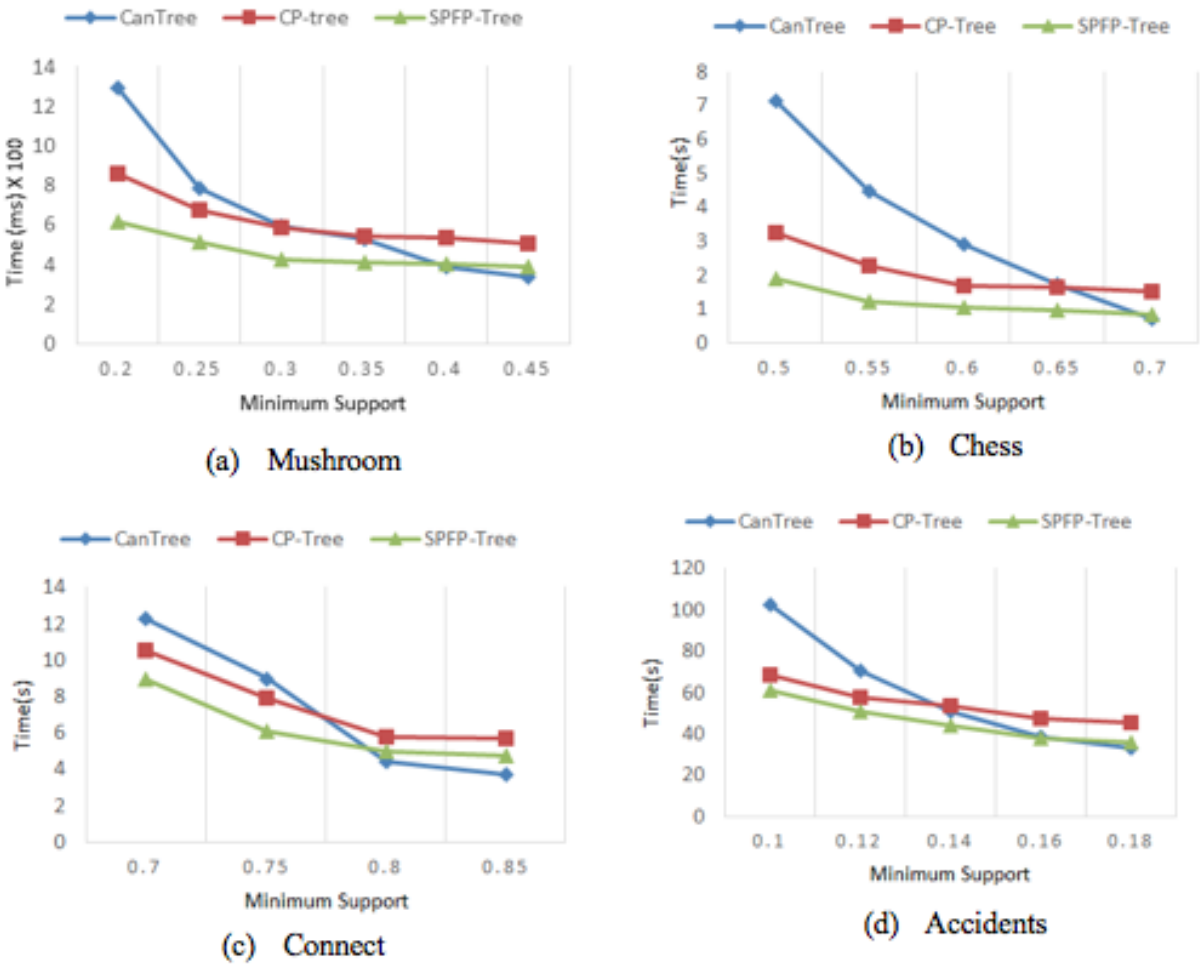


Figure 3.8: *Performance as a function of min_sup*

in the proposed algorithm is lower than that of CP-tree (see Figure 3.9). Therefore, in all cases it outperforms the CP-tree algorithm.

Comparing SPFP-tree with CanTree, it can be seen that with an increase in the support threshold (for example, more than 0.8 in the Connect dataset) the total required time for CanTree becomes smaller than that of the SPFP-tree algorithm. This is due to the fact that a higher support threshold makes *construction time dominant*, causing CanTree to have better performance due to its fast prefix-tree creation.

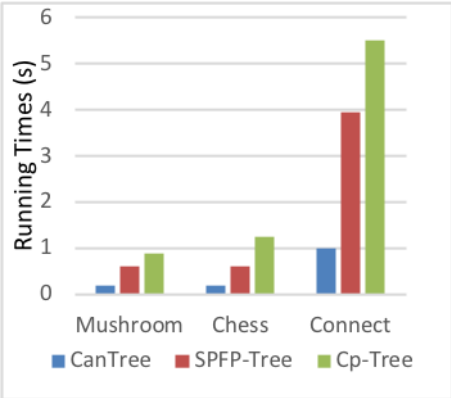


Figure 3.9: *Tree Construction Time*

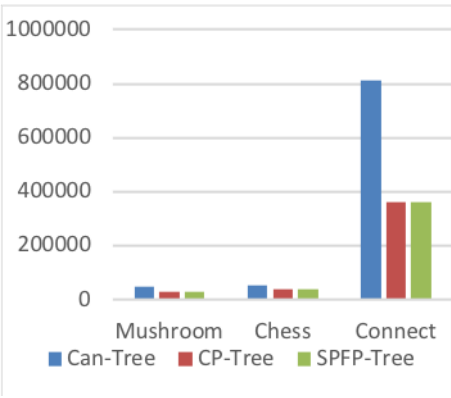


Figure 3.10: *Number of nodes in each tree*

3.3.2 Performance Study of Incremental Mining

In the next experiment, we compare the performance of the respective algorithms for incremental updates of SPFP-tree and CanTree. The experiment is performed on the Mushroom dataset for which 90% of the transactions are preloaded, and the remaining 10% is incrementally added to the tree in ten steps. The running time will be the time required for inserting the updated part of the database to the prefix-tree and mining the prefix-tree. Figure 3.11 shows the performance of SPFP-tree versus CanTree.

The experiment demonstrates that SPFP-tree outperforms Can-Tree on running time. This is due to the compact tree structure of SPFP-tree, which makes the mining part of its process much faster than CanTree.

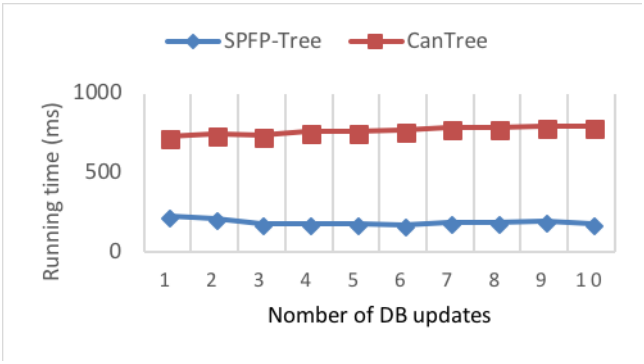


Figure 3.11: *Incremental mining on mushroom with min_sup = 0.1*

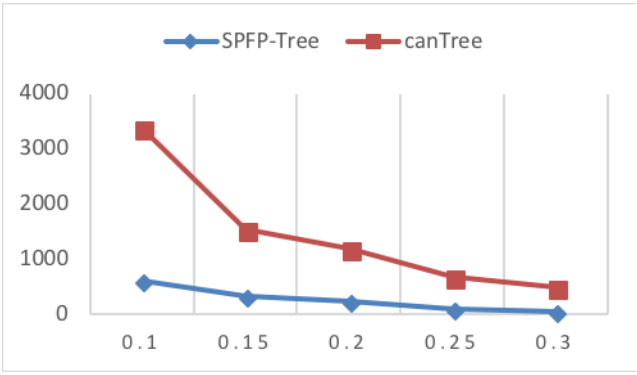


Figure 3.12: *Interactive Mining for Mushrooms Dataset*

3.3.3 Performance study of interactive mining

Interactive mining occurs when the user plans to mine a fixed database with different minimum support thresholds. The results of interactively mining the Mushroom dataset with the proposed SPFP-tree and CanTree are shown in Figure 3.12.

Both algorithms need to construct the tree once, and then prune it based on a min_sup interactively. The time reported here covers only the mining time (we assume the tree has already been built). The result shows that the SPFP-tree out-performs CanTree, due to its frequency-descending item ordering and more compact tree structure.

3.4 SUMMARY

In this chapter, a new method called SPFP-tree (single pass frequent pattern tree) for incrementally constructing FP-tree with a single pass of the data set has been proposed. The proposed algorithm rearranges the tree on the fly, and keeps the items in each branch of the tree in frequency-descending order (just as in FP-tree) after each transaction is added. Performance analysis results of SPFP-tree were reported against other algorithms for incremental mining, including CanTree and CP-tree. Our results show that SPFP-tree outperforms CP-tree in all the datasets on various support thresholds, and outperforms CanTree on lower support thresholds. Moreover, SPFP-tree is more memory efficient compared to CanTree because of its dense frequency-descending prefix-tree structure. The feasibility of the algorithm for incremental and interactive mining was also presented.

Chapter 4

Memory Efficient Frequent Itemset Mining

4.1 INTRODUCTION

When dealing with large databases, the representation and storage of the data becomes a critical factor in the processing time of the mining algorithms. The existing techniques deploy either list-based [61, 26, 25] or tree based [47, 50, 32, 65] structures to store data. The problem with both structures, however, is that with a large number of itemsets processed by the algorithm, the application may run out of memory.

The most popular structure for frequent pattern mining is FP-tree [32], which is a prefix-tree representing the transactions with only the frequent items in a compact way. In an FP-tree, the number of nodes required to create the tree can be substantially greater than the total number of distinct items in the dataset. This happens because, in general, items can be repeated multiple times in the tree. In the worst case (admittedly unlikely), the prefix tree can grow to a maximum of 2^n nodes, where n is the number of distinct items in a database. If a dataset holds 100 distinct items, to mine such a dataset a computer would require storage

capable of holding 2^{100} or approximately 10^{30} nodes. However, with the deployment of the method we propose in this chapter, we would only require a storage capacity capable of storing a total of 100 nodes (and at most $100 * (100 - 1)/2$ labeled edges), which is much more feasible.

The main objective of this work is to propose a memory efficient method to store the data processed by the frequent itemset mining algorithm. Instead of a prefix-tree, we store the data in a compact directed graph whose nodes represent items. In this way, the size of the graph is bounded by the number of distinct items present in the database.

Our algorithm was tested on six different data sets from the UCI Machine Learning Repository [12] and the results were analyzed and compared against results obtained from two state-of-the-art methods: FP-Growth [32] and CanTree [47]. The results not only showed a dramatic reduction in the amount of memory required to store the data structure required for mining, but also showed that running time of our algorithm was comparable to that of tree based frequent item set mining methods. In fact, in four of the tested datasets (Connect, Accident ,T10I4D100K and Pumsb) our algorithm is superior with respect to runtime performance to the CanTree method.

The rest of this chapter is organized as follows. Section 4.2 offers the details of the data structure and the mining algorithm. Experimental results are presented in Section 4.3, while the summary is in Section 4.4.

4.2 METHODOLOGY

As indicated above, the goal of our method is to decrease memory requirements for the data structure on which the mining for frequent patterns is performed. Thus, we replace the tree structure used in previous methods [47, 32] with a labeled directed graph.

4.2.1 Graph Construction

Definition 4.1. *Recall Definition 2.1 in chapter 2.*

For this problem we require ordered transactions which are sorted with respect to their indices, that is, $\mathcal{T}_k = \langle a_{k_1}, \cdots, a_{k_l} \rangle$, where $a_{k_i} \in A, 1 \leq i \leq l, l > 1$.

Transactions with $l = 1$, only increase the total counts of an item and do not add an edge to the graph. We define an edge-labeled directed graph as $G = (N, E)$ where:

- $N = A$
- $E = \bigcup_{k=1}^{m} (\{\langle a_{k_i}, a_{k_{i+1}} \rangle | \mathcal{T}_k = \langle a_{k_1}, \cdots, a_{k_l} \rangle \in D, 1 \leq i \leq l-1\} \cup \{\langle a_{k_l}, a_{k_1} \rangle | \mathcal{T}_k = \langle a_{k_1}, \cdots, a_{k_l} \rangle \in D\})$

We call the edges $\langle a_{k_i},\ a_{k_{i+1}} \rangle$, $1 \leq i \leq l-1$ *forward edge s*, and the edge $\langle a_{k_l},\ a_{k_1} \rangle$ a *backward edge*. Labels in our graph are binary strings, and we call them edge codes. A Label on a backward edge is called *backward edge code* or *BEC*, and label on a forward edge is called *forward edge code* or *FEC*. For a transaction with l items, $l-1$ forward edge between two contiguous items and one backward edge between the last and the first items in the transaction will be created.

Consider the following example.

Example 4.1. *Let* $T_1 : \{a_3, a_2, a_4, a_1\}$, $T_2 : \{a_4, a_3, a_1\}$, *and* $T_3 : \{a_3, a_2\}$ *be three transactions in a database, with four distinct items* $\{a_1, a_2, a_3, a_4\}$. *In order to add a transaction to the graph, the items within each transaction are first sorted based on their indices. Therefore, items in* T_1 *are sorted so that* $\mathcal{T}_1 = \langle a_1, a_2, a_3, a_4 \rangle$. *Then the following set of edges* $E = \{\langle a_1, a_2 \rangle, \langle a_2, a_3 \rangle, \langle a_3, a_4 \rangle, \langle a_4, a_1 \rangle\}$ *is created for transaction* T_1. *In this example three forward edges and one backward edge are created for the transaction (as shown in Figure 4.1.a with red edges). Then* T_2 *is added to the graph with two forward edges from* a_1 *to* a_3 *and from* a_3 *to* a_4 *and with one backward edge from* a_4 *to* a_1 *(as shown in Figure 4.1.b with green edges).*

Finally, T_3 is added to the graph. Since it shares its forward edge with T_1 and has a distinct backward edge, no new label is created for it (the labeling method is discussed in depth below). The graph with all three transactions is shown in Figure 4.1.c.

Each transaction requires a unique backward edge which completes a cycle for that transaction in the graph. When T_1 was added to the graph a label r was assigned to it. Since T_2 shares a common backward edge with T_1 we need to distinguish it from T_1 s backward edge, hence we labeled it g . For the overlapping edges such as these two we merge the labels ($r+g$ in this example). When T_3 is added to the graph, we create a new backward edge $\langle a_3, a_2 \rangle$ in the graph and since that edge is not yet in the graph an overlap of codes will not occur on that edge and the existing code (r) is used for it. We will show later that each transaction can be extracted from the graph with its unique backward edge code.

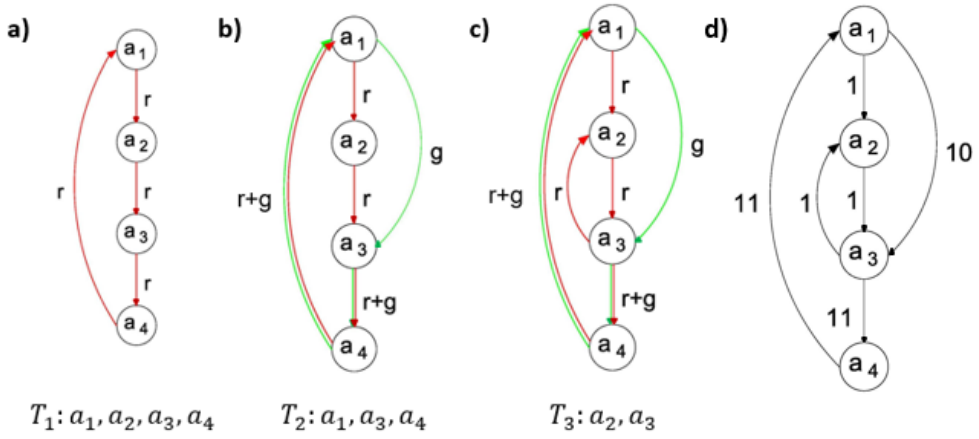


Figure 4.1: *Resulting graph after adding each of the transactions: T_1 (a) T_2 (b) T_3 (c), and (d) the completed graph after adding T_1, T_2 and T_3 with the new coding structure.*

4.2.2 Edge Labeling

For efficiency of implementation, we designed a coding structure based on binary strings. Each transactions code will be binary representation of powers of 2 values $\left(2^0,\ 2^1, 2^2, \cdots\right)$. In our example, r is represented

by 2^0 ($r \rightarrow 1$) and g is represented by 2^1 ($g \rightarrow 10$). We call the code assigned to each transaction a *unique backward code* (UBC) as this code is unique for a backward edge of every transaction. In our example, T_1 and T_2 share a backward edge so they need two distinct UBCs; 1 is used for T_1 and 10 is used for T_2. (Notice that T_3 does not share a backward edge $\langle a_3, a_2 \rangle$ with any other transaction hence it does not require a distinct UBC.) When two transactions with different UBCs share an edge, we need to label that edge with their two distinct UBCs. Again, for efficiency of implementation, we can do better: we use the logical OR of their UBCs. T_1 and T_2 have two common edges in the graph, $\langle a_3, a_4 \rangle$ and $\langle a_4, a_1 \rangle$. The UBCs of these transaction: 1 and 10 respectively, are ORed together to create FEC and BEC with the value 11. On the other hand, knowing the value of the edge code to be 11 we can deduce that this it is a combination of the UBC 1 and the UBC 10. The complete labeled graph for the three transactions is shown in Figure 4.1.d.

We now show formally how transactions are assigned their UBCs and how these UBCs are represented as graph labels. For the smallest transactions containing just two items, the corresponding UBC is calculated as follows:

$$UBC\left(< a_i, a_j >\right) = \begin{cases} 1 & j - i = 1 \\ 10^{\frac{(j-2)(j-3)}{2}}.10^i & otherwise \end{cases} \tag{4.1}$$

Thus, in Example 1, the UBC of T_3 is equal to 1.

For transactions with more than two items, the maximum UBC of consecutive pairs within that transaction is calculated as follows:

$$\begin{gathered} UBC\left(\langle a_1, a_2, \cdots, a_{k-1}, a_k \rangle\right) = \\ max\left(UBC\left(\langle a_1, a_2 \rangle\right), \cdots, UBC\left(\langle a_{k-1} \cdots a_k \rangle\right)\right) \end{gathered} \tag{4.2}$$

In Example 4.1, the UBCs for T_1 and T_2 respectively, are 1 and 10. Note that the UBCs calculated via Equation 4.1. are not necessarily unique for transactions. But any clashes of UBCs of two different

transactions will be identified *before* these transactions are inserted into the graph (this is taken care in Line 3 of **Procedure 1** below).

After assigning each transaction a UBC, we need to create labels (edge codes) of the graph edges, that is, the BECs and the FECs. As shown in Example 1, if two or more transactions share a backward edge, the labels of the edges shared by these transactions will be created via a logical OR of their UBCs. We say that the labels generated in this way *contain* the UBCs of their transactions. In fact, there is a straightforward way of telling when an edge code contains a given UBC.

Observation 4.1. *BEC L contains UBC X, if and only if, the logical AND between L and X is not zero.*

We now ready to present an algorithm for inserting a transaction into a graph.

Procedure 1: Insert Transaction T in to graph G

1. Calculate the UBC of T using Equation 4.1 and Equation 4.2; call it X.
2. Create forward and backward edges (unless they are already in G) for T and assign 0 for all its BECs and FECs.
3. If BEC of T already contains X, generate a new UBC for T: X =binary representation of $\left(2^{number\_of\_bits(BECofT)}\right)$.
4. Perform a logical OR on BECs and FECs of T with X

Two elements of the procedure require an explanation. First, assigning 0 in Line 2 initializes the values of the edge codes for edges in the graph that did not exist before so that the operations in Line 4 could be performed. Second, Line 3 verifies whether the UBC of the new transaction is unique on the backward edge of that transaction: if the BEC of that edge contains that UBC, it means that some other transaction with the same backward edge has the same UBC. To make the UBC of the new transaction unique, we assign it a new UBC as specified in Line 3.

4.2.3 Identifying Transactions in the Graph

We now present an example where we show how transactions are identified in the graph generated via Procedure 1.

Example 4.2. *Consider a database with four items and all possible transactions with length greater than one. Fig. 2.a. shows the transactions and their corresponding UBCs. The graph representing the transactions with appropriate edge codes is shown in Figure 4.2.b. Consider as an example the edge* $< a_1, a_2 >$*. This edge is shared by transactions* T_1*,* T_7, T_8 *, and* T_{11} *with their respective UBCs 1, 1, 1000, 1. A logical OR between these codes results in 1001. Note that even in this relatively complex example, the condition in Line 3 of* ***Procedure 1*** *is never satisfied and the transactions keep their original UBCs.*

The ultimate goal for creating the graph was to enable mining for frequent itemsets. For this, we need to identify the transactions in the graph and create conditional trees[31] and then the same FP-Growth mining algorithm will be used. As we have observed before, for every transaction there exists a loop in the resulting graph. In order to extract all transactions from the graph, a traversal of all backward edges is required. When a given backward edge is traversed, all UBCs of its BEC are extracted. These are the UBCs of all transactions sharing that backward edge. **Procedure 2** describes how the nodes of a transaction with a given UBC is identified. The procedure starts with node v which is the in-vertex of the given backward edge.

Procedure 2: Extract nodes of transaction with UBC X originating from node v

1. Perform the logical AND between X and all FECs of edges outgoing from v.
2. Follow the edge with non-zero result to reach the next node.
3. Repeat steps 1 and 2 until a loop is completed.

Consider again the graph shown in Figure 4.2.b. and assume we are considering the backward edge which connects a_4 to a_1. BEC of that edge is 1111 which contains four UBCs: 1, 10, 100, and 1000. Suppose that the transaction with UBC 10 is chosen for traversal. We initialize **Procedure 2** with UBC 10 and node a_1. Of the three outgoing edges, only the one from a_1 to a_3 with the FEC=10 succeeds ($10\ AND\ 10 \neq 0$). For the remaining two outgoing edges the AND result of their FECs and UBC 10 is zero ($(1001\ AND\ 10) = 0\ \ (100\ AND\ 10) = 0$). After reaching a_3 we can only go to a_4 ($10\ AND\ 11\ \neq\ 0$). The loop is then complete and transaction T_9 with items $\langle a_1,\ a_3,\ a_4\rangle$ is retrieved.

In order to extract all transactions within a graph in Figure 4.2.b., a traversal of all backward edges with their containing UBC is required. For example, a_4 has three backward edges with edges codes 1, 1001 and 1111. Code 1 contains UBC 1. Code 1001 contains UBC 1000 and 1. Code 1111 contains UBC 1, 10, 100 and 1000. By traversing all the corresponding UBCs on a_4 backward edges, all seven transactions containing item a_4 are extracted ($T_3, T_5, T_6, T_8, T_9, T_{10}\ and\ T_{11}$). These extracted transactions (all transactions containing item a_4) are used to create conditional trees [31] and then the same FP-Growth mining algorithm find the frequent itemsets [32]. UBCs are used to obtain transactions, but do not reflect the number of times that

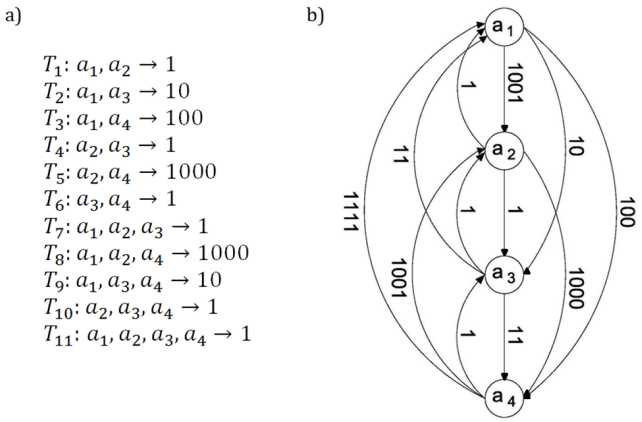


Figure 4.2: *a) A Transactional Dataset with four distinct items and all transactions with a length of greater than 2 b) the proposed methods resulting graph*

transaction has been repeated. If a transaction like T_9 with items $\langle a_1,\ a_3,\ a_4\rangle$ appears more than once (ex. three times) then an integer three is assigned for its corresponding UBC, as shown in Figure 4.3. Therefore, when a new transaction is parsed, if it already exists in the graph its integer count is incremented by one. In other words, the corresponding count of a transaction indicates its frequency within the database.

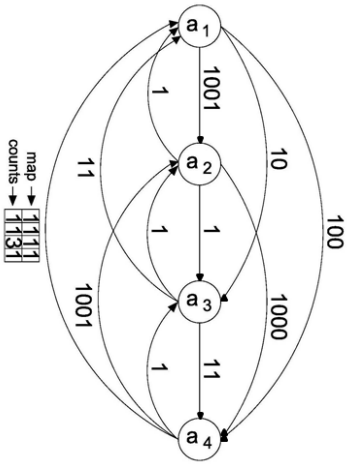


Figure 4.3: *For transactions which happen more than once we can set an integer to represent its count*

To speed up conditional tree [32] creation another step which can be taken is when we have a transaction such as T_{11} with items $\langle a_1, a_2, a_3, a_4\rangle$ not only do we add a directional link from a_4 to a_1 with a code of 1, but we do this for all other nodes connected to a_1 (as shown in Figure 4.4). This will help in speeding up the conditional tree creation. This is because the conditional tree which is based on T_{11} with items $\langle a_1, a_2, a_3, a_4\rangle$ also includes transactions $\langle a_1, a_2, a_3\rangle$ and $\langle a_1, a_2\rangle$. This extra step does not stall transaction processing, and is not a requirement.

4.2.4 Algorithms for graph construction

Algorithm 1 in Figure 4.5 shows the corresponding pseudo code of the graph construction phase. The graph itself is defined to be an adjacency list called graph list which uses an array indexed by item (node in graph) number. Each array item points to a list of the neighboring items of that item. A header list is also defined

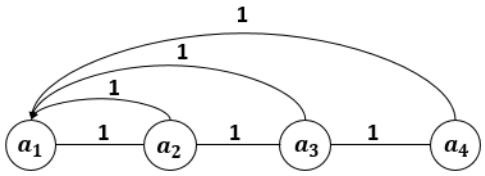


Figure 4.4: *In order to speed up conditional tree creation, we can add a directional link for all codes connecting to the first node as well.*

to store the distinct items in the database. The transactions will be processed one by one and if an item is reached that is not already in the header list, the item will be added to the end of the header list and correspondingly a node containing that item will be added to the end of the graph list. Each transaction will be sorted based on the header list and then each item in that transaction will be added to the neighbor list of its predecessor and the value of nodes will be updated accordingly. After constructing the graph, the graph can be mined to find the complete set of frequent patterns. To accomplish this job, conditional tree [31] creation phase will begin. The mining algorithm is the same as FP-Growth after conditional trees are created. In the conditional tree phase the graph is parsed according to the pseudo code offered in Algorithm 2 Figure 4.6. The graph list is parsed in reverse order, and all the nodes within the graph containing backward edges are found. The graph is then traversed based on their codes, and the conditional tree for each distinct item is generated.

4.3 PERFORMANCE RESULTS

In this section we present performance evaluation of our method (we call it a compact graph) in comparison to FP-Growth and CanTree is given. FP-Growth and CanTree demonstrate good performance in incremental frequent item set mining and were chosen as a baseline to assess the Compact Graphs results. All computational aspects were conducted in Java on a Linux Centos 6 with Intel Core 2 Duo 3.00 GHz CPU and 4 GB memory. From the UCI Machine Learning Repository the Chess, Connect, Mushroom, Accidents, Pumsb

Algorithm 1: FP_Graph construction

Input: transactional DB Output: FP_Graph

Method: the FP_Graph is constructed as follow.

```
Begin
For each transaction T in DB
    For each item K in T
        If K is not in header_List
            Insert K into header_list
            Insert a new Node N into Graph_list
            Set N.value = K
        End
    End
    Sort T based on header_list
    k = 0
    For each contiguous items a, b in T compute the UBC[k] as follow :
        i = header_list[a].index
        j = header_list[b].index
                UBC[ k ] = { 1                          j − i = 1
                           { 10^((j−2)(j−3)/2) . 10^i   otherwise
        k = k+1
    End
    Set T.UBC=  max_k UBC[ k ]
    Pre := first item in T
    Lst := last  item in T
    TBEC = graph_list[Lst][Pre].BEC
    If TBEC & T.UBC <> 0 :
        T.UBC = 2^length(TBEC)
    From second item in T to the end as L Do:
        If node N contain L is not in graph_list[pre] :
            N = new node
            N.value= L
            N.FEC= T.UBC
            insert N into graph_list[pre]
        Else
            N.FEC = N.FEC | T.UBC
        End
        Pre=L
    End
    If node N contain Pre is not in graph_list[Lst] :
        N = new node
        N.value= Pre
        N.BEC= T.UBC
        insert N into graph_list[Lst]
    Else
        N.BEC = N.BEC | T.UBC
    End
End
End
```

Figure 4.5: *Algorithm 1 shows FP Graph Construction*

```
Algorithm 2: Conditional Tree Creation
Input:FP_graph constructed based on algorithm 1
Output:Conditional Tree for each item
Method: the FP_graph is mined as follow.
Begin
For each item L in reverse of header_list
    For each node M in graph_list[item]
        If header_list[M.value].index less than header_list[L].index :
          Travers Graph_list from graph_list[M.value] until graph_list[L] reached based on M.BEC
          Add resulting path to the conditional tree of item M
        End
    End
End
End
```

Figure 4.6: *Algorithm 2 shows the Conditional Tree Creation*

and T10I4D100K datasets were used. Seven runs were conducted and reported runtime figures represent the average of their execution times.

Table 4.1 shows the number of nodes required to generate the prefix tree for CanTree, FP-Growth and the Compact Graph. Tests were also conducted on the total memory consumption of CanTree and FP-growth in comparison to the Compact Graph with the results compiled for each of the tested datasets as shown in Table 4.2. Compact Graph performs better in all cases, and other than the mushroom dataset, outperforms the other algorithms by a large margin. Due to its alphabetically structured tree and high node count, Can-Tree consumes the most amount of memory. FP-Growth has a more efficient tree structure which leads to smaller memory utilization and fewer nodes. However, Compact Graph utilizes the least number of nodes possible with a compact structure, and therefore is the most memory efficient.

