Data Mining
Lecture Notes F2023

Davide Mottin
Department of Computer Science
Aarhus University
Aabogade 34, 8200 Aarhus N
davide@cs.au.dk

Acknowledgements

I want first to extend a special gratitute to Mathias Tversted who gently passed me the summary of last year's slides and offered to be a TA for this year and help with the notes and the exercises. I would also like to thank Wenye Ma, Zhile Jiang, Petros Petsinis, Maximilian Egger, and Konstantinos Skitsas for supporting and help correcting and revising these notes.

Contents

Preface

One day, in the 2021 winter, I had the crazy idea to start reorganizing the notes for the data mining Master's course I am teaching. My original thought was to get an even deeper understanding of the concepts and at the same time to provide an extra tool for students to understand the concepts underlying the different algorithms explained during the lectures. These lecture notes are the result of my thought and my hours of back pain.

They are still a work in progress and they will probably be for a long time, but at least they represent some good initial starting point to organize the growing discipline of data mining. Any comment is well accepted and well received, so feel free to reach me out and proposing any change. These notes are far from being a mature book. As such, please refer to excellent sources in Data Mining to have a more exhaustive introduction to the topic. The notes are intended as a complement, and often a summary, of the course slides.

The course, as it was taught in the 2022 Spring semester is divided into three modules. The first module, Part I is about clustering algorithms and outlier detection, and how to deal with high-dimensional data. Part II introduces the unsupervised problems with graphs and network data; graphs are inherently high-dimensional and dealing with them requires specialize techniques. The last module in Part III digs into some of the problems of retrieving patterns and dealing with large number of similar elements.

Part I

Clustering

This module introduces the concept of clustering. The purpose of clustering is to find groups of objects that share similar characteristics. Clustering refers to a set of techniques that takes a set of points as input and tries to partition these points into "homogeneous" groups. The concept of homogeneity and similarity will become more apparent in the next sections. We would like to group each data point such that it is:

1. Highly similar to any other point in its cluster
2. Dissimilar to any other point in another cluster

More formally, we are given a set $\mathcal{D}$ of n data **points** $\mathbf{x} = (x_1, ..., x_d)$ represented as a vector (i.e., a sequence of real numbers) in a d-dimensional space $\mathbb{R}^d$. **Clustering** aims at finding a set of k clusters $\mathcal{C} = \{C_1, ..., C_k\}$ such that a similarity σ between each pair of points $(\mathbf{x}_i, \mathbf{x}_j)$ is "high" if both points belong to the same cluster and "low" otherwise. It is clear that such definition is quite imprecise. However, as we will see different techniques provide different notions of similarity, which greatly affect the final result.

Example 1: Take for instance the set buyers in our online shop. If we analyse their profile based on their preferences, the shopping cart, and their items bought in previous transactions, we can identify different types of users. For instance, we could find that some users buy mostly cosmetics while others are mostly interested in books. Having such information is important to optimize our website and increase the revenue.

Example 2: Another example is related to fraudulent activities. The police investigation is interested in finding whether companies are involved in fiscal frauds. By grouping transactions it is possible to distinguish normal from fraudulent transactions and trace back to the company.

➳ Objectives

After finishing this module, you should be able to:

- Describe the purpose of clustering and why classification is not a viable alternative.
- Discuss the main applications of clustering and outlier detection based on the type of data.
- Compare clustering and outlier detection.
- Categorize clustering algorithms based on their characteristics (hard vs. soft clusters, efficiency, high vs. low-dimensional data, etc.).
- Apply the main clustering algorithms and run them over small datasets.
- Generalize the algorithms to other kind of data.

Lecture 2

Representative-based clustering

Instructor: *Davide Mottin*

In this lecture, we introduce representative-based clustering. In representative-based clustering a point is chosen as a summary or a *representative* of the cluster. The representative is intuitively a central point in the cluster, such that the other points in the cluster are close to it.

We consider three notable representative-based clustering algorithms

- k-means in which each cluster is represented by a central point potentially not part of the dataset $\mathcal{D}$
- k-medoid in which each cluster is represented by the point with the minimum distance among all the others in the cluster
- Expectation-Maximization (EM) in which a cluster is represented by the parameters (mean and standard deviation) of a Gaussian distribution.

1.1 k-means clustering

The k-means algorithm returns a clustering $\mathcal{C} = (C_1, ..., C_k)$ such that the sum of the squared distances between the mean (or centroid) of the cluster to which each point is assigned is minimized.

For a cluster C_i the centroid $\boldsymbol{\mu}_i$ is a vector, such that

$$\boldsymbol{\mu}_i = \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}$$

For any distance measure *dist* we compute the *compactness* of cluster C_i

$$TD(C_i) = \sqrt{\sum_{\mathbf{x} \in C_i} dist(\mathbf{x}, \boldsymbol{\mu}_i)^2}$$

k-means aims at minimizing the overall sum of squares expressed as

$$TD = \sqrt{\sum_{i=1}^{k} TD^2(C_i)} \tag{1.1}$$

Note that if *dist* is the Eucledean distance, Eq. 1.1 can be equivalently expressed as

$$\sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 \tag{1.2}$$

The above objective cannot be easily minimized. To intuitively understand why, you can see how Eq. 1.2 requires to first know the centroids $\boldsymbol{\mu}_i$, which in turns requires to know which points are assigned to each cluster. In particular there are two objectives. However, if we knew the centers, we could assign each point to the closest center. This intuition is the idea behind the Lloyd's Algorithm [Llo82].

Lloyd's Algorithm Lloyd's algorithm iterates over two phases:

1. **Cluster assignment:** Assign the points to the cluster with the closest centroid. Formally, assign a point $\mathbf{x} \in \mathcal{D}$ to the cluster j such that

$$j = \arg\min_i \|\mathbf{x} - \boldsymbol{\mu}_i\|$$

2. **Centroid update:** Move each centroid according on the points assign to it, as follows

$$\boldsymbol{\mu}_j = \frac{1}{|C_j|} \sum_{\mathbf{x} \in C_j} \mathbf{x}$$

The algorithm keeps running until convergence. In other words, when there is only negligible change in the centroids in two consecutive iterations $t-1$ and t, i.e., $\sum_{i=1}^{k} \|\boldsymbol{\mu}_i^t - \boldsymbol{\mu}_i^{t-1}\|^2 \leq \varepsilon$ for a very small value of ε.

✔ Advantages

- Efficient $\mathcal{O}(tkn)$, where n is number of points, k is number of clusters and t is number of iterations.
- Normally k, t is much smaller than n.

✘ Disadvantages

- Mean (or centroid) point over the dataset needs to be defined.
- k needs to be specified ahead of running the algorithm.
- Sensitive to outliers.
- Can only model convex[a] (in k-means' case spherical) clusters.
- Often terminates at local optimum and runtime depends heavily on initial partition.

[a] A subset of a Euclidean space is convex if, given any two points in the subset, the subset contains the whole line segment that joins them.

1.1.1 Initialization of the clusters

Since the Lloyd's algorithm is heuristic, there is no guarantee that its solution minimizes the function in Eq. 1.2. This means that the performance of the algorithm depends on how the initial centroids are chosen. One strategy to avoid non optimal solutions is the one proposed by [BF98]. The intuitive idea is to find good initial clusters by applying Lloyd's algorithm on different samples of the data.

- Draw m different samples from the data set (of fixed size)
- Cluster each sample to get m estimates for k representatives. That is,

$$A = (A_1, \ldots, A_k), B = (B_1, \ldots, B_k), ..., M = (M_1, \ldots, M_k) \tag{1.3}$$

- Cluster $DS = A \cup B \cup ... \cup M$ exactly m-times with $A, B, ..., M$ as initial partitioning
- Use the best of these m representatives as initial clustering of the entire set.

1.1.2 Choosing k

Another issue with k-means is the choice of the parameter k that is usually provided as an input to the algorithms. However, a natural question arises *is it possible to discover the parameter k automatically?*. In principle, we could try different values of k and check which one minimizes the objective in Eq. 1.2. However, note that the objective decreases as k increases and is 0 when all points are each assigned to a different cluster. To avoid this phenomenon we can use the Silhouette coefficient [Rou87].

The main idea behind the Silhouette coefficient is that in order to decide whether a clustering is good, one needs to look not only at the distances within the cluster in which the point is assigned, but also the distances between the second nearest cluster.

We define $int(\cdot)$ as the average distance between a point $\mathbf{x}$ and the objects in its own cluster $C(\mathbf{x})$

$$int(\mathbf{x}) = \frac{1}{|C(\mathbf{x})| - 1} \sum_{\substack{\mathbf{p} \in C(\mathbf{x}) \\ \mathbf{p} \neq \mathbf{x}}} dist(\mathbf{p}, \mathbf{x})$$

and *ext* is the average distance between the point $\mathbf{x}$ and objects in its second closest cluster

$$ext(\mathbf{x}) = \min_{C_i \neq C(\mathbf{x})} \frac{1}{|C_i|} \sum_{\mathbf{p} \in C_i} dist(\mathbf{p}, \mathbf{x})$$

Definition 1.1.1. *The Silhouette $s(\mathbf{x})$ of a point $\mathbf{x}$ is defined as the normalized difference between the average distance of the object with the second closest cluster and the average distance of the objects in its own cluster*

$$s(\mathbf{x}) = \frac{ext(\mathbf{x}) - int(\mathbf{x})}{\max\{int(\mathbf{x}), ext(\mathbf{x})\}}, \text{if } |C(\mathbf{x})| > 1 \tag{1.4}$$

In clusters with only one point, i.e., $|C(\mathbf{x})| = 1$, the Silhouette coefficient $s(\mathbf{x})$ is 0.

The global Silhouette coefficient $s_{\mathcal{C}}$ for a clustering $\mathcal{C}$ is the average of the Silhouette coefficients for all the points $\mathbf{x} \in \mathcal{D}$, namely

$$s_{\mathcal{C}} = \frac{1}{|\mathcal{D}|} \sum_{\mathbf{x} \in \mathcal{D}} s(\mathbf{x}) \tag{1.5}$$

Interpretation of the silhouette coefficient The silhouette coefficient in Eq. 1.4 takes values between -1 and 1. In particular,

- $s(\mathbf{x}) = -1$ is bad because it is on average closer to members of the second closest cluster
- $s(\mathbf{x}) = 0$ means $\mathbf{x}$ is in between two clusters
- $s(\mathbf{x}) = 1$ means that $\mathbf{x}$ is closest to members of its own cluster, which is the best

If we compute the average coefficient of a clustering s_C, then

- $0.7 < s_{\mathcal{C}} \leq 1$ indicates *strong structure*
- $0.5 < s_{\mathcal{C}} \leq 0.7$ indicates *medium structure*
- $0.25 < s_{\mathcal{C}} \leq 0.5$ indicates *weak structure*
- $s_{\mathcal{C}} \leq 0.25$ indicates *no structure*

However, the absolute values of the Silhouette coefficient do not have a precise meaning. Thus, one can compare these values across data and algorithms, or plot the value of $s_{\mathcal{C}}$ to determine the value k.

1.2 Kernel k-means

One, less apparent, limitation of k-means is that it considers linear boundaries, meaning that the separation among clusters is a line. This limitation is intuitively arising by the use of linear differences among points. To overcome this limitation, we will use the "kernel trick".

✍ Recap

Check the Zaki and Meira book [ZMJ20, Chapter 5] if you don't remember the kernel trick.

We are going to use kernels to project points into a different space. In particular, we first *map* the points into a different space and separate the points with linear boundaries in the transformed space. The mapping function ϕ projects each point $\mathbf{x}$ into a different space $\phi(\mathbf{x})$ with, potentially, different dimensionality. If we use this transformation we can rewrite k-means objective in 1.2 as

$$\min_{\mathcal{C}} \sum_{C_i \in \mathcal{C}} \sum_{\mathbf{x} \in C_i} \|\phi(\mathbf{x}) - \phi(\boldsymbol{\mu}_i)\|^2 \tag{1.6}$$

By expanding $\|\phi(\mathbf{x}) - \phi(\boldsymbol{\mu}_i)\|^2$ we obtain

$$\begin{aligned}\|\phi(\mathbf{x}) - \phi(\boldsymbol{\mu}_i)\|^2 &= \|\phi(\mathbf{x})\|^2 - \|\phi(\boldsymbol{\mu}_i)\|^2 + 2\phi(\mathbf{x})^\top \phi(\boldsymbol{\mu}_i) \\ &= \phi(\mathbf{x})^\top \phi(\mathbf{x}) - \phi(\boldsymbol{\mu}_i)^\top \phi(\boldsymbol{\mu}_i) + 2\phi(\mathbf{x})^\top \phi(\boldsymbol{\mu}_i)\end{aligned}$$

The dot product $\phi(\cdot)^\top \phi(\cdot)$ is called a kernel and can be represented as a function $\kappa(\cdot, \cdot)$ that takes in input two vectors and returns a dot-product, intuitively a similarity, among the two vectors. By observing that $\|\boldsymbol{\mu}_i\|^2$ can be computed as

$$\phi(\boldsymbol{\mu}_i)^\top \phi(\boldsymbol{\mu}_i) = \frac{1}{|C_i|^2} \sum_{\mathbf{x}_a \in C_i} \sum_{\mathbf{x}_b \in C_i} \kappa(\mathbf{x}_a, \mathbf{x}_b)$$

we can further expand $\|\phi(\mathbf{x}) - \phi(\boldsymbol{\mu}_i)\|^2$ into

$$\kappa(\mathbf{x}, \mathbf{x}) - \frac{2}{|C_i|} \sum_{\mathbf{x}_a \in C_i} \kappa(\mathbf{x}_a, \mathbf{x}) + \frac{1}{|C_i|^2} \sum_{\mathbf{x}_a \in C_i} \sum_{\mathbf{x}_b \in C_i} \kappa(\mathbf{x}_a, \mathbf{x}_b) \tag{1.7}$$

Putting all together, we can easily transform the Lloyd's algorithm by adapting the computation of the centroids and the cluster assignments. Besides the choice of the initial points, kernel k-means depends only on the choice of the kernel. We apply Eq. 1.7 to find the cluster assignment $C^*(\mathbf{x}) = \arg\min_i \|\phi(\mathbf{x}) - \phi(\boldsymbol{\mu}_i)\|^2$ and adapt the equation to find the new centers. The rest of the algorithm remains the same as before.

✔ Advantages

- Allows detection of clusters with arbitrary shape.
- Fits any possible kernel.

✘ Disadvantages

- Inefficient, as it requires computation of a square kernel matrix, taking $\mathcal{O}(n^2 k)$.
- Need to specify k ahead of time.
- Needs to specify a kernel ahead of time.

1.3 k-Medoid

k-medoid is another popular representative-based clustering method in which the representative point for each cluster is a *medoid*, that is *one of the points in the cluster* as opposed to k-means in which the centroid can be an arbitrary point in the space. Another difference with k-means is that k-medoid uses the Manhattan distance (also called taxicab distance, L_1 distance, or ℓ_1 norm).

Definition 1.3.1 (Manhattan distance)**.** *The Manhattan distance between two vectors (or points)* $\mathbf{x}$ *and* $\mathbf{y}$ *is the sum of the absolute distances between the values of* $\mathbf{x}$ *and* $\mathbf{y}$*,*

$$d_1(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{d} |\mathbf{x}_i - y_i| \tag{1.8}$$

As opposed to the Euclidean distance, typically used by k-means, the Manhattan distance is more robust to noise and, as such, less affected by outliers. The reason is that the differences among point coordinates are not squared as in the Euclidean distance. k-medoids aims at finding a set of k points in the datasets that minimize the overall sum of Manhattan distances among the points in the clusters and the respective medoid $\mathbf{m}_i$.

$$\sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} |\mathbf{x} - \mathbf{m}_i| \tag{1.9}$$

Finding clusters that minimizes the Manhattan distance is still **NP**-hard. As such, the only hope is to resort to heuristic methods. The most popular heuristic is the Partitioning Around Medoids (PAM) [KR90].

Partitioning Around Medoids The PAM algorithm starts by guessing a first set of medoids and iteratively refines this set by swapping medoids with points that reduce the overall sum of distances.

Algorithm 1 PAM algorithm

Input: Number of cluster k, dataset $\mathcal{D}$

1: Select k arbitrary medoid objects
2: Assign every point to nearest medoid
3: $TD_{current} \leftarrow$ Compute the value of Eq. 1.9
4: **for each** pair of medoid M and non-medoid N **do**
5: Compute $TD_{N \leftrightarrow M}$, which is Eq. 1.9 for the partition that results when swapping M with N
6: Select the non-medoid N, for which $TD_{N \leftrightarrow M}$
7: **if** $TD_{N \leftrightarrow M}$ is smaller than $TD_{current}$ **then**
8: Swap N with M
9: $TD_{current} \leftarrow TD_{N \leftrightarrow M}$
10: Repeat step 3
11: **else**
12: Stop

The PAM algorithm (Algorithm 1) is, unfortunately, quite slow as any swap requires to consider all pairs of points at each iteration. Assuming t iterations, the PAM algorithm runs in $\mathcal{O}(tn^2)$. For these reasons the PAM algorithm has a few optimizations.

CLARA:

1. Introduces parameter *numlocal*, which it draws from the dataset
2. Applies PAM on each sample
3. Returns the best of these sets of medoids as output

CLARANS:

1. Two additional parameters: *maxneighbor* and *numlocal*
2. At most *maxneighbor* many pairs (Medoid M, non-medoid N) are evaluated by the algorithm
3. The first pair (M, N) for which $TD_{N \leftrightarrow M}$ is smaller than $TD_{current}$ is swapped, instead of the minimal
4. Finding the local minimum with this procedure is repeated *numlocal* times

CLARAN's runtime is smaller than CLARA, which is faster than PAM.

✔ Advantages

- Applicable to arbitrary objects with a distance function.
- Not as sensitive to noisy data and outliers (They will not influence the medoid, which is now not the average).

✘ Disadvantages

- Inefficient.
- Like k-means, we need to specify the number of clusters in advance.
- Can only detect convex shapes.
- CLARA and CLARANS vary a lot due to randomization.

1.4 Expectation-Maximization

Until now we have considered clustering methods that assign each point to exactly one cluster. These methods return a *hard clustering* and are basically partitions of the space. Another option is to allow points to participate in multiple clusters. This is a common situation in reality, as a data point might belong to different categories.

Example 1: Take for instance our example of buyers in our online shop. A buyer could be interested both in cosmetics and books, but both k-means and k-medoids would either assign the buyer to one of those category.

To move from hard to soft cluster, we need first to assume a model for our data. We consider a simple Gaussian model, which illustrates well our intuition. Let us now assume that each cluster $C_i \in \mathcal{C}$ is represented as a (multidimensional) Gaussian distribution

$$f(\mathbf{x}|\boldsymbol{\mu}_i, \Sigma_i) = \frac{1}{\sqrt{(2\pi)^d |\Sigma_i|}} e^{-\frac{1}{2}(x-\boldsymbol{\mu}_i)^\top (\Sigma_i)^{-1}(\mathbf{x}-\boldsymbol{\mu}_i)} \tag{1.10}$$

The parameters $\boldsymbol{\mu}, \Sigma$ are the mean and the covariance of the distribution and are the parameters we wish to discover. In particular, those two parameters are unknown in the beginning and represents the centers and the spread of each of the clusters. Similarly, to k-means we want to find representative centers. Our data is modeled as a Gaussian mixture

$$f(\mathbf{x}) = \sum_{i=1}^{k} f(\mathbf{x}|\boldsymbol{\mu}_i, \Sigma_i) P(C_i) \tag{1.11}$$

where we define a prior $P(C_i)$ on cluster C_i. This prior is a probability distribution over the clusters, i.e., $\sum_i P(C_i) = 1$. In an intuitive manner, our data is a landscape of mountains where each mountain's height $P(C_i)$, position $\boldsymbol{\mu}_i$, width Σ_i has to be discovered. See Figure 1.1 for an illustration of a Gaussian mixture.

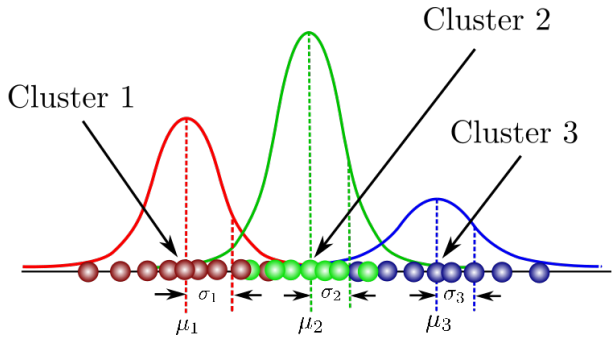


Figure 1.1: A Gaussian mixture is a set of mountains for which we need to determine the position of each mountain, the height, and the width.

In a sense, though, we only know about a few piles of earth that form the mountains, i.e., our data, which is far from being complete. Following the Bayesian approach we want to discover the parameters $\boldsymbol{\theta}_i = (\boldsymbol{\mu}_i, \Sigma_i, P(C_i))$ based on an incomplete, and potentially biased dataset. Assuming each point to be an independent sample, we define a likelihood function

$$P(\mathcal{D}|\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_k) = \prod_{\mathbf{x} \in \mathcal{D}} f(\mathbf{x})$$

We can find the best parameters by maximizing the likelihood, which tells us how probable it is that the data has been generated by a distribution having such parameters. In practice maximizing, the likelihood and its logarithm is equivalent; however, maximizing the logarithm is convenient as every product becomes a sum.

$$\arg\max_{\boldsymbol{\theta}_1,...,\boldsymbol{\theta}_k} \ln P(\mathcal{D}|\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_k) = \ln \prod_{\mathbf{x} \in \mathcal{D}} f(\mathbf{x}) = \sum_{\mathbf{x} \in \mathcal{D}} \ln f(\mathbf{x}) = \sum_{\mathbf{x} \in \mathcal{D}} \ln(\text{Eq. 1.10}) \tag{1.12}$$

Unfortunately, the direct maximization of Eq. 1.12 is very hard due to the unknown assignments to the clusters. We follow again a heuristic approach, which makes use of some more advanced theory that can be found in [ZMJ20, Chapter 13.3.2]. Here we provide some intuitions on the method, which should be sufficient to apply the method and understand the underlying reasoning.

We start by assuming we already know the parameters $\boldsymbol{\theta}$. If that is the case, we could compute the posterior $P(C_i|\mathbf{x})$ through the Bayesian theorem

$$P(C_i|\mathbf{x}) = \frac{P(\mathbf{x}|C_i)P(C_i)}{P(\mathbf{x})} = \frac{P(\mathbf{x}|C_i)P(C_i)}{\sum_{i=1}^{k} P(\mathbf{x}|C_i)P(C_i)} \tag{1.13}$$

We denote these posteriors as weights $w_{ij} = P(C_i|\mathbf{x}_j)$ The probability $P(\mathbf{x}|C_i)$ can be approximated considering a small interval ε around the density $f(\mathbf{x}|\boldsymbol{\mu}_i, \Sigma_i)$

$$P(\mathbf{x}|C_i) \approx 2\varepsilon \cdot f(\mathbf{x}|\boldsymbol{\mu}_i, \Sigma_i) \tag{1.14}$$

Once we know the posteriors, we can calculate a better estimate of the parameters $\boldsymbol{\theta}_i$ for each cluster C_i.

Algorithm 2 EM algorithm

```
Initialize the parameters θ
while not converged do                    ▷ The algorithm does not always converge.
  // Expectation step: Assign points to clusters
    for each point x_j ∈ D and cluster C_i do
        Compute P(C_i|x_j) = w_ij as in Eq. 1.13
  // Maximization step: Compute the model
    for each cluster C_i do
        Update θ_i = P(C_i), μ_{C_i}, Σ_{C_i}
```

The last question we need to answer is *how do we update the parameters* θ_i? The parameters are updated based on the posterior. Intuitively, the posterior, gives an idea on how likely is a cluster for a specific point. In this sense, the posterior is really a weighted way to denote how many points belong to a certain cluster. Let us have a look to the update rules.

- Mean $\boldsymbol{\mu}_i = \dfrac{\sum_{j=1}^{n} \mathbf{x}_j w_{ij}}{\sum_{j=1}^{n} w_{ij}}$
- Covariance $\Sigma_i = \dfrac{\sum_{j=1}^{n} w_{ij}(\mathbf{x}_j - \boldsymbol{\mu}_i)(\mathbf{x}_j - \boldsymbol{\mu}_i)^\top}{\sum_{j=1}^{n} w_{ij}}$
- Prior $P(C_i) = \dfrac{\sum_{j=1}^{n} w_{ij}}{n}$

Note that the formulas are not too far from k-means. In fact, since in k-means a point is either assigned or not to a single cluster, we can assume that $w_{ij} = 1$ if $\mathbf{x}_j \in C_i$ and $w_{ij} = 0$ if $\mathbf{x}_j \notin C_i$. In such a case, the formula for the mean becomes exactly the centroid, while the covariance becomes the radius, and the prior the probability of drawing randomly a point from cluster C_i.

✔ Advantages

- Flexible and powerful probabilistic model.
- Captures overlapping clusters.

✘ Disadvantages

- Possibly converges to local minimum.
- $O(nkd)$ where d is number of iterations. Can be quite high. Runtime highly dependent on initial assignment and number of clusters.
- To find cluster assignment requires to find a threshold on the probability values.

Lecture 3 *February 14, 2023*

Density-based clustering

Instructor: *Davide Mottin*

In this lecture we introduce density-based clustering. Density-based departs from the rigid geometrical structure of representative-based clustering and defines clusters as *high-density regions separated by low density region*. Intuitively, *density* is the number of points in a certain region in proportion to the size of the region. Density-based clustering approaches differ from one another for the definition of density.

We consider two main density-based clustering algorithms:

- DBSCAN in which a cluster is defined by two parameters that regulates the size and minimum number of points in the cluster.
- DENCLUE in which the use of kernels allows for more flexible notions of density.

2.1 DBSCAN

DBSCAN [EKS+96] is a density-based clustering algorithm grounded into the intuition that a point in a cluster should be *density reachable* from any other point in that cluster. This means that there should be "enough" points around each point in a cluster. As you can notice, this *modus operandi* does not include any specific on the shape of the cluster, but only constrains the neighborhood of a point. As such, DBSCAN can handle clusters with arbitrary shape as Figure 2.1 shows.

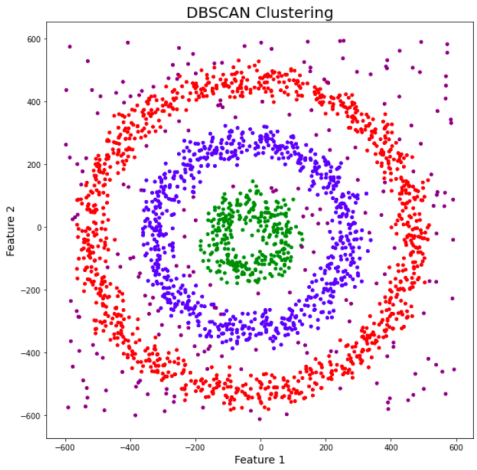


Figure 2.1: DBSCAN can detect clusters with arbitrary shape.

Definition 2.1.1. *For each point* $\mathbf{x}$*, we define the* neighborhood *the set of points that are at distance at most* ε *from* $\mathbf{x}$*, formally* $N_\varepsilon(\mathbf{x}) = \{\mathbf{y} \in \mathcal{D} | dist(\mathbf{x}, \mathbf{y}) \leq \varepsilon\}$

We additionally say that a neighborhood is *dense* if it contains at least $MinPts$ points.

Definition 2.1.2. *A* core object *is a point* $\mathbf{x}$ *whose neighborhood contains at least* $MinPts$ *points, i.e.,* $|N_\varepsilon(\mathbf{x})| \geq MinPts$

In order to identify clusters, we extend the notion of density to neighbor points.

Definition 2.1.3. $\mathbf{y}$ *is* **directly** density-reachable *from* $\mathbf{x}$ *within* ε*,* $MinPts$ *if* $\mathbf{y} \in N_\varepsilon(\mathbf{x})$*, and* $\mathbf{x}$ *is a core-object.*

A point $\mathbf{p}$ is *density reachable* from $\mathbf{x}$ if $\mathbf{p}$ is directly density reachable from a point $\mathbf{y}$ in the neighborhood $N_\varepsilon(\mathbf{x})$. Density reachability is the transitive closure of directly density reachability. Note that the density reachability is **not symmetric**. In particular, since $\mathbf{p}$ might not have $MinPts$ in its neighborhood, it cannot "reach" $\mathbf{x}$ with a sequence of dense areas.

Definition 2.1.4. $\mathbf{x}$ *is* density-connected *to a point* $\mathbf{y}$ *if there is a point* $\mathbf{p}$ *such that both* $\mathbf{x}$ *and* $\mathbf{y}$ *are density-reachable from* $\mathbf{p}$.

Finally, a *density-based cluster* is a non-empty subset $S \subseteq \mathcal{D}$ of the data $\mathcal{D}$ that satisfies both

- **Maximality:** If a point $\mathbf{x} \in S$ and $\mathbf{y}$ is density-reachable from $\mathbf{x}$, then $\mathbf{y} \in S$.
- **Connectivity:** Each object in S is density-connected to all other objects.

A density-based clustering of a database $\mathcal{D}$ is a partition $\{C_1, \ldots, C_k; N\}$ such that $\bigcup_{i=1}^{k} C_i \cup N = \mathcal{D}$, $C_i \cap C_j = \emptyset, j > i, i, j \in [1, k]$, and $C_i \cap N = \emptyset$, where $C_1, \ldots, C_k$ are the density-based clusters. $N = \mathcal{D} \setminus \{C_1, \ldots, C_k\}$ is the noise cluster of objects that do not belong to any cluster.

Algorithm 3 DBSCAN

Input: dataset $\mathcal{D}$, ε, $MinPts$

```
1: for each x ∈ D do
2:     if x is not part of any cluster C then
3:         if x is a core object then
4:             C ← Compute the density-reachable points from x,
5:             Create a new cluster C
6:         else
7:             N ← N ∪ {x}
```

DBSCAN algorithm The main theoretical idea behind the DBSCAN algorithm is that each object in a density-based cluster C is density-reachable from *any* of its core objects and nothing else is reachable from the core objects. In this manner, the DBSCAN algorithm is complete and ensures the two properties of maximality and connectivity. The DBSCAN algorithm is a simple iteration around the objects in the database, starting from a random point and finding all reachable points.

Density-reachable objects are collected by performing successive ε-neighbourhood queries that find the points at distance ε from another point.

We now analyze the complexity of DBSCAN. When the dimensionality is high, DBSCAN runs in $\mathcal{O}(n^2)$. By using specialized indexes for spatial queries, e.g., R^* trees, DBSCAN can runs faster than CLARANS that has a runtime complexity of $\mathcal{O}(k^3 + nk)$.

	N_ε-query	DBSCAN
Without support (worst case)	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Tree based support (like R^*)	$\mathcal{O}(\log(n))$	$\mathcal{O}(n \log n)$
Direct access to the neighborhood	$\mathcal{O}(1)$	$\mathcal{O}(n)$

2.1.1 Tuning ε and $MinPts$

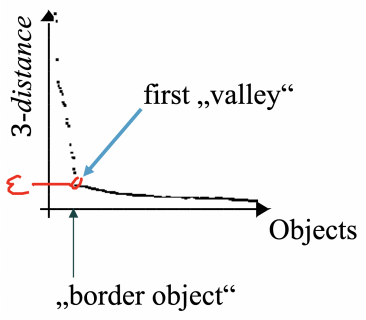
DBSCAN does not need the number of clusters to group points. However, before using DBSCAN, we need to decide on the paramaters ε and $MinPts$. This choice determines the shape and the number of clusters; a low ε or high $MinPts$ would find too many small clusters, a high ε might trivially terminate with a single cluster containing the entire dataset. However, it is not immediately clear what is the density of each of the clusters.

To this end, a number of heuristics have been proposed to find good settings for ε and $MinPts$. For example, one could use the point density of the least dense cluster to determine the parameters. One common heuristic uses the distance from k-nearest neighbours.

We need a couple of different concepts.

- $kdist(\mathbf{x})$ is the distance from $\mathbf{x}$ to its k-nearest neighbour
- $kdist$ plot is a chart of the k-distances of all points, sorted in decreasing order of distance.

The $kdist$ plot looks like the one below.



The heuristic works as follows:

- Fix a value for $MinPts$; if there is no domain knowledge available, a common choice is $2d-1$ where d is the dimension
- Select a "border object" $\mathbf{x}$ from the $MinPts$ distance plot, ε is set to $MinPtsdist(\mathbf{x})$.

✔ Advantages

- Does not require to specify number of clusters.
- Performs well with clusters of arbitrary shape.
- DBSCAN is robust to outliers and is able to handle them.

✘ Disadvantages

- It requires domain knowledge for $MinPts$ and it is not easy to determine ε.
- If clusters are very different in terms of in-cluster densities, then DBSCAN is not well suited to define clusters as it cannot generalise well to clusters with very different densities.

2.2 DENCLUE

DENCLUE [HK03] tackles density-based clustering in statistical manner. While DBSCAN defines clusters using two parameters that are the same across the dataset, DENCLUE detects dense regions without specifically specifying the size of the neighborhood. The main idea behind DENCLUE a generalization of neighbourhood density in terms of **density distribution**. The question is *how can we estimate (or measure) the density in different regions of the dataset?*

✍ Recap

Check the Zaki and Meira book [ZMJ20, Chapter 1] to refresh the memory on distributions and random variables.

2.2.1 Density estimation

Density estimation (DE) refers to a set of non-parametric models to estimate an unknown density function in a dataset. "Non-parametric" means that the model has no parameters to fit (as opposed, for instance, to a Gaussian distribution in which μ and σ are unknown for a data sample). In a few words, density estimation

- Determines an unknown probability density function
- It is non-parametric and does not assume a fixed probability model
- Estimates the probability density at each point

As it will appear more clear later, DBSCAN actually uses a simplified version of DE. Also, there is a strong connection between density-based clustering and DE.

We first look at univariate DE which assumes that the data is one-dimensional, i.e., $d = 1$, and then generalize to the multidimensional case.

2.2.1.1 Univariate density estimation

In one dimension, we can model the data as a random variable X. As such, our unidimensional data points can be treated as n samples or observations $\{x_1, \ldots, x_n\}$ from such a random variable.

We can estimate the cumulative density function (CDF) by counting the number of points less than a certain value v.

$$\hat{F}(v) = \frac{1}{n} \sum_{i=1}^{n} I(x_i \leq v) \tag{2.1}$$

where I is the indicator functions, which is 1 if the conditions inside the $(\cdot)$ is satisfied and 0 otherwise.

The density function is then estimated by taking the derivative of the CDF

$$\hat{f}(x) = \frac{\hat{F}(x + \frac{h}{2}) - \hat{F}(x - \frac{h}{2})}{h} = \frac{k/n}{h} = \frac{k}{nh} \tag{2.2}$$

where k is the number of points in a window of width h centered on x. The density estimate is ratio of points in the window (k/n) to the volume of the window h. Intuitively, our estimate is quite rough and requires to fix a bin size h. This is what happens when we compute *histograms* to visualize data points; it seems quite brutal.

Kernel density estimation We have seen that histograms could provide a first, rough estimate of the density in various parts of the dataset. However, we need to be careful on the window (or bin) size h. We now turn our attention to a more robust estimate, which uses special functions called Kernels.

> ⚠ Remark
>
> The name kernel is a bit unfortunate as it is not to be confused with the kernel used in k-means. Rather, these kernel should be considered as a weight for each individual point. The kernel defines how the weight is calculated based on each point's neighborhood.

We define a *kernel function* K that is a (localized) probability density function

- non-negative $K(x) \geq 0$
- symmetric $K(-x) = K(x)$
- integrates to 1, i.e., $\int K(x)dx = 1$

The discrete kernel is the simplest kernel and it corresponds to a degenerate uniform probability density function

$$K(z) = \begin{cases} 1 & \text{if } |z| \leq \frac{1}{2} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

when $|z| = \frac{x - x_i}{h} \leq \frac{1}{2}$ the point x_i is inside a window of size h centered at point x. We can then rewrite the density estimator in Eq. 2.2 as

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right) \tag{2.4}$$

Note that the kernel now appears explicitly in the formula and becomes a choice that determines the quality of the estimator. The discrete kernel is not smooth, though, that means that the kernel estimate looks again like a histogram with the consequence that many useful calculus operators (such as derivatives) cannot be comfortably performed.

Gaussian kernel We introduce a different kernel also used in DENCLUE. The Gaussian kernel is one of the most popular kernels and represents a Gaussian (or normal) distribution centered around a point. The Guassian kernel is defined as follows.

$$K(z) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z^2}{2}\right)$$

As usual, by substituting $z = \frac{x-x_i}{h}$ we obtain

$$K\left(\frac{x-x_i}{h}\right) = \frac{1}{\sqrt{2\pi}} exp\left(-\frac{(x-x_i)^2}{2h^2}\right) \tag{2.5}$$

where x at the center of the window plays the role of the mean, and the window size h becomes standard deviation.

2.2.1.2 Multivariate density estimation

For d-dimensional data $\mathbf{x} = (x_1, ..., x_d)$, the window h becomes a hypercube centered at x with volume h, that is $vol(H_d(h)) = h^d$. Density estimation is the same as above, except that the density is now divided by the volume of the hypercube.

$$\hat{f}(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{x-x_i}{h}\right)$$

where the multivariate kernel must preserve non-negativity, symmetry, and integrate to 1. It is then straightforward to redefine the discrete and Gaussian kernels.

Discrete multivariate kernel Similarly to its univariate counterpart, the multivariate kernel is as follows

$$K(\mathbf{z}) = \begin{cases} 1 & \text{if } |\mathbf{z}| \leq \frac{1}{2} \text{ for all dimensions} \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

The Gaussian kernel uses a equi-variate covariance matrix which translates into using the identity matrix $\Sigma = \mathbf{I}$.

$$K(\mathbf{z}) = \frac{1}{(2\pi)^{\frac{d}{2}}} \exp\left(-\frac{\mathbf{z}^\top \mathbf{z}}{2}\right) \tag{2.7}$$

2.2.1.3 Nearest neighbor estimation

Another way to estimate density builds on the k-nearest neighbors. Intuitively, instead of fixing the size h of the region, the nearest neighbor estimate fixes the number of points k in the neighborhood of a point $\mathbf{x}$ and finds the size of the region.

k-nearest neighbour proceeds as follows

1. Fix k, the minimum number of points that defines a dense region
2. Compute the volume $vol(S_d(h_\mathbf{x}))$ as a function of k

Given k, we estimate density at $\mathbf{x}$ as

$$\hat{f}(\mathbf{x}) = \frac{k}{n \cdot vol(S_d(h_x))} \tag{2.8}$$

where $h_\mathbf{x}$ is the distance from $\mathbf{x}$ to its k-nearest neighbour. The volume is the volume of the d-dimensional hypersphere centered at $\mathbf{x}$.

2.2.2 Density attractors

We now describe another important concept before explaining the DENCLUE algorithm. Assume you have estimated the density using one of the density estimators above. *How can we find dense clusters?* Again intuitively, one can look at the density function and set a threshold ξ, i.e., a horizonal line in a density-points histogram. However, if we do so, some of the points in regions with low-density might be not be assigned to any cluster.

To this end, we introduce density attractors that are points with maximal density.

Definition 2.2.1. *A point* $\mathbf{x}^*$ *is a* density attractor *if it is a local maxima of the probability density function* f

In order to discover these "peaks" in the density function we need to compute its gradient.

$$\nabla \hat{f}(\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}} \hat{f}(\mathbf{x}) = \frac{1}{nh^d} \sum_{i=1}^{n} \frac{\partial}{\partial \mathbf{x}} K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right) \tag{2.9}$$

For the Gaussian kernel, the gradient takes a particularly neat form (see the book for the derivation)

$$\frac{\partial}{\partial \mathbf{x}} K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right) = K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right) \cdot \left(\frac{\mathbf{x}_i - \mathbf{x}}{h}\right) \cdot \left(\frac{1}{h}\right)$$

which, substituted in Eq. 2.9 becomes

$$\nabla \hat{f}(\mathbf{x}) = \frac{1}{nh^{d+2}} \sum_{i=1}^{n} \underbrace{K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right)}_{\substack{\text{Weight relative} \\ \text{to } \mathbf{x}}} \cdot \underbrace{(\mathbf{x}_i - \mathbf{x})}_{\substack{\text{Difference} \\ \text{vector}}} \tag{2.10}$$

We can now reason on the equation above by considering its various parts. In particular, the gradient provides an overall weighted deviation from the point $\mathbf{x}$. Therefore, we can "traverse" the density function and discover where it forms peaks, i.e., where the density is locally maximized. The points that maximizes the density are called density attractors.

We can now mathematically define what a density-based cluster is.

Definition 2.2.2 (Density-based cluster)**.** *A set of* C *of data points from a data set* $\mathcal{D}$ *is a density-based cluster w.r.t to some threshold* ξ *if there exists a set of attractors* $\mathbf{x}_1^*, \ldots, \mathbf{x}_k^*$ *if*

- *Each point* $\mathbf{x}$ *in* C *is attracted to some attractor* $\mathbf{x}_i^*$*.*
- *Each density attractor* $\mathbf{x}_i^*$ *exceeds some density threshold* ξ*,* i.e., $\hat{f}(\mathbf{x}_i^*) \geq \xi$*.*
- *Any two density attractors* $\mathbf{x}_i^*, \mathbf{x}_j^*$ *are density reachable. That is, there exists a path from* $\mathbf{x}_i^*$ *to* $\mathbf{x}_j^*$ *such that for all points* $\mathbf{y}$ *on the path,* $\hat{f}(\mathbf{y}) \geq \xi$

Note that from the above definition, it is clear that not all points will be part of a cluster. As such, we can treat those points as noise.

The DENCLUE algorithm works as follow. For each point in the data, it finds a candidate attractor. If the attractor's density is above the threshold ξ, it is added to the set A of attractors and the point is added to the set R of points attracted by $\mathbf{x}^*$. Then all the maximal sets of mutually density reachable attractors computed.

The FindAttractor function iteratively computes the attractors. As per definition, an attractor is a point of maximum density. To find an attractor for a point $\mathbf{x}$ we can solve $\nabla \hat{f}(\mathbf{x}) = 0$. This gives us

$$\begin{aligned} \frac{1}{nh^{d+2}} \sum_{i=1}^{n} K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right) \cdot (\mathbf{x}_i - \mathbf{x}) &= 0 \\ \mathbf{x} &= \frac{\sum_{i=1}^{n} K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right) \mathbf{x}_i}{\sum_{i=1}^{n} K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right)} \end{aligned} \tag{2.11}$$

Since $\mathbf{x}$ appears on both sides, we can use it as an iterative rule until convergence. This gives us the following algorithm. The above algorithm runs in $\mathcal{O}(n^2 T)$ where T is the number of iterations.

Algorithm 4 DENCLUE

Input: Dataset $\mathcal{D}$, window h, density threshold ξ

$A \leftarrow \emptyset$
for each $\mathbf{x} \in D$ **do**
  // Compute the attractor for point x
  $\mathbf{x}^* = \mathsf{FindAttractor}(\mathbf{x})$
  // Filter attractors by density threshold
  if $\hat{f}(x^*) \geq \xi$ **then**
    $A \leftarrow A \cup \{\mathbf{x}^*\}$
    $R(\mathbf{x}^*) \leftarrow R(\mathbf{x}^*) \cup \{\mathbf{x}\}$
$\mathcal{C} = \{\text{maximal } C \subseteq A | \forall \mathbf{x}_i^*, \mathbf{x}_j^* \in C, \mathbf{x}_i^*, \mathbf{x}_j^* \text{ density reachable}\}$
for each $C \in \mathcal{C}$ **do**
  for each $\mathbf{x}^* \in C$ **do**
    $C \leftarrow C \cup R(\mathbf{x}^*)$
Return $\mathcal{C}$

Algorithm 5 FindAttractors

$\mathbf{x}_0 \leftarrow \mathbf{x}$
repeat
  $\mathbf{x}_{t+1} \leftarrow \frac{\sum_{i=1}^{n} K(\frac{\mathbf{x}_t - \mathbf{x}_i}{h})\mathbf{x}_t}{\sum_{i=1}^{n} K(\frac{\mathbf{x}_t - \mathbf{x}_i}{h})}$
until $\|\mathbf{x}_t - \mathbf{x}_{t-1}\| \leq \varepsilon$
return $\mathbf{x}_t$

DENCLUE and DBSCAN DENCLUE and DBSCAN are formally different, but are they not related? The short answer is, no. If thought carefully, DBSCAN corresponds to a discrete kernel, which is more efficient but also more rough. The two paramaters $\varepsilon, MinPts$ of DBSCAN are actually the h, ξ parameters of DENCLUE. So, in principle the two algorithms share uncanny similarities. DENCLUE uses Guassian kernel density based attractors, which is a smooth model and able to capture subtlities of the underlying data distribution that DBScan disregards.

✔ Advantages

- Clusters can arbitrary shape and size, clusters are not restricted to have convex shapes.
- Number of clusters is determined automatically.
- Can separate clusters from surrounding noise.
- Can be supported by spatial index structures.

✘ Disadvantages

- Input parameters may be difficult to determine.
- Can be sensitive to input parameter setting.

Lecture 4 *February 21, 2023*

Hierarchical clustering

Instructor: *Davide Mottin*

In this lecture we introduce techniques to deal mostly with high-dimensional data. When the data has high dimensions or clusters have different densities, methods like DBSCAN and k-means might fail. We will not go too deep in each technique, but rather give an overview of the main techniques. In general, this week's topic build nicely on the concepts of previous lectures and does not require any specific concept. We look at two different strategies and several algorithms:

- **Hierarchical clustering** partitions the space in progressively smaller clusters
  - Hierarchical agglomerative clustering: based on previous algorithms we study *single-link, complete-link, and average-link* strategies to merge clusters and form larger clusters
  - OPTICS clustering that order density reachable points to find cluster structures
  - BIRCH that summarizes the data to understand similarities
- **Subspace clustering** finds different clusters in different subsets of the dimensions in which the data lies
  - CLIQUE: A clustering algorithm which divides the space into grids and exploit the apriori principle on density to compute dense subpaces
  - SUBCLU: Similar to CLIQUE but uses DBScan
  - PROCLUS: A projected clustering which finds the best dimensions for each clusters

3.1 Agglomerative hierachical clustering

One of the main visualization tools for hierarchical clustering is the *dendrogram*. A dendrogram is a tree, in which the root is a cluster containing all the points in the dataset $\mathcal{D}$ and the leaves are clusters of single points. A dendrogram can be constructed bottom-up (agglomerative) by progressively merge smaller clusters into large one or top-down (divisive) by splitting larger clusters. The height of a branch in the dendrogram show the distance. A cut-off value as a vertical line identifies a partition of the data points. Figure 3.1 shows an example of a dendrogram.

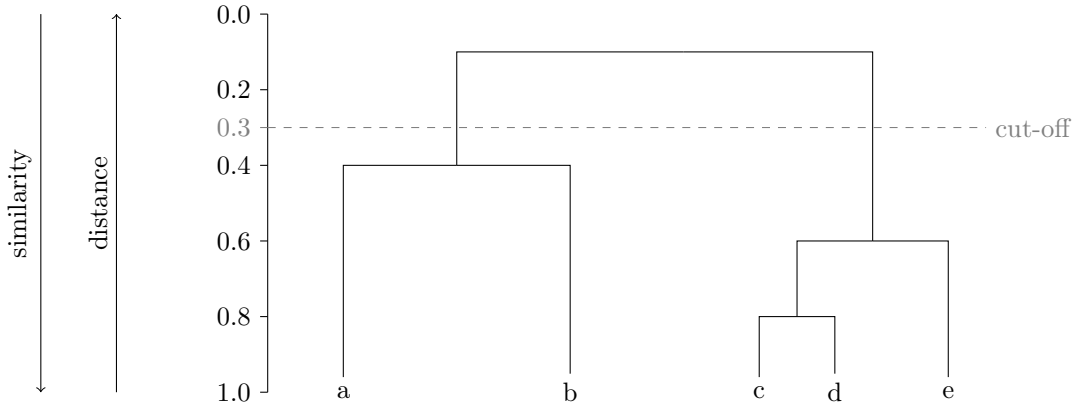


Figure 3.1: A dendrogram for 5 points {a,b,c,d,e}.

Agglomerative hierarchical clustering takes a bottom-up approach; this means that the clusters are progressively merged to form larger clusters. An agglomerative hierarchical clustering requires the choice of a distance function *dist*. We have seen different possibilities in the previous chapters.

Algorithm 6 shows the skeleton for any agglomerative hierachical clustering algorithm.

Algorithm 6 Agglomerative hierarchical clustering

Input: dataset $\mathcal{D}$, clustering distance *dist*

```
// Create a cluster for each point
```

$t \leftarrow 1$

$\mathcal{C}^t \leftarrow \{\{\mathbf{x}\}_{\mathbf{x}\in\mathcal{D}}\}$

while $\mathcal{C}^t \neq \mathcal{D}$ **do**

  for each $C_i, C_j \in \mathcal{C}^t$ **do**

    Compute $dist(C_i, C_j)$

```
// Merge the clusters with the minimum distance
```

  $C_i, C_j \leftarrow \arg\min_{C_i,C_j\in\mathcal{C}^t} dist(C_i, C_j)$

```
// Create a new level in the dendrogram
```

  $\mathcal{C}^{t+1} \leftarrow (\mathcal{C}^t \setminus C_i) \setminus C_j \cup (C_i \cup C_j)$

  $t \leftarrow t + 1$

3.1.1 Distance among clusters

The algorithm above assumes that we are able to compute the distance among two clusters. We now describe three simple strategies to compute such a distance. The choice of the strategy determines the quality of the clustering.

- Single-link distance $dist_{sl}(C_i, C_j) = \min_{\mathbf{x}\in C_i,\mathbf{y}\in C_j} dist(\mathbf{x}, \mathbf{y})$: compute the minimum distance among two clusters
- Complete-link $dist_{cl}(C_i, C_j) = \max_{\mathbf{x}\in C_i,\mathbf{y}\in C_j} dist(\mathbf{x}, \mathbf{y})$: compute the maximum distance among two clusters
- Average link $dist_{al}(C_i, C_j) = \frac{1}{|C_i||C_j|} dist(\mathbf{x}, \mathbf{y})$: compute the average distance among two clusters

3.2 OPTICS

✍ Recap

Check the DBSCAN algorithm in 2.1 in these lecture notes or the book.

We observed that DBSCAN performs poorly when the clusters have different densities. OPTICS [ABKS99], that stands for Ordering Points To Identify the Clustering Structure, solves this problem with a simple method. It is easy to observe that for a fix value of ε in DBSCAN a point $\mathbf{x} \in \mathcal{D}$ might have a very large ε-neighborhood $N_\varepsilon(\mathbf{x})$. This observation suggests the possibility to define the minimum distance under which a point is a core point. This distance is called the core distance *cdist*.

Definition 3.2.1. *The* core-distance *cdist is the smallest distance such that* $\mathbf{x}$ *is a core point*

$$cdist(\mathbf{x}) = \begin{cases} MinPts\text{-}th\ smallest\ distance\ dist(\mathbf{x}, \mathbf{y}) & if\ |N_\varepsilon(\mathbf{x})| \geq MinPts \\ ? & otherwise \end{cases} \tag{3.1}$$

Similarly, we define the *reachability distance* as the distance of the closest point reachable by $\mathbf{x}$

Definition 3.2.2.

$$rdist(\mathbf{x}, \mathbf{y}) = \begin{cases} dist(\mathbf{x}, \mathbf{y}) & if\ dist(\mathbf{x}, \mathbf{y}) > cdist(\mathbf{y}) \\ cdist(\mathbf{y}) & if\ dist(\mathbf{x}, \mathbf{y}) < cdist(\mathbf{y}) \\ ? & if\ dist(\mathbf{x}, \mathbf{y}) > \varepsilon \end{cases} \tag{3.2}$$

An illustration of the reachability distance and core distance is in Figure 3.2.

OPTICS is an extension of DBSCANthat builds upon the idea of core-distance *cdist* and reachability-distance *rdist*. The principal idea is to order the points based on reachability-distance. OPTICS main ingredients are the following.

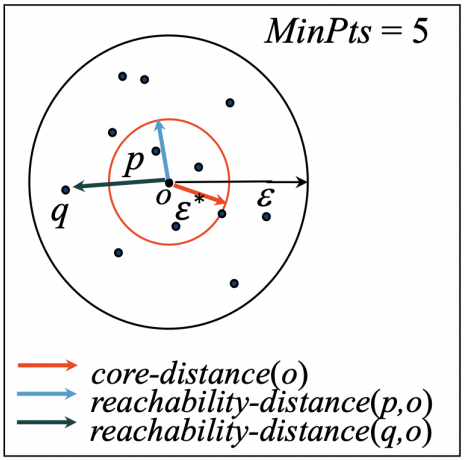


Figure 3.2: Reachability and core distance of a point o

- OPTICS maintains a basic data structure (control list) that stores the shortest reachability-distance seen so far (distance of a jump to that point)
- OPTICS visits each point once and moves to the next point based on the smallest reachability-distance
- OPTICS outputs the order of points, the core and the reachability distance of all points in a *reachability plot*

The OPTICS algorithm is as follows.

Algorithm 7 OPTICS

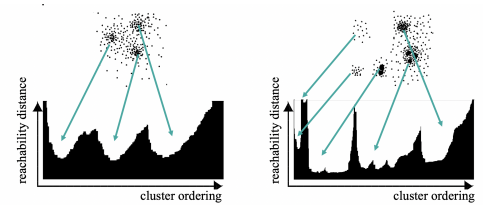
```
1:  for each x ∈ 𝒟 do
2:      if x is not processed then
3:          insert (x, "?") intro CL
4:      while CL ≠ ∅ do
5:          select first element (x, rdist) ∈ CL
6:          Retrieve N_ε(x) and
7:          cdist ← cdist(x)
8:          x.processed ← True
9:          Write (x, rdist, cdist) to file
10:         if x is a core object at distance dist ≤ ε then
11:             for each p ∈ N_ε(x) do
12:                 if p.processed = false then
13:                     rdist_p = rdist(p, x)
14:                     if (p, _) ∉ CL then
15:                         Insert (p, rdist_p) into CL
16:                     else if (p, old_rdist) ∈ CL and rdist_p < old_rdist then
17:                         update (p, rdist_p)inCL
```

Finding a density based clustering with respect to $\varepsilon^* \leq \varepsilon$ and $MinPts$ is easy:

- Start with $\mathbf{x}$ with $cdist(\mathbf{x}) \leq \varepsilon^*$ and $rdist(\mathbf{x}, \_) > \varepsilon^*$
- Continue while $rdist(\mathbf{x}, \_) \leq \varepsilon^*$

Running time. OPTICS running time is bounded by the running time of DBSCAN, i.e., $\Omega(n^{4/3})$. OPTICS only maintains additional data structures, that do not hinder the overall time complexity.

Analysis of reachability plot. The reachability plot shows both the reachability and the core distance. However, in some cases it mighe be sufficient to show only the reachability distance. Contiguous areas in the plot with large reachability and core-distance indicate regions with low density. At the same time low reachability-distance indicates high-density areas. A jump in the plot occurs when the next point in order falls outside the current cluster..



OPTICS is relatively insensitive to parameter settings, and results are good if the parameters are large enough.

✔ Advantages

- It does not require the number of clusters to be known in advance
- Very roboust optics (OPTICS)
- Computes complete hierarchy
- Good visualizations
- A flat partition can be derived, such as cutting through the dendrogram or reachability plot

✘ Disadvantages

- No backtracking, greedy splits and merges
- May not scale well. It examines many objects in order to split and merge. Runtime for standard methods is $O(n \log n)$, and for OPTICS without indexing it is $O(n^2)$

3.3 BIRCH

BIRCH [ZRL96] stands for **B**alanced **I**terative **R**educing and **C**lustering using **H**ierarchies. BIRCH first summarizes the data and then clusters the summarized data with any clustering algorithm. BIRCH can cluster large datasets that do not fit into memory with only a negligible loss in accuracy. Moreover, the BIRCH algorithm is also an incremental algorithm for accommodating new data points.

The BIRCH algorithm has two main phases.

- **Phase 1**: Scan the dataset $\mathcal{D}$ to build an in-memory CF (Clustering Feature) tree. The CF-tree is a multi-level compression that preserves inherent clustering structure
- **Phase 2**: Use an arbitrary clustering algorithm to cluster the leaf nodes of the CF tree.

⚠ Remark

With Phase 1, BIRCH summarizes individual data points. Point that end up having the same summary are likely to be very close and not adding any extra information in the clustering.

BIRCH scales linearly $\mathcal{O}(n)$ in the size of the dataset. A single scans over the data attains already good cluster. However, to improve the quality, BIRCH can optionally scan the data multiple times. BIRCH is a fast algorithm but can only deal with numerical data.

The basic idea of BIRCH is as follows:

- Constructs a partitioning of the data into micro-clusters using an efficient index-like structure
- The micro-clusters are representations of multiple points described by Clustering Features (CFs).

- CFs are organised hierarchically in a balanced tree
- A standard clustering algorithm is then applied to the leaf entires of the CF-tree

3.3.1 Clustering features (CFs)

A clustering feature $CF = (N, LS, SS)$ of a micro-cluster $C \subseteq \mathcal{D}$ consists of

- $N = |C|$: Number of points in the micro-cluster
- $LS = \sum_{\mathbf{x} \in C} \mathbf{x}$: Linear sum of the data points in C
- $SS = \sum_{\mathbf{x} \in C} \mathbf{x}_i^2$: Square sum of the data points in C

Note that both LS and SS are vectors. The CF representation, although quite simple, is enough to compute a number of statistics about a cluster, such as centroids, measures of compactness, and distance measures for clusters.

Additivity theorem An important property of the CFs is the **additivity theorem** that allows CFs to be computed incrementally on unions of micro-clusters. If two clusters are merged the CF of the new cluster is simply the element-wise sum of the respective CFs.

3.3.2 CF-tree

The CF-tree is an index similar to a B+-tree that contains the CF representations of the micro-cluster. The tree-structure can be easily updated without scanning the entire data, by virtue of a nice data organization.

A CF tree with parameters B, L, T is a tree-like structure, such that

- An internal node contains at most B entries $[CF_i, child_i]$
- A leaf node contains at most L entries $[CF_i]$
- The diameter or radius of all entries in a leaf node is at most T according to a user-defined distance metric
- Leaf nodes are connected via prev and next pointers

The tree is constructed bottom-up in the following manner:

- Transform point $\mathbf{x}$ into CF vector $CF_{\mathbf{x}} = (1, \mathbf{x}, \mathbf{x}^2)$
- Insert $\mathbf{x}$ in the same way as in a B+-tree
- If threshold T is violated by insertion, then split the leaf node and reorganize tree analog to that of B+-trees.

3.3.3 The BIRCH algorithm

After introducing the necessary elements we can summarize the BIRCH algorithm as follows. The algorithm has two optional phases that can be skipped but are typically performed to further reduce the time and improve the quality.

- Phase 1: Scan all data points and build in memory CF tree using given amount of memory and recycling space
- Phase 2 (Optional): Condense into desirable length by building a smaller CF tree
- Phase 3: Global clustering (with any clustering algorithm) on the CF feature vectors
- Phase 4 (Optional): refine with more passes

✔ Advantages

- Compression factor can be tuned to the available memory
- Efficient construction of a microclustering $O(n)$
- Good clustering result for the partitioning iterative refine clustering algorithm algorithms such as k-means and k-medoid when applied to only the leaf nodes of a CF tree.

✘ Disadvantages

- Only for data from a Euclidean vector space (linear sum, square sum, mean etc must be defined)
- Handles only numeric data
- Sensitive to the order of data records
- Entries are limited by disk page size
- Different parameters to tune

3.4 Subspace clustering

One of the main problems with high-dimensional data is the *curse of dimensionality* is the inherent dispersion of the points and the difficulty in comparing distances.

⚠ Remark

Curse of dimensionality. The course of dimensionality is a general phenomenon that occurs on points as the number of dimensions increases. Two important effects of the increase in dimensionality are

- *Sparsification*: The data points grow further apart with the dimension
- *Incomparability*: The probability that points fall into a sphere of a certain radius becomes negligible, thus the distances of points from a center or another points become closer with the dimension.

One example of curse of dimensionality is Figure 3.3.

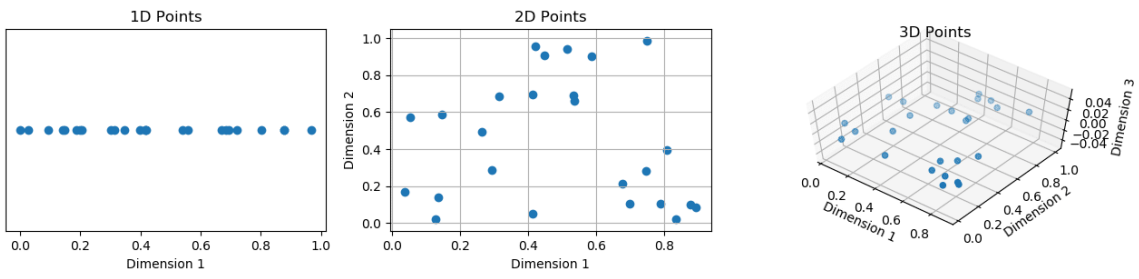


Figure 3.3: Points in high-dimension grow further apart, while distances become meaningless.

As a result of the curse of dimensionality, a cluster in low dimension might not appear as such in higher dimension. Does it mean that the cluster does not exist? *Subspace clustering* attempts to tackle this problem by clustering the points in different subsets of the dimensions. Intuitively, in 3D it is like projecting the points on some of the coordinates. The selection of the coordinates where to project is

the main difference between the subspace clustering algorithms. Another major different is the clustering algorithm employed in the subspaces.

3.4.1 CLIQUE

CLIQUE [AGGR98] is one of the first algorithms for subspace clustering. CLIQUE is a density based algorithm based on a simple idea: divide the space into n/h cubic regions, prune regions that have less than a certain number of points ξ (called, density threshold) in low dimension and repeat the process on subspaces obtained by aggregating dimensions that contains at least one cluster.

The naïve approach computes clustering for subsets of dimensions. However, this approach considers all $\mathcal{O}(2^d)$ clusters, which is prohibitive.

CLIQUE exploits the *monotonicity of density* for which a dense cluster in dimension d is dense in every subset of the d dimensions. A cluster is the union of dense regions. Therefore, we can construct a **Greedy algorithm** based on pruning regions that are not dense. The idea is to start with an initial empty set of dimensions and iteratively include dimensions that are not pruned in the previous step.

⚠ Remark

It is useful to consider the parallel between CLIQUE and the Apriori algorithm in frequent itemsets in Lesson 12.

The greedy algorithm in CLIQUE first finds 1-dimensional R_1 candidate regions, then generate 2-dimensional candidates R_2 from the 1-dimensional candidates, prune the candidates using the monotonicity of density, and repeat the process increasing the dimension until no candidate is generated.

Algorithm 8 CLIQUE

Input: dataset $\mathcal{D}$, region size h, density threshold ξ

$R_1 \leftarrow$ Candidate regions in 1-dimension
$d \leftarrow 2$
repeat
    $R_d \leftarrow$ Generate all candidate d-dimensional cells from R_{d-1}
    `// Prune cells with fewer than ξ points`
    $R_d \leftarrow \{R : |R| \geq \xi\}$
    $d \leftarrow d + 1$
until $R_d \neq \emptyset$
$\mathcal{C} \leftarrow$ Compute clusters from each R_i
Summarize clusters

In the last two steps, CLIQUE computes the clusters from the dense regions and summarizes them.

Compute clusters from the union of adjacent candidate regions. Compute a graph for each candidate susbpace in dimension d. The graph has a node for each dense region and an edge if two regions are adjacent. The connected components of the graph represent the clusters. This steps takes $\mathcal{O}(2dn)$ for each candidate subspace.

Summarizes cluster with a minimal number of inequalities, by defining boundaries on the regions. Since each region is represented by a number of dimensions, its boundaries are values of the dimensions that describe the region's hypercube.

✔ Advantages

- Automatic detection of subspaces with clusters
- Automatic detection of clusters
- No assumptions about the distribution of data
- Insensitivity to the order of the data
- Good scalability w.r.t to the number n of data points

✘ Disadvantages

- Accuracy of result depends on the number of partitions $\frac{h}{n}$
- Requires heuristics to avoid constructing all subspaces.

3.4.2 SUBCLU

CLIQUE might miss clusters due an early pruning of the regions that cross the clusters, such as in the example in Figure 3.4.

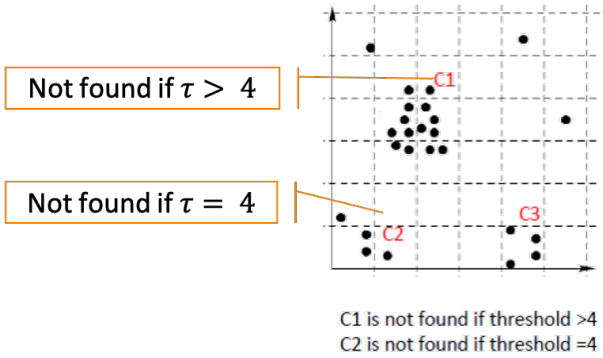


Figure 3.4: CLIQUE does not detect clusters C_1, C_2, C_3 due to an early pruning of their regions.

SUBCLU (Density-connected subspace clustering) [KKK04] remedies to CLIQUE's deficiency by using DBSCAN as the basis to find dense clusters in a subspace. Without entering into details the definition of core-objects, density reachability and connectivity can easily adapt to subspaces. Similarly, SUBCLU exploits the *monotonicity of density connectivity*:

Property 1 (Monotonicity of density connectivity)**.** *If* $\mathbf{x}$ *and* $\mathbf{y}$ *are density-connected in* d*-dimensional space, all their projections in* $k-1$ *are also density-connected.*

The algorithm iteratively applies DBSCAN at each step to generate d-dimensional clusters as in CLIQUE.

✔ Advantages

- Automatic detection of subspaces containing clusters
- Automatic detection of clusters
- No assumptions on data distributions (density-based approach)

✘ Disadvantages

- Parameter settings highly affect clustering results
- Inherits the drawback of DBSCAN, such as the sensitivity w.r.t. to clusters with different densities.
- At least $\mathcal{O}(n^2)$ time. Typically worse depending on max-dimensionality of the subspace containing clusters.

3.5 Projected clustering

Projected clustering tackles the curse of dimensionality differently from subspace clustering. The main idea is to project each point individually into the **only** subspace in which the point is clustered. This intuitive idea requires a sophisticated notion of cluster significance. Indeed, while projecting everything to

a low-dimensional plane might reveal more clusters, we might instead collapse clusters together and render the clustering meaningless. The goal of projected clustering is correctly identify the right projections, such that each point is assigned exactly to one cluster. Projected clustering avoids rediscovering the clusters in multiple subspaces but at the same time, might miss clusters.

3.5.1 PROCLUS

PROCLUS [AHWY04] is a representative methods for projected clustering. In simple terms, PROCLUS is a top-down approach that iteratively refines clusters and projections starting from an initial clustering of the data in the full-dimension. PROCLUS uses k-medoid algorithm to find clusters.

> ✍ Recap
>
> Check the k-medoids algorithm in 1.3.

We first need to define a *projected cluster*.

Definition 1 (Projected cluster)**.** *A* projected cluster *is a set of objects C and a set of dimensions D, such that the objects in C are closely clustered in the subspace defined D.*

Definition 2. *A cluster C is closely clustered if it is compact under the Manhattan segmential distance*

$$d(\mathbf{x}, \mathbf{y}) = \frac{1}{|D_C|} \sum_{i \in D_C} |x_i - y_i| \tag{3.3}$$

that measures the Manhattan distance of the objects in the projected dimensions D of cluster C

PROCLUS requires the specification of the number k of clusters and the the average number $\hat{d}$ of dimensions. PROCLUS follows a multistep approach in three phases

1. **Initialization** selects the initial set of medoids.
2. **Iterative** refines the medoids and computes the projections
3. **Refinement** improves the quality of the clustering.

We now describe each phase separately.

Initialization. The initialization phase selects a sample of representative data points from the entire dataset, where the sample should ideally (1) characterize the entire dataset and (2) should contain at least k medoids. In practice, since finding such an ideal a sample is hard, PROCLUS greedily selects $A \cdot k$ well-separated points.

Iterative. The iterative phase is the main body of the PROCLUS algorithm that determines the best set of dimensions for a set of candidate medoids.

The iterative phase goes as follows:

- **Choose** random set of k-medoids from representative points
- **Determine** optimal set of dimensions for each medoid
- **Assign** all objects to the nearest medoid
- **Update** current clustering if it is better than the previous until termination

The iteration continues from step 2. by replacing bad medoids with random medoids from the sample of representatives until the clusters do not change anymore.

The first issue in the iterative phase is how to determine an optimal set of dimensions for each medoid. To this end, PROCLUS proposes a score Z_{ij} that measures the reduction in dispersion of the cluster C_i on dimension j as opposed to the full dimensionality d.

For each medoid $\mathbf{m}_i$, calculate the average distance X_{ij} on dimension j of objects in the hyper-sphere $\mathcal{L}_i$ with radius the distance from $\mathbf{m}_i$ to its nearest medoid (i.e., $\min_{t \neq i} d(\mathbf{m}_t, \mathbf{m}_i)$). The dispersion of cluster i is the average distance X_{ij} for all the dimensions

$$Y_i = \frac{1}{d} \sum_{j=1}^{d} X_{ij} \tag{3.4}$$

The deviation $X_{ij} - Y_i$ is negative if the clusters on dimension j are correlated to those in the full dimension.

The score Z_{ij} is the normalized deviation $X_{ij} - Y_i$

$$Z_{ij} = \frac{1}{\sigma_i}(X_{ij} - Y_i) \tag{3.5}$$

where σ_i is the empirical standard deviation of X in cluster i

$$\sigma_i = \sqrt{\frac{1}{d-1} \sum_{j=1}^{d} (X_{ij} - Y_i)^2} \tag{3.6}$$

Small values indicate better dimensions to find clusters. The score Z allows to assign each cluster to the best dimensions. PROCLUS assigns greedily each cluster to $k(\hat{d} - 1)$ dimensions and computes the set of projected dimensions D_i for cluster i.

After determining the set of dimensions $D_1, ..., D_k$ for each medoid $\mathbf{m}_1, ..., \mathbf{m}_k$, PROCLUS assigns the points to the nearest medoid using the Manhattan segmental distance in Equation 3.3.

Finally, the algorithm updates the current set of medoids by computing the spread around the **centroids** in each dimension. A measure of dispersion similar to Z_{ij} is used to refine the clusters.

⚠ Remark

Note that in the update step, the algorithm uses Euclidean distance.

By substituting uninformative clusters with a random representative from the initial sample. A medoid is substituted if the cluster it represents has the smallest number of points among the rest or if the number of points is below a threshold.

Refinement. The refinement phase improves the clustering. First, it determines the optimal dimension for each medoid, using a procedure similar to that of the iterative phase, but substituting $\mathcal{L}_i$ with the clusters C_i. Second, each point is assigned to the closest cluster C_i. A point is an outlier if its segmental distance to each medoid exceeds the radius of the medoids sphere of influence.

✔ Advantages

- Based on full high-dimensional space
- Simple but efficient projected clustering

✘ Disadvantages

- Relies on cluster-based locality assumption: subspace of each cluster is learned from local neighborhood of its medoid
- Forces projected clusters to have convex shape
- Medoids are chosen only from initial sample set
- Heighly heuristic and is sensitive a large number of input parameters $(k, l, A, mindeviation)$

Lecture 5 *February 28, 2023*

Outlier detection

Instructor: *Davide Mottin*

This lecture introduces outlier detection as the task of finding anomalous points in data. The definition of outlier typically depends on the application, however, Hawkins [Haw80] defines an outlier as

> "An object that deviates so much from the rest of the data as to arouse suspicion that it was generated by a different mechanism"

The intuitive definition that Hawkins provide sets the boundaries between *noise* and *outliers.* Noise is a measuring error, which, itself can have its own distribution. On the other hand, an outlier is a point that belongs to the same distribution from which the is sampled but is considered as a rare event.

We distinguish different types of outliers:

- **Global outliers:** Points that significantly deviate from the data set.
- **Contextual outliers:** Points that deviate from a subset of points. For instance, global temperature might be within normal measurements, however, in one city there could be a suspicious spike for one day. Context is defined as fixed values in some dimensions.
- **Collective outliers:** A set of points might represent an outlier with respect to the rest of the points. In this case, the single point might not be an outlier, though such is the whole set.

Outlier detection methods can be supervised or unsupervised. Supervised outlier detection requires points labeled as normal and outliers. These methods typically deal with a large class imbalance, since outliers are small percentage (often $< 1\%$) of the data. However, in the data mining context we are interested in unsupervised outlier detection.

This lecture ventures into different definitions of outliers as well as unsrupervised algorithms to detect them:

- **Low-dimensional**: Approaches that work only withing few dimensions. They are particularly useful due to their easy interpretability:
  - **Model-based**: Assume a known distribution and see if the points falls in a low-probability area
  - **Depth-based**: Enclose the points into convex hulls to identify those points in the border
  - **Distance-based**: Define whether a point is "far enough" from all the others.
- **High-dimensional**: Approaches for high-dimensional data that work well in low-dimension as well:
  - **Angle-based**: Analyze the angles to detect whether a point
  - **Local outliers**: Compare the density of the locality of a point as opposed to the density of the neighbors
  - **Clustering-based**: Use a clustering method and return the points that are further in clusters or classified as noise (e.g., placed in noise cluster like in DBSCAN).
  - **Isolation forests**: Random partition of the space lead to earlier isolation of outliers from the rest

4.1 Low-dimensional outlier detection

4.1.1 Model-based methods

Model-based statistical methods assume that the data is a sample from a *known* distribution $P(X)$. After fitting the parameters θ of the distribution, return the points having low probability $P(\mathbf{x})$, i.e., points that are unlikely to be generated by the distribution $P(X)$.

The simplest model-based method is assuming the data $\mathcal{D}$ belongs to a (multivariate) normal distribution $\mathcal{D} \sim f(\mathbf{x}|\boldsymbol{\mu}, \Sigma) = \frac{1}{\sqrt{(2\pi)^d|\Sigma|}} e^{-\frac{1}{2}(x-\boldsymbol{\mu})^\top(\Sigma)^{-1}(\mathbf{x}-\boldsymbol{\mu})}$. A rule of thumb is considering a pointan outlier if it falls more than three times the standard devitaion from the mean in a given dimension.

The Grubbs' Test provides a statistical confidence whether the point is generated by the same normal distribution. However, such a test works only with univariate data. The test's null hypothesis H_0 is the absence of outliers in the data. The test works as follows.

1. Compute the Grubbs' test statistics $G = \frac{\max_{\mathbf{x}\in\mathcal{D}} |\mathbf{x}-\boldsymbol{\mu}|}{\sigma}$
2. Reject the null hypothesis H_0 if $G > \frac{n-1}{\sqrt{n}} \sqrt{\frac{t^2_{(\frac{\alpha}{n}-2)}}{n-1+t^2_{(\frac{\alpha}{n}-2)}}}$ where t is the t-student distribution.

Other model-based approaches generalize to arbitrary distributions, such as Gaussian mixtures. In case of Gaussian mixtures, we fit the model parameters using methods such as the Expectation-Maximization algorithm described in Section 1.4.

✔ Advantages

- Strong statistical foundation
- Output labels for outliers and non-outliers but can also output a score of outlierdness
- Efficient to compute and easy to interpret

✘ Disadvantages

- Often work only in the 1-dimensional case
- Require fixing a model for the data distribution
- The Gaussian distribution is affected by the dimensionality

4.1.2 Depth-based approaches

Depth-based approaches [Bar76] search for outliers that stands at the "border" of the data. One such approach is to run the convex hull algorithm to enclose the points inside a shape. The implicit assumption is that outliers are always located at the border of the data space, while the normal points are inside a convex shape.

The algorithm alternates the convex hull algorithm to the removal of the points at the border. The points are marked with the number of iteration in which they have been removed. The number represents the depth of the point.

Finally, the algorithm marks as outliers all the points having a depth $\geq t$. Figure 4.1 shows an example of the algorithm for a simple dataset.

Although simple, depth-based approaches have severe limitations. First, the data is assumed to lie on a convex shape, which is not the case for multimodal distributions in which the data contains many clusters. Second, convex hull computation is only efficient on low dimensional data. For these reasons, depth-based approaches can only be used in specific settings.

✔ Advantages

- Uses a global reference set for outlier detection
- Originally outputs a label, but can be extended for scoring easily, take depth as a scoring value for example

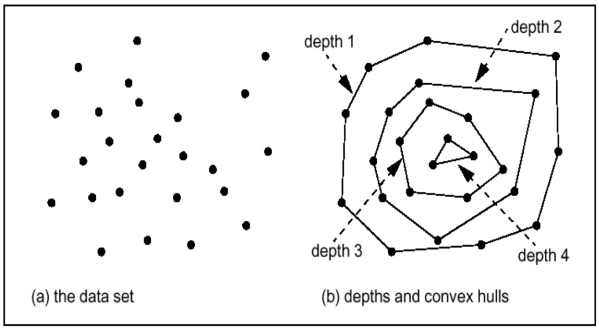


Figure 4.1: Depth-based approached for outlier detection detects outliers at different depths.

✘ Disadvantages

- The data must be unimodal
- Convex hull computation is usually only efficient in 2D and 3D spaces

4.1.3 Distance-based methods

Distance-based approaches can be thought as a direct formalization of the outlier notion by Hawkins. In Distance-based outlier detection a point $\mathbf{p}$ is an outlier if most points are further than ε from $\mathbf{p}$. Formally

Definition 3. *Given a radius ε and a percentage π, a point $\mathbf{p}$ is an outlier if at most π percent of the points $\mathbf{x} \in \mathcal{D}$ are at distance less than ε from $\mathbf{p}$.*

The set of outliers $\mathcal{O}_\mathcal{D}$, given Definition 3 is

$$\mathcal{O}_\mathcal{D} = \left\{ \mathbf{p} \text{ s.t. } \frac{|\{\mathbf{x} \in \mathcal{D} | dist(\mathbf{p}, \mathbf{x}) < \varepsilon\}|}{|\mathcal{D}|} \leq \pi \right\} \tag{4.1}$$

Distance-based outliers have a convenient notion of outlier score and they are typically easy to implement. However, since the radius ε and π are globally defined, distance-based outliers cannot detect outliers with different densities. This limitation is apparent in Figure 4.2; in that case o_2 might not be captured with large radius ε, while the points in C_1, might all be classified as outliers with the same ε

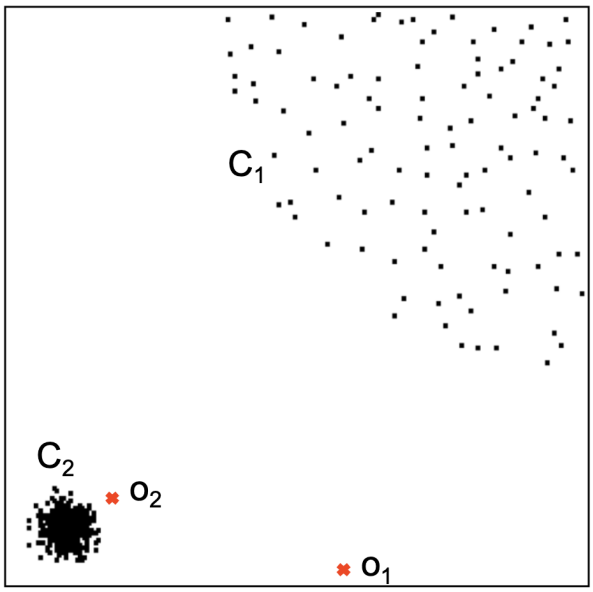


Figure 4.2: Distance-based methods fail to detect outliers if clusters have different densities.

✔ Advantages

- Easy to implement
- Highly interpretable

✘ Disadvantages

- The effectiveness of distance-based methods depends on the choice of the distance measure *dist*.
- Cannot detect collective outliers
- Sensitive to the choice of parameters ε, π.
- Sensitive to data density.

4.2 High-dimensional outlier detection

High-dimensional data suffers from the curse of dimensionality, which hampers the ability of methods to find patterns in the data. This phenomenon hides outliers within inliers and renders analyses harder.

✍ Recap

Check the curse of dimensionality in Section 3.4.

4.2.1 Angle-based outlier detection (ABOD)

The curse of dimensionality does not affect angles as much as distances. This is because the relative positions of the points do not change in higher dimension. Angle-based outlier detection (ABOD) [KSZ08] exploits this property to detect outliers. Since an outlier typically stands apart from other points, the angle formed with any pair of other points is typically small in variance. On the other hand, inliers form both small and large angles with the other points. Figure 4.3 nicely illustrates the ABOD idea.

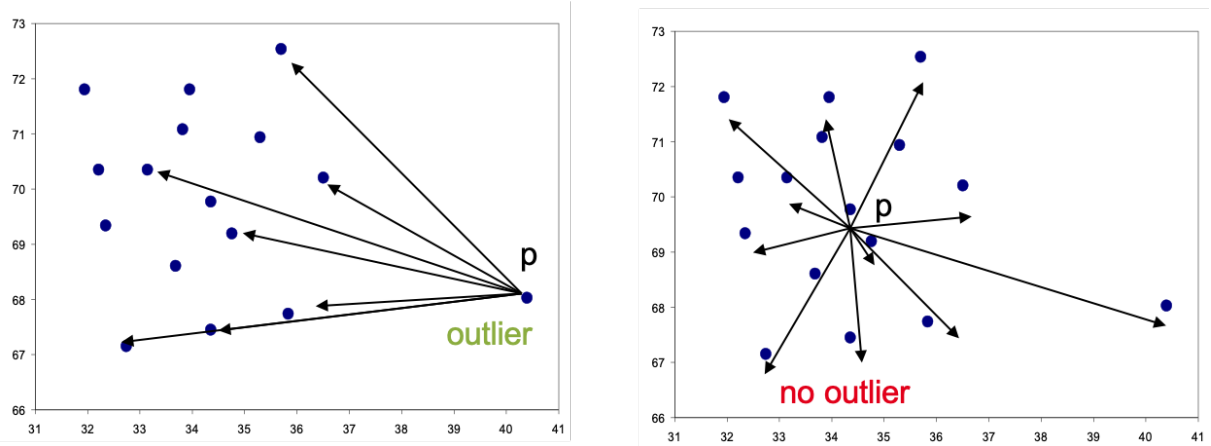


Figure 4.3: Angles between a point p and any pair of other points indicates whether p is an outlier. In the left image the point, by laying outside of the rest, forms angles with a much smaller variation than those in the right image.

ABOD intuitively works as follows.

- Consider the angles formed by a point $\mathbf{p}$ with any point $\mathbf{x}, \mathbf{y} \in \mathcal{D}$
- Plot the angles in a chart (the order does not matter).
- The variance of the angles represents an indication on whether a point is an outlier (small variance) or an inlier (large variance).

ABOD measures the variance of the angles, weighted by the corresponding distances by

$$ABOD(\mathbf{p}) = VAR_{\mathbf{x},\mathbf{y}\in\mathcal{D}}\left(\frac{\langle \mathbf{x}-\mathbf{p}, \mathbf{y}-\mathbf{p}\rangle}{\|\mathbf{x}-\mathbf{p}\|^2 \cdot \|\mathbf{y}-\mathbf{p}\|^2}\right) \tag{4.2}$$

A point with a small ABOD denotes an outlier. The naïve computation of ABOD takes $\mathcal{O}(n^3)$ due to the number of pairs for each point. One possible solution is to consider only a subset of pairs. If the sample has guarantees, the final ABOD is a good approximation to the exact computation. Moreover, the variance can be lower-bounded by sampling and the bound used to prune inliers.

✔ Advantages

- Global approach to outlier detection
- Performs well on high-dimensional data

✘ Disadvantages

- The naïve algorithm runs in $\mathcal{O}(n^3)$
- Cannot detect collective outliers

4.2.2 Local methods

Local outliers defines an outlier a point which local density is larger than that of its own neighbors. As such, local outliers are defined in terms of nearest neighbors and require only to specify the parameter k to define how many neighbor points to consider. One of the popular local outlier methods is the Local Outlier Factor (LOF) [BKNS00].

The *k-nearest neighbor distance* is the distance $dist_k(\mathbf{p})$ between a point $\mathbf{p}$ and its k-nearest neighbor. We denote as $N_k(\mathbf{p})$ the k-nearest neighors of point $\mathbf{p}$.

Definition 4. *Reachability distance sets a lower bound between the k-nearest neighbor distance and the distance of a point* $\mathbf{x}$ *from a point* $\mathbf{p}$

$$rdist_k(\mathbf{p},\mathbf{x}) = \max\{dist_k(\mathbf{x}), dist(\mathbf{p},\mathbf{x})\} \tag{4.3}$$

The local reachability distance (lrd) of a point $\mathbf{p}$ is the average inverse reachability distance of a point $\mathbf{p}$ from its k-nearest neighbors. Intuitively, the higher the $lrd(\mathbf{p})$, the closer $\mathbf{p}$ is to its k-nearest points.

$$lrd_k(\mathbf{p}) = \frac{1}{|N_k(\mathbf{p})|}\frac{1}{\sum_{\mathbf{x}\in N_k(\mathbf{p})} rdist_k(\mathbf{p},\mathbf{x})} \tag{4.4}$$

Finally, the Local Outlier Factor (LOF) is the average ratio of the local reachability distances of the neighbors of $\mathbf{p}$ and that of $\mathbf{p}$. In particular, the local outlier factor measures the relative density of the neighbors of a point and the point itself.

$$LOF_k(\mathbf{p}) = \frac{1}{|N_k(\mathbf{p})|}\sum_{\mathbf{x}\in N_k(\mathbf{p})}\frac{lrd_k(\mathbf{x})}{lrd_k(\mathbf{p})} \tag{4.5}$$

If the density of points around $\mathbf{p}$ is less than the density around its neighbors, the ratio in Eq. 4.5 will be large.

LOF nicely cope with the curse of dimensionality, however, LOF struggles in regions with low-density as shown in Figure 4.4. To remedy this problem, the Connectivity-based outlier factor proposes to distinguish between low-density and isolation, by introducing the concept of *set-based nearest paths*. A set-based nearest path substitutes the nearest neighbor by weighing the scores on the number of steps required to reach a certain neighbor from a point $\mathbf{p}$. In this way, although the density might be low, in regions where points are somewhat connected (e.g., a ring shape), there exists a low-weight path to reach a certain $\mathbf{p}$.

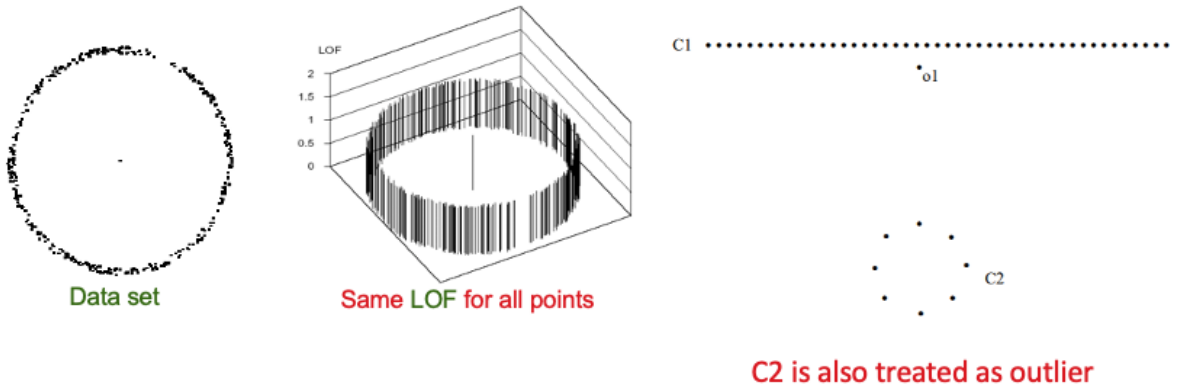


Figure 4.4: LOF is not able to detect outliers in low-density regions.

✔ Advantages

- Provides a scores of outlierdness.
- Easy interpretability.
- Not affected by the course of dimensionality.

✘ Disadvantages

- Inefficient due to the computation of the neighbors.
- Requires the correct setting of the parameter k.
- Hard to detect outliers in low-density regions.

4.2.3 Clustering-based methods

The clustering methods introduced in previous chapters can detect outliers by building on the definition of cluster. For instance, in DBSCAN, a cluster is a maximal set of density-connected objects, separated by sparsely populated areas in feature space. As such, every object not contained in a cluster must be an outlier. In particular, for DBSCAN the points in the noise cluster can be treated as outliers. Similarly, in k-means, a point $\mathbf{p}$ is an outlier if it is far from its closest centroid C_i; alternatively, $\mathbf{p}$ is an outlier if $dist(\mathbf{p}, C_i)/\sum_{\mathbf{x}\in C_i}\mathbf{x}$ is large.

FindCBLOF. In order to find outliers using clustering techniques we can use a general method called *FindCBLOF* [BKNS00].

- Compute clusters using a clustering algorithm
- Sort the clusters in decreasing order by number of points
- Assign to each point a cluster-based local outlier factor (CBLOF)
  - If point $\mathbf{p}$ belongs to a large cluster C_i, then $CBLOF(\mathbf{p}) = |C_i| \cdot S$, where S is similarity between $\mathbf{p}$ and the cluster
  - If $\mathbf{p}$ belongs to a small cluster, then $CBLOF(p) = |C_i| \cdot S$ where S is the similarity between $\mathbf{p}$ and the closest large cluster.

✔ Advantages

- Detect outliers without requiring labeled data
- Works on several types of data
- Clusters can be regarded as summaries of data

✘ Disadvantages

- The effectiveness depends on the chosen clustering method.
- High computational cost to compute the clusters

4.2.4 Isolation Forest

Isolation Forest (IF) [LTZ08] is a method to detect outliers based on the notion of separability. Based on separability, a point $\mathbf{p}$ is separable if it is easy to find a random hyperplane that separates the point from the rest. The idea is similar to a decision tree where the split and the attribute is decided randomly. The probability of a point $\mathbf{p}$ to be separated from the others in a random split of the data is higher if the point is an outlier. This means that such a point is, with high-probability, a leaf close to the root of a randomly-built decision tree.

However, the point $\mathbf{p}$ can also be separated by chance. As such, the algorithm builds a number of trees (i.e., a forest) and calculates the expected height $E(h(\mathbf{p}))$ of the root-leaf path and uses such expectation to compute an outlier score

$$s(\mathbf{p}) = 2^{-\frac{E(h(\mathbf{p}))}{c(n)}} \tag{4.6}$$

where $c(n)$ is the expected path length of a balanced binary search tree with n data points used for normalization. An outlier has a score close to 1, i.e. the expected length of the path from the root to the leaf $h(\mathbf{p}) \approx 0$.

Algorithm 9 describes how an IF is constructed. The first step randomly sample a set of points from the data $\mathcal{D}$; then, the function iTree construct a tree based on a sequence of random splits. The tree has depth at most L. After a number of iterations *ntrees* the algorithm computes the score $s(\cdot)$ for each point.

Algorithm 9 Isolation Forest

Input: Data $\mathcal{D}$, Maximum tree level L, Number of trees *ntrees*

$i \leftarrow 1$
$\mathcal{T} \leftarrow \emptyset$
while $i < ntrees$ **do**
    $S \leftarrow \text{Sample}(k, \mathcal{D})$
    $T \leftarrow \text{iTree}(S, 1, L)$
    $\mathcal{T} \leftarrow T \cup \mathcal{T}$
for each $\mathbf{x} \in \mathcal{D}$ **do**
    Compute $s(\mathbf{x})$ using from the forest $\mathcal{T}$

procedure iTree(S, l, L)
    if $l \geq l_{max}$ **or** $|S| \leq 1$ **then**
        return $exNode(S)$
    else
        $q \leftarrow$ Randomly select attribute
        $v \leftarrow$ Randomly select a split value p for q
    // Filter points by value p for attribute q
        $S_l = \text{Filter}(S, q < p)$
        $S_r = \text{Filter}(S, q \geq p)$
    // Return a new node with the left and right branch determined by the split
        return node(iTree($S_l, l+1, l_{max}$), iTree($S_r, l+1, l_{max}$), q, v)

✔ Advantages

- Easy to construct (no distance or density function required) avoiding difficult decisions on if data points are anomalies or not.
- Can achieve sublinear time complexity and a small memory footprint by exploiting subsampling. By eliminating major computational cost of distance calculation in all the distance and density based AD methods.
- Can provide anomaly explanations.

✘ Disadvantages

- hyperparameter tuning required such as number of trees, sample size and heights of trees.
- Requires a high percentage of relevant features to identify anomalies. In presence of features do that not provide information over the anomaly, iForest increases height randomly by ignoring this fact.

Clustering evaluation

This lecture provides a small account on how to evaluate the quality of a clustering algorithm. Being able to provide a number that explains the performance of an algorithm is paramount in any discipline. However, after the measure is chosen, it is also important to draw proper conclusions based on the measure's assumptions. There is nothing more wrong than concluding good performance of an algorithm without assuming that the measure can capture only a few aspects of a the multifarious problem of unsupervised techniques.

There are two main types of measures. **External measures** use typically external knowledge over the data, such as some ground truth. Section 5.1 presents some of the most common external measures for cluster analyses. **Internal measures** introduced in Section 5.2 are only based on data observations.

5.1 External measures

External measures assumes that, at least a part, of the ground-truth clustering is known *a priori*. A *ground truth cluster* is typically indicated with $\mathcal{T} = \{T_1, ..., T_k\}$. A *ground truth cluster* is the set of points belonging to the cluster, i.e., $T_j = \{\mathbf{x}_i \in \mathcal{D} | y_i = j\}$. The *clustering assignment* for point $\mathbf{x}_i$ is $y_i \in \{1, ..., k\}$.

5.1.1 Contingency table

The contingency table is a simple instrument to inspect the pairwise overlap among the ground-truth clusters and the clusters obtained by a clustering algorithm. Assume that our algorithm of choice detected r clusters, while we have k clusters in the ground-truth.

Definition 5.1.1. *For a clustering C and a ground-truth clustering $\mathcal{T}$, the* contingency table *is a $r \times k$ matrix $\mathbf{N}$ such that*

$$\mathbf{N}_{ij} = n_{ij} = |C_i \cap T_j|$$

An ideal clustering shows for each ground truth cluster T_j an assigned cluster C_i that contains all the points in T_j, i.e., $T_j = C_i$. Computing the contingency table takes $\mathcal{O}(nrk)$.

5.1.2 Purity

Purity quantifies to what extent a cluster C_i contains points **only** from one ground truth cluster.

$$purity_i = \frac{1}{|C_i|} \max_{j=1..k} \{n_{ij}\} \tag{5.1}$$

The purity of a clustering is the weighted sum of the purity of each cluster.

$$purity = \frac{1}{n} \sum_{i=1}^{r} \max_{j=1..k} \{n_{ij}\} \tag{5.2}$$

5.1.3 Maximum matching

Maximum matching overcomes the independent view of Purity. In particular, with purity we consider each cluster separately and potentially assign the same ground-truth cluster T_j to two different clusters. Another solution is to consider a one-to-one assignment that maximizes the overall overlap between ground-truth clusters and $C_1, ..., C_r$. This measure is called *maximum matching*. It works as follows:

1. Create a weighted bipartite graphs, where
   - the nodes V of the graph correspond to the clusters (ground-truth and computed), $V = \mathcal{C} \cup \mathcal{T}$;
   - the edges $E = \{(C_i, T_j\}_{i=1..r, j=1..k}$ are all the pairs (ground-truth cluster, computed clusters);
   - the weight of each edge $w((C_i, T_j)) = n_{ij}$ is the number of common points between C_i and T_j.

2. Compute the maximum weight matching of the graph using an algorithm such as the Hungarian algorithm which maximizes the sum of weights in the match $M^* \subseteq E$, i.e.

$$M^* = \arg\max_{M \subseteq E} \left\{ \frac{\sum_{(C_i,T_j) \in M} w((C_i, T_j))}{n} \right\}$$

Computing the maximum weight matching takes $\mathcal{O}((r+k)^3)$.

5.1.4 F-measure

Another possible way to measure clustering is *F-measure*, that is the harmonic mean between *precision* and *recall*. Precision $prec_i$ for cluster C_i is defined in the same way as $purity_i$ in Eq. 5.1. Recall $recall_i$ for cluster C_i quantifies the percentage of points included in C_i from the most common cluster. We denote such a cluster $T_{j_i} = \arg\max_{T \in \mathcal{T}} \{n_{ij}\}$

$$recall_i = \frac{1}{|T_{j_i}|} \max_{j=1..k} \{n_{ij}\} = \frac{n_{ij_i}}{|T_{j_i}|} \tag{5.3}$$

The F-measure for cluster C_i is then

$$F_i = \frac{2 \cdot prec_i \cdot recall_i}{prec_i + recall_i} = \frac{2n_{ij_i}}{|C_i| + |T_{j_i}|} \tag{5.4}$$

The F-measure for clustering $\mathcal{C}$ is the mean of the F-measure of its clusters

$$F = \frac{1}{r} \sum_{i=1}^{r} F_i \tag{5.5}$$

5.1.5 Conditional entropy

Conditional entropy, and especially, mutual information are popular choices for external quality measures. While both purity and F-measure can be easily swayed by a large number of small clusters, information theoretic approaches are more robust to this scenario.

Entropy for a clustering $\mathcal{C}$ is a measure of balance of the clusters. In particular, given the probability $p_{C_i} = \frac{|C_i|}{n}$ for a cluster C_i, entropy is

$$H(\mathcal{C}) = -\sum_{i=1}^{r} p_{C_i} \log p_{C_i} \tag{5.6}$$

It is easy to see that entropy is maximum when all the probability for all clusters is the same. We define $H(\mathcal{T})$ similarly. The conditional entropy for the ground-truth clustering $\mathcal{T}$ with respect to cluster C_i is

$$H(\mathcal{T} \mid C_i) = -\sum_{i=1}^{k} \frac{n_{ij}}{n_i} \log \frac{n_{ij}}{n_i} \tag{5.7}$$

Similarly the conditional entropy for the clustering $\mathcal{C}$ is

$$H(\mathcal{T} \mid \mathcal{C}) = -\sum_{i=1}^{r} \sum_{j=1}^{k} \frac{n_{ij}}{n} \log \frac{n_{ij}}{n_i} \tag{5.8}$$

As we can immediately notice the conditional entropy does not exclude that the right assignment for a ground-truth cluster T_j is the cluster C_i with the maximum overlap, but rather evaluates the contribution of each cluster. The perfect clustering achieves a conditional entropy of 0. On the other hand if $H(\mathcal{T} \mid \mathcal{C}) = H(\mathcal{T})$ the clustering $\mathcal{C}$ does not add any information to the initial entropy of $\mathcal{T}$ and, as such the two clusterings are independent.

5.1.5.1 Mutual information

A derived measure, which is popular for clustering is the *mutual information* and its normalized variant. Mutual information quantifies the amount of shared information between $\mathcal{C}$ and the ground-truth $\mathcal{T}$. In particular, how much $\mathcal{C}$ helps explaining $\mathcal{T}$. Intuitively, mutual information can be thought as a measure of correlation between the two clustering. Mutual information is defined as

$$I(\mathcal{C}, \mathcal{T}) = \sum_{i=1}^{r} \sum_{j=1}^{k} p_{ij} \log \left(\frac{p_{ij}}{p_{C_i} p_{T_j}} \right) \tag{5.9}$$

where p_{ij} is the normalized count of common points $\frac{n_{ij}}{n}$. It is again easy to notice that if both $\mathcal{C}$ and $\mathcal{T}$ are independent, they probabilities are the product of the probabilities and $I(\mathcal{C}, \mathcal{T}) = 0$. The common form of mutual information is it's normalized version

$$NMI(\mathcal{C}, \mathcal{T}) = \frac{I(\mathcal{C}, \mathcal{T})}{\sqrt{H(\mathcal{C}) H(\mathcal{T})}} \tag{5.10}$$

which takes values in $[0, 1]$ where 1 is an indication for a good clustering.

5.2 Internal measures

Internal measures depends only on the data and does not use any external information. These measure include the Silhouette coefficient, the sum of square distances, and the Davies-Bouldin index. More information can be found in the book [ZMJ20, Chapter 17.2].

The use of internal measures typically only complement the analysis but, given the potential bias of the algorithm towards such measures, they provide less information on the overall quality.

Bibliography

[ABKS99] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *ACM Sigmod record*, 28(2):49–60, 1999.

[AGGR98] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD*, pages 94–105, 1998.

[AHWY04] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. A framework for projected clustering of high dimensional data streams. In *VLDB*, pages 852–863, 2004.

[Bar76] Vic Barnett. The ordering of multivariate data. *Journal of the Royal Statistical Society: Series A (General)*, 139(3):318–344, 1976.

[BF98] Paul S Bradley and Usama M Fayyad. Refining initial points for k-means clustering. In *ICML*, volume 98, pages 91–99. Citeseer, 1998.

[BKNS00] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *SIGMOD*, pages 93–104, 2000.

[EKS+96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, volume 96, pages 226–231, 1996.

[Haw80] Douglas M Hawkins. *Identification of outliers*, volume 11. Springer, 1980.

[HK03] Alexander Hinneburg and Daniel A Keim. A general approach to clustering in large databases with noise. *Knowledge and Information Systems*, 5(4):387–415, 2003.

[KKK04] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. Density-connected subspace clustering for high-dimensional data. In *SDM*, pages 246–256. SIAM, 2004.

[KR90] Leonard Kaufman and Peter J Rousseeuw. Partitioning around medoids (program pam). *Finding groups in data: an introduction to cluster analysis*, 344:68–125, 1990.

[KSZ08] Hans-Peter Kriegel, Matthias Schubert, and Arthur Zimek. Angle-based outlier detection in high-dimensional data. In *KDD*, pages 444–452, 2008.

[Llo82] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[LTZ08] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *ICDM*, pages 413–422. IEEE, 2008.

[Rou87] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[ZMJ20] Mohammed J Zaki and Wagner Meira Jr. *Data Mining and Machine Learning: Fundamental Concepts and Algorithms*. Cambridge University Press, 2020.

[ZRL96] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. *ACM sigmod record*, 25(2):103–114, 1996.

Part II

Graph Mining

This module introduces **graph mining**. Graph mining is a discipline of data mining that specializes in the analysis of data in which data points form networks, or graphs, of connections. The purpose of graph mining is to extract patterns from graphs. Graphs are popular in a multitude of application scenarios from biomedical to social analysis. We will look at three different areas of graph mining:

- Community detection: the task of finding groups of nodes with similar characteristics
- Link analysis: the task of understanding which node is more important than others
- Embedding: the task of capturing similarities among nodes

Let us first introduce graphs.

Definition 5.2.1. *A graph is a set of vertices $V = (v_1, ..., v_n)$ connected through edges $E \subseteq V \times V$. A graph is represented by a pair $G = (V, E)$*

Alternatively, we can define a graph a set of points $\mathcal{D} = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ connected through edges E, i.e. $G = (\mathcal{D}, E)$. The latter definition extends the previous data model we have used in Module I. Sometimes we will use the shorthand notation $(i, j) \in E$ to indicate and edge between node v_i and node v_j. Similarly, we will use i, j instead of v_i, v_j, for simplicity.

A graph can be *undirected* if for every two nodes v_i, v_j, if $(v_i, v_j) \in E$ then $(v_i, v_j) \in E$ or *directed* if the implication does not always hold.

Graph representations. A graph can be represented by *adjacency lists* or an *adjacency matrix*. These two representations are semantically equivalent but they are used in different applications.

An **adjacency list** for a node v_i is the set of nodes v_j directly connected to v_i, i.e., for which there exists an edge $(v_i, v_j) \in E$. The set of such a nodes are called first-order *neighbors* or simply neighbors $N(v_i) = \{v_j | (v_i, v_j) \in E\}$. The neighbors extends the definition of DBScan 2.1 to graphs. In a certain manner, graphs are structures in which the neighborhood is explicitly defined; however, the distance among points is not provided.

The **adjacency matrix A** is a $n \times n$ matrix that denotes all the edges in the graph. In it its simplest form, the adjacency matrix is such that $a_{ij} = 1$ if $(v_i, v_j) \in E$ and 0 otherwise. An adjacency matrix is typically a sparse matrix as most of its entries are 0. If the graph is undirected, the adjacency matrix is a symmetric matrix.

Basic definitions. We provide some basic terminology that will be useful in the rest of the module.

Definition 5.2.2. *The degree of a node is the number of neighbors of a node. We denote as $d_i = |N(v_i)|$ the degree of node v_i.*

A *path* in a graph is a sequence $\langle v_1, ..., v_p \rangle$ such that, for $(v_i, v_{i+1}) \in E$ for each $i \in [1, p-1]$. The *diameter of a graph* is the size of the longest shortest-path.

Graph construction. Sometimes the edges of a graph, i.e., the graph structure, is not provided. In that case we can directly construct a graph from a dataset $\mathcal{D}$. There are a number of popular ways to construct graphs from data:

- k-NN graph: Connect every point $\mathbf{x}$ with its closest k points
- ε-neighbourhood: Connect all points $\mathbf{x}$ with the points within ε distance
- Fully-connected graph: Choose a distance *dist*, then connect any point to any other point. The weight of each edge (v_i, v_j) corresponds to $(1 - dist)$ between $\mathbf{x}_i$ and $\mathbf{x}_j$

➳ Objectives

After finishing this module, you should be able to:

- Describe the purpose of graph mining and its main tasks
- Discuss the main applications for graph mining and when it is preferrable to traditional data analysis
- Compare different methods for community detection among each other
- Categorize graph mining tasks and techniques by their strengths and weaknesses
- Connect the link analysis task to embeddings
- Apply the main graph mining techniques and run them over small datasets.
- Generalize the algorithms to other kind of data.

Lecture 6 *March 7, 2023*

Spectral graph theory and clustering

Instructor: *Davide Mottin*

In this lecture, we will look at graphs from a different angle and a different perspective. We will study how the structure, intuitively, reacts to signals sent over such a structure. A useful analogy is to imagine a graph as a set of marbles connected through strings. A force on one of the marbles would propagate on the rest of the marbles. By looking at how the different marbles vibrate we could conclude important properties on the weight of each marbles as well as on the strength in the strings.

This exquisitely exotic analogy is connected to the adjacency matrix and the fact that such a matrix, from the point of view of a single node, represents a way to propagate a message or a force over the neighbors of the graph. If we know how a message propagate over the graph, we will be able to study properties of a graph, such as degree, paths, diameters under a completely different perspective. This is the purpose of this lecture.

✍ Recap

For this lecture it is important to refresh some basics in linear algebra, especially matrix-vector multiplication, eigenvalues and eigenvectors.

6.1 The adjacency matrix

We consider our graphs undirected. As such, the graph's adjacency matrix is a symmetric matrix. For reasons that will appear clearer later, the fact that we have a symmetric matrix is important for our analysis.

One important characterization of the adjacency matrix is that it is an operator over the nodes in the graph. To appreciate this fact, we start by multiplying a vector $\mathbf{x} \in \mathbb{R}^n$ with the adjacency matrix. We can think the vector as values associated to each of the nodes. In some other cases, it is useful to think that $\mathbf{x}$ is the vector form of a function $f : V \to \mathbb{R}$. Let us look at the the operation $\mathbf{Ax} = \mathbf{y}$

We analyze the meaning of

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \tag{6.1}$$

The vector $\mathbf{y}$'s i-th element is $y_i = \sum_{(i,j)\in E} x_j$. Therefore, there entry y_i is the sum of labels x_j of neighbours of i. This means that $\mathbf{A}$ is a matrix that transforms the value of a node as a sum of the value of the neighbors. If we repeat this aggregation multiple times we obtain a propagation of the initial labels $\mathbf{x}$ over the network, i.e. $\mathbf{x}^{(t)} = \mathbf{A}\mathbf{x}^{(t-1)}$.

One interesting question is whether there exists a fixed-point in this iteration. In other words, whether we can a $\mathbf{x}$, such that

$$\mathbf{Ax} = \lambda\mathbf{x} \tag{6.2}$$

Equivalently, can we find an eigenvector $\mathbf{x}$ with eigenvalue λ for the matrix $\mathbf{A}$? This legit question is the essence of spectral graph theory.

6.2 Spectral graph theory

Spectral graph theory is the theoretical framework that analyses properties of the graph by inspecting the eigenvalues and eigenvectors of matrices that represent the graph structure. As we have seen, the most common of such matrices is the adjacency matrix.

Given a graph G and a matrix $\mathbf{M}$ that describes the graph's structure, the *graph spectrum* is the set of $\mathbf{M}$'s eigenvalues $\Lambda = \{\lambda_1, \dots, \lambda_n\}$ sorted in descending order $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$.

⚠ Remark

In some books and papers the spectrum is sorted in ascending order.

Recall that $\lambda \in \mathbb{C}$[1] is an eigenvalue if there exists a vector $\mathbf{x} \in \mathbb{C}^n, \mathbf{x} \neq 0$ such that $\mathbf{Ax} = \lambda\mathbf{x}$ or, alternatively $(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = 0$, which implies that $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$.

Symmetric matrices. The adjacency matrix $\mathbf{A}$ is a symmetric matrix. This means that $a_{ij} = a_{ji}$ for all i, j. If $\mathbf{A}$ is a symmetric matrix, the following statements are equivalent

- All eigenvalues are positive $\lambda \geq 0$
- The matrix is positive semi-definite, i.e., $\mathbf{x}^\top\mathbf{Ax} \geq 0$ for all $\mathbf{x}$
- There exists a matrix $\mathbf{N}$, such that $\mathbf{A} = \mathbf{N}^\top\mathbf{N}$

Another important characterization of symmetric matrices is that all eigenvalues are real numbers and that the eigenvectors are orthogonal. That means that if we have two eigenvectors $\mathbf{u}, \mathbf{v}$ of a symmetric matrix, $\mathbf{u}^\top\mathbf{v} = 0$

Eigenvalues and eigenvectors of A. Let us now look to what are possible eigenvalues and eigenvectors of the adjacency matrix. We start with two simple examples.

Example 1: Let us suppose we have a connected d-regular graph, where each vertex has exactly degree d and there are no disjoint subgraphs. Since the graph is d-regular if we sum all the rows of $\mathbf{A}$ we obtain d. That means that if take the vector of 1's

$$\mathbf{1} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

the operation $\mathbf{A1} = d\mathbf{1}$. This means that $\mathbf{1}$ is an eigenvector with eigenvalue d.

We slightly complicate the previous example by describing a more interesting case.

Example 2: We have a graph with two components A and B, each of the components is d-regular. Since the sum of each row in the adjacency matrix is still d, the vector $\mathbf{1}$ is still an eigenvector. However, since there are two components we can also assign a 1 to the nodes in one component and a 0 to the nodes of the other component, such that

$$\mathbf{x}^\top = (\underbrace{1, \ldots, 1}_{A's \text{ nodes}}, \underbrace{0, \ldots, 0}_{B's \text{ nodes}})$$

The vector $\mathbf{x}$ is still an eigenvector with corresponding eigenvalue d. Moreover, the complementary vector $\mathbf{x}'$ that has 0 entries in the nodes corresponding to the component A and 1 elsewhere is also a valid eigenvector with eigenvalue d.

To summarize, in a d-regular graph we already know that $\mathbf{1}$ is an eigenvector. Additionally, we know that in symmetric matrices all the eigenvectors are orthogonal. That means that all the eigenvectors $\mathbf{u}$ should be such that $\mathbf{1}^\top\mathbf{u} = 0 \implies \sum u_i = 0$. This is a very important fact that will be used later. Moreover while we have discovered that the two first eigenvalues in a d-regular graph are equal and $\lambda_1 = \lambda_2 = d$, we believe that if we add only one edge between the two components the two eigenvalues should not be too different, i.e., $\lambda_1 \approx \lambda_2$.

However, we are still at the beginning of our journey into spectral properties.

[1] $\mathbb{C}$ is the set of a complex numbers.

6.2.1 Graph matrices

Before diving into the core part of spectral clustering, we define a set of useful matrices that can be easily computed from the adjacency matrix. The **degree Matrix** or $\mathbf{\Delta}$ is an $n \times n$ diagonal matrix where $\mathbf{\Delta}_{ii} = \sum_j a_{ij}$ is the degree of node i.

The most important matrix for spectral clustering is the Laplacian matrix. There is a number of Laplacian matrices, all of them with specific characteristics.

- **(Unnormalized) Laplacian Matrix** or L is an $n \times n$. It is defined as $L = \mathbf{\Delta} - \mathbf{A}$
- **Normalized (Symmetric) Laplacian** $L^{sym} = \mathbf{\Delta}^{-\frac{1}{2}} L \mathbf{\Delta}^{-\frac{1}{2}} = \mathbf{I} - \mathbf{\Delta}^{-\frac{1}{2}} \mathbf{A} \mathbf{\Delta}^{-\frac{1}{2}}$ is the normalized version of L.
- **Asymmetric (Random Walk) Laplacian** is $L^{rw} = \mathbf{I} - \mathbf{\Delta}^{-1} A$

Laplacian's properties. The Laplacian matrix has a trivial eigenvector $\mathbf{1}$ corresponding with the eigenvalue 0 since $L\mathbf{1} = 0$ Moreover, since the Laplacian is symmetric and positive semi-definite (PSD), all the eigenvalues are non-negative real numbers, and the eigenvectors are real and orthogonal.

Since the Laplacian is PSD, we know that $\mathbf{x}^\top L\mathbf{x} \geq 0$, but what does that $\mathbf{x}^\top L\mathbf{x}$ represent?

$$\mathbf{x}^\top L\mathbf{x} = \sum_{(i,j)\in E} (x_i - x_j)^2 \tag{6.3}$$

So, if we think $\mathbf{x}$ to a 1-D projection of our graph, the value in Eq. 6.3 is 0 if two connected nodes (i,j) have the same value x_i, x_j. Does it mean that $\mathbf{x}$ could provide useful structural information? We will see that in the next section.

6.3 Spectral clustering

At this point we are interested in understanding what is the quantity $\mathbf{x}^\top L\mathbf{x}$ and what can we understand from the values of $\mathbf{x}$. To do this, let us consider a problem which seems unrelated to then realize that what we are looking for is related to Eq. 6.3.

Consider the problem of graph partitioning that aims at finding sets of nodes that can be easily separated. For instance, we are interested to split a computer network into set of computers, i.e., subnetworks, such that the number of connections among different subnetworks is minimized. This problem is particularly important when the graph is so large that it cannot fit into one machine (e.g., the Internet graph). We start out with the simpler case of bi-partitioning, that is finding two disjoint groups of nodes A and B, such that the number of edges crossing the two groups is minimized.

We can express this objective as a function of the "edge cut" of the partition. A cut is a set of edges with only one vertex in a group.

$$W(A, B) = \sum_{i\in A, j\in B} a_{ij} \tag{6.4}$$

If we directly minimize the cut we might end up in unfortunate solutions in which a partition contains only one node. We want to avoid this case, so that the two partitions have a similar number of nodes. To do that we can minimize a *normalized cut*

$$J_{rc}(C) = \frac{W(A, B)}{|A|} + \frac{W(A, B)}{|B|} \tag{6.5}$$

where $C = \{A, B\}$. The quantity in Eq. 6.5 relates to density as it computes the amount of edges crossing the partition in relation to the nodes in the partition. However, computing the optimal normalized cuts is **NP**-hard.

From cut to Laplacian. An alternative manner to write the cut is to consider a variable x_i for each node i, such that for a partition (A, B)

$$x_i = \begin{cases} +1 & \text{if } i \in A \\ -1 & \text{if } i \in B \end{cases} \tag{6.6}$$

In this way, we can rewrite Eq. 6.4 as

$$W(A,B) = \sum_{(i,j)\in E} |x_i - x_j| \tag{6.7}$$

So, if we want to minimize the cut we can just minimize the sum of absolute differences among $\mathbf{x}$ entries. Often, it is more convenient instead to minimize the square distances $\sum_{(i,j)\in E}(x_i - x_j)^2$. However, as we have seen in Eq. 6.3 this corresponds to $\mathbf{x}^\top L\mathbf{x}$. Similarly, we can see that

$$\frac{\mathbf{x}^\top L\mathbf{x}}{\mathbf{x}^\top\mathbf{x}}$$

corresponds to a normalized version of the cut and is called the Rayleigh's quotient.

6.3.1 Minimum-cut relates to spectral properties

We have already assessed that $\mathbf{x}^\top L\mathbf{x}$ relates to cut, but we also know that if the vector $\mathbf{x}$ is discrete the problem is **NP**-hard. But what happens if we allow the vector $\mathbf{x}$ to take real values? Do we have an effective way to solve this problem? The answer is, yes.

Recall that $L\mathbf{x} = \lambda\mathbf{x}$. This equation can be equivalently written as $\mathbf{x}^\top L\mathbf{x} = \lambda$. We are getting closer to our cut problem, but in order to do so we need to state on important characterization of eigenvalues.

Variational characterization of eigenvalues. Consider a matrix $\mathbf{M}$ real and symmetric. The eigenvalues $\lambda_1, \ldots, \lambda_n$ are sorted in decreasing order. The variational characterization of the eigenvalues states that the smallest eigenvalue corresponds to

$$\lambda_n = \min_{\mathbf{x}\neq 0} \frac{\mathbf{x}^\top\mathbf{M}\mathbf{x}}{\mathbf{x}^\top\mathbf{x}} = \min_{\mathbf{x}\neq 0} \frac{\sum_{ij} m_{ij}x_i x_j}{\sum_i x_i^2} \tag{6.8}$$

where $\mathbf{x}_n$ is the eigenvector corresponding to the eigenvalue λ_n Similarly, the second smallest eigenvector λ_{n-1} is the solution

$$\lambda_{n-1} = \min_{\mathbf{x}\neq 0, \mathbf{x}^\top\mathbf{x}_1 = 0} \frac{\mathbf{x}^\top\mathbf{M}\mathbf{x}}{\mathbf{x}^\top\mathbf{x}} \tag{6.9}$$

And so on with all the eigenvalues. Finally, λ_1

$$\lambda_1 = \max_{\mathbf{x}\neq 0} \frac{\mathbf{x}^\top\mathbf{M}\mathbf{x}}{\mathbf{x}^\top\mathbf{x}} \tag{6.10}$$

Minimum cut as eigenvalues of the Laplacian. We should now be able to connect all the ingredients together

- All eigenvectors of a symmetric matrix are orthogonal and that the Laplacian matrix for an undirected graph is a symmetric matrix.
- The eigenvalue that minimizes Eq. 6.8 is the eigenvalue 0 that corresponds to the eigenvector $\mathbf{1}$ of the graph's Laplacian L. This corresponds to placing all nodes into a single partition.
- The second smallest eigenvalue λ_{n-1} corresponds to an eigenvector $\mathbf{x}$ that minimizes Eq. 6.9 and is orthogonal to the eigenvector $\mathbf{1}$. That is, $\sum x_i = 0$.
- This eigenvector is called the Fiedler's vector and minimizes a relaxed version of the normalized cut.

6.3.2 Spectral Clustering Algorithm

After having analyzed the spectral properties, we can now define the Spectral clustering algorithm.

Spectral clustering work in three phases.

- **Preprocessing:** Construct a matrix representation of the graph

- **Decomposition:** Compute the eigenvalues and eigenvectors of the matrix. Map each point to a lower-dimensional represenation based on one or more eigenvectors
- **Grouping:** Assign points to two or more clusters, based on the new representation

The *Spectral Partitioning Algorithm* finds a partition of the graph that is close to the optimal partition. In the decomposition phase, Spectral clustering maps vertices to corresponding components of λ_{n-1} and its vector. In the grouping phase, we sort the components of the new 1-dimensional vector and split the nodes in positive and negative or using the median value.

Example 1: Consider for example (the first column is the node index) the eigenvector associated to the eigenvalue λ_{n-1}

$$\begin{pmatrix} 1 & 0.3 \\ 2 & 0.6 \\ 3 & 0.3 \\ 4 & -0.3 \\ 5 & -0.3 \\ 6 & -0.6 \end{pmatrix} \tag{6.11}$$

Then we could split them into $A = \{1, 2, 3\}, B = \{4, 5, 6\}$. We could also do something more expensive and try to minimize normalized cut in 1D.

6.3.3 k-way spectral clustering

We can easily extend spectral clustering to the case of finding k-partitions. One such strategy would be to recursively split the graphs into two partitions, until k partitions are returned.

Another strategy entails the use of multiple eigenvectors besides the one corresponding to the second smallest eigenvalue [NJW01, SM00]. By "stacking on" the eigenvectors each node is embedded into a multi-dimensional space. Finally, we run k-means in this new space to find k-partitions.

The use of multiple eigenvectors has the following advantages:

- Approximates the optimal cut (k-way normalized cut)
- Emphasizes cohesive clusters by increasing the unevenness in the distribution in the data. The associations between similar points are amplified. The data begins to approximate clustering
- Provides a well-separated space. It transforms data to a new embedded space consisting of k **orthogonal** basis vectors
- Multiple eigenvectors prevent instability caused by information loss

Finally, spectral clustering can connect nicely to Kernel k-means.

✔ Advantages

- Clusters can have arbitrary shapes and size and are not restricted to convex shapes.
- Efficient in normal (sparse) graphs
- Robust to noise and is theoretically grounded and well-connected to graph properties

✘ Disadvantages

- Inefficient on dense graphs ($\mathcal{O}(n^3)$ worst case)
- Need to provide number of clusters k

Lecture 7 *March 14, 2023*

Community detection

Instructor: *Davide Mottin*

In this lecture, we introduce the task of community detection. Community detection finds sets of nodes, i.e., communities, that are highly connected inside the community and coarsely connected outside the community. Communities have a strong sociological interpretation as they are defined as groups of individuals sharing common interests. The main question for community detection algorithms is how to define communities in mathematical terms and how to retrieve such communities fast.

Formally, a community is a subset $C \subseteq V$ of the nodes V. Non-overlapping communities or partitions are pairwise disjoint sets C_i, C_j such that $C_i \cap C_j = \emptyset$ for each i, j. Overlapping communities are not necessarily pairwise disjoint; as such, communities potentially share common nodes.

We will look at different definitions and depart from the cut-based approach of spectral clustering to define modularity. Modularity is a measure of cohesiveness among the nodes of the communities. While spectral clustering and modularity optimization find partitions of the graph, we will also look at overlapping communities as the counterpart of soft clustering in Lecture 1. We study two algorithms, clique percolation and a probabilistic model called AGM / BigClam.

7.1 Non-overlapping community detection

Non-overlapping community detection finds disjoint communities, i.e., partitions. There is a number of partitioning methods. Two common algorithms for non-overlapping community detection are Spectral clustering and Modularity optimization. We have already talked about spectral clustering in Lecture 6. We now look at Modularity optimization.

7.1.1 Modularity optimization

Modularity [New06] is a measure of cohesiveness among communities. Intuitively, a set of communities should be dense inside and coarsely connected among one another. This intuition is the driving principle in modularity. In practice, modularity $\mathcal{Q}$ compares the density inside communities with *a null model* of density. The null model assumes that nodes can form random connections with any other node.

$$\mathcal{Q} \propto \sum_{C \in \mathcal{C}} [\text{number of edges within community } C - \text{expected number of edges within community } C]$$

The main question is how do we quantify the edges within a community and what is a good null model.

Edges within community This is the easiest term. In order to find how many edges belong to the same community we simply sum the edges in each community and normalize by the total number of edges.

$$\frac{1}{2m} \sum_{C \in \mathcal{C}} \sum_{i \in C} \sum_{j \in C} a_{ij} \tag{7.1}$$

Note that as we consider all pairs of nodes, we need to normalize by twice the number of edges $2m$ as we count both (i, j) and (j, i).

Modularity's null model Given a graph G with $n = |V|$ vertices and $m = |E|$ edges, construct a rewired graph G'. The rewired graph has the same degree distribution but the connections among nodes appear at random. By ensuring the same degree, we assert that we do not compare the graph G with a completely random one; that would be meaningless. To understand why it is meaningless we need to think that nodes are only allowed to change community but not "type". If a node has a large number of neighbors, changing the size would completely destroy its meaning. It is useful to take an example in this case. Consider a social network. Nodes with large degrees typically represent celebrities. As such, altering their degree might end up transforming Brad Pitt into a common Davide.

If nodes are allowed to form any connection with any other node as long as the degree is preserved, the probability that given one node i the other one is j is

$$p_{ij} = \frac{d_i d_j}{4m^2} \tag{7.2}$$

where d_i, d_j are the degrees of node i and j, respectively. The expected number of edges between nodes i and j is $2m \cdot p_{ij} = 2m\frac{d_i d_j}{4m^2} = \frac{d_i d_j}{2m}$ As such, the expected number of edges in G' is

$$\begin{aligned} &= \frac{1}{2}\sum_{i\in V}\sum_{j\in V}\frac{d_i d_j}{2m} = \frac{1}{2}\cdot\frac{1}{2m}\sum_{i\in V} d_i\left(\sum_{j\in V} d_j\right) = \\ &= \frac{1}{4m}2m\cdot 2m = m \end{aligned} \tag{7.3}$$

Modularity Putting all together, the modularity of a partion $\mathcal{C}$ of G is

$$\mathcal{Q}(G,\mathcal{C}) = \frac{1}{2m}\sum_{C\in\mathcal{C}}\sum_{i\in C}\sum_{j\in C}\left(a_{ij} - \frac{d_i d_j}{2m}\right) \tag{7.4}$$

Modularity in Eq. 7.4 takes values in the range $[-1, 1]$. A modularity close to 1 indicates a good partition, while a negative value indicates a partition in which connected nodes are separated. A strong community structure is usually considered between $\mathcal{Q} = 0.3$ and $\mathcal{Q} = 0.7$.

The direct modularity optimization is **NP**-hard. The Louvain algorithm [BGLL08] propose a greedy algorithm that runs in $\mathcal{O}(n \log n)$. The Louvain method works in a bottom-up fashion by progressively merging the two communities that bring the largest increase in modularity.

7.1.1.1 Spectral modularity

We present here instead a spectral method for modularity maximization. We first start, as usual, with a bi-partition of the graph, i.e., by dividing the graph in exactly two communities C_1, C_2. Let $\mathbf{c} \in \mathbb{R}^n$ be a real vector that indicates whether a node belongs to one or the other communit. An entry c_i is

$$c_i = \begin{cases} 1 & \text{if } v_i \in C_1 \\ -1 & \text{if } v_i \in C_2 \end{cases}$$

We can rewrite Eq. 7.4 as

$$\mathcal{Q} = \frac{1}{2m}\sum_{i,j}\left(a_{ij} - \frac{d_i d_j}{2m}\right)\delta(c_i, c_j) \tag{7.5}$$

where $\delta(c_i, c_j) = 1$ if $c_i = c_j$ and 0 otherwise. Notice that $\delta(c_i, c_j) = \frac{1}{2}(c_i c_j + 1)$. As such, we can further simplify Eq. 7.5 as

$$\mathcal{Q} = \frac{1}{4m}\sum_{i,j}\underbrace{\left(a_{ij} - \frac{d_i d_j}{2m}\right)}_{B_{ij}} c_i c_j = \frac{1}{4m}\mathbf{c}^\top\mathbf{B}\mathbf{c} \tag{7.6}$$

where $\mathbf{B}$ with entries $B_{ij} = a_{ij} - \frac{d_i d_j}{2m}$ is referred to as the *modularity matrix*.

The objective in Eq. 7.6 reminishes very close to that of spectral clustering. However, in this case we are trying to maximize the quantity. Since the matrix $\mathbf{B}$ is a symmetric matrix, the variational characterization of the eigenvalues tells us that the solution of the optimization $\max \frac{\mathbf{x}^\top\mathbf{M}\mathbf{x}}{\mathbf{x}^\top\mathbf{x}}$ is the eigenvecor that corresponds to the largest of the eigenvalues of the modularity matrix. Is that the case? Let us see another proof.

In order to compute the partition $\mathbf{c}$, let us relax the vector $\mathbf{c}$ to take any real value. Our objective, omitting the constant term $\frac{1}{4m}$, becomes

$$\begin{aligned} &\max_{\mathbf{c}} \mathbf{c}^\top\mathbf{B}\mathbf{c} \\ &\text{s.t. } \mathbf{c}^\top\mathbf{c} = n \end{aligned} \tag{7.7}$$

where the condition $\mathbf{c}^\top\mathbf{c}$ tells us that $c_i^2 + c_2^2 + ... + c_n^2 = n$. To solve the optimization, we compute the partial derivatives in $\mathbf{c}$

$$\frac{\partial}{\partial \mathbf{c}}\mathbf{c}^\top\mathbf{B}\mathbf{c} = 0 \implies \frac{\partial}{\partial \mathbf{c}}\left[\mathbf{c}^\top\mathbf{B}\mathbf{c} - \beta(n - \mathbf{c}^\top\mathbf{c})\right] \implies 2\mathbf{B}\mathbf{c} = 2\beta\mathbf{c}$$

The last equality is (again!) an eigenvalue-eigenvector equation, where β is an eigenvalue for the eigenvector $\mathbf{c}$. If we substitute $\mathbf{B}\mathbf{c} = \beta\mathbf{c}$ into 7.7 we obtain

$$\mathbf{c}^\top\mathbf{B}\mathbf{c} = \mathbf{c}^\top\beta\mathbf{c} = \beta\mathbf{c}^\top\mathbf{c} = \beta n$$

where the latter equality holds for the condition $\mathbf{c}^\top\mathbf{c} = n$. As such, in order to maximize the objective we need to select the largest eigenvalue and the corresponding eigenvector.

Finally, the partition of the graph is obtained by

$$c_i = \begin{cases} 1 & \text{if } v_i > 0 \\ -1 & \text{otherwise} \end{cases}$$

Detecting multiple communities Similar to spectral clustering, we would be tempted to generalize the spectral modularity to the case of multiple communities by recursive bi-partitioning. This method, however, assumes that the graph is cut into two parts and that the edges among the two partitions are removed. Unfortunately, removing edges changes also the degree of the nodes, and in turns the null model that forms the matrix $\mathbf{B}$, ending up maximizing the wrong quantity. As a remedy, we first consider the change $\Delta\mathcal{Q}$ in modularity after splitting a community C into two parts

$$\begin{aligned}
\Delta\mathcal{Q} &= \frac{1}{2m}\left[\frac{1}{2}\sum_{i,j\in C} B_{ij}\left(c_i c_j + 1\right) - \sum_{i,j\in C} B_{ij}\right] \\
&= \frac{1}{4m}\left[\sum_{i,j\in C} B_{ij}c_i c_j - \sum_{i,j\in C} B_{ij}\right] \\
&= \frac{1}{4m}\sum_{i,j\in C}\left[B_{ij} - \delta_{ij}\sum_{k\in C} B_{ik}\right]c_i c_j \\
&= \frac{1}{4m}\mathbf{c}^\top\mathbf{B}^{(C)}\mathbf{c}
\end{aligned} \tag{7.8}$$

where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise is called the Kronecker delta and $B_{ij}^{(C)} = B_{ij} - \delta_{ij}\sum_{k\in C} B_{ik}$. The above is a generalized spectral method, in which we can repeatedly find eigenvectors in the modified matrix $\mathbf{B}^{(C)}$. It also provides a convenient stopping criteria: Once the increment in modularity is ≤ 0 there is no other community that will increase the modularity further and the method can stop.

✔ Advantages

- Clusters can have arbitrary shape and size, i.e clusters are not restricted to have convex shapes.
- Efficient in common real graphs.
- Robust to noise.

✘ Disadvantages

- Inefficient on dense graphs with $\mathcal{O}(n^3)$ in the worse case.
- Resolution limit: Merge communities if the number of edges is large.

7.2 Overlapping community detection

Overlapping community detection finds communities in which nodes are shared among them. This problem is harder and the methods are not so clearly defined as in the case of partitioning.

7.2.1 Clique percolation

The first method, *clique percolation* detects overlapping communities by exploiting the definition of clique. A *clique* is a fully connected subgraph. In other words, a clique is a set of nodes that have an edge with any other node in the clique. A *k-clique* is a complete subgraph on k nodes.

A k-clique is *adjacent* to another clique if it shares $k-1$ vertices. In clique percolation we define a *community* the union of adjacent k-cliques. Given such a definition it is easy to formulate our first algorithm for overlapping community detection.

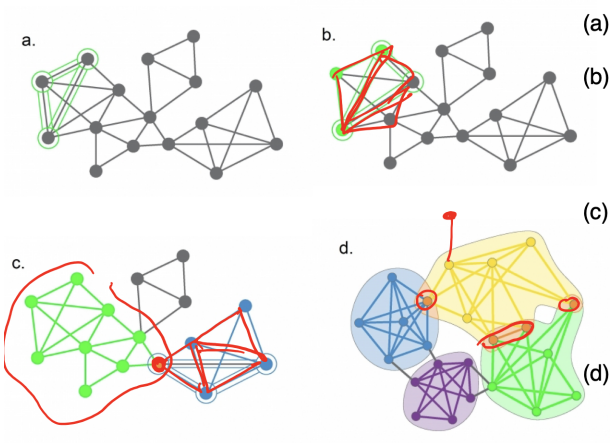
CFinder [PDFV05]. The CFinder algorithm starts with a k-clique defining a community, where k is a user parameter, and expands the community until there is no more adjacent cliques to add.

Algorithm 10 CFinder

Input: Graph G, Size of cliques k

1: Start with a k-clique
2: Roll the clique over adjacent cliques.
3: A k-clique community is the largest subgraph obtained by the union of all adjacent k-cliques.
4: Other k-cliques that can not be reached correspond to other clique-communities

Algorithm 10 shows the pseudocode of the algorithm. The picture below shows an example execution of this iterative algorithm.



✔ Advantages

- Easy to implement and understand.
- Finding k-cliques is polynomial.
- Communities are easily interpetable

✘ Disadvantages

- Requires to provide the size of the cliques k
- Rigid communities: The results depend on the existance of the cliques.

7.2.2 AGM / BigCLAM

Clique percolation defines communities as unions of cliques. AGM / BigCLAM [YL12, YL13] instead take a probabilistic approach. AGM and BigCLAM are two different models but with very similar premises. As such, the description of the two algorithms follows one after the other.

The main intuition behind these two probabilistic models is that a graph is a sample of an unknown distribution in which we flip a coin on the existence of an edge. In particular, for each pairs of nodes we flip a biased coin in which the probability determines how likely an edge exists. The existence probability

is determined by the community structure: the more communities two nodes share, the higher the probability. This is an intuitive fact, the strength of a relationship between two individuals is determined by the amount of common interests.

Let us first define our model of which our graph is a sample. We are given a graph G with communities $\mathcal{C}$ each of them existing with probability p_C, a membership set $M_i = \{C \in \mathcal{C} | v_i \in C\}$. The probability of existence of an edge in this model is

$$P(i,j) = 1 - \prod_{C \in M_i \cap M_j} (1 - p_C) \tag{7.9}$$

Intuitively, the above probability ensures that an edge exists if the nodes share at least one community. Technically speaking the above is an instance of a noisy-OR. If the nodes share no community, we assign a fixed small probability ε.

Our model assumes that each edge is generated independently by any other edge. As such, given the parameters $\Theta = (\mathcal{C}, M, \{p_C\})$, the likelihood that our graph G is sampled by the AGM model is

$$P(G|\mathcal{C}, M, \{p_C\}) = \prod_{(i,j)\in E} P(i,j) \prod_{(i,j)\notin E} (1 - P(i,j)) \tag{7.10}$$

AGM is a rather flexible model and can express a variety of community structures, such as non-overlapping and hierarchical communities.

The model assumes that we know already the parameters in order to compute the likelihood of a network. However, this is not the case. As usual, we reverse the process and find the parameters that most likely generated our graph. By maximizing Eq. 7.10 we can find the parameters of the model that maximize the likelihood. But how do we find this model? Finding Θ means finding a community membership for each node. This task is hard and computational infeasible.

> ⚠ Remark
>
> This modus operandi should be already familiar. We have already studied a similar approach in the context of EM 1.4.

From AGM to BigCLAM AGM model is hard to optimize. BigCLAM drops some of the requirements of AGM to attain efficiency. Under BigCLAM each node belongs to some community C_j with some strength F_{ij}. If $F_{ij} = 0$, node i is not a member of community j. The probability p_C that an edge between i and j exists if they belong to community C is

$$p_C(i,j) = 1 - \exp(-F_{iC}F_{jC}) \tag{7.11}$$

This means that the probability that an edge exists is proportional to the product of the strengths. If one node does not belong to the community $C, (F_{iC} = 0)$ then its probability is 0. The probability that at least one common community C links two nodes is

$$P(i,j) = 1 - \prod_{C}(1 - p_C(i,j)) = 1 - \exp(-\mathbf{F}_{i\cdot}\mathbf{F}_{j\cdot}^{\top}) \tag{7.12}$$

where $F_{i\cdot}$ is the row vector of the matrix $\mathbf{F} \in \mathbb{R}^{n \times |\mathcal{C}|}$.

To compute F, or better, to fit the model to the data, we use a maximum likelihood approach.

$$\arg\max_{\mathbf{F}} \prod_{i,j\in E} p(i,j) \prod_{(i,j)\notin E} (1 - p(i,j)) \tag{7.13}$$

Where $p(u,v) = 1 - \exp(-\mathbf{F}_{i\cdot}\mathbf{F}_{j\cdot}^{\top})$. Instead of maximizing the likelihood, we maximize the log-likelihood

$$P(G|\mathbf{F}) = \sum_{(i,j)\in E} \log(1 - \exp(-\mathbf{F}_{i\cdot}\mathbf{F}_{j\cdot}^{\top})) - \sum_{(i,j)\notin E} \mathbf{F}_{i\cdot}\mathbf{F}_{j\cdot}^{\top} \tag{7.14}$$

Since the matrix $\mathbf{F}$ is real, the model can be optimized with coordinate ascent over the rows.

Efficiency optimization Also, the gradients can be optimized by caching and compute an iteration in linear time.

$$\nabla P(G|\mathbf{F}) = \sum_{v \in N(i)} \mathbf{F}_{v\cdot} \frac{\exp\left(-\mathbf{F}_{i\cdot}\mathbf{F}_{v\cdot}^{\top}\right)}{1 - \exp\left(-\mathbf{F}_{i\cdot}\mathbf{F}_{v\cdot}^{\top}\right)} - \sum_{v \notin N(i)} \mathbf{F}_{v\cdot} \tag{7.15}$$

The factor $\sum_{v \notin N(i)} \mathbf{F}_{v\cdot}$ can be rewritten as

$$\sum_{v} \mathbf{F}_{v\cdot} - \mathbf{F}_{i\cdot} - \sum_{v \in N(i)} \mathbf{F}_{v\cdot}$$

which runs in linear time as both $\sum_{v} \mathbf{F}_{v\cdot} - \mathbf{F}_{i\cdot}$ can be cashed in advance.

✔ Advantages

- Detect overlapping and non-overlapping communities.
- Cluster can have arbitrary, e.g., non-convex, shapes.
- Nearly linear complexity.
- Flexible model.

✘ Disadvantages

- Requires the number of communities upfront.
- Hard to find the threshold for determining the membership of each nodes.

Lecture 8 *March 21, 2023*

Link Analysis

Instructor: *Davide Mottin*

This lecture introduces PageRank as a fundamental tool in graph mining, analyses its drawbacks, its connections to Markov processes and linear algebra, and proposes simple solutions to the overall problem of assigning importance to webpages or, more generally, to nodes in a graph.

The fundamental tool for analysing link are Random Walks. Surprisingly, if we move randomly from one node to another, following the edges and, on the way, we leave a token every time we visit a node, at some point, the normalized number of tokens, will not change. Why this happens and how this is so important for graph analysis will become more apparent in this lecture.

8.1 PageRank

We start our journey into link analysis with one of the most popular algorithms. In 1998 two guys from Stanford, Sergey Brin and Larry Page invented an algorithm, called PageRank [BP98]. They later implemented their algorithm to search web pages to help people in an online web that was becoming more and more complicated. The company became famous with the name Google. PageRank is such a revolutionary algorithm that it was inserted in 2007 in the list of top-10 data mining algorithms [WKRQ+08]. But why was PageRank so much better than the previous algorithms? What is the main idea behind PageRank?

PageRank is an algorithm that provides a score to each webpage. The importance of a page is defined as the sum of the scores of the page that link to it. This intuitive definition is inherently recursive and leads to the following voting process:

- Each link's vote is proportional to the importance of its source page.
- If node j with importance r_j has n outlinks, each each of the j's neighbors gets r_j/n votes.
- Page j's own importance is the sum of the votes on its in-links.

The PageRank then can be simply expressed as the sum of the scores of the neighbors. Let $m_{ji} = \frac{1}{d_i^-}$ the probability that from i we end up to i's neighbor j and d_i^- is the out-degree of node i.

Definition 8.1.1. *The* PageRank *of a page is the weighted sum of the PageRank of the pages that links to it*

$$r_j = \sum_i m_{ji} r_i \tag{8.1}$$

or, in vector format

$$\mathbf{r} = \mathbf{M}\mathbf{r} \tag{8.2}$$

where $\mathbf{M}$ is the matrix of the probability of landing on page i from j. $\mathbf{M}$ can be computed as $\mathbf{A}^\top \mathbf{\Delta}^{-1}$.

8.1.1 The Power Iteration method

Another way to think about Eq. 8.2 is as an iterative process that updates the previous value of $\mathbf{r}^t$ at iteration t with the new $\mathbf{r}^{t+1} = \mathbf{M}\mathbf{r}^t$. This iterative process leads to the Power Iteration method.

The PageRank vector $\mathbf{r}$ is initially uniform with $1/n$ for all nodes. The iteration continues until convergence $\|\mathbf{r}^t - \mathbf{r}^{t+1}\|_2 < \varepsilon$. Note that the choice of the stopping criteria $\|\cdot\|$ is arbitrary and can be easily substituted with a simple sum of absolute differences.

Algorithm 11 Power Iteration Algorithm for PageRank

Input: Transition matrix $\mathbf{M}$, number of nodes n
Output: PageRank vector $\mathbf{r}$

$\mathbf{r}^0 = \frac{1}{n}\mathbf{1}$
$t \leftarrow 0$
repeat
    $t \leftarrow t + 1$
    $\mathbf{r} \leftarrow \mathbf{r}^t$
    $\mathbf{r}^t \leftarrow \mathbf{Mr}$
until $\|\mathbf{r}^t - \mathbf{r}\|_2 < \varepsilon$

Dominant eigenvector. An important realization about the PageRank vector is that it is an eigenvector of the matrix $\mathbf{M}$. In fact, the PageRank corresponds to the eigenvalue 1. This is not hard to see from $\mathbf{Mr} = 1\mathbf{r}$. Furthermore, the PageRank is the dominant eigenvector (i.e., the one corresponding to the largest eigenvalue) of the matrix $\mathbf{M}$. But does the Power Iteration method converges to the dominant eigenvector? Yes, according to the following theorem.

Theorem 8.1.1. *The sequence* $\mathbf{Mr}^0, \mathbf{M}^2\mathbf{r}^0, \ldots, \mathbf{M}^t\mathbf{r}^0$ *approaches the dominant eigenvector of* $\mathbf{M}$.

Random walk interpretation Another useful characterization of the PageRank is in terms of random walks. This characterization assumes that at any point in time t an internet surfer lands on page i, and at time $t+1$ the surfer follows an outlink from i uniformly at random. Once the surfer lands on a different pages, they repeat the process of selecting a linked page uniformly at random.

Let $\mathbf{p}(t)$ be a vector whose ith coordinate is the probability that the surfer is at page i at time t. Then $\mathbf{p}(t)$ is a probability distribution over the pages. At time $t+1$ the surfer will reach any of the linked pages with probability $\mathbf{p}(t+1) = \mathbf{Mp}(t)$. At convergence $\mathbf{p}(t+1) = \mathbf{Mp}(t) = \mathbf{p}(t)$, then $\mathbf{p}(t)$ is a **stationary distribution** of the random walk.

The PageRank $\mathbf{r}$ satisfies $\mathbf{r} = \mathbf{Mr}$, so $\mathbf{r}$ is a stationary distribution for the random walk. A central result from the theory of random walks (Markov processes) says that graphs that satisfy certain conditions (check irreducibility of $\mathbf{M}$) have a unique stationary distribution and will eventually converge regardless of the initial distribution $t = 0$.

Convergence of PageRank Does the iteration $\mathbf{r} = \mathbf{Mr}$ converge? This is equivalent to ask, does the random walk process represented by the matrix $\mathbf{M}$ always admit a stationary distribution? The answer is unfortunately negative. To see that, it is easy to design a graph in which the process never converges by taking a clique. In a clique the nodes are connected to each other and the process will keep iterating indefinitely.

Other problems with PageRank The convergence problem of PageRank is not the only problem. Sinks and spider traps are also problematic for PageRank. A sink is a node without outgoing edges. A spider trap is a group of nodes without outgoing edges. Both expose a problem with PageRank: if there is no outgoing edges the PageRank cannot "escape".

✔ Advantages

- PageRank is fast to compute with Power Iteration method $\mathcal{O}(mt)$ where t is the number of iterations.
- PageRank provides a measure of relevance that does not depend on the content of a page.

✘ Disadvantages

- It measures generic popularity of a page, but is biased against topic-specific authorities. Solution is Topic-specific PageRank.
- Susceptible to link spam. Artificial link topographies created in order to boost page rank. TrustRank can remedy this.
- Uses a single measure of importance. Hubs and authorities may solve this.

8.2 Solving PageRank issues

The solution for all the PageRank's problems is one and is very simple: *teleportation*. Teleportation is the process under which, with some probability the surfer chooses a random node where to teleport. The process is the following. At each iteration t we toss a coin. With probability α we continue in our surfing by choosing randomly among the node's neighbors, else with probability $1-\alpha$ we select a random node to teleport.

$$r_j = \sum_{i,j \in E} \alpha \frac{r_i}{d_i} + (1-\alpha)\frac{1}{n} \tag{8.3}$$

Equivalently written in matrix notation

$$\mathbf{r} = \alpha \mathbf{M}\mathbf{r} + (1-\alpha)\left[\frac{1}{n}\right]_{n\times n} \tag{8.4}$$

One important observation is on the value of the teleport probability α. If α is close to 0 the PageRank will converge to the uniform distribution $1/n$; if $\alpha \to 1$ the PageRank might converge slower as it exhibits the same problem of the original version. A typical choice of α is in the range $[0.8, 0.9]$ [BP98].

8.2.1 Personalized PageRank

In Eq. 8.3 the teleportation term $(1-\alpha)\frac{1}{n}$ teleports equally on all nodes. However, in some applications we might need to teleport only on a small number of nodes by substituting the vector $\left[\frac{1}{n}\right]_n$ with a generic probabilistic vector $\mathbf{p}$ obtaining

$$\mathbf{r} = \alpha \mathbf{M}\mathbf{r} + (1-\alpha)\mathbf{p} \tag{8.5}$$

Equation 8.5 is called Personalized PageRank (PPR) and, while the original PageRank corresponds to random walks, Personalized PageRank corresponds to Random Walks with Restart (RWR)

$$\mathbf{r} = \alpha \mathbf{A}\mathbf{\Delta}^{-1}\mathbf{r} + (1-\alpha)\mathbf{p} \tag{8.6}$$

where $\mathbf{A}\mathbf{\Delta}^{-1}$ is a stochastic matrix as the sum of each column is 1. $\mathbf{r}$ is a stochastic vector as the sum of elements is 1. $\mathbf{p}$ is the restart vector, and is also stochastic.

8.2.2 Topic-specific PageRank

Topic-specific PageRank [Hav03] computes the Personalized PageRank for a group of nodes S. These nodes are selected among the pages that have the same topic. For instance, if we cluster the pages by words we might find different "communities". One way to restrict to one of these "communities" is to run PPR on the of initial pages with the same words. We can compute PPR for each topic set S and get a different vector r_s. We update the transition matrix $\mathbf{M}$ with a probability of teleporting to the nodes in the set S

$$w_{ij} = \begin{cases} \alpha m_{ij} + (1-\alpha)/|S| & \text{if } i \in S \\ \alpha m_{ij} + 0 & \text{Otherwise} \end{cases} \tag{8.7}$$

As we teleport uniformly to any node in S we can simply use the personalization vector $\mathbf{p}$ and define

$$p_i = \begin{cases} \frac{1}{|S|} & i \in S \\ 0 \end{cases}$$

In this way, the rest of the PPR definition remains the same besides $\mathbf{p}$.

8.2.3 TrustRank

Although the teleportation probability solves a deal of problems, PageRank may still be biased towards pages that spread false information. TrustRank [GGMP04] tries to remedy this issue with the use of trusted pages.

TrustRank is a topic-specific PR with a teleport set of trusted pages. The main idea behind PageRank is the principle of approximate isolation under which it is unlikely that a good page points to a bad page. In order to compute TrustRank we need to solve two problems. The first is how to select the set of trusted pages and the second is how to decide whether a page is a spam or is a good one.

Selecting a subset of trusted pages. In order to select a list of trusted pages we can use different approaches:

1. Sample a set of pages and ask a set of volunteers to evaluate whether the pages are good or bad. This is potentially expensive.
2. Select the k pages with the highest PageRank, assuming that bad pages cannot eventually reach too high PR values.
3. Select a set of trusted domains, e.g., au.dk as universities do not want to propagate bad information.

Detecting spam pages. One way to detect spam pages is to define a threshold value and mark all pages below such threshold as spam. However, finding the right threshold might be cumbersome.

Another solution is to compute **spam mass** as the deviation of TrustRank from PageRank. The spam mass is the fraction of a page's PageRank that comes from spam pages. Since in practice, we are oblivion of all the spam pages, we compute an estimate.

Let r_x be the PR of page x. Then r_x^+ is the PR of x with teleport only into trusted pages. Then spam mass is $x = \frac{r_x^-}{r_x}$ where $r_x^- = r_x - r_x^+$. Pages with high spam mass are considered spam.

8.3 Link farms

The last excruciating problem for PageRank is the one of link farms. A *link farm* is a page s, usually controlled by a malicious spammer, that is connected to a set of good and trusted pages. Such page creates a number T of fictitious pages that links from and to s with the purpose of increasing artificially its PageRank to appear higher in search engines. But how many of such pages does s needs?

Let be p_s the PageRank of the page s and p_X the PageRank of the "honest" pages that link to s. The PageRank p_t of any of fake pages T is then

$$p_t = \frac{\alpha p_s}{T} + \frac{1-\alpha}{n}$$

The PageRank p_s is the sum of the contribution of the honest pages p_X and the fake pages

$$\begin{aligned} p_s &= p_X + \alpha \frac{\alpha p_s}{T} + \underbrace{\frac{1-\alpha}{n}}_{\text{Small, ignore}} \\ &= \dots \\ &= \frac{p_X}{1-\alpha^2} + \frac{\alpha}{1+\alpha}\frac{T}{n} \end{aligned} \tag{8.8}$$

If $\alpha = 0.85$, the quantity $\frac{1}{1-\alpha^2} = 3.6$. As such for each honest page the malicious page s is rewarded 3.6 times the honest page's PageRank. This effect can be amplified with a large T. This is a very surprising ending.

8.4 HITS: Hubs and Authorities

The Hypertext-induced topic selection (HITS) [Kle99] is a measure of importance of pages and documents similar to PageRank. The main intuition behind HITS is that in order to find good newspapers, we can find experts that support the newspaper. Any link becomes a vote.

In HITS every page has two scores, a hub score and an authority score. The hub score is the sum of the scores to the authorities pointed by the hub; the authority score is the sum of votes coming from hubs.

HITS uses the *principle of repeated improvement* that iterates over hubs an authorities in turn. Initially, each page i starts with hub score 1. The iteration works in alternating manner: after the authorities collected their votes, the hubs sum the scores from the authorities. Once again the computation of the scores is a mutually recursive definition. A good hub links to many good authorities, and a good authority is linked to many good hubs. We represent hub and authority scores as h and a.

Algorithm 12 HITS

1: Initialize $a_i^0 = \frac{1}{\sqrt{n}}, h_i^0 = \frac{1}{\sqrt{n}}$ for all i
2: **repeat**
3: $\quad \forall i : a_i^{t+1} = \sum_{j,i\in E} h_j^t$
4: $\quad \forall i : h_i^{t+1} = \sum_{i,j\in E} a_j^t$
5: $\quad$ Normalize $\sum_i (a_i^{t+1})^2 = 1, \sum_j (h_j^{t+1})^2 = 1$
6: **until** $\sum_i \left(h_i^t - h_i^{t-1}\right)^2 < \varepsilon$ and $\sum_i \left(a_i^t - a_i^{t-1}\right)^2 < \varepsilon$ ▷ Convergence criterion

HITS converges to a single stable point. We can see that HITS indeed converge by considering the authority vector $\mathbf{a} = (a_1, \ldots, a_n)$ and the hub vector $\mathbf{h} = (h_1, \ldots, h_n)$. In this way we can rewrite $h_i = \sum_j A_{ij} a_j \implies \mathbf{Aa}$ and $a_i = \sum_{j,i\in E} h_j = \sum_j A_{ji} h_j \implies \mathbf{A}^\top \mathbf{h}$. For two consecutive iterations HITS iterations we obtain

$$\mathbf{a} = (\mathbf{A}^\top \mathbf{A})\mathbf{a} \tag{8.9}$$

$$\mathbf{h} = (\mathbf{A}\mathbf{A}^\top)\mathbf{h} \tag{8.10}$$

and repeat until the difference of $\mathbf{h}, \mathbf{a}$ in two different iterations is less than ε. From the two equations above it is easy to see that HITS converges to the principal eigenvectors of $\mathbf{h}^*$ for $(\mathbf{A}\mathbf{A}^\top)$ for and $\mathbf{a}^*$ for $(\mathbf{A}^\top \mathbf{A})$.

Lecture 9 *March 28, 2023*

Graph Embeddings

Instructor: *Davide Mottin*

This lecture introduces graph embeddings, that solve multiple tasks in graphs without requiring ad-hoc algorithms. The main problem with graphs is their lack of predefined node ordering. This means that it does not matter in what order we visit the nodes, the graph remain the same. This is not true for images, in which the pixels lay on a strict grid; therefore, a swap in the order of two pixels change the image. This also means that a graph is not-Eucledian, so the distance among two nodes does not depend on the (missing) coordinates of the nodes.

Graph embeddings sprout from the idea that the graph could become again a set of multi-dimensional points in which Euclidean distances represent similarities among the nodes. In its simplest form, a graph embedding takes in input a graph $G : (V, E)$ and returns an embedding function $f : V \to \mathbb{R}^d$ that projects a node into a d-dimensional point. The points in the d-dimensional space can then be analyzed using traditional data mining techniques. The reason why this is a good idea will be explained in this lecture.

We study two families of embedding methods:

- Linear embeddings that perform a linear transformation of the edges.
- Random-walk embeddings that preserve the probability that a node is reachable from another node.

9.1 Linear embeddings

Linear embeddings are the simplest form of graph embeddings. Let $\mathbf{z}_i \in \mathbb{R}^d$ the embedding vector of the node i. Intuitively, if we know the similarity of two nodes of two nodes v_i, v_j, the dot-product between the embedding vector $\mathbf{z}_i$ and $\mathbf{z}_j$, should approximate such a, similarity $sim(i, j) \approx \mathbf{z}_i^\top \mathbf{z}_j$. If two nodes are "close" under some definition of similarity, they should be also similar in the embedding space. How do we find such embedding vectors? In order to define such embedding vectors, we need

1. An encoder from nodes to the embedding space
2. A similarity among nodes in the graph

Once we have decided these two ingredients, we need to define an objective or loss function, such that the embeddings are the parameters of our optimization. Depending on the loss function, we can then use the appropriate optimizer to find the embeddings $\mathbf{z}_i$.

Shallow enconding We define the embedding matrix $\mathbf{Z}^{n \times d}$ in which row i is the embedding $\mathbf{z}_i$ of node i. Given $\mathbf{Z}$ we can encode each node in its corresponding embedding by a lookup on the embedding matrix. Let $\mathbf{v} \in \mathbb{I}^n$ the indicator vector that has a 1 in position i and 0 otherwise. Then

$$\mathbf{z}_i = \mathbf{Z}^\top \mathbf{v}$$

The above encoding is called *shallow encoding*.

Node similarity There are a number of possible ways in which we can define the similarity between two nodes. In the absence of any additional information, we can intuitively check whether two nodes are connected, are far from each other in shortest-path, are reachable from random walks, or simply share any common information (e.g., if we have node attributes). The richer the graph, the easier and more expressive our similarity is. In what follows, we will survey the most popular similarity measures and their corresponding embeddings.

9.1.1 Adjacency-based similarity

The adjacency-based similarity is the simplest form of similarity: two nodes are similar if they share an edge between them. As such, $sim(i,j) = a_{ij}$ implies that the dot-product among the embedding vectors should approximate the entries of the adjacency matrix $a_{ij} \approx \mathbf{z}_i^\top \mathbf{z}_j$. We can then find embeddings that minimize the distance among the entries a_{ij} and the dot-product among the embedding vectors $\mathbf{z}_i^\top \mathbf{z}_j$

$$\mathcal{L} = \sum_{(i,j)\in V\times V} |\mathbf{z}_i^\top \mathbf{z}_j - a_{ij}|^2 \tag{9.1}$$

which corresponds to the minimization of the norm between $\mathbf{A}$ and the matrix $\mathbf{Z}\mathbf{Z}^\top$, $\|\mathbf{Z}\mathbf{Z}^\top - \mathbf{A}\|_F^2$.

In order to minimize the objective in Eq. 9.1 we can use gradient descent (or stochastic gradient descent) or use matrix decomposition, such as SVD or QR-decomposition, on $\mathbf{A}$.

✔ Advantages

- The similarity function is straightforward.
- Many methods for solving the optimization problem.

✘ Disadvantages

- $\mathcal{O}(n^2)$ runtime to consider all node pairs, $\mathcal{O}(|E|)$ if we sum only over the non-zero entries [ASN+13].
- $\mathcal{O}(dn)$ parameters, one vector for each node.
- Only consider direct and local connections through the adjacency matrix.

9.1.2 Multi-hop similarity

Multi-hop similarity methods [OCP+16, CLX15] consider connections at distance k from a certain node to remedy to the strict requirements of the adjacency-based similarity. The kth power of the adjacency matrix captures k-hop connections. Indeed the position i, j in $\mathbf{A}^k$ represents the number of paths of lengths k from node i to node j.

⚠ Remark

There is a simple proof by induction to prove the above fact.

Similarly to the case above our loss function minimizes the differences between the entry i, j in the kth power of the adjacency matrix and the dot-product among the embedding vectors of the nodes i, j.

$$\mathcal{L} = \sum_{(i,j)\in V\times V} |\mathbf{z}_i^\top \mathbf{z}_j - a_{ij}^k|^2 \tag{9.2}$$

Similarly, this corresponds to minimizing $\|\mathbf{Z}\mathbf{Z}^\top - \mathbf{A}^k\|_F^2$. In practice, to avoid numerical explosion, GraRep [CLX15] use a log-transformed probabilistic adjacency matrix.

$$\tilde{a}_{ij}^k = \max\left\{\log\left(\frac{a_{ij}/d_i}{\sum_{l\in V}(a_{lj}/d_l)^k}\right)^k - \alpha, 0\right\} \tag{9.3}$$

where d is node degree, and α is a constant shift. GraRep trains embeddings with different k and concatenates the embedding vectors.

A more elastic option to capture multi-hop similarities relies on the overlap between node neighbourhoods, using overlap functions such as Jaccard similarity and Adamic-Adar score which is $s_{ij} = \sum_{l\in N(i)\cap N(j)} \frac{1}{\log N(l)}$. The loss function becomes

$$\sum_{(i,j)\in V\times V} |\mathbf{z}_i^\top \mathbf{z}_j - s_{ij}|^2 \tag{9.4}$$

One drawback with Linear embeddings is that, in their current form, they require to compute the pairwise similarities for all nodes, taking $\mathcal{O}(n^2)$ in the worst case. For large graphs, a quadratic complexity is not feasible. Moreover, we need to define the similarity beforehand.

9.2 Neural Embeddings

Neural embeddings offer a palatable alternative to the the computation of the pairwise similarities by exploiting the efficiency of probability distributions. Under this category fall random-walk based embeddings, and the non-linear similarity embeddings.

9.2.1 Random walk embeddings

As we have seen in Lecture 8, random walks determines the probability of reaching a certain node in the graph. This appealing definition can become integral part of an embedding method. Let $\mathbf{z}_i^\top \mathbf{z}_j$ be the probability that node i and node j co-occur in a random walk over the graph. We can estimate the probability of visiting node i from node j using random-walks or methods like Personalized PageRank (PPR). Recall that, if we sample a sufficiently large number of random walks, the process converges to the PageRank. This is an important fact that will be used later. Random walks are expressive and efficient as they do not require to traverse the entire graphs, they can be easily simulated and they converge to probability distributions.

Equipped with such a knowledge, assume we have a strategy R to sample random walks (e.g., PPR). We can define our embedding algorithm as follows.

1. Run short random walks starting from each node using strategy R
2. For each node i, collect $N_R(i)$, the multiset of nodes visited on random walks starting from i
3. Optimize the embeddings according to

$$\mathcal{L} = \sum_{i\in V} \sum_{j\in N_R(i)} -\log(p(j|\mathbf{z}_i)) \tag{9.5}$$

The intuition for the optimization in Eq. 9.5 is to find embeddings that maximize the likelihood of random walk co-occurrences. In practice, we can parameterize $p(j|\mathbf{z}_i)$ using softmax

$$p(j|\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i^\top \mathbf{z}_j)}{\sum_{l\in V} \exp(\mathbf{z}_i^\top \mathbf{z}_l)} \tag{9.6}$$

As such, we obtain the loss function

$$\mathcal{L} = \sum_{i\in V} \sum_{j\in N_R(i)} -\log\left(\frac{\exp(\mathbf{z}_i^\top \mathbf{z}_j)}{\sum_{l\in V} \exp(\mathbf{z}_i^\top \mathbf{z}_l)}\right) \tag{9.7}$$

The above loss can be optimized using gradient descent. However, the nested sum in Eq. 9.7 takes $\mathcal{O}(n^2)$, although the real bottleneck is the denominator of the probability, that sums over all nodes. One strategy to reduce the complexity is to employ a *negative sampling* approach.

Negative sampling treats the problem as a classification problem and splits the fraction into two parts.

$$\log\left(\frac{\exp(\mathbf{z}_i^\top \mathbf{z}_j)}{\sum_{l\in V} \exp(\mathbf{z}_i^\top \mathbf{z}_l)}\right) \sim \log(\sigma(\mathbf{z}_i^\top \mathbf{z}_j)) - \sum_{l=1}^{k} \log(\sigma(\mathbf{z}_l^T \mathbf{z}_{n_l})) \tag{9.8}$$

where $n_l \sim p_V$ is a node sampled from a negative distribution p_V over the nodes and σ is the sigmoid function. A typical choice of the negative distribution is to sample negative nodes proportional to the

degree of each node. The more we sample, the more robust is the approximation. A higher k corresponds to higher prior on negative events.

✔ Advantages

- Expressive and non-linear
- Can scale up with sampling techniques

✘ Disadvantages

- Similarity is fixed to random walks

9.2.2 General similarities

Another approach to graph embeddings is to generalize the similarity-based approaches to any *samplable* similarity. An appealing choice for similarity is PPR. For a given restart vector $\mathbf{v}$ for node i the PPR $\mathbf{r}$ is

$$\mathbf{r}_i = (\mathbf{I} - \alpha\mathbf{A}\mathbf{\Delta}^{-1})^{-1}(1-\alpha)\mathbf{v} \tag{9.9}$$

from which we obtain the PPR-matrix

$$\mathbf{S} = ((1-\alpha)\mathbf{I})(\mathbf{I} - \alpha\mathbf{A}\mathbf{\Delta}^{-1})^{-1} \tag{9.10}$$

where each row i represents the PPR vector $\mathbf{r}_i$.

One choice to embed $\mathbf{S}$ is to use SVD; however this embedding method is again linear. We can instead consider each row $\mathbf{S}_i$. separately as probability distributions to reach any node from node i. This is the approach of VERSE [TMKM18]. VERSE computes the embedding matrix $\mathbf{Z}$ to preserve the distribution of the similarities node by node.

VERSE optimizes the KL-divergence between the distribution of node i and node j, where the KL-divergence between distributions p and q is defined as

$$D(p||q) = \sum_x p(x) \log\left(\frac{p(x)}{q(x)}\right) \tag{9.11}$$

Typically, with KL-divergence the distribution p is fixed and given as input, while the distribution q is the one to learn. If that is the case, in order to learn p notice that

$$D(p||q) = \sum_x \underbrace{p(x)\log p(x)}_{\text{Entropy of } p} - \underbrace{p(x)\log q(x)}_{\text{Cross entropy}}$$

The first term, the entropy of p is constant if we minimize for q. As such, we can reduce the objective to

$$-\sum_x p(x)\log q(x) \tag{9.12}$$

For each node i we can parametrize $q(i)$ as the similarity one-to-all in embedding space given by softmax of the dot-product of the embeddings

$$sim(i,\cdot) = \frac{\exp(\mathbf{Z}\mathbf{z}_i)}{\sum_{j=1}^n \exp(\mathbf{z}_i^\top \mathbf{z}_j)} \tag{9.13}$$

Notice the the numerator multiplies for the entire embedding matrix $\mathbf{Z}$. The final loss becomes

$$\mathcal{L} = -\sum_{i\in V} \mathbf{S}_i \log sim(v,\cdot) \tag{9.14}$$

Eq. 9.13 can benefit from a strategy similar to negative sampling. Another alternative is to use Noise Contrastive estimation and change the loss function into.

$$\mathcal{L} = \sum_{\substack{i\sim\mathcal{P}\\ j\mathbf{S}_i \text{ sim}}} \left[\log p(D=1|sim(i,j) + s\mathbb{E}_{\tilde{j}\sim Q(i)} \log p(D=0 \mid \text{sim}_{\text{E}}(i,\widetilde{j}))\right] \tag{9.15}$$

Lecture 10 *April 11, 2023*

Graph neural Networks

Instructor: *Davide Mottin*

This lecture describes graph neural networks (GNNs) that abstract the concept of neural networks on graphs. The main intuition behind graph neural networks is the possibility to learn how to propagate information over the neighbors in a non-linear manner. While for graph embeddings the guiding principle is the abstraction (or compression) of pairwise distances among the nodes in the graph, a graph neural network generalizes diffusion processes, such as PageRank. This generalization brings to an important connection between GNNs and spectral filters.

In the first part of the lecture, we draw a connecton between matrix factorization and embeddings. In the second part of the lecture, we study GNNs, more specifically two popular models GCNs and GraphSAGE. Finally, we describe how GCNs could be seen as special cases of filters on the eigenvectors of the graph's Laplacian.

10.1 Graph embeddings as matrix factorization

We first look at graph embeddings from a completely different angle. Let us appreciate one connection that we already noticed in Lecture 9, namely that linear embeddings correspond to SVD of the similarity matrices chosen in the objective. For instance, for adjacency-based similarity, the best linear d-dimensional embedding corresponds to the projection of the dataset into the d directions associated to the top-d singular values, namely if $SVD(\mathbf{A}) = \mathbf{U}\Sigma\mathbf{V}^\top$, then the embedding matrix $\mathbf{Z} = \mathbf{U}_d\Sigma_d$.

But for neural embeddings? It seems legit to ask the question whether there exists a matrix that, once factorized, would be approximately equivalent to run gradient descent over the non-linear objective. The answer to this quesiton lies in a method called NetMF [QDM+18].

10.1.1 NetMF

Let us first describe DeepWalk [PARS14], one of the most popular random-walk embedding methods. Recall that a random-walk approach samples random walks using a certain strategy R. Subsequently, a RW embedding method optimizes the likelihood that a node j is reachable from a random-walk starting from node i. Deepwalk's random walk strategy is to generate random-walks of length L and then generate pairs of co-occurring nodes in a window of size T. The co-occurrences are stored in a multiset $\mathcal{D}$. The algorithm to generate the random walks is the following.

Algorithm 13 Deepwalk Random Walk Strategy

```
for each n = 1, 2, ..., N do
    Pick w_1^n according to probability distribution p(w_1)
    Generate vertex sequence (w_1^n, ..., w_L^n) of length L by random walk on network G
    for each j = 1, 2, ..., L − T do
        for each r = 1, ...T do
            add vertex-context pair (w_j^n, w_{j+r}^n) to multiset D
            add vertex-context pair (w_{j+r}^n, w_j^n) to multiset D
```

The first vertex of the random walk w_1^n is sampled from a prior distribution $p(w_1)$ that in undirected, connected bipartite graphs correspond to $p(i) = d_i/2|E|$.

The question is now. Does this process and the relative transformation through softmax corresponds to a similarity matrix? The answer is positive, as we can represent the similarity among two nodes i, j as

$$\log\left(\frac{(\text{number random walks between i and j})\cdot(\text{number of node/context pairs})}{k(\text{number of occurrences of i})\cdot(\text{number of occurrences of j})}\right) = \log\left(\frac{\#(i,j)|\mathcal{D}|}{k\#(i)\#(j)}\right) \tag{10.1}$$

where k is the number of negative samples. This intuitive formula requires a quite elaborate theoretical understanding. We first partition the set of multiset $\mathcal{D}$ of pairs of nodes co-occurring in walks into two sets $\mathcal{D}_{\vec{r}}$ and $\mathcal{D}_{\overleftarrow{r}}$

$$\mathcal{D}_{\vec{r}} = \{(w,c) : (w,c) \in \mathcal{D}, w = w_j^n, c = w_{j+r}^n\}$$
$$\mathcal{D}_{\overleftarrow{r}} = \{(w,c) : (w,c) \in \mathcal{D}, w = w_{j+r}^n, c = w_j^n\}$$

The then observe that

$$\log\left(\frac{\#(i,j)|\mathcal{D}|}{k\#(i)\#(j)}\right) = log\left(\frac{\frac{\#(i,j)}{|\mathcal{D}|}}{k\frac{\#(i)}{|\mathcal{D}|}\frac{\#(j)}{|\mathcal{D}|}}\right) \tag{10.2}$$

$$\frac{\#(i,j)}{|\mathcal{D}|} = \frac{1}{2T}\sum_{r=1}^{T}\frac{\#(i,j)_{\vec{r}}}{|\mathcal{D}_{\vec{r}}|} + \frac{\#(i,j)_{\overleftarrow{r}}}{|\mathcal{D}_{\overleftarrow{r}}|} \tag{10.3}$$

where the latter observation comes from $\frac{|\mathcal{D}_{\vec{r}}|}{|\mathcal{D}|} = \frac{|\mathcal{D}_{\overleftarrow{r}}|}{|\mathcal{D}|} = \frac{1}{2T}$.

We now observe what happens as the length of the random walk $L \to \infty$. We first define the *random walk matrix* $\mathbf{P} = \mathbf{\Delta}^{-1}\mathbf{A}$. The *volume* of a graph G is the sum of the entries of adjacency matrix $vol(G) = \sum_i \sum_j a_{i,j}$. We now report three important Lemmas from [QDM+18].

$$\frac{\#(i,j)\vec{r}}{|\mathcal{D}\vec{r}|} \xrightarrow{p} \frac{d_i}{\text{vol}(G)}\left(\boldsymbol{P}^r\right)_{i,j} \text{ and } \frac{\#(i,j)\overleftarrow{r}}{|\mathcal{D}\overleftarrow{r}|} \xrightarrow{p} \frac{d_j}{\text{vol}(G)}\left(\boldsymbol{P}^r\right)_{j,i} \tag{10.4}$$

$$\frac{\#(i,j)}{|\mathcal{D}|} \xrightarrow{p} \frac{1}{2T}\sum_{r=1}^{T}\left(\frac{d_i}{\text{vol}(G)}\left(\boldsymbol{P}^r\right)_{i,j} + \frac{d_j}{\text{vol}(G)}\left(\boldsymbol{P}^r\right)_{j,i}\right) \tag{10.5}$$

$$\frac{\#(i,j)|\mathcal{D}|}{\#(w)\cdot\#(c)} \xrightarrow{p} \frac{\text{vol}(G)}{2T}\left(\frac{1}{d_j}\sum_{r=1}^{T}\left(\boldsymbol{P}^r\right)_{i,j} + \frac{1}{d_i}\sum_{r=1}^{T}\left(\boldsymbol{P}^r\right)_{j,i}\right) \tag{10.6}$$

Putting all together we obtain that DeepWalk with infinite length random walks converges to

$$\log\left(\frac{\text{vol}(G)}{k}\left(\frac{1}{T}\sum_{r=1}^{T}\left(\mathbf{\Delta}^{-1}\mathbf{A}\right)^r\right)\mathbf{\Delta}^{-1}\right) \tag{10.7}$$

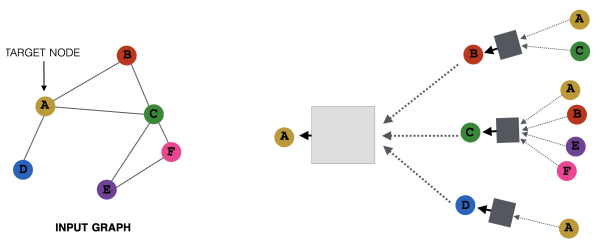
We have found our similarity matrix. Notice that the internal sum reminds very closely to a random walk iteration as the power of the matrix $\mathbf{\Delta}^{-1}\mathbf{A}$ encodes the probability of reaching nodes in r steps through random walking the graph. By factorizing such a similarity with SVD we can obtain the desired result that approximates the embeddings of a random-walk approach.

10.2 Graph neural networks

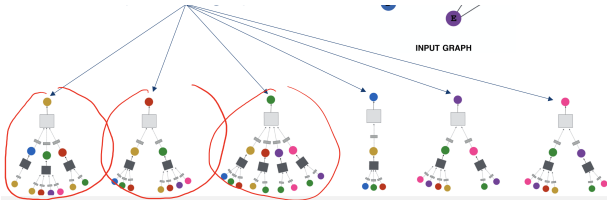
A graph neural network is an architecture that allows nodes to propagate information to the neighbors in a non-linear manner. We have an *attributed graph* $G = (V, E, \mathbf{X})$ with matrix of node features $\mathbf{X} \in \mathbb{R}^{m \times n}$. Such a matrix encodes node attributes such as labels, user profiles, node characteristics, and so on. The main ingredient of a graph neural network is the aggregation function that takes in input the features from the neighbors of a node i and finds an embedding $\mathbf{h}_i$.

10.2.1 Neighborhood aggregation

The main idea is to generate node embeddings based on local neighborhoods of each node, using neural networks to represent the complex relationships of a graph.

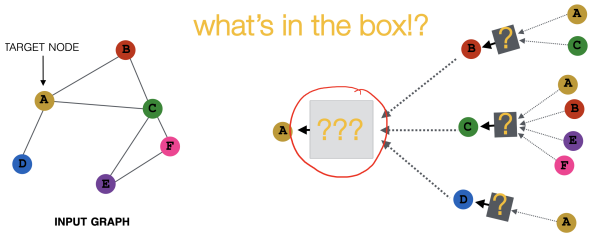


Intuitively, we can think the neighbor of a node as a computational graph reminiscent of a neural network.



The figure above shows the intuition. Nodes are associated embeddings at each layer, and the model can be arbitrarily big. The embeddings in the first layer 0 for a node i correspond to its attributes $\mathbf{z}_i$. The embeddings in the layer l is an aggreagation of the embeddings in layer $l-1$.

The key distinction between different approaches lays in the way they aggregate information across the layers. The simplest approach is to average neighbor information and apply a neural network.



The embedddings of the initial layer 0 are the attributes $\mathbf{z}_i^0 = \mathbf{z}_i$. The l-th layer embedding of i is

$$\mathbf{z}_i^l = \sigma\left(\mathbf{W}_l \sum_{j \in N(i)} \frac{\mathbf{z}_j^{l-1}}{|N(i)|} + \mathbf{B}_l \mathbf{z}_i^{l-1}\right), \forall t > 0 \tag{10.8}$$

Here σ is a non-linear function such as ReLU or tanh. The sum is the average of each neighbour's embeddings compute in layer $l-1$.

Loss function In order to find the embeddings we need to provide a suitable loss function for our optimization. In Eq. 10.8 $\mathbf{W}$ and $\mathbf{B}$ are the trainable weight matrices. After k-layers of neighborhood aggregation, we get output embeddings for each node. Any differentiable loss function studied before would serve the purpose. We optimize the parameters of our model through stochastic gradient descent.

We can also rewrite our model in matrix notation

$$\mathbf{Z}^{l+1} = \sigma(\mathbf{Z}^l \mathbf{W}^l + \tilde{\mathbf{A}} \mathbf{Z}^l \mathbf{W}_1^l) \tag{10.9}$$

Where $\tilde{\mathbf{A}} = \mathbf{\Delta}^{-\frac{1}{2}} \mathbf{A} \mathbf{\Delta}^{-\frac{1}{2}}$. To optimize the parameters, a cross-entropy loss function would work in a supervised setting where we know, for instance, some of the node's labels. An important properties of these models is the ability to share the parameters $\mathbf{W}, \mathbf{B}$ and, as such, allowing for generalization to unseen nodes, edges, or graphs. Depending on the loss function and the embedding matrix, a graph neural network can learn node embeddings, graph embeddings, or clusters.

10.2.2 Graph convolutions networks

Graph convolutional networks (GCNs) [KW16] use a different neighbor aggragation.

$$\mathbf{z}_i^l = \sigma\left(\mathbf{W}_l \sum_{j \in N(i) \cup i} \frac{\mathbf{z}_j^{l-1}}{\sqrt{|N(i)||N(j)|}}\right) \tag{10.10}$$

The main idea is node with a large number of connections should be less important. As opposed to the simple neighborhood in Eq. 10.8, there is no need for the bias term $\mathbf{B}$ and the per-neighbor normalization.

This normalization empirically attains better results and better parameter sharing.We can efficiently compute the embeddings using sparse batch operations

$$\mathbf{Z}^{l+1} = \sigma(\mathbf{\Delta}^{-\frac{1}{2}}\tilde{\mathbf{A}}\mathbf{\Delta}^{-\frac{1}{2}}\mathbf{Z}^{l}\mathbf{W}_{l}) \tag{10.11}$$

Where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and δ is the diagonal degree matrix. The time complexity to propagate the information is $\mathcal{O}(|E|)$.

10.2.3 GraphSAGE

So far we have aggregated the neighbor messages by taking their weighted average; however, this can be generalized further. Instead of fixed aggregation, GraphSAGE [HYL17] use any differentiable function that maps set of vectors to a single vector. So, if we use a generalized aggregation we get

$$\mathbf{z}_i^l = \sigma([\mathbf{W}_l \cdot AGG(\{\mathbf{z}_j^{l-1}, \forall j \in N(i)\}), \mathbf{B}_l\mathbf{z}_i^{l-1}]) \tag{10.12}$$

Here we concatenate self embedding of neighbor embedding. We also use a generalized aggregation. Some variants are

- **Mean:** Take weighted average of neighbours

$$AGG = \sum_{j \in N_(i)} \frac{\mathbf{z}_j^{l-1}}{|N_(i)|} \tag{10.13}$$

- **Pool:** Transform neighbor vectors and apply symmetric vector function

$$AGG = \gamma(\{\mathbf{Q}\mathbf{z}_j^{l-1}, \forall j \in N_(i)\}) \tag{10.14}$$

  Where we take element-wise mean/max

- **LSTM:** Apply LSTM to reshuffled neighbors $\pi(N(i))$

$$AGG = LSTM([\mathbf{z}_j^{l-1}, \forall j \in \pi(N_(i))]) \tag{10.15}$$

Bibliography

[ASN+13] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd international conference on World Wide Web*, pages 37–48, 2013.

[BGLL08] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[CLX15] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pages 891–900, 2015.

[GGMP04] Zoltan Gyongyi, Hector Garcia-Molina, and Jan Pedersen. Combating web spam with trustrank. In *Proceedings of the 30th international conference on very large data bases (VLDB)*, 2004.

[Hav03] Taher H Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *IEEE transactions on knowledge and data engineering*, 15(4):784–796, 2003.

[HYL17] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[Kle99] Jon M Kleinberg. Hubs, authorities, and communities. *ACM computing surveys (CSUR)*, 31(4es):5–es, 1999.

[KW16] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2016.

[New06] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[NJW01] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14, 2001.

[OCP+16] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114, 2016.

[PARS14] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[PDFV05] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *nature*, 435(7043):814–818, 2005.

[QDM+18] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. In *Proceedings of the eleventh ACM international conference on web search and data mining*, pages 459–467, 2018.

[SM00] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[TMKM18] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. Verse: Versatile graph embeddings from similarity measures. In *Proceedings of the 2018 world wide web conference*, pages 539–548, 2018.

[WKRQ+08] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, Philip S Yu, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[YL12] Jaewon Yang and Jure Leskovec. Community-affiliation graph model for overlapping network community detection. In *2012 IEEE 12th international conference on data mining*, pages 1170–1175. IEEE, 2012.

[YL13] Jaewon Yang and Jure Leskovec. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 587–596, 2013.

Part III

Pattern Mining

This module introduces **pattern mining**. Pattern mining refers to a set of algorithms that finds recurring structures in data. These structures can be items, connections, subgraphs, or anything for which we can count frequency. Frequency and structure are task-specific. In our case, we will provide three different tasks:

- **Frequent subgraphs** in graph data, with the purpose of finding structures that repeat in multiple graphs or inside the same large graph.
- **Frequent itemsets and rules** that appear in transactions or item baskets.
- **Contiguous segments** that appear in sequences or time series.

For this module the notation will be introduced in the respective lecture. We will also, more extensively, analyze space and time requirements for each algorithm and propose extensions and optimizations.

➳ Objectives

After finishing this module, you should be able to:

- Describe the purpose of pattern mining and its main tasks.
- Discuss the main applications for pattern mining and what can we do to reduce the time complexity.
- Compare different methods in terms of definition of support and error.
- Categorize pattern mining tasks and techniques by their strengths and weaknesses.
- Apply the main pattern mining techniques and run them over small datasets.
- Generalize the algorithms to other kind of data.

Lecture 11 *April 18, 2023*

Frequent subgraph mining

Instructor: *Davide Mottin*

This lecture introduces frequent subgraphs, that are frequently recurring structures in a large graph or a set of graphs. The problem is traditionally studied in the context of *labeled graphs* since in general graphs, a pattern can only express small structures with little information, such as stars, triangles, and paths.

A *labeled graph* is a triple $G = (V, E, \ell)$, where $\ell : V \cup E \to L$ is a labelling function assigning a label from a set of labels L to each node and each edge.

In order to mine frequent subgraphs, we need a definition of frequency, which is encoded into a *support* measure that return the number of occurrences of the subgraph in the graph. We will see that such a definition has tremendous implications for the algorithm.

We study two different formulations for frequent subgraph mining (FSM). First, we look at the problem of detecting frequent subgraphs in a collection of labeled graphs. Then, we generalize the problem to that of mining subgraphs in a large labelled graph. We will investigate which support measure is more suitable in both casese. Sometimes we refer to patterns instead of subgraphs to avoid confusion with the actual subgraph of a graph.

11.1 FSM in Graph collections

We first look at the simplest scenario, that is mining frequent subgraphs in collections of graphs. The reason why considering graph colections is easier will be clear in Section 11.2.

Definition 11.1.1. *A* graph collection *(a.k.a. a graph database) is a set $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$ of labelled graphs $G_i = (V_i, V_i, \ell_i), i \in [1, n]$.*

Since a subgraph is uniquely identified by the nodes and edges of a certain graph, we need to relax this definition, so that it counts the occurrences of that subgraph in different graphs. To this end, we recall the notion of isomorphism and subgraph isomorphism.

Definition 11.1.2. *A graph $G_1 = (V_1, E_1, \ell_1)$ is* isomorphic *to a graph $G_2 = (V_2, E_2, \ell_2)$ if there exists a bijective function $f : V_1 \to V_2$ such that if $u \in V_1$ then $f(u) \in V_2$ and $\ell_1(u) = \ell_2(f(u))$ and if $(u, v) \in E_1$ then $(f(u), f(v)) \in E_1$ and $\ell_1((u, v)) = \ell_2((f(u), f(v)))$.*

If a graph G_1 is isomorphic to a subgraph of another graph G, we say that G_1 is *subgraph isomorphic* to G. Note that, although isomorphism and subgraph isomorphism are very similar problems, subgraph isomorphism is **NP**-complete, while graph isomorphism is in NP, but we do not know whether it is **NP**-complete. We denote with $G \sqsubseteq G_1$ a subgraph isomorphism from G to G_1.

We can now define the support of a subgraph that provides a measure of frequency of a subgraph in a graph collection.

Definition 11.1.3. *Given a collection of graphs $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$ and a graph G, the support set $\mathcal{D}_G$ is the subset of $\mathcal{D}$, such that G is subgraph isomorphic to every graph in $G' \in \mathcal{D}_G$, i.e., $\mathcal{D}_G = \{G' \in \mathcal{D} | G \sqsubseteq G'\}$. The* support $\sup(G)$ *of G is the proportion $\sigma(G) = \frac{|\mathcal{D}_G|}{|\mathcal{D}|}$.*

The Frequent Subgraph Mining (FSM) problem in collection of graphs $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$ finds all the subgraphs G having at least support *minsup*, i.e., $\sigma(G) \geq minsup$.

Downward closure property. One important property of the support in Def. 11.1.3 is the downward closure. Downward closure is the property under which a support measure decreases by expanding a pattern. In particular in our case $\sup(G) \leq \sup(G')$ if $G' \sqsubseteq G$.

The downward closure property leads to two different family of approaches. The first, called *apriori-based approaches* generate a new subgraph by merging two frequent subgraphs. The second, called *pattern-growth approaches*, expand subgraphs by adding one extra edge at the time. The order in which these edges are added, impact the time.

11.1.1 Apriori-based approaches

The apriori-based approaches stem from the observation that any subgraph of a frequent subgraph must be frequent as well.

FSG [KK04] is one of the first methods that utilizes the apriori principle. The algorithm builds on the idea that a frequent k-subgraph is built on top of frequent $(k-1)$-subgraphs, where k indicates the number of edges. The algorithm simply starts enumerating frequent 1-subgraphs and 2-subgraphs and store them into the set F_1 and F_2, respectively. Similarly, F_k is the set of frequent k-subgraphs. FSG then builds larger k-subgraphs by merging the subgraphs obtained in the previous iteration.

Algorithm 14 FSG

1: $F_1 \leftarrow$ frequent 1-subgraphs with counts
2: $F_2 \leftarrow$ frequent 2-subgraphs with counts
3: $k \leftarrow 3$
4: **while** $F_{k-1} \neq \emptyset$ **do**
5: **Candidate generation:** Get C_k, the set of candidate k-subgraphs from F_{k-1}.
6: **Candidate pruning:** For a candidate to be frequent, each $(k-1)$-subgraph should be frequent
7: **Frequency pruning:** Scan $\mathcal{D}$, count occurrences of subgraphs in C_k
8: $F_k \leftarrow \{c \in C_k | \sigma(c) \geq minsup\}$
9: $k \leftarrow k+1$
10: **return** $F_1 \cup \cdots \cup F_k$

Algorithm 14 iterates over three steps: candidate generation, candidate pruning, and frequency pruning. Each of these steps run isomorphism checks to understand whether some candidate is isomorphic to any of the other candidates. Moreover, in order to compute the support of a subgraph the algorithm runs subgraph isomorphism on all the candidate graphs in C_k. FSG employs smart heuristics to reduce the number of isomorphism tests at each iteration.

Candidate generation Candidate generations generates candidates by merging the frequent $(k-1)$-subgraphs. The main idea is to first detect cores to generate the candidates easily. A core is the maximum common subgraph among two subgraphs. Although generating the core requires to run subgraph isomorphism, in practice (1) one core is common to many subgraphs, (2) the subgraphs are typically small.

Candidate pruning In order to prune the candidates we can check if **every** subgraph of a candidate is also frequent. If not, the candidate is safely pruned. This property requires to split a candidate into smaller subgraphs and check the sets $F_1, ...F_{k-1}$ whether the subgraph is frequent. In theory, this requires to perform isomorphism tests for each of the candidate's subgraph. Again, this would be infeasible. Instead, FSG employs a hashing scheme for graphs.

In order to hash a graph, we require a signature, i.e., a numeric representation, that is unique for each graph. This is possible through *canonical labeling*. Canonical labeling, finds a unique representation of a graph in a smart manner. In particular, FSG employs a canonical scheme that reorders the column and the rows of the adjacency matrix. The string obtained by reading the sequence of node and edge labels in the column of the matrix that is the smallest lexicographic string among all of those obtained by reordering is the canonical code and is unique, i.e., $Code(G) = \min code(M) | M$is adjacency matrix.

⚠ Remark

One could argue that this method effectively solves graph isomorphism in $\mathcal{O}(1)$. However, this is not the case as the canonical labeling is obtained by a, potentially exponential, number of reordering of the adjacency matrix. By reordering, we are computing automorphisms over the graph. In practice, not all the permutations lead to a valid matrix and can be pruned by simple heuristics.

Frequency counting Frequency counting requires to scan the data to find which graphs contain a subgraph. In frequency counting, most of the subgraph isomorphism tests can be avoided with a simple shortcut. Let $TID(G')$, $TID(G'')$ the set of graphs in the collection $\mathcal{D}$ that contain the subgraph G' and G'', respectively. If we merge the two patterns, it follows immediately that $TID(merge(G', G'')) \subseteq$

$TID(G') \cap TID(G'')$. As such all the graphs in $(TID(G') \cup TID(G'')) \setminus (TID(G') \cap TID(G''))$ can be ignored as they will not contribute to the support.

✔ Advantages

- Finds all frequent subgraphs.
- Easy to include heuristics to reduce the computation time.

✘ Disadvantages

- FSG generates the same subgraph a large number of times.
- The canonical labeling scheme is quadratic in the size of the subgraph.
- FSG requires several isomorphism and subgraph isomorphism tests.

11.1.2 Pattern-growth approaches

Pattern-growth approaches follow a different philosophy. Such methods start first with empty subgraphs and progressively expand these subgraphs by adding one edge at the time. The pattern expands always in a certain order to avoid redundancy as much as possible.

gSpan [YH02] is the most famous algorithm in the pattern-growth family. The interested readers should better refer to the gSpan's technical report[1] that includes important details for the understanding of the algorithm. gSpan organizes the patterns in a *prefix-tree*, a data structure for alphabetically ordered strings that store string prefixes as nodes of a tree. However, graphs do not have a predefined order of traversal or, even worse, a unique way of enumerating the nodes. This modus operandi should remind you of the canonical labeling we introduced earlier. However, before describing the gSpan's canonical labeling let us introduce the main components of the gSpan's algorithm.

To create a prefix-tree of non-repeating patterns gSpan:

1. Defines a traversal over the nodes in the graphs;
2. Identifies in advance which nodes of a subgraph to expand;
3. Defines a total ordering over the subgraphs, so as to detect duplicate patterns.

Node traversal. gSpan traverses the subgraphs in a Depth-First-Search (DFS) manner, that means that the next node v_j to visit after v_i is the first in the, potentially sorted, list of neighbors $N(i)$. A DFS vist defines a spanning-tree over the nodes in the subgraph. The edges in the spanning tree are called **forward edges**; the remaining edges are called **backward edges**. In the beginning of the DFS a counter assigns a unique identifier to each node. The counter represents the discovery "time" of the node. The counter increases for every visited node.

Rightmost expansion. Since a DFS defines a specific traversal and the order of the visit is marked by a counter, there is only one way to add an edge to the subgraph, namely expanding from the **rightmost path**. The rightmost path is the path from the root of the spanning tree to the leaf with the highest identifier.

Total ordering. The main innovation of gSpan is to treat canonical labelling as a DFS-visit and, thus, ensure a reduced number of operations and redundant subgraphs. This new canonical labelling, called DFS codes is what makes gSpan one of the most popular algorithms for FSM.

11.1.2.1 DFS Codes

DFS codes is a representation of a graph that is comparable among other DFS codes. Such a comparison is a total ordering for which a minimum exists. The **minimum DFS-code** is a canonical labeling of a graph. Recall that we traverse the graph in a DFS manner and store the discovery time of each node. We define a DFS-edge as follows.

[1] `https://sites.cs.ucsb.edu/~xyan/papers/gSpan.pdf`

Definition 11.1.4. *A DFS-edge is a tuple* $(i, j, \ell(i), \ell((i,j)), \ell(j))$ *where* i, j *are identifiers given by the DFS traversal order,* $\ell(i), \ell((i,j)), \ell(j)$ *are the labels of node* i*, edge* (i,j)*, and node* j*.*

Having the definition of DFS-edge, a DFS-code for a graph $G = (V, E, \ell)$ is a sequence $\langle e_1, ..., e_t \rangle$ where each e_i is a DFS-edge and $t = |E|$.

Valid DFS-codes We define a specific precedence order $\prec$ on DGS-edges corresponding to the DFS traversal. Given two DFS-edges e_1 connecting nodes with discovery time (i_1, j_1) and e_2 connecting nodes with discovery time (i_2, j_2). A DFS-code is **valid** if it complies to the following rules.

1. $i_1 = i_2$ and $j_1 < j_2 \implies e_1 \prec e_2$ (backward edges ordering)
2. $i_1 < j_1$ and $j_1 = i_2 \implies e_1 \prec e_2$ (forward edges ordering)
3. $e_1 \prec e_2$ and $e_2 \prec e_3 \implies e_1 \prec e_3$ (transitive property)

An example of a valid DFS-cde is shown in Figure 11.1.

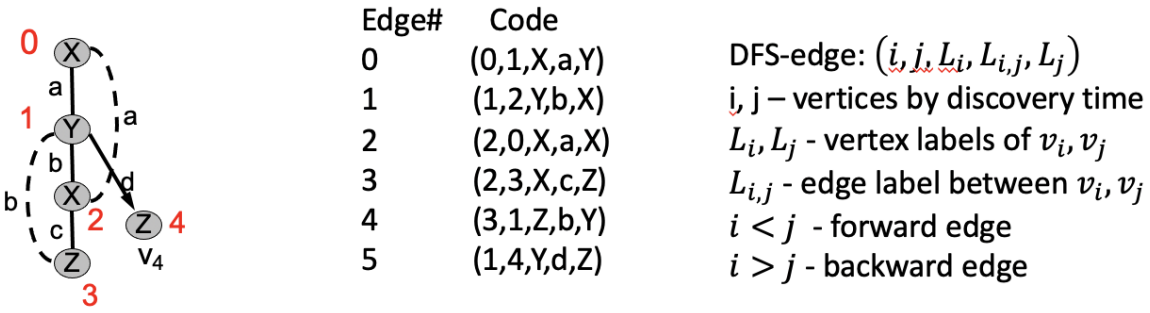


Figure 11.1: A valid DFS-code (right) for a small graph (left).

Total ordering and minimum codes A graph can have multiple valid DFS-codes depending on the DFS traversal order. gSpan further defines a total ordering over the DFS-codes so to define a minimum DFS-code for a graph. The minimum DFS-code is unique and, therefore, a canonical labeling of the graph. This implies that two graphs are isomorphic if and only if they have the same DFS-code.

Let $\alpha = (a_0, a_1, \ldots a_m), \beta = (b_0, b_1, \ldots, b_n)$ two DFS-codes. We say that $\alpha \prec \beta$ (α precedes β) iff **either** of the following conditions are true:

- $\exists t, 0 \leq t \leq min(n, m)$ such that $a_k = b_k$ for $k < t$ and $a_t \prec_e b_t$
- $a_k = b_k$ for $0 \leq k \leq m$ and $n \geq m$

In simpler words $\alpha \prec \beta$ if either α and β are equal up to some point t and the next corresponding DFS-edges is such that $a_t \prec_e b_t$, OR, α and β are the same in terms of DFS-edges but β contains more edges.

In order to complete our reasoning, we need to be able to compare DFS-edges among different codes. In other words, we need to define the relationship $\prec_e$. Let $a_t = (i_a, j_a, L_{i_a}, L_{i_a,j_a}, L_{j_a})$ be two DFS-edges. We say that $a_t \prec_b t$ if

1. Both are forward edges (hence both arrive at new node indexed as $j_a = j_b$) and either
   - $i_b < i_a$ (edge starts from later visited vertex) OR
   - $i_b = i_a$ and the labels of a are lexicographically less than labels of b
2. Both are backward edges (hence both start from node indexed as $i_a = i_b$ and either
   - $j_a < j_b$ (edge connected to an earlier discovered vertex) OR
   - $j_a = j_b$ and edge label of a is lexicographically less than the one of b
3. a_t is backward and b_t is forward

The above enforce a total order on DFS-codes and, as such, the min-DFS(G) of a graph G is a canonical labeling of G.

DFS-code tree The DFS-codes allows also to organize the data into a DFS-tree that is a prefix tree. This is because the codes have a precedence order. If min-DFS(G_1) = $\{a_0, a_1, \ldots, a_n\}$ and min-DFS(G_2) = $\{a_0, a_1, \ldots, a_n, b)\}$, then G_1 is parent of G_2, since min-DFS(G_1) is a prefix of min-DFS(G_2).

⚠ Remark

Recall that, due to DFS-traversal, a subgraph grows only on the rightmost path

Given a parent-child relationship defined above, we can construct a DFS Tree. An interesting property of the DFS-tree is that an in-order traversal follows DFS lexicographic order.

gSpan algorithm Constructs the DFS-tree by progressively expanding the subgraphs.

Algorithm 15 gSpan

Input: Collection of graphs $\mathcal{D}$, minimum support *minsup*
Output: Frequent subgraph set S

```
Let S be frequent one-edge subgraphs in D using DFS code.
Sort S in lexicographic order
N ← S
for each n ∈ N do
    gSpanExtend(D, n, minsup, S)
return S

function gSpanExtend(D, n, minsup, S)
// Checking minimality of DFS-codes is computationally expensive
    if n not minimal then return
    else
        S ← S ∪ n
        for each Rightmost expansion e of n do
            if σ(e) ≥ minsup then
                gSpanExtend(D, e, minsup, S)
```

✔ Advantages

- Finds all frequent subgraphs.
- Generates less redundant subgraphs.

✘ Disadvantages

- Works only in collections of graphs.
- Generally slow for low *minsup*.

11.2 Frequent subgraph mining on a single graph

One of the main questions is whether gSpan or methods that work with collections of graphs also work for mining subgraphs in a single large graph. The main issue is that such methods rely on the downclosure property of the support. Does the same property holds in a large graph? The answer is no. If the support is defined as the number of times a specific subgraph appears, the downward property does not hold anymore as bigger subgraphs can have a larger support.

We defined a support measure to be *admissible* if for any graph P and any subgraph $Q \sqsubseteq P$, the support of P is not larger than the support of Q. In the last few years, a number of admissible support measures have appeared. These include

- Maximum independent set support (MIS): Based on maximum number of non-overlapping matches

- Harmful overlap support (HO): Based on number of patterns for which no multi-node subgraph is identical
- Minimum image-based support (MNI): Based on the number of times a node in the pattern is mapped into a distinct node in the graph

11.2.1 MIS support

The Maximum independent set support σ_{MIS} counts how many times a pattern maps to a subgraph that does not overlap to any other subgraphs. The computation of MIS is as follows

1. Construct an overlap graph $G_O = (V_O, E_O)$, in which the set of nodes V_O is the set of matches of a pattern P into a graph G and $E_O = \{(f_1, f_2) | f_1, f_2 \in V_O \vee f_1 \neq f_2 \vee f_1 \sqcap f_2 \neq \emptyset\}$, i.e., E_O has an an edge among each pair of overlapping matches.
2. Compute the σ_{MIS} as the size of the maximum independent set of nodes in the overlap graph.

Recall that finding the maximum independent set is **NP**-hard.

11.2.2 Harmful Overlap support

The HO support is less restrictive than MIS support as it considers overlap only those matchings of a pattern that share one ore more edges. Two matching subraphs are connected through a node are not considered overlapping.

11.2.3 NMI support

The minimum image-based support is even less restrictive as it considers the minimum number of times a node of the pattern matches a node in the graph.

Definition 11.2.1. *Let $f_1, \ldots, f_m$ be the set of isomorphisms of a pattern $P : \langle V_P, E_P, \ell_P \rangle$ in a graph G. Let $F(v) = |\{f_1(v), \ldots, f_m(v)\}|$ be the number of distinct mappings of a node $v \in V_P$ to a node in G by functions $f_1, \ldots, f_m$. The Minimum Image-based support (MNI)*

$$\sigma_{MNI}(P) \text{ of } P \text{ in } G \text{ is } \sigma_{MNI}(P) = \min \{F(v), v \in V_P\}$$

Since MNI considers one node at the time, given the matching of pattern in the graph, NMI support can be computed in polynomial time.

11.2.4 Approaches for large graphs

With an anti-monotone support measure, we can apply any previous support measure (such as gSPAN) just changing how the support is computed and leaving the rest unchanged. Such approaches are called grow-and-store

- Add all frequent edges to the list of frequent subgraphs
- Extend frequent subgraphs to larger candidates
- Compute candidate frequency (find all occurrences)
- Repeat step 2 until no more frequent subgraphs is found

One of the methods that exploits and optimizes gSpan is GraMi [EASK14].

Lecture 12 *April 25, 2023*

Frequent Itemsets and Association Rules

Instructor: *Davide Mottin*

This lecture introduces frequent itemsets and association rules, one of the earliest problems in data mining. This problem arises in supermarkets, where people often buy products together. The purchased items reveal important implicit patterns that help the store organizing the shelves in a more logical manner, so as to increase the revenue and the customer satisfaction.

The frequent itemsets problem aims to find a sets of items that are bought together. We will see how these itemsets can be efficiently mined by exploiting the apriori principle of the support measure. A specialization of the problem mines rules, that describe whether a customer will likely buy some items knowing that their basket already contains some initial items.

We will look at some popular algorithms, such as the Apriori algorithm and FP-growth that improves the Apriori's efficiency. We will also draw some important considerations on how to trade memory and time.

12.1 Frequent itemsets mining

Frequent itemsets are collections of objects that appear a minimum number of times in the baskets of the customers. In our shop, we have a set of d items $\mathcal{I}$ and a set of n transaction identifiers $\mathcal{T}$, each of them representing the items purchased by the customers. Our dataset $\mathcal{D}$ is a set of pairs $(i, t) \in \mathcal{I} \times \mathcal{T}$. Note that a basket is a transaction-set $t(\mathcal{D}) = \{i_1, ..., i_{|t|}\}$. An *itemset* is a subset of $I \subseteq 2^{\mathcal{I}}$ items, where $2^{\mathcal{I}}$ is the powerset of all subsets of $\mathcal{I}$.

Definition 12.1.1. *For an itemset $I \subseteq 2^{\mathcal{I}}$ we define the support $\sigma(I)$ as the number of transactions that contain itemset I, i.e., $\sigma(I) = |\{t \in \mathcal{T} | I \subseteq t(\mathcal{D})\}|$.*

The **frequent itemset mining** problem finds all the itemsets $I \subseteq 2^I$ that have a support $\sigma(I) \geq$ *minsup*. The support can be expressed as a number or a percentage, similarly to frequent subgraph mining in Lecture 11.

It is clear that if we have d items, we have potentially 2^d itemsets. Itemsets are typically represented in a lattice structure, in which the root is empty, and each node is the union of itemsets in the parent nodes. The question is *how do we find frequent itemsets in an efficient manner?*

Naïve solutions One way to mine all frequent patterns is to count each itemset individually. This requires to construct all the 2^d itemsets and compute the frequency for each of them, reaching $\mathcal{O}(2^d nw)$ time complexity and $\mathcal{O}(d)$ space complexity, where w is the size of the largest transaction. On the other hand we can generate all the possible candidate itemsets for each transaction. In that case, we will obtain a $\mathcal{O}(n2^w)$ time and $\left(2^d\right)$ space. Clearly, these two solutions are impractical.

One of the major bottlenecks for frequent itemset mining is memory, as the algorithm has to keep a large list of candidates to check. In the naïve solutions above, memory is a critical asset. We now consider a clever algorithm that utilizes the apriori property of the support, hence the name *Apriori algorithm*.

12.1.1 The apriori algorithm

The Apriori algorithm [AS+94] for frequent itemset mining and rule mining is an elegant algorithm that exploits the apriori property of the support. The Apriori property states that, if $X, Y \in 2^{\mathcal{I}}$ are two itemsets such that $X \subseteq Y$, the support $\sigma(X) \geq \sigma(Y)$. In other words, the support of Y does not exceed that of X. Equivalently, we can say that the support is anti-monotone.

The Apriori algorithm exploits the Apriori property by means of an iterative process that alternates **frequent itemset generation**, and **candidate generation** as shown in Algorithm 16. Let C_k be the candidate itemsets of size k, let L_k be the frequent itemsets of size k. The Apriori algorithm starts from the intuition that, instead of generating all the itemsets, the candidates of size k should only contain frequent itemsets of size $k - 1$.

Algorithm 16 Apriori algorithm

Input: Data $\mathcal{D}$, items $\mathcal{I}$, minimum support *minsup*

1: $k \leftarrow 1$
2: $C_1 \leftarrow \mathcal{I}$ ▷ C_1 contains all items
3: **while** $C_k \neq \emptyset$ **do**
   // Frequent itemset generation
4: Scan DB to find which itemsets in C_k are frequent, put them into L_k
   // Candidate generation
5: Use L_k to generate a collection of candidate itemsets C_{k+1} of size $k+1$
6: $k \leftarrow k+1$

Candidate generation The candidate generation (Line 5) constructs the candidate itemsets of size $k+1$ by combining frequent itemsets of size k. For simplicity, we assume that all itemsets are ordered, such as lexicographically. We also assume that itemsets in L_k are ordered. The main idea is that we can generate candidates of size $k+1$ by joining two itemsets of size k that share the first $k-1$ items. In a database terms, we are computing a self-join between the list of candidates of size k using the first $k-1$ columns as joining attributes. Once a new candidate is generated we can check whether all the ordered subsets are also frequent, i.e., they are in the set L_k. This ensure the correct application of the Apriori principle. The generation of candidates C_{k+1} proceeds as follows.

1. **Self-join** L_k: Create set C_{k+1} by joining frequent k-itemsets that share the first $k-1$ items.
2. **Pruning step:** Remove a C_{k+1} candidate if a k-subset is not frequent.

Frequent itemsets generation One of the fundamental steps of the Apriori algorithm is the counting of the support for each of the candidates C_k. To expedite the computation Apriori uses a hashmap, that contains a counter for each cndidate itemset in the data. In such a structure, ecah itemset is a key and the value is the support, initially set to 0. Every time the itemset is found in the data, the support counter increases by 1.

Is there a better hash structure rather than a hash-map? We can employ a simple hash-tree in which the nodes at level l split via a hash function on the $l-th$ element of the (ordered) itemset. Additionally, the tree organizes the candidates in a prefix-like manner, substantially reducing the size and the number of possible candidates to check in each transaction.

Itemsets of size $k=2$ In practice, for a typical shop-basket data the number of $k=2$ items is by far the largest among other k values. As such, it is important to devise efficient, in-memory optimizations. There are two main approaches, (1) counting all pairs using a triangular matrix, or (2) keeping a table of triples $[i,j,c]$ where the count of the pair of items (i_i, i_j) is c. Clearly, the approach (1) requires 4 bytes per pair, while the second approach requires 12 bytes per pair but only for those pairs with positive count. While it is not clear which of the two approaches is superior, empirically, since the second approach requires $4 * d(d-1)/2 \approx 2 * d^2$ bytes, the second is favorable in case 1/3 of the possible pairs occur.

12.1.1.1 Factors affecting the complexity

The Apriori algorithm is a heuristic method, whose runtime depends on a multitude of factors, such as

- Minimum support threshold (the lower the more we have to deal with)
- Dimensionality of the dataset
- Size of Database (since Apriori makes multiple passes)
- Average transaction width. It may increase max length of frequent itemsets and traversals of hash tree

12.2 Association rule mining

Association rules are a refinement of frequent itemesets, in which an itemset is frequent conditionally to another itemset. For example, $\{a,b\} \implies \{c,d,e\}$ means that when items A, B occur, also C, D, E occur.

Definition 12.2.1. *More formally, an association rule is an implication of the form* $X \implies Y$ *where* X, Y *are itemsets.*

To find significant rules, we introduce the *confidence* γ of a rule $X \implies Y$ as the number of times Y appears in transactions that contain X, i.e. $\gamma(X \implies Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}$. The **association rule mining** task finds all rules at least with support $minsup$ and confidence $minconf$. The algorithm for mining association rules resembles the original apriori, with an additional step to generate the rules. after the computation of the frequent itemsets. The algorithm proceeds as follows.

- **Frequent itemset generation:** Generate all itemsets whose support is greater than $minsup$
- **Rule generation:** Generate rules with confidence at least $minconf$ for each frequent itemset, where a rule is a partitioning of a frequent itemset into left-hand-side (LHS) and right-and-side (RHS).

So the pending issue for new algorithm is *how do we generate such rules?*

12.2.1 Rule generation

The naïve way to generate the rules is to enumerate all the LHS and RHS. The process takes for each frequent itemset S all subsets $L \subset S$ and generates rules of the form $S - L \implies L$ that satisfy minimum confidence.

Example 1: Take for example the frequent itemset $S = \{A, B, C, D\}$. Then the candidate rules are $BCD \implies A$, $ACD \implies B$, $ABD \implies C$, $ABC \implies D$, $CD \implies AB$, $BD \implies AC$, $BC \implies AD$, $AD \implies BC$, $AB \implies CD$, $AC \implies BD$, $D \implies ABC$, $C \implies ABD$, $B \implies ACD$, $A \implies BCD$. Unfortunately, this naïve process takes $2^{|S|}$ and ignores properties on the confidence.

But *does confidence possess the desired Apriori property?* Not in the strict sense, or at least, adding an extra item to the rule, changes arbitrarily the confidence. Luckily, the Apriori property holds for rules generated from the same itemsets. In particular, the confidence decreases if we "move" items from the LHS to the RHS as the support in the denominator of the confidence increases. As such, if a rule $X \implies Y$ from the same item has confidence $\leq minconf$ all rules in which the RHS is a superset of Y also have low confidence and can be pruned. Equipped with this knowledge, we generate a candidate rule by merging two rules (on the same itemset) that share the same prefix on RHS. This process is essentially the Apriori algorithm on RHS.

Example 2: Given two rules $CD \implies AB$ and $BD \implies AC$ their merge produces $D \implies ABC$. We prune rule $D \implies ABC$, if its subset $AD \implies BC$ does not have high confidence.

12.3 FP-Tree and FP-Growth

We have seen the Apriori algorithm for generating frequent itemsets. We now turn our attention to FP-Growth [HPY00], a smarter yet more intricate algorithm, that exploits a tree-structure called FP-tree.

FP-tree construction An *FP-tree* is a prefix-tree (or a trie) that contains a compressed representation of the transaction database. Each transaction in the FP-tree is a, potentially overlapping, path in the tree. The first step before constructing the tree is sorting the itemsets to ensure the uniqueness of prefixes.

1. The construction of the FP-tree starts from an empty node. Each node has a label corresponding to an item and a counter, corresponding to the number of transactions in the paths containing the item.
2. For each transaction follow the prefix-path and increase the counters on the way.
3. If, at some point we reach a leaf node we expand the path adding nodes with counter 1.
4. Add a pointer between nodes that refer to the *same item.*

We add a header table corresponding to the first (from the left of the tree) correspondence of the item as in Figure 12.1

The final size of the tree depends on the redundancy in the transactions. In the best case, all transactions are the same and the FP-tree is a path. In the worst case, all the transactions are different and the size of the tree is bigger than that of the data.

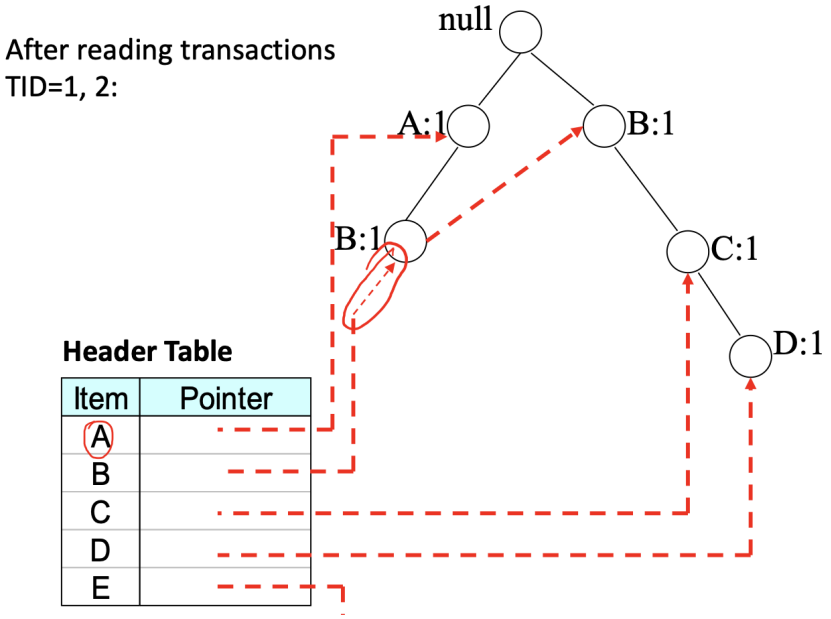


Figure 12.1: Header table for FP-tree

FP-Growth is an algorithm that takes the FP-tree as input and performs a divide-and-conquer strategy to enumerate all the frequent itemsets. The recipe for FP-growth is the following recursive algorithm.

Algorithm 17 Itemsets mining with FP-Tree

Output: The FP-tree, minimum support *minsup*
Input: All frequent itemsets and their support

1: Proceed in divide-and-conque manner
2: **for each** Items in $i \in \mathcal{I}$ **do**
3: Consider candidate itemsets C ending with i
4: Recursion on C

The meta-algorithm above works bottom-up. This means that for each possible last item, consider itemsets with last items one of items preceding it in the ordering, e.g for E, consider all itemsets with last item $D, C; B, A$, this way we get all itemsets ending at DE, CE, BE, AE. We can use the FP-tree to check recursively whether an itemset is frequent and prune the possible candidates in advance. Why are suffixes so convenient in an FP-tree?

⚠ Remark

The items are lexicographically ordered and the header-table contains the first occurrence of an item.

Example 1: Let us look to the FP-tree in Figure 12.1. If we want to find all the itemsets ending in B we just need to look at the header-table for B, the pointers of all the transactions containing B and considering **only** the the subtree growing from the nodes with label "B" up-wards to the root.

The FP-Growth algorithm is depicted in Algorithm 18

Observe that at each recursive step, we solve a subproblem of finding a conditioanl FP-tree, computing new support and pruning nodes. The computation of the support of the suffix entails summing the counters for the nodes with such suffice. If the support of the items is less than *minsup* the node is pruned.

Algorithm 18 FP-Growth

Output: The FP-tree, minimum support *minsup*
Input: All frequent itemsets and their support

```
1: Proceed in divide-and-conque manner
2: for each Suffix X do
   // Phase 1:
3:     Construct a prefix-tree for X, compute support using header table and pointers
   // Phase 2:
4:     if X is frequent then
   // Construct conditional FP-tree for X
5:         Recompute support
6:         Prune infrequent items
7:         Prune leaves and recur
```

⚠ Remark

Note, however, that the items counter of the items need to be updated as the (global) FP-tree contains the counters for all transactions, including those not ending with the considered suffix. The counters propagates bottom-up from the leaves to the root.

Once the counters are updated, if any of the labels have support less than *minsup* the node is pruned and the subtree of the node is attached to the node's father. After pruning, the suffix is removed from the tree and the computation proceeds recursively bottom-up.

✔ Advantages

- FP-tree is typically an efficient algorithm.
- Allows to compute support along with the frequent items.

✘ Disadvantages

- The efficiency of the algorithm depends on the compaction factor.
- If the tree is "bushy" the number of subproblems to solve is large.

Lecture 13 *May 2, 2023*

Sequence Segmentation and Similarities

Instructor: *Davide Mottin*

This lecture explains how to segment a sequence of points with an efficient algorithms and how to find similar documents in a linear manner.

The first algorithm is a result from the 60's that shows how clustering (i.e., summarizing) a sequence to form k partitions can be efficiently solved with a dynamic programming algorithm from Bellman.

The second algorithm is an approximation that allows to compute pairwise distances avoiding $\mathcal{O}(n^2)$ comparisons. The idea is to use efficient signatures (min-hashing) and hash the signatures in bands so as similar items will likely end up in the same bucket.

13.1 Sequence Segmentation

Data in which the points arrive in a certain order has a number of applications such as estimating the price of a stock over time, predicting the next query in a search engine, weather forecasting, DNA anomalies and so on. In its simplest form, a sequence is a vector $(x_1, ..., x_n)$ were each $x_i \in \mathbb{R}$. In this case we refer to one-dimensional sequences, or time series. Similarly, a multidimensional time series is a sequence of d-dimensional points. One of the earliest problems for sequences is **sequence segmentation**: Given a set sequence $(x_1, ..., x_n)$ find k contiguous segments that are as homogeneous as possible.

The sequence segmentation problem requires a notion of homogeneity similar to the one of clustering. However, points in the segments should be subsequences rendering the problem different from that of clustering. A sequence is *homogeneous* if its points have a small deviation from the mean. Let $\mathcal{D} = (\mathbf{x}_1, ..., \mathbf{x}_n)$ a sequence of n d-dimensional points $\mathbf{x} \in \mathbb{R}^d$.

Definition 13.1.1. *A k-segmentation S is a partition of $\mathcal{D}$ into k **contiguous** segments $(s_1, ..., s_k)$. Each segment s_i is represented by a d-dimensional **mean vector** $\boldsymbol{\mu}_i = \frac{1}{|s|} \sum_{\mathbf{x} \in s} \mathbf{x}$.*

The mean vector has the same role of the centroid for k-means clustering. The error $E(S)$ of a k-segmentation S is the error of replacing each individual point with the means. Similar to k-means, one of the most common errors is **sum of squares error (SSE)**

$$E(S) = \sum_{s \in S} \sum_{\mathbf{x} \in S} (\mathbf{x} - \boldsymbol{\mu}_s)^2 \tag{13.1}$$

Problem 1 (k-segmentation problem)**.** *Given a sequcence $\mathcal{D} = (\mathbf{x}_1, ..., \mathbf{x}_n)$ of length n, find a set of k segments $S = (s_1, ..., s_k)$ of $\mathcal{D}$ such that the SSE error is minimized.*

Although formally very close to k-means, the constraint that the segments are sequence of points makes the problem easier than k-means. To see why, we first observe that a k-segmentation S is equally defined by $k + 1$ boundary points $b_0, b_1, ..., b_k$ that represent the first and the last index of the point in each segment. Note that $b_0 = 1$ and $b_k = n$.

13.1.1 A dynamic programming solution

Surprisingly, a dynamic programming solution from Bellman (1961) [Bel61] solves the k-segmentation problem in polynomial time. The idea is to build the solution bottom-up solving progressively larger instances of the problem. A dynamic programming solution requires the property of optimal subproblem that states that the solution on a subset of the input is also optimal.

We denote $D[1, i]$ the subsequence $\mathbf{x}_1, ..., \mathbf{x}_i$ with $i \leq n$. $E(i, s)$ is the error of the optimal segmentation of the subsequence $D[1, i]$ with s segments with $s \leq k$. The dynamic programming algorithm follows the recursion

$$E(i, s) = \min_{s \leq j \leq i-1} \left(E(j, s-1) + \sum_{j+1 \leq t \leq i} (\mathbf{t} - \boldsymbol{\mu}_{j+1,i})^2 \right) \tag{13.2}$$

In practice we can use a two-dimensional table $A[1...k, 1...n]$ that stores in position s, i the value $A[s, i] = E(i, s)$.

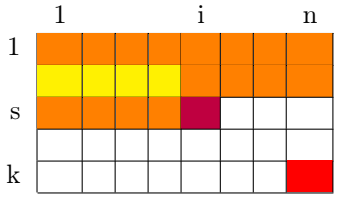


Table 13.1: k-segmentation table; cell $A[s, i]$ contains $E(i, s)$ while the red cell $A[k, n]$ contains the error of the optimal k-segmentation.

The dynamic programming algorithm works as follows.

Algorithm 19 Bellman algorithm

Input: Sequence $\mathcal{D}$, k, error $E(\cdot)$

for each $i = 1...n$ **do** ▷ Initialize first row
    $A[1, i] = E(D[1...i])$ ▷ Error when everything is in one cluster
for each $s = 1...k$ **do** ▷ Initialize diagonal
    $A[s, s] = 0$ ▷ Error when each point is in its own cluster
for each $s = 2...n$ **do**
    for each $i = s + 1ton$ **do**
        $A[k, i] = \min_{j<i}(A[s - 1, j] + E(D[j + 1....i]))$

⚠ Remark

Note that to recover the actual segmentation, the matrix A should store also the the values j minimizing the error.

Algorithm Complexity. The algorithm needs to fill $\mathcal{O}(nk)$ table. Additionally each cell computes the minimum, which in the worst case takes $\mathcal{O}(n)$ cells to check, each of it computes the error in $\mathcal{O}(n)$, bringing the total complexity to $\mathcal{O}(n^3k)$ in the naïve version. However, the errors per cell can be further optimized by noticing that $\sum_{j+1\leq t\leq i}\left(\mathbf{t}-\boldsymbol{\mu}_{[j+1,i]}\right)^2 = \sum_{j+1\leq t\leq i}\mathbf{t}^2 - \frac{1}{i-j}\left(\sum_{j+1\leq t\leq i}\mathbf{t}\right)^2$ that requires the repeated computation of $\mathbf{t}^2$ and $\mathbf{t}$. These two terms can be easily precomputed for all $i = 1...n$. As such, the algorithm complexity is $\mathcal{O}(n^2k)$.

Heuristics. We can further reduce the time by considering some heuristics instead of the optimal algorithm. These heuristics reduce the computational burden by using greedy approaches (top-down or bottom-up) or local search that assigns border points randomly and move them in a way to reduce the errors. The total time is $\mathcal{O}(nk)$.

13.2 Finding similar points efficiently

Finding similar items typically requires comparing each point in the data with any other, raising the overall compelxity to $\mathcal{O}(n^2)$. Even assuming that the computation of the similarity is negligible, we would incur a quadratic cost that might be prohibitive for large datasets. This problem tends to exacerbate in high-dimensional spaces. The problem of finding similar points is the following

Problem 2. *Given a dataset $\mathcal{D} = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ of d-dimensional data points, find all the pairs of data points $(\mathbf{x}_i, \mathbf{x}_j)$ such that $d(\mathbf{x}_i, \mathbf{x}_j) \leq s$ for some distance function d.*

Is there a way to avoid the $\mathcal{O}(n^2)$ comparisons? Yes, with Locality-Sensitive Hashing (LSH). LSH allows for $\mathcal{O}(n)$ comparisons instead of $\mathcal{O}(n^2)$. The goal of LSH is to detect similar (or nearly duplicate) items in a high-dimensional space.

Although the LSH method is general enough to accommodate most distance measures, we concentrate on a popular measure for sets, called Jaccard distance/similarity.

Definition 13.2.1. *The Jaccard similarity of two sets is the size of their intersection* $sim(C_1, C_2) = |C_1 \cap C_2|/|C_1 \cup C_2|$. *The Jaccard distance is then* $d(C_1, C_2) = 1 - |C_1 \cap C_2|/|C_1 \cup C_2|$.

Moreover, we focus on a specific application, whether our data points are documents expressed as sequences of words. Clearly, a large number of documents do not fit into memory and, as such we need a representation of each document that allows for preserving the distances while considerably reducing the memory footprint. In order to achieve such a goal, we will follow a three step approach:

- **Shingling:** Convert documents to sets
- **Min-hashing**: Convert large sets to short signatures, while preserving similarity
- **Locality-sensitive hashing:** Focus on pairs of signatures likely to be from similar documents

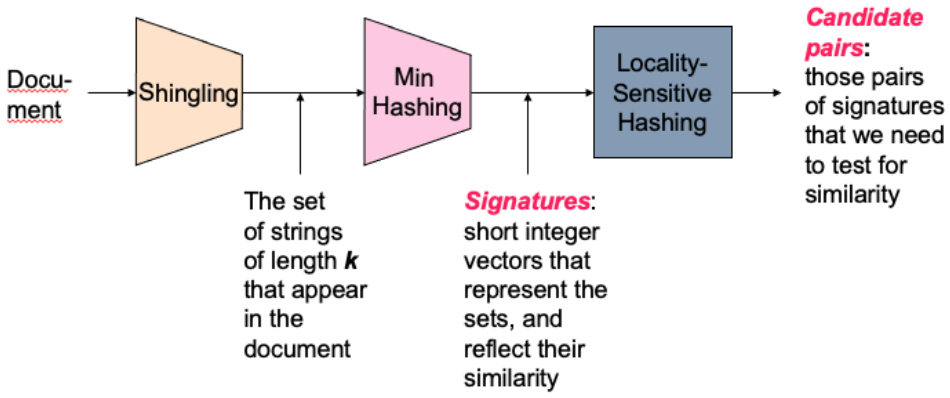


Figure 13.1: Our pipeline to find similar items efficiently

13.2.1 Shingling

One first challenge when dealing with documents is the large vocabulary of words and the sequential nature of text. However, if the vocabulary is large, we would return a large number of sparse vectors with, potentially, low similarity. The solution is to represent the documents as set of *shingles*.

Definition 13.2.2. *A k-shingle (a.k.a. k-gram) is a set of k tokens that appear in the document.*

For our scope, a token is a set of characters. For instance, the set of 2-shingles for document $D_1 = abcab$ is $S(D_1)$=$\{ab, bc, ca\}$.

Long shingles can be easily compressed by hashing; documents are compared through the hash values. Finally, each document is a binary vector in the space of k-shingle in which a shingle is a dimension.

⚠ Remark

We assume that documents have a large number of common shingles if they are similar. However, k is a critical choice to compare documents. A small k leads to a large number of similar documents. As a rule of thumb $k = 5$ (respectively, $k = 10$) is a reasonable choice for short (respectively, large) documents.

Clearly, shingling does not solve the problem of a large number of similarity computation. In the next sections, we will examine methods to reduce the size of the documents representation and the number of comparisons.

13.2.2 Minhash / LSH

So far, we have represented a document into a set of shingles and consequently as a binary vector, where each dimension is a shingle and a 1 in position i of document j indicates that the document contains the shingle i. In this way, we can represent our dataset as a matrix where each row represents a shingle and each column a document. Computing the Jaccard similarity in such a matrix relies on simple bitwise AND, intersection, and OR, for union, operations. Still the problem of comparing two documents remains unsolved even with faster bitwise representations.

The technique that allows for faster comparison is called *Min-Hashing*. Min-Hashing converts the binary representation into small "signatures" through hashing. The key idea is to hash each column to a small signature, $h(C)$ such that

- $h(C)$ is small enough that the signature fits in RAM
- $sim(C_1, C_2)$ is the same as the similarity of signatures $h(C_1)$ and $h(C_2)$.

As such, we seek to find a hash function h such that $sim(C_1, C_2)$ is high, then with high prob the hashes are equal, otherwise with high probability they are not equal $h(C_1) \neq h(C_2)$. For the Jaccard similarity Min-Hashing is such a hash function.

Let us take a random permutation π of the rows of the boolean shingles-document matrix.

Definition 13.2.3. *The hashing function $h_\pi(C)$ returns the index of the first (in the permuted order π) row in which column C has value* 1.

$$h_\pi(C) = \min_\pi \pi(C) \tag{13.3}$$

Clearly, one hashing function is a coarse approximation of a set. In practice, we use several independent hash functions (permutations) to create a signature of a column.

Theorem 13.2.1. *Min-hash preserves the property that $P[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$*

Proof. Let X is a document represented as a set of shingles, and $y \in X$ is a shingle. Then $P[\pi(y) = \min(\pi(X))] = 1/|X|$, i.e., it is equally likely that any $y \in X$ is mapped to the min element. Let y satisfy $\pi(y) = \min(\pi(C1 \cup C_2))$. Then either

- $\pi(y) = \min(\pi(C_1))$ if $y \in C_1$ OR
- $\pi(y) = \min(\pi(C_2))$ if $y \in C_2$

In other words, one of the two columns has 1 at position y. So the probability that both are true is the probability that $y \in C_1 \cap C_2$. □

Now we know that $P[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$. We generalize our reasoning to multiple hash functions such that the similarity of two signatures is the fraction of the hash functions in which they agree. Because of the minhash property, the similarity of columns is the same as the *expected similarity* of their signatures. If we take 100 random permutations our signature vector is only 100 integers, that is much smaller than the original binary vectors.

13.2.2.1 A further trick

The Min-hashing algorithm computes random row permutations that require a number of calls to a random number generator. Instead of actually permuting the row one could get K hash functions k_i. The hash function induces a permutation on the items.

The one-pass implementation of such a trick is as follows. For each column C, and hash function, k_i keep a "slot" for the min-hash value. We initialize $Sig(C)[i] = \infty$, then we scan rows looking for $1s$. Suppose row j has 1 in column C, then for each k, if $k_i(j) < Sig(C)[i]$, then $Sig(C)[i] = k_i(j)$.

What is a good choice for k_i such that it simulates a permutation? We can use *universal hashing* $h_{a,b}(x) = ((ax + b) \mod p) \mod n$, where a, b are random integers and p is a prime number $p > n$.

13.2.3 Locality Sensitive Hashing (LSH)

Min-Hashing provides a faster method to compute the Jaccard similarity. Still, we need to compare $\mathcal{O}(n^2)$ signatures in order to understand find the most similar documents. LSH [IM98] tackles the problem of the comparison by hashing the signatures in an appropriate manner. The main intuition behind LSH is to use a function $f(x, y)$ to determine whether x and y is a candidate pair whose similarity must be evaluated. For Min-Hash matrices we hash the columns of the signature matrix $\mathbf{M}$ to different buckets. If two documents hashes into the same bucket, they become a candidate pair, i.e., a potential pair of documents with similarity $sim \geq threshold$.

LSH works as follows. Pick a similarity threshold $s, (0 < s < 1)$. The columns of x and y of $\mathbf{M}$ are a candidate pair if their signatures agree on at least a fraction s of their rows. $M(i, x) = M(i, y)$ for at least a fraction s of i. The idea is to hash columns of signature matrix $\mathbf{M}$ several times. Similar columns should be likely to hash to the same bucket with high probability.

LSH partitions $\mathbf{M}$'s columns into b bands each of r rows. For each band, compute a signature using a hash function with k buckets, where k is arbitrarily large. The candidate column pairs are those that hash to the same bucket for at least 1 band. Tune b and r to catch most similar pairs, but few non-similar pairs.

Example 1: Suppose $\mathbf{M}$ has 100 000 columns (documents) and 100 rows (signatures). Let $b = 20$ and $r = 5$. We want to find pairs of documents where the similarity is at least $s = 0.8$. If the similarity $sim(C_1, C_2) = 0.8$, then (C_1, C_2) should be a candidate pair. The probability that C_1, C_2 are identical in one particular band is 0.8^5, while the probability that C_1, C_2 are not similar in any of the 20 bands is $(1 - 0.8^5)^20 = 0.00035$. This means that the false negatives are very low while we find truly similar documents with nearly probability 1. Similarly, if C_1, C_2 are only 0.3 similar there is a 4.74% false positive rates.

The critical choice for LSH is on the number of bands b and the number of rows r that balances false positives / negatives. Ideally we would like to have probability 1 if $t = sim(C_1, C_2) \geq s$ and 0 otherwise. In practice, we need to strike a trade-off with the number of bands and the number of rows. As the number of bands increases, the method becomes more reliable, but the time increases as well. As we have seen in the example above, if $t = sim(C_1, C_2)$ the probability that all rows in a band are equal is t^r and the probability that some row in a band are different is $1 - t^r$. As such the probability that no band is identical is $(1 - t^r)^b$ that leads to $1 - (1 - t^r)^b$ the probability that at least 1 band is identical. If we plot such function we find a curve such as the one in Figure 13.2.

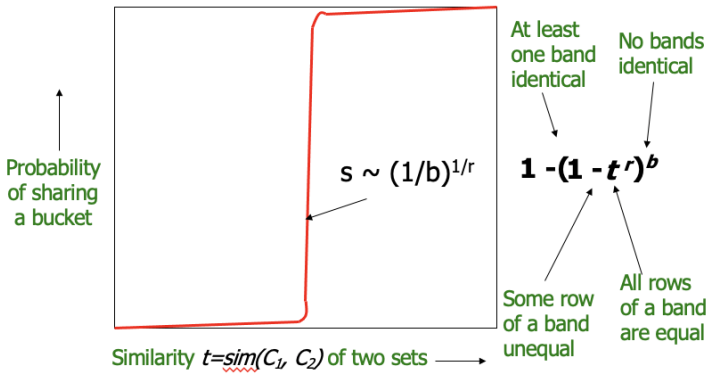


Figure 13.2: Collision probability in LSH as a function of similarity.

✔ Advantages

- Finds similar items in linear time
- Easy and fast to implement
- Generalizes to other similarity measures

✘ Disadvantages

- Performance depend on the choice of b, r
- Requires a fairly large number of buckets.

Bibliography

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

[Bel61] Richard Bellman. On the approximation of curves by line segments using dynamic programming. *Communications of the ACM*, 4(6):284, 1961.

[EASK14] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7(7):517–528, 2014.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2):1–12, 2000.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[KK04] Michihiro Kuramochi and George Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE transactions on Knowledge and Data Engineering*, 16(9):1038–1051, 2004.

[YH02] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724. IEEE, 2002.