Clustering

Sergei Vassilvitskii Suresh Venkatasubramanian

Contents

1 Introduction

Water, water, everywhere,
Nor any drop to drink

Samuel Taylor Coleridge

Clustering is perhaps the most important tool in the arsenal of the exploratory data scientist. It's certainly the most important *unsupervised learning* framework. On the research side too, clustering is invariably one of the five most popular topics for papers in data science and machine learning.

And yet, clustering is often portrayed as a collection of tricks, or algorithms. Run k-means on this data set, or a single linkage on that one. If you're feeling fancy, use spectral clustering, or even an ensemble method. What the traditional discourse on clustering lacks is a collection of deeper insights that might help guide how we think about clustering a data set.

We believe those insights exist. And this book is our attempt to make them explicit. In our view, clustering is more than just a collection of tools. It is a systematic way to think about how data is represented, and how it should be organized. Different ways to cluster data represent different intuitions on what the shape of the data is, or should be. And becoming proficient in clustering is about understanding those intuitions and how they naturally lead to different formulations of a clustering problem.

This book is about a *conceptual* understanding of clustering. Rather than being exhaustive (a wholly impossible task!), we wish to isolate the *conceptual directions* that describe the space of ideas in clustering, so that the reader is then empowered to mix and match them as she sees fit.

Befitting our backgrounds, this book also attempts to bridge the divide between the applied and theoretical perspectives on clustering. We will not seek out the algorithm that trims the last log factor in a running time, and nor will we explore every heuristic that appears to be effective. Rather, we will make explicit the connections between theory and practice, showing how theoretical intuition can guide (and is guided by) the design of practical methods.

This is a work in progress. Comment are greatly appreciated.

Part I

Basics

2 Partition Clustering

Divide et Impera

Julius Caesar

2.1 Metric Spaces

In its most general form, clustering is the following problem: given **objects** $x_1, \dots, x_n$ from some set X, place them in **groups** so that all objects in a group are in some way **related** to each other. Every particular instance of clustering we will encounter in this book merely specifies what these notions mean.

Let us start with the meaning of "related". The easiest way to say that two objects are related is to define a distance between them: objects that are close to each other are then considered related, and objects that are far away are not. Such a distance should satisfy some natural properties. First, the distance of an object to itself should be zero because every object is most closely related to itself. A slightly less natural, but equally desirable, property is that it shouldn't matter which direction the distance is measured in[1]. And finally, a critical property that a useful distance measure must satisfy is the "shortcuts are good" rule: it should always be shorter to measure distance directly from x to y rather than going through an intermediate point z. A distance function satisfying these properties is called a *distance metric*.

Definition 2.1 (Distance Metric)**.** *A function $d : X \times X \to \mathbb{R}^+$ is said to be a* distance metric *if it satisfies:*

Reflexivity $d(x, y) = 0 \Leftrightarrow x = y$

Symmetry $d(x, y) = d(y, x)$

Triangle Inequality $\forall x, y, z \in X, d(x, y) \leq d(x, z) + d(z, y)$

The set X of objects together with the distance metric d is called a *metric space*, denoted by (X, d).

[1] While a city map with one-way streets doesn't satisfy this property, it turns out that such *asymmetric* problems are significantly harder to solve. See Section **??** for more details.

Examples.

- $X = \mathbb{R}^m$, $d(x,y) = \sqrt{\sum_{i=1}^{m}(x_i - y_i)^2}$ describes Euclidean space.
- $X = \mathbb{R}^m$, $d(x,y) = \sum_{i=1}^{m} |x_i - y_i|$ describes the *Manhattan distance*.
- $X = \Sigma^*$ is the set of finite length strings over an alphabet Σ, $d(x,y)$ is the edit distance between the strings.
- $X = G(V,E)$ is a graph on n vertices. $d(x,y)$ is the length of the shortest path between x and y.

Defining a metric on a space is a choice: the same space can admit different metrics depending on the application, and the different metrics will result in different clusterings of the data.

2.2 From Distances To Clusterings

Given a distance between objects, we can define the quality of a group of objects—the degree to which they are all related. For a set of points $C = \{x_1, \ldots, x_k\}$, let

$$\Delta(C) = \max_{x,y \in C} d(x,y)$$

denote its *diameter*. Intuitively, a cluster is good if it has small diameter.

Now that we have a notion of quality for a cluster, we'd like to define a similar notion for a clustering. To start with, we should define a clustering itself. This is a well known combinatorial object called a *partition*.

Definition 2.2 (Partition)**.** *A* partition *of a set X of size k is a collection of sets* $\Pi(X) = \{C_1, C_2, \ldots, C_k\}$ *such that*

- $C_i \subset X$ *for all* i
- C_i *and* C_j *are disjoint for all* i, j
- *The* C_i *cover X:* $\cup_{i=1}^{k} C_i = X$

Now we can talk about the cost[2] of a partition. Using the above definition of the cost of a cluster in terms of its diameter, we can define the

[2] People use *cost* and *quality* interchangeably to quantify a clustering. Typically, the cost is something that you want to make smaller, and the quality is something that you want to make bigger. It usually doesn't matter which way you think about it as long as you optimize in the right direction.

cost of a partition of X as the maximum diameter of any set in the partition: $\text{Cost}(\Pi) = \max_i \Delta(C_i)$. This brings us to our first concrete clustering problem.

Problem 2.1 (k-partition)**.** *Given a metric space (X, d), find a partition $\Pi(X)$ of size k that minimizes $Cost(\Pi)$.*

Fixing k: the problem of model selection. At this point, you might wonder why we require that a partition have size k. Consider what happens if we drop this requirement. Then the trivial solution $\Pi = \{\{x_1\}, \{x_2\}, \ldots, \{x_n\}\}$ has cost, $\text{Cost}(\Pi) = 0$, and is therefore optimal. However, this solution is useless as an actual clustering of the data.

This is an example of a general problem in statistics and learning called *model selection*. Here our "model" is the number of clusters in the final partition. If we allow too many clusters, we overfit the data to the model, preventing us from generalizing to new data. If we allow too few, we get a poor quality fit of the data to the model, and our result may be quite inaccurate. We'll have much to say on the topic in Chapter 12.

Cluster Representatives. It is extremely useful to have a single point that represents all of the points in a single cluster. This point can be used *in lieu* of the data as a compression mechanism, or it can be used as an exemplar to describe the class concisely. Most (though not all) clustering algorithms will, in addition to the partition, produce a collection of cluster representatives or *centers*, one for each cluster.

Should the cluster representatives belong to the set X ? As with most things, it depends. Suppose X is a set of brain scans and the goal of clustering is to identify key patterns of variations. Then a cluster representative will necessarily be one of the input scans. On the other hand, if X is a collection of documents represented as sets of words and the goal of clustering is to determine collection of words that capture different topics, then the best representative for a topic cluster might be an entirely new "document".

In our discussion of clustering, we will distinguish two cases. The first is when the set of points X is all we know about the data. In this case, cluster centers will obviously be drawn from X. In the second case, X is drawn from a larger set U (the *universe*), and here cluster centers will always belong to U, but may or may not belong to X.