Figure 4.7 shows runtime results for the Compact Graph, FP-growth, and Can-Tree utilizing different support thresholds on the tested datasets. The purpose of this experiment was to measure effects of the minimum support threshold on the runtime of algorithms and to confirm that the Compact Graph is not sacrificing the execution time of the mining algorithm. Total execution time consist of the amount of time

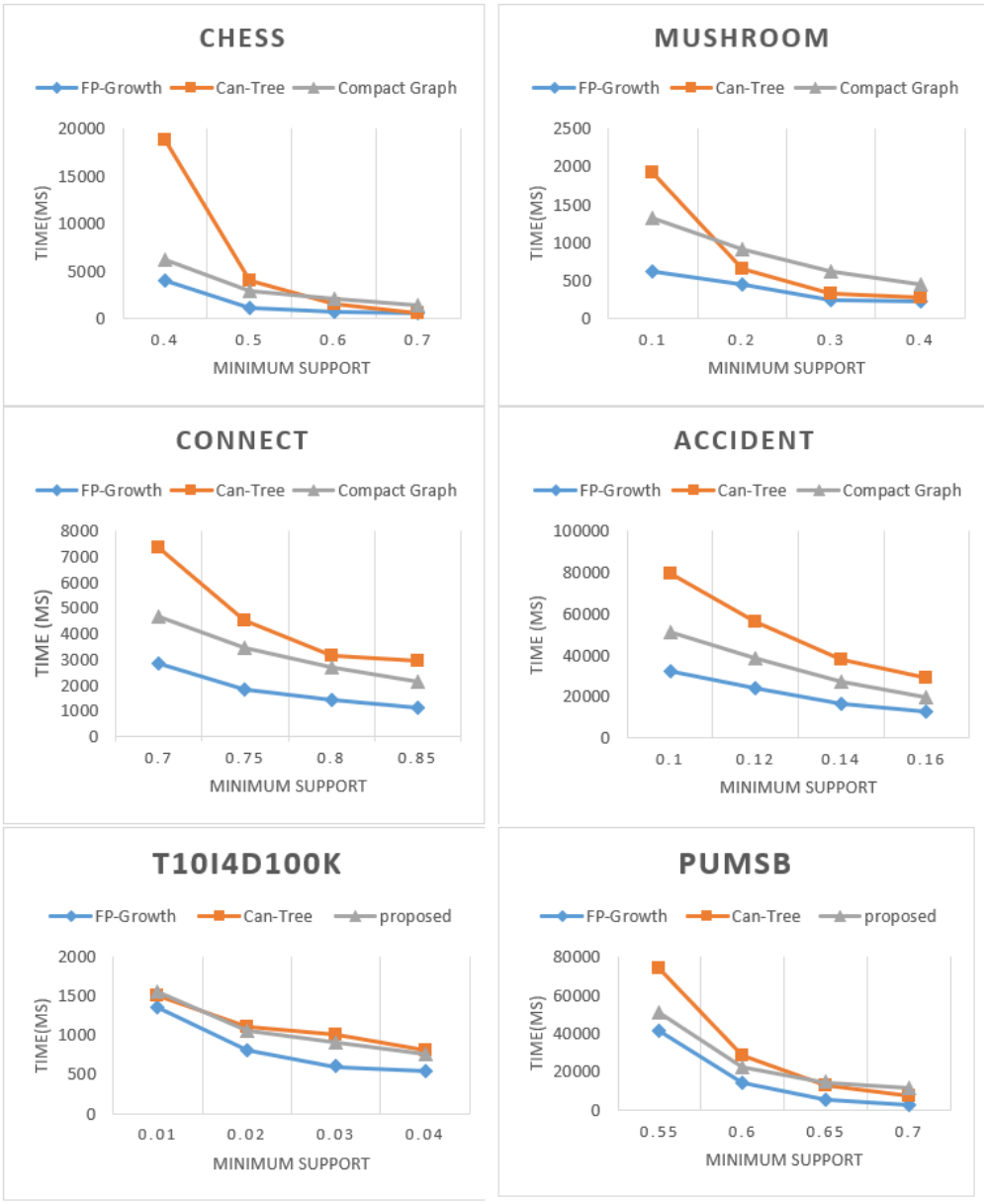


Figure 4.7: *Performance of each of the algorithms on 6 different datasets*

Table 4.1: *Number of nodes used for data representation by FP-Growth, CanTree, and the Compact graph.*

Dataset	FP-Growth	Can-Tree	Compact graph
Chess	38610	39551	75
Mushroom	27122	34004	119
Accident	4243242	5262556	468
Connect	359292	1092209	129
Pumsb	1126155	2773441	2088
T10I4D100k	714731	748409	870

required for prefix tree or graph construction in addition to the frequent item set mining step (Equation 4.3):

$$TotalTime = ConstructionTime + MiningTime. \tag{4.3}$$

The construction phase is shortest for Can-Tree in comparison to FP-Growth and the compact graph method; this is due to Can-Tree not requiring tree reconstruction and deploying an alphabetical order to add transactions. Despite having the fastest prefix tree construction phase, Can-Tree generates a larger tree than the ones generated by FP-Growth or the Compact Graph.

Mining time is highly correlated to the number of nodes in the pruned prefix tree (pruned based on the minimum support threshold), therefore larger prefix-trees result in larger mining times. This causes mining time to become a dominant factor, whereas when we have a high minimum support, the construction time will be dominant.

The Compact Graph has a higher construction time than both CanTree and FP-Growth, which is due to the need for encoding data in the graph. In the mining phase the Compact Graph is slower than FP-Growth, but based on the support threshold used it may prove a better option to Can-Tree. This is due

Table 4.2: *Memory Ratio Comparison.*

Minimum support	0.0	0.4	0.5	0.6	0.7
FP-Growth/compact	12	20	21	17	11
Can_Tree/compact	14	26	35	42	86

(a) *Chess*

Minimum support	0.0	0.1	0.2	0.3	0.4
FP-Growth/compact	4	8	7	5	3
Can_Tree/compact	5	17	29	67	98

(b) *Mushroom*

Minimum support	0.0	0.7	0.75	0.8	0.85
FPgrowth/compact	28	7	6	3	2
Can_Tree/compact	73	987	1067	1146	1463

(c) *Connect*

Minimum support	0.0	0.1	0.12	0.14	0.16
FPgrowth/compact	48	120	116	118	122
Can_Tree/compact	69	195	215	223	234

(d) *Accident*

Minimum support	0.0	0.55	0.6	0.65	0.7
FP-Growth/compact	9	12	26	23	18
Can_Tree/compact	17	323	376	485	523

(e) *Pumsb*

Minimum support	0.0	0.01	0.02	0.03	0.04
FP-Growth/compact	12	27	52	66	34
Can_Tree/compact	14	43	86	165	436

(f) *T10I4D100k*

to the complex structure of Can-Tree; with a lower support threshold, the Compact Graph mines the data more efficiently.

4.4 SUMMARY

The size of memory becomes a limiting factor in mining large databases. The solutions offered so far, such as those proposed in incremental mining methods, called for dividing the database into subsets and mining each of them separately. In this chapter, we took a different approach and faced the problem head-on. We proposed a new approach for storing large collections of frequent itemsets in a compact graph. This method utilizes only one node per each distinct item in the database, therefore reducing dramatically the amount of required memory. Our experimental results have shown that our method performs much better than both FP-Growth and Can-Tree in terms of memory usage and is comparable to them in terms of runtime.

Chapter 5

Upper Bounds for Alphabetical and FP-trees

Two efficient tree structures known as alphabetical based (Can-tree) [45] and FP-tree based [31] are used to store a database in memory for mining the frequent patterns. However, there has been no discussion on a tight upper bound for the number of nodes and the memory requirements for these trees. In the literature, an upper bound of 2^n is used, where n is the number of distinct items in the database. In this chapter, we provide an upper bound for the number of nodes in alphabetical and FP-trees. The upper bound on the number of nodes is provided in the context of a greedy algorithm for the alphabetical tree structure while a closed form solution for the FP-tree is derived. These results are illustrated in the context of various examples both in graphical and mathematical forms. We anticipate these upper bounds would be useful in various applications.

The structure of the chapter is as follows. Section 5.1 provides the basic terminology that is used in this chapter. Section 5.2 follows up with an algorithm for the Alphabetical tree upper bound. In this section, we provide an algorithm that calculates this upper bound in a systematic way. In Section 5.4 we derive a

closed formula to find an upper bound on the number of nodes in FP-tree, this derivation is shown in the context of an illustrative example. Finally, Section 5.5 provides the concluding remarks of this research.

5.1 PRELIMINARIES

Definition 5.1. *Recall Definition 2.1 in chapter 2.*

Definition 5.2. Database $\mathcal{D}$ *is a sequence of transactions:* $\mathcal{D} = \langle \mathcal{T}_1, \ldots, \mathcal{T}_m \rangle$ *and* total count *or* $TC = \sum_{i=1}^{m} |\mathcal{T}_i|$. *Also adding transaction* $\mathcal{T}_x$ *to database* $\mathcal{D}$ *is shown as:* $\mathcal{D}; \mathcal{T}_x = \langle \mathcal{T}_1, \ldots, \mathcal{T}_m, \mathcal{T}_x \rangle$.

Definition 5.3. $<_A$ *represents item ordering as alphabetical name order of the items (e.g.* $a <_A b, b <_A c$*).*

Definition 5.4. $<_{\mathsf{Freq}}$ *represents item ordering as frequency descending order of the items in database, defined below:*

$$a <_{\mathsf{Freq}} b \Leftrightarrow (\mathsf{Freq}_{\mathcal{D}}(a) > \mathsf{Freq}_{\mathcal{D}}(b) \text{ or } (\mathsf{Freq}_{\mathcal{D}}(a) = \mathsf{Freq}_{\mathcal{D}}(b) \text{ and } a <_A b))$$

$$\text{where } a, b \in A \text{ and } \mathcal{D} = \langle \mathcal{T}_1, \ldots, \mathcal{T}_m \rangle,\ a \in A, i \in [1, m] \text{ and } \mathsf{Acc}(a; \mathcal{T}_i) = \begin{cases} 1 & a \in \mathcal{T}_i \\ 0 & a \notin \mathcal{T}_i \end{cases}$$

$$\mathsf{Freq}_{\mathcal{D}}(a) = \sum_{i=1}^{m} \mathsf{Acc}(a; \mathcal{T}_i)$$

$\mathsf{Freq}_{\mathcal{D}}(a)$*, also called support count of* a *in* $\mathcal{D}$*, is the number of transactions that contains* a *in database* $\mathcal{D}$ *(we omit the subscription* $\mathcal{D}$ *when it is clear from the context).*

Table 5.1: *Transaction Database*

tid	Content	Sort Based on Frequency descending order $<_{\mathsf{Freq}}$ (FP-tree)	Sort Based on Alphabetical order $<_A$ (Can-tree)
1	$\{a, d, b, g, e, c\}$	$\langle b, c, a, e, d, g\rangle$	$\langle a, b, c, d, e, g\rangle$
2	$\{b, f, c, a, e\}$	$\langle b, c, a, e, f\rangle$	$\langle a, b, c, e, f\rangle$
3	$\{b\}$	$\langle b\rangle$	$\langle b\rangle$
4	$\{d, b\}$	$\langle b, d\rangle$	$\langle b, d\rangle$
5	$\{a, c, b\}$	$\langle b, c, a\rangle$	$\langle a, b, c\rangle$
6	$\{c, b, e\}$	$\langle b, c, e\rangle$	$\langle b, c, e\rangle$
7	$\{b, c\}$	$\langle b, c\rangle$	$\langle b, c\rangle$
8	$\{c, b\}$	$\langle b, c\rangle$	$\langle b, c\rangle$

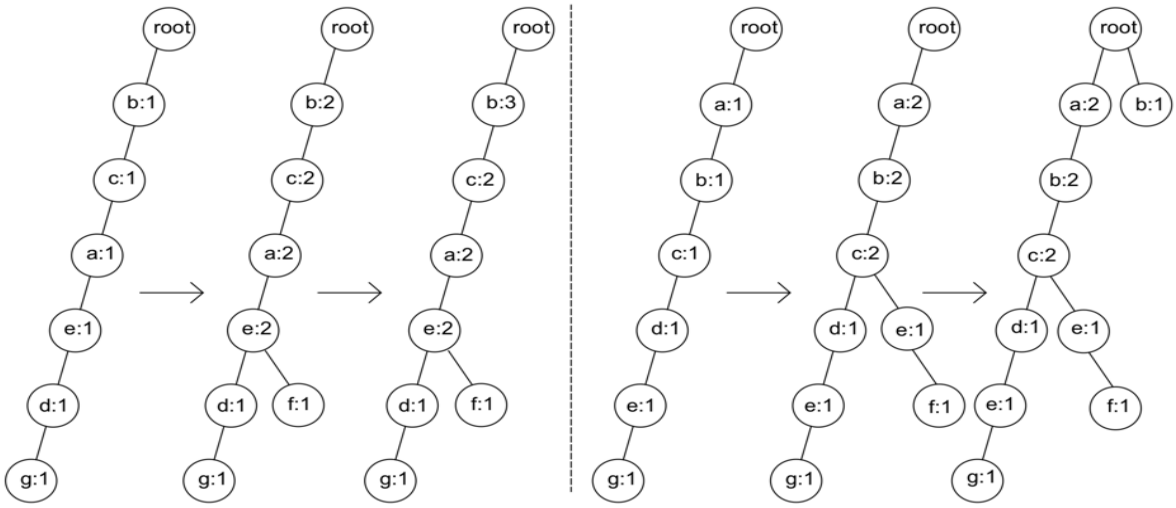
In order to represent a transaction database with a Can-tree/FP-tree structure, transactions are read one at a time with a predefined item order ($<_A$ item ordering for Can-tree and $<_{\mathsf{Freq}}$ item ordering for FP-tree) and mapping each transaction onto a path in the tree from the root (prefix merging). Since different transactions can have several items in common, their paths may overlap. Each node in the tree consists of two fields: *item-name : count*, where *item-name* registers which item this node represents and *count* registers the number of transactions represented by the portion of the path reaching this node. When a new node is added in the tree the *count* field is initialized to 1, and when two prefixes are merged, the *count* field increases by 1. The following example illustrates the Can-tree/FP-tree construction from a database.

Example 5.1. *Suppose that the transaction database, DB, be the first two columns of Table 5.1. We have* $N = 7$ *distinct items* $A = \{a, b, c, d, e, f, g\}$*, and the database consists of 8 transactions. A scan of DB derives the support count (*Freq*) of each item as follows;* $\mathsf{Freq}(b) = 8, \mathsf{Freq}(c) = 6, \mathsf{Freq}(a) = 3, \mathsf{Freq}(e) = 3, \mathsf{Freq}(d) = 2, \mathsf{Freq}(f) = 1$ *and* $\mathsf{Freq}(g) = 1$*, with* $TC = 24$*. Hence, the FP-tree item ordering will be:*

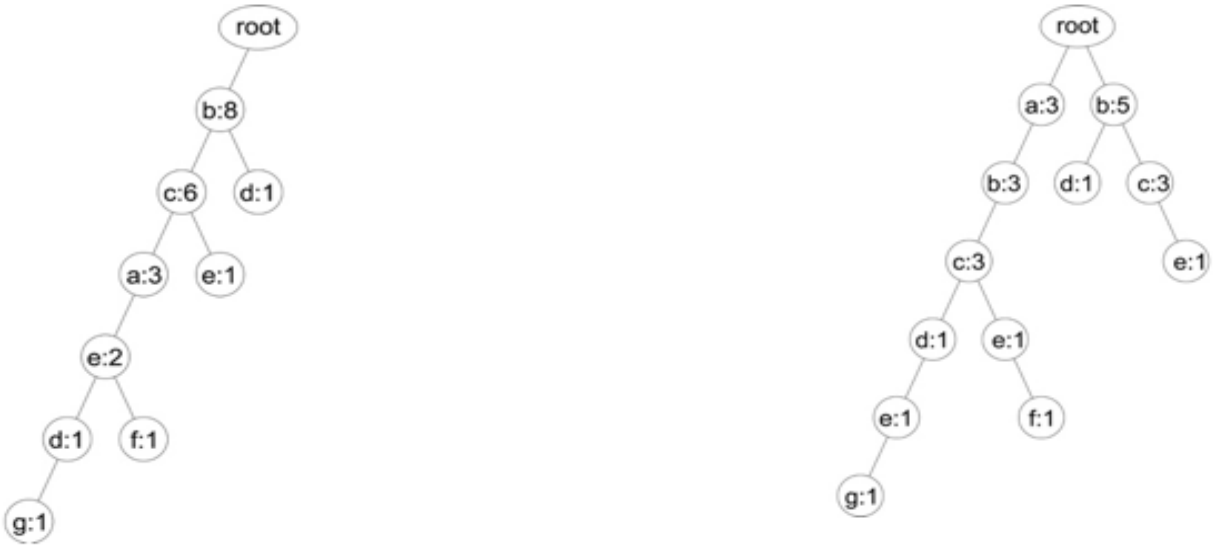
$b <_{\mathsf{Freq}} c <_{\mathsf{Freq}} a <_{\mathsf{Freq}} e <_{\mathsf{Freq}} d <_{\mathsf{Freq}} f <_{\mathsf{Freq}} g$. *Also, alphabetical tree item ordering will be:* $a <_A b <_A c <_A d <_A e <_A f <_A g$. *For constructing a FP-tree/Can-tree, first we need to create ordered transactions based on* $<_{\mathsf{Freq}}$ *and* $<_A$*, as shown in third and fourth column of Table 5.1 (i.e.* $\{a, d, b, g, e, c\} \rightarrow < b, c, q, e, d, g >$ */* $< a, b, c, d, e, g >$*) respectively, and then each transaction is attached to the root of the tree with prefix merging.*

Consider adding the first two transaction for Can-tree as shown in Figure 5.1a (right). First note that $\langle a, b, c \rangle$ *is a common prefix of the first two transactions* $\langle a, b, c, d, e, g \rangle$ *and* $\langle a, b, c, e, f \rangle$*. Adding the first transaction* $\langle a, b, c, d, e, g \rangle$ *results in 6* new nodes *in the tree* (a, b, c, d, e, g)*. Therefore, the* count *field will be 1 for all of them as shown in Figure 5.1a. The second transaction* $\langle a, b, c, e, f \rangle$*, has prefix* $\langle a, b, c \rangle$ *sharing with the first transaction. So, adding the second transaction will increase the* count *field by 1 (now* count $=$ 2*) for the prefix items in the tree (we call these nodes* merged *or* common *nodes). Also, two* new nodes e *and* f *with* count *field equal to 1 are added to the tree. Hence the first transaction which has 6 items, adds 6 new nodes to the tree while the second transaction which has 5 items, only adds 2 new nodes to the tree. As a whole by adding first two transactions, the tree would have 8 nodes. Figure 5.1a. shows a step by step procedure for adding the first three transactions and, Figure 5.1b. shows the complete tree for FP-tree and Can-tree, respectively.*

A set with n distinct items has 2^n subsets. Therefore, for a transaction database with n distinct items, the number of nodes in the tree representation with prefix-merging (which it is called prefix-tree) is bounded by 2^n [31] (The 2^n upper bound is called a loose upper bound in the rest of the chapter. For example, with 100 distinct items the loose upper bound of the prefix-tree is $2^{100} \approx 10^{30}$!). The intuition behind that is if A has n items then the transaction database which has *all subsets* of A (i.e., the power-set of A) has 2^n distinct transactions (counting null). Hence, in case of *all subsets* are attached to the root of the tree (with a predefined order) then the tree will have 2^n nodes, and is called a *complete prefix-tree*. If the $<_A$ item ordering is used for a complete prefix-tree, we call the tree a *complete alphabetical prefix-tree* and if the $<_{\mathsf{Freq}}$ item ordering is used for a complete prefix-tree, we call it a *complete FP prefix-tree*.



(a) *Adding $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ from left to right for FP-tree (left) and Can-tree (right) with prefix-merging.*



(b) *Complete FP-tree (left) Can-tree (right) based on Table 5.1*

Figure 5.1: *Constructing FP-tree and Can-tree from Table 5.1*

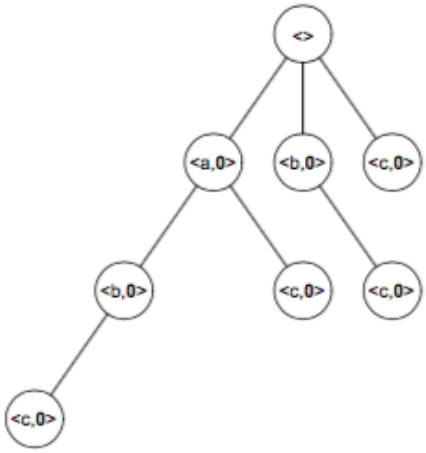


Figure 5.2: *Alphabetical layout-tree on* $A = \{a, b, c\}$ *(total* $2^3 = 8$ *nodes).*

Definition 5.5. Alphabetical layout-tree ($\mathbb{T}_{Layout(A,<_A)}$) *is complete alphabetical prefix-tree on* A*, where for each item we add another field called* count *with 0 value.*

Alphabetical layout-tree for $A = \{a, b, c\}$ *is shown in Figure 5.2. For simplicity we will remove* $<_A$ *from Alphabetical layout-tree notation when it is clear from the context and represent it as* $\mathbb{T}_{Layout(A)}$*. Consequently, if the* $<_{\mathsf{Freq}}$ *item ordering is used in this definition then we call it* FP layout-tree*.*

Definition 5.6. *Suppose that database* $\mathcal{D}$ *on a set of items* A *with the* $<_A$ *item ordering is given. Alphabetical Prefix-tree* $\mathbb{T}_{\mathcal{D},<_{\mathbb{A}}}$ *on database* $\mathcal{D}$*, can be defined as follows:*

1. *Root node is labeled by* $\langle\rangle$ *and each non-root node is labeled by* $\langle a, \textbf{\textit{count}}\rangle$ $(a \in A,\ \textbf{\textit{count}} \in \mathbb{N})$*.*

2. *if* $\mathcal{D} = \varnothing$ *then* $\mathbb{T}_{\mathcal{D},<_{\mathbb{A}}} = \mathbb{T}_{Layout(A)}$*.*

3. *Read each transaction* $\mathcal{T}$ *(ordered by* $<_A$*) from* $\mathcal{D}$ *one at a time and do the following:*

   Find a path *in the tree starting from the root with the same ordered items as in* $\mathcal{T}$*, and increase the items'* ***count*** *by 1.*

4. *Remove nodes with count = 0 in the tree.*

Note that:

- *for simplicity we will remove (Alphabetical) / $<_A$ from Alphabetical Prefix-tree / $\mathbb{T}_{\mathcal{D},<_{\mathbb{A}}}$ notation when its clear from the context, and will represent it as $\mathbb{T}_{\mathcal{D}}$.*
- *Consequently, if the $<_{\mathsf{Freq}}$ item ordering is used in this definition, we call it* FP Prefix-tree or FP-tree for short*.*

Below is an example for creating Alphabetical prefix-tree based on the above definition.

Example 5.2. *Suppose that $A = \{a, b, c\}$, $\mathcal{D} = \langle \mathcal{T}_1, \mathcal{T}_2 \rangle$, $\mathcal{T}_1 = \langle a, b, c \rangle, \mathcal{T}_2 = \langle a, c \rangle$ then Alphabetical Prefix-tree $\mathbb{T}_{\mathcal{D}}$ is created from definition 5.6 in two steps:*

- *Step1: Alphabetical Layout-tree is created as shown in Figure 5.2.*
- *Step2: Transactions are read one at a time:*
  1. *$\mathcal{T}_1 = \langle a, b, c \rangle$ is parsed, then $Path_1$ is found in the tree which has the same ordered items as in $\mathcal{T}_1$, finally the items' count increased by 1 as shown in Figure 5.3.*
  2. *$\mathcal{T}_2 = \langle b, c \rangle$ is parsed, then $Path_2$ is found in the tree which has the same ordered items as in $\mathcal{T}_2$. Finally the items' count increased by 1 as shown in Figure 5.4.*
- *Step3: Nodes with item count = 0 is removed from the tree as shown in Figure 5.5.*

Definition 5.7. *Below are some definitions on (Alphabetical/FP) prefix-tree:*

- *count = 0 means this node is not in any transactions (we called this node **unused** node) and will be removed.*
- *If count = 1, we call this node **new** node.*
- *If count > 1 we call this node **merged** or **common** node.*

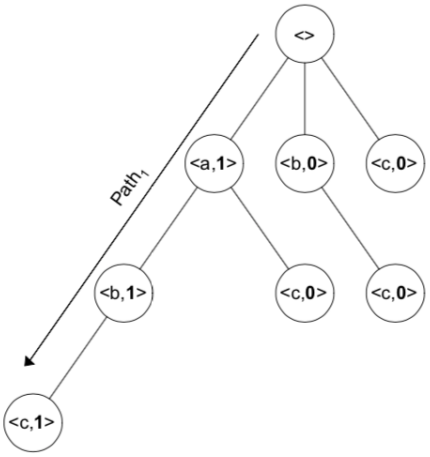


Figure 5.3: *Adding $\langle a, b, c\rangle$ to the layout-tree of Example 5.2*

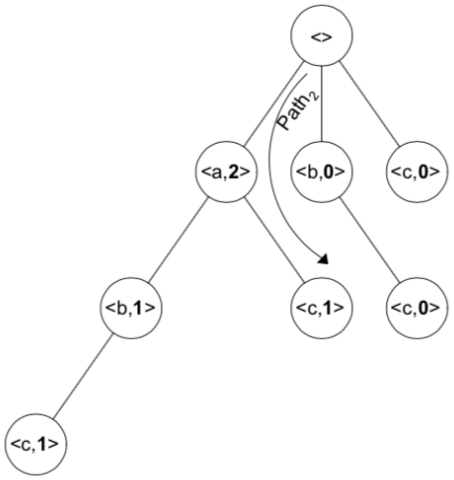


Figure 5.4: *Adding $\langle a, c\rangle$ to the tree of Figure. 5.3*

- *Size of the prefix-tree: $|\mathbb{T}_{\mathcal{D}}| = |\{\langle a, count\rangle \in \mathbb{T}_{\mathcal{D}} \mid count \neq 0\}| + 1$, which is number of nonzero labeled nodes plus one for the root. We can also say $|\mathbb{T}_{\mathcal{D}}| = 2^n - |\{\langle a, count\rangle \in \mathbb{T}_{\mathcal{D}} \mid count = 0\}|$ where n is the size of items. Therefore, $|\mathbb{T}_{\mathcal{D}}| = 4 + 1 = 5$ or $|\mathbb{T}_{\mathcal{D}}| = 2^3 - 3$ (unused node)=5.*

In the following sections, we find the upper bound procedure of the number of nodes in the tree for Can-tree and closed form upper bound of the number of nodes in the tree for FP-tree, given n and TC.

5.2 ALPHABETICAL-TREE UPPER BOUND PROCEDURE

In this section wherever we use layout-tree we mean alphabetical layout-tree, also we will remove $<_A$ ordering from notations.

Lemma 5.1. *if $\mathcal{D}_1 = \langle \mathcal{T}_1, \ldots, \mathcal{T}_m\rangle$ and $\mathcal{D}_2 = \langle \mathcal{R}_1, \ldots, \mathcal{R}_m\rangle$ and $\{\mathcal{T}_1, \ldots, \mathcal{T}_m\} = \{\mathcal{R}_1, \ldots, \mathcal{R}_m\}$ then $\mathbb{T}_{\mathcal{D}_1} = \mathbb{T}_{\mathcal{D}_2}$.*

Proof. Straightforward. ■

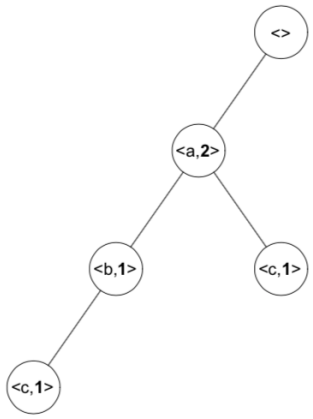


Figure 5.5: *Final alphabetical prefix-tree, after removing unused nodes (count = 0) of Example 5.2*

Lemma says that the prefix-tree created from transactions is independent of transaction's order. From definition 5.6 each transaction will find its path and increase the *count* field for its corresponding path in the tree, and this process is clearly independent of their ordering. Considering example 5.2, the prefix-tree constructed from $\langle \mathcal{T}_1, \mathcal{T}_2 \rangle$ is clearly the same as $\langle \mathcal{T}_2, \mathcal{T}_1 \rangle$.

Note: As the transaction ordering in database does not matter for creating the prefix-tree, we may also show database as a set of transactions in this study.

In order to illustrate the upper bound procedure, we start with a small value for number of distinct items in a database, although the loose upper bound based on small value is not large, we use these small examples to construct an algorithm for tighter upper bound based on TC and n that could be generalized for any values of these parameters.

Example 5.3. *What is the Can-tree upper bound for $n = 3$ and $TC = 5$?*

For $n = 3$ we have $A = \{a, b, c\}$ and suppose that we have two different databases $\mathcal{D}_1 = \langle \mathcal{T}_{11}, \mathcal{T}_{12} \rangle$ and $\mathcal{D}_2 = \langle \mathcal{T}_{21}, \mathcal{T}_{22} \rangle$. Where:

1. $\mathcal{T}_{11} = \langle a, b, c \rangle$, $\mathcal{T}_{12} = \langle b, c \rangle$

2. $\mathcal{T}_{21} = \langle a, b, c \rangle$, $\mathcal{T}_{22} = \langle a, b \rangle$

Can-tree constructed from $\mathcal{D}_1$ and $\mathcal{D}_2$ are shown in Figure 5.6 and Figure 5.7 respectively. Can-tree based on $\mathcal{D}_1$ has 6 nodes and based on $\mathcal{D}_2$ has 4 nodes (counting root). For this small example one can confirm that maximum number of nodes in Can-tree with $n = 3$ and $TC = 5$ is 6. Hence, Can-tree based on the first database creates the maximum number of nodes in the tree (so the upper bound is equal to 6). But in the second database, $\mathcal{T}_{22}$ transaction did not create a new node in the tree and two previous nodes (a and b) are merged (prefix-merging). Therefore, if the canonical ordering of transactions in $\mathcal{D}_1$ is used, the maximum number of nodes appeared in the tree (hence, the upper bound achieved). Therefore, we are looking for a canonical order of transactions which creates maximum number of nodes in the tree based on n and TC. *TC is our total budget and ideally, we want to use all of the budget to create new nodes in the tree, therefore at some point, we are not able to create a new node and we are forced to have some merged nodes in the tree.*

Upon adding a transaction with size k to the tree; new nodes are created, and some previously created nodes may merge in the tree, the total number of new nodes added to the tree is represented by w and total nodes shared in the tree is represented by c ($k = w + c$). For example by adding $\mathcal{T}_{11}$ where $k = 3$, we have $w = 3$ and $c = 0$, and by adding $\mathcal{T}_{22}$ where $k = 2$, we have $w = 0$ and $c = 2$. Also we have $\{0 \leq w, c \leq n \text{ and } 0 < w + c \leq n\}$ where n is A size.

For the upper bound procedure of Can-tree we propose our transactions' ordering and later in this section we show that this transactions' ordering will results in the biggest tree possible (maximum number nodes).

Example 5.4. *Suppose that we want to find the upper bound for alphabetical tree, based on $n = 5$, $A = \{a, b, c, d, e\}$ and $TC = 19$.*

Layout-tree for this example has $2^5 = 32$ nodes. *To reduce the complexity in figures, we remove* count *field in the nodes, and when* count *field of a node becomes greater than 1, that node becomes* ***bold****.* Figure. 5.8

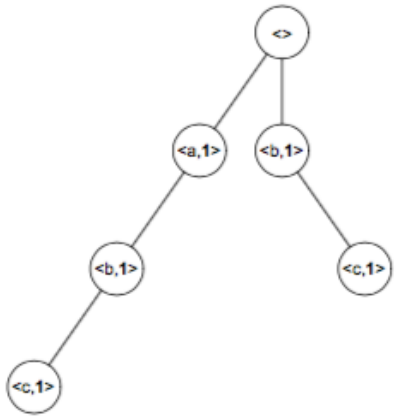


Figure 5.6: *Can-tree of Example 5.3, based on* $\mathcal{D}_1$

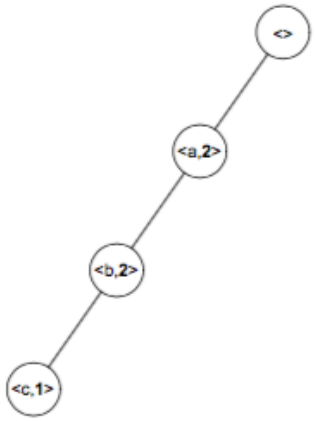


Figure 5.7: *Can-tree of Example 5.3, based on* $\mathcal{D}_2$

shows the layout-tree on $A = \{a, b, c, d, e\}$.

To construct our canonical order of transactions, we start with layout-tree and construct a greedy algorithm that finds transaction one at a time with the following *goal*.

Greedy algorithm goal: Find a *transaction* that adds more new nodes (represented by w) while keeping the number of common nodes (represented by c) minimal to the current tree structure. In order to achieve this goal, we define a measure that quantifies our objective as $\frac{w}{c}$. *The bigger the fraction we have, the more nodes are added to the tree.* If we are in a position to compare two fractions with $c = 0$, we choose the fraction with larger numerator w to be the larger fraction (because it will add more new nodes to the tree). Definition 5.8 shows this comparison formally, and based on this we will show how we will choose which transaction to pick next.

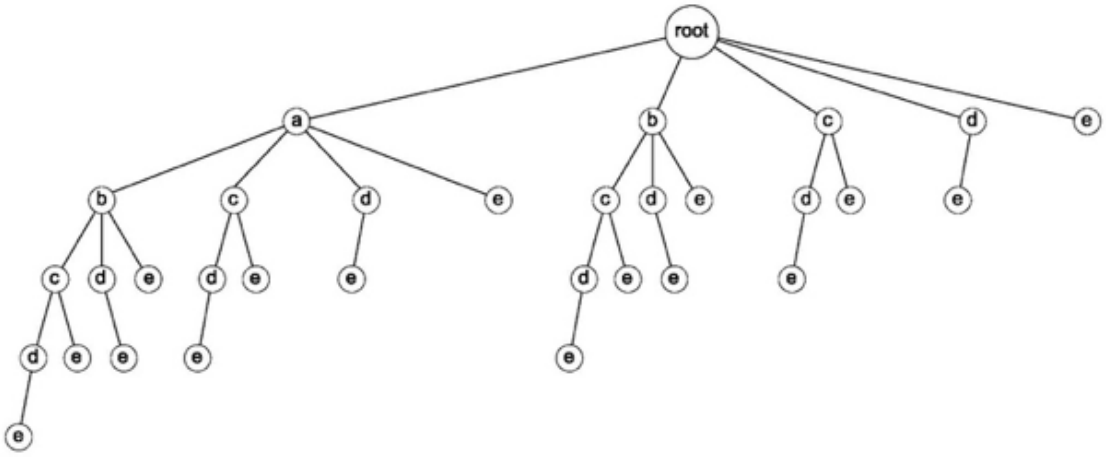


Figure 5.8: *Layout-tree on* $A = \{a, b, c, d, e\}$.

Definition 5.8. *For* $\langle w_1, c_1\rangle$ *and* $\langle w_2, c_2\rangle$ *where* w_1, w_2, c_1, c_2 *are nonnegative integers, let*

$$\langle w_1, c_1\rangle \succ \langle w_2, c_2\rangle \iff \begin{cases} \textit{any} & c_1 = 0 \textit{ and } c_2 \neq 0 \ ^* \\ w_1 > w_2 & c_1 = 0 \textit{ and } c_2 = 0 \\ \frac{w_1}{c_1} > \frac{w_2}{c_2} \textit{ or } (\frac{w_1}{c_1} = \frac{w_2}{c_2} \textit{ and } w_1 > w_2) & c_1 \neq 0 \textit{ and } c_2 \neq 0 \end{cases}$$

equivalently

$$\langle w_1, c_1\rangle \succ \langle w_2, c_2\rangle \iff \langle (1 - \operatorname{sgn}(c_1)), \frac{w_1}{c_1}, w_1\rangle \gg \langle 1 - \operatorname{sgn}(c_2), \frac{w_2}{c_2}, w_2\rangle$$

where $\gg$ *is lexicographic order* [2].

* *This means that if one of the c values is 0 and the other one is not, the pair with c value equal to 0 is* $\succ$ *than the other pair.*

$\succ$ is an order on pairs of nonnegative integers, and we call it **Corder**, it is clear that this order is a *total order* since when two fractions are equal, we chose the one with bigger numerator.

[2] *https : en.wikipedia.org/wiki/Lexicographical_order*

Now a set P is created on *all possible values* for w and c based on n as:

$$\mathsf{P} = \{\langle w, c\rangle \mid 0 \leq w, c \leq n \text{ and } 0 < w + c \leq n\}$$

where n is the size of A.

Example 5.5. *Suppose* $A = \{a_1, \ldots, a_5\}$ *where* $n = 5$. *Then:*

- $\mathsf{P} = \{\langle 5,0\rangle, \langle 4,1\rangle, \ldots, \langle 0,5\rangle, \langle 4,0\rangle, \langle 3,1\rangle, \ldots, \langle 0,4\rangle, \langle 3,0\rangle \ \ldots, \ldots, \langle 1,0\rangle, \langle 0,1\rangle\}$, *below are some examples:*
- $\langle 5,0\rangle \succ \langle 4,1\rangle$,
- $\langle 4,1\rangle \succ \langle 3,2\rangle$, *and*
- $\langle 2,0\rangle \succ \langle 4,1\rangle$.

We also need to assign a number for each pair in P that represent its ordering position with respect to $\succ$, defined as: $\ell(\langle w, c\rangle) = |\{\langle w', c'\rangle \mid \langle w', c'\rangle \succ \langle w, c\rangle\}|$.

Example 5.6. *By using the same items in Example 5.5, below are some examples for the* ℓ *value.*

- $\ell(\langle 5,0\rangle) =$ ***0****, there is* ***no*** *pair Corder than* $\langle 5,0\rangle$.
- $\ell(\langle 4,0\rangle) =$ ***1****, there is* ***1 pair*** *Corder than* $\langle 4,0\rangle$ *which is* $\langle 5,0\rangle$.
- $\ell(\langle 3,0\rangle) =$ ***2****, there are* ***2 pairs*** *Corder than* $\langle 3,0\rangle$ *which are* $\langle 5,0\rangle$ *and* $\langle 4,0\rangle$.

then we can define:

$$\mathsf{W}(pos) = \pi_1(\ell^{-1}(pos)) \text{ and } \mathsf{C}(pos) = \pi_2(\ell^{-1}(pos)), \text{ where } pos \geq 0$$

where π_1 and π_2 are projections on first and second elements in the pair, respectively. For example:

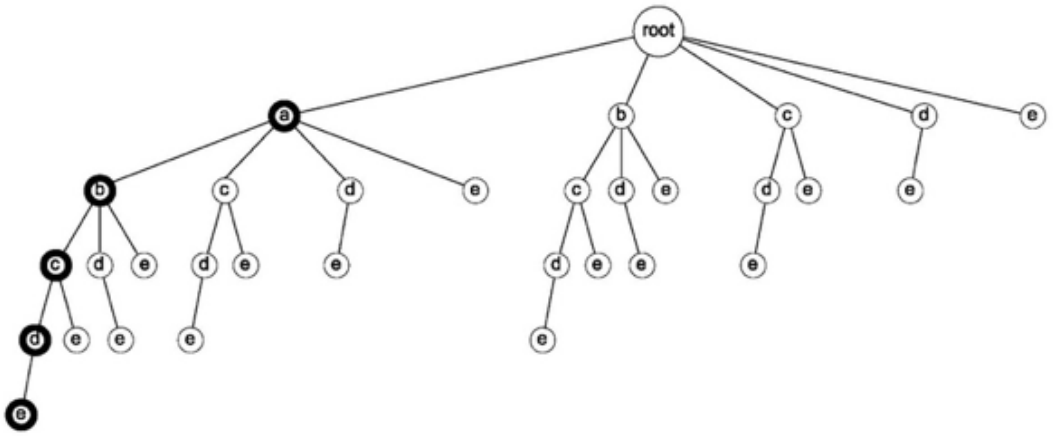


Figure 5.9: *Parsing transaction* $\mathcal{T}_1 : \langle a, b, c, d, e \rangle$.

- at $pos = 0$, $\mathsf{W}(0) = 5$ and $\mathsf{C}(0) = 0$,
- at $pos = 1$, $\mathsf{W}(1) = 4$ and $\mathsf{C}(1) = 0$, and
- at $pos = 2$, $\mathsf{W}(2) = 3$ and $\mathsf{C}(2) = 0$.

Consider again Example 5.3. To find the upper bound, first we start with layout-tree (no transaction added yet), and based on the greedy algorithm goal, we want to find a transaction with greater Corder pairs.

For this goal, all we need to do is to find the transaction which adds $\mathsf{W}(pos)$ new nodes, with $\mathsf{C}(pos)$ common nodes to the tree (starting from $pos = 0$ and increases it by one each time). Start at $pos = 0$, we are after transaction(s) which adds $\mathsf{W}(0) = w = 5$ new nodes with $\mathsf{C}(0) = c = 0$ common nodes to the current tree structure. Transaction $\mathcal{T}_1 = \langle a, b, c, d, e \rangle$ will add 5 new nodes to the layout-tree with 0 common nodes ($k = w = 5, c = 0$) as shown in Figure 5.9 (one could verify that there is no transaction that can add more nodes to the tree). Each time we add a transaction to the tree, we subtract transaction size from our budget TC and add w new nodes to U (which is initialized to 0) until TC reaches 0. Therefore, $TC = 19 - 5 = 14$ and $U = U + 5 = 5$. For now suppose that we add the transaction(s) to our ordering sequence represented by $\langle \mathcal{S}_{pos} \rangle$ ($0 \leq pos$). Therefore , $\mathcal{S}_0 = \mathcal{T}_1$ and the current sequence is: $\langle \mathcal{S}_0 \rangle$.

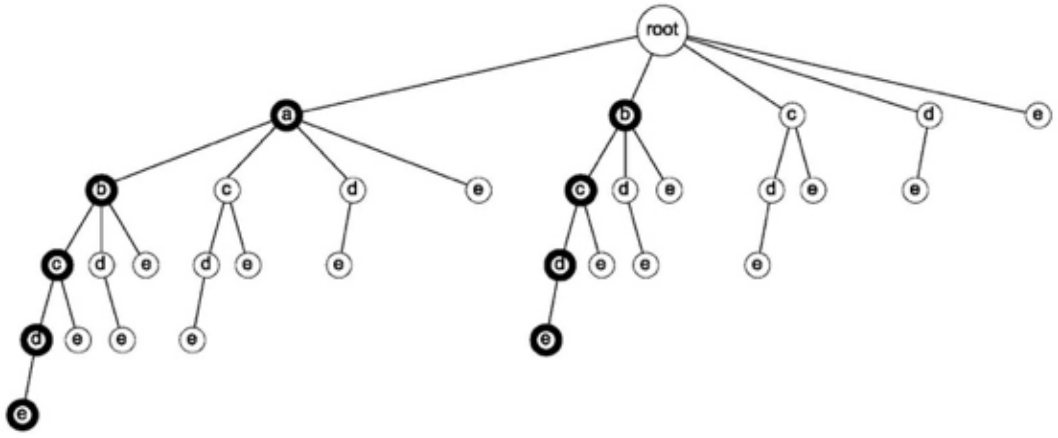


Figure 5.10: *Parsing transaction* $\mathcal{T}_2 : \langle b, c, d, e \rangle$.

At $pos = 1$ we are after transaction(s) which adds $\mathsf{W}(1) = w = 4$ new nodes with $\mathsf{C}(1) = c = 0$ common nodes to the current tree structure. Transaction $\mathcal{T}_2 = \langle b, c, d, e \rangle$, will add 4 new nodes to the current tree with 0 common nodes ($k = w = 4, c = 0$) as shown in Figure 5.10. One could verify that there is no transaction that can add more nodes to the tree. Also, $TC = 14 - 4 = 10$ and $U = U + 4 = 9$, $\mathcal{S}_1 = \mathcal{T}_2$. Now current ordering is:$\langle \mathcal{S}_0, \mathcal{S}_1 \rangle$.

At $pos = 2$ we are after transaction(s) which adds $\mathsf{W}(2) = w = 3$ new nodes with $\mathsf{C}(2) = c = 0$ common nodes to the current tree structure. Transaction $\mathcal{T}_3 = \langle c, d, e \rangle$, will add 3 new nodes to the current tree with 0 common nodes ($k = w = 3, c = 0$) as shown in Figure 5.11. One could verify that there is no transaction that can add more nodes to the tree. Also, $TC = 10 - 3 = 7$ and $U = U + 3 = 12$, $\mathcal{S}_2 = \mathcal{T}_3$. Now current ordering is:$\langle \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2 \rangle$.

With the same reasoning, the next transaction is $\mathcal{T}_4 = \langle d, e \rangle$, which will add 2 new nodes to the current tree with 0 common nodes ($k = w = 2, c = 0$) as shown in Figure 5.12. Also, $TC = 7 - 2 = 5$ and $U = U + 2 = 14$, $\mathcal{S}_3 = \mathcal{T}_4$. Now current ordering is:$\langle \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3 \rangle$.

The next transaction is $\mathcal{T}_5 = \langle e \rangle$, which will add 1 new node to the current tree with 0 common nodes ($k = w = 1, c = 0$) as shown in Figure 5.13. Also, $TC = 5 - 1 = 4$ and $U = U + 1 = 15$, $\mathcal{S}_4 = \mathcal{T}_5$. Now

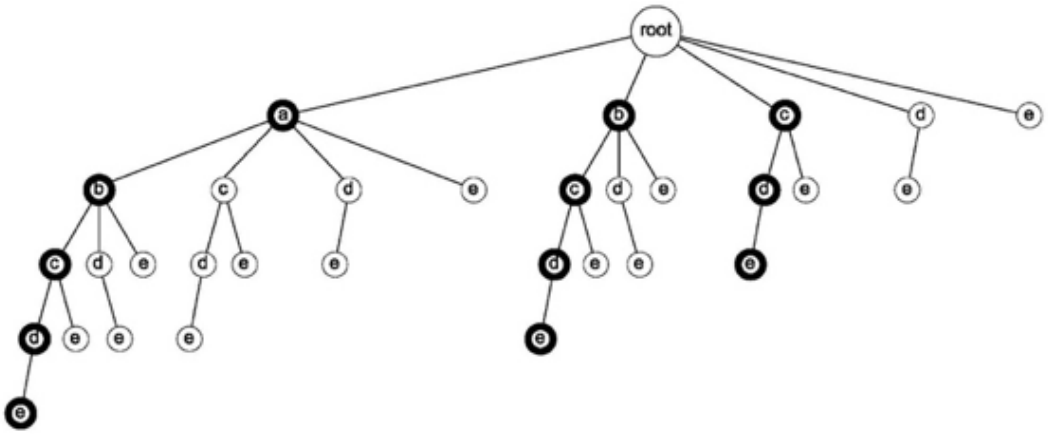


Figure 5.11: *Parsing transaction* $\mathcal{T}_3 : \langle c, d, e \rangle$.

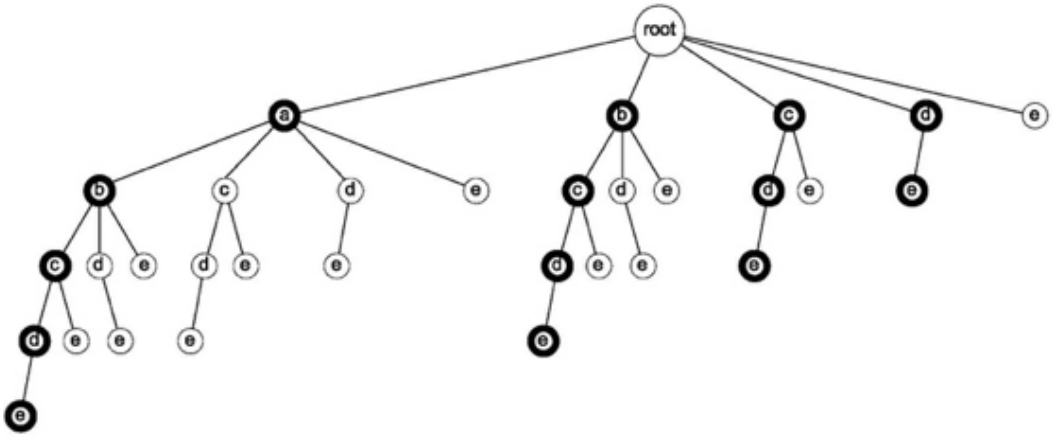


Figure 5.12: *Parsing transaction* $\mathcal{T}_4 : \langle d, e \rangle$.

current ordering is:$\langle \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4 \rangle$.

From now on, adding any transaction to the database, has to have at least one common node in the tree.

The next transaction is $\mathcal{T}_6 = \langle a, c, d, e \rangle$, which will add 3 new node to the current tree with 1 common nodes ($k = 4, w = 3, c = 1$) as shown in Figure 5.14 (there is no other transaction with bigger $\frac{w}{c}$). Also, $TC = 4 - 4 = 0$ and $U = U + 3 = 18$, $\mathcal{S}_5 = \mathcal{T}_6$. Now the ordering is:$\langle \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5 \rangle$. At this point our budget TC becomes 0 and we output $U = 18$ as upper bound for the Can-tree of Example 5.3. One could verify that this is the maximum size of Can-tree for this example.

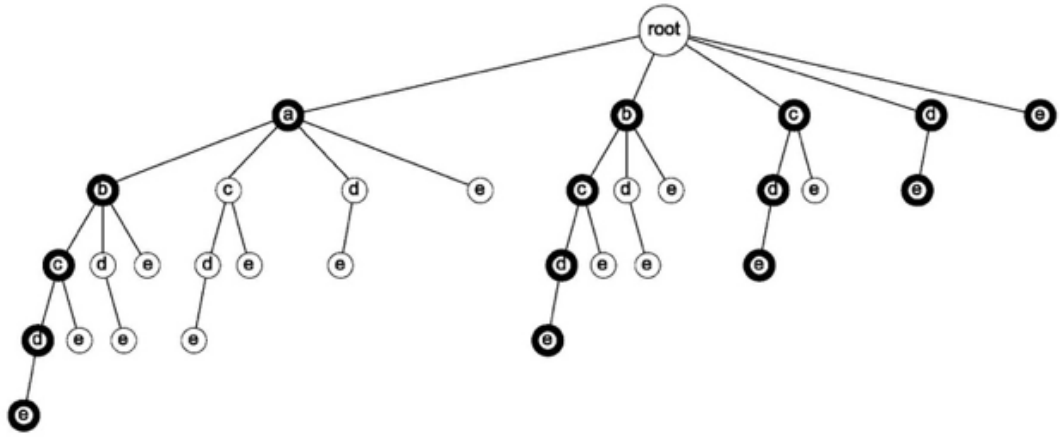


Figure 5.13: *Parsing transaction* $\mathcal{T}_5 : \langle e \rangle$.

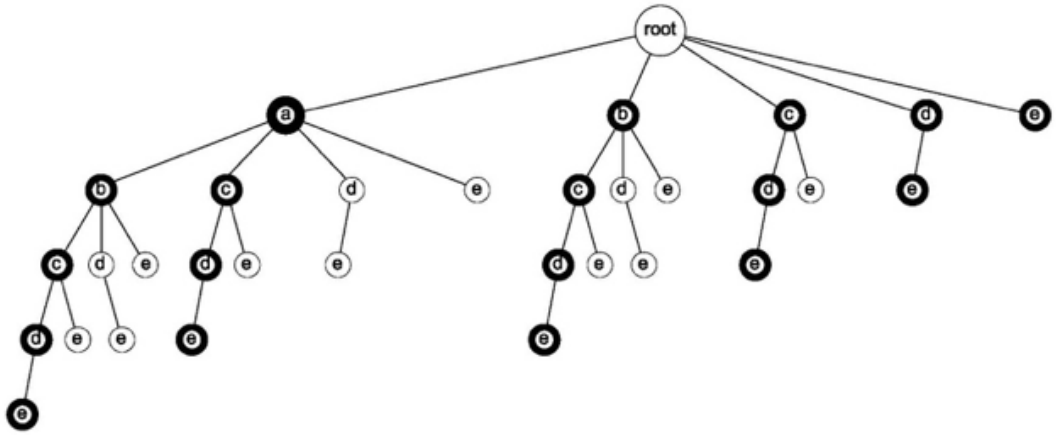


Figure 5.14: *Parsing transaction* $\mathcal{T}_6 : \langle a, c, d, e \rangle$ *with 1 common node.*

In general setting, in order to find all the transactions with their corresponding w and c, lets consider Table 5.2. In this table each column shows an item and each row indicates a transaction. The table shows which items appear in which transactions. If an item appears in a transaction the corresponding cell is 1 otherwise it is 0. Cells that are filled with dashed line indicate the corresponding item may or may not appear in the specified transaction.

Parsing Table 5.2 row by row, for all the transactions and their corresponding $\frac{w}{c}$, will lead to the following observations:

Table 5.2: *Items in transactions for Example 5.3.*

Tran\items	a	b	c	d	e
Trans1	1	1	1	1	1
Trans2	0	1	1	1	1
Trans3	—	0	1	1	1
Trans4	—	—	0	1	1
Trans5	—	—	—	0	1
Trans6	—	—	—	—	0

Row 1, which corresponds to transaction $\mathcal{T}_1 = \langle a, b, c, d, e\rangle$, with $\frac{w}{c} = \frac{5}{0}$.

Row 2, which corresponds to transaction $\mathcal{T}_2 = \langle b, c, d, e\rangle$, with $\frac{w}{c} = \frac{4}{0}$.

Row 3, with $a = 0$ corresponds to $\mathcal{T}_3 = \langle c, d, e\rangle$, with $\frac{w}{c} = \frac{3}{0}$. If $a = 1$, this row corresponds to $\langle a, c, d, e\rangle$ with $\frac{w}{c} = \frac{4-1}{1}$.

Row 4, with $a = 0$, $b = 0$ corresponds to transaction $\mathcal{T}_2 = (d, e)$, with $\frac{w}{c} = \frac{2}{0}$. There are 3 more transactions in this row ($a = 1, b = 0$ or $a = 0, b = 1$ or $a = 1, b = 1$). If $a = 1$, $b = 0$ this row corresponds to $\langle a, d, e\rangle$, with $\frac{w}{c} = \frac{3-1}{1}$. If $a = 0$, $b = 1$ this row corresponds to $\langle b, d, e\rangle$, with $\frac{w}{c} = \frac{3-1}{1}$. Finally if $a = 1$, $b = 1$ this row corresponds to $\langle a, b, d, e\rangle$, with $\frac{w}{c} = \frac{4-2}{2}$.

$\frac{w}{c}$s in this row are $\frac{2}{0}$, $\frac{(4-2)}{2}$, $\frac{(3-1)}{1}$ and $\frac{(3-1)}{1}$. We can recount them as: $\frac{2}{0}$, $\binom{2}{2}\frac{(4-2)}{2}$ and $\binom{2}{1}\frac{(3-1)}{1}$.

Note that the *combinatorial coefficient represents* ***only*** *the number of times each* $\frac{w}{c}$ *fraction occurs.*

Row 5, if $a = 0$, $b = 0$, $c = 0$, corresponds to transaction $\mathcal{T}_1 = \langle e\rangle$,with $\frac{w}{c} = \frac{1}{0}$. There are 7 more transactions in this row depending on the values of a,b and c to be 0 or 1. With the same reasoning shown

Table 5.3: *$\frac{w}{c}$ values for all the transactions in Table 5.2*

Row 1					$\frac{5}{0}$
Row 2					$\binom{0}{0}\frac{4}{0}$
Row 3				$\binom{1}{1}\frac{4-1}{1}$	$\binom{0}{1}\frac{3}{0}$
Row 4			$\binom{2}{2}\frac{4-2}{2}$	$\binom{2}{1}\frac{3-1}{1}$	$\binom{0}{2}\frac{2}{0}$
Row 5		$\binom{3}{3}\frac{4-3}{3}$	$\binom{3}{2}\frac{3-2}{2}$	$\binom{3}{1}\frac{2-1}{1}$	$\binom{0}{3}\frac{1}{0}$
Row 6	$\binom{4}{4}\frac{4-4}{4}$	$\binom{4}{3}\frac{3-3}{3}$	$\binom{4}{2}\frac{2-2}{2}$	$\binom{4}{1}\frac{1-1}{1}$	$\binom{0}{4}\frac{0}{0}$

in Row 4, $\frac{w}{c}$s in this row are $\binom{3}{1}\frac{2-1}{1}$, $\binom{3}{2}\frac{3-2}{2}$ and $\binom{3}{3}\frac{4-3}{3}$. In this row the combinatorial coefficients are $\binom{3}{1}$, $\binom{3}{2}$ and $\binom{3}{3}$.

Row 6, There are 15 more transactions in this row depending on the values of a, b, c and d to be 0 or 1 (excluding all equal to 0). With the same reasoning as above, the $\frac{w}{c}$ values for all the possible transactions are shown in Table 5.3.

5.3 The Upper Bound Procedure for Alphabetical-tree

If we create the $\langle \mathcal{S} \rangle$ transactions' ordering in advance for A, all we need to do is to extract transactions one by one from the sequence and compute the U as illustrated in the above example. Therefore, to obtain an upper bound for n and TC (where n is the size of A), first, we order *all possible transactions* with respect to $\succ$ ordering of their corresponding w and c in a sequence represented by $\langle \mathcal{S} \rangle$. Then we start removing transactions one by one from our created sequence, and keep updating TC and U, until TC reaches 0 (as shown in the above example). In the next section we formally state this sequence of transactions and the algorithm. Finally we proof that the this assumption will lead to the biggest tree possible for Can-tree. But, first we need one more definition in order to formally define our canonical ordering of transaction.

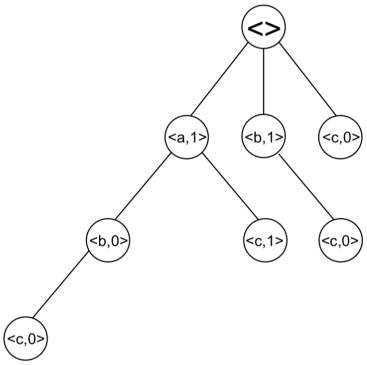


Figure 5.15: *$\mathbb{T}_{\mathcal{D}}$ of Example 5.7*

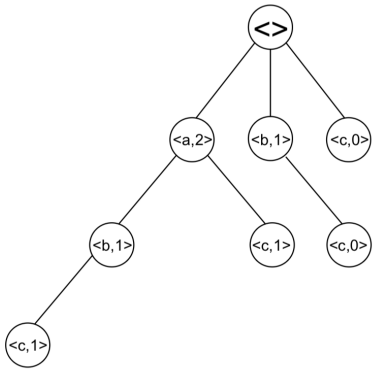


Figure 5.16: *adding $\mathcal{T}_3 = \langle a, b, c\rangle$ to $\mathbb{T}_{\mathcal{D}}$ results in $w = 2$ and $c = 1$*

In order to compute w and c as a result of adding transaction $\mathcal{T}$ to database $\mathcal{D}$, two functions called $\mathcal{W}$ and $\mathcal{C}$ are defined as follow:

Definition 5.9. *Suppose that for a database $\mathcal{D}$ we add a new transaction $\mathcal{T}$:*

- $\mathcal{W}(\mathcal{D}, \mathcal{T}) = |\mathbb{T}_{\mathcal{D};\mathcal{T}}| - |\mathbb{T}_{\mathcal{D}}|$,
- $\mathcal{C}(\mathcal{D}, \mathcal{T}) = |\mathcal{T}| - \mathcal{W}(\mathcal{D}, \mathcal{T})$.

$\mathcal{W}$ and $\mathcal{C}$ are computing w and c of the newly added transaction $\mathcal{T}$, respectively. The following example illustrate the above definition.

Example 5.7. *Suppose that $\mathcal{D} = \{\mathcal{T}_1, \mathcal{T}_2\}$, $\mathcal{T}_1 = \langle a, c\rangle, \mathcal{T}_2 = \langle b\rangle$. $\mathbb{T}_{\mathcal{D}}$ is shown in Figure 5.15 with size $|\mathbb{T}_{\mathcal{D}}| = 3$. By adding $\mathcal{T}_3 = \langle a, b, c\rangle$ to $\mathcal{D}$ the result tree is shown in Figure 5.16 with size $|\mathbb{T}_{\mathcal{D};\mathcal{T}_3}| = 5$, then we have:*

- $\mathcal{W}(D, \mathcal{T}_3) = |\mathbb{T}_{\mathcal{D};\mathcal{T}_3}| - |\mathbb{T}_{\mathcal{D}}| = 5 - 3 = 2$, *so $w = 2$* new *nodes added to the tree (b and c).*
- $\mathcal{C}(D, \mathcal{T}_3) = |\mathcal{T}_3| - \mathcal{W}(D, \mathcal{T}_3) = 3 - 2 = 1$ *, so adding $\mathcal{T}_3$ results in $c = 1$* common *node in the tree (node a with count=2).*

- *Also note that* $|\mathcal{T}_3|$ = new *nodes* + common *nodes* = *2 + 1 = 3*

Now we have all the ingredients to define our canonical ordering of transactions, which exactly creates the sequence described in Example 5.3. In this sequence, transactions are ordered based on the Corder defined in Definition 5.8.

Definition 5.10. *For* $A = \{a_1, \ldots, a_n\}$ *we define* $\langle \mathcal{S} \rangle$ *and functions* $kf()$ *and* $wf()$ *on this sequence simultaneously. For all* 2^n *transactions and non-negative integer pos,* $0 \leq pos$*:*

- $\mathcal{S}_{pos} = \{\mathcal{T} \mid \mathcal{W}(\mathcal{D}_{pos}, \mathcal{T}) = \mathsf{W}(pos)$ *and* $\mathcal{C}(\mathcal{D}_{pos}, \mathcal{T}) = \mathsf{C}(pos)\}$ *where* $\mathcal{D}_{pos} = \bigcup_{i<pos} \mathcal{S}_i$ *and* $\mathcal{D}_0 = \varnothing$
- $wf(\mathcal{S}_{pos}) = \mathsf{W}(pos)$
- $kf(\mathcal{S}_{pos}) = \mathsf{W}(pos) + \mathsf{C}(pos)$.

Note that $\mathcal{S}_{pos}$ *uses* $\mathcal{D}_{pos}$ *that depends on* $\mathcal{S}_0 \ldots \mathcal{S}_{pos-1}$. *Also it is clear that* $wf(.)$ *is a function that maps transactions to their corresponding* w *new nodes, and* $kf(.)$ *is a function that maps transactions to their corresponding size* k.

Bellow example better illustrate the definition.

Example 5.8. *Suppose that* $A = \{a_1, \ldots, a_5\}$.

Starting from $pos = 0$, $\mathsf{W}(0) = 5$ *and* $\mathsf{C}(0) = 0$*:*

- $\mathcal{S}_0 = \{\mathcal{T} \mid \mathcal{W}(\mathcal{D}_0, \mathcal{T}) = \mathsf{W}(0) = 5$ *and* $\mathcal{C}(\mathcal{D}_0, \mathcal{T}) = \mathsf{C}(0) = 0\}$ *where* $\mathcal{D}_0 = \varnothing$
- $wf(\mathcal{S}_0) = 5$
- $kf(\mathcal{S}_0) = 5 + 0$.

$\mathcal{S}_0$ has the first transaction which corresponds to $w = 5$ and $c = 0$, which is the transaction shown in Figure 5.9.

If pos $= 1$, $\mathsf{W}(1) = 4$ *and* $\mathsf{C}(1) = 0$*:*

- $\mathcal{S}_1 = \{\mathcal{T} \mid \mathcal{W}(\mathcal{D}_1, \mathcal{T}) = \mathsf{W}(1) = 4 \text{ and } \mathcal{C}(\mathcal{D}_1, \mathcal{T}) = \mathsf{C}(1) = 0\}$ *where* $\mathcal{D}_1 = \{\mathcal{S}_0\}$
- $wf(\mathcal{S}_0) = 4$
- $kf(\mathcal{S}_0) = 4 + 0$.

$\mathcal{S}_1$ *has the second transaction that adds 4* new *nodes with 0* common *node in the tree, which is the transaction shown in Figure 5.10.*

If $m = 2$, $\mathsf{W}(2) = 3$ *and* $\mathsf{C}(2) = 0$*:*

- $\mathcal{S}_2 = \{\mathcal{T} \mid \mathcal{W}(\mathcal{D}_2, \mathcal{T}) = \mathsf{W}(2) = 3 \text{ and } \mathcal{C}(\mathcal{D}_2, \mathcal{T}) = \mathsf{C}(2) = 0\}$ *where* $\mathcal{D}_2 = \{\mathcal{S}_0, \mathcal{S}_1\}$
- $wf(\mathcal{S}_2) = 3$
- $kf(\mathcal{S}_2) = 3 + 0$.

$\mathcal{S}_2$ *has the third transaction that adds 3* new *nodes to the tree with 0* common *node, which is the transaction shown in Figure 5.11. The rest of the transactions are added in the same manner.*

Now we can propose our algorithm for the upper bound of Alphabetical tree:

Algorithm 5.2 (FindAlphabeticalUpperBound)**.** *For $A = \{a_1, \ldots, a_n\}$ and TC, the biggest possible size for alphabetical tree is produced by this algorithm:*

Data: n: size of A, t: TC

Result: U: Upper bound for Alphabetical tree. Initialization: $i \leftarrow 0$, $U \leftarrow 0$

Create $\langle \mathcal{S} \rangle$ and $kf()$ and $wf()$ functions for A with size n; **while** $t \geq 0$ **do**

  while $\langle \mathcal{S}_i \rangle \neq \varnothing$ **do**

    remove transaction $\mathcal{T}$ from $\mathcal{S}_i$

    $t \leftarrow t - kf(\mathcal{S}_i)$

    $U \leftarrow U + wf(\mathcal{S}_i)$

  end

  $i \leftarrow i + 1$

end

return U

This algorithm iterates through $\langle \mathcal{S} \rangle$. The inner loop removes transactions one by one, and adds the transaction's w *new* nodes to upper bound and subtract the added transaction's size k from TC, until TC reaches 0.

Theorem 5.3. *For every n and TC the result of Algorithm* FindAlphabeticalUpperBound *is the tightest upper bound for alphabetical tree.*

Proof. First, we need to show that there is a database $\mathcal{D}$ s.t. $|\mathbb{T}_{\mathcal{D}}| = U$. As the transactions used in the while loop of the algorithm create a database $\mathcal{D}$, it is clear that such database $\mathcal{D}$ with $|\mathbb{T}_{\mathcal{D}}| = U$ exists.

Second, we need to show that there is no alphabetical tree with a larfer size. Hence, for the sake of contradiction assume that there is a database like $\mathcal{D}'$ s.t. $|\mathbb{T}_{\mathcal{D}'}| = U'$, where $U' > U$, then there are two cases:

- Case 1: Suppose that $\mathcal{D}'$ has the same transactions in $\mathcal{D}$ but in a different order. By Lemma 5.1, the ordering of transactions do not matter. Hence, we can re-order $\mathcal{D}'$ to match $\mathcal{D}$ which results in $U' = U$.

- Case 2: There are some transactions that appears in $\mathcal{D}$ but not in $\mathcal{D}'$ and vice versa. First of all, we order transactions in both databases with respect to our canonical ordering defined in Definition 5.10. Since there are some transactions in $\mathcal{D}$ and not in $\mathcal{D}'$, we have: $\mathcal{D}\backslash\mathcal{D}' = \{\mathcal{T}_{i_1} \dots \mathcal{T}_{i_t}\}$ and $\mathcal{D}'\backslash\mathcal{D} = \{\mathcal{R}_{j_1} \dots \mathcal{R}_{j_s}\}$. Obviously as both uses the same TC, we have $\sum_{k=i_1}^{i_t} |\mathcal{T}_k| = \sum_{k=j_1}^{j_s} |\mathcal{R}_k|$. As we ordered transactions in both databases with our canonical ordering and database $\mathcal{D}'$ differs database $\mathcal{D}$ in some transactions, this means that database $\mathcal{D}'$ *missed* some transaction(s) with respect to the $\succ$ ordering as defined in Definition 5.8. We know that in the $\succ$ ordering, transactions are chosen based on bigger $\frac{w}{c}$, which means adding more new nodes to the tree with less common nodes to the current tree structure. Therefore, $\mathcal{R}$s transactions are not better than $\mathcal{T}$s in terms of the $\succ$ ordering. More precisely, if $\mathcal{D}^{(\ell)} = \langle \mathcal{T}_{i_1} \dots \mathcal{T}_{i_\ell} \rangle$ and $\mathcal{D}'^{(\ell')} = \langle \mathcal{R}_{j_1} \dots \mathcal{R}_{j_{\ell'}} \rangle$ such that $\ell \leq t$ and $\ell' \leq s$ we have:

$$\frac{\mathcal{W}(\mathcal{D}, \mathcal{T}_{i_\ell})}{\mathcal{C}(\mathcal{D}, \mathcal{T}_{i_\ell})} \geq \frac{\mathcal{W}(\mathcal{D}', \mathcal{R}_{i_{\ell'}})}{\mathcal{C}(\mathcal{D}', \mathcal{R}_{i_{\ell'}})}$$

This non-equality results in: $|\mathbb{T}_{\mathcal{D}}| > |\mathbb{T}_{\mathcal{D}'}|$ or $U > U'$, which contradicts with our assumption.

■

Table 5.4 shows the upper bound for different n and TC. Table 5.5 shows upper bound when n is increased and TC is not increased. Table 5.6 shows upper bound when TC is increased and n is not increased.

5.4 FP-TREE UPPER BOUND

In this section wherever we use layout-tree we mean FP layout-tree, also we will remove the $<_{\mathsf{Freq}}$ item ordering from notations when it is clear from context.

In the previous section, we developed a greedy algorithm to compute an upper bound for the number of nodes in an alphabetical tree based on two variables, TC and n. In this section, we derive a formula to compute an upper bound for the FP-tree given the same two parameters.

Table 5.4: *Upper Bound for different n and TC*

n	TC	**loose Upper bound** 2^n	**Upper bound**
5	19	32	18
10	200	2^{10}	171
30	10^6	$2^{30} \approx 1.07 \times 10^9$	7.35559×10^4
60	10^{10}	$2^{60} \approx 1.16 \times 10^{18}$	7.55889×10^7
70	10^{15}	$2^{70} \approx 1.2 \times 10^{21}$	5.866134×10^{13}
100	10^{22}	$2^{100} \approx 1.2 \times 10^{30}$	5.192964×10^{19}
500	10^{50}	$2^{500} \approx 3.2 \times 10^{150}$	8.653038×10^{48}
2000	10^{60}	$2^{2000} \approx 1.15 \times 10^{602}$	9.768086×10^{59}

Table 5.5: *Upper Bound for $n = 100$ and different TC*

n	TC	**loose Upper bound** 2^n	**Upper bound**
100	19	$2^{100} \approx 1.2 \times 10^{30}$	19
100	200	2^{100}	200
100	10^6	2^{100}	9.7241×10^5
100	10^{10}	2^{100}	9.019651×10^8
100	10^{15}	2^{100}	7.807619×10^{13}
100	10^{22}	2^{100}	5.192964×10^{19}
100	10^{25}	2^{100}	3.86918×10^{23}

Table 5.6: *Upper Bound for* $TC = 10^{10}$ *and different* n

n	TC	**loose Upper bound** 2^n	**Upper bound**
10	10^{10}	2^{10}	1024
20	10^{10}	2^{20}	1.048576×10^6
40	10^{10}	$2^{40} \approx 1.1 \times 10^{12}$	5.028888×10^9
60	10^{10}	$2^{60} \approx 1.16 \times 10^{18}$	7.55889×10^7
80	10^{10}	$2^{80} \approx 1.2 \times 10^{24}$	8.535977×10^8
100	10^{10}	$2^{100} \approx 1.2 \times 10^{30}$	9.019651×10^8
200	10^{10}	$2^{200} \approx 1.6 \times 10^{60}$	9.688576×10^9

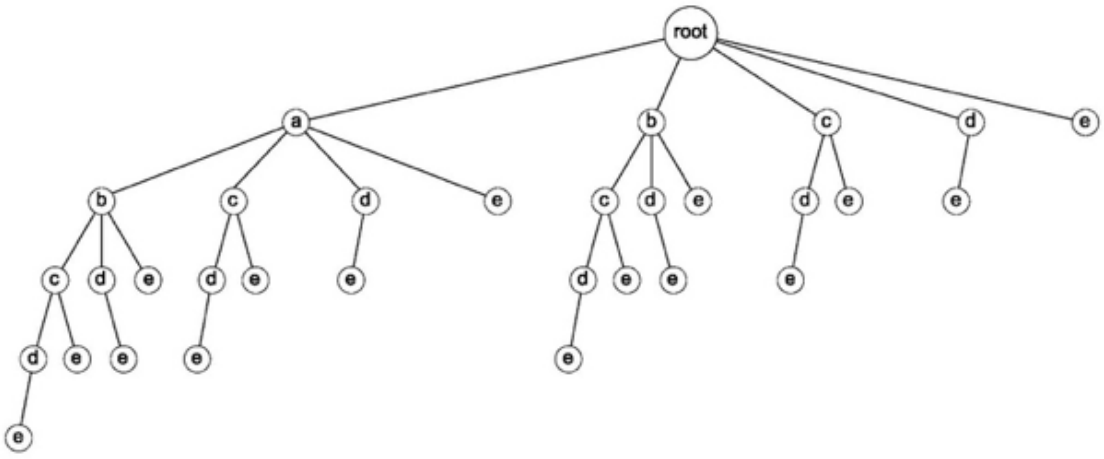


Figure 5.17: *FP layout-tree on* $A = \{a, b, c, d, e\}$*, where* $a <_{\mathsf{Freq}} b <_{\mathsf{Freq}} c <_{\mathsf{Freq}} d <_{\mathsf{Freq}} e$ *(*count $= 0$ *fields are removed for simplicity.*

Figure 5.17 shows the layout-tree on $A = \{a, b, c, d, e\}$ (*count = 0* fields are removed for simplicity), hence we are assuming that $a <_{\mathsf{Freq}} b <_{\mathsf{Freq}} c <_{\mathsf{Freq}} d <_{\mathsf{Freq}} e$. As illustrated in Figure 5.17, the total nodes of item 'a' (most frequent item) is 2^0. The total nodes of item 'b' (second most frequent item) is 2^1, total nodes of item 'c' is 2^2 and so on. We can simply assume that ($a = a_1$, $b = a_2$, $c = a_3$, $d = a_4$ and $e = a_5$, such that $i < j \Leftrightarrow a_i <_{index} a_j$).Therefore, total appearance of node a_i in the layout tree will be 2^{i-1}.

Lemma 5.4. *Assume $A = \{a_1, \ldots, a_n\}$ such that $i < j \Leftrightarrow a_i <_{index} a_j$. We claim that node a_i in $\mathbb{T}_{Layout(A)}$ appears 2^{i-1} times.*

Proof. By induction on n we prove that our claim is true for all $i \leq n$ and for any $n \in \mathbb{N}$.

- if $n = 1$ then $A = \{a_1\}$ and hence $\mathbb{T}_{Layout(A)}$ is a tree with root and just one child (2^{1-1}) labeled by a_1.
- Suppose that $n = k$, $0 \leq i \leq k$, a_i appears 2^{i-1}.
- if $n = k + 1$, $0 \leq i \leq k + 1$ then a_{k+1} appears 2^{k+1-1}.

  with $n = k+1$, $A = \{a_1, \ldots a_k, a_{k+1}\}$. $\mathbb{T}_{Layout(A)}$ with $k+1$ items is constructed from $\mathbb{T}_{Layout(A)}$ with k items, where the new node a_{k+1} is attached to all the nodes including root, so a_{k+1} appears:

$$1 + \sum_{i=1}^{k} 2^{i-1} = 1 + \sum_{i=0}^{k-1} 2^i = 1 + (2^k - 1) = 2^k = 2^{k+1-1}$$

■

Below is an example illustrating the proof.

Example 5.9. *Suppose that $A = \{a_1, a_2, a_3\}$ and $i < j \Leftrightarrow a_i <_{index} a_j$. Figure 5.18 shows layout tree on A. One can confirm that number of appearance of a_1 is $2^{(1-1)}$, a_2 is $2^{(2-1)}$ and a_3 is $2^{(3-1)}$. Figure 5.19 shows the previous layout tree with additional node a_4. This a_4 node is added to all the nodes with the total appearance of $2^{(4-1)}$.*

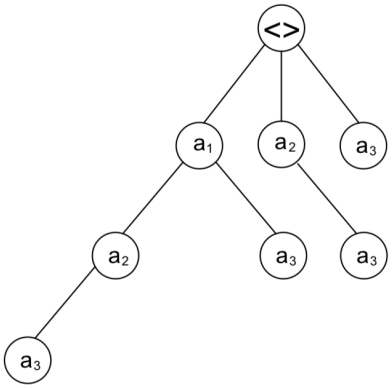


Figure 5.18: *layout tree on A where each a_i appeared $2^{(i-1)}$*

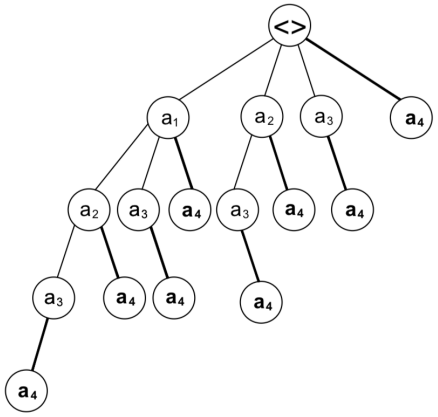


Figure 5.19: *adding a_4 with total appearance of $1 + \sum_{i=1}^{3} 2^{i-1} = 2^{4-1}$*

We build our formula for the upper bound of the number of nodes in the context of the following simple examples.

Example 5.10. *What is the upper bound for FP-tree if $n = 5$, $A = \{a, b, c, d, e\}$, and $TC = 15$. But suppose that we have extra information on the* Freq *of items, given as follows:* $\mathsf{Freq}(a) = 5, \mathsf{Freq}(b) = 4, \mathsf{Freq}(c) = 3, \mathsf{Freq}(d) = 2$ *and* $\mathsf{Freq}(e) = 1$.

There are 2^4 nodes for item e in the layout-tree (shown in Figure 5.17), but in our example, $\mathsf{Freq}(e) = 1$. *Hence, $(2^4 - 1) = 15$ nodes will not appear in this tree (*unused nodes*). For item d, we have 2^3 nodes in the layout tree, but in this example* $\mathsf{Freq}(b) = 2$, *therefore, minimum number of $(2^3 - 2) = 6$ nodes, will not appear in the tree. For item c there are 2^2 nodes in the layout-tree, but in this example* $\mathsf{Freq}(c) = 3$, *thus minimum number of $(2^2 - 3) = 1$ node will not appear. For item b there are 2^1 nodes in the layout tree and* $\mathsf{Freq}(b) = 4$. *Note that in this case $(2^1 - 4) = -2$ is negative, so this item has no* unused *nodes (we cannot have more nodes than in layout-tree). Therefore, when this computation becomes negative we know that it means* 0 unused *node. Similarly, for item a there are 2^0 nodes in the tree, but again* $\mathsf{Freq}(a) = 5$, *and $(2^0 - 5) = -4$ is negative, therefore, there is no* unused *node for item a.*

Table 5.7: *Unused nodes in FP-tree with cut off subtraction.*

Items	Counts	Number of unused nodes
a	5	$2^0 \dot{-} 5 = 0$
b	4	$2^1 \dot{-} 4 = 0$
c	3	$2^2 \dot{-} 3 = 1$
d	2	$2^3 \dot{-} 2 = 6$
e	1	$2^4 \dot{-} 1 = 15$
		$Total = 0 + 0 + 1 + 6 + 15 = 22$

Based on our calculations of the number of unused *nodes for each item described above, we introduce an operator called cut-off subtraction:*

$$\textit{cut off Subtraction:} \quad x \dot{-} y = \begin{cases} x - y & x \geq y \\ 0 & \textit{otherwise} \end{cases} \tag{5.1}$$

Therefore, by using this equation we will change the negative values for unused node counts to 0. In order to get the maximum number of nodes in the tree, we just need to subtract 2^5 *(which is the total number of nodes in the layout-tree) from the total number of unused nodes described above. In this example as shown in Table 5.7, there are total of 22* unused *nodes. Hence, the upper bound is,* $U =$ $2^5 -$ *total number of unused nodes , i.e.* $U = 2^5 - 22 = 10$.

In what follows we investigate possibility of finding the U only based on TC and n (size of A).

Example 5.11. *What is the* U *for FP-tree if* $TC = 15$ *and* $n = 5$*?*

$$U = 2^5 - \textit{total number of unused nodes.} \tag{5.2}$$

$$\textit{total number of unused nodes} = 2^0 \dot{-} \mathsf{Freq}(a) + 2^1 \dot{-} \mathsf{Freq}(b) + 2^2 \dot{-} \mathsf{Freq}(c) + 2^3 \dot{-} \mathsf{Freq}(d) + 2^4 \dot{-} \mathsf{Freq}(e). \tag{5.3}$$

Therefore, based on (Eq 5.2) and (Eq5.3) the upper bound is,

$$U = 2^5 - (2^0 \dot{-} \mathsf{Freq}(a) + 2^1 \dot{-} \mathsf{Freq}(b) + 2^2 \dot{-} \mathsf{Freq}(c) + 2^3 \dot{-} \mathsf{Freq}(d) + 2^4 \dot{-} \mathsf{Freq}(e)). \tag{5.4}$$

Constraint 5.1. *We want to maximize U with the following* constraints*:*

1. $\mathsf{Freq}(a) \geq \mathsf{Freq}(b) \geq \mathsf{Freq}(c) \geq \mathsf{Freq}(d) \geq \mathsf{Freq}(e)$
2. $\mathsf{Freq}(a) + \mathsf{Freq}(b) + \mathsf{Freq}(c) + \mathsf{Freq}(d) + \mathsf{Freq}(e) = 15$
3. $\mathsf{Freq}(a) > 0, \mathsf{Freq}(b) > 0, \mathsf{Freq}(c) > 0, \mathsf{Freq}(d) > 0, \mathsf{Freq}(e) > 0$

To maximize U one needs to ***minimize the total number of unused nodes****. Note that each term in the total number of unused nodes is larger than or equal to zero, i.e. $(2^1 \dot{-} \mathsf{Freq}(b) \geq 0)$. Now suppose that we have a* ***total budget*** *count of 15 for TC to be distributed to the items. Assigning frequency one for each distinct item has to be done to satisfy the Constraint No. 3. Hence, 1 for a, 1 for b, 1 for c, 1 for d and 1 for e. This already minimizes the first term, i.e., $(2^0 \dot{-} \mathsf{Freq}(a))$. Total 5 assigned, and now $TC = 15 - 5 = 10$. Adding more to the frequency of a does not help to minimize the total unused nodes as we have exhausted the amount that we could minimize this term. Clearly, we would like most to assign the budget to e (since the term for item e has the biggest value in cut-off subtraction, which is $2^4 \dot{-} \mathsf{Freq}(e)$) but the Constraint No. 1, prevents us from assigning a budget to e before assignment to the more frequent items. To best use the budget, we only add one to the more frequent item, till the permission for adding to the least frequent item is issued. We keep repeating this process till the budget is finished, therefore again we need to assign 1 for a, 1 for b, 1 for c, 1 for d and 1 for e, total 5 assigned, and now $TC = 10 - 5 = 5$. The remaining budget of size 5 is also assigned in the same order form a to e ,1 by 1. This means, to maximize U, TC should evenly be distributed through the items as much as possible. To formally proof this claim first consider Lemma 5.5.*

Lemma 5.5. *Suppose $\mathcal{D}$ is a database on A and $\exists$ a,b $\in A$ where $\mathsf{Freq}_{\mathcal{D}}(a) > \mathsf{Freq}_{\mathcal{D}}(b)$. Then a database $\mathcal{D}'$ exists such that by decreasing the* Freq *of a by 1 and adding 1 to* Freq *of b (such that the frequency ordering remains unchanged), may result in an* **increase** *in FP-tree size of the $|\mathbb{T}_{\mathcal{D}'}|$ or equivalently:*

(I) $\mathsf{Freq}_{\mathcal{D}'}(a) = \mathsf{Freq}_{\mathcal{D}}(a) - 1$

(II) $\mathsf{Freq}_{\mathcal{D}'}(b) = \mathsf{Freq}_{\mathcal{D}}(b) + 1$

(III) $\forall x(x \neq a \wedge x \neq b) \Rightarrow \mathsf{Freq}_{\mathcal{D}'}(x) = \mathsf{Freq}_{\mathcal{D}}(x)$

(IV) $\mathsf{Freq}_{\mathcal{D}'}(a) > \mathsf{Freq}_{\mathcal{D}'}(b)$

and

$$|\mathbb{T}_{\mathcal{D}'}| \geq |\mathbb{T}_{\mathcal{D}}|$$

Proof. We distinguish two cases:

- There exist a node in $\mathbb{T}_{\mathcal{D}}$ such that *count* of a is greater than one. Then decreasing Freq of a by 1, means that simply removing a from a transaction on this path (tree size remain unchanged). Increasing Freq of b by 1 may result in a node of b in tree from $count = 0$ to $count = 1$, which means a *new* node is created (tree size increased). If a node of b with $count \neq 0$ increased, then no new node is created (tree size remain unchanged). Therefore,

  $$|\mathbb{T}_{\mathcal{D}'}| \geq |\mathbb{T}_{\mathcal{D}}|$$

  .

- All nodes of a's have a *count* equal to one. Since $\mathsf{Freq}_{\mathcal{D}}(a) > \mathsf{Freq}_{\mathcal{D}}(b)$, there exist a transaction $\mathcal{T}$ where $\mathsf{Acc}(a; \mathcal{T}) = 1$ but $\mathsf{Acc}(b; \mathcal{T}) = 0$. We can simply replace a by b in $\mathcal{T}$ to get $\mathcal{D}'$. Therefore, we ended up with the same tree size in $\mathcal{D}'$.

■

Below example better illustrate the proof.

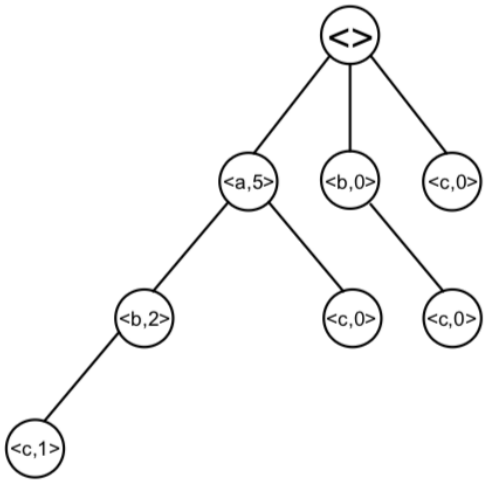


Figure 5.20: *FP-tree $\mathbb{T}_{\mathcal{D}}$ for Example 5.9*

Example 5.12. *Suppose that $\mathcal{D} = \{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4\, \mathcal{T}_5\}$, $\mathcal{T}_1 = \langle a \rangle, \mathcal{T}_2 = \langle a, b, c \rangle, \mathcal{T}_3 = \langle a \rangle, \mathcal{T}_4 = \langle a, b \rangle$ and $\mathcal{T}_5 = \langle a \rangle$. FP-tree $\mathbb{T}_{\mathcal{D}}$ is shown in Figure 5.20 where* $\mathsf{Freq}(a)$*=5,* $\mathsf{Freq}(b)$*=2,* $\mathsf{Freq}(c)$*=1 and $|\mathbb{T}_{\mathcal{D}}|$ =4.*

By decreasing Freq *a by 1 and increasing b's* Freq *by 1, we will end up to two different trees as shown in Figure 5.21 and Figure 5.22 (affected nodes are bold). Figure 5.21 shows where increasing b's* Freq *leads to a new node and $|\mathbb{T}_{\mathcal{D}'}| = |\mathbb{T}_{\mathcal{D}}|+1 = 5$. Figure 5.22 shows where increasing b's* Freq *do not create a new node and $|\mathbb{T}_{\mathcal{D}'}| = |\mathbb{T}_{\mathcal{D}}|$.*

Corollary 5.6. *For any given A and $\mathcal{D}$, the largest size of $\mathbb{T}_{\mathcal{D}}$ requires that for any $a, b \in A$ we have $|\mathsf{Freq}_{\mathcal{D}}(a) - \mathsf{Freq}_{\mathcal{D}}(b)| \leq 1$.*

Proof. Suppose that for the sake of contradiction $\exists\ a, b \in A$ where $|\mathsf{Freq}_{\mathcal{D}}(a) - \mathsf{Freq}_{\mathcal{D}}(b)| \geq 2$. Then we can safely decrease Freq of a by 1 and increase Freq of b by 1 as described in Lemma 5.5 (notice that frequency ordering remains the same) and expect a tree that may increase in size. ■

This corollary means, to maximize U, all item frequencies must be equal, and if they are not, they should

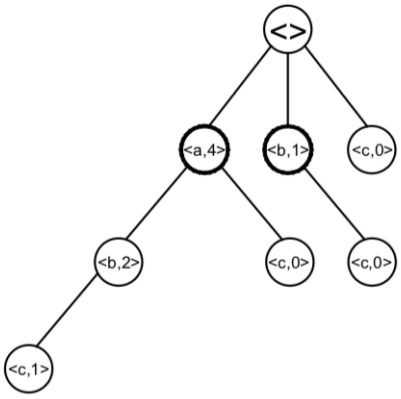


Figure 5.21: *Increasing b's* Freq *leads to a new node*

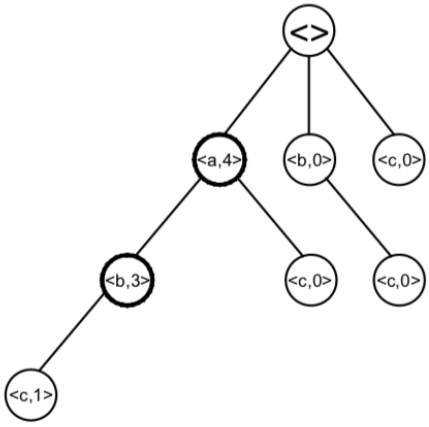


Figure 5.22: *Increasing b's* Freq *do not create a new node*

differ by **only 1** in size. In other words TC should evenly be distributed through the items as much as possible.

Lemma 5.7. *Maximum size of FP-tree on* $A = \{a_1, \ldots, a_n\}$ *based on* $\mathsf{Freq}_{\mathcal{D}}(a)$ *for all* $a \in A$*, equals to:*

$$2^n - \sum_{i=1}^{n} (2^{i-1} \dot{-} \mathsf{Freq}_{\mathcal{D}}(a_i)). \tag{5.5}$$

Proof. Base on Equation 5.2, $U = 2^n -$ *total number of unused nodes.* Lemma 5.4 says that ith element of A appears 2^{i-1} times in the layout tree. In addition, we know a_i occurs $\mathsf{Freq}_{\mathcal{D}}(a_i)$ times in $\mathbb{T}_{\mathcal{D}}$, hence, minimum number of unused nodes of a_i will equal to $2^{i-1} \dot{-} \mathsf{Freq}_{\mathcal{D}}(a_i)$, and adding it up for all n items, will result in the *total number of unused nodes* $= \sum_{i=1}^{n} (2^{i-1} \dot{-} \mathsf{Freq}_{\mathcal{D}}(a_i))$. ■

Lemma 5.8. *Upper bound for FP-tree given* n *and* TC *equals to:*

$$U(n, TC) = \begin{cases} 2^n - \sum_{i=1}^{n} (2^{i-1} \dot{-} \dfrac{TC}{n}) & n \text{ divides } TC \\ 2^n - \sum_{i=1}^{n} \left(2^{i-1} \dot{-} \left(\left\lfloor \dfrac{TC}{n} \right\rfloor + \operatorname{sign}\left((TC - \left\lfloor \dfrac{TC}{n} \right\rfloor \times n) \dot{-} (i-1) \right) \right) \right. & \text{otherwise} \end{cases} \tag{5.6}$$

Proof. Corollary 5.6 condition happens when all frequencies are equal, and if they are not, they should differ

by only 1 in size. To achieve this goal, if n divides TC then $\frac{TC}{n}$ is equally assigned for each item frequency. By plugging in $\frac{TC}{n}$ in Lemma 5.7's equation, *Upper bound* equals to $2^n - \sum_{i=1}^{n}(2^{i-1} \dot{-} \frac{TC}{n})$.
If TC is not divisible to n, First $\left\lfloor \frac{TC}{n} \right\rfloor$ is equally assigned for each item frequency. Then, we would need another term to divide the *residual budget* $(TC - \left\lfloor \frac{TC}{n} \right\rfloor \times n)$ to items (1 for each item until the residual is finished), which is: $\text{sign}\left((TC - \left\lfloor \frac{TC}{n} \right\rfloor \times n) \dot{-} (i-1)\right)\right)$. Note that the argument inside the sign function is positive or zero (so sign function will return either 0 or 1). In the latter case if the frequencies are not equal, it is guaranteed to differ only 1 in size. ■

Example 5.13. *Suppose that* $A = \{a_1, a_2, a_3, a_4\}$ *where* $n = 4$*, and consider two TC values 12 and 15.*

- *If* $TC = 12$ *then* n *divides* TC *and* $\frac{12}{4} = 3$ *is equally assigned for each item frequency.*

$$U = 2^4 - \sum_{i=1}^{4}(2^{i-1} \dot{-} 3) = 10$$

- *If* $TC = 15$ *then* TC *is not divisible to* n*. First* $\left\lfloor \frac{15}{4} \right\rfloor = 3$ *is equally assigned for each item frequency, and the* residual budget $(15 - \left\lfloor \frac{15}{4} \right\rfloor \times 4 = 3)$ *split between* a_1,a_2 *and* a_3 *(1 for each).*

$$U = 2^4 - \sum_{i=1}^{4}\left(2^{i-1} \dot{-} \left(\left\lfloor \frac{15}{4} \right\rfloor + \text{sign}\left((15 - \left\lfloor \frac{15}{4} \right\rfloor \times 4) \dot{-} (i-1)\right)\right)\right) = 11$$

5.5 SUMMARY

In this chapter, we propose a new tight upper bound for the number of nodes and hence the memory requirements of the alphabetical and FP-tree based structure. These upper bounds are calculated based on the total number of the items and distinct items in the database. The upper bound for the alphabetical tree structure is provided as a simple algorithmic routine while the upper bound of the FP-tree is formulated in a closed form. We anticipate a wide range of applications for these upper bounds in various applications. The recent acceleration in the amount of big data storage and the need to analyze the frequent patterns in

these large databases would indeed be a great indication for usefulness of having upper bounds on the size of these tree structure in order to perform efficient analysis.

Chapter 6

Improving the Cold-Start Problem in Recommender Systems with Association Rule Mining and SVD-based Features

6.1 INTRODUCTION

Recommender systems are often used by online retailers and service providers to aid users with their selection of products, services, songs, etc. Such systems generate recommendations based on available information on the user from their online profiles, previous purchases, as well as popular pairings from previous transactions by other users. However, such information may not always be available, or sufficient enough so as to generate good recommendations for the user, a problem that is commonly referred to as the cold-start problem in the literature. This section explores the feasibility of using SVD-based embedding features and association rules to enhance user profile information and help generate more useful recommendations in cases where client information is not available directly. A new approach is proposed herein, and its applicability and efficiency regarding redundant and non-redundant rule sets is investigated as part of this thesis. Given that

rich content information is often available for both music-seeking online users and the wide of assortment of music available online, music recommender systems were selected to test the applicability of the herein developed models.

Until recently, people listened to their own local music library and rarely curated collections online. However, with the advent of streaming services, which circumvent the need to download manually and organize songs within local libraries, the era of local music libraries has been slowly giving way to a new area, where music curation primarily occurs online. Indeed, personalization algorithms and unlimited streaming services such YouTube, Spotify, etc. are emerging as top picks for curation of music. However, the online availability of an impossibly large collection of music by innumerous artists has brought new challenges to online music service providers. As opposed to the traditional user, whose regular musical exposure was often confined to his own locally downloaded and organized library, or top picks on the radio, todays commuters are now given the choice to listen to any of the millions of songs available online today, and will often vary in their selections so as to encompass a wide variety of songs stemming from varied genres and decades. For instance, as part of his commute, a new generation user may choose to listen to songs by varied artists such as Lady Gaga, Vivaldi, and the Beatles, none of which is from the same genre or decade. Furthermore, new songs and new artists emerge every week, adding another great challenge to recommendation algorithms. Without enough historical data, how can an algorithm predict whether a listener will like a new song or a new artist? And how can it recommend songs to brand new users? The popularity of online content services and social media has demonstrated the value of providing relevant information to users. Recommender systems have proven to be an effective tool for this purpose and, as such, have been increasingly receiving more attention. In addition, the enormous and ever increasing amount of available training data together with advances in hardware (such as GPUs) have made it possible to tackle these problems in a reasonable amount of time.

6.2 USER PROFILING

Briefly, a recommender system consists of a system that uses information about its users (e.g. their profiles) as input to generate as output a set of recommendations for a user in an area of interest based on its decision logic. Input can include information such as user ratings, ratings of previous users with similar taste, the contents of the ratings or comments (if applicable), and the contents of the item itself. In this regard, the profile of the user plays a major role in the decision of a recommender system. User profiles (referred to in this chapter as User Item Profile), consist of users ratings on items, while 'items' are understood to be songs within the context of the currently discussed work . These two designations are used interchangeably within the chapter. Ratings can be explicit, where the system asks users to explicitly rate an item on a binary scale (e.g. like/dislike), or discrete scale (e.g, 1-10), or implicit, in which case the rating is inferred from the history of the users actions or interactions with an item (e.g. purchase, navigation, clicks, listen and ranks.). Let $U = \{u_1, u_2, ..., u_n\}$ be a set of users and $S = \{s_1, s_2, ..., s_m\}$ be a set of items or songs. The user item profile can then be defined as:

Definition 6.1. *(User Item Profile):* $P_S : U \rightarrow [0, r_S]^m$*, where* $[0, r_S]$ *is the rating scale such that for a user* $u_i \in U, P_S(u_i) =< v_{i1}, v_{i2}, , v_{im} >$*, each* v_{ij} *is the rating to item/song* s_j*, and* $1 \leq i \leq n, 1 \leq j \leq m$*.*

In order to recommend relevant choices for a target user, collaborative filtering recommender systems have to first generate a pool of other users (neighbours) with user profiles similar to the target user. As a next step, items that have been preferred or highly rated by neighbours would then be recommended to the target user as possible choices. This approach works well when the number of items and the pool of neighbours is large. However, when the target user has a relatively small collection of rated items (referred to as a 'short' profile), it becomes difficult for the system to generate an adequate pool of neighbours, resulting in a low-quality recommendation system. This is an example of a cold-start problem.

In a study by Ziegler et al., a taxonomy-based product recommender system was proposed (TPR) [81].

This approach generates ratings for the categories or features of items based on users item ratings. In this chapter, the users category rating is referred to as the User Extended Profile. Let $T = \{t_1, t_2, ..., t_l\}$ be a set of features (or topics or categories) belonging to S item-set. With that in mind, the user extended profile can be defined as Follows.

Definition 6.2. *(User Extended Profile):* $P_T : U \rightarrow [0, r_T]^l$*, where* $[0, r_T]$ *is the scale of the feature or category ratings. This means for a user* $u_i \in U, P_T(u_i) =< a_{i1}, a_{i2}, ..., a_{il} >$*, each* a_{ij} *represents user's interest in feature or category* $t_j \in T, 1 \leq j \leq l$*.*

Given that the user extended profile is a representation of the user's interest in topics rather than individual items, the TPR system thus becomes a better candidate for generation of overlapping and more expressive similarity computations. The main differentiator in the TPR system is that it capitalizes on the similarity of users extended profiles rather than item profiles to generate neighbourhoods for users. The number of items is often much bigger than the number of categories of items available at any given time (i.e, $m \gg l$); thus, the system is able to generate much denser user taxonomy profiles as compared to user item profiles. This aids in the generation of more accurate neighbourhoods, which in turn improves the quality of the recommendation system [81]. Despite this advantage of extended-based approaches, the cold-start problem still exists in cases where user ratings are insufficient.

To illustrate an example of user item and extended profiles, as well as to evaluate our model, a dataset provided by KKBOX at the ACM WSDM18 competition will be used throughout this chapter.

Example 6.1. *KKBOX has provided a data set that consists of information regarding the first observable listening event for each unique user-song pair (E) within a specific timeframe. Intuitively, if a user really enjoys a song, s/he will repetitively listen to it. We are asked to predict the chances of a user listening to a song repetitively after the first observable listening event within a given time window. If there are recurring listening event(s) triggered within a month after the user's very first observable listening event, its target is marked 1, and 0 otherwise. More formally, let us assume an event* $E(U, S, T_1)$ *in which a user U listened*

Table 6.1: *Distributions of users and songs in train and test sets*

Train	**Test**
7.377.418 events	2.556.790 events
30.755 unique users	25.131 unique users
359.966 unique songs	224.753 unique songs
9.272 users are in train but not in test	3.648 users are in test but not in train; this corresponds to 14.51% new users
195.086 songs are in train but not in test	59.873 songs are in test but not in train; this corresponds to 26.64% new songs

to a song S at time T_1. If we observe a subsequent event $E'(U, S, T_2)$ where $T_2 - T_1 < 1$ month (that is, repetitive listening took place within one month), event E will be marked as 1; otherwise, it will be marked as 0. The models are evaluated on an area under the ROC curve between the predicted probability and the observed target. Training and the test data are selected from the user's listening history within a given time period and have around 7 and 2.5 million unique user-song pairs respectively. It is worth mentioning that this structure also suffers from the cold start problem: 14.5% of the users and 26.6% of the songs in test do not appear in the training data. Table 6.1 contains some statistics on users and songs in the training and test data, while Table 6.2 summarizes all meta data provided by KKBOX, which demonstrate the presence of the cold start problem. Hence, the main task of the model is to predict whether or not a new listener will like a new song or a new artist.

Table 6.3 illustrates a user item (song) profile based on this dataset. As illustrated by this example, for each user item profile, every individual entry pertaining to preference is processed as a binary rating (e.g., like/dislike) that indicates whether the user likes a given song or not, while NA values indicate a missing entry (like/dislike) for the user song pair. The dataset used in this example contained 30,755 users and 359,966 songs, which results in a sparse matrix of $n = 30,755$ rows and $m = 359,966$ columns. Table 6.4

Table 6.2: *Meta-data for users and songs*

Train and Test	**Songs Meta-data**	**Users Meta-data**
user_id	song_length	city
song_id	genre_ids	age
source_system_tab (tab name)	artist_name	gender
source_screen_name (layout name)	Composer	registration_method
source_type (entry point)	Lyricist	registration_date
target (train only)	Language	expiration_date
	song_name	
	ISRC: song code	

shows user extended profile from meta-data provided in Table 6.2. In the user item profile, rows represent users, while columns are used to track individual preference ratings (or lack thereof; i.e. NA) for each individual song contained in the dataset. In the extended user profile, however, rows will be the users and columns are features or categories belonging to song meta-data. Using the same dataset as above, we will have $n = 30,755$ rows and $l = 5314$ columns (in this example the number of extracted features or categories was 5314, we can also confirm that $m \gg l$). To create a profile for each user, categorical features, which indicate a users taste for that feature, were aggregated (i.e. artist-name). NA values indicate a missing entry for the metadata feature associated with the users feature pair. To illustrate how the extended user profile is processed, suppose that user_1 likes 100 songs, and that 10 of those songs have genre ids_1; as a result, the entry for user_1 under the column designated for genre_ids_1 will be equal to 10, as shown in Table 6.4.

Weng et al. [71] developed the user taxonomy profile recommendation system based on previous work by Ziegler [81]. Among others, relevant literature [5, 37, 55, 58, 71, 78, 81], suggests the use of implicit song ratings to capture user preferences through item taxonomy as a means of to achieve a solution for the cold-start problem. The above cited literature suggests that songs be used to capture user preferences by converting

Table 6.3: *User item profile*

	song_1	**song_2**	**song_3**	...
user_1	0	NA	1	...
user_2	NA	NA	0	...
user_3	0	1	NA	...
...	...	...	...	...

Table 6.4: *User extended (taxonomy) profile*

	genre_ids_1	**genre_ids_2**	**artist_name_1**	**artist_name_2**	**language_1**	**composer_1**	...
user_1	10	NA	3	NA	21	2	...
user_2	NA	NA	NA	NA	7	1	...
user_3	3	200	340	NA	NA	7	...
...	...	...	...	...	...	...	...

them into concepts or concept-weighted vectors. One of the main advantages of using such an approach is that it ranks such features by their importance rather than via explicit ratings. Let us define the profile vectors for users u_i and u_j and item s_k as $P_T(u_i) =< a_{i1}, a_{i2}, ..., a_{il} >, P_T(u_j) =< a_{j1}, a_{j2}, ..., a_{jl} >$. Then let $P_T(s_k) =< a_{k1}, a_{k2}, ..., a_{kl} >$, for each s_k, where l is the number of features and a_{ij} denotes a given user's taste in that feature $t_j \in T$. In this example $T = \{\text{song\_length}, \text{genre\_ids\_1}, \text{genre\_ids\_2}, \text{artist\_name\_1}, \text{artist\_name\_2}, \text{language\_1}, \text{composer\_1}, ...\}$ and $P_T(\text{user\_1}) =< 10, NA, 3, NA, 21, 2, ... >, P_T(\text{user\_2}) =< NA, NA, NA, NA, 7, 1, ... >$. For example for song_1 we have $P_T(\text{song\_1}) =< 356789, 1, 0, NA, 1, 1, 0, ... >$. These profile vectors are used to measure similarity between users, as well as between users and items, via the Pearson correlation coefficient measure [81]. Recommendations for a particular user are thus generated based on a subset of u_i neighbours, who have also rated the same item [71]. However, despite the power of taxonomy-driven approaches, the cold-start problem still remains when applying these approaches to cases where short profiles are dominant in the rating data.

Figure 6.1 illustrates a high-level process of the user-profile-based recommender system. The raw data in Figure 6.1 consists of original data for users and songs, which is used to generate user profiles. These profiles are then used in the next step to generate neighbourhoods/clusters. These clusters include groupings of users with similar interests in items (songs). Using these groups, the system generates a ranked recommendation list that includes items of interest for a particular user based on its neighbours and their likings [81].

Here, the short profile problem can be overcome via generation of association rules, while missing features can be computed based on known user profiles. This will enable the use of rules, which contain concepts or categories, by a recommender system as a means to improve the quality of recommendations for short profiles.

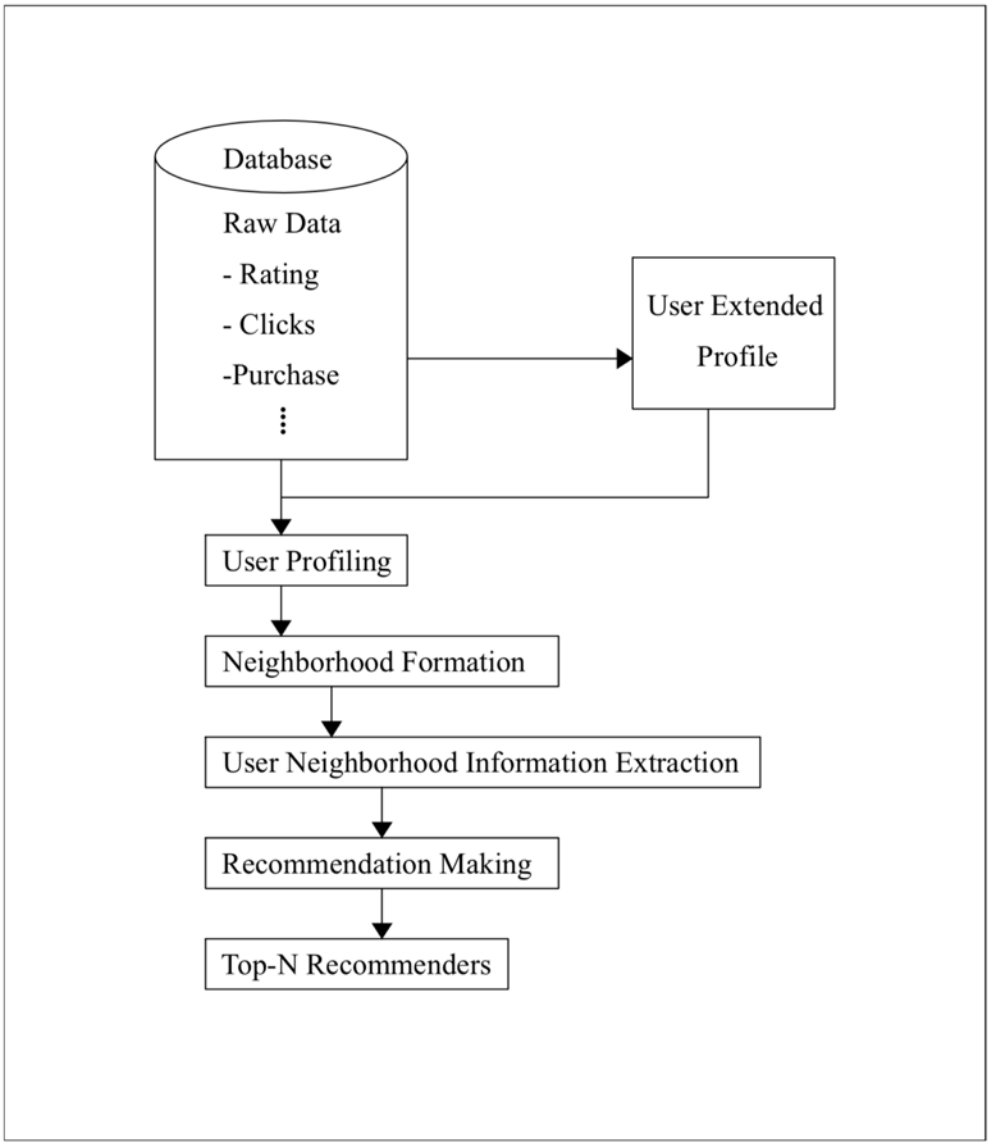


Figure 6.1: *High level view of the TPR approach*

6.3 HYBRID METHOD: ASSOCIATION RULE AND SVD-BASED FEATURE ENGINEERING AS A SOLUTION FOR THE COLD-START PROBLEM

In this section, the developed methodology to solve the cold-start problem is detailed through association rules and feature engineering. Following profile expansion and extraction of features, the problem is cast as a classification problem and addressed with the use of gradient boosted decision trees. Three different types of features were created to feed into the classifier.

1. Statistical features through expansion of user profiles by association rules.
2. Truncated SVD-based embedding features for users, songs, and artists.
3. Statistical-based features for users, songs, artists, and time.

Of note, the developed model herein described won the ACM WSDM challenge. Given that no music domain knowledge was needed to create the features, the currently presented model only relied on information gain and prediction accuracy for feature selection. The following sections elaborate on the process by individually addressing the three above-mentioned features.

6.3.1 Using Association Rules to Expand User Profiles

Often, at the heart of each recommender system lies a user profile from which the system develops its recommendations. Therefore, to address the cold-start problem particularly caused by short profiles, it is paramount to expand user profiles by increasing the number of entries (e.g, ratings and likes) in such short profiles.

Table 6.5: *An example of transactional dataset base on extended profile*

	genre_ids_1	**genre_ids_2**	**artist_name_1**	**artist_name_2**	**Language_1**	**composer_1**	...	...
user_1	1	0	1	0	1	1	...	...
user_2	0	0	0	0	1	1	...	...
user_3	1	1	1	0	0	1	...	...
...	...	...	...	...	...	...	...	...

Figure 6.2 illustrates a high-level process used to expand short user profiles through association rules. This method was used to help improve the quality and accuracy of the recommendation system.

As can be seen in Figure 6.2, the process itself is not modified; it is the input of the process (i.e. the user neighbourhood information extraction) that undergoes modifications. Therefore, this alternation does not change the mechanics of how a recommender system works, but rather, it changes what should be fed into it for better recommendations to end users.

A key difference between the currently described process and previous approaches consists of the creation of a transactional dataset, with each transaction consisting of a set of categories/topics in which the user is interested. This dataset is then mined for association rules, which are used to expand the previously generated user profiles. The expanded profiles are then used to improve recommendations.

To generate the required association rules, a transactional dataset is constructed, where each user u_i is a transaction, denoted as $Tran(u_i)$. In addition, all the topics t in the taxonomy make up the dataset's attributes, i.e, $Tran(u_i) =< b_{i1}, b_{i2}, ..., b_{il} >$, while each value b_{ij} shows if user u_i is interested in topic t_j or not. Let $P_T(u_i) =< a_{i1}, a_{i2}, ..., a_{il} >$ be user's taxonomy profile and $b_{ij} = 1$ if $a_{ij} \neq 0$ and $b_{ij} = 0$ otherwise. This designation helps create a transactional dataset towards user's interests in topics rather than items. Table 6.5 show a transactional dataset created based on the explained procedure where 'genre_ids_1', 'artist_name_1', etc. are the categories of interest.

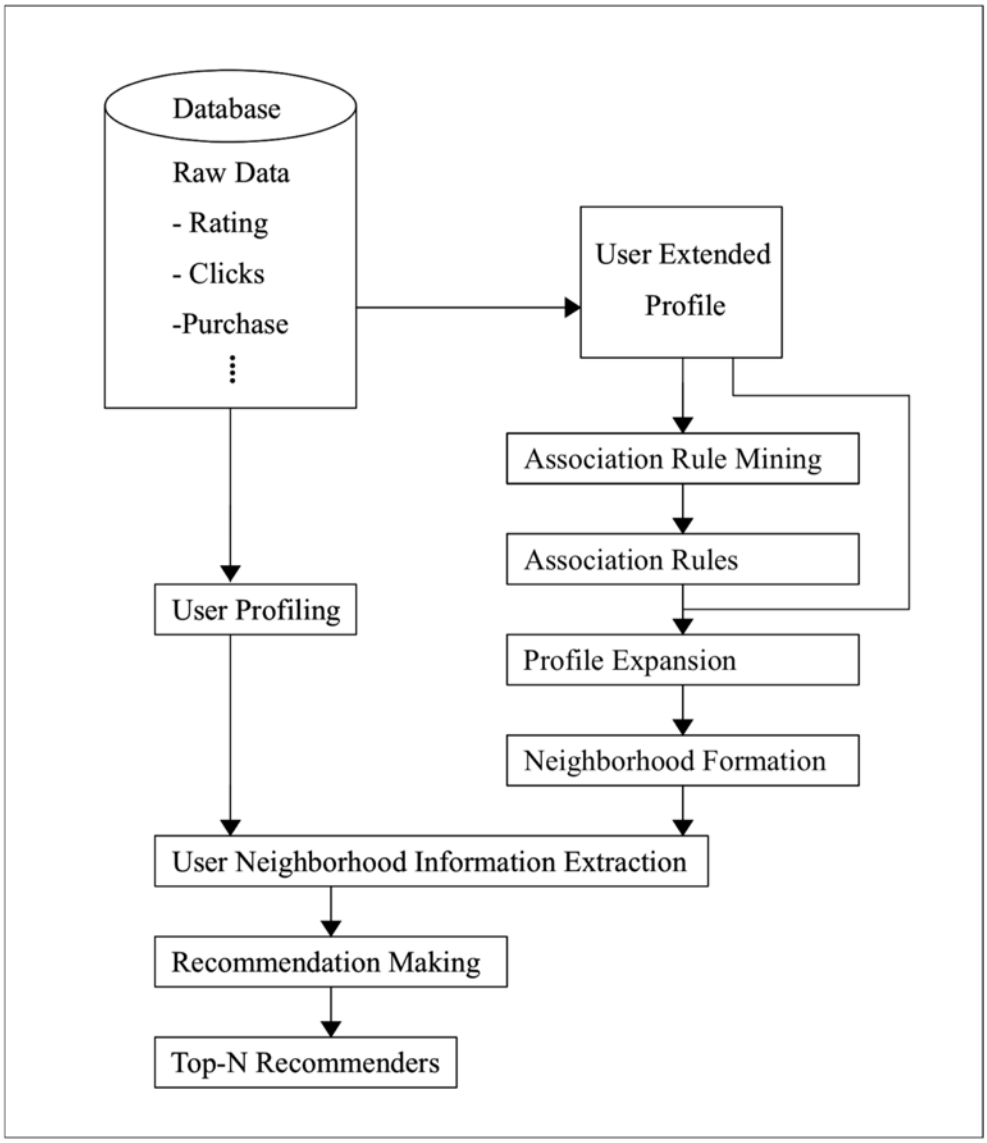


Figure 6.2: *High level flow of the proposed method for expanding short profiles*

Briefly, the goal of the developed method is to fill in the NA values in user extended profiles so as to avoid short profiles. Thus, the dataset is mined to find frequent patterns, and necessary association rules are then derived from these patterns. From these association rules, the available data on short profiles for users with a limited number of likes/dislikes can be expanded. After mining the association rules, each rule is then compared in the list to find the exact matches of its ancestor. If a match is found, its contents are added to the users profile. Each new rule has a weight as well, which is computed from its relation to its ancestor. In order to generate the association rules, we build a transactional dataset in which each user u_i is a transaction, shown as $Tran(u_i)$, and all the categories/topics t create the datasets attributes, i.e, $Tran(u_i) =< b_{i1}, b_{i2}, ..., b_{il} >$, each b_{ij} value represents if the user is interested in the category/topic t_j or not. This expansion process can be formally described as follows:

For user u_i, let $T_i = \{t_j | Tran(u_i) = \{b_{i1}, b_{i2}, , b_{il}\}, b_{ij} = 1\}$ be the set of categories that the user is interested in. For each $A \in 2^{(T_i)}$, A is a possible antecedent of an association rule. If a rule exists such that $A \rightarrow B$, the categories in $B \setminus T_i$ are thus potentially categories to be used to expand a short profile for a particular user. For each $t_j \in B \setminus T_i$, we have $t_j \in T_i, b_{ij} = 0, a_{ij} = 0$. Therefore, from the user's current profile $P_T(u_i) =< a_{i1}, a_{i2}, ..., a_{il} >$ and the confidence of the rule $A \rightarrow B$, the computation method to calculate the strength of user's interest in a particular topic can be described as follows.

$$t_j \in B \setminus T_i, t_{ij} = \frac{\Sigma_{k=1}^{|A|} a_{ij_k}}{|A|} \times conf(A \rightarrow B) \tag{6.1}$$

where $A = \{t_{j_1}, t_{j_2}, ..., t_{j_r}\}$ is a set of categories from the ancestor and $conf(A \rightarrow B)$ denotes the rule's confidence value [71]. Furthermore, based on the design of the TPR method, the values for topics in the expanded profile need to be normalized. Example 6.2 further describes this method.

This then generates a set of expanded user profiles $P'_T(u_i) =< a'_{i1}, a'_{i2}, ..., a'_{il} >$, which can be used to improve the quality of the recommendation system compared to what is traditionally used as user profiles. By using the rich user profile information generated by the above steps, we can cluster users based on profile

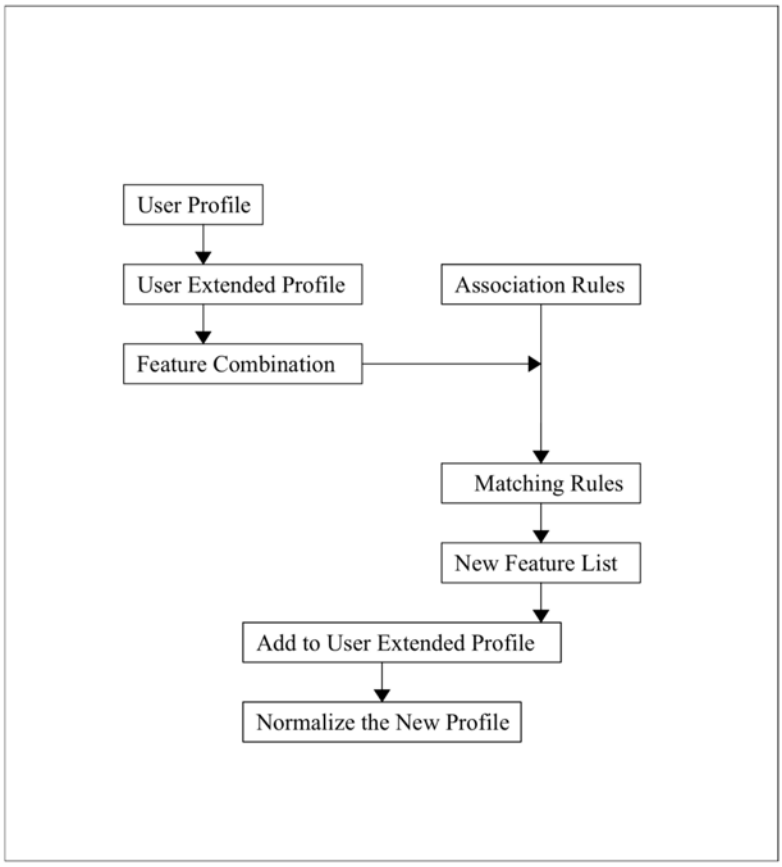


Figure 6.3: *The process of expanding user profile through association rules*

similarity and derive informative features to feed into the classifier. This process is illustrated in Figure 6.3.

While it is possible to use all matching antecedents and consequent sets from association rules based on the procedure described above, it would be more effective to use a ranking system to rank all generated rules, and only use the top ranked rules so as to not include poorly related topics to the users interest, or overburden the system with computations. In addition, this method can be used to expand any user profile in the set. However, the aim of this work is to find a solution for the cold-start problem, which is caused by the existence of short profiles. Therefore, certain restrictions need to be imposed on how the work should be implemented to expand user profiles.

Table 6.6: *User extended (taxonomy) profile*

	genre_ids_1	**genre_ids_2**	**artist_name_1**	**artist_name_2**	**language_1**	**composer_1**	...
user_1	10	NA	3	NA	21	2	...
user_2	NA	NA	NA	NA	7	1	...
user_3	3	200	340	NA	NA	7	...
...	...	...	...	...	...	...	...

- **Short Profiles:** A limitation should be set so as to only impose the method on short profiles that lack enough information for proper recommendations. It is important to note that as the number of items increase in a dataset, the available combinations also increase, resulting in more demand for computational resources to determine whether there is a match in their ancestors, and to rank existing rules as they are found. Therefore, if caution is not taken in setting a limit, at some point, the computation demand would hinder the system from generating recommendations in a timely manner, hence defeating its purpose.

- **Number of Rules:** As described above, if caution is not taken when selecting the number of rules used to expand a profile, the expansion can become too large and include poorly related topics to the users interests. Therefore, there should be a limit set on the number of rules that can be used in the expansion of a profile based on some measures, including their support, confidence and interestingness. In this work, the limitation, set based on confidence, is a hyper parameter in our model, and is obtained during model learning.

Figure 6.4 illustrates the expansion process for a user profile when the above restrictions are considered.

Example 6.2. *Using association rules to expand a user profile for user_2. In this example a simple user profile expansion is demonstrated. The assumptions held for this demonstration are based on Table 6.4 (also copied in Table 6.6 for user_2). In this example, the user is interested in language_1 and composer_1. Here,*

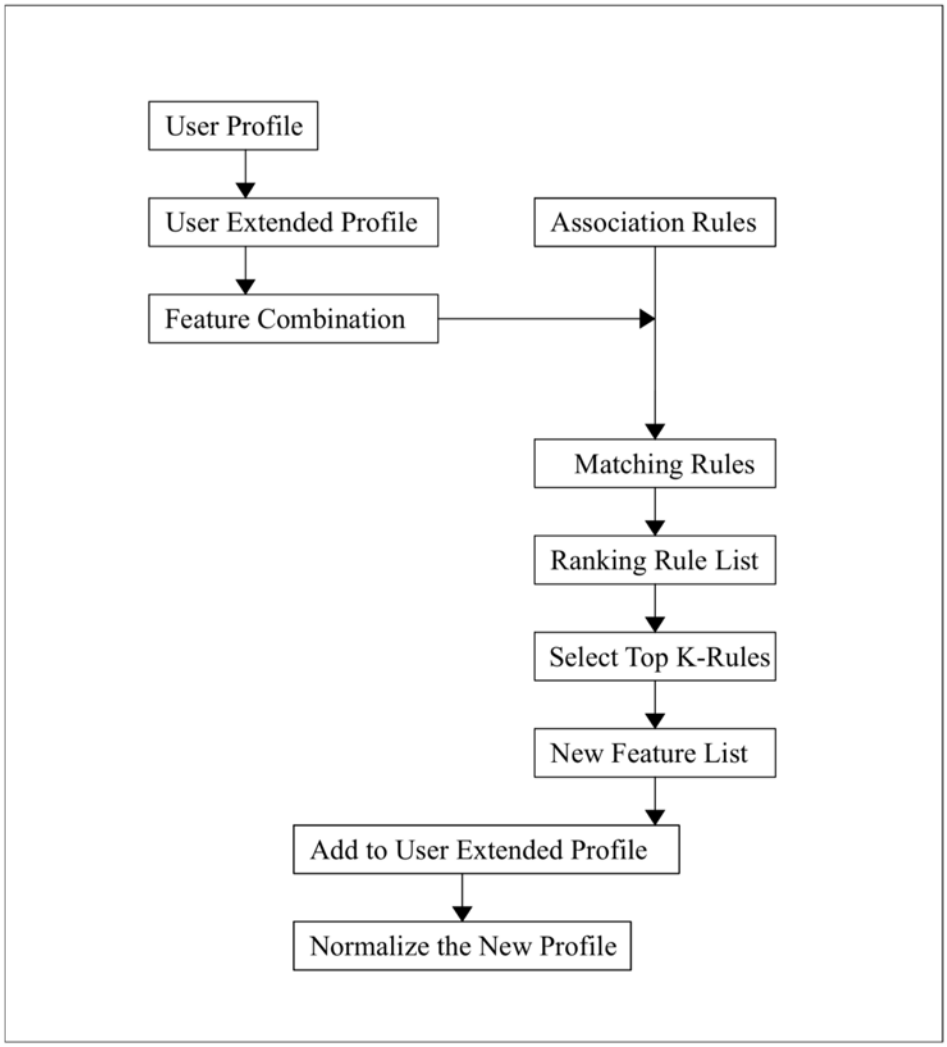


Figure 6.4: *The expansion process for a user profile with restrictions imposed*

each feature is given a weight representing the users interest in said feature. In this case, the user is more interested in language_1, with a weight of 7, as compared to composer_1with the score of 1. For this user, suppose that a set of 3 rules were found that matched feature combinations from the users profile:

$$R_1 : language\_1 \rightarrow genre\_1 \; confidence = 90\%. \tag{6.2}$$

$$R_2 : composer\_1 \rightarrow genre\_1 \; confidence = 85\%. \tag{6.3}$$

$$R_3 : language\_1, composer\_1 \rightarrow artist\_name\_2 \; confidence = 80\%. \tag{6.4}$$

Therefore, in order to work with the top 3 ranked rules, it would be beneficial to start with the highest ranked item and move down the list towards lower ranked items. Hence, the first rule to be extracted is R_1.

Adding the new topic to the user profile and calculating a score for it would result in a new profile as follows. Using Equation 6.1 the score of the new topic genre_id_1 generated can be calculated as follows:

$$t_{ij} = \frac{\Sigma_{i=1}^{|A|} a_{ij_k}}{|A|} \times R_{conf} \rightarrow t_{ij} = \frac{\Sigma_{i=1}^{|1|} a_{ij_k}}{|1|} \times 0.9 \rightarrow t_{ij} = \frac{7}{1} \times 0.9 = 6.3 \tag{6.5}$$

where $|A|$ denotes the number of topics included in the ancestor of the rule. In this case $|A|$ is equal to 1. R_{conf} denotes the confidence score, set to 0.9 in this case. Thus, the user profile is updated to $P_T(\text{user\_2}) =< 6.09, NA, NA, NA, 7, 1, ... >$.

Since the goal of the current exercise was to add up to three rules, and only one rule is used so far, the next rule to be added would be next rule on the list with the second highest score, R_2. The consequent of R_2 in this case is 'genre_ids_1', which already exists in the user's profile. Therefore, there is no need to add it to the profile and the confidence for the item also remains the same. Next, the last highest ranked rule would be R_3 with its consequent as artist_name_2, which does not exist in the user's profile. Therefore, artist_name_2 is added by using Equation 6.1, and the user's profile is expanded to $P_T(\text{user\_2}) =< 6.09, NA, NA, 3.16, 7, 1, ... >$.

After the user extended profile is updated with association rule mining, the number of short profiles reduced significantly. This procedure results in a rich user extended profile that can now be used for user-user similarity and consequent feature extraction based on that, which will be described in the following section. With this expansion of the profile, the cold-start problem, which existed due to the limited number of available data from the user, is sufficiently addressed, thus allowing the recommender system to better provide satisfactory results.

6.3.2 User-based similarity

After creating the user profile, the most common approach to find similar users is to employ the k-nearest neighbour (kNN) algorithm [41]. The targets for a collaborative filtering are two kinds: users and items. User-based kNN features can be computed via user profile vectors in high-dimensional space where each dimension represents a feature. There are two factors to be addressed when employing kNN as follows.

- The rationale behind choosing the number of nearest neighbours (i.e, k parameter),
- The measurement method to compute distance between pairs of users or items.

An optimal k value is usually obtained by repeated trials, taking the value which minimizes the prediction error. For recommender systems, this is computed based on the evaluation metric on rating scores or ranking.

Many measurement methods have been proposed for similarity computations including Pearson correlation similarity, cosine similarity and mean squared differences [24]. In this example, kNN is not directly applied to the classification. Rather, kNN is first applied on similarity between users to create additional features and then used to train the model for classification.

The cosine similarity between user u_i and user u_j can is defined as:

$$S(u_i, u_j) = \frac{R_{u_i} R_{u_j}}{||R_{u_i}|| \cdot ||R_{u_j}||} \tag{6.6}$$

where R_{u_i} and R_{u_j} denote the rating vectors for u_i and u_j, respectively.

It is very common to see features having unpredicted effects on the overall ranking. For example, it could be the case where two users sharing a popular feature may not be as similar as two users sharing a rare feature. Therefore, in this work the Inverse Document Frequency [6] was used to rank each feature based on its popularity in the dataset. To do so, the weight for feature i is defined based on its frequency q_i as

$$W_i = 1/log_2(1 + q_i) \tag{6.7}$$

Then Equation 6.6 becomes

$$S(u_i, u_j) = \frac{\Sigma_{k \in r_i \cap r_j} W_{u_{i,k}} . W_{u_{j,k}}}{\sqrt{\Sigma_{m \in r_i} W_m^2} . \sqrt{\Sigma_{n \in r_j} W_n^2}} \tag{6.8}$$

where r_i and r_j denote the indices for features rated by u_i and u_j, respectively.

Equation 6.8 can be used to compute the weighted average for feature i for user u from its k neighbours. This information can also be stored in the user profile and be fed to the classifier. It is important to note that these features computed based on user's neighbours are not limited to average ratings. In fact, other features can be used and some of which are listed following:

- Average rating on Artist, Track and album. from user's neighbours,
- Number of rating on Artist, Track and album. from user's neighbours,
- Average of similarity score from the user's neighbours who rated track, album and artist.

The overall structure for adding similarity-based feature on extended profile is shown in Figure 6.5.

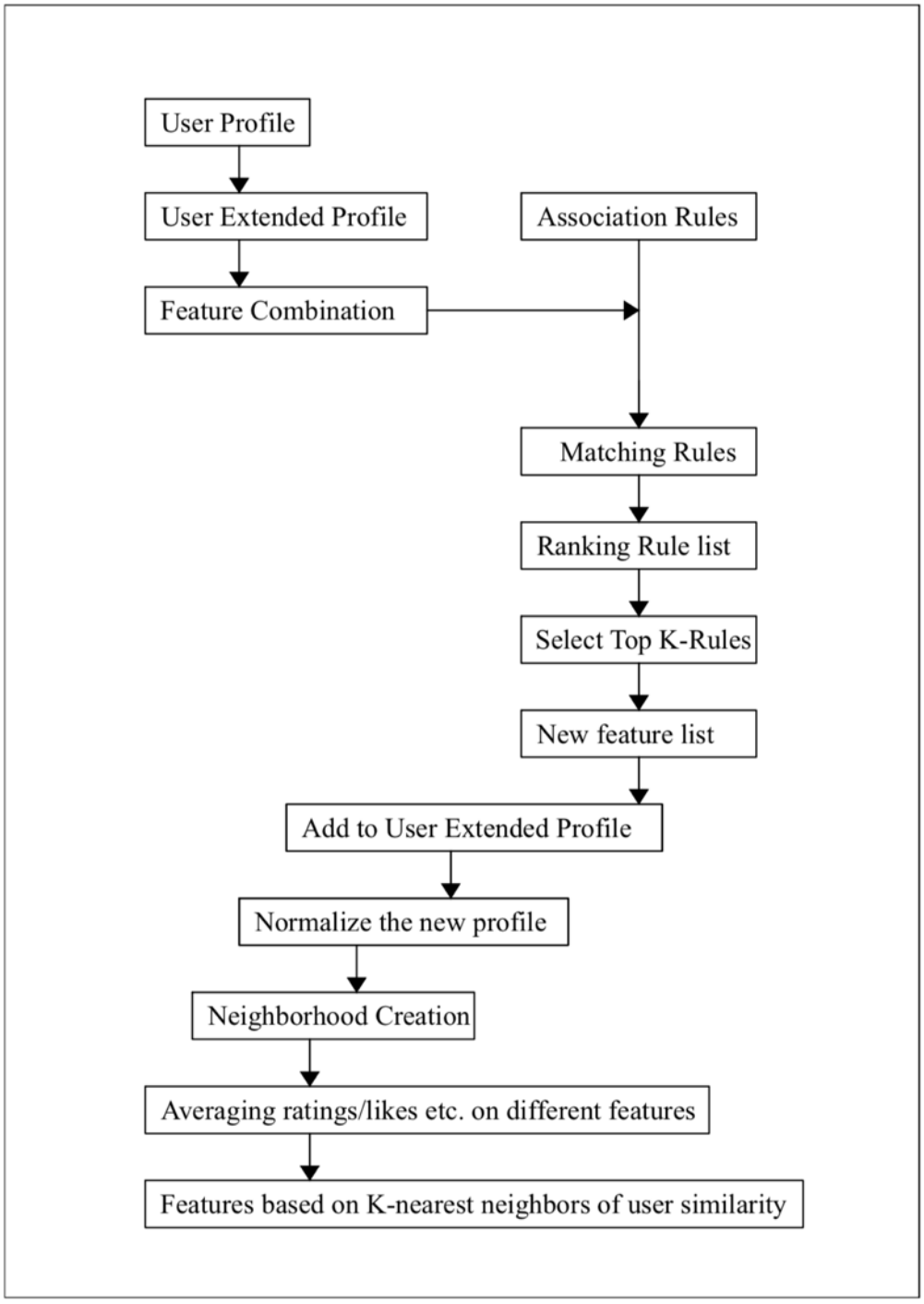


Figure 6.5: *The overall structure for adding similarity-based feature on extended profile*

6.3.3 Conditional Probability / Expectation Features

It is common among recommender systems to generate many categorical features. In the example provided here, some of these features include user_id, song_id, language, city and artist_name. Therefore, it is useful to use the conditional probability of one feature given another feature, to improve the recommendations for a user or song. The conditional probability would help gain better insight into each user's profile and recommend better options for them accordingly. For example, $P(\text{source\_type}|\text{user\_id})$ can be used to determine whether the user is using the same source that he/she is used on a regular basis or has it changed over time. These statistical properties include expectation $E(\text{song\_length}|\text{user\_id})$ and standard deviation $\sigma(\text{song\_length}|\text{user\_id})$ per user and per property.

Since the recommendation dataset is ordered chronologically, we can use the index as the timestamp. This feature can help the model find the pattern evolution during the long period of time within the training set. Furthermore, we can also count the user/song activity within a time window regarding a record of the first observable listening event.

A set of features is created based on conditional probabilities and extracted features on users, songs and artists. But, before turning into more feature engineering for this dataset, we would like to highlight some important aspects of the analysis. We noticed that statistical features based on the target feature did not work well enough during the training stage. This is probably due to the dynamics involved in the data structure and the sampling strategy made by KKBOX as shown in Figure 6.6. This figure shows the evolution of the target mean over time. The plot is set up by aggregating observations target mean in bins of size 100000. It can be seen from the figure that the target distribution is significantly decreasing over time. Consequently we should expect a target mean to decline in the test dataset too.

Further analysis of the given features shows a strong correlation between source_type feature and the target value. In Figure 6.7, we plot the distribution of song counts per source_type category for the first one

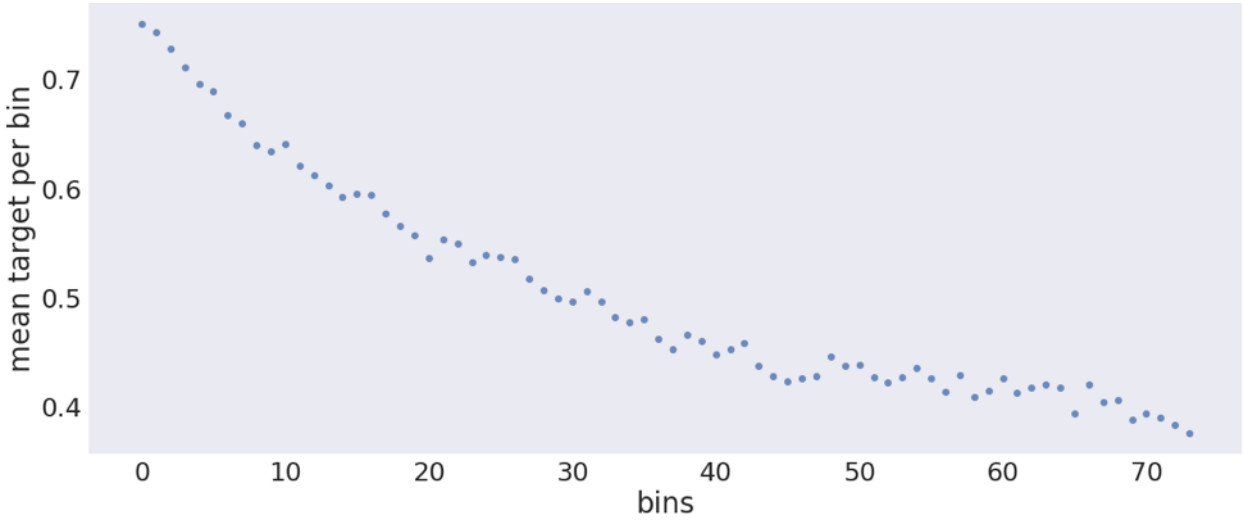


Figure 6.6: *Evolution of repeated listening (target) in time.*

million observations in the training dataset. We can observe that local_library and local_playlist categories have the highest count with a large number of re-listening.

In Figure 6.8, we plot the same distribution as above but for the last 1 million observations in the training dataset and we noticed a huge drop of re-listening in local_library and local_playlist. Also, the online_playlist counts have increased significantly. It seems that users' behaviours changed over time, influenced by online users who are more interested in what other users listened to via the online_playlist. As a result, they are more interested in discovering new songs rather than re-listening again to what they already liked. Still, we can see from the figure that listening to what other people like does not guarantee that they will re-listen to the song, which explains why the target mean drops over time.

Figure 6.9 shows that some songs are very popular and have been played more frequently than the others. We also notice a large variance in the target value for a large number of songs as the number of times a song is played increases. But the most important fact is that the chances of re-listening increases with its popularity (number of times it is played). Hence, we have introduced seven features to capture the popularity of a song and its artist:

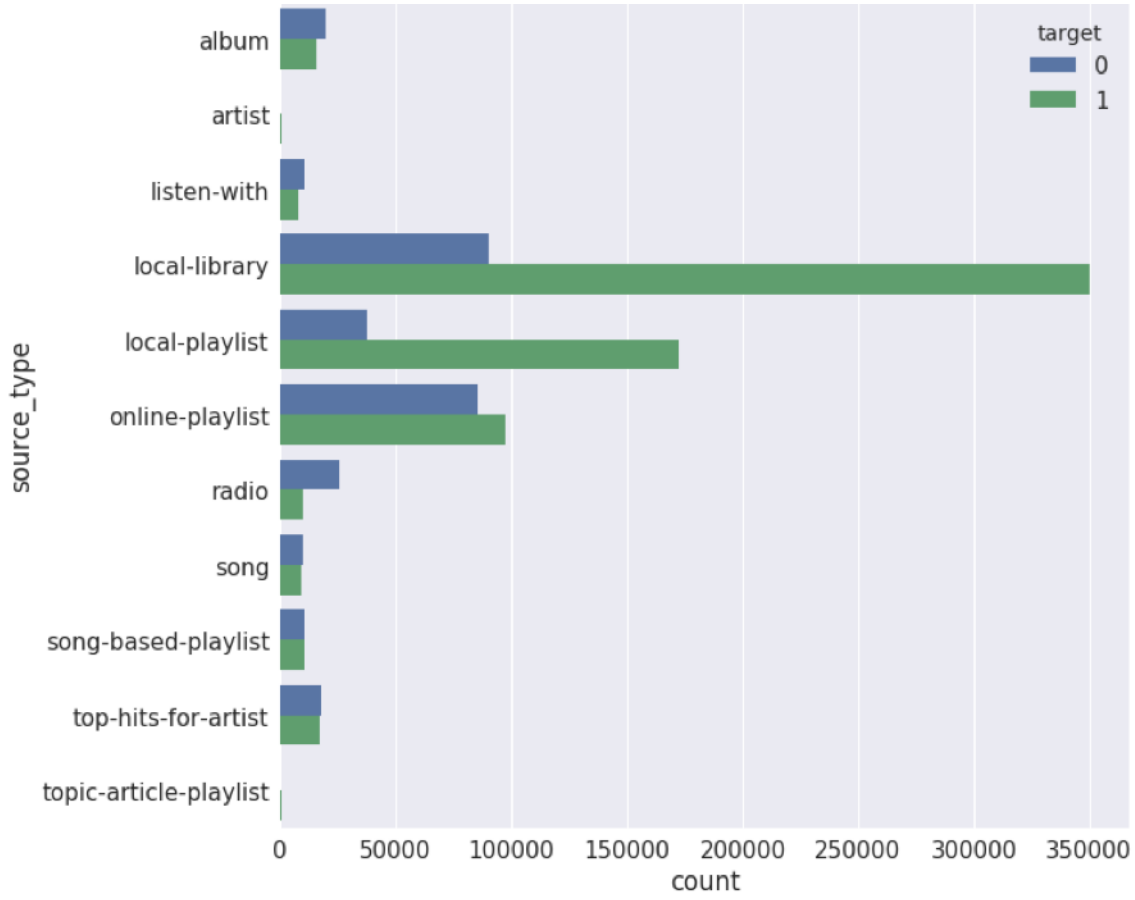


Figure 6.7: *Number of songs per source_type for the first 1 million observations and the corresponding target values*

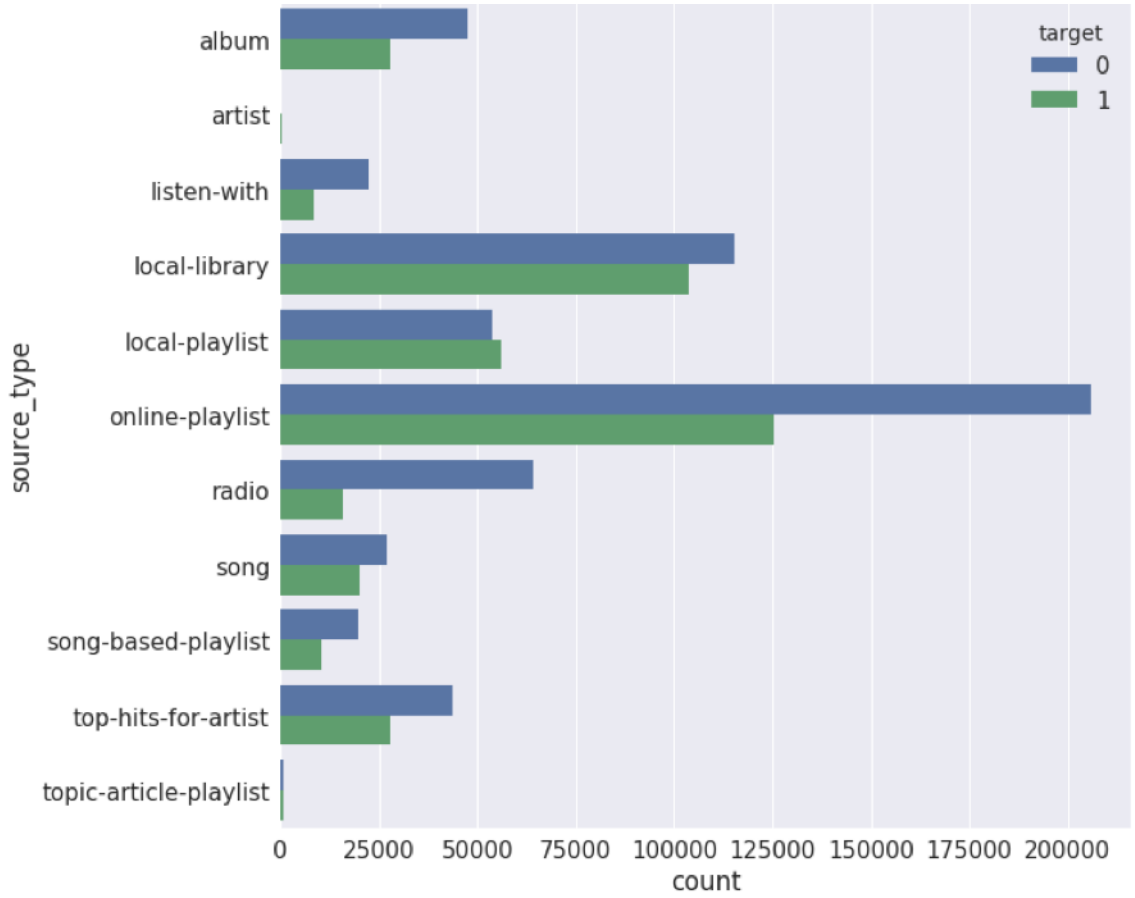


Figure 6.8: *Number of songs per source_type for the last 1 million observations and the corresponding target values*

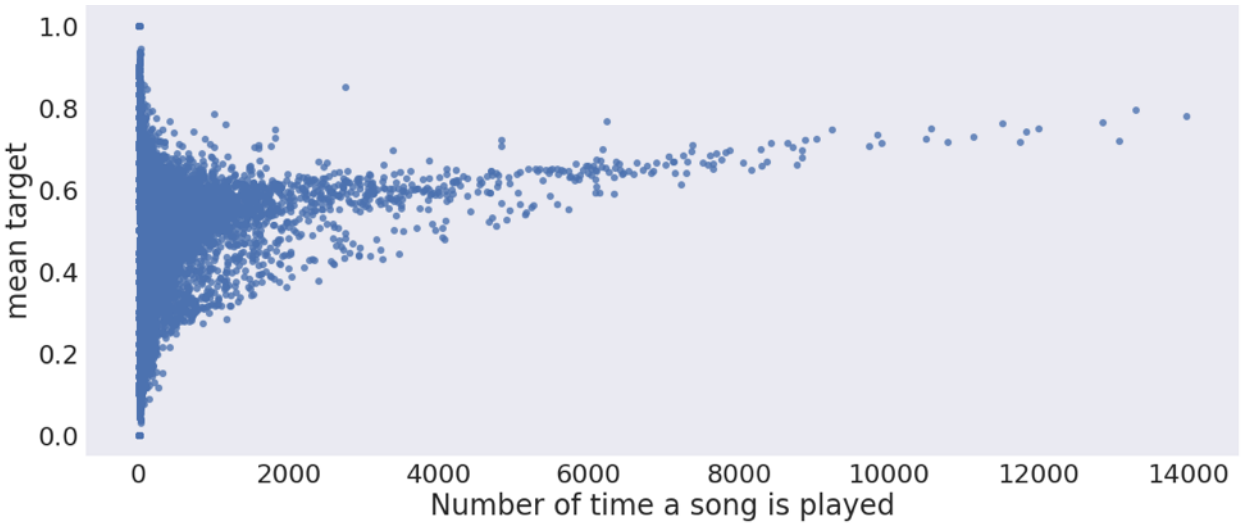


Figure 6.9: *Distribution of played songs*

- year of the song;
- country of the song;
- total count of each song_id;
- cumulative count of each song_id;
- number of times a song_id appears in each source_system_tab category;
- cumulative count of each artist_name; and
- number of times an artist_name appears in each source_system_tab category.

Several different kinds of features most of which are statistical features based on interactions between a user and a song or an artist are created. These features are summarized in Table 6.7 and Table 6.8.

Table 6.7: *Features based on user.*

Feature Notation	**Feature Description**
Yr(user_id)	Users registration year
Duration(user_id)	Number of days between users expiration date and registration date
mean((user_id, sessions),song_id)	Mean value of number of songs per user sessions
std((user_id, sessions),song_id	Standard deviation of number of songs per user sessions
sum(user_id, song_length)	Sum of songs length for each user Cumulative count of users activity
cumcount(user_id)	Cumulative count of users activity
cumcount(user_id,genre_id)	Cumulative count of user-genre interaction
cumcount(user_id, artist_name)	Cumulative count of user-artist interaction
n_unique(user_id, genre_id)	Number of unique genre categories for each user
n_unique(user_id, artist_name)	Number of unique artist counts for each user
n_unique(user_id, language)	Number of unique language categories for each user
n_unique((user_id, session),genre_id)	Number of unique genre categories for each users session
n_unique((user_id,session),artist_name)	Number of unique artist names for each users session
n_unique((user_id,artist_name),session)	Number of unique session values for each users artist

Table 6.8: *Features based on user.*

Feature Notation	Feature Description
n_unique((user_id,session,artist_name),song_id)	Number of unique songs for each artist in a users session
merge_count(user_id,session)	count number of the occurrence of merged user and session
merge_count(user_id,artist_name)	count number of the occurrence of merged user and artist
merge_count(user_id,source_type)	count number of the occurrence of merged user and source_type
merge_count(user_id,source_screen_name)	count number of the occurrence of merged user and source_screen_name
merge_count(user_id,source_system_tab)	count number of the occurrence of merged user and source_system_tab
merge_count(user_id,genre_id)	count number of the occurrence of merged user and genre
merge_count(user_id,artist_name,song_year)	count number of the occurrence of merged user, artist and song year
n_unique((user_id,artist_name),song_year)	Number of unique song years for each users artist
n_unique((user_id,artist_name),song_country)	Number of unique song countries for each users artist
n_unique((user_id,artist_name),genre_id)	Number of unique genres for each users artist
n_unique((user_id,artist_name),gender)	Number of unique genders for each users artist
n_unique((user_id,session),song_id)	Number of unique songs for each users session
n_unique((user_id,artist_name),song_id)	Number of unique songs for each users artist

6.3.4 Matrix Factorization and Truncated SVD-based features

Among the latest state-of-the-art, Matrix Factorization (MF) techniques have gained a lot of popularity in recent decades [42, 56, 66], where they are most commonly used for solving recommendation system problems such as data sparsity and cold-start. The key aspect of this method is finding the unknown ratings in the matrix that involves users and items, sorting the ratings and selecting the top k items.

One of the well-known factorization techniques is Singular Value Decomposition (SVD) and it represents one of the elements in our model. The singular value decomposition of an $n \times d$ matrix A expresses the matrix as the product of the three simple matrices:

$$A = USV^T \tag{6.9}$$

where: U is an $n \times n$ orthogonal matrix. V is an $d \times d$ orthogonal matrix, and S is an $n \times d$ diagonal matrix with non-negative entries, and with the diagonal entries sorted from high to low (as one goes "northwest" to "southeast"). A set of latent user-based features and a set of item-based features can be derived from the user-song interaction matrix using SVD technique [30, 57]. In our problem user-song matrix is huge and sparse (there are 34403 users and 419839 songs). A variation of SVD called truncated-SVD is used to approximate the user-song matrix and decompose the latent factors (or embedded feature) [54].

Truncated-SVD consists in building rank-k approximation A_k to the rank r matrix A by using the k most significant singular components, where $k < r$, that is:

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T, A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T = U_k S_k V_k^T, A \approx A_k \tag{6.10}$$

where δ_i is the i-th singular value of A, and u_i and v_i are the corresponding singular vectors. The low rank approximation reveals hidden links between users or songs only latent in the original data matrix. Also, from a mathematical view, the latent vectors A_k obtained from the truncated SVD are the best rank-k approximation in the sense that the Frobenius norm $||A - A_k||_F$ is minimized [10].

From a practical point of view, truncated SVD is fast and the decomposition is unique which is a nice property that allows reproducible results. To create the embedding features based on truncated SVD, we proceed as follows. First, the sparse matrix A is created and then the k dimensions of the matrices U or V are extracted as embedding factors.

Feature X represents the row of the matrix and a concatenation of $Y_1, Y_2, Y_3, ...$ features represent the column of the matrix, called Y. Each entry of the matrix is equal to 1 if X and Y_1 or X and Y_2 or X and $Y_3, ...$ appear in the data.

By setting the X and Y matrices with different features, the latent factors stored in U and V matrices are extracted as follows:

1. Set X = user_id and Y = song_id with $k = 35$, return both U and V matrices as embedding features
2. Set X = user_*id* and Y = artist_name with $k = 5$, return both U and V matrices as embedding features
3. Set X = user_*id* and Y = source_type with $k = 10$, return only U as embedding features
4. Set X = song_id and Y = source_type with $k = 10$, return only U as embedding features
5. Set X = user_*id* and Y = genre_*id* with $k = 10$, return only V as embedding features
6. Set X = user_id and Y_1 = song_id, Y_2 = artist_name, Y_3 = genre_id, Y_4 = source_type with $k = 10$, return only U as embedding features
7. Set X = song_id and Y = song_id with $k = 5$, return only U as embedding features. In this matrix for each user, we took the last 30 songs s/he listened to (it may be a re-listening or not). Two songs s_1 and s_2 have entry equal to 1 if a user listened to both in his last 30 listenings. The idea behind this matrix creation is to overcome the cold-start problem related to the songs.

Two important issues must be addressed in the use of truncated SVD above: 1) how to set the value for the parameter k (the number of embedding factors to consider), and 2) whether to use matrix U or V

for the embedding space of the users, songs or artists. The optimal k value, or the choice between U or V matrix, is obtained by repeated trials, taking the value, which maximizes the prediction accuracy, which for this task was the area under the ROC curve.

By using the embeddings for users, we anticipated that the ones who listened to the same songs or artists will be close to each other in the new embedding space. Also, we anticipated that the songs and artists will also be close to each other in the new embedding space if they were listened to by the same song.

The overall structure for adding engineering and embedding features on extended profile is shown in figure 6.10.

6.4 TRAINING AND VALIDATION

For this problem to examine the performance of the features more quickly, we used a subset of the training data (the last 41% of observations) to create our own training and validation sets. From this subset, the last 877417 observations were used for validation and the rest for the training set. The created validation and training sets have the same user and song distribution as it appears in the original training and test data (in order to have the same unseen user and song distributions). Within this schema, every time that we had an improvement on our local validation set, it was guaranteed that we will get the same improvement on the original test data. Also, as we have to examine lots of features, having a small validation schema for the test set is vital. As a result, we came up with indices 4400000 to 6500000 for training and from 6500001 to 7377417 for validation.

Using the complete training set as input, we computed all of the features described above and fed these results into our classification model for the test set prediction.

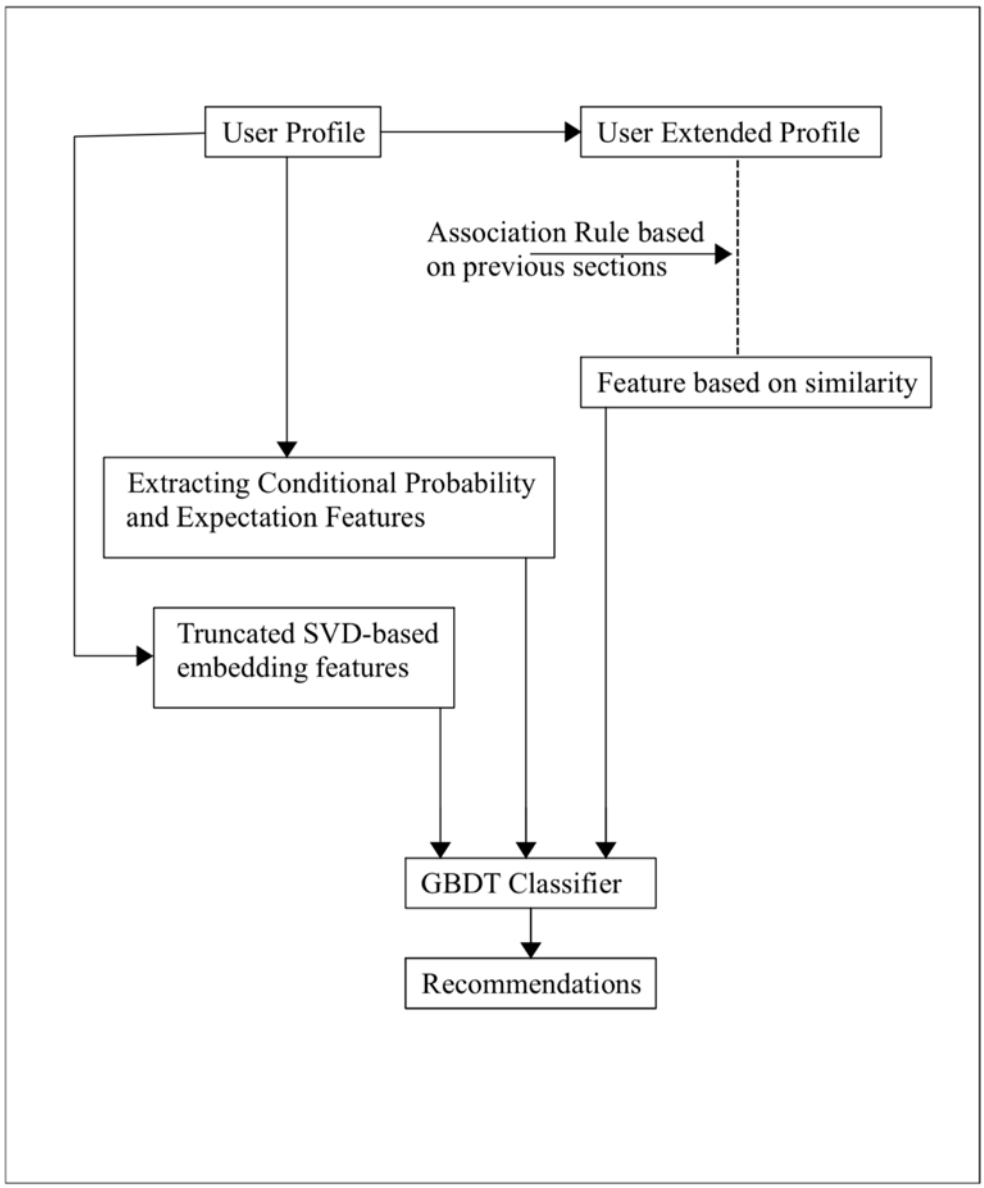


Figure 6.10: *The overall structure of adding engineering and embedding features to the Classifier*

6.4.1 Model Selection and Tuning

As the response matrix in our example includes only 0 and 1, we cast this problem as a binary classification problem, there are various learning algorithm than can be used: logistic regression, support vector machines (SVM), neural nets, random forest, gradient boosting decision trees, etc. For this problem, we used Microsoft LightGBM implementation of gradient boosting decision trees as our classifier which was presented as NIPS'17 due to its simplicity and superior accuracy in many real world applications [36].

We have a total number of 185 features as described above. A subset of those features was used to train five different models with different hyper parameters. There were 80 features in common between all models, but the rest were different. The average Pearson correlation of the five-model prediction was around .89, which gives us a good boost in prediction accuracy by using blending or stacking, described in the following section.

For model parameters tuning, after several experiments, we found that the best performance was achieved for the parameters summarized in Table 6.9. Parameters used in Table 6.9 are described in the Github repository [3].

Amazon EC2 c5.4xlarge-c5.9xlarge instances were used for validation-training with 16-36 CPUs and 32-72 GB-RAM respectively. Each validation run took around 20 minutes and the time for training was around one hour only.

6.4.2 Blending and Stack Generalization

In general stacking is ensemble of models combined sequentially. Blending is just averaging the output predictions of each model with different weights. While both techniques have improved the score in this

[3] https://github.com/Microsoft/LightGBM

Table 6.9: *Model Parameters*

Parameter	Values		
learning_rate	0.1	0.1	0.1
bagging_fraction	0.9	0.8	0.8
sub_feature	0.8	0.4	0.4
min_hessian	50	500	1000
max_depth	90	63	16
num_leaves	511	200	250
num_rounds	850	80	900

problem, we used blending approach due to its simplicity and slightly better results on our validation set.

Bagging method (averaging predictions from single models with the same features and the same parameters but with different random seeds) was used for three of the models, and two of the models are just single run. The final model was the weighted average of the five models' predictions. To find the best weights for the model blending, We used an optimization function, which is based on Nelder-Mead, quasi-Newton and conjugate-gradient algorithms [9, 19, 28].

6.5 EVALUATION

The model achieved 0.74693 AUC on private leaderboard with the score difference of 0.00094 from the first place among 1081 teams. Table 6.10 shows the improvement of the AUC score as we replace the user, song and artist with their corresponding embeddings and after adding engineered features. Note that each row in this table adds a new feature to the features introduced in the rows above. Truncated SVD-based embedding with association rule mining features proved to be the most important and result in an increase of nearly 0.07 in the AUC score from the raw feature set.

Table 6.10: *Result on AUC score for different set of features*

Feature	**Validation AUC score**
Raw features without user_id, song_id, artist_name	0.65510
Raw features and four additional features: Duration(user_id), song_year, song_country and Yr(user_id)	0.66451
replacing user_id, song_id with their corresponding embedding	0.68781
replacing artist_name with its embedding	0.69122
adding the rest of embeddings describe in	0.72145
session, user, song and artist engineered features	0.74257
with association rule mining features	0.74920

Many efforts were made in this work to produce more predictive features instead of tuning model parameter. Feature selection was conducted after all feature values were calculated. This step was performed based on the importance of each feature to the overall prediction performance [16]. Table 6.10 shows the average importance gain for different set of features.

6.6 SUMMARY

In this chapter, an approach was proposed to address the cold-start problem through expansion of short user profiles. This improves the quality of the recommender system recommendations. The proposed approach helps expand user profiles with the help of available data in the system from other users (neighbours) and without having the user input more data in the system. Also, the solution was presented to the 2018 ACM WSDM recommender system challenge. Our team, *Magic Recommenders* won the competition. We were able to come up with a promising AUC score for this task. For future work, a promising area of research is

Table 6.11: *Average Importance Gain for different set of features*

Feature	**Importance Gain**
user_id and song_id embedding features	0.17778
artist_name embedding features	0.11949
Association rule mining features	0.14743
user engineered features	0.19250
song engineered features	0.18310
artist engineered features	0.09254

to explore further the user and song behaviour and interaction. Users typically have specific tastes when a new song by an artist is released but these taste change over time.

Chapter 7

Using Frequent item-set mining for Maintenance Issue Classification

7.1 INTRODUCTION

Facility Management (FM) activities are knowledge-intensive and require information to be gathered from a range of sources and integrated into a coherent understanding of a building. A significant, but difficult to use data source is occupant-generated complaints or Work Orders (WOs) describing issues requiring resolution by the FM team or requesting specific action. Also referred to as maintenance requests or occupant complaints, WOs are key forms of feedback, providing facility engineers with valuable insight regarding building operational performance, and their resolution forms much of the core work of an FM team [29]. As noted by Goins and Moezzi [29], the *"analytical and systematic study of complaints may offer new insights into many building-related concerns"*, further noting both the paucity of existing academic research on this topic and the need for *"more researcher, designer, and building management attention to the potential value*

of using occupant complaints as a tool for diagnosing what goes wrong in buildings, from the occupants' points of view".

Occupant-generated WOs are recognized as a good potential data to support FM activities. However the unstructured nature of written complaint descriptions is a key challenge in analysing and integrating this data within the FM domain [2, 3]. Furthermore, they rarely contain the specific information required by engineers to resolve reported issues, often requiring multiple trips to the field to resolve. First to identify the required trade, next to specifically define the problem and identify any specific parts or tools required, and finally to resolve the issue. The ability to ask each user topic-specific questions regarding the nature of their complaint will allow this specific information to be gathered. To this end, an automated system is being developed that analyses the unstructured text, classifies the work order by category (L1) and subcategory (L2) and prompts the user with FM-team generated follow-up questions in real time. This chapter compares the prediction accuracies of a number of machine learning classifiers. The contribution of this chapter is a detailed investigation of various machine learning and frequent itemset mining algorithms to analyze unstructured text from occupant-generated work orders and classify it with high accuracy into one of multiple classes. From a research standpoint, the key problem of this chapter is this classification in light of the limitations of previous studies on this type of data, which have had limited success. While 70-95% accuracy has been achieved for k=2 class data classification, class accuracies for multiple (k=7) classes have rarely exceeded an average of 70%. This is not adequate to support work order classification and thus the application of alternative techniques are warranted. This chapter focuses on the testing of classifiers using existing textual feature extraction methods and the development of a frequent itemset approach to classify the WO category and subcategory. The main objective of the task is to classify complaints or Work Orders (WOs) that describe issues requesting specific action. Given that the important features extracted by machine learning algorithms are mainly words that appear together, it should therefore be possible to address this task via application of frequent itemset algorithms to create a powerful yet simple learning algorithm. Following the presentation of the algorithms used and prediction results, this chapter presents

the strategy developed to integrate this FM information into an FM-enabled Building Information Model (BIM).

7.2 Machine Learning Algorithms for Textual Classification

In order to identify the most promising algorithms for WO textual classification, a survey of machine learning algorithms for similar problems was completed. Machine learning approaches are broadly classified into two categories: supervised [44] and unsupervised [34]. In supervised learning, correct data labels are available to permit the machine learning algorithm to develop classification hypotheses based on the training data labels. Unsupervised learning discovers hidden patterns or grouping in data with no such labels available.

Automatic text classification applies machine learning techniques to train an algorithm to extract features from a set of pre-labeled text documents and classify new documents based on their contents. Aggarwal et al. [1] present a summary of textual classification algorithms, highlighting decision trees, Bayesian networks, support vector machines (SVM). In order to use all of these techniques, features representing a text document need to be extracted from the training data. The most common features for representing text documents are a "bag of words". A simple way to select a bag of words is based on document frequency (DF) [73], which ranks and selects the words based on the number of training documents containing a word. However, such frequent words may not be able to distinguish different classes well because they may occur in all types of documents. A more commonly used feature selection method is information gain (IG) [73], which is a supervised feature selection method that maximizes the information gained by knowing the word is present or absent. To represent a document using a set of selected words, term weighting methods, such as term frequency (TF) within a document or term frequency and inverted document frequency (TF-IDF) can be used. The weighting information extracted by TF-IDF can efficiently use by a tree-based classifier for classification and numerous clustering and classification methods have used TF-IDF as an input feature [80].

Decision tree classifiers [62] are broadly adopted because they are non-parametric, robust to outliers,

and they can process mixed data types [51]. Random Forest [15] creates multiple decision trees (a "forest") from the training data as follows. For each decision tree, N cases are randomly sampled with replacement from the original training data to form a training set, where N is the number of entries in a dataset. At each node, a subset of m features are selected at random and the best features are used to split the node. Each tree is grown to the largest extent possible with no pruning. New instances are classified by each tree and a class probability is assigned for each. These probabilities from each tree are then used to develop an overall prediction. Random forests have been frequently used for recommender systems development, including a recent study to incorporate user feedback [77]. By combining decision trees into an ensemble model, Random Forests have demonstrated increased performance due to reduced model variance without increasing bias. It has been shown that randomization can de-correlate the trees in the ensemble [51], resulting in a highly effective classification method. Based on these qualities, Random Forest techniques prove themselves promising for textual classification problems.

Bayesian classification [27] considers networks of probabilities of feature independence to predict the most likely class. Hierarchical classification leverages Bayesian approaches and several have been considered for textual classification. Hierarchical classification approaches used in the literature include [20], in which a hierarchical SVM classifier is combined with Bayesian classification, document classification using Bayesian networks [40]. In a survey of studies comparing hierarchical and flat classification techniques, Zimek et al. [82] note that results vary but improve when localized feature selection is present and there are relationships between features. Since decision tree methods incorporate localized feature selection, this suggests a potential benefit to hierarchical classification by problem type and subcategory versus flat classification by subcategory.

The development of a successful algorithm will enable the generation of new WO reporting systems to both structure and improve the quality of problem description data and this forms the motivation for the classifier development presented in this chapter.

7.3 OVERAL STRUCTURE

The overall methodology used in this chapter consists of data structure analysis and cleaning, identification of potential machine learning classifiers, preprocessing, modeling and model tuning. For WOs, the data structure consists of the problem classification structure used to both identify necessary sub-trade(s) to resolve the work issue, as well as subcategories describing specific types of problems, equipment involved, and/or types of action required. Data cleaning is necessary to create the testing and training datasets used for classifier development. These datasets cannot contain those WOs that cannot be accurately classified, for example, those containing blank work descriptions or containing multiple (different) types of work requests, for example, "the light in my office is burnt out". Also, "there is a stained ceiling tile". The latter WOs can be divided into multiple requests, each associated with a single subtype, but kept as a whole; these cannot be correctly classified into a single category by a domain expert and thus cannot be properly labelled. Data preprocessing also includes the deletion of irrelevant fields (columns) and removal of punctuation and common but meaningless words such as "are" from the work description. A lemmatization (word stemming) process is finally used to group words with similar meanings together.

The identification of potential classifiers is strongly related to the data structure identified. As noted in the literature review, the Random Forest algorithm (a decision tree method) and rules-based classifiers show significant promise, as does a Bayesian or hierarchical approach when the data structure itself has hierarchy and meaningful relationships between levels.

In order to train and test each classifier, testing (2015 WOs) and training (2010-2014) datasets are developed. Each classifier is further tuned to determine the optimal parameters by using cross-validation within the training set. Once tuned, the test data is used to test each classifier and results are analysed using confusion matrices, which indicate the frequency of predicted vs actual data labels and confounding between categories. Accuracy statistics, including both the prediction accuracy and the no information rate (NIR, sometimes referred to as null accuracy), which represents the proportion of the data points included

in the largest class and thus the best guess results, must also be considered in evaluation, along with the probability that the calculated prediction accuracy is not greater than the NIR. The detailed execution and model tunning is unique to each data set/data structure, thus the methodology is presented as applied to the case study in the next section.

7.4 CASE STUDY

7.4.1 Data Restructuring and Cleaning

The G1 dataset used in [69] included 32 problem type categories, each with numerous subcategories, many of which overlapped as described previously. A revised data structure was co-developed with the domain expert (Ryerson Facility Engineer) along with specific follow-up questions to obtain targeted details from the occupant based on the type of problem reported. This additional information is critical to support root cause analysis algorithm development and is anticipated to reduce the time to resolve each WO due to the avoidance of the information gathering initial field visit by general maintenance staff.

The restructuring eliminated duplicate or overlapping categories, grouped related subcategories, and removed all -Miscellaneous categories and associated WOs (approx. 70,000), which were reviewed and found to be similar enough to the remaining WOs that their omission from the Generation 2 (G2) test/train dataset was unlikely to significantly change the extracted features. The revised data structure consisted of two levels. The top category (L1) label denotes the problem category, which is often named for the associated trade (i.e. HVAC, Plumbing). The subcategory (L2) label describes the type of problem occurring within the defined category, for example *L1 — L2 could be HVAC — Temperature or Plumbing — Drain.*

A sample G2 classification structure for the Plumbing category is shown in Figure 7.1. Note that while most of the original G1 subcategories have been grouped under new L2 labels, the Plumbing Misc. subcategory was excluded from the new structure since previous research indicated that the presence of such

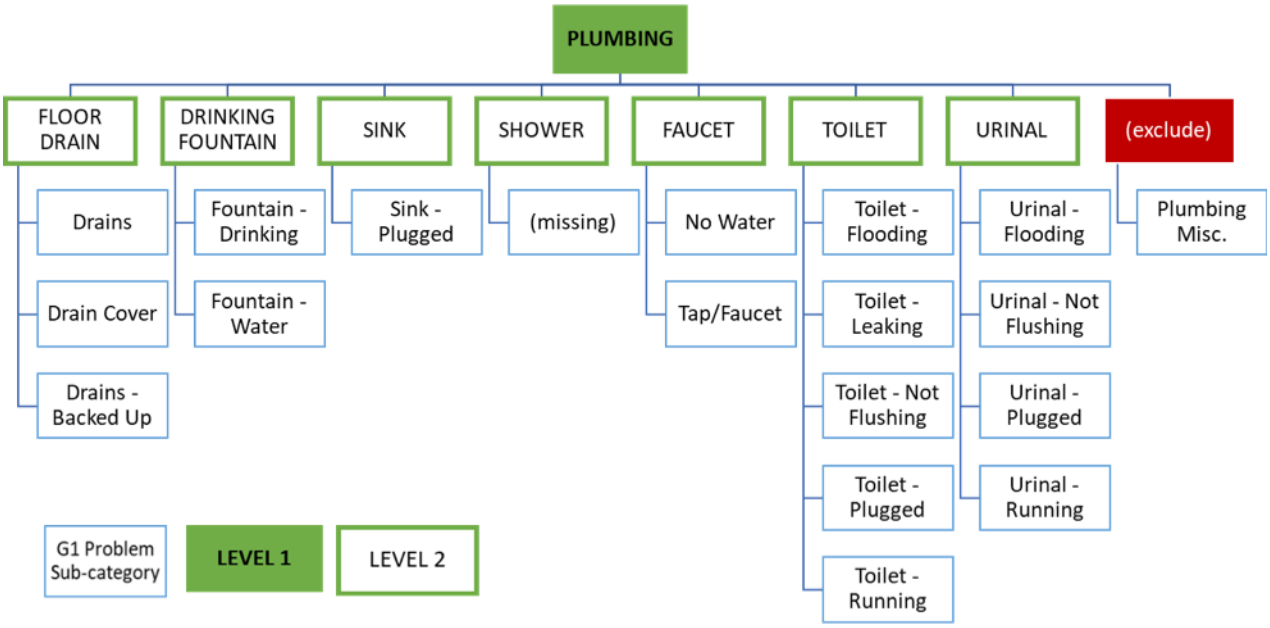


Figure 7.1: *Example reclassification of problem type categories*

catch-all categories significantly reduced classification accuracy. Once reclassified, custodial requests were excluded as this project focused on maintenance tasks only. Second, work orders covering multiple issues from unrelated subcategories, for example "the light in my office is burnt out". Also, "there is a stained ceiling tile" were removed; these were consistently poorly classified due to the presence of multiple feature words from multiple categories.

7.4.2 L1 Classifier Development

Based on the review of available machine learning approaches, along with a review of the dataset structure used in the case study the authors determined that tree-based algorithms and frequent itemset mining techniques are the most promising avenues of investigation. TF and TF-IDF have been widely used in information retrieval [63] and a classifier was developed using the features extracted using these approaches. These constitute the first three classifiers selected and used in the first phase of research.

Two types of approaches were used to develop the L1 classifiers: (1) methods classifying according to the frequency of mined keywords in each category, and (2) those using frequent itemsets identified with the frequent itemset mining algorithm. In the first set of approaches, two methods were applied to generate representative words: TF and TF-IDF, with the random forest algorithm using the latter. In the second set, word frequencies were analysed using frequent itemset mining techniques and used to develop support counts for classification. All model development and testing was performed using R.

Three learning methods were used to learn representative words for each problem type category and integrated into algorithms to classify WOs based on problem descriptions. In the Term Frequency (TF) method, the set of work descriptions for each problem type was analysed and the most frequently used words ("representative words") for each category were determined and used to represent that category. Each unlabelled WO was then scored against each category based on the number of representative words from that category present and assigned to the category with the highest score.

In the Term Frequency-Inverse Document Frequency (TF-IDF) model, the prevalence of representative words within the work description were weighted in inverse proportion to overall category frequency - i.e. those words occurring primarily in a single problem type category were weighted more heavily than those occurring across all work description categories. In this method, the weight of the frequency term i in problem type category j is weighted using the log of the ratio of the total number of problem type categories over the number of categories in which term i appears.

Finally, a Random Forest classifier was built using the features (10 distinct representative words per category) extracted from work descriptions using the TF-IDF method. Decision trees were not pruned in order to permit fair comparison between standard and hierarchical RF approaches; in the latter approach, the inclusion of pruning would result in different tree depths for each L1 category.

A new algorithm (Frequent Itemset Analysis or 'FIA') was developed and applied within the final two classifiers. FIA was inspired by the observation that several representative words, for example {ceiling},

Table 7.1: *Sample* support *matrix*

V1	V1	V2	V2	V3	V3	V4	V4
FFE	FFE	PLUMBING	PLUMBING	HVAC	HVAC	LOCK + KEY	LOCK + KEY
{tabl}	0.27487	{toilet}	0.46030	{cold}	0.35109	{key}	0.66081
{door}	0.23203	{washroom}	0.20671	{hot}	0.30755	{lock}	0.27587
{set}	0.22098	{clog}	0.19827	{room}	0.16254	{attach}	0.24554
{room}	0.18934	{plug}	0.18433	{temperatur}	0.14204	{form}	0.19899
{floor}	0.15657	{urin}	0.13851	{heat}	0.13718	{attach,form}	0.17558
{chair}	0.15291	{drain}	0.13155	{air}	0.13380	{attach,key}	0.16361

were present in multiple categories, but sets of keywords tended to be unique. For example, itemsets such as {lights}, {outlet},{ceiling, lamp} and {breaker} have high frequency counts in electrical and itemsets like {washroom}, {leak}, {ceiling, water} and {plugged} are frequent in the Plumbing problem type. In this algorithm, every work description is considered a transaction, resulting in over 44,000 transactions to review. Given N categories $(k_1, k_2, ..., k_N)$, the frequent itemset mining algorithm was used on the training set to calculate the *support* - the percentage of the transactions in the training data set in the relevant category that contains the itemset - of each frequent itemset. A support threshold of 2% was used to select itemsets, which were stored along with their supports in a *scoring matrix*, illustrated for $m = 6$ frequent itemsets (within each category) and $N = 4$ classification categories in Table 7.1.

Once the scoring matrix has been created, the FIA algorithm classifies the test data as follows. First, all category scores (denoted as $CS_i, 1 \leq i \leq N$) is initialized with zero. When a new work order is presented during the prediction phase, the category scores $CS_i, 1 \leq i \leq N$, are calculated as follows: for each frequent itemset j in each category in the support matrix, the transaction text is checked to determine whether the itemset is present. If so, the support value from the support matrix is added to the CS_i. Finally, the record is classified based on the maximum CS_i. This algorithm has significant advantages over simple term frequency

approaches due to its consideration of feature supports, while having a significantly lower computational cost than RF classifiers.

A sliding window variation was also applied to FIA, which formed the final classifier 'FIA + sliding window'. This approach generated new frequent item sets using recent transaction subsets and applied them for rule development and subsequent testing. A subset size of $N = 1000$ was used and the full 2010-2015 dataset was re-split into 80% training, 20% testing. The support matrix was then developed using the first 1000 chronological transactions, and tested on the following 1000. This was repeated for the full dataset, allowing the most recent data to be used to develop the support prediction matrix for each instance and thus considering short-term trends in language used to describe particular maintenance issues.

7.4.3 L2 Classifier Development

The models applied for L1 were developed and tested on L2 data using the methods described above. To limit computational cost, only the most common cause codes - those accounting for 90% of the WOs - were considered in L2 classifier development.

To take advantage of the two-level structure of the category labels, a hierarchical approach was developed in Python to investigate whether representative words created using TF-IDF L1 subsets (by class) rather than full data would be a) substantially different, and b) more effective for prediction. In this hierarchical approach, a random forest model was first trained on the L1 categories. For each L1 category, a subset was created and a new random forest model was trained on each to predict L2 labels. To reduce computational cost for the hierarchical clustering, each WO was transformed into a vector representation with each element denoting the occurrences of each word in the WO to create the TF-IDF matrix. This matrix has high dimensionality and 11 random forest models were trained, so singular value decomposition was used to reduce its dimension and speed up the training process. One L1 classifier and ten L2 classifiers were developed for this hierarchical model; L1 categories without sub-classification did not require rules for further sorting and

thus did not result in L2 classifiers. Dimensionality reduction was necessary to reduce computational cost for L2 classification; to do so, a 10% sample of the dataset was used with the full complement of representative words to identify those most significant. The top 20 representative words were used to build the random forest classifier on the full dataset. All models were then combined using conditional probabilities as follows. Suppose that C_2 denotes the L2 categories and $y_{1,2}$ are the correct labels for L1 and L2, respectively, the probability of C_2 of a correct L2 prediction is:

$$P(C_2 = y_2) = P(C_2 = y_2|C_1 = y_1) * P(C_1 = y_1) \tag{7.1}$$

Using the trained models, the prior $P(C_1 = y_1)$ and posterior $P(C_2 = y_2|C_1 = y_1)$ probabilities are computed and combined using Equation 7.1 to find the probability that $C_2 = y_2$. The predicted label y_{pred} is the category that results in the highest probability. Both random sampling (90% train; 10% test) and time series distribution (historical 2010-2014 train data; 2015 testing data) were used to separate testing sets with negligible impact on accuracy for this technique.

7.5 EVALUATION

7.5.1 L1 Classifier Results

Classification models used in the initial data analysis (TF, TF-IDF, and RF using TF-IDF) were revisited with the full G2 dataset to determine whether the reclassification allowed the full range of categories to be modelled with better accuracy than the most-frequent categories modelled in previous work [69]. In that previous study, classifiers were trained and tested on the data subset containing the k=5 most common classes, representing 70% of all WOs. The results from this study are compared with the results for the same data subset as well as the $k = 11$ classes representing 99% of all WOs in Table 7.2. Note that a 99% threshold was used to avoid classes with too few samples for adequate training. The FIA classifier was developed in the second phase of this research and was not tested on the G1 dataset.

Table 7.2: *L1 classifier performance for all categories*

	G1 Dataset , $k = 5$ (50% of WOs)		G2 Dataset, $k = 3$ (50% of WOs)		G2 dataset, $k = 11$ (99% of WOs)	
	Accuracy	NIR	Accuracy	NIR	Accuracy	NIR
TF	0.5789	0.247	0.8305	0.315	0.5838	0.2306
TF-IDF	0.6007	0.2442	0.7977	0.3041	0.6353	0.2290
Random Forest	0.7047	0.2698	0.8946	0.3671	0.8602	0.2482
FIA	Not tested in this phase		0.8161	0.3671	0.7223	0.2487
FIA w/ Sliding Window	Not tested in this phase		0.8472	0.3951	0.6788	0.2833

Table 7.3 breaks down these results by class for the full G2 dataset. Both FIA algorithms consistently achieve class accuracies of 90% or higher for three of the most common categories, while RF achieves this for six of the top eight. The worst results across all classifiers were related to the EQUIPMENT category, which included equipment of all types (HVAC, Electrical, Plumbing, etc.) and had several confounded features with the other categories. Contrasting previous results, the term frequency methods used on G2 data produced notably different representative word fragments for the full spectrum of problem types. The resultant L1 classification accuracies for the reclassified categories containing 50% of all work orders (G2, $k = 3$) significantly exceed those obtained in the initial work (G2, $k = 5$) for the same volume of work orders. This difference in performance is most notable when a high number of classes (G2, $k = 11$, representing 99% of all WOs) is considered, which also has higher classification accuracy than the initial $k = 5$ most common categories. In all cases, the Random Forest algorithm was the best overall performer, followed by both FIA variants.

As noted above, the two models using the FIA algorithm compared well to one another, with less than 5% accuracy difference overall, regardless of the number of L1 categories included in the data. The class-by-class comparison presented in Table 7.3 indicates that while some categories benefited from the sliding window

approach when compared with FIA, most showed a marked decrease in class accuracy using this approach, most likely due to the smaller training dataset. Note that Prevalence is the true percentage of items in a class, Detection Rate refers to the percentage of true positive predictions, and Detection Prevalence refers to percentage of positive predictions (true and incorrect); note that all percentages are based on the total number of items in the dataset. The Balanced Accuracy considers both model sensitivity (percentage of items predicted in the correct class) and specificity (percentage of items correctly not predicted to be in the wrong class).

The confusion matrices for the five L1 classifier models are shown as heatmaps in Figure 7.2. These have been normalized, ranging from 0% of an actual class (black) to 100% of the class (white). The output of a perfect algorithm would have white squares on the diagonal (where the predicted (row) = actual (column) labels, and black squares in all non-diagonal cells. Classifiers using the TF and TF-IDF feature extraction techniques have a high misclassification rate as shown by the off-diagonal entries in the respective confusion matrices. The two FIA classifiers perform noticeably better, however the confusion matrix shows a high misclassification of Leak issues as Finishes. This is not unexpected since both contain subcategories related to ceiling tiles and thus there are significant overlaps in the frequent itemsets contained within these categories. This is again evident in a close observation of the class accuracy of the Finishes category in Table 7.2. The RF classifier shows the best overall performance, and the confusion matrix shows minimal clustering of erroneous results.

7.5.2 L2 Classifier Performance

Table 7.3 summarizes the L2 classifier performance for the five models previously discussed considering only the top k categories accounting for 99%, 90%, and 70% of all WOs, respectively. Table 7.4 provides overall model performance for the top k=10 subcategories. Comparing these results to those obtained at L1, the L2 predictions improved when the number of categories included was similar (k=13 for L1; k=10 for L2),

Table 7.3: *Class accuracy of L1 classifiers (8 most common L1 categories)*

		ELECTRICAL	EQUIPMENT	FFE	FINISHES	HVAC	LIGHTING	LOCK + KEY	PLUMBING
TF	Prevalence	0.054	0.035	0.231	0.079	0.111	0.17	0.109	0.132
	Det. Rate	0.028	0.017	0.136	0.039	0.082	0.064	0.081	0.102
	Det. Prev.	0.035	0.053	0.266	0.103	0.122	0.08	0.145	0.126
	Balanced Accuracy	0.758	0.727	0.712	0.714	0.847	0.677	0.832	0.873
TF-IDF	Prevalence	0.054	0.034	0.229	0.079	0.11	0.171	0.109	0.133
	Det. Rate	0.043	0.022	0.109	0.062	0.089	0.111	0.089	0.107
	Det. Prev.	0.065	0.049	0.129	0.099	0.113	0.136	0.136	0.134
	Balanced Accuracy	0.885	0.804	0.726	0.871	0.892	0.807	0.881	0.887
Random Forest	Prevalence	0.055	0.032	0.248	0.078	0.117	0.162	0.105	0.128
	Det. Rate	0.044	0.024	0.227	0.067	0.093	0.149	0.086	0.113
	Det. Prev.	0.048	0.039	0.303	0.072	0.101	0.156	0.096	0.122
	Balanced Accuracy	0.900	0.863	0.905	0.930	0.89	0.955	0.903	0.937
FIA	Prevalence	0.055	0.032	0.249	0.078	0.117	0.034	0.162	0.106
	Det. Rate	0.033	0.008	0.142	0.065	0.083	0.030	0.147	0.097
	Det. Prev.	0.060	0.010	0.204	0.096	0.094	0.066	0.167	0.135
	Balanced Accuracy	0.782	0.615	0.744	0.898	0.845	0.925	0.942	0.939
FIA + Sliding Window	Prevalence	0.045	0.033	0.283	0.102	0.148	0.027	0.152	0.091
	Det. Rate	0.030	0.017	0.162	0.071	0.094	0.026	0.129	0.086
	Det. Prev.	0.056	0.027	0.230	0.097	0.103	0.086	0.147	0.144
	Balanced Accuracy	0.820	0.745	0.739	0.836	0.811	0.941	0.914	0.943

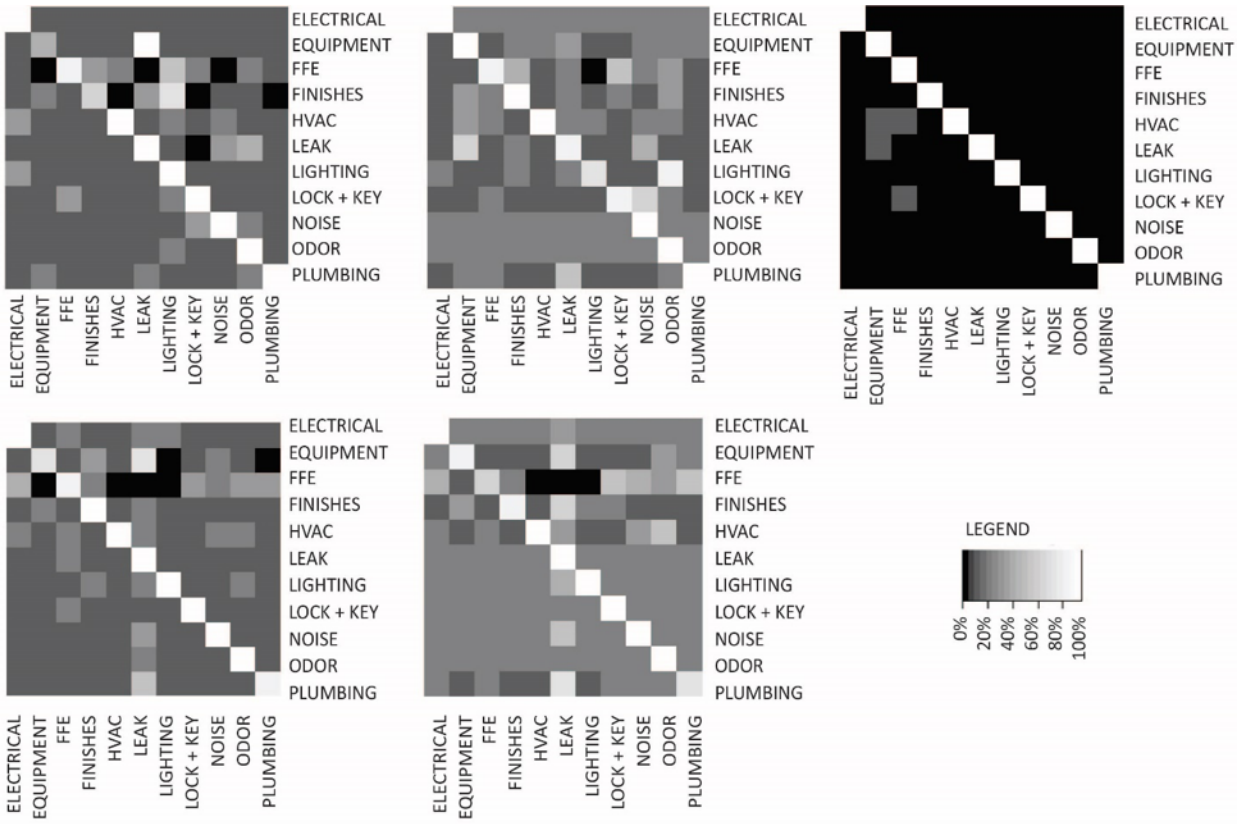


Figure 7.2: *Confusion matrices for L1 classifier models (99% of cases): TF (top left), TF-IDF (top centre), Random Forest (top right), FIA (bottom left) and FIA with sliding window (bottom right)*

Table 7.4: *L2 classifier accuracy*

	k = 32 (99% of records)		k = 20 (90% of records)		k = 10 (70% of records)	
	Accuracy	NIR	Accuracy	NIR	Accuracy	NIR
TF	0.4822	0.1377	0.5161	0.1491	0.6393	0.1846
TF-IDF	0.5528	0.1385	0.6286	0.1500	0.7413	0.1850
FIA	0.4695	0.1346	0.634	0.1457	0.8146	0.1778
FIA + sliding window	0.4564	0.2833	0.6902	0.1938	0.8212	0.1687
Random forest	0.8398	0.1348	0.8478	0.1461	0.900	0.1779
Hierarchical Random Forest	0.804	N/A	0.840	N/A	0.8829	N/A

on the order of 4-6% for TF, TF-IDF, and RF classifiers, and 8-12% for FIA, due to the specificity of the subcategories. The FIA algorithm outperformed both TF and TF-IDF, for the k=10 and k=20 tests but this benefit was lost at higher numbers of classes.

Figure 7.3 presents the confusion matrices for the six L2 classifier models for the $k = 10$ (70% of WOs) subcategory classes. In these confusion matrices, the poor relative performance of the TF algorithm compared to the remaining algorithms is evident by the significant amount of grey (indicating 10-40% classification rates) in incorrect categories. The increase in performance from TF to TF-IDF is substantial, and increases substantially for Random Forest, but decreases for hierarchical (prediction errors from both steps compounded). Overall, RF provided the best predictions for fully unlabelled data.

Comparing the hierarchical and direct RF techniques, it is notable that the hierarchical classification results were slightly worse than the direct L2 classification using the random forest model. Upon review of the keywords identified using the TF-IDF for both the overall dataset and within each L1 category, fewer than 2% of keywords changed. The prediction accuracy of each L2 subcategory calculated using conditional probabilities is summarized in Table 7.5. No clear relationship was observed between the number

Table 7.5: *Hierarchical Prediction Accuracy of L2 Models using (Random Forest)*

L1 Category	WOs in L1	# L2 within L1 category	P(C1=y1) is correct	P(C2=y2 — C1=y1) is correct	P(C2=y2) is correct
Electrical	1790	2	80%	99%	79%
Equipment	3355	3	74%	88%	66%
FFE	14253	7	91%	89%	81%
Finishes	4108	2	100%	98%	98%
Fire & Life Safety	200	2	87%	100%	87%
HVAC	5241	3	79%	100%	79%
Leak	1070	3	67%	98%	66%
Lighting	3497	4	92%	74%	68%
Lock + Key	4203	5	82%	87%	71%
Noise	861	1	85%	100%	85%
Odour	386	1	84%	100%	84%
Plumbing	5434	6	88%	88%	78%

of subcategories of L2 within L1 or the number of data points in L1, and prediction accuracy.

Overall, the hierarchical model is able to achieve high predictive accuracies for L2 categories only if L1 categories are given, and is extremely sensitive to poor L1 class accuracy. The range of prior $P(C_1 = y_1)$ class prediction accuracies was 67%-100% (weighted average of 86.7%), while the posterior $P(C_2 = y_2|C_1 = y_1)$ was higher, ranging from 75%-100% (weighted average of 90.5%). For categories with perfect L1 class accuracy, such as L1= Finishes, there is no impact on results. Even a 10% misclassification rate, for example the L1 = FFE or Lighting significantly affected the results due to the multiplicative probability effect, and this was most pronounced in the L1= Leak category, which had 98% posterior prediction accuracy for subclasses but an overall 66% prediction accuracy due to the very low (67%) L1 class accuracy. The Lighting category remains a source of significant prediction error, indicating that some L2 subcategories remain confounded in the new work order structure.

Given a correct L1 class, the L2 prediction accuracy $P(C_2 = y_2|P_1 = y_1)$ is slightly higher than the non-hierarchical approach, indicating a slight improvement in performance due to more targeted representative words developed within each L1 category for L2 classification. Since this research began, the work order system has been updated to prompt the user to indicate the L1 category most closely associated with their request, which will permit deployment of this classifier. Similarly, the FIA algorithm achieved accuracy above 90% with sliding window when the problem type was given.

7.6 SUMMARY

This chapter investigated a series of classifier models tested to predict WO subcategories based on unstructured and highly variable occupant-generated WOs. A new efficient and low computational cost algorithm was also developed with frequent itemset mining called FIA. FIA algorithm not only predict classes with more than 90% accuracy, but the running time is twenty times faster than random forest and memory usage is also one third of random forest. This forms the basis for a long-term research project to not only classify

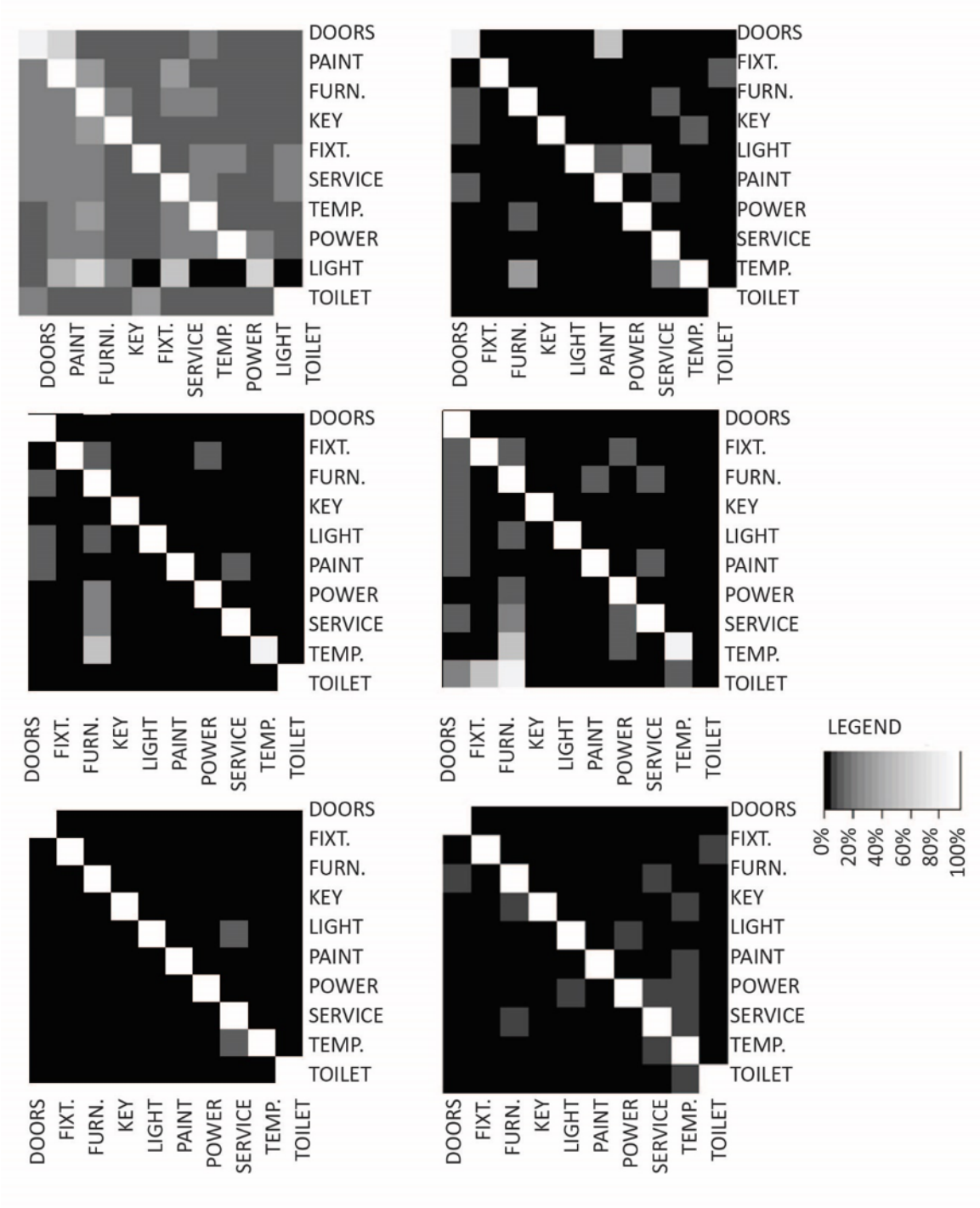


Figure 7.3: *Confusion matrices for L2 models (top row: TF (left), TF-IDF (right), middle row: FIA (left) and FIA + sliding window (right); bottom row: Random Forest (left), and hierarchical (right)*

and visualize these WOs, but also permit the enhancement of the WO reporting system to prompt the user with targeted follow-up questions in real time. This will enable the structured collection of additional information necessary to assign priority and urgency levels, prioritize FM response to WOs, and facilitate root cause identification and analysis. The follow-up questions have been developed and approved for each maintenance-related L2 subcategory by the Facilities Engineer and Maintenance Manager; those developed for the LEAK — FLOOD complaint are listed following:

1. Where is the leak coming from? (e.g, the ceiling, wall, radiator, window or a floor)
2. Is the liquid pooling on the floor or any other surface?
3. Please describe the leaking liquid (e.g, is it water, colored liquid, a chemical, etc.)
4. Is there a visible source to the flood (e.g, leaking faucet, overflowing sink/toilet, ceiling leak, etc.)?
5. Is there electrical equipment/outlets in close proximity to the water?
6. Do you believe the flood water is contaminated with sewage?
7. Do you know when the flood started?

The best-performing classifiers for the top 10 subcategories accounting for more than 70% of the total work orders were the FIA classifiers without a 1000-transaction sliding window and the hierarchical Random Forest when problem type was provided by the user. Both such classifiers exceeded 90% overall accuracy. When the full dataset was considered, random forest using TF-IDF, which achieved high (90%) accuracy on the $k = 10$ classes, and moderately high (82%) classification accuracy on the full dataset, providing the best results for the less-common categories. The random forest classification model will be integrated into the pilot recommender system to identify the top subcategory prediction, with the L2 hierarchical classifier used as a check. Because system users will be prompted to confirm that the classification is correct, and select a more appropriate classification if incorrect, as part of the recommender system, it will be possible

to monitor the real-time performance of these models. This will permit further increase in accuracy through the integration of online learning to refine the representative words used to minimize classification error.

The relatively low prediction accuracy within some categories such as Lighting indicates the need to review and refine the L2 subcategory definitions to reduce misclassification. Further refinements to the recommender system will include the assignment of priority based on a combination of problem subcategory, specific trigger words, issue location, and/or clustering of repeated WOs. Further, because the frequent itemset feature extraction had higher accuracy than TF-IDF for up to $k = 20$ classes, the development a random forest classifier using frequent itemsets rather than TF-IDF representative words offers significant potential and warrants future research.

Chapter 8

CONCLUSIONS

The herein presented thesis proposes a new method, namely SPFP-tree (single pass frequent pattern tree), for incremental construction of FP-trees via a single pass of a given data set. The proposed algorithm rearranges the tree on the fly, arranging items in each branch of the tree in frequency-descending order (just as proposed in the FP-tree algorithm) after each transaction is added. SPFP-tree performance analysis results were compared against those corresponding to other algorithms for incremental mining, including CanTree and CP-tree. The attained results demonstrate that SPFP-tree outperforms CP-tree for all analyzed datasets on various support thresholds, while outperforming CanTree on lower support thresholds. Moreover, SPFP-tree is herein demonstrated to be a more memory efficient alternative to CanTree owing to its dense frequency-descending prefix-tree structure. The feasibility of the algorithm for incremental and interactive mining is also presented.

As discussed throughout this thesis, memory size is considered a major limiting factor in large database mining. Previous solutions, such as those presented by incremental mining methods, generally involve

division of datasets into subsets, which can then be mined separately. In this thesis, a different approach is presented to address this issue, wherein the storage of large collections of frequent itemsets is carried out in the form of a compact graph. The herein proposed method utilizes a single node per each distinct item in the database, therefore reducing the amount of memory required for a given mining task. Experimental results corroborate the usefulness of the presented theoretical model by demonstrating that the proposed algorithm delivers performance superior to those of FP-Growth and Can-Tree in terms of memory use, while delivering otherwise comparable results in terms of run time.

Further, an upper bound for the number of nodes for a given tree is proposed as part of this work as a way to also regulate the memory requirements of alphabetical and FP-tree based structures. These upper bounds are calculated based on the total number of items as well as the total number of distinct items present within a given database. The upper bound for the alphabetical tree is provided as a simple algorithmic routine, while the upper bound of the FP-tree is formulated in a closed form. It is anticipated that the proposed upper bounds will be implemented in a wide variety of applications encompassing varied fields.

As part of this thesis, an approach was proposed to address the cold-start problem of recommender systems through expansion of short user profiles. Such an expansion would in turn improve the quality of generated user recommendations across a variety of platforms. The proposed approach helps expand user profiles by using available data in the system from other users (neighbors), circumventing the need for further user input into the system. Of note, the proposed solution was presented in the 2018 ACM WSDM recommender system challenge, earning a promising AUC score.

This thesis also investigated a series of classifier models aimed at predicting WO subcategories from unstructured and highly variable occupant-generated WOs. The main objective of this task included practical classification of complaints or Work Orders (WOs) that describe issues requesting specific action. Automatic text classification applies machine learning techniques such as decision trees, Bayesian networks, and support vector machines (SVM) to train an algorithm to extract key features from a set of pre-labeled text documents

as a way to classify new documents based on their contents. Given that the key features extracted by these algorithms are mostly attained through analysis of words that appear together in a text, a simple and yet powerful frequent itemset mining approach was herein used to tackle this problem. The attained experimental results show the effectiveness of this technique in classifying WOs.

The following publications have been produced during the course of this research period.

- Shahbazi, Nima, Rohollah Soltani, and Jarek Gryz. "Memory Efficient Frequent Itemset Mining." In Proceedings of the International Conference on Machine Learning and Data Mining in Pattern Recognition, pp. 16-27. Springer, 2018.
- Shahbazi, Nima, Rohollah Soltani, Jarek Gryz, and Aijun An. "Building FP-Tree on the Fly: Single-Pass Frequent Itemset Mining." In Proceedings of the International Conference on Machine Learning and Data Mining in Pattern Recognition, pp. 387-400. Springer, 2016.
- McArthur, J. J., Nima Shahbazi, Ricky Fok, Christopher Raghubar, Brandon Bortoluzzi, and Aijun An. "Machine learning and BIM visualization for maintenance issue classification and enhanced data collection." Advanced Engineering Informatics, 38 (2018): 101-112.
- Shahbazi, Nima, Mohammed Chahhou, and Jarek Gryz. "WSDM Cup 2018: Truncated SVD-based Feature Engineering for Music Recommendation", the Eleventh ACM International Conference on Web Search and Data Mining. ACM, 2018.

8.1 FUTURE WORKS

A possible future direction of this work regards further developments in twitter data mining via application of the techniques presented in chapters three and four, with further improvements to these techniques enabling the search of frequent patterns among thousands of tweets within very short time periods. While

many techniques have been presented to date to mine special patterns in twitter data, one of main barriers preventing future advancements in the speed and utility of such methods has been the underwhelming simplicity and efficiency of said techniques, a shortcoming the currently presented work may be able to easily address.

Given the overall trends observed in technology nowadays, whereby databases continue expanding in size while computers gain more processor cores rather than faster ones, ample consideration should be given to adapting the proposed algorithms for parallel compatibility. If the above-mentioned observed tendencies persist, parallel algorithms, which capitalize on these trends, will likely play a key role in technologies aimed at maximizing the use of computation resources; as such, adaptation of the herein proposed algorithms for parallel compatibility would be of large benefit in such endeavors. As well, given the novelty of the newly presented algorithm described in Chapter 4, an in-depth optimization analysis of said algorithm and its structure (Chapter 4) would allow for more concrete determinations regarding the current structure, and possibly enable further optimizations to its structure aimed at reducing computation costs.

Another promising area of research regards further developments in recommender systems directed at generation of song recommendations through further exploration of user and song behavior and interaction. For instance, while the probability of a user enjoying a given new song by an artist upon release is generally known to be largely dictated by the users specific music taste at that moment, musical taste is also known to change over time. Algorithms aimed at frequently assessing such changes would thus generate better recommendations over time.

Further research into the performance of the proposed approaches for other types of dataset may shed further light into the level of performance said approaches are capable of delivering. Given that the WSDM dataset employed for this research is quite sparse in terms of rating, dense datasets should be employed to evaluate the level of reductions achievable by the proposed hierarchical redundancy removal approaches.

Further refinements to the recommender system will include the assignment of priority based on a com-

bination of problem subcategory, specific trigger words, issue location, and/or clustering of repeated work orders. Further, because the frequent itemset feature extraction had higher accuracy than TF-IDF for large number of classes, the development of random forest classifier using frequent itemsets rather than TF-IDF representative words offers significant potential and warrants future research.

Finally, we propose further work into the development of average memory requirement for the number of nodes of alphabetical and FP-tree based structures. Such fine-tuning would enable better predictions concerning expected memory consumption for these well-known structures, a task that would definitely be of immense value, particularly for practical applications that require consistent storage and analysis of large amounts of data. Certainly, the recently observed acceleration in the amount of big data storage, as well as the increasing need for fast, frequent, and efficient analysis of frequent patterns in such large databases are a great indication of the usefulness of such an application.

Bibliography

[1] Aggarwal, C. C. and Zhai, C. (2012). A survey of text classification algorithms. In *Mining text data*, pages 163–222. Springer.

[2] Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *Acm sigmod record*, volume 22, pages 207–216. ACM.

[3] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A. I., et al. (1996). Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307–328.

[4] Agrawal, R., Srikant, R., et al. (1994). Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499.

[5] Ahmed, A., Kanagal, B., Pandey, S., Josifovski, V., Pueyo, L. G., and Yuan, J. (2013). Latent factor models with additive and hierarchically-smoothed user preferences. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 385–394. ACM.

[6] Aizawa, A. (2000). The feature quantity: an information theoretic perspective of tfidf-like measures. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 104–111. ACM.

[7] Amir, A., Feldman, R., and Kashi, R. (1997). A new and versatile method for association generation. In *European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 221–231. Springer.

[8] Ayan, N. F., Tansel, A. U., and Arkun, E. (1999). An efficient algorithm to update large itemsets with early pruning. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 287–291. ACM.

[9] Bélisle, C. J. (1992). Convergence theorems for a class of simulated annealing algorithms on rd. *Journal of Applied Probability*, 29(4):885–895.

[10] Berry, M. W., Drmac, Z., and Jessup, E. R. (1999). Matrices, vector spaces, and information retrieval. *SIAM review*, 41(2):335–362.

[11] Blake, C. (1998). Uci repository of machine learning databases. *http://www. ics. uci. edu/˜ mlearn/ML-Repository. html*.

[12] Blake, C. and Merz, C. J. (1998). Uci repository of machine learning databases [http://www. ics. uci. edu/˜ mlearn/mlrepository. html]. irvine, ca: University of california. *Department of Information and Computer Science*, 55.

[13] Borgelt, C. (2003). Efficient implementations of apriori and eclat. In *FIMI03: Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations*.

[14] Borgelt, C. and Kruse, R. (2002). Induction of association rules: Apriori implementation. In *Compstat*, pages 395–400. Springer.

[15] Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

[16] Breiman, L. (2017). *Classification and regression trees*. Routledge.

[17] Brin, S., Motwani, R., Ullman, J. D., and Tsur, S. (1997). Dynamic itemset counting and implication rules for market basket data. *Acm Sigmod Record*, 26(2):255–264.

[18] Bykowski, A. and Rigotti, C. (2001). A condensed representation to find frequent patterns. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 267–273. ACM.

[19] Byrd, R. H., Lu, P., Nocedal, J., and Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208.

[20] Cesa-Bianchi, N., Gentile, C., and Zaniboni, L. (2006). Hierarchical classification: combining bayes with svm. In *Proceedings of the 23rd international conference on Machine learning*, pages 177–184. ACM.

[21] Cheung, D. W., Han, J., Ng, V. T., and Wong, C. (1996). Maintenance of discovered association rules in large databases: An incremental updating technique. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 106–114. IEEE.

[22] Cheung, W. and Zaiane, O. R. (2003a). Incremental mining of frequent patterns without candidate generation or support constraint. In *Database Engineering and Applications Symposium, 2003. Proceedings. Seventh International*, pages 111–116. IEEE.

[23] Cheung, W. and Zaiane, O. R. (2003b). Incremental mining of frequent patterns without candidate generation or support constraint. In *Database Engineering and Applications Symposium, 2003. Proceedings. Seventh International*, pages 111–116. IEEE.

[24] Cremonesi, P., Koren, Y., and Turrin, R. (2010). Performance of recommender algorithms on top-n recommendation tasks. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 39–46. ACM.

[25] Deng, Z., Wang, Z., and Jiang, J. (2012). A new algorithm for fast mining frequent itemsets using n-lists. *Science China Information Sciences*, 55(9):2008–2030.

[26] Deng, Z.-H. and Lv, S.-L. (2015). Prepost+: An efficient n-lists-based algorithm for mining frequent itemsets via children–parent equivalence pruning. *Expert Systems with Applications*, 42(13):5424–5432.

[27] Devroye, L., Györfi, L., and Lugosi, G. (2013). *A probabilistic theory of pattern recognition*, volume 31. Springer Science & Business Media.

[28] Fletcher, R. and Reeves, C. M. (1964). Function minimization by conjugate gradients. *The computer journal*, 7(2):149–154.

[29] Goins, J. and Moezzi, M. (2013). Linking occupant complaints to building performance. *Building Research & Information*, 41(3):361–372.

[30] Golub, G. H. and Van Loan, C. F. (2012). *Matrix computations*, volume 3. JHU Press.

[31] Han, J., Pei, J., and Yin, Y. (2000a). Mining frequent patterns without candidate generation. In *ACM sigmod record*, volume 29, pages 1–12. ACM.

[32] Han, J., Pei, J., and Yin, Y. (2000b). Mining frequent patterns without candidate generation. In *ACM Sigmod Record*, volume 29, pages 1–12. ACM.

[33] Han, J., Pei, J., and Yin, Y. (2000c). Mining frequent patterns without candidate generation. In *ACM sigmod record*, volume 29, pages 1–12. ACM.

[34] Hastie, T., Tibshirani, R., and Friedman, J. (2009). Unsupervised learning. In *The elements of statistical learning*, pages 485–585. Springer.

[35] Hipp, J., Güntzer, U., and Nakhaeizadeh, G. (2000). Mining association rules: Deriving a superior algorithm by analyzing todays approaches. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 159–168. Springer.

[36] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154.

[37] Koenigstein, N., Dror, G., and Koren, Y. (2011). Yahoo! music recommendations: modeling music

ratings with temporal dynamics and item taxonomy. In *Proceedings of the fifth ACM conference on Recommender systems*, pages 165–172. ACM.

[38] Koh, J.-L. and Shieh, S.-F. (2004a). An efficient approach for maintaining association rules based on adjusting fp-tree structures. In *International Conference on Database Systems for Advanced Applications*, pages 417–424. Springer.

[39] Koh, J.-L. and Shieh, S.-F. (2004b). An efficient approach for maintaining association rules based on adjusting fp-tree structures. In *International Conference on Database Systems for Advanced Applications*, pages 417–424. Springer.

[40] Koller, D. and Sahami, M. (1997). Hierarchically classifying documents using very few words. Technical report, Stanford InfoLab.

[41] Konstan, J. A., Miller, B. N., Maltz, D., Herlocker, J. L., Gordon, L. R., and Riedl, J. (1997). Grouplens: applying collaborative filtering to usenet news. *Communications of the ACM*, 40(3):77–87.

[42] Koren, Y., Bell, R., and Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, (8):30–37.

[43] Kosala, R. and Blockeel, H. (2000). Web mining research: A survey. *ACM Sigkdd Explorations Newsletter*, 2(1):1–15.

[44] Kotsiantis, S. B., Zaharakis, I., and Pintelas, P. (2007). Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24.

[45] Leung, C. K.-S., Khan, Q. I., Li, Z., and Hoque, T. (2007a). Cantree: a canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems*, 11(3):287–311.

[46] Leung, C. K.-S., Khan, Q. I., Li, Z., and Hoque, T. (2007b). Cantree: a canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems*, 11(3):287–311.

[47] Leung, C. K.-S., Khan, Q. I., Li, Z., and Hoque, T. (2007c). Cantree: a canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems*, 11(3):287–311.

[48] Li, Z. and Zhou, Y. (2005). Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM.

[49] Liu, G., Lu, H., and Yu, J. X. (2007a). Cfp-tree: A compact disk-based structure for storing and querying frequent itemsets. *Information Systems*, 32(2):295–319.

[50] Liu, G., Lu, H., and Yu, J. X. (2007b). Cfp-tree: A compact disk-based structure for storing and querying frequent itemsets. *Information Systems*, 32(2):295–319.

[51] Louppe, G. (2014). Understanding random forests: From theory to practice. *arXiv preprint arXiv:1407.7502*.

[52] Mannila, H. (1997). Inductive databases and condensed representations for data mining. In *ILPS*, volume 97, pages 21–30.

[53] Mannila, H. and Toivonen, H. (1997). Levelwise search and borders of theories in knowledge discovery. *Data mining and knowledge discovery*, 1(3):241–258.

[54] Martinsson, G., Gillman, A., Liberty, E., Halko, N., Rokhlin, V., Hao, S., Shkolnisky, Y., Young, P., Tropp, J., Tygert, M., et al. (2010). Randomized methods for computing the singular value decomposition (svd) of very large matrices. *Works. on Alg. for Modern Mass. Data Sets, Palo Alto*.

[55] Matos-Junior, O., Ziviani, N., Botelho, F., Cristo, M., Lacerda, A., and da Silva, A. S. (2012). Using taxonomies for product recommendation. *Journal of Information and Data Management*, 3(2):85.

[56] Melville, P. and Sindhwani, V. (2011). Recommender systems. In *Encyclopedia of machine learning*, pages 829–838. Springer.

[57] Meyer, C. D. (2000). *Matrix analysis and applied linear algebra*, volume 71. Siam.

[58] Mnih, A. (2011). Taxonomy-informed latent factor models for implicit feedback. In *Proceedings of the 2011 International Conference on KDD Cup 2011-Volume 18*, pages 169–181. JMLR. org.

[59] Park, J. S., Chen, M.-S., and Yu, P. S. (1995). *An effective hash-based algorithm for mining association rules*, volume 24. ACM.

[60] Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L. (1999). Discovering frequent closed itemsets for association rules. In *International Conference on Database Theory*, pages 398–416. Springer.

[61] Pei, J., Han, J., Lu, H., Nishio, S., Tang, S., and Yang, D. (2001). H-mine: Hyper-structure mining of frequent patterns in large databases. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 441–448. IEEE.

[62] Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, 1(1):81–106.

[63] Ramos, J. et al. (2003). Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142.

[64] Savasere, A., Omiecinski, E. R., and Navathe, S. B. (1995). An efficient algorithm for mining association rules in large databases. Technical report, Georgia Institute of Technology.

[65] Shahbazi, N., Soltani, R., Gryz, J., and An, A. (2016). Building fp-tree on the fly: Single-pass frequent itemset mining. In *Machine Learning and Data Mining in Pattern Recognition*, pages 387–400. Springer.

[66] Takács, G., Pilászy, I., Németh, B., and Tikk, D. (2008). Matrix factorization and neighbor based algorithms for the netflix prize problem. In *Proceedings of the 2008 ACM conference on Recommender systems*, pages 267–274. ACM.

[67] Tan, P.-N., Kumar, V., and Srivastava, J. (2002). Selecting the right interestingness measure for association patterns. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 32–41. ACM.

[68] Tanbeer, S. K., Ahmed, C. F., Jeong, B.-S., and Lee, Y.-K. (2009). Efficient single-pass frequent pattern mining using a prefix-tree. *Information Sciences*, 179(5):559–583.

[69] Volk, R., Stengel, J., and Schultmann, F. (2014). Building information modeling (bim) for existing buildingsliterature review and future needs. *Automation in construction*, 38:109–127.

[70] Wang, J. T., Zaki, M. J., Toivonen, H. T., and Shasha, D. (2005). Introduction to data mining in bioinformatics. In *Data Mining in Bioinformatics*, pages 3–8. Springer.

[71] Weng, L.-T., Xu, Y., Li, Y., and Nayak, R. (2008). Exploiting item taxonomy for solving cold-start problem in recommendation making. In *Tools with Artificial Intelligence, 2008. ICTAI'08. 20th IEEE International Conference on*, volume 2, pages 113–120. IEEE.

[72] Yan, X., Han, J., and Afshar, R. (2003). Clospan: Mining: Closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 166–177. SIAM.

[73] Yang, Y. and Pedersen, J. O. (1997). A comparative study on feature selection in text categorization. In *Icml*, volume 97, pages 412–420.

[74] Zaki, M. J. (2000). Scalable algorithms for association mining. *IEEE transactions on knowledge and data engineering*, 12(3):372–390.

[75] Zaki, M. J. and Gouda, K. (2003). Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335. ACM.

[76] Zaki, M. J. and Hsiao, C.-J. (2002). Charm: An efficient algorithm for closed itemset mining. In *Proceedings of the 2002 SIAM international conference on data mining*, pages 457–473. SIAM.

[77] Zhang, H.-R. and Min, F. (2016). Three-way recommender systems based on random forests. *Knowledge-Based Systems*, 91:275–286.

[78] Zhang, Y., Ahmed, A., Josifovski, V., and Smola, A. (2014). Taxonomy discovery for personalized

recommendation. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 243–252. ACM.

[79] Zhao, Q. and Bhowmick, S. S. (2003). Association rule mining: A survey. *Nanyang Technological University, Singapore*.

[80] Zhao, Y., Karypis, G., and Fayyad, U. (2005). Hierarchical clustering algorithms for document datasets. *Data mining and knowledge discovery*, 10(2):141–168.

[81] Ziegler, C.-N., Lausen, G., and Schmidt-Thieme, L. (2004). Taxonomy-driven computation of product recommendations. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 406–415. ACM.

[82] Zimek, A., Buchwald, F., Frank, E., and Kramer, S. (2010). A study of hierarchical and flat classification of proteins. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 7(3):563–571.