POLITECNICO DI TORINO

Master's Degree in Mathematical Engineering

Master's Degree Thesis

TOPOLOGICAL DATA ANALYSIS AND FREQUENT ITEMSETS MINING: POTENTIAL SYNERGIES AND CURRENT CHALLENGES

Supervisors

Prof. Francesco VACCARINO

Prof. Luca CAGLIERO

Candidate

Rocco ELIA

October 2022

Abstract

Frequent itemset mining is an established data mining technique focused on discovering recurrent item combinations from transactional data. The extraction process can be modelled as an exploration of a lattice representing high-order item relations and is typically driven by ad hoc quality metrics (e.g., the support index). The exploration of the candidate item relations can be addressed using established graph generalization, called simplicial complexes. This thesis work analyzes the similarities between simplicial complexes and frequent itemsets, discusses the formal relations between them, and prospects for new and more advanced synergies between topological analyses and pattern mining techniques.

KEYWORDS: Frequent Itemset Data Mining Topological Data Mining Association Rules Simplicial Complex Simplets

Acknowledgements

Desidero ringraziare i miei relatori Francesco Vaccarino e Luca Cagliero per la grandissima disponibilità e per il supporto che mi hanno dimostrato in questi mesi, per avermi introdotto a questo interessante argomento ed aiutato a comprenderlo in maniera approfondita.

Ringrazio i miei genitori e le mie sorelle per tutto il supporto morale che mi hanno dato in questi anni di università ed i miei amici per avermi reso questo percorso più leggero. Senza tutti voi non sarebbe stato possibile raggiungere questo traguardo.

Grazie di tutto!

Rocco Elia

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data mining is the process of extracting implicit or previously unknown information and patterns from large databases. It collect a variety of methods at the intersection of statistics, machine learning, database systems and more generally information theory. Common tasks of Data mining are frequent itemset mining, association

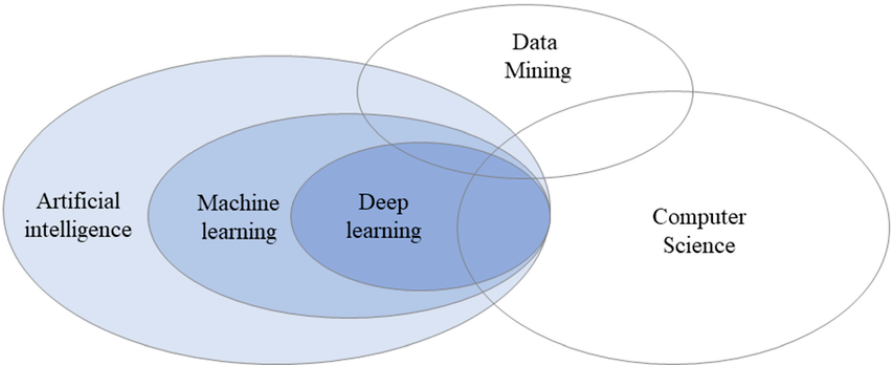


Figure 1.1: Relationship between Data Mining, AI/ML/DL,Computer Science. Taken from [Zha+21]

rules, clustering, classification, regression.
In this thesis we focus on frequent itemsets mining and association rules learning. First we provide an overview of Frequent Itemsets task: we will introduce a set of fundamentals definitions and introduce a classical application e.g. Association rules mining, providing some practical example.
Frequent Itemset Mining (FIM) is one of the most popular data mining methods and it aims to extract efficiently frequent occurring items in data that allow to extract knowledge relative to events and patterns. Insights from such pattern analysis offer important benefits in decision-making processes. We show the most popular FIM algorithms comparing them and showing their main limitations and

possible solutions.
In order to overcoming these limitation we present a different approach based on Topological data Analysis methods; Topological data analysis (TDA) is a recent and fast-growing field that provide a set of topological and geometric tools to infer relevant features for possibly complex data. We will compare theoretically this two different approach and we will report and discuss the most important algorithms showing the synergies between them.
Following the historical evolution of FIM task, in the first part of this work we introduce the basic definitions (database, item, itemset) and fundamentals metrics (support, confidence) defining th FIM task; then we will propose the three classical algorithms: Apriori, ECLAT and FP-Growth comparing their performance and discussing their advantages and limitations basing on experimental results.

Apriori algorithm, the first presented in this thesis, is also the first FIM algorithm developed ([AS+94]). Apriori works recursively on "horizontal database" and a support threshold (we explain and provide example of this database typology), computing support for each items and step by step on its combinations, filtering out the itemsets basing on the *Apriori Principle*, using the support threshold passed as input.
ECLAT (Equivalence Class Clustering and Bottom-Up Lattice, [Ban14]) is the second algorithm proposed: it's widely used nowadays and it aims to solve any important Apriori limitations: time consuming of frequent itemset research process and high memory cost. How we will observe, ECLAT take in input a "vertical database" and use bottom up approach like first depth search generating frequent items only once.
At the end of this part, we will present the FP-Growth algorithm [Sid+14] (where FP means "Frequent Patterns"): actually it is considered the state of art (excluding Topological based algorithm) for Frequent Itemset Mining problems (we will present its first implementation that which is improved over the years with some extension that we will mention). We can considered FP-Growth as an interesting application of "divide and conquer" paradigma; it use a different approach based on tree structure that allow to provide a compressed representation of the itemset database. As we will see, FP-Growth is the more efficient algorithm with respect to Apriori and ECLAT (even if under some conditions ECLAT has comparable performance).

In the second part of the thesis, we will face to Frequent Itemset Mining using a Topological approach. For first we will introduce the basic theoretical concepts of topology: topological and metrics spaces will be presented, then we will focus on simplex and simplicial complexes providing a set of theorems and properties useful for our scope. The simplicial complex is the core idea of the TDA algorithms presented for FIM: we can imagine it as a generalization of graph and the algorithms

presented aims to exploit this properties to find frequent itemesets. Hence we present and discuss the FreSCo Algorithm ([PDB22]) that is the state of art of application of TDA to FIM task and at the end we will propose an alternative solution that is the "Traversal Algorithm" ([Lin16]. Nowadays FreSCo is the state of art of the synergies between TDA and FIM, experimental results proof that it's less time and memory consuming with respect to the other classical algorithms. FreSCo use a different approach based on "simplets" concept that represent data structure in a more efficient way in case of high order relation dataset: frequent pattern mining in graph-structured data aims at finding structures that occur frequently in a given simplicial complexes. How we will observe, in case of a "simple" graph problem FreSCO is comparable to a graph pattern search (e.g. FP-Growth) but it is very powerful in case of high order relation database. The fundamental intuition is that any simplet represent a database relation in place of an edge (in a graph): this generalization is the key to overcomes the limitations of the classical FIM algorithms. The key idea of FreSCo algorithm is to extracts the frequent simplets in a simplicial complex, given a maximum size s* and a minimum dimension d* recursively expands each frequent simplet (using "widen" and "inflate" rules) until it becomes infrequent (pseudo-code implementation are provided in this section). As last step of this thesis we propose an alternative solution based on "Traversal Algorithm". The key idea of this algorithm is based on the geometrical concepts of cone and subcone construction that is applied recursively (we will explain in detail how it works), then the frequent itemsets mining search is transformed into geometric traversal problem. In the end we provide an example and a simple python implementation.

Chapter 2

Frequent itemset and association rules mining

2.1 Frequent Itemset Mining

Purpose of this chapter is to illustrate the main capabilities and applications of Frequent Itemset Mining (FIM).
Frequent Itemset Mining is a fundamental branch of Data Mining that collect a set of techniques in order to extract knowledge from data. Basically these techniques aim to find the most frequent subset of items within a database.
The most important application of FIM in data analysis is the so called "Association rules learning": in a few words it is a method for discovering interesting relations between variables in large databases, in order to do that Association rules use a Frequent Itemsets, so we can see Association Rules as a "second (and final) step" that follow FIM.

In this chapter will be defined the criteria for which a subset is considered "frequent"; the most used algorithms to search them will be presented.
Discovery of all frequent itemsets is a typical data mining task. The original use has been as part of association rule discovery.
In this chapter we will present the classical algorithms for finding frequent itemsets: Apriori, ECLAT, FP-Growth.

According to Agrawal's definition (1993) [Toi10] we can define **Frequent Itemset** in this way: given examples that are sets of items and a minimum support threshold, any set of items that occurs at least in the minimum number of examples is a frequent itemset [Toi10]. As already mentioned Frequent Itemset Mining is a set of techniques that aimed to extract frequent itemsets within a Database (these are also referred with the therm *knowledge discovery* and actually are widely used in a

lot of practical application).
For instance, customers of an on-line bookstore could be considered examples, each represented by the set of books he or she has purchased. A set of books, such as "Machine Learning," "The Elements of Statistical Learning," "Pattern Classification," is a frequent itemset if it has been bought by sufficiently many customers. Given a frequency threshold, perhaps only 0.1 or 0.01 for an on-line store, all sets of books that have been bought by at least that many customers are called frequent.

2.2 Basic definitions

In this section are proposed a set of definitions useful to define the basic metrics for FIM.
As will be seen in the following sections, these are the basis of the algorithms soon presented.

Transaction ID	**Items**
1	Bread, Milk, Beer
2	Beer, Diapers, Milk
3	Diapers, Beer

Table 2.1: Database

Definition 2.2.1. Let $I = \{i_1, i_2, i_3, ...i_n\}$ be a set of n binary attributes called **Items**.

Definition 2.2.2. An *Itemset* X is a subset of items of I ($X \subset I$); we call **k-itemset** an Itemset that contains a number k of items.

Definition 2.2.3. Let $D = \{d_1, d_2, d_3, ...d_n\}$ be a set of n transactions called **Database**.

Definition 2.2.4. The **Support** is defined as:

$$Supp(A) = \frac{number\ of\ transactions\ containing\ A}{total\ transactions}$$

Support is an important metric in Itemset mining, it intuitively indicates how frequently an item (e.g. "A") appears in a Data Base (effectively is the fraction of transactions that contain both A and B). For example consider the Table 1.1 the support of item "Milk" is $2/3 = 0.667$.
Based on this definition is possible to define:

Definition 2.2.5. Frequent Itemset is an Itemset whose support is greater than or equal to a pre-established threshold.
Consider the example table, the itemsets Beer,Diapers is "frequent" considering a support threshold of 0.6.

The search for them will be the focus of next chapters.

Definition 2.2.6. Confidence is defined as the ratio of the number of transactions that contains A and B on the total amount of transactions that contain A. With formula:

$$Conf(A \Rightarrow B) = \frac{number\ of\ transactions\ containing\ A\ and\ B}{number\ of\ transactions\ containing\ A}$$

In our example $Conf(Beer \Rightarrow Diapers) = 2/3$.

Definition 2.2.7. The measure of **Lift** between A and B is defined as:

$$Lift(A \Rightarrow B) = \frac{Supp(A \cup B)}{supp(A)Supp(B)}$$

If Lift$A \Rightarrow B$ is equal to 1, we can say that the occurrence in Database D of the itemset A and B are independent from each other; if Lift$A \Rightarrow B$ is greater than 1 we can say that the occurrence of A and B in D are positive dependent from each other and if it's less tahn 1 there is a negative dependence between them.
In our example $Lift(Beer \Rightarrow Diapers) = \frac{Supp(Beer \cup Diapers)}{supp(Beer)Supp(Diapers)} = 2/(3 * 2) = 1/3$.

Definition 2.2.8. We can define the **Conviction** as:

$$Conv(A \Rightarrow B) = \frac{1 - Supp(B)}{1 - Conf(A \Rightarrow B)}$$

this measure indicates the expected frequency that A occurs without B.

In our example $Conv(Beer \Rightarrow Diapers) = \frac{1 - Supp(Diapers)}{1 - Conf(Beer \Rightarrow Diapers)} = \frac{(1 - 0.667)}{(1 - 0.667)} = 1$.

2.3 Algorithms

In this section will be shown the most used Frequent Itemset mining algorithms. All of them given in input a Table and a support threshold and returns the Frequent Itemsets.
We will start from Apriori algorithm. Historically it was the first algorithm presented ([AS+94]), then we proceed to show ECLAT and FP-Growth.

2.3.1 Apriori algorithm

The goal of Apriori Algorithm is to find the *frequent itemsets* from a *Database* (as defined in previous section).
The inspiring idea of Apriori algorithm can be stated in this theorem:

Theorem 2.3.1. *(Apriori Principle). If an itemset is frequent, then all of its subsets must also be frequent.*

As a corollary of this principle, if an itemset is not frequent, then the sets containing it are not frequent either.
Apriori takes a Database and a support threshold as inputs and return requent Itemsets.
The support threshold has two important roles: functionally give a measure of "how frequent" are the discovered itemsets; technically this input parameter has a bearing on memory usage in the computation. So, using a too small value for support threshold risk to returns insignificant outputs (for first reason) and is also computationally expensive (for second reason); a value too large risk to returns poor results.

Apriori works with "horizontal table", a table formatted as a list of itemsets indexed by an ID (in the example below is called Transaction ID).

Transaction ID	Items
1	Bread, Milk, Beer
2	Bread, Deapers, Milk
3	Deapers, Chocolate
4	Bread, Beer, Deapers
5	Bread, Beer, Milk, Chocolate
6	Bread, Milk, Deapers, Beer

Table 2.2: Horizontal Database

Basically Apriori works recursively, computing support for each items and its combinations filtering out using the support threshold passed as input (how we say it is decided before).
Below is explained how it works step by step:

Step 1. For each item, support must be computed.
This basically just comes down to counting, for each product, in how many transactions it occurs (see definition 2.2.4 in previous section).

Step 2. Decide on the basis of support threshold: now that we have the support for each of the individual products, we will use that to filter out some of the products that are not frequent.

Step 3. Selecting the frequent items.

Step 4. Finding the support of the frequent itemsets: next step is to do the same analysis, but now using pairs of products instead of individual products. The number of combinations can quickly become large here, especially if we have a large number of products. The useful idea behind the Apriori algorithm is that we will directly ignore all pairs that contain any of the non-frequent items.This allow a great reduction of item pairs to scan, all of those can be filter out by support threshold and this will speed up the execution of our counts.

Step 5. Repeat for larger sets. We will now repeat the same thing for the sets with three products. As before, we will not score sets that were eliminated in the previous step.
Following an example code for Apriori Algorithm written in python using mlxtend (Machine Learning Extensions) library [Ras18] that use Apriori for Frequent Itemset computation and the lift and confidence calculus for Association Rules mining:

```
1  import pandas as pd
2  import numpy as np
3  from mlxtend.frequent_patterns import apriori, association_rules
4  import matplotlib.pyplot as plt
5
6  df = pd.read_csv('new 15.csv', sep=',')
7
8  items = set()
9  for col in df:
10     items.update(df[col].unique())
11 print(items)
12
13 itemset = set(items)
14 encoded_vals = []
```

```
for index, row in df.iterrows():
    rowset = set(row)
    labels = {}
    uncommons = list(itemset - rowset)
    commons = list(itemset.intersection(rowset))
    for uc in uncommons:
        labels[uc] = 0
    for com in commons:
        labels[com] = 1
    encoded_vals.append(labels)
encoded_vals[0]
ohe_df = pd.DataFrame(encoded_vals)

freq_items = apriori(ohe_df, min_support=0.2, use_colnames=True,
    verbose=1)
freq_items.head(7)

rules = association_rules(freq_items, metric="confidence",
    min_threshold=0.6)
rules.head()

#Support vs Confidence
plt.scatter(rules['support'], rules['confidence'], alpha=0.5)
plt.xlabel('support')
plt.ylabel('confidence')
plt.title('Support vs Confidence')
plt.show()

#Support vs Lift
plt.scatter(rules['support'], rules['lift'], alpha=0.5)
plt.xlabel('support')
plt.ylabel('lift')
plt.title('Support vs Lift')
plt.show()

#LIFT VS CONFIDENCE
fit = np.polyfit(rules['lift'], rules['confidence'], 1)
fit_fn = np.poly1d(fit)
plt.plot(rules['lift'], rules['confidence'], 'yo', rules['lift'],
 fit_fn(rules['lift']))
```

2.3.2 ECLAT

ECLAT (stands for "Equivalence Class Clustering and Bottom-Up Lattice") is a widely used algorithm for Frequent Itemset Mining. This algorithm is derived from Apriori and initially devised to solve its limits (time consuming of frequent itemset research process and high memory cost).

For first ECLAT try to solve the Apriori limitation generating frequent items only once [Ban14]. In fact the basic idea of ECLAT (and the first great difference with respect to Apriori) is to scan a Database "vertically" with a bottom up approach like first depth search. Furthermore ECLAT count only support in frequent itemset research process.
For this reason ECLAT need slightly different input table, e.g. the so called "vertical table" that is a table in which each tuple is composed by an item (first column in the example below) and a list of transaction i which the item appears . Notice the difference between a "vertical table" presented below with respect to a classical "horizontal table" (presented in previous section). This kind of table and a support threshold (for the choice of the support threshold the same considerations made for the Apriori) are the input parameter for ECLAT (if an horizontal table is passed as input a preliminary step is to transform it in a vertical table is needed); Frequent Itemsets are the output.

Itemset	**Transaction ID**
Bread	1, 2, 4, 5, 6
Milk	1, 2, 5, 6
Chocolate	3, 5
Beer	1, 4, 5, 6
Deapers	2, 3, 4, 6

Table 2.3: Vertical Database

Below we will explain how ECLAT works step by step:

Step 1. If an horizontal table is passed as input, make a list that contains, for each product, a list of the Transaction IDs in which the product occurs in order to obtain a vertical table.

Step 2. Take the minimum support value (support threshold) passed as input. The minimum support will serve to filter out products that do not occur often enough to be considered.

Step 3. Compute the Transaction ID set of each product pair: we now move on to pairs of products. We will basically repeat the same thing as in step 1, but now for

product pairs.
The interesting thing about the ECLAT algorithm is that this step is done using the Intersection of the two original sets. This makes it different from the Apriori algorithm.
In this step we can notice the advantages of ECLAT with respect to Apriori: ECLAT is faster because it is much simpler to identify the intersection of the set of transactions IDs than to scan each individual transaction for the presence of pairs of products (as Apriori does).

Step 4. Filter out the pairs that do not reach minimum support, as before, we need to filter out results that do not reach the minimum support.

Step 5. Continue as long as you can make new pairs above support From this point on, you repeat the steps as long as possible.

At the end of step 5 a Frequent Itemset is return as output.
It's useful to remark how the improvement of ECLAT with respect to Apriori resided in step 3 and it is based on different input table that is passed (or computed otherwise as we seen in step 1) to ECLAT that allow a different type of search.

2.3.3 FP-Growth

The last algorithm which we cover in this chapter is the so called FP-Growth algorithm (FP stands for Frequent Pattern) [HKP12]. FP-Growth is actually one of the state of the art algorithms for Frequent Itemset Mining and we present the first version (although it has been improved during the years, see for example the LCM2 version [UKA+04]). Given its efficiency FP-Growth is considered the base for other extension of FIM task (e.g. infrequent itemset mining [CG14] and "High-Utility Itemset Mining" [Fou+19])

FP-Growth is an improvement of the Apriori algorithm and it can be considered an application of "divide and conquer" paradigma.
To solve the weakness of Apriori, in FP-Growth a frequent pattern is generated without the need for candidate generation (this process as we have seen is a lot time consuming).
How we see in ECLAT, but in a still different way, the first difference betwen FP-Growth and Apriori (and thus also with respect to ECLAT) is the way in which FP-Growth represents the database. FP growth stores the record of table (passed as usually as input) in the form of a tree called "frequent pattern tree" (FP-Tree, from which the algorithm takes its name).
This tree structure (its construction will be explained later, in algorithm steps) will maintain the association between the itemsets and the database is fragmented using

one frequent item (fragmented part is called "pattern fragment"). The itemsets of these fragmented patterns are analyzed. Using this method, the search for frequent itemsets is reduced comparatively.

Technically, FP-Growth get a Database table in input and returns a Frequent Itemset as output and like ECLAT use a vertical table, then in case of horizontal table as input, the first step is to transform it (as we seen in ECLAT). Other input parameter is a support threshold.

Below the algorithm steps are presented:

Step 1. Counting the occurrences of individual items: the first step of the FP-Growth algorithm is to count the occurrences of individual items.

Step 2. Filter out non-frequent items using minimum support: as usually the support threshold is passed as input and basing on this every item or itemset with fewer occurrences than the minimum support will be excluded.

Step 3. Order the itemsets based on individual occurrences: for the remaining items, we will create an ordered table. This table will contain the items that have not been rejected yet, and the items inside a transaction will be ordered based on individual product occurrence.

Step 4. Create the tree and add the transactions one by one: now, we can create the tree starting with the first transaction. Each product is a node in the tree. The root of the tree is initialized to Null value (in Fig. 1.2 is shown FP-Tree for Table 1.2).

Step 5. Mine the tree: once this FP-Tree is constructed, it is much faster to traverse it and find information on the most frequent itemsets. In order to traverse FP-Tree and return the Frequent Itemsets is sufficient to scan the tree from the base node cutting of the node that not lead minimum support. Each sub-tree obtained are treated as a Database.

Example 2.3.2. Following an example of FP-Tree construction and mining (steps 4 and 5) starting from a given table:

Construction of the tree (Support Threshold chosen is 3 (minsup is 3/6=0,5):

1. As we say root node is null.
2. The first scan of Transaction T1: A, B, C contains three items $\{A : 1\}, \{B : 1\}, \{C : 1\}$, where B is linked as a child to root, A is linked to B and C is linked to A.
3. T2: B, C, D contains B, C, and D, where B is linked to root, C is linked to B and D is linked to C. But this branch would share B node as common as it is already used in T1.

Transaction ID	**Itemset**
T1	A, B, C
T2	B, C, D
T3	D, E
T4	A, D, B
T5	A, B, C, E
T6	A, B, C, D

Table 2.4: Database

Item	**Frequency**
B	5
A	4
D	4
C	3
E	2

Table 2.5: Frequency Table of Database

4. Increment the count of B by 1 and C is linked as a child to B, D is linked as a child to C. The count is $\{B:2\}, \{C:1\}, \{D:1\}$.

5. T3: D, E. Similarly, a new branch with E is linked to D as a child is created.

6. T4: A, B, D. The sequence will be B, A, and D. B is already linked to the root node, hence it will be incremented by 1. Similarly A will be incremented by 1 as it is already linked with B in T1, thus $\{B:3\}, \{A:2\}, \{D:1\}$.

7. T5:A, B, C, E. The sequence will be I2, I1, I3, and I5. Thus $\{B:4\}, \{A:3\}, \{C:2\}, \{E:1\}$.

8. T6: A, B, C, D. The sequence will be B, A, C, and D. Thus $\{B:5\}$, $\{A:4\}$, $\{C:3\}$, $\{D1\}$.

Mining of the tree: FP-Growth work recursively on each conditional tree removing on each iteration the items that does not lead the minimum support.

1. The lowest node item E is not considered as it does not have a min support count, hence it is deleted.

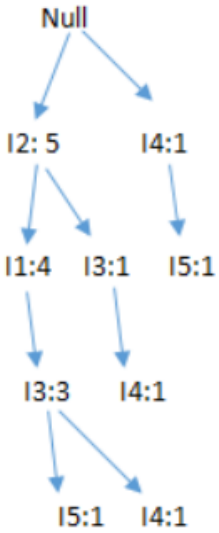


Figure 2.1: FP-Tree

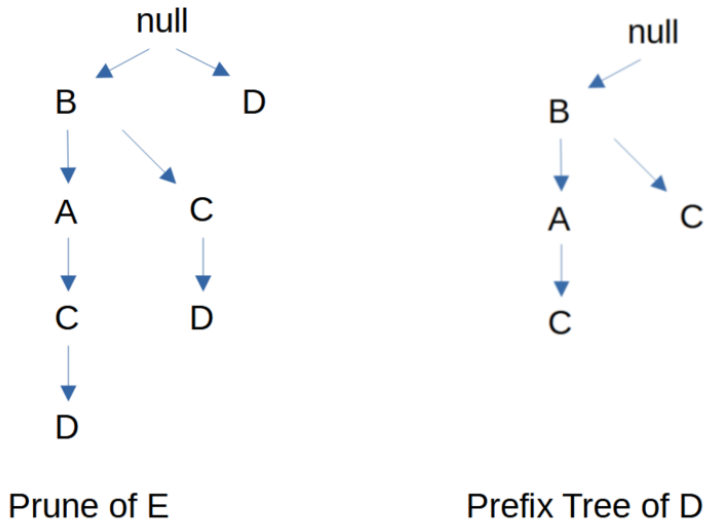


Figure 2.2: Prefix tree of D (E is pruned at first step): prefix tree represent a reduced version of Database (projected Database); applied this recursively (basing on minsup) we obtain the Database with only frequent itemsets (that are all the permutation of not-pruned items).

2. The next lower node is D. D occurs in 2 branches , {B,A,C:,D:1},{B,C,D:1}. Therefore considering D as suffix the prefix paths will be {B, A, C:1}, {B, C:

1}. This forms the conditional pattern.

3. The conditional pattern base is considered a transaction database, an FP-tree is constructed. This will contain {B:2, C:2}, A is not considered as it does not meet the min support count.

4. This path will generate all combinations of frequent patterns: {B,D:2},{C,D:2},{B,C,D:2}.

5. For C, the prefix path would be: {B,A:3},{B:1}, this will generate a 2 node FP-tree : {B:4, A:3} and frequent patterns are generated: {B,C:4}, {A:C:3}, {B,A,C:3}.

6. ForA, the prefix path would be: {B:4} this will generate a single node FP-tree: {B:4} and frequent patterns are generated: {B,A:4}.

Finally, Frequent Itemset generated are: F = {B,C}, {A,C},{B,A,C},{B,A}

Some implementation of FP-Growth are available on [Goe] or for distributed computation and the most used programming languages (python, R, Scala, Java) there is a Spark package under ml library (see [Spa] for complete documentation).

2.3.4 Algorithm comparison and experimental results

As mentioned before, Apriori is the first algorithm used for Basket Analysis. Actually is the less efficient in terms of computationally cost and memory usage. This is due to its breadth-first search approach: frequent itemset are searched with an iterative method that generate candidate at each step; in addition, at each step Apriori save a sub-table on memory (starting from input table) and it is memory consuming. Other notable point is that Apriori need to compute both support and confidence.
ECLAT and FP-Growth aims to overcome this limitation using a depth-first search and optimizing memory usage during their steps, both compute only support as metrics. FP-Growth as seen, generates conditional FP-Tree for every item in the data and requires only one scan of the database in its beginning steps: it consumes less time.
ECLAT Algorithm instead, has an approach similar to FP-Growth, and it's faster and less memory consuming than Apriori.
Between FP Growth and ECLAT there is no obvious winner in terms of execution times: generally it will depend on different data and different settings in the algorithm. But an interesting research show that FP-Growth in most cases is slightly faster than ECLAT [GK13].

Experimental result (see [GK13]) proof that on standard dataset FP Growth performs the best and Apriori takes the maximum execution time but when the number of transactions (rows in dataset) increase, Apriori is less sensitive than ECLAT and Apriori; when the number of attributes increase, ECLAT is comparable with FP-Growth (especially for medium-high value of support) and Apriori shows a sharp increase.

We can conclude that also for Itemset Mining and the algorithms presented presented previously the usual rule of Machine Learning algorithm is valid: there is no better algorithm but all depends on dataset.

2.4 From Frequent Itemsets to Association Rules

In this section we present one of the most useful application of Frequent Itemset Mining. More precisely, we describes how to extract association rules efficiently from a given frequent itemset.
The association rule mining task was first introduced by Agrawal ([AIS93a], [AIS93b]) to discover interesting relationships among items in market basket transactions. We can start from a definition of association rules:

Definition 2.4.1. Given A,B two different itemset of I a **Rule** is an implication $A \Rightarrow B$. Consider a dataset D, having n number of transactions containing a set of items. An association rule is the relationship between those items. An association rule is represented by $A \Rightarrow B$, where A and B are the distinct itemsets. The Association rule exposes the relationship between the itemset A with the itemset B.

Notice that for each frequent k-itemset, Y , can produce up to $2^k - 2$, association rules, ignoring rules that have empty antecedents or subsequent. [TSK16]

In a few words the role of Frequent Itemset Mining is to identify often occurring product combinations with a fast and efficient algorithm; based on its results is possible to derive association rules.

2.5 Association Rules and Basket Analysis

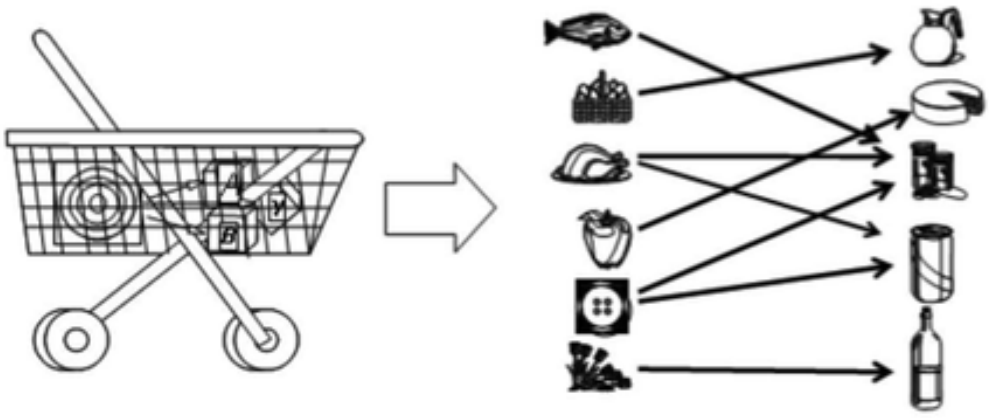


Figure 2.3: Market Basket Analysis. Taken from [httb]

One of the majors application of Frequent Itemset Mining is the *Association rule learning*, a rule-based machine learning method for discovering interesting

relations between variables in large databases. It is intended to identify strong rules discovered in databases using some measures of interesting.
Association rule-mining is usually a data mining approach used to explore and interpret large transactional database to identify unique patterns and rules. During transactions, these patterns define fascinating relationships and interactions between different items.Association rules help identify and forecast transactional behaviors based on information from training transactions utilizing beneficial properties. Using this approach, we can answer questions such as what items human beings tend to buy together, indicating frequent sets of goods.[Sub20].
How we just say, historically this kind of approach were used in the so called "Basket Market Analysis" application (originally used in purchasing problems, hence the name), that is the study of baskets in shopping.
A much-studied use case in basket analysis is to study products that are frequently bought together in order to find the **association rules** (so the first step of the analysis as it is easy to guess is the Frequent Itemset Mining).Purchases could be online or offline, with a basic requirements: any purchase must be tracked by a transaction.
This type of insight can be used for making recommendations to the customer for online shopping, or for re-arranging the products in a regular store so that it is easier for customers to add them to their basket.
A practical result example for association rules is $(Flour, Eggs \Rightarrow Milk)$: this means that observing the database of billing previously correlation is detected so typically a customer that but flour and eggs probably (in following section this concept will be explained) will buy also milk.

Example 2.5.1. Take for example the transactional table [**Tab 2.1**], consider the rule $\{Milk, Diapers\} \Rightarrow \{Beer\}$. Because the support count for $\{Milk, Diapers, Beer\}$ is 2 and the total number of transactions is 5, the rule's support is 2/5=0.4. The rule's confidence is obtained by dividing the support count for $\{Milk, Diapers, Beer\}$ by the support count for $\{Milk, Diapers\}$. Since there are 3 transactions that contain milk and diapers, the confidence for this rule is 2/3=0.67.

2.6 Association Rules generation

How we can say in previous sections the already shown FIM algorithms returns a Frequent Itemset as output: once we have the largest frequent itemsets, the next step is to convert them into *Association Rules*hence the frequent itemset generated by one of the previous algorithms becomes the input to search on the Association

Transaction ID	**Items**
1	Bread, Milk
2	Bread, Diapers, Beer, Eggs
3	Milk, Diapers, Beer, Cola
4	Bread, Milk, Deapers, Beer
5	Bread, Milk, Diapers, Cola

Table 2.6: Example 2.1 Table

Rules.
By definition (see previous section) association rules are written in the format: $ProductX \Rightarrow ProductY$. For example, in Market Basket Analysis, this means that we obtain a rule that tells us that if you buy product X, you are also likely to buy product Y.
How we already explained minimum support is applied to find all frequent itemsets in a data set (see the support threshold passed as input in FIM Algorrithms).
Purpose of this section is to compose the rules using the others fundamentals metrics (defined in Basic definitions section): **confidence**, **lift** and **conviction** are used as constraint in order to generate the rules. These measures define the "strenght" of an association rule; confidence is the most used measure, that we can formally state the association rules problem as:

Definition 2.6.1. Given a set of transactions T, find all the rules having $support \geq minsup$ and $confidence \geq minconf$, where minsup and minconf are the corresponding support and confidence thresholds.

How we can see in previous definition, asically association rules generation happens adding at Frequent itemset mining other steps that consider this last metrics. Hence for each metrics a threshold is set as parameter to be passed in input to the FIM algorithm chosen.

Follow are presented the steps that generated associations rules; starter point (or input) are the frequent itemset finding how explained in previous section:

Step 1. Compute confidence and Generate Association Rules: .
The confidence tells us a percentage of cases in which this rule is valid. Based on this metrics (computed for each frequent itemsets discovered) is possible to choose the rules which have a coefficient greater than the set threshold (passed as input).

Step 3. Compute lift. According to the definition, the lift of a rule is a performance metric that indicates the strength of the association between the products in the rule. This means that lift basically compares the improvement of an association

rule against the overall dataset.
If the lift of a rule is higher than 1, the lift value tells you how strongly the right hand side product depends on the left-hand side.
Once you have obtained the rules, the other step is to compute the lift of each rule. Any rule that has a lift of 1 (or near) can be discarded.

Step 2. Compute conviction. Unlike confidence, conviction factors always has a value of 1 when the relevant items are completely unrelated. In contrast to lift, conviction is a directed measure because it also uses the information of the absence of the consequent; conviction is monotone in confidence and lift.
In this step we compute conviction for all detected rules. At this point, we filter out the rules which have a coefficient lower than the set conviction threshold (passed as input).
In conclusion, once the Frequent Itemsets are detected, confidence is computed for all combination within frequent itemset in order to generate association rules based on confidence threshold; furthermore are computed lift and conviction for all the rules detected in order to discard trivial rules.

2.7 A general Implementation

Now we present a simple python implementation that use association-rules libraries.

```
from mlxtend.frequent_patterns import apriori, association_rules

def searchrules(frequent_items,threshold):
    rules = association_rules(frequent_items, metric ="confidence",
    min_threshold = threshold)
    rules = rules.sort_values(by='confidence', ascending =False)
    #print rules table
    rules
```

This piece of code take in input a frequent itemset (previously computed using any algorithm) and a threshold and returns a table that contain finded association rules. The set threshold is simply the confidence that we would pass to algorithm. Following a complete code that take in imput a dataset, extract Frequent Itemset and compute strong association rules.

```
import pandas as pd
import numpy as np
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules

```

```
# Create data

data = [['Apple', 'Banana', 'Guava', 'Eggs', 'Milk', 'Steak', 'Yogurt', 'Tomatos', 'Onion', 'Nuts'],
        ['Apple', 'Banana', 'Orange', 'Eggs', 'Milk', 'Pork', 'Potatos', 'Onion', 'Choclate'],
        ['Spinach', 'Eggs', 'Milk', 'Yogurt', 'Bagel', 'Onion', 'Watermelon'],
        ['Pineapple', 'Banana', 'Eggs', 'Lemon', 'Steak', 'Broccolli', 'Onion', 'Lamb'],
        ['Apple', 'Banana', 'Beer', 'Lettuce', 'Ice cream', 'Tomatos', 'Grape', 'Strawberry']]

# One-hot encoding

te = TransactionEncoder()
data_transformed = te.fit_transform(data)
df = pd.DataFrame(data_transformed, columns=te.columns_)
df

# Find frequent item sets
frequent_items = apriori(df, min_support = 0.6, use_colnames = True)


# Generate strong association rules
def searchrules(frequent_items,threshold):
    rules = association_rules(frequent_items, metric ="confidence", min_threshold = threshold)
    rules = rules.sort_values(by='confidence', ascending =False)
    #print rules table
    rules

#call the method
searchrules(frequent_items,0.8)
```

Chapter 3

Topological Backgrounds

3.1 Introduction

In mathematics Topology is concerned with the properties of a geometric object that are preserved under continuous deformations, such as stretching, twisting, crumpling, and bending; that is, without closing holes, opening holes, tearing, gluing, or passing through itself.
A topological space is a set endowed with a structure, called a topology, which allows defining continuous deformation of sub-spaces, and more generally all kinds of continuity. Euclidean spaces, and more generally metric spaces are examples of a topological space, as any distance or metric defines a topology. The deformations that are considered in topology are homeomorphisms and homotopies. A property that is invariant under such deformations is a topological property.
Basic examples of topological properties are: the dimension, which allows distinguishing between a line and a surface; compactness, which allows distinguishing between a line and a circle; connectedness, which allows distinguishing a circle from two non-intersecting circles.

Purpose of this chapter is to introduce the fundamentals concepts of topology, introducing the basic concepts of topological and metric spaces with their main properties. The scope of it within the thesis is to apply this concepts to Data Analysis. We will link this concepts to data perspective and we will often use the term *Data Cloud point* (PCD) that is a representing of a collection of high dimensional clouds of points (PCDs) through the space[Car15].

3.2 Topological Spaces

The notion of topological space represents a very general concept of space having a notion of "closeness" between elements defined in the weakest possible way.[Man14]

More specifically, a topological space is a set of points, along with a set of neighbourhoods for each point, satisfying a set of axioms relating points and neighbourhoods.

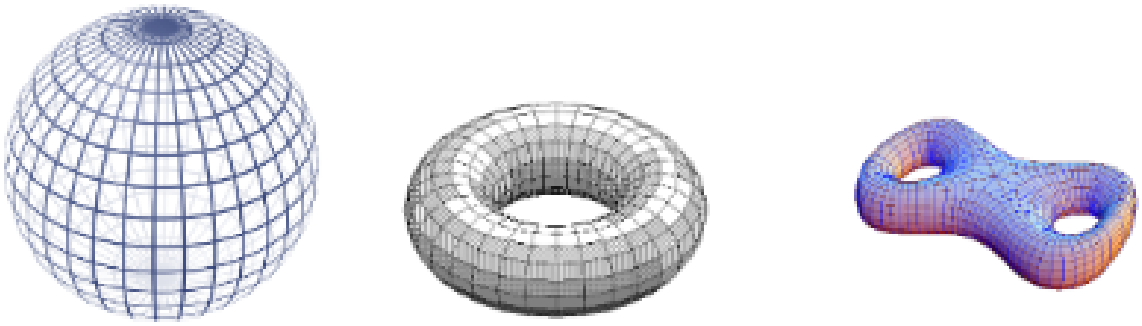


Figure 3.1: Topological Spaces. Taken from [htta]

Definition 3.2.1. Given a set $X \neq \emptyset$, a set τ is a topology on X if and only if

1. $\tau \subseteq P(X)$ (that is, τ is a set of subset of X),
2. $\emptyset, X \in \tau$,
3. τ is closed with respect to arbitrary union,
4. τ it is closed with respect to finite intersection. Elements of τ are called open sets in τ , while elements in X that are not in τ are called closed set in τ .

Intuitively, open sets are subsets of topological spaces which do not contain their boundaries [Tie17]. For example in $\mathbb{R}$, $(-\infty,0) \cup (1,-\infty)$ and [0,1] are respectively open and closed sets.

Definition 3.2.2. Given a set $X \neq \emptyset$, and a topology on it τ, (X, τ) is a topological space.

Example 3.2.3. Given the set $X = \{1,2,3\}$ the sets:

- the set: $R = \{\{\}, X, \{1\}\}$ is a topology on X

- the set: $S = \{\{\}, X, \{1\}, \{2\}\}$ is not a topology on X (the union of $\{a\}, \{b\}$ is missing)

Definition 3.2.4. Let (X, τ) be a topological space, and let $Y \subseteq X$. The subspace topology τ_Y on Y is $\tau_Y = (\cup \cap Y : \cup \in \tau)$. τ is the topology "induced by" or "inherited from" τ.

Definition 3.2.5. Given a $x \in (X, \tau)$, a set A is called neighbourhood of x if $A \subseteq X$ and A contains an open set containing x.

Intuitively, the concept of neighbourhood of x represents a set of points "close" or "similar" to x.

Definition 3.2.6. The trivial topology is the topology with the least possible number of open sets, namely the empty set and the entire space. All points here are closed because they all are sharing the same neighbourhood.

Example 3.2.7. Given the set $X = \{1,2\}$, the family $\tau = \{\{\}, \{1,2,3\}\}$ is the *trivial topology* on X

Observation 3.2.8. Even if we are used to see $\mathbb{R}$ intuitively as a straight line, $\mathbb{R}^2$ as a plane, etc., these representations are valid only in Euclidean topology. In fact the shape of a set is determined by its topology.

Example 3.2.9. $\mathbb{R}$ with the discrete topology can be seen as a cloud of separate and unordered points, while with the trivial one as a single "big" point because all real numbers are neighbors.

Definition 3.2.10. Let (X, τ) be a topological space, and let $\sim$ be an equivalence relation on X. The corresponding quotient topological space is given by the topological space $(X/\sim, \tau_\sim)$ where $\tau_\sim$is the quotient topology where the open sets are defined to be those sets of equivalence classes whose unions are open sets in X.

Definition 3.2.11. A topological space X is compact if for every collection C of open subsets of X such that $X = \cup_{x \in C} x$, there is a finite subset F of C such that $X = \cup_{x \in C} x$

Definition 3.2.12. A topological space X is connected if for any two points of X there exists a path between them on X.

Definition 3.2.13. The maximally connected subsets of a topological space are its connected components.

Example 3.2.14. Euclidean space is connected while any discrete space of size more than one is not connected.

3.3 Homeomorphisms

An homeomorphism is a function between topological spaces that models the intuitive idea of deformation without tearing, gluing or overlapping.

Definition 3.3.1. Given two topological spaces X and Y, a function $f : X \to Y$ is a homeomorphism among them if and only if f it is bijective, continuous and its inverse is continuous.

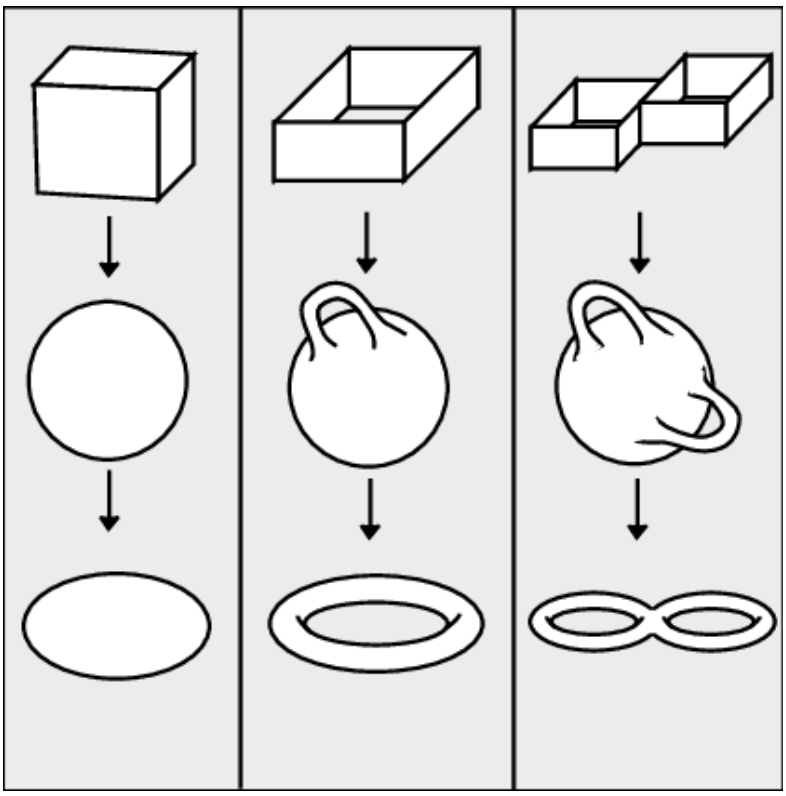


Figure 3.2: Classes of homeomorphic surfaces. The digits below the columns denote the number of holes, a topological invariant. Taken from [LL96]

Definition 3.3.2. Two topological spaces X and Y are homeomorphic, $X \simeq Y$, if and only if there exists a homeomorphism from one to another (or equivalently if they have the same topological properties).

Another and intuitive way to define two homeomorphic spaces is that two surfaces that are homeomorphic if they have the same boundary number.

General Topology studies the topological properties of topological spaces. These features are the ones preserved by homeomorphisms, so they are also called topological

invariant. In a few words the topological properties are preserved by homeomorphisms, so they are also called topological invariant.
Using the concepts of homeomorphism in topology, a cube (like a brick) is the same of a disc (see fig.3.2).

3.4 Homotopy

A more flexible notion of equivalence is the homotopy one. Intuitively, two continuous functions, defined from a topological space to another, are homotopic if one of them can be continuously deformed into the other.

The notion of homotopy equivalence is weaker than the notion of homeomorphism: all homeomorphic spaces are also homotopy equivalent but the converse is not necessarily true. However, spaces that are homotopy equivalent share the same homology.

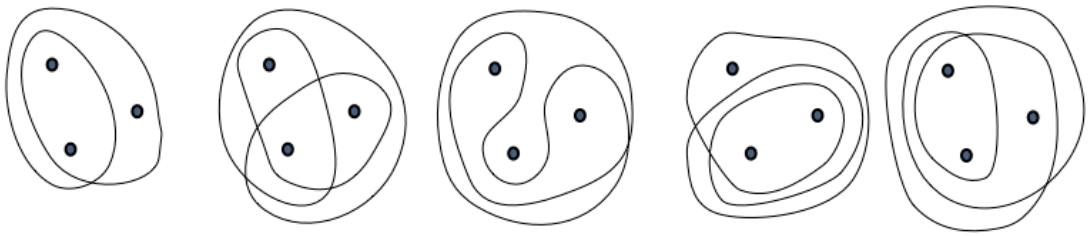


Figure 3.3: Examples of curves $\gamma \in R2\ C$ so that the homotopy class $[\gamma]$ is admissible. Taken from [Cas16]

Definition 3.4.1. Given two topological spaces X and Y and two continuous functions f and g from X to Y, an *homotopy* from f to g is defined to be a continuous function $H : X \times [0,1] \to Y$ such that, if $x \in X$ then $H(x,0) = f(x)$ and $H(x,1) = g(x)$.

Definition 3.4.2. Two topological spaces X and Y are *homotopy equivalent* if there exist continuous maps $f : X \to Y and g : Y \to X$ such that $g \circ f$ is homotopic to the identity map id_X and $f \circ g$ is homotopic to id_Y .

Definition 3.4.3. A space is said to be contractible if it's homotopy equivalent to a point.

Definition 3.4.4. In the n-sphere S^n n we choose a base point a. For a space X with base point b, we define $\pi_n(X)$ to be the set of homotopy classes of maps $f : S^n \to X$ such that a is mapped into b.

The concept of homotopy is important because allow to classify topological spaces up to homotopical equivalence, in fact spaces that are homotopy equivalent share many topological properties, in particular they have the same homology[CM].

3.5 Metric Spaces

A metric space is a set of elements, called points, in which a distance between them is defined. It represents a particular type of topological space (every metric space is a topological space).

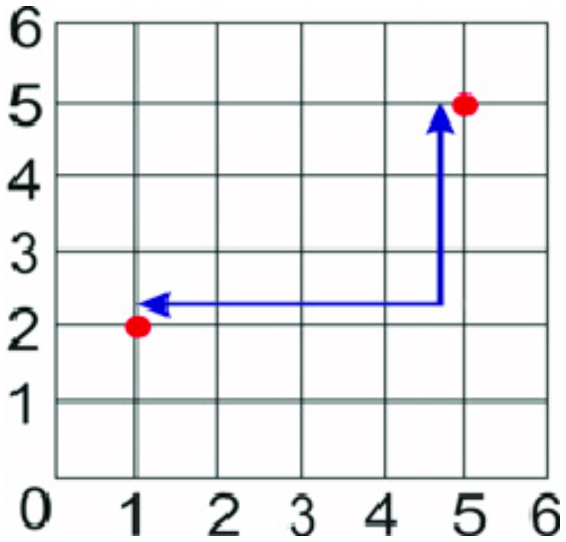


Figure 3.4: Manhattan distance of points x_i = [1,2]ax_j = [5,5] Taken from [TH12]

Example 3.5.1. One of the most popular distance function is the Euclidean distance,given two points $P = (p_x), Q = (q_x)$ defined as: $\sqrt{(p_x - q_x)^2}$

Example 3.5.2. Another used distance function is the "Manhattan distance". Given two points $P = (p_1, .., p_n)$ and $Q = (q_1, .., q_n$ the Manhattan distance is defined as:[Bla19]

$d_M(p, q) = \sum \mid p_i - q_i \mid$ for $i = 1, ..., n$

Definition 3.5.3. A distance (or metric), on a not empty set X, is any function $d : X \times Y \to \mathbb{R}$ such that $\forall x, y, z \in X$:

1. $d(x, y) \geq 0$
2. $d(x, y) = 0 \iff x = y$

3. $d(x, y) = d(y, x)$ (symmetry)

4. $d(x, y) \leq d(x, z) + d(z, y)$ (triangular inequality)

Definition 3.5.4. A metric space is a mathematical structure consisting of a pair (X, d) of elements, where X is a non-empty set and d is a metric on X.

Example 3.5.5. Classical examples of metric spaces are the real numbers with the distance function $d(x, y) = |y - x| d(x, y) = |y - x|$ given by the absolute difference, and, more generally, Euclidean n-space with the Euclidean distance, are complete metric spaces.

Definition 3.5.6. A finite metric space is a metric space having a finite number of points, that is a PCD.

Definition 3.5.7. Any metric space (X, d) is compact if and only if it is complete and totally bounded.

Observation 3.5.8. Any PCD is compact.

Example 3.5.9. The real numbers with the distance function $d(x, y) = |y - x|$ given by the absolute difference, and, more generally, Euclidean $n - space$ with the Euclidean distance, are complete metric spaces. The rational numbers with the same distance function also form a metric space, but not a complete one.

Example 3.5.10. The discrete metric, where $d(x, y) = 0$ if $x = y$ and $d(x, y) = 1$ otherwise, is a simple but important example, and can be applied to all sets. This, in particular, shows that for any set, there is always a metric space associated to it. Using this metric, the singleton of any point is an open ball, therefore every subset is open and the space has the discrete topology.

3.6 Simplices and Simplicial Complex

3.6.1 Simplices

A **Simplex** is a generalization of the notion of a triangle or tetrahedron to arbitrary dimensions. The simplex is so-named because it represents the simplest possible polytope in any given space. For example a *0-simplex* is a point, *1-simplex* is a line, *2-simplex* is a triangle and *3-simplex* is a tetrahedron.

Definition 3.6.1. Generally a $k - Simplex$, $\Delta(p_{j_1}, ..., p_{j_k})$ is a k-dimensional polytope which is the convex hull of its k + 1 vertices.

More formally, suppose the k + 1 points $p_0, \ldots, p_k \in \mathbb{R}^k$ are affinely independent, which means that $p_1 - p_0, \ldots, p_k - p_0$ are linearly independent. Then, the simplex determined by them is the set of points: $C = \left\{ \theta_0 v_0 + \cdots + \theta_k v_k \;\middle|\; \sum_{i=0}^{k} \theta_i = 1 \text{ and } \theta_i \geq 0 \text{ for } i = 0, \right.$

This representation in terms of weighted vertices is known as the barycentric coordinate system. A regular simplex is a simplex that is also a regular polytope. A regular $k - simplex$ may be constructed from a regular (k-1)-simplex by connecting a new vertex to all original vertices by the common edge length.

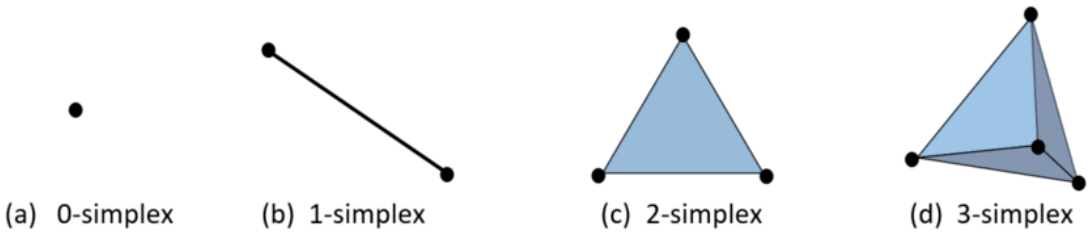


Figure 3.5: k-simplices.Taken from [Iqb+21]

Definition 3.6.2. Given a set of simplices K their union is a subset of $\mathbb{R}^d$ called underlying space of K that inherits from topology of $\mathbb{R}^d$

Definition 3.6.3. Let be a ς a a k-simplex defined by $P = p_0, p_1, ..., p_n$. A simplex ϑ defined by O with $O \subset P$ is a face of ς , ς ¿ ϑ.

Definition 3.6.4. The vertices of P are the zero-simplices of P.

Definition 3.6.5. A key concept in defining simplicial homology (this concept will be presented in the next section) is the notion of **orientation** of a simplex. By definition, an orientation of a k-simplex is given by an ordering of the vertices, written as $(v_0, ..., v_k)$, with the rule that two orderings define the same orientation if and only if they differ by an even permutation.

3.6.2 Simplicial Complexes

Definition 3.6.6. A simplicial complex $\mathcal{K} = (V, S)$ consist in a set of vertices $V = (p_1, ..., p_n)$ and a bag $S = (\sigma_1, ..., \sigma_n)$ of finite nonempty subset of simplex (faces) that satisfies the following conditions:

1. Every face of a simplex from $\mathcal{K}$ is also in $\mathcal{K}$.
2. The non-empty intersection of any two simplices $\sigma_1, \sigma_2 \in \mathcal{K}\sigma_1, \sigma_2 \in \mathcal{K}$ is a face of both $\sigma_1\sigma_1$ and $\sigma_2\sigma_2$.

Definition 3.6.7. The dimension of P is Dim(P) = max(dim(ς—ς)∈P)

Definition 3.6.8. An *abstract simplicial* complex is a set P with its collection S of subsets such that $\forall\rho \in P, \{\rho\} \in S$ and $\forall\sigma \in S$ if $\vartheta \subset \sigma$ then $\vartheta \in S$.

Definition 3.6.9. Given a finite point cloud X and an $\varepsilon > 0$, we construct the **Čech Complexes** $\check{C}_\varepsilon(X)$ as follows: Take the elements of X as the vertex set of $\check{C}_\varepsilon(X)$. Then, for each $\sigma \subset X\sigma \subset X$, let $\sigma \in \check{C}_\varepsilon(X)\sigma \in \check{C}_\varepsilon(X)$ if the set of ε-balls centered at points of σ has a nonempty intersection.

Theorem 3.6.10. ***"The Nerve Theorem"****:*
Given a topological space X and an open cover $\mathcal{H} = \{H_i\}_{i\in I}$ of it such the intersection of any subset of the $\{H_i\}$ is either empty or contractible, X and the Nerve $Nev(H)$ are homotopy equivalent.

Definition 3.6.11. Vietoris–Rips complex also called the Vietoris complex or Rips complex, is a way of forming a topological space from distances in a set of points. It is an abstract simplicial complex that can be defined from any metric space M and distance φ by forming a simplex for every finite set of points that has diameter at most φ.

3.7 Homology

Definition 3.7.1. A **Filtrations** $\mathcal{F}$ is an indexed family $(S_i)_{i\in I}$ of subobjects of a given algebraic structure S, with the index i running over some totally ordered index set I, subject to the condition that: if $i \leq j$ in I , then $S_i \subseteq S_j$.

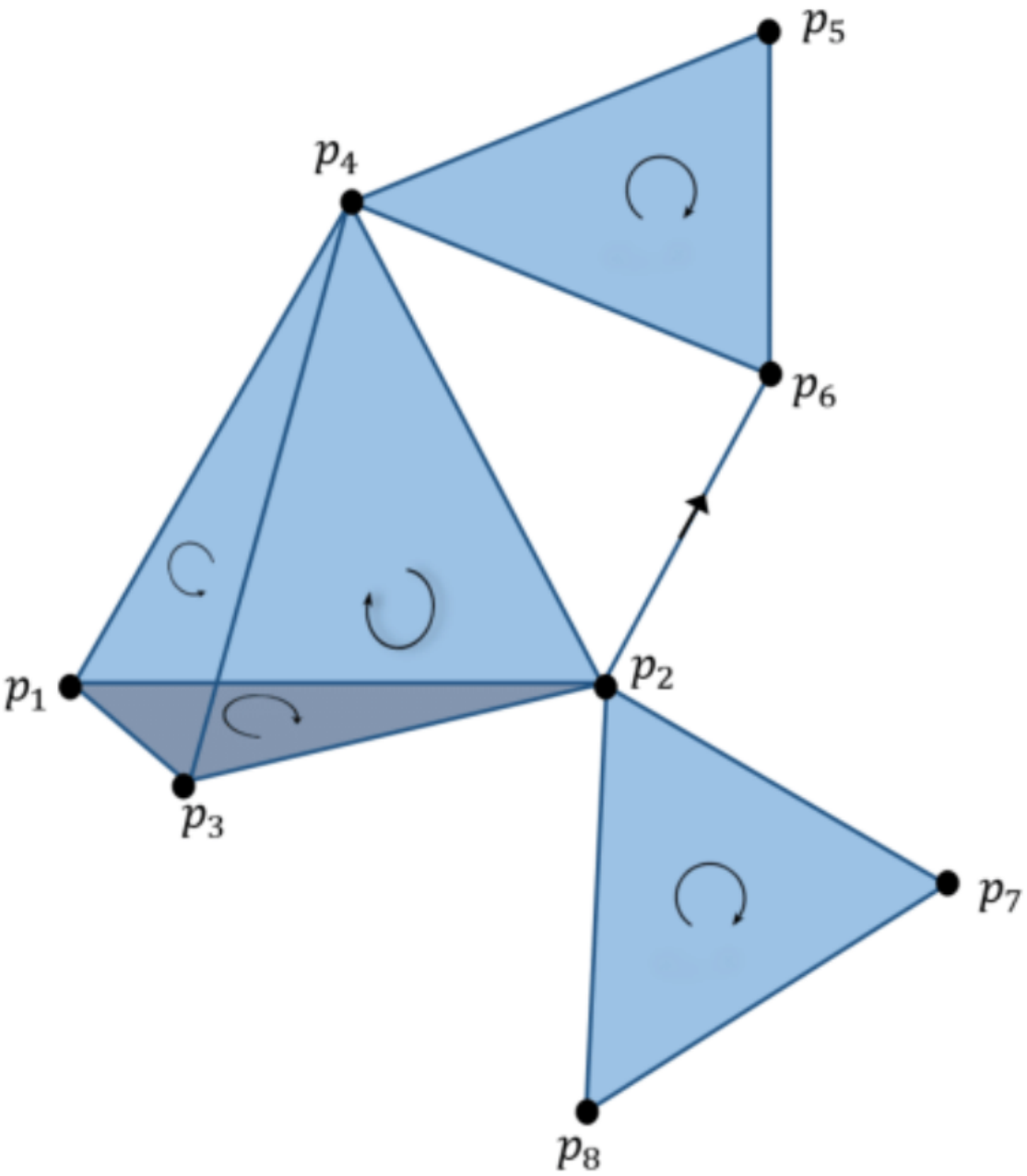


Figure 3.6: Simplicial Complexes. Taken from [Iqb+21]

Definition 3.7.2. A **manifold** is a topological space that locally resembles Euclidean space near each point. More precisely, an n-dimensional manifold, or n-manifold for short, is a topological space with the property that each point has a neighborhood that is homeomorphic to an open subset of n-dimensional Euclidean

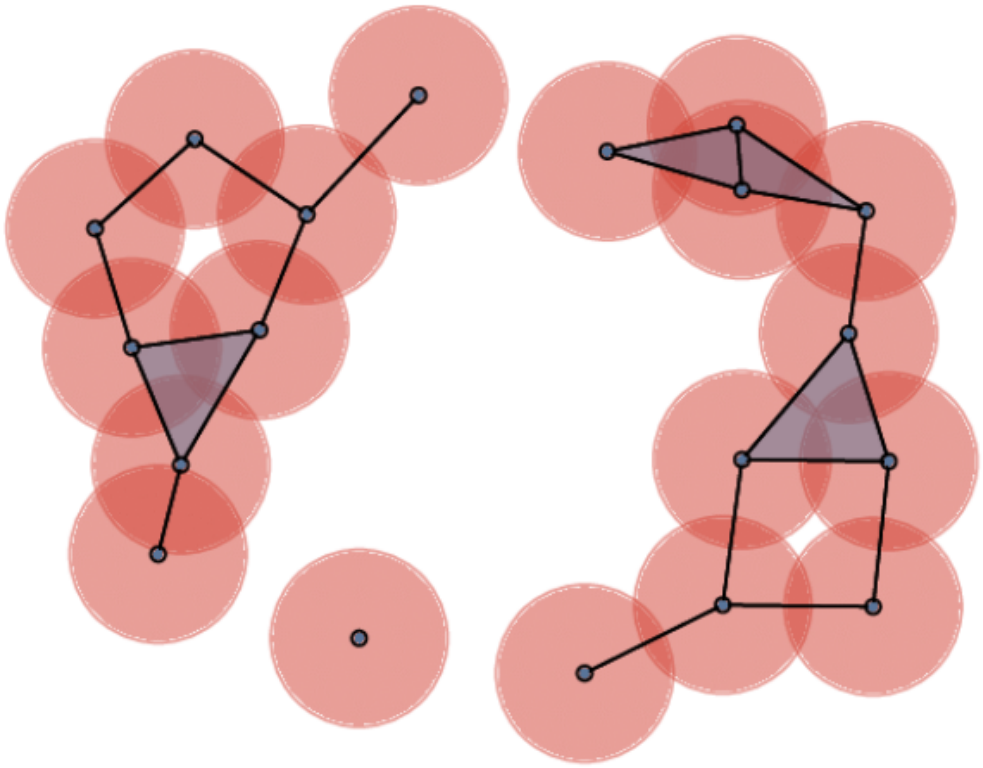


Figure 3.7: Čech Complexes. Taken from [RWS16]

space.

Homology was originally a rigorous mathematical method for defining and categorizing holes in a *manifolds.*The original motivation for defining homology groups was the observation that two shapes can be distinguished by examining their holes. For instance, a circle is not a disk because the circle has a hole through it while the disk is solid, and the ordinary sphere is not a circle because the sphere encloses a two-dimensional hole while the circle encloses a one-dimensional hole. However, because a hole is "not there", it is not immediately obvious how to define a hole or how to distinguish different kinds of holes.

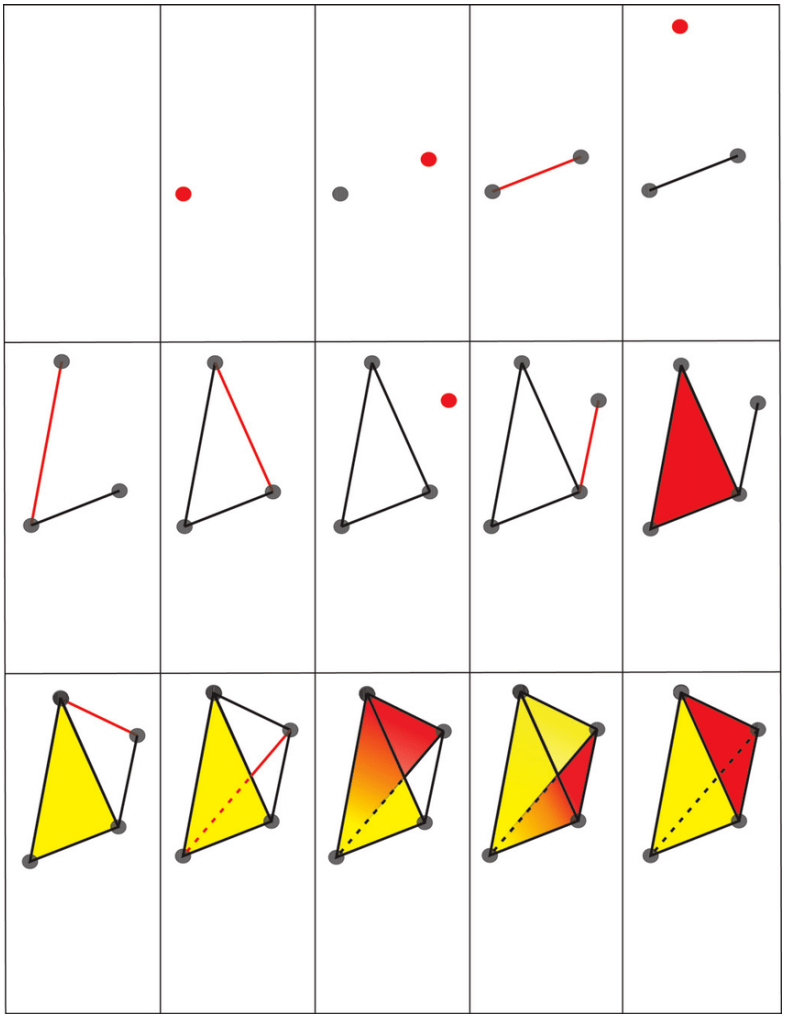


Figure 3.8: Filtration of a simplicial complex (tetrahedron) and its topological characterization.
At each stage a vertex or a face is added. represented in red. Taken from [HMR08]

3.7.1 Simplicial Homology

Definition 3.7.3. Let S be a simplicial complex. A simplicial **k-chain** is defined as: $\sum_{i=1}^{N} c_i \sigma_i$, where each c_i is an integer and σ_i is an oriented k-simplex.

Definition 3.7.4. The **boundary operator**: $\partial_k : C_k \to C_{k-1}$ is the homomorphism defined by:

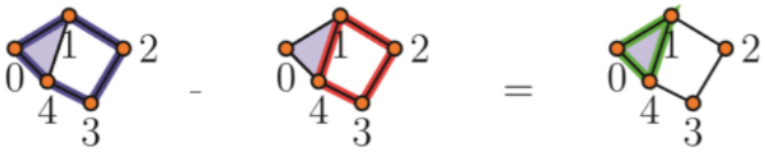


Figure 3.9: Homologus Simplicial complexes. Taken from [TZH14]

$$\partial_k(\sigma) = \sum_{i=0}^{k} (-1)^i (p_0, \ldots, \widehat{p_i}, \ldots, p_k),$$

where the oriented simplex:

$$(v_0, \ldots, \widehat{v_i}, \ldots, v_k)$$

is the i-th face of σ, obtained by deleting its i-th vertex.

Definition 3.7.5. The elements of:

$$Z_k = ker(\sigma_k)$$

defines the **cycles** of C_k.

The elements of:

$$B_k = Im(\sigma_(k+1))$$

defines the **boundaries** of C_k.

Definition 3.7.6. The k-th **homology group** H_k of a simplicial complex S is the quotient abelian group $H_k(S) = Z_k/B_k$

3.7.2 Persistent Homology

Persistent homology is a method for computing topological features of a space at different spatial resolutions.
More persistent features are detected over a wide range of spatial scales and are deemed more likely to represent true features of the underlying space rather than artifacts of sampling, noise, or particular choice of parameters [AAE19].
To find the persistent homology of a space, the space must first be represented as a simplicial complex. A distance function on the underlying space corresponds to a filtration of the simplicial complex, that is a nested sequence of increasing subsets. This approach is recently exploited in Data Analysis in order to find the underlying space of data robustly to perturbations. In other words we try to analyze and draw conclusion from dataset by studying its "shape".

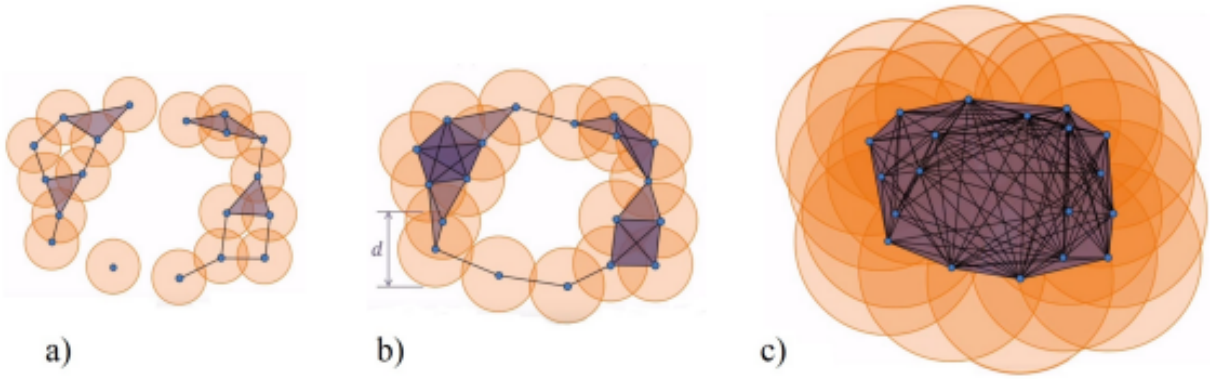


Figure 3.10: Simplicial complexes build with a different distance values. Taken from [Wri16]

3.8 Application

Topology is widely used in Data Analysis, in the latest years topological properties of data sets are exploited in order to develop new techniques: this branch of Data Science is known Topological Data Analysis (TDA).
The key idea is to map data point (PCD) in a featured dimensional space in order to study its topological structure and extract knowledge from them.
[Mas20]. For example, topological data properties are useful to solve Frequent

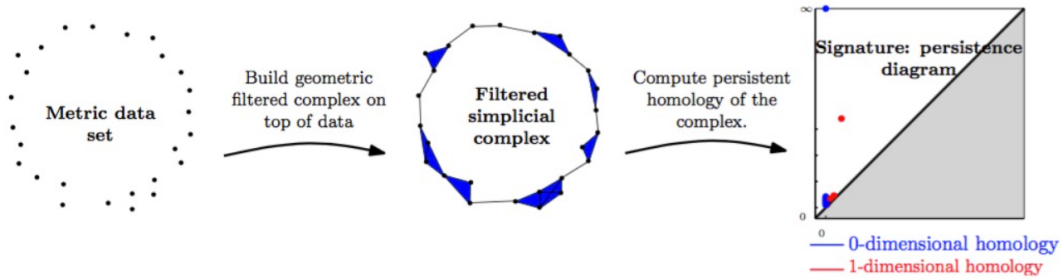


Figure 3.11: TDA - Persistence diagram workflow [Cha+13]

Itemset mining problems (that we will deepen in next chapters) or find association rules. Other important and widely used application of TDA are based on computation of Persistent Homology of a complex and plot a persistence diagram finding application in Time Series [Mas21] or for Biological datasets.
Basically a Persistence Graph can be used to gain information otherwise concealed

by the low-dimensional nature of graphs exploiting information carried by the k-PBN function; in this way is possible to represents the information associated

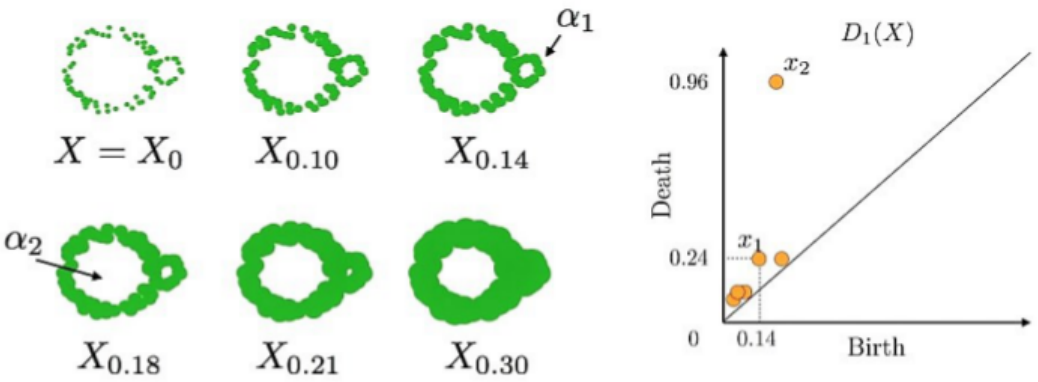


Figure 3.12: TDA - Persistence diagram of a Filtration [KHF16]

with the evolution of the k-holes, i.e. the connected components [BFZ20].
Persistent homology can help gaining new perspectives on the analysis of complex networks, by combining graph-theoretical and associated topological constructions, with persistent homology and persistence diagrams (for further information see the works [HMR09], [MRV08]).

Chapter 4

Simplicial Complexes and Frequent Itemsets

4.1 Introduction

Topological data analysis (TDA) is a recent field that emerged from various works in applied (algebraic) topology and computational geometry during the first decade of the century. Although one can trace back geometric approaches to data analysis quite far into the past, TDA really started as a field with the pioneering works of Edelsbrunner et al. (2002 [Bre+04]) and Zomorodian and Carlsson (2005 [Car+05]) in persistent homology and was popularized in a landmark article in 2009 Carlsson in 2009.
TDA is mainly motivated by the idea that topology and geometry provide a powerful approach to infer robust qualitative, and sometimes quantitative, information about the structure of data.
TDA aims at providing well-founded mathematical, statistical, and algorithmic methods to infer, analyze, and exploit the complex topological and geometric structures underlying data that are often represented as point clouds in Euclidean or more general metric spaces.

Purpose of this chapter is to apply topological concepts presented in previous chapter in order to solve Frequent Itemset Mining problem.

4.2 Data structures

In this section we will discuss and present some techniques that aims to map data into topological structures (simplicial complexes) presented in previous chapter. The final scope is to exploit the topological properties of simplicial complexes in order to find efficient algorithms for Frequent Itemsets Mining problems (presented in first chapter).
How we can see in the next sections (when we present FreSCo algorithm) data structure mapping are a crucial steps: in our case we will see how generalize a graph structure to a simplicial complex structure and and we will show that this more powerful representation gives rise to different properties compared to the graph-based one [PDB21a].

How we have seen in previous chapter data can be represented in high dimensional space as "cloud of points" (PCD). So the first basic idea is to map in a n-dimensional space data point starting from a dataset.
Each row of the input table represent a data point and each column is an axes: this is sufficient to map data into our space.
Once data points are mapped in the space, is possible to build on it a simplicial complex, for example the Čech complex is a very good discretization of the space, but it is rarely used in practice because it is computationally heavy to construct;instead Vietoris-Rips complex is popular in topological analysis thanks to the ease of its construction in every dimension.[Pat19]

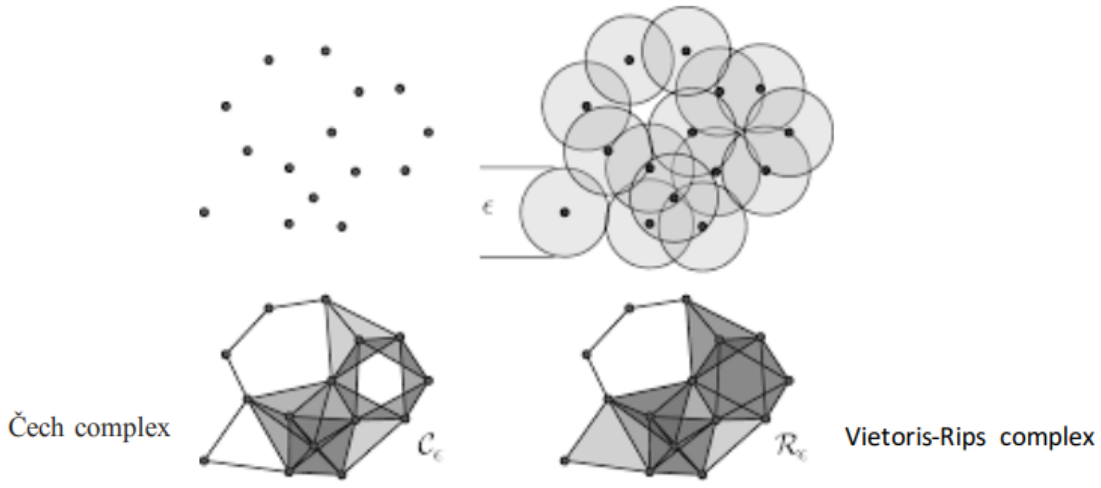


Figure 4.1: Data can be represented by PCD in an high dimensional space; from this representation is possible to build a simplicial complex. Taken from [Zan17].

4.2.1 Data structure for a Market Basket Analysis case

Now we will present another way to map the classical tabular data structure into a simplicial complex. This based on the works of Garcia, Molina (2008) [GUW08] and Greenberg (2018) [GH18] and modified by Subasi (2020) [Sub20] is specific for typical market basket analysis problems; then using this data mapping we will present an interesting algorithm to find frequent itemsets.

Using a common example of a Market Table e.g. a table (take for example the below table 4.1) is a classical table (used in previous chapter as FIM algorithms input) for basket market analysis: and each columns represents a purchased product (A,B,...) and each (row also called tuple) identified by a transaction ID is the basket of products purchased by customer ; 1 values represents the presence of the product in the transaction.

	A	**B**	**C**	**D**	**E**	**F**	**G**
T1	1	1	1	1	0	0	0
T2	0	1	1	1	1	0	0
T3	1	1	0	0	0	1	1
T4	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-

Table 4.1: Transactions Table

Here we try to provide a geometrical interpretation of this table in order to turn it in a simplicial complex. Each product (in our example A,B,...) is a data point in the space and it is linked to the other points in the same transaction. Hence is possible to see every tuple as a k-simplex in which the number k is determined by the number of elements that compose the tuple (for our market example, the number of purchased products), therefore any tuples is turned into a k-simplex. Hence the combination of all tuples build a simplicial complex (see 4.5).

4.3 The Truss Decomposition

4.3.1 Problem definition

In this section we show an interesting way that allow to map a dataset into a topological data structure (simplicial complex): the Truss Decomposition (presented by [PDB21b]).
The starting intuition is that we can see a simplicial complex as a generalization of a graph in which each simplex is a n-ary relation (then we can see a simplicial complex as a collection of n-ary relationship) instead of an edge that is a binary relation in a graph. We use the concept of "trusses": a cohesive subgraphs that are rich in triangles, a relaxation of cliques.
The definition of truss decomposition in a graph is based on triangles, i.e., a k-truss is a subgraph whose edges participate in at least k triangles; when the input is a simplicial complex, a truss can as well be determined by the existence of higher-order structures. Below a set of definitions:

Definition 4.3.1. The joist of a simplex K is the set J_K of all its co-faces.

Definition 4.3.2. Given a simplicial complex $\mathcal{K}$ a *k-truss* in $\mathcal{K}$ is a maximal set of simplices T_k of dimension greather than 0 such that for each simplex in $\mathcal{K}$ there exist at least a k joist $J_1, ..., J_k$ such that $\forall i \in [1, k], K \in J_i$ and $\forall \tau \in J_i$, $\tau \in T_k$. The maximal k such that $K \in T_k$ is called *simplicial trussness* of K and we denote it as $tr_k(K)$.

Property 4.3.3. The k-truss of simplicial complex $\mathcal{K}$ is unique.

Property 4.3.4. The (k+1)-truss of simplicial complex $\mathcal{K}$ is a subset of its k-truss.

A key observation towards developing this method is an a-priori property similar to the one from frequent itemset mining: a (q + 1)-simplex has simplicial trussness no larger than a q-simplex that it contains. Based on this key observation, we derive lower and upper bounds for the simplicial trussness of a simplex which are at the basis of our method.
Formalizing this idea:

Property 4.3.5. *(Apriori property)* Let K' a k-simplex that is a face of a $(k + 1)$*simplex* K, then $tr_K \geq tr_{K'}$.

Property 4.3.6. *(*Lower Bound) Let K a k-simplex and H a (k+h)-simplex that is the largest simplex in $\mathcal{K}$ such that $K \subset H$. Then $tr_{\mathcal{K}}(K) \geq (h - q)$

Property 4.3.7. *(*Upper Bound) Let K a k-simplex in $\mathcal{K}$ and H a (k+1) simplex and J_H indicate a joist of H. Then:

$tr_{\mathcal{K}}(K) \leq |J_H \mid J_H \subseteq \mathcal{K} \wedge K \in J_H|$.

4.3.2 The STruD Algorithm

Now we present the STruD algorithm: it take in input a simplicial complex $\mathcal{K}$ and return the trussness tr of simplices in $\mathcal{K}$.
Basically the algorithm use the following steps, (apriori-like approach) that materializes and examines simplices of increasing dimension):

1. extend the simplices retained in the previous iteration by adding an additional vertex;
2. search for all the sets of simplices that form a joist;
3. compute the trussness of each simplex; the simplices with positive trussness will be extended in the following iteration.
4. at the end of the procedure each simplex such that $tr(k) = lb(k)$ (where $lb(k)$ is the lower bound of k) are removed.

Now we explain step by step how the algorithm works. First of all the algorithm initialize $tr = \emptyset$ then:

Step 1: *(Extend simplices)* In the first step the set C of the connected components of $\mathcal{K}$ is computed; basing on C, algorithm compute the set E of the "extended simplices" and for each simplices $k \in E$ lower bound is computed.

The procedure that extend simplices receives in input a set of simplices to extend S (initialized as $\emptyset$) and the set of simplices in the connected component of $\mathcal{K}$ under examination C; the procedure extends each simplex $k \in S$ by appending each vertex $v \in V_k$, (the set of all the vertices that appear in some $k' \in \mathcal{K}$ such that $k \subset k'$), generating the extensions by using only those vertices guarantees that we create only simplices that exist in $\mathcal{K}$. At the end of the computation the procedures return the set E of the extended simplices.

If not exist a simplex k with lower bound $lb(k) \neq tr(k)$ the set S is updated with the simplex k such that $tr(k) > 0$, thus $S = \{k | k \in E \wedge tr(k) > 0\}$.

Step 2: *(Finding joists)* We call CJ the set of joists of E (initialized to $\emptyset$).
First of all the algorithm associate to each simplex k a set of fingerprints, or codes, defined as subsets of vertices of k; then dynamically creates an inverted index that maps a code ξ to a set of simplices that contain ξ. The algorithm find all the simplices τ that share an indexed code with the current simplex k and at the end the algorithm indexes all the codes of k .
Once found all the simplices that share a coface ξ with k, in this step the procedure updates the data structure CJ, which tracks candidate joists.
Now, we need to extract the real joist. In order to do that the procedure extracts

subsets of simplices in the candidate set CJ[k] of a simplex $k = [v_0, ..., v_k]$ that could form, together with k, the joist of a $(k+1) - simplex[v_0, ..., v_k, w]$.
Each subset of (k + 1) simplices in a joist J of a (k + 1)-simplex share a vertex w, because each pair of simplices in J have k vertices in common. Therefore, for each vertex w in some simplex in CJ[k] but not in k, the algorithm checks if the total number of simplices in CJ[k] that contain w is equal to k +1. When this condition holds, it adds w to the set of real joists $\mathcal{I}[k]$, and τ k to the set of real joists $\mathcal{I}[\tau]$ of all the simplices τ containing v. Furthermore the algorithm use an additional methods to ensure that each candidate joist is examined only once.

Step 3: *(Compute trussness)* For each simplex $k \in E$ is computed its trussness (basing on the computation of CJ[k] at the previous step.

At the end of this steps, the set S is updated with the simplex $\{k|k \in E \wedge tr(k) > 0\}$ if not exist a simplex $k|lb(k) \neq tr(k)$ and continues, otherwise the set tr is computed.

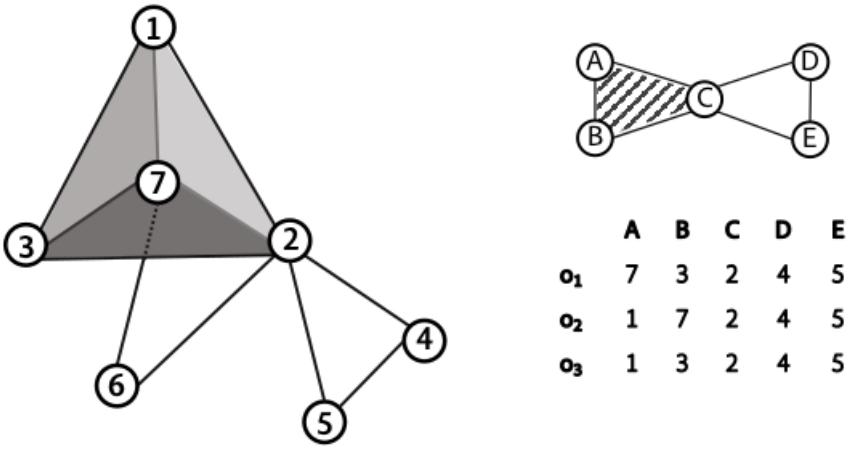
Step 4: *(Remove trivial trussness values)* At the end of the computation, the algorithm remove all the simplices k such that $tr[k] = lb[k]$ from tr, so that tr contains only the non-trivial simplicial trussness values.

4.4 The FreSCo Algorithm

In this section we will present an opposite way to apporoach to our problem: starting from a topological data structure we will find Frequent Itemsets.
This approach is particularly interesting because it allows to analyze problems that require high-order relations: for example three entities co-occurring in pairs in three different Wikipedia pages, from the case where all three entities co-occur in the same page [BGH21]; in this case a graph data structure (e.g. used in previous shown "Traversal Algorithm" or in FP-Growth is not enough expressive to represent and solve FIM problem.
This approach finds a lot of applications in data analysis: such as fraud detection , network intrusion detection, trend discovery, Web usage mining, link prediction for people recommender systems, predicting group activity on social networks.

We generalize the task of frequent pattern mining from graphs (the case in which entities are involved in pairwise interactions) to the case of higher-order relations. In fact, many interactions in the real-world occur among more than two entities at once. The key idea is to extract all frequent "simplicial patterns" from a given siplicial complex (as defined in chapter 3); notice how the concept of "itemset" is

turned into "simplicial patterns": that allow the high order generalization.



	A	B	C	D	E
o_1	7	3	2	4	5
o_2	1	7	2	4	5
o_3	1	3	2	4	5

Figure 4.2: Simplicial complex and simplets with its occurrences table, image taken from [PDB22]

Now the "simplicial pattern" is our reference entity that we call "simplet" (notice that the therm tha analogy with a "graphlet"), intuitively it is a "subcomplex" (a precise definition will be provided in following section).

We proceed as previous section: first data mapping structure will be shown, then we describe an efficient algorithm mine frequent simplets and for last we present an implementation.

4.4.1 Definitions and problem formalization

In this section we introduce the basic definitions (and notation) useful for the the formalization of previously presented problems and next its theoretical solution and implementation. We start from the definition of "simplet" and then proceed with its properties, we notice that how we already mentioned before, it represent an high order generalization of a graph data structure (e.g. used in FP-Growth algorithm presented in Chapter 2.

Definition 4.4.1. Given a set of simplies C_P and vertices V_P, a simplet P is a tuple (V_P, C_P) where C_P is a set of simplices such that:

- Any simplices $\in C_p$ only if all of its faces are in C_P

- For each pairs of vertices v,u in V_P there exists a sequence of 1-simplices that connect u,v.

Definition 4.4.2. The dimension $dim(P)$ of a simplet is the dimension of the largest simplex in C_P

Definition 4.4.3. We call the size of P the number of vertex that compose P

Definition 4.4.4. Given two distinct simplets P_1,P_2; $P_1 \leq_C P_2$ if and only if and only if for each simplex K $\in C_P 1$ exist a simplex $K^{'} \in P_2$ such that $K \subseteq K^{'}$.

Definition 4.4.5. We say that a complex K_1 is isomorphic to another complex K_2, denoted as $K_1 \backsimeq K_2$ if and only if there exist a bijection $\phi : V_{K_1} \rightarrow V_{K_2}$ that preseves the relations between the simplices, i.e., for each $\sigma = [v_0, .., v_q] \in K_1$ it holds that $\sigma^{'} = [\phi(v_0), ..., \phi(v_q)] \in K_2$
A bijection from a complex K to itself is denoted $Auto(K)$.

Notice that when the dimension of K is 1 these definition coincides whit graph isomorphism and automorphism.

Definition 4.4.6. The subcomplexes of K isomorphic to P are called occurrences of P in K (a.k.a. instances or embedding)

Definition 4.4.7. Let the image set $Im_P(v)$ of $V \in V_P$ of a simplet P be the set of vertices in K that are mapped to v by some isomorphism ϕ , i.e. $Im_P(v) = \{u \in K | \exists \phi s.t. \phi(u) = v\}$. Then $SUP(P, K) = min_{v \in p} |I_P(v)|$

Notice that SUP is anti-monotone.

Definition 4.4.8. Given two distinct simplets P_1 and P_2 , we say that $P_1 \leq_C P_2$ (the simplet P_1 is included in simplet P_2) if and only if for each simplex $\sigma \in P_1$ there exist a simplex $\sigma^{'} \in P_2$ such that $\sigma \subseteq \sigma^{'}$.

With this set of definitions, we can strictly formulate the problem presented in this section:
Given a simplicial complex K, a maximum size $s*$, a minimum dimension $d*$, and minimum frequency threshold τ, find all the simplets P such that:

- $|V_P| \leq s^*$
- $dim(P) \geq d^*$
- $SUP(P, K) \geq \tau$

Frequent pattern mining in simplicial complexes presents numerous challenges. First, the search space is extremely large, due to all the possible combinations of higher-order simplices that can constitute the simplet. Second, finding the occurrences of

simplets in a complex entails determining if two simplets are isomorphic. While there exist many libraries for solving the graph isomorphism problem, little effort has been devoted to complex isomorphism. We show that finding the occurrences of simplets in a complex can be reduced to a bipartite graph isomorphism problem, in linear time and at most quadratic space. We then propose an anti-monotonic frequency measure that allows us to start the exploration from small simplets and stop expanding a simplet as soon as its frequency falls below a minimum frequency threshold. Based on these ideas and a clever data structure, we devise a memory-conscious algorithm called FreSCo FreSCo (Frequent Simplets in a Complex, Algorithm), that carefully exploits the relations among the simplices in the complex and among the simplets, to achieve efficiency.
FreSCo comes in two flavors: it can compute the exact frequency of the simplets or, more quickly, it can determine whether a simplet is frequent, without computing the exact frequency

4.4.2 Data structure

Frequent pattern mining in graph-structured data aims at finding structures that occur frequently in a given simplicial complexes under the assumption that frequency indicates relevance.
Frequent itemset mining entails discovering all the subsets of items appearing frequently in a transactional database.
Given that a simplex can be represented as a set of vertices with their edges, a complex can be seen as a family of sets, i.e., a transactional database.
Hence, the task of finding frequent simplets in a complex might resemble that of frequent itemset mining. However, the latter counts the number of transactions in the database that contain all the elements in the itemset, while in our setting, simplets are connected sub-complexes, and as such, they consist of many interconnected simplices that can span across multiple simplices in the complex.
Now we introduce two important theoretical concepts that allow us to understand the basic "data structure" that will be exploited in next section (algorithm presentation).

Each node in the lattice corresponds to a simplet, with the root being the simplet that consists of a 0-simplex. A directed edge connects a node to a node A to a node B if and only if $A \leq_C B$. and so we say that A precedes B in the lattice. The lattice can be generated by recursively applying the following expansion rules to a simplet P initially set equal to the root:

- **widen**: inserting a new 1-simplex $[U, V]$ in C_P such that $VinV_P$ and $U \notin V_P$,
- **inflate**: inserting a new q-simplex σ^q in C_P such that its joist is already

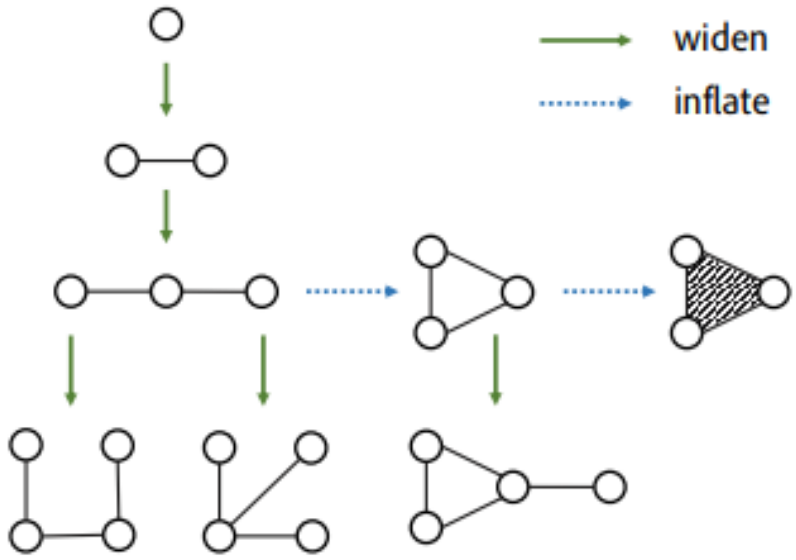


Figure 4.3: Widen and Inflate rules, image taken from [PDB22]

contained in C_P.

Notice that applying widen expansion rule a simplet $P' := (V_P \cup \{u\}, C_P \cup \{[u, v]\})$ is generated: this increase the size of vertex set in V_P. Applying inflate expansion a simplet $P' := (V_P, C_P \cup \{\sigma^{(q)}\})$ is generated: this increase the dimension of P. Now we provide a fundamental theorem that we exploit in following section (FreSCO algorithm) in order to scan data structure and find the frequent simplets:

Theorem 4.4.9. (Completeness) Starting from a 0-simplex the recursive application of widen and inflate can generate any simplets.

4.4.3 The Algorithm

FreSCo algorithm [PDB22] extracts the frequent simplets in a simplicial complex, given a maximum size s* and a minimum dimension d* recursively expands each frequent simplet until it becomes infrequent.
For each candidate frequent simplet, FreSCo searches for subcomplexes in the complex that are isomorphic to the simplet, and stores the corresponding mapping from complex vertices to simplet vertices.

Simplet expansion
The computation starts from a simplet O consisting of a single vertex v, whose

support set $I_O(v)$ is the vertex set of the complex. The set of frequent simplets F is initialized as the empty set. The key idea is to expand recursively simplets P proven to be frequent. If the number of vertices in P is below the maximum size s*, the simplet is first expanded by adding a new vertex D and connecting it to a vertex in V_P (**widen**).
As second step, the algorithm applies the **inflate**: for each joist of a simplex K not already in the simplet P a new simplet $P^{'}$ is instantiated by adding a new simplex K to C_P.
Notice that starting from two different sequence of widen and inflate procedures is possible to generate the same simplet $P^{'}$: then is necessary to check if a simplet is already generated.
Once the algorithm has created all the expansions $P^{'}$ of P, the image sets are computed.

Frequency computation
In the following, we say that two vertices are neighbors if they belong to a common simplex, and denote with $\Gamma(u)$ the set of neighbors of a vertex u.
The algorithm takes in input a parameter t* for the maximum amount of time that the algorithm can spend trying to find a complete valid assignment for a candidate D, and the parent simplet of P, denoted with SP.The parent is the simplet that we expanded to generate P. The threshold t* is needed to avoid spending too much time on problematic candidates.
We initialize the set of valid assignments $I_P(v)$ with the set of all the vertices of simplices having the same or more vertices than P, as all the simplets with 3 vertices are sub-simplets of simplices with at least 3 vertices. The sets of non candidate complex vertices SP.NC (see implementation) are used to initialize the non-candidate sets of those vertices cannot be valid assignments for P either.
The algorithm iterates over the vertices of P examining only those for which it has not found enough assignments yet. When examining a vertex v, the set of candidates is initialized as the set V_K minus $NC(v) \cup I_P(v)$ as these vertices cannot increase the size of $I_P(v)$, and hence do not help in determining if P is frequent.

4.4.4 Implementation

Now we can provide a pseudo-code implementation of FreSCo Algorithm (taken from [PDB22].
How we say FreSCo algorithm aims to extracts frequent simplets in a simplicial complex (simplets are now the representation of the itemset in our high order search space).
The algorithm presented below is composed essentially by two procedure: "expand" and "examine".
Expand function (that call "examine" function) take as input parameter the complex

K, a "start simplet" $P := (\{u\}, \{v\})$ and three other parameter: t,s,d, in which t is a frequency threshold (used in "examine" procedure), s and d are respectively size(P) and dim(P) thresholds. Expand procedure return as output the set F of frequent simplets.

Now we can explain more precisely how the algorithm work.
In the *expand* procedure (above the code), the computation start from a simplet O consisting of a single vertex v ($I_O(v)$ is vertex set of the complex); notice that the set F of frequent simplets is initialized with an empty set (line 7).
Core of the procedure is the iterative apply of widen (line 8-14) and inflate steps (line 16-20) in order to generate the set E of all simplets; notice that both in widen and inflate step new candidate simplets are added to E after a duplicate check (line 13 and 19).
At the end of the function, "examine" procedure is called in order to choose (examine) the frequent simplets set F from the generate simplets set E, using the parameter constraints s,d,t (reported and commented below).

```
1  ########################## EXPAND PROCEDURE ######################
2  O = ({v},{[v]});
3  I_o(v) = V_K;
4
5  function expand(K,P,t,s,d):
6      E = {};
7      F = {};
8      if |V_P| < s:
9          for v in V_P:
10             u = vertex();
11             P' = (V_P U {u}, C_P U {[u,v]}  ###### widen step
12             I_P'(w) = V_K for w in V_P'
13             if P' not yet generated:
14                 E.add((P',I_P'))
15
16     for J_s(q) such that s(q) not in C_P:
17         P' = (V_P,C_P U {s(q)})
18         I_P'(w) = V_K for w in V_P
19         if P' not yet generated:
20             E = E.add((P',I_P)) ########### inflate step
21
22     for P U I in E:
23         I_P = examine(K;P;U I,t);
24         sup = min(I_P (v))
25         if sup >= t:
26             if(dim(P) >= d:
27                 F = F U {P}
28                 F = F U expand(K,P,t,s,d)
```

```
29
30      return F
```

How we say before examine procedure aims to examine if the simplets candidate generated before are "frequent".
In the following, we say that two vertices are neighbors if they belong to a common simplex, and denote with G(v) the set of neighbors of a vertex v. The procedure searches for assignments from vertices in the complex to vertices in P, such that the membership relations to the simplices in C_P , are preserved.
Procedure findMatch (called in line 16 and implemented at the bottom) recursively visits all the not-yet-explored vertices, according to an order obtained by a d.f.s. starting from v, so that the recursive call visits neighbor vertices most of the times. When visiting vertex x, the algorithm first computes a restricted candidate set Cands, and then tries to find a valid assignment for x among the complex vertices in Cands. To create the restricted candidate set, we first compute the intersection among the neighbors of the complex vertices associated to simplet vertices that are neighbors of x. Thus, we can avoid examining complex vertices that would not preserve the simplex membership relation. A vertex assignment is valid if Procedure satisfiesConstraints (line 30) returns true.
If a candidate n is a valid vertex assignment, the assignment is added to M (set of frequent simplets), and Procedure findMatch is recursively called. For each simplex σ in the simplet containing the vertex x, Procedure satisfiesConstraints checks whether the complex vertices already assigned to the vertices in σ form, together with the candidate assignment n of x, a simplex in the complex. If this is the case for each σ, the procedure returns true.

```
1  ############ EXAMINE PROCEDURE ######################
2
3  H = {v in V_K | exist s in K and v in s and |V_s|>=|V_P|}
4  I_P(v) = H for v in V_P
5
6  sort v in V_P by size of UI(v)
7
8  for v in V_P:
9      Cands = UI(v)\I_P(v);
10     c = 0;
11     for u in Cands:
12         c = c+1
13         if |G(u)| < |G(v)|:
14             M = {};
15             M[v] = u
16             M = findmatches(K,P,UI,M)
17             if |M| = |V_P|:
```

```
                update and propagate I_P with M
            if |Cands U I_P(v)| - c < t:
                return {}
return I_P


function findmatches (K,P,UI,M):
    if |M| = V_P:
        return M

    x = next vertex to match
    for n in Cands:
        if satisfiesconstraints(K,P,M,x,n):
            M[x] = n
            M = findmatches(K,P,UI,M)
            if |M| = |V_P|:
                return M
    return M
```

At the end of Procedure findMatch, M contains an assignment found starting with $M[v] = n$. If the assignment is complete, the image sets I_P are updated with M.

4.4.5 Conclusion

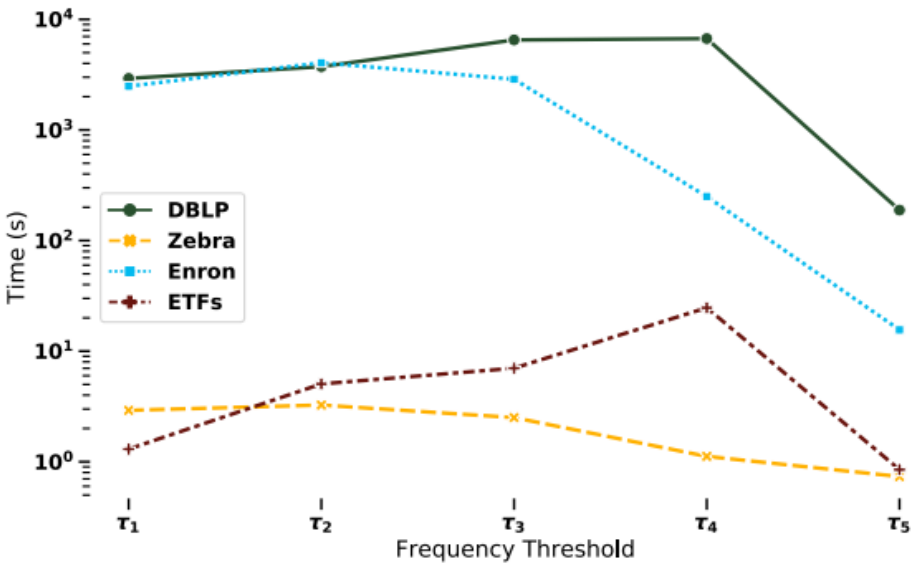


Figure 4.4: Scalability in various simplicial complexes.. Taken from [PDB22]

Using this implementation (executable is available on [lad21]) and using the provided datasets, the algorithm turns out to be very scalable, using five value of frequency thresholds, the algorithm finds roughly the same number of simplets in each dataset see figure.
Notice that the running time follows a concave curve: at higher frequencies there are few candidate frequent simplets, and therefore the algorithm terminates earlier; while at lower frequencies, even though the number of candidate frequent simplets is much larger, the presence of higher order simplices in the complex simplifies the task of searching for occurrences.
Furthermore, observing the figure we can notice that if we modeled the datasets as simple graphs, we would find only the simplets denoted in dark green, corresponding to 1-dimensional structures which can be mapped to graph edges without loss.
By looking for frequent simplets in the datasets modeled as simplicial complexes, we are able to detect a large number of higher-order structures. In particular, in orange we can see the number of frequent simplets consisting of simplices with dimension at most 2 (i.e., closed triangles), while the light blue bar indicates the number of simplets including simplices with dimension at most 3 (i.e., closed tetrahedra). Some of the simplets of dimension 3 persist over multiple frequency thresholds, which indicates that these higher-order structures are not trivially frequent. All the simplets with dimension larger than 1 can be neither modeled nor mined by adopting traditional graph-based approaches.

Finally, we observe that The differences in the distributions of frequent simplets highlight that the our proposal captures different characteristics of the underlying simplicial complexes.

4.5 An alternative solution: the Traversal Algorithm

In this section we will present an interesting application of TDA used for solve Frequent Itemset Mining problems, the Traversal Algorithm [Lin16]. This algorithm exploit the data mapping structure presented in previous section and find the frequent itemset using an iterative method based on the "subcone construction".

4.5.1 The cone and subcone construction.

Using the Definition 2.4.6., the set V of vertex is composed by the "cell" of the table; the bag (also called in following section "bag relation" are composed by each tuple (row of the table or transaction).

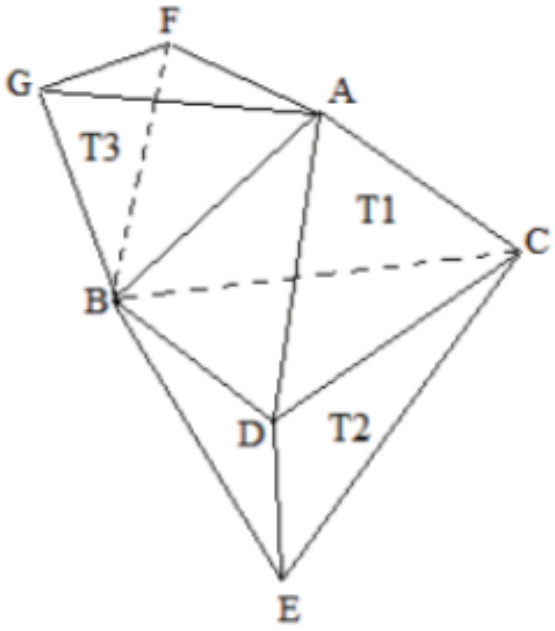


Figure 4.5: Tetrahedron, taken from [Lin16]

Now we present a set of theorem that allow to treat classical FIM problems using a simplicial complex structure.

Theorem 4.5.1. For every bag relation there is an isomorphic geometric bag relation.

Theorem 4.5.2. For every geometric bag relation H there is a bag simplicial complex (V_H, S_H) such that every sub-tuple (frequent itemset without threshold) corresponds to an open simplex and every tuple corresponds to a closed simplex.

Theorem 4.5.3. The collection of frequent itemsets (above a threshold) determines a bag simplicial sub-complex (V_f, S_f) of the simplicial complex (V_H, S_H) (above the same threshold) of the relation H.

At this point is simple to observe that frequent itemsets problem can be approached geometrically using a simplicial complex data structure.

Below is presented an algorithm based on the "sub-cone construction".

The cone construction Let $(V; S)$ be a given simplicial sub-complex as defined in definition 3.6.6. Let n be a new point not in V (this point represents the "sky vertex"). A new bag simplicial complex, denoted by $n(V; S)$ and called the cone on $(V; S)$ with a new vertex n, is the pair $(nV; nS)$, where nV consists of all the original vertices and new vertex n, and nS consists of all original simplices and new simplices.

The sub-cone construction Roughly, it is constructed by the cone construction but with one additional constraint that is all new simplices have to be a cone with a base being an existing simplexe in S. Let $(VW; SW)$ be a given simplicial sub-complex of $(V; S)$. Let n be a point in V but not in VW. Then a new bag simplicial sub-complex of $(V; S)$, denoted by $(VW; SW)$ called the sub-cone in $(V; S)$ with a new vertex n, is the pair $(nVW; nSW)$, where nVW; consists of all the original vertices and new vertex n, and nSW consists of all original simplices and new simplices $\Delta(n; originalvertices) \in S$.

4.5.2 The Algorithm

The key idea is to apply recursively the sub-cone construction in order to count the points that have the highest repetitions.

A	**B**	**C**	**D**
1	1	1	0
1	0	1	1
0	1	1	1
0	1	0	1
1	1	0	0
1	0	0	1

Table 4.2: Transactions Table

Using the iterative sub-cone construction (starting from an empty set of vertex V) we obtain:

1. Using vertex B:

   V1 = BV = ($\triangle B$,4)
   were the second component is the frequency

2. Adding A we have:

   V2 = AV1 = ($\triangle A4, \triangle B4, (\triangle(A, B)$,2)

3. Adding C we have:

   V3 = CV2 = ($\triangle A4, \triangle B4, (\triangle(A, B)2$,
   $\triangle C, \triangle(A, C)2, \triangle(B, C)2, \triangle(A, B, C)$,1)

4. Adding D we have:

   V4 = DV3 = ($\triangle A4, \triangle B4$,
   $(\triangle(A, B)2, \triangle C, \triangle(A, C)2, \triangle(B, C)2, \triangle(A, B, C)$,
   $\triangle\ D, \triangle(A, D)2, \triangle(B, D)2, \triangle(A, C, D)1, \triangle(B, C, D)1)$

The second component represents the traverse of tetrahedron.

4.5.3 Implementation

Following is presenting a python code that implements the Traversal Algorithm. In this code, (pandas and itertools libraries are used [Sub20]) we have used the same data of Table 3.1 in order to obtain the same results presented before.

```
#import libraries
import pandas as pd
import itertools

#set input parameters and convert it in pandas dataframes

t = {'A': [1,0,1,1], 'B': [1,1,0,1],'C': [1,1,1,0],'D': [0,1,1,1]}
e = {'A': [0,0,0,0], 'B': [0,0,0,0],'C': [0,0,0,0],'D': [0,0,0,0]}

t1 = pd.DataFrame(data=t)
e1 = pd.DataFrame(data=e)


#print(t1)
#print(e1)
```

```
16
17 def print_col(list_col):
18
19     string = ''
20     for i in range(len(list_col)):
21         string = string+str(list_col[i])
22     return string
23
24
25 def counting_occurrencies(t,list_col):
26
27     count = 0
28     t_temp = t[list_col]
29     occurrency = []
30
31     for i in range(t.shape[0]):
32
33         sum_temp = 0
34
35         for j in range(len(list_col)):
36
37             sum_temp = sum_temp + t_temp.iloc[i][j]
38
39             if(sum_temp == len(list_col)):
40                 count = count + 1
41
42     if(count > 0 ):
43         occurrency.append(str(count)+print_col(list_col))
44
45     return occurrency
46
47
48 #construct subcone
49 def subcone_contruction(t):
50
51     list_col = t.columns
52     perm_list = []
53     traversal_list = []
54     traversal_list_temp = []
55
56     for i in range(len(list_col)):
57
58         perm_temp = (list(itertools.combinations(list_col,i+1)))
59         perm_list.append(perm_temp)
60
61
62     for i in range(len(perm_list)):
63
64         for j in range(len(perm_list[i])):
```

```
65              traversal_list_temp.append(counting_occurrencies(t,list(
    perm_list[i][j])))
66
67      for i in range(len(traversal_list_temp)):
68
69          for j in range(len(traversal_list_temp[i])):
70              traversal_list.append(traversal_list_temp[i][j])
71
72      return (traversal_list)
73
74 #implementing traversal search using iteratively subcone_construction
       method
75 def traversal_search(t):
76      subcone = []
77      subcone_tmp = []
78      sky_points= []
79
80      for i in range(t.shape[1]):
81          sky_points.append(t.columns[i])
82          #print(sky_points)
83          subcone_tmp.append(subcone_contruction(t[sky_points]))
84          subcone = (subcone_tmp[-1])
85          #print(subcone)
86
87      return subcone
88
89 a = (traversal_search(t1))
90
91 print('Frequent Itemsets found is: ')
92 print(a)
```

How we can see, Traversal Algorithm is based on iterative *sub-cones* construction (see method at line 42 which is invoked iteratively later).

The output of this simple code is (notice that at every iteration step, results are printed):

['3A']
['3A', '3B', '2AB']
['3A', '3B', '3C', '2AB', '2AC', '2BC', '1ABC']
['3A', '3B', '3C', '3D', '2AB', '2AC', '2AD', '2BC', '2BD', '2CD', '1ABC', '1ABD', '1ACD', '1BCD'].

4.5.4 Advantages

Experimental results (see [Lin16]) prove that simplicial complex method is faster than FP-growth method furthermore its memory usage is more efficient. We can find theoretical reason of that in different data representation between FP-Growth and Traversal Algorithm: the FP-Growth method represents each tuple by a linear sequence of n-branches from root to leaf; such a representation implicitly regard the tuple as an ordered sequence, hence FP-Growth method has to deal with such an order. While simplicial complex method represents a tuple by closed simplex, there is no order in this representation, and this improve both computation speed. Furthermore FP-Growth need to save intermediate data structure (iteration) in memory scanning the input dataset more than once; it scans data during constructing of conditional FP-Tree, which is a recursive procedure; instead given a bag relation, Traversal Algorithm is that once a bag relation is given, its bag simplicial complex of the relation and its bag simplicial sub-complex of frequent itemsets are there, so the association rule mining algorithm does not need to "construct the FP-tree" (computationally and memory expensive); it just simply goes directly to traversal the sub-complex of frequent itemsets.

The stress test tells us that we can handle frequent itemsets of length 30 in reasonable time; no existing method can do that. This method has a great potential in big data analysis.

The presented algorithm it is not only very fast but also use much less memory than existing methods.
Using a set of data provided by National Taiwan University Hospital consisting of a Database with 65.536 transaction and 1257 columns (binary values), using a support threshold of 0,042, the presented new algorithm takes only 5 seconds to find all frequent itemsets using 35,2 MB of memory, while FP-Growth takes 14,6 minutes and use 100MB of memory; so simplicial complex method is about 200 times faster and moreover the memory usage is substantially less [Lin16].

Conclusions and future challenges

In this thesis we have defined and discussed the Frequent Itemset Mining problem, retracing since its original formulation and solution proposed by Agrawal, leading to FreSCo Algorithm that represent the state of art Topological Data Analyisis (TDA) application to FIM.
How we can say, even if Apriori algorithm (the first presented in this thesis) turn out to be obsolete, its key idea (the *Apriori principe*) is still relevant today and constitutes a great source of inspiration.
In fact FP-Growth (actually how we can say, one of the most used algorithm for FIM tasks) could be considered a reinterpretation of Apriori as it reuses di principal concept (see the pruning of not relevant item from the conditional tree) of Apriori, and then exploit the graph data structure in order to improve time execution and reduce memory consumption.
On other hand also the STruD decomposition (prior work for FreSCo) use the Apriori principle to map efficiently datapoints in the topological data structure exploited by FreSCo.
The great innovation and extension of FreSCo with the respect to the classical FIM algorithm is this capability to manage high order relation data structure replacing the concept of edge (in graphs data structure based algorithm) with the concept of simplex. For example this allow to overcomes a structural limitation of FP-Growth that occurs when is not possible to map the data relation of an input dataset into a graph; we notice also that when FreSCo works with 1-simplex relation (hence data structure is mapped into a graph) its performances are comparable with FP-Growth.

As future challenges we could apply FreSCo algorithm to relate FIM tasks as Infrequent Itmeset Mining ([CG14]) or to "Generalized itemset mining" ([SA97]) borrowing from what has been done with FP-Growth in order to extend this algorithms to high order relation datasets. Furthermore, the powerful representation provided by simplicial complexes could be exploited in the "high-utility itemset mining" ([Fou+19]) in which the high order relation expressed by the simplex can be used to represent the high utility itemsets. In the end, considering that the experimental results provided in the works [PDB22], [PDB21b] are obtained

without using a distributed infrastructure and being the Truss Decomposition and frequent simplet search tasks, more computationally expensive (as we have seen the great challenges in STruD is to manage the combinatorial explosion caused by the downward closure property of complexes) a good idea could be to provide a spark-based application.

Bibliography

[AAE19] Mehmet E Aktas, Esra Akbas, and Ahmed El Fatmaoui. «Persistence homology of networks: methods and applications». In: *Applied Network Science* 4.1 (2019), pp. 1–28.

[AIS93a] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. «Database mining: A performance perspective». In: *IEEE transactions on knowledge and data engineering* 5.6 (1993), pp. 914–925.

[AIS93b] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. «Mining association rules between sets of items in large databases». In: *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 1993, pp. 207–216.

[AS+94] Rakesh Agrawal, Ramakrishnan Srikant, et al. «Fast algorithms for mining association rules». In: *Proc. 20th int. conf. very large data bases, VLDB*. Vol. 1215. Citeseer. 1994, pp. 487–499.

[Ban14] Urvashi Bansal. «ECLAT Algorithm for Frequent Item sets Generation». In: *International Journal of Computer System* (Jan. 2014).

[BFZ20] Mattia Bergomi, Massimo Ferri, and Lorenzo Zuffi. «Topological graph persistence». In: *Communications in Applied and Industrial Mathematics* 11 (Dec. 2020), pp. 72–87. DOI: 10.2478/caim-2020-0005.

[BGH21] Austin R Benson, David F Gleich, and Desmond J Higham. «Higher-order network analysis takes off, fueled by classical ideas and new data». In: *arXiv preprint arXiv:2103.05031* (2021).

[Bla19] Paul E. Black. «"Manhattan distance". Dictionary of Algorithms and Data Structures.» In: (Oct. 2019).

[Bre+04] P-T Bremer et al. «A topological hierarchy for functions on triangulated surfaces». In: *IEEE Transactions on Visualization and Computer Graphics* 10.4 (2004), pp. 385–396.

[Car+05] Gunnar Carlsson et al. «Persistence barcodes for shapes». In: *International Journal of Shape Modeling* 11.02 (2005), pp. 149–187.

[Car15] G. Carlsson. «Carlsson, The Shape of Data conference, Graduate School of Mathematical Sciences, University of Tokyo». In: (Nov. 2015).

[Cas16] Roberto Castelli. «Topologically Distinct Collision-Free Periodic Solutions for the N-Center Problem». In: *Archive for Rational Mechanics and Analysis* Accepted (Oct. 2016). DOI: 10.1007/s00205-016-1049-0.

[CG14] Luca Cagliero and Paolo Garza. «Infrequent Weighted Itemset Mining Using Frequent Pattern Growth». In: *IEEE Transactions on Knowledge and Data Engineering* 26.4 (2014), pp. 903–915. DOI: 10.1109/TKDE.2013.69.

[Cha+13] Frédéric Chazal et al. *Optimal rates of convergence for persistence diagrams in topological data analysis*. 2013.

[CM] F. Chazal and B. Michel. «An introduction to Topological Data Analysis: fundamental and practical aspects for data scientists». In: ().

[Fou+19] Philippe Fournier-Viger et al. «High-utility pattern mining». In: *Cham: Springer* (2019).

[GH18] Marvin J Greenberg and John R Harper. *Algebraic topology: a first course*. CRC Press, 2018.

[GK13] Kanwal Garg and Deepak Kumar. «Comparing the Performance of Frequent Pattern Mining Algorithms». In: *International Journal of Computer Applications* 69 (May 2013), pp. 21–28. DOI: 10.5120/12129-8502.

[Goe] Bart Goethals. *http://fimi.uantwerpen.be/*.

[GUW08] H Garcia-Molina, JD Ullman, and J Widom. *Database Systems: The Complete Book. 2. international ed edn*. 2008.

[HKP12] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Jan. 2012. DOI: 10.1016/C2009-0-61819-5.

[HMR08] Danijela Horak, Slobodan Maletić, and Milan Rajkovic. «Persistent Homology of Complex Networks». In: *J. Stat. Mech.: Theory Experiment* 2009 (Nov. 2008). DOI: 10.1088/1742-5468/2009/03/P03034.

[HMR09] Danijela Horak, Slobodan Maletić, and Milan Rajković. «Persistent homology of complex networks». In: *Journal of Statistical Mechanics: Theory and Experiment* 2009.03 (2009), P03034.

[htta] nlab portal at site: https://ncatlab.org/nlab/show/Introduction+to+Topology+1 +1. «Introduction to Topology 1». In.

[httb] https://uwsdatamining.wordpress.com/market-basket-analysis/. «Data Mining: Market Basket Analysis». In.

[Iqb+21] Sohail Iqbal et al. «Classification of COVID-19 via Homology of CT-SCAN». In: Feb. 2021.

[KHF16] Genki Kusano, Yasuaki Hiraoka, and Kenji Fukumizu. «Persistence weighted Gaussian kernel for topological data analysis». In: *International conference on machine learning*. PMLR. 2016, pp. 2004–2013.

[lad21] lady-bluecopper. «https://github.com/lady-bluecopper/FreSCo». In: 2021.

[Lin16] Tsau-Young Lin. «Very fast frequent itemset mining: Simplicial complex methods». In: *2016 IEEE International Conference on Big Data (Big Data)*. IEEE. 2016, pp. 1946–1949.

[LL96] Marc Lachièze-Rey and Jean-Pierre Luminet. «Cosmic Topology». In: *Physics Reports* 254 (May 1996), pp. 135–214. DOI: 10.1016/0370-1573(94)00085-H.

[Man14] Marco Manetti. *Topologia*. Vol. 78. Springer Science & Business Media, 2014.

[Mas20] Hosein Masoomy. «Topological Analysis of Biological Datasets». In: June 2020. DOI: 10.13140/RG.2.2.33608.60163.

[Mas21] Hosein Masoomy. «Persistent Homology: Methods and Applications». In: July 2021. DOI: 10.13140/RG.2.2.25639.42401.

[MRV08] Slobodan Maletić, Milan Rajković, and Danijela Vasiljević. «Simplicial complexes of networks and their statistical properties». In: *International Conference on Computational Science*. Springer. 2008, pp. 568–575.

[Pat19] Alice Patania. «Simplicial Data Analysis». In: (2019).

[PDB21a] Giulia Preti, Gianmarco De Francisci Morales, and Francesco Bonchi. «STruD: Truss Decomposition of Simplicial Complexes». In: New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450383127. DOI: 10.1145/3442381.3450073. URL: https://doi.org/10.1145/3442381.3450073.

[PDB21b] Giulia Preti, Gianmarco De Francisci Morales, and Francesco Bonchi. «STruD: Truss Decomposition of Simplicial Complexes». In: *Proceedings of the Web Conference 2021*. 2021, pp. 3408–3418.

[PDB22] Giulia Preti, Gianmarco De Francisci Morales, and Francesco Bonchi. «FreSCo: Mining Frequent Patterns in Simplicial Complexes». In: *Proceedings of the ACM Web Conference 2022*. 2022, pp. 1444–1454.

[Ras18] Sebastian Raschka. «MLxtend: Providing machine learning and data science utilities and extensions to Python's scientific computing stack». In: *The Journal of Open Source Software* 3.24 (Apr. 2018). DOI: 10.21105/joss.00638. URL: http://joss.theoj.org/papers/10.21105/joss.00638.

[RWS16] Krzysztof Rykaczewski, Piotr Wiśniewski, and Krzysztof Stencel. «An Algorithmic Way to Generate Simplexes for Topological Data Analysis». In: Sept. 2016.

[SA97] Ramakrishnan Srikant and Rakesh Agrawal. «Mining generalized association rules». In: *Future generation computer systems* 13.2-3 (1997), pp. 161–180.

[Sid+14] Shivam Sidhu et al. «FP Growth Algorithm Implementation». In: *International Journal of Computer Applications* 93 (May 2014), pp. 6–10. DOI: 10.5120/16233-5613.

[Spa] Apache Spark. *https://spark.apache.org/docs/latest/ml-frequent-pattern-mining*.

[Sub20] Abdulhamit Subasi. «Chapter 3 - Machine learning techniques». In: *Practical Machine Learning for Data Analysis Using Python*. Ed. by Abdulhamit Subasi. Academic Press, 2020, pp. 91–202. ISBN: 978-0-12-821379-7. DOI: https://doi.org/10.1016/B978-0-12-821379-7.00003-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128213797000035.

[TH12] Peter Trebuňa and Jana Halčinová. «Experimental Modelling of the Cluster Analysis Processes». In: *Procedia Engineering* 48 (Dec. 2012), pp. 673–678. DOI: 10.1016/j.proeng.2012.09.569.

[Tie17] Julien Tierny. «Introduction to topological data analysis». In: (2017).

[Toi10] Hannu Toivonen. «Frequent Itemset». In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 418–418. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_317. URL: https://doi.org/10.1007/978-0-387-30164-8_317.

[TSK16] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to data mining*. Pearson Education India, 2016.

[TZH14] Chad Topaz, Lori Ziegelmeier, and Tom Halverson. «Topological Data Analysis of Biological Aggregation Models». In: *PloS one* 10 (Dec. 2014). DOI: 10.1371/journal.pone.0126383.

[UKA+04] Takeaki Uno, Masashi Kiyomi, Hiroki Arimura, et al. ≪LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets≫. In: *Fimi*. Vol. 126. 2004.

[Wri16] M. Wright. ≪Introduction to Persistent Homology video on M. Wright channel https: //www.youtube.com/watch?v = 2PSqW BIrn90≫. In: (Oct. 2016).

[Zan17] Paolo Zanardi. ≪Algebraic Topology, Quantum Algorithms and Big Data≫. In: (2017).

[Zha+21] Wengang Zhang et al. ≪Application of deep learning algorithms in geotechnical engineering: a short critical review≫. In: *Artificial Intelligence Review* 54 (Dec. 2021), pp. 1–41. DOI: 10.1007/s10462-021-09967-1.