Partitions versus Centers: the Voronoi property Fix a partition Π of X and let C be a cluster in this partition. Instead of defining the cost of

C as its diameter $\Delta(C)$, we can define the cost of C as its *radius*: $r(C) = \min_{y \in C} \max_{x \in C} d(x, y)$. The center of C is then the point that realizes $r(C)$:

$$c = \arg\min_{c \in C} \max_{x \in C} d(x, c).$$

We can now define our clustering problem as a search for good cluster centers.

Problem 2.2 (k-center)**.** *Given a metric space* (X, d)*, find centers* $c_1, \ldots, c_k \in X$ *such that* $\max_i \min_{x \in X} d(x, c_i)$ *is minimized.*

Effectively, we are searching for a partition in which the maximum radius of a cluster is minimized. Recovering the partition Π associated with this clustering is also easy: assign a point to cluster C_i if the closest center to it is c_i. This follows by observing that assigning the point to any other cluster can never decrease the overall cost of the partition.

This rule is sufficiently general that it even has its own name: the *Voronoi property*. A clustering problem admits the Voronoi property if each point is always assigned to its *closest* cluster center. This property makes it very easy to assign a new point to a cluster: merely find its nearest neighbor among the centers. It also induces a partition of the underlying space: the "region of influence" of c_i consists of all points x such that $d(x, c_i) < d(x, c_j), j \neq i$ [3].

2.3 Finding a Good Partition

Our new formulation of the clustering problem has one major advantage. Instead of searching over the space of partitions $\Pi(X)$, we can search over subsets of X of size k. So how do we find these centers ?

To gain some intuition, consider the case when $k = 2$ and we simply want to divide the points into two groups. One thing that is clear is that the two points that are furthest away from each other should be placed in different groups. Otherwise the radius of the partition containing both of these points will be at least $\Delta(X)/2$, which is as good as not dividing the points into two groups at all!

We can use the same logic when reasoning about the more general case. Suppose we have already selected some number of cluster centers, which point should we select as a center next? Define the cost of the partial solution $C = \{c_1, c_2, \ldots, c_j\}$ as the maximum distance between any point and its nearest center:

[3] This is merely the Voronoi diagram of the cluster centers.

$$\text{Cost}(C) = \max_{x \in X} \min_{1 \le i \le j} d(x, c_i).$$

Given a precise definition of Cost, we can now compare the quality of different clusterings, and reason about the best one. The optimum solution to the k-center problem is the one that minimizes the cost above. This cost is always determined by a pair of points, a cluster center $\hat{c} \in C$ and a data point $\hat{x} \in X$, so that $\text{Cost}(C) = d(\hat{x}, \hat{c})$. The only way to reduce the cost of the overall clustering is to select a point closer than $\hat{c}$ to $\hat{x}$ as a cluster center as selecting any other point will not reduce the cost of the overall clustering. A sure way to do this is to add $\hat{x}$ itself as a center!

Thus we have our first algorithm for the k-center problem:

Algorithm 2.1 FurthestPoint

```
Select $c_1$ arbitrarily from $X$.
$C \leftarrow \{c_1\}$.
**for** $i = 2, \ldots, k$ **do**
  $c_i \leftarrow \arg\max_x \min_{c \in C} d(x, c)$.
  $C \leftarrow C \cup \{c_i\}$
**return** $C$
```

Evaluating the algorithm. How do we reason about the quality of this algorithm ? One measure of quality is the running time: the lower the better. FurthestPoint can be implemented by doing $O(n)$ distance computations every iteration, leading to an $O(nk)$ overall running time.

But of course it is easy to find very fast clustering algorithms: just pick k random points! We need to instead measure how the solution produced by the algorithm stacks up against the optimal solution.

Consider the cost of the solution as the algorithm proceeds. When we pick the first point arbitrarily, the cost $\text{Cost}(\{c_1\})$, is no more than the diameter $\Delta(X)$. As we pick more and more points this cost never increases. Let Cost_i denote the cost after selecting i centers. Then $\Delta(X) \ge \text{Cost}_1 \ge \text{Cost}_2 \ldots \ge \text{Cost}_k$. The key to the analysis lies in showing that the points we select are far away from each other. In particular, when we select the i^{th} center c_i it must be at least Cost_{i-1} away from any previously selected points.

Let us suppose that we run the algorithm for $k + 1$ steps. By the above argument the $k + 1$ centers selected are at least Cost_k apart form each other. By the pigeonhole principle any solution that uses k clusters would have to put two of these points in the same group. The radius of that group must be at least $\text{Cost}_k/2$. Therefore we have an algorithm that gives a solution with cost Cost_k and we have shown that *any* solution has cost at least $\text{Cost}_k/2$. Therefore, the solution we produce is always at most twice the optimal cost.

It is worth taking a step back and remarking on this fact. Although finding the optimum solution is computationally hard (the problem is NP-complete), the simple algorithm above *always* produces a solution within a factor of two of the optimum. We call such an algorithm a *2-approximation* algorithm.

We will see the notion of approximation algorithms throughout the book. More generally,

Definition 2.3 (Approximation Algorithms)**.** *An algorithm is a c-approximation algorithm if the solution it produces is always within a factor of c of the optimum.*

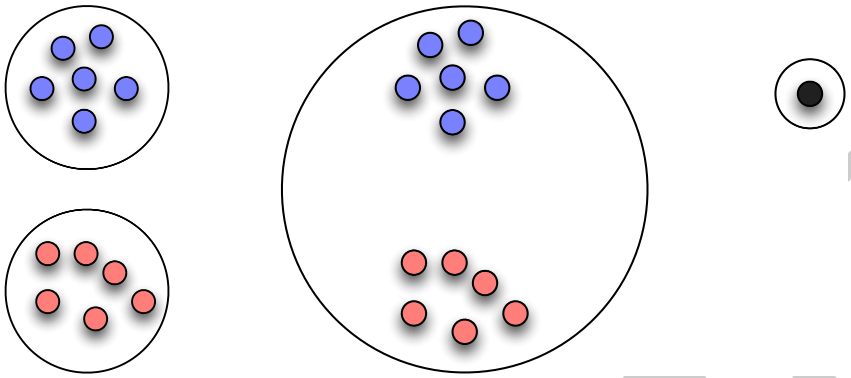
It seems both miraculous and a little shocking that such a simple algorithm can be proven to have reasonable quality. Unfortunately it turns out that we can't do any better: improving the approximation ratio beyond a factor of 2 is NP-hard, and is therefore as hard as solving the problem exactly.

2.4 Finding Robust Partitions

Our first foray into clustering algorithms has been quite satisfying. We defined an intuitive way to measure the quality of a clustering, and presented an algorithm that computes a high-quality solution efficiently. But a high-quality solution is not the same thing as a meaningful solution, and the k-center-based clustering formulation has some important weaknesses.

Consider the two 2-clusterings depicted in Figure 2.1. It should be clear that in the first clustering, one center is forced to move far away from its natural location in order to reduce the cost created by a single point. This is a problem because now the cluster has become much larger than it really needs to be.

The problem is that the k-center objective is very sensitive and changing the location of a single point can dramatically change both the cost and the structure of the optimal solution. While this sensitivity makes for simple and nearly optimal algorithms, it means that a few stray points can artificially distort the location of centers and reduce the qualitative utility of the solution. Such points are called *outliers* and represent a difficult problem for any clustering algorithm.



(a) No outliers

(b) With an outlier

Figure 2.1: The single black outlier forces the clustering to allocate a single cluster for it in order to minimize the maximum radius

There are two ways to deal with this problem. One approach is to explicitly acknowledge that outliers exist and design algorithms that are allowed to ignore a few points if they seem to affect the clustering disproportionately. We will see more of this approach in Chapter 18. A second approach is to design a cost function that is less sensitive to outliers, and that is the approach we take next.

Robust estimators and the median

Statisticians have a name for estimators that can resist corruption by outliers: they are called *robust estimators*. For our purposes, it suffices to consider an intuitive definition of robustness: in order for the cluster center to change drastically, many points must be corrupted.

There is a very well known entity that satisfies this property: the median of a set of numbers. Given a set of numbers $x_1 \leq x_2 \leq \ldots \leq x_n$, the median is the middle point (in case n is even it lies between the two middle points). While a few outliers will change the median, to increase it to beyond x_n would require an addition of at least n numbers above x_n.

While the median is less sensitive to outliers on the line, the concept of a median doesn't generalize to even two dimensions, let alone an arbitrary metric space. There is no general way to "sort" points and take the middle element. However, there is an alternate way of defining the median that does generalize to a metric space, it is the point that minimizes the *average* distance between it and all of the points. Note this is in contrast to the center, which minimizes the *maximum* distance between it and all of the points.

Theorem 2.1. *Let $X = \{x_1, \ldots, x_n\}$ be a set of points on the line. Then the median m of X satisfies*

$$m = \arg\min_{x \in \mathbb{R}} \sum_{i=1}^{n} d(x_i, x)$$

Proof Sketch: Consider any point $x \in \mathbb{R}$ and assume it has k points of X to its left and $n - k$ points of X to its right. Imagine moving x a tiny amount δ to the right. Its distance to all points on the right will decrease by δ, and its distance to all points on its left will *increase* by δ. The distance between any two numbers on the line is the absolute difference between the two, therefore the net decrease in distance will be $\delta(n - k - k) = \delta(n - 2k)$. Therefore, as long as $k < n/2$, moving to the right will decrease the overall sum of distances. A similar argument holds for moving to the left. Thus, the place where the sum of distances is minimized is when x has the same number of points on both sides, which means that it's the median. □

Thus we can generalize the concept of a median to general metric spaces:

Definition 2.4 (Medoid)**.** *Given a set of points $x_1, \ldots, x_n$ in a metric space (X, d), the medoid m is defined as*

$$m = \arg\min_{x \in X} \sum_{i=1}^{n} d(x_i, x)$$

With this definition of a cluster representative in hand, we can now define the k-median objective.

Definition 2.5 (k-median)**.** *Given a metric space (X, d), find a set of medoids $m_1, \ldots, m_k$ such that*

$$\sum_{x \in X} \min_{i=1}^{k} d(x, m_i)$$

is minimized.

Algorithms for k-median

It may be tempting to simply re-use the FurthestPoint algorithm for the k-median problem. However, it is easy to construct examples where this approach will be far from optimal. This is exactly what we expect to see. In a sense, k-median solutions are required to trade-off the radius of a cluster with the robustness of the solution, and it would be surprising if an algorithm ignoring robustness would perform well on this objective.

We thus turn to a different class of algorithms, those that slowly change the optimal solution while decreasing the cost and searching for a (nearly) optimal set of medians.

Local Search A simple approach for solving the k-median problem is to keep trying new points as medoids and keeping the best solution so far. A brute force search would be too costly—there are $O(n^k)$ possible candidates for the set of medoids and trying them all is computationally infeasible. Instead we proceed by only changing one medoid at a time. More precisely, we fix $k-1$ of the medoids, and try swapping the k^{th} medoid with one of the unassigned points. If the cost of the solution decreases significantly, we keep it, otherwise, we try a different pair to swap.

Algorithm 2.2 Local Search

```
Start with medoids $M = \{m_1, m_2, \ldots, m_k\}$ chosen arbitrarily from $X$.
repeat
  change ← false
  for $x \in X, m \in M$ do
    $M' = M \setminus \{m\} \cup \{x\}$
    if $\mathrm{Cost}(M') < (1-\epsilon)\mathrm{Cost}(M)$ then
      $M \leftarrow M'$, change ← true
until !change
return $M$
```

Notice that in the algorithm, we don't make a change if the new medoid reduces the cost ($\mathrm{Cost}(M') < \mathrm{Cost}(M)$) by an infinitesimal amount. Rather, (setting the parameter $\epsilon = 0.01$ for example), we only switch if the new medoid reduces the cost to at most 99% of the previous cost. This minor modification ensures that the algorithm terminates after polynomially many iterations, and moreover that the final solution will be within a factor of five of optimal.

The proof is non-trivial and is too difficult to reproduce here. It also shows that one can further improve the algorithm by swapping multiple medoids at a time. For example, if we swap pairs of medoids, the algorithm converges to a 4-approximation, and more generally, swapping p medoids leads to a $3 + 2/p$-approximate solution. However, doing so increases the running time per iteration from $O(k^2n^2)$ to $O(k^2n^{p+1})$.

3 Clustering in vector spaces and k-means

It is no mean pleasure to be seated in the mean

Antonio, The Merchant of Venice

General metrics like the ones we saw in Chapter 2 typically come from a similarity function between pairs of elements. But what if you have more information about the points, such as numerical features ? There is a general principle of clustering: more structure on the data can lead you to more targeted algorithms that will probably work better. Resist the urge to be too general !

In many clustering problems we encounter, the objects we are clustering have numerical features. For example, when clustering a group of people, we may consider their height, their weight, and so on. In this case, it makes sense to talk about an *average* height or weight. Formally, we say that these items lie in a *vector space*.

Definition 3.1 (Vector Space). *A vector space consists of points that we are allowed to add and subtract, as well as multiply by a number. For example, if we consider the space of pairs of numbers* (x, y), *we can*

add them: $(x, y) + (x', y') = (x + y, x' + y')$

scale them: $c(x, y) = (cx, cy)$

Vector spaces provide a useful representation for numerical features because it is possible to combine multiple features them while retaining the vector space properties, simply by concatenating the features together. For example, we can augment the dataset with a person's age, also represented as a numerical feature, simply by adding it as an additional dimension.

In contrast to numerical features, when working with categorical features, such as gender, it does not make sense to talk about averages. Rather we must choose one from a finite list of possibilities. Such a dataset is not typically represented as a vector space.

When working with vector spaces, we have a a natural way of representing the center of a cluster, namely the *average* point, $\boldsymbol{\mu}$:

$$\boldsymbol{\mu} = \frac{1}{|C|} \sum_{x_i \in C} x_i.$$

As we will see this natural representation gives rise to simple and efficient clustering algorithms.

3.1 k-Means Algorithm

The ability to easily define centers leads to a very simple algorithm for clustering points into k groups. As in Chapter 2, let $X = \{x_1, x_2, \ldots, x_n\}$ denote the set of points. Our goal is to partition these points into k clusters, $C_1, C_2, \ldots, C_k$.

Given a set of initial cluster centers (See Section 3.3), the algorithm repeats the following two steps:

i) Assign each point to its nearest cluster center.

ii) Recompute each cluster center as the *mean* of all of the points assigned to it.

This approach is known as Lloyd's method or, more simply, k-means. We give the formal pseudocode description below and show its execution on a simple example in Figure 3.1.

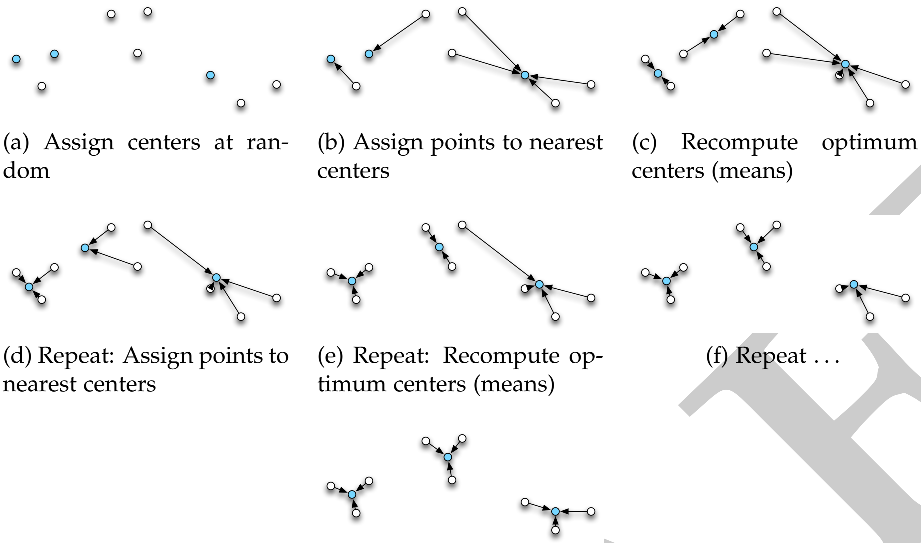
Algorithm 3.1 k-Means

```
Select $c_1, c_2, \ldots, c_k$ distinct points from $X$.
repeat
  for each point $x_i \in X$ do
    Assign $x_i$ to cluster $C_j$ that minimizes $d(x_i, c_j)$
  for Each cluster $C_i$ do
    set $c_i = \frac{1}{|C_i|} \sum_{x \in C} x$
until cluster centers don't change
return $C_1, C_2, \ldots, C_k$.
```

As with any algorithm we must ask two questions about Lloyd's method—how fast is it, and how good is the answer?

3.2 Running Time Analysis

The algorithm above is simple and natural. It repeatedly assigns each point to its nearest cluster center, and then recomputes the center as the mean of the points assigned to it. The second step changes the location of the centers, and may lead to some points no longer being assigned to their closest center.



(a) Assign centers at random

(b) Assign points to nearest centers

(c) Recompute optimum centers (means)

(d) Repeat: Assign points to nearest centers

(e) Repeat: Recompute optimum centers (means)

(f) Repeat . . .

(g) . . . until clustering does not change

Figure 3.1: The execution of *k*-Means on an example with 9 points and 3 clusters.

While the algorithm is simple, it is not clear whether it always finishes or if it might get stuck in an infinite loop.

One way to prove that the algorithm terminates is to find an objective function that improves with every step. What could be a viable candidate? We saw in the last chapter that for a set of points on a line, it is the *median* that minimizes the sum of distances from the point to the nearest center. Since this method computes the *mean* rather than the median, and the two may be quite different, a different objective function must be in play.

The question we are asking then, is for what objective functions is selecting the mean the right thing to do? Put another way, what are the functions $g(\cdot,\cdot)$ such that

$$\frac{\sum_i x_i}{n} = \arg\min_c \sum_i g(x_i, c)?$$

It turns out that there is a very large class of functions that satisfy this equation. They are called *Bregman divergences* and encompass many familiar metrics, including the KL-divergence and the Itakura-Saito distance. In addition, this class contains a special distance function that we're very familiar with: the squared Euclidean distance.

Therefore:

$$\frac{\sum_i x_i}{n} = \arg\min_c \sum_i \|x_i - c\|_2^2$$

In other words, the mean of a set of points is the point that minimizes the sum of squared distances between it and all of the points in the set. This leads us to a natural candidate for an objective function, ϕ, for the algorithm.

Definition 3.2 (k-means objective)**.** *Given a set of points X in Euclidean space find a set of centers $C = \{c_1, \ldots, c_k\}$ such that:*

$$\phi(X, C) = \sum_{x \in X} \min_{c_i \in C} \|x - c_i\|^2$$

is minimized.

Armed with this objective function, also called a *cost* function, we can argue that the k-means algorithm always terminates. Consider what happens to the objective in every step. When we recompute the centers given an assignment of points to a cluster, we decrease the objective, since the mean always minimizes the sum of squared distances to the center. Furthermore, whenever we reassign a point to a closer center, we decrease the objective as well. Therefore, while the algorithm is running, the total objective value keeps on decreasing. This implies that the algorithm never cycles: we can never see the same set of cluster centers twice, since that would imply that the objective increased at some point. As there are a finite number of possible assignments of points to centers, the algorithm is bound to eventually terminate.

"Eventually" sounds like a very weak guarantee, and indeed, there are specific configurations on which k-means may take exponential time to converge, even when all of the points lie in two dimensions. However, these lower bound constructions are extremely brittle, and in practice the algorithm almost always converges after a few dozen iterations. This fact can be proved rigorously, by showing that the running time is polynomial if the initial points come from noisy observations, but this so-called *smoothed analysis* is beyond the focus of this book.

The k-means algorithm is a local search method, like others we saw in Section 2.4 for the k-median problem. Since it monotonically decreases the cost function, ϕ, we can always stop the algorithm early without running it to completion if time is of the essence.

3.3 Initialization and Approximation Guarantees

The k-MEANS algorithm always converges to a solution. How good is this solution? We saw other local search methods converge to approximately optimal clusterings. But this is an exception. Most local search algorithms do *not* converge to approximately optimal solutions. The k-MEANS algorithm can result in highly suboptimal clusterings.

Before we talk about the quality of approximation, we must talk about how we initialize the algorithm. A common, but naive, approach is to initialize randomly, selecting k random points from the dataset and using them as the initial set of k centers. After all, for the k-median algorithm, the initialization did not matter—the local search always converged to an approximately optimal solution.

However with k-MEANS this turns out to be a very poor thing to do. Consider a point set in one dimension, consisting of k groups of points located far apart from each other, shown in Figure 3.2. A randomized initialization is very likely to miss some of these groups, and place multiple points in others. Because of the nature of the k-MEANS algorithm, it cannot "unbreak" these bad decisions, and will terminate with some of the groups of points split into multiple clusters, and other groups merged together. Locally such an assignment is optimal: each point is assigned to its closest center, and each center is the mean of all of the points assigned to it; however it is easy to see that this clustering is far from optimal.

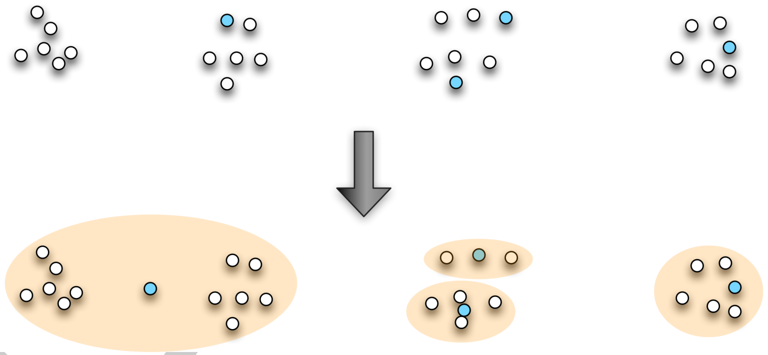


Figure 3.2: An example dataset where random initialization works poorly for k-MEANS. Even though there are four well separated clusters, the locally optimum solution merges two of these together.

The question is whether there is a simple initialization that does better.

Intuitively we want to initialize the algorithm with well-separated centers—the problem shown in Figure 3.2 stems from the fact that some of the initial centers are located very close together. One approach is to first find a nearly optimal k-center solution using the FurthestPoint algorithm and use that to initialize or *seed* the k-means algorithm.

However, this approach is overly sensitive to outliers and may result in initial seed clusterings that converge to bad solutions .

What we are looking for is for a method that lies between these two extremes. On the one hand, selecting points randomly is problematic, as we may choose points too close together. On the other, we should only consider far away points as candidate centers if there are many points in the vicinity, rather than giving weight to a single outlier.

There are many ways to convert this intuition into an algorithm. There is a method that balances these two objectives and still employs random sampling, albeit from a distribution biased towards far away points.

The method, known as k-means++ selects one center at a time. It picks the first center uniformly at random from all of the data points. Then it computes a biased distribution from which to sample the next center. Recall the value of the objective function in Definition 3.2:

$$\phi(C, X) = \sum_{x \in X} \min_{cinC} \|x - c\|_2^2.$$

A single point $x \in X$ contributes

$$\text{Cost}(x, C) = \min_{c \in C} \|x - c\|_2^2$$

to the objective. For each point x_i, let $p_i = \text{Cost}(x_i, C)/\phi(X, C)$ represent the contribution of x_i to the overall cost. The values $(p_1, p_2, \ldots, p_n)$ form a probability distribution: each p_i is non-negative, and $\sum_i p_i = 1$. Now to select a center, the method samples from this probability distribution. That is, it picks x_i as the next cluster center with probability p_i. Once the next center is selected, the value of the objective function changes, as does the distribution p (because C has changed). The algorithm computes this new distribution p and once again samples from it. This process repeats till the algorithm has k centers to initialize Lloyd's algorithm.

The intuition behind this approach is that it seeks out far-away points as initial centers, but in a more relaxed way (because of the randomness) than the FurthestPoint algorithm might work. The proof showing that this tradeoff is correct is quite involved, but we give the highlight of the analysis here.

Algorithm 3.2 k-Means++

```
Select $c_1$ randomly from $X$ and let $C = \{c_1\}$.
for $j = 2$ to $k$ do
  for Each point $x_i \in X$ do
    Let $\mathrm{Cost}(x_i) = \min_{\ell=1}^{j-1} \|x_i - c_j\|^2$.
  Let $\phi = \sum_{x_i \in X} \mathrm{Cost}(x_i)$.
  Sample a random point $y \in X$, selecting each $x_i$ with probability proportional to $\mathrm{Cost}(x_i)/\phi$.
  Let $c_j = y$, and $C = C \cup \{c_j\}$.
Invoke Algorithm 3.1 on C.
```

Analysis of k-means++

Consider the optimal k-centers, $C^* = \{c_1^*, \ldots, c_k^*\}$, and let X_j^* be the points assigned to center j in the optimum solution, that is

$$X_j^* = \{x \in X : \|x - c_j^*\| \leq \min_{\ell=1}^{k} \|x - c_\ell^*\|\}.$$

The analysis proceeds in two steps. First we show that whenever we select a center from some X_j^* it is a good approximation for all points in X_j^*. Second, we argue that during the course of the algorithm we select points from almost all optimum clusters.

First suppose that we are trying to find the optimum center for a single cluster. We know that selecting the mean is the best thing to do, but what happens when we select one of the input points at random. How bad is this algorithm?

Let X be the set of points. The cost of using a specific center c is:

$$\mathrm{Cost}(X, \{c\}) = \sum_{x \in X} \|x - c\|_2^2.$$

If we select the center at random, then each $x \in X$ has an equal probability of being selected, namely $1/|X|$. The expected cost is then:

$$\sum_{c \in X} \frac{1}{|X|} \sum_{x \in X} \|x - c\|_2^2$$

In exercises , we show that:

$$\sum_{c \in X} \frac{1}{|X|} \sum_{x \in X} \|x - c\|_2^2 = 2 \sum_{x \in X} \|x - c^*\|,$$

where c^* is the optimum center (the mean). Therefore, in expectation, picking a random center serves as a 2-approximation to all of the points in the cluster.

It seems then, that randomly selecting centers is a good thing to do. Where does this logic break down? Consider what happens when the optimum solution one cluster has many more points than in the others. In this case, if we sample two points at random, it is likely that both of them will come from the same optimum cluster, leaving one cluster untouched. This is precisely the situation that we saw in Figure 3.2.

We can fix this issue by biasing our selection to points that are far away from the currently selected cluster, just like *k*-means++, however then the centers are not selected uniformly at random from a cluster, and therefore may no longer lead to a 2-approximation.

To analyze what happens during the execution of *k*-means++, suppose we have already selected centers $c_1, c_2, \dots, c_j$, and are now selecting center c_{j+1}. Let X be the optimum cluster that c_{j+1} belongs to.

The cost of using $\{c_1, c_2, \dots, c_{j+1}\}$ to cover all of the points in X is then:

$$\mathrm{Cost}(X, \{c_1, c_2, \dots, c_{j+1}\}) = \sum_{x \in X} \min_{\ell=1}^{j+1} \|x - c_\ell\|_2^2.$$

We break the minimum term into two, to isolate exactly the improvement from adding c_{j+1}.

$$\begin{aligned}\mathrm{Cost}(X, \{c_1, c_2, \dots, c_{j+1}\}) &= \sum_{x \in X} \min \left[\min_{\ell=1}^{j} \|x - c_\ell\|_2^2, \|x - c_{j+1}\|_2^2 \right] \\ &= \sum_{x \in X} \min \left[D^2(x), \|x - c_{j+1}\|_2^2 \right],\end{aligned}$$

where

$$D^2(x) = \mathrm{Cost}(\{x\}, \{c_1, c_2, \dots, c_j) = \min_{\ell=1}^{j} \|x - c_\ell\|_2^2.$$

The probability of selecting the specific c_{j+1} is not the same for all points in X. Rather, it is:

$$\frac{D^2(c_{j+1})}{\sum_{x \in X} D^2(x)}.$$

Therefore, the expected cost is:

$$\sum_{c_{j+1} \in x} \frac{D^2(c_{j+1})}{\sum_{x \in X} D^2(x)} \cdot \sum_{x \in X} \min \left[D^2(x), \|x - c_{j+1}\|_2^2 \right] \tag{3.1}$$

To make sense of this expected cost, we will expand the $D^2(c_{j+1})$ term. By the triangle inequality for any point x,

$$\min_{\ell=1}^{j} \|c_{j+1} - c_\ell\|_2 \leq \min_{\ell=1}^{j} \|x - c_\ell\|_2 + \|c_{j+1} - x\|_2,$$

Since we are considering squared distances, we must use the relaxed triangle inequality:

$$\begin{aligned} \min_{\ell=1}^{j} \|c_{j+1} - c_\ell\|_2^2 &\leq 2 \min_{\ell=1}^{j} \|x - c_\ell\|_2^2 + 2\|c_{j+1} - x\|_2^2 \\ D^2(c_{j+1}) &\leq 2D^2(x) + 2\|c_{j+1} - x\|_2^2 \end{aligned}$$

Since this inequality holds for all $x \in X$, we can sum it across all of the points, and then divide by $|X|$:

$$D^2(c_{j+1}) \leq \frac{2}{|X|} \sum_{x \in X} D^2(x) + \frac{2}{|X|} \sum_{x \in X} \|c_{j+1} - x\|_2^2.$$

In the last equation the first term is independent of c_{j+1}, rather it gives more weight to points in clusters that are poorly covered by the current solution, formalizing the intuition behind the method.

Now let's plug this bound into the expect cost we computed in Equation 3.1. The total expected cost is :

$$\begin{aligned} &\sum_{c_{j+1} \in x} \frac{D^2(c_{j+1})}{\sum_{x \in X} D^2(x)} \cdot \sum_{x \in X} \min\left[D^2(x), \|x - c_{j+1}\|_2^2\right] \\ &\leq \sum_{c_{j+1} \in x} \left(\frac{2}{|X|} \sum_{x \in X} D^2(x) + \frac{2}{|X|} \sum_{x \in X} \|c_{j+1} - x\|_2^2. \right) \cdot \frac{1}{\sum_{x \in X} D^2(x)} \cdot \sum_{x \in X} \min\left[D^2(x), \|x - c_{j+1}\|_2^2\right]. \end{aligned}$$

If we distribute the sum, the first term is:

$$\sum_{c_{j+1} \in X} \frac{2}{|X|} \frac{\sum_{x \in X} D^2(x)}{\sum_{x \in X} D^2(x)} \cdot \sum_{x \in X} \min\left[D^2(x), \|x - c_{j+1}\|^2\right] \leq \sum_{c_{j+1} \in X} \sum_{x \in X} \frac{2}{|X|} \|x - c_{j+1}\|^2$$

Similarly, the second term is:

$$\sum_{c_{j+1} \in X} \frac{2}{|X|} \frac{\sum_{x \in X} \|c_{j+1} - x\|_2^2}{\sum_{x \in X} D^2(x)} \cdot \sum_{x \in X} \min\left[D^2(x), \|x - c_{j+1}\|^2\right] \quad \leq \sum_{c_{j+1} \in X} \sum_{x \in X} \frac{2}{|X|} \|c_{j+1} - x\|^2.$$

Summing these two bounds together we get:

$$4 \cdot \frac{1}{|X|} \sum_{c_{j+1} \in X} \sum_{x \in X} \|x - c_{j+1}\|^2,$$

which is exactly four times the cost of sampling uniformly at random from the cluster, which we know to be 2-approximate! Therefore even with this biased sampling in expectation, the selected center serves as a 8-approximation to all of the points in the optimum cluster.

To complete the proof we must argue that we select centers from all optimum clusters. The formal proof is by double induction, and is rather tedious. In the end, we can show that in expectation the set of clusters produces by k-MEANS++ forms an $O(\log k)$ approximation to the optimum solution (See end of chapter notes). This is far better than a simple randomized assignment, which offered no performance guarantees, and the difference is often borne out in practice with k-MEANS++ leading to clusterings with significantly lower cost.

3.4 *Beyond Squared Distances

The objective of minimizing the sum of squared distances is a very specific one, but we can extend the analysis of the k-MEANS and k-MEANS++ algorithms to other metrics, such as the k-MEDIAN and KL-DIVERGENCE.

First, suppose our objective function used ℓ-th powers, instead of squares,

$$\phi_\ell(C) = \sum_{x \in X} \min_{c \in C} \|x - c\|_2^\ell.$$

If we set $\ell = 1$ we recover the *k-median* objective. Is there a single algorithm that performs well for all ℓ? We can generalize the k=MEANS++ seeding method to this setting, so that after selecting the first center uniformly at random, it selects centers 2 through k with probability proportional to D^ℓ.

Theorem 3.1. *The generalized k-MEANS++ algorithm gives an $O(2^\ell \log k)$ approximation to ϕ_ℓ.*

Although we get a good approximation after the seeding step, we can no longer run the k-Means local search method to improve the solution, since the optimum center for a cluster no longer lies at the mean of the points.

Bregman Divergences

Instead of looking for other local search algorithm, we focus on functions where the mean is the optimal center for a single cluster. While we may not have a good initialization procedure for such functions, we know that the k-Means algorithm will at least converge to some local optimum.

The sum of squared distances is not the only function for which this property holds. The same is true for a large family of functions known as *Bregman Divergences*. Formally, let F be a convex and differentiable function. For two points u and v we can define the Bregman divergence, D_F as

$$D_F(u, v) = F(u) - F(v) - \langle \nabla F(v), u - v \rangle,$$

where $\langle s, t \rangle$ represents the dot product between vectors s and t. For example, when F is the squared distance function, $F(x) = \|x\|_2^2$, then

$$D_F(u, v) = \|u\|_2^2 - \|v\|_2^2 - \langle 2v, u - v \rangle = \|u\|_2^2 - 2\langle u, v \rangle + \|v\|_2^2 = \|u - v\|_2^2$$

i.e. the squared Euclidean distance.

Except for the squared Euclidean distance, Bregman divergences are not symmetric That is, the distance from u to v may be very different than that from v to u. A classic example is the Kullback-Leibler divergence, which measures the relative entropy between probability distributions. Suppose that P and Q are discrete distributions over k items, then

$$D_{KL}(P, Q) = \sum_{i=1}^{k} p_i \log \frac{p_i}{q_i}.$$

It is easy to see that this distance is not symmetric. We can however verify that it is a Bregman divergence, with $F(x) = \sum_i x(i) \left(\log x(i) - 1\right)$.

The k-means algorithm for Euclidean metrics is really just a special case of clustering under Bregman divergences. Specifically, let F be a convex function, and D_F the induced Bregman divergence. For a point set X, and a set of clusters C, consider the objective function,

$$\phi_F(X, C) = \sum_{x \in X} \min_{c \in C} D_F(x, c)$$

Then the k-means algorithm under D_F is defined as given in Algorithm 3.3.

We can use the same logic as before to argue that Bregman k-Means will reduce the potential ϕ_F at every step, and thus is guaranteed to converge to a local minimum. Unfortunately, unlike with the k-Means algorithm, there are no known initialization methods that will guarantee an approximately optimal solution.

Algorithm 3.3 Bregman k-Means

```
Select $c_1, c_2, \ldots, c_k$ as distinct points from $X$.
repeat
    for each point $x_i \in X$ do
        Assign $x_i$ to cluster $C_j$ that minimizes $D_F(x_i, c_j)$
    for each cluster $C_i$ do
        set $c_i = \frac{1}{|C_i|} \sum_{x \in C} x$
until cluster centers don't change
return $C_1, C_2, \ldots, C_k$.
```

Hierarchical Clustering 4

Mind the gap

London Underground

All of the clustering algorithms we have seen so far have a parameter k denoting the desired number of clusters. Sometimes we know that we want to partition the data into a specific number of groups (to load onto machines, or because we know something about the data). But more often than not, since clustering is an *exploratory mechanism*, we don't really know what the right value of k is.

There are two ways to address this problem:

(a) Figure out the "right" k for a problem. This is a complicated matter, and will be the topic of Chapter 12.

(b) Not choose: give a compact universal representation which encodes the clustering for every value of k.

This latter formulation takes us into the world of *hierarchical clustering*, which we discuss in this Chapter.

4.1 The message is in the gaps

To formulate a clustering problem we have used a simple recipe. First, we defined some measure of distance between points. Second, we constructed a notion of the quality of a cluster using the defined distance. Finally, we used this notion of quality to define the quality of a partition into clusters, and search for a single partition of high quality. The particular way in which we define these notions gives us different kinds of clustering problems with different assumptions on what constitutes a clustering.

Hierarchical clustering represents a fundamentally different perspective on what a clustering is. Rather than search for a *single* clustering that captures the structure we are looking for, we instead search for a collection of clusterings that captures successively finer partitions of the data. While each clustering is represented as a single point in this collection, it is the entire ensemble of clusterings together that constitutes the answer. It is the sequence of merges of points into bigger and bigger clusters that tells us where structure lies in the data, rather than any individual snapshot.

Consider the example data set shown in Figure 4.1. On the left, we see what the data looks like at one level of resolution. On the right, we see what happens when we zoom into one of the clusters. In the perspective from afar, the data appears to admit three clusters; as we move closer, one cluster resolves itself into three more clusters.

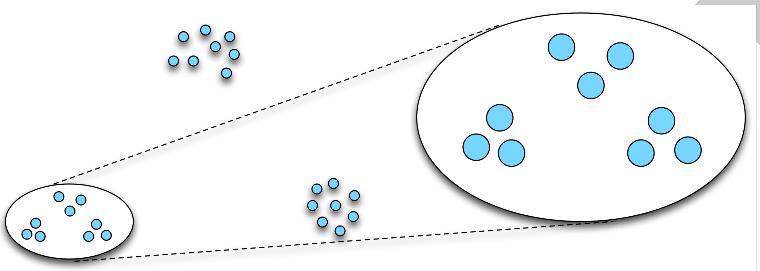


Figure 4.1: Multiple views of a single data set

Which of the two perspectives is the "right" one ? A distance-based clustering problem of the kind we saw in the last chapter will carefully evaluate the cost of each of these partitions, and decide on one of them as the "better" answer. A hierarchical clustering algorithm will (or should) declare both to be valid answers, but at different scales.

It is this view from multiple scales that often gives hierarchical clustering the sobriquet *multi-scale clustering*. Multi-scale analysis in general is based on the idea that data admits different structure at different scales, and a true representation of the data must capture all of these scales, rather than one of them. Multi-scale analysis is particularly effective in exploratory settings where we might not know the right perspective on the data; it is therefore not surprising that hierarchical clustering is one of the most popular methods for clustering.

4.2 Representation

A clustering is a partition of the input X into clusters $C_1, \ldots, C_k$. A hierarchical clustering is a rooted tree that represents a collection of partitions of X. Each node v of the tree is associated with a subset $S(v) \subseteq X$, with the following properties:

- The root r is associated with X: $S(r) = X$
- If v is a child of u, then $S(v) \subset S(u)$

- If a node u has children $v_1, v_2, \ldots v_m$, then $\{S(v_1), S(v_2), \ldots, S(v_m)\}$ forms a partition of $S(u)$.

Figure 4.2 illustrates what the data from Figure 4.1 might look like as a hierarchical clustering.

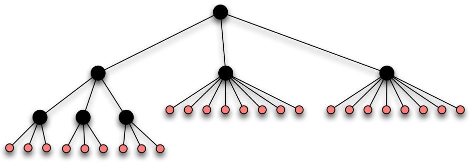


Figure 4.2: Representation of a hierarchical clustering

Each level of the tree represents a single clustering of X, and deeper levels represent a finer partition of the data, equivalent to using a larger value of k. What is also interesting about this representation is that a valid clustering need not merely be a single level. In fact, any collection of nodes $v_1, v_2, \ldots, v_k$ of the tree such that

- no two v_i have an ancestor-descendant relationship
- any path from the root r to a leaf intersects exactly one of the v_i

will yield a partition $S(v_1), S(v_2), \ldots S(v_k)$ of X. We think of such a set as a *cut* of the tree, in that it "cuts" the root off from the leaves. For example, Figure 4.3 illustrates how the 5-clustering described earlier can be represented in terms of such a cut.

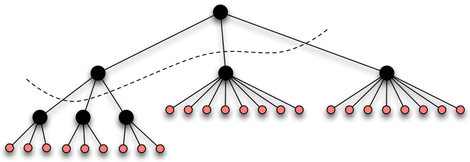


Figure 4.3: An illustration of a cut in a hierarchical clustering

4.3 Algorithms

There are two primary strategies for constructing a hierarchical clustering: a top-down approach and a bottom-up approach.

Divisive Hierarchical Clustering

The top-down approach is often called *divisive hierarchical clustering (DHC)*. We split our set of points into a few parts, and recursively find a clustering of each part. The union of the resulting clusterings yields the desired answer. While divisive clustering lends itself to a simple recursive formulation, it is not obvious how one might split the point set without knowing more about the structure of the input. Of course, we already have strategies to split our set of points into a few parts: it is just a regular clustering algorithm of the kind we saw in the previous chapters. So given any standard k-clustering algorithm, we have the following algorithm for DHC.

Algorithm 4.1 Divisive hierarchical clustering (DHC)

```
Input: $X$, root node $r$.
  Set $S(r) = X$
  if $|X| \leq k$ then
    return $r$.
  Partition $X$ into $k$ pieces $C_1, \ldots, C_k$ using any $k$-clustering algorithm
  Fix nodes $v_1, v_2, \ldots, v_k$. Set $S(v_i) = C_i$. Set $r$ as the parent of each $v_i$.
  Recursively call DHC on each $(v_i, C_i)$
  Return the tree rooted at $r$, and all $S(v)$.
```

Figure 4.4 illustrates the process of running Algorithm 4.1 and the resulting clustering tree.

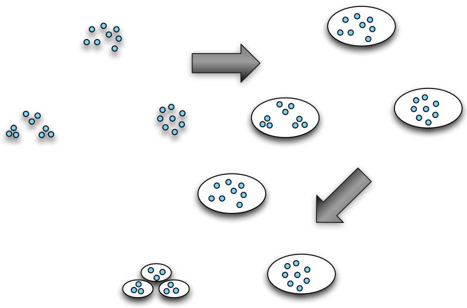


Figure 4.4: An illustration of DHC

Agglomerative Clustering

In the bottom-up view, we begin with all items in separate clusters and repeatedly select two clusters for merging. This method is called hierarchical agglomerative clustering (HAC). To define a bottom-up merge, we need some way to prioritize the merge operations: which clusters do we merge in the next step?

Intuitively, we want to first merge clusters that are very close to each other. This captures the idea of a hierarchical clustering as "zooming out" from the individual points. Assuming we have some way to measure distance between clusters, then the algorithm for HAC is straightforward:

Algorithm 4.2 Template for Hierarchical Agglomerative Clustering algorithm

```
$\mathcal{S} \leftarrow \{(x, \{x\}) \mid x \in X\}$
$\mathcal{C} = \mathcal{S}$
while $|\mathcal{C}| > 1$ do
    Find $(v, C), (v', C')$ in $\mathcal{C}$ that are closest.
    Create node $r$ and cluster $\hat{C} = C \cup C'$. Set $S(r) = \hat{C}$.
    Assign $r$ as the parents of $v$ and $v'$.
    Insert $(r, \hat{C})$ and remove $(v, C)$ and $(v', C')$ from $\mathcal{C}$. Insert $(r, \hat{C})$ into $\mathcal{S}$.
```

In a metric space, the distance between a pair of points is well defined, but what about the distance between two clusters ? There are many ways we could imagine defining the distance between two clusters, and each of these methods gives rise to a slightly different clustering.

Single Link. We can define the distance between two clusters C and C' as the distance between the closest pair of points, one in C, one in C': $\min_{x \in C, y \in C'} d(x, y)$. This is the most common method for doing HAC, and is also reminiscent of the k-center method we talked about in Section 2.3.

The single-link method is optimistic. Suppose x, x' are the closest pair of points in C, C' respectively. Imagine constructing a graph on the points by adding an edge between x, x' when we merge C, C'. Then at any point in the algorithm, the clusters consist of connected components in this graph. The assumption is that any two points that are connected should be in the same cluster. The resulting algorithm resembles Kruskal's algorithm for the minimum spanning tree closely; in fact if you were to run Kruskal's algorithm on the input and terminate it when there were k connected components left, you'd have a k-clustering that could be produced by single-link HAC.

One of the reasons single-link HAC is popular is because it is content to make local decisions to connect points together in a single cluster. This is particularly useful when clusters might be connected but don't have a canonical shape like a ball, as in Figure 4.5. In such a scenario, the pure distance-based methods we saw in Chapter 2 would fail because they're trying to carve space into balls. But HAC will not because it doesn't care about the shape of the cluster, merely its topology.

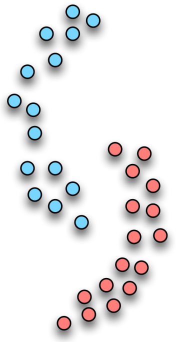


Figure 4.5: An example where single-link HAC can separate the blue points from the red points.

Complete-link. The single-link strategy, by virtue of being aggressive, can also chain clusters together when it shouldn't. The complete-link strategy is an attempt to fix this by defining the distance between two clusters as the distance between furthest away points, $\max_{x \in C, y \in C'} d(x,y)$. This prevents a transitive chain reaction from starting: two clusters are deemed to be close only if *all* their points are close to each other, rather than relying on just two points being close.

Average-link. A compromise between the two strategies above is the average-link approach: the distance between two clusters is the *average* distance between pairs of points, or $\frac{1}{|C||C'|} \sum_{x \in C, y \in C'} d(x,y)$

Centroids. The three methods above can be defined in a generic metric space. If instead we have points in a vector space endowed with a metric, a natural way to define the distance between two clusters is to measure the distance between the *centroids* of the clusters. This is equivalent to the average-link strategy when all points lie in a Euclidean space and the average is taken over the *squares* of the Euclidean distances between points.

4.4 Visualizing Hierarchical clusterings: Dendrograms

A hierarchical clustering has much more information in it than a mere partition. The tree structure is an important information channel and is what makes hierarchical clusterings so effective in an exploratory environment.

We can visualize trees in many ways. treevis.net, the definitive source for tree visualizations, lists two hundred and ninety two tree visualization methods[1]. But one of the most appealing ways to visualize hierarchical clusterings is the *dendrogram* (literally, "tree drawing").

Think again of the hierarchical clustering as a movie that starts with all points in their own clusters and ends with all points in a single cluster. As time progresses, clusters move towards each other at constant speed and merge. The dendrogram is a two-dimensional representation of this movie with points laid out along the x-axis and time measured along the y-axis.

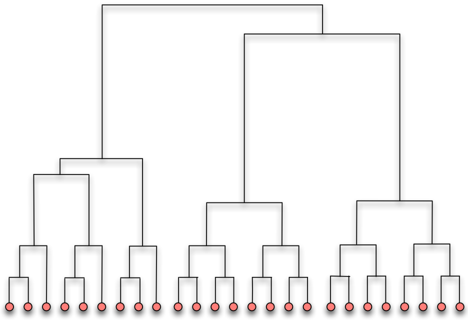


Figure 4.6: Dendrogram for the data in Figure 4.1

The dendogram is visually appealing because it does two things: first, it depicts the tree of merges, permuting the nodes so that there are no crossings. Second, and more importantly, it uses the lengths of edges as a visual marker to indicate the 'time of merge' for clusters. If we think of starting at time 0 and merging clusters, then a cluster that sticks around for a long time will have a tall edge connecting it to its parent, in comparison with a cluster that gets merged quickly.

[1] As of 17 September, 2016.

4.5 How good is hierarchical clustering?

Hierarchical clusterings are a compromise between getting the single best clustering for some fixed value of k and providing a collection of solutions for different values of k that have nice relationships to each other. How much is is lost in this compromise?

Suppose we take a particular hierarchical clustering and run the process of merging clusters bottom-up like a movie. We start off with all points in their own cluster. Call this clustering $\mathcal{C}_n$. As the movie progresses, two points merge into a new cluster. Now we have $n-1$ clusters: call this clustering $\mathcal{C}_{n-1}$. Continuing this process, we end with all points in a single cluster $\mathcal{C}_1 = X$ and a sequence of clusterings $\mathcal{C}_n, \mathcal{C}_{n-1}, \ldots, \mathcal{C}_1$.

The clustering $\mathcal{C}_k$ has k clusters. How does the cost of this clustering compare to the cost of the *best* clustering on k clusters, for any particular cost function ? And if we compare these two costs for all values of k, what is the worst possible situation we can encounter ?

The Bad News

It turns out that there is both good news and bad news when it comes to analyzing hierarchical clusterings. We begin with the bad. The result, due to Dasgupta and Long, combines known properties of the single-link method with new observations about complete-link and average-linkage methods.

Lemma 4.1 (Dasgupta and Long [2005]). *A single-link-based k-clustering can be a factor of k worse than the best k-clustering for the k-center objective function. For complete-link and average-link, the corresponding factor can be as bad as* $\log k$.

The result for single link is easy to explain: we describe here the construction given by Dasgupta and Long. The proof rests on exploiting the fact that single-link HAC operates *locally* and therefore cannot see potentially better clusterings coming from merging more distant clusters.

Let $\{x_1, \ldots, x_n\}$ be a set of points on the line, and let the distance between x_j and x_{j+1} be $1 - j\epsilon$, for some tiny $0 < \epsilon < 1/n$. As j increases, the distance between adjacent points gets closer and closer.

It is easy to see that single-linkage will start by merging x_{n-1} and x_n, and then will continue by merging the rightmost remaining point into this cluster. At the point when k clusters remain, these k clusters will be the points $x_1, x_2, \ldots x_{k-1}$ and the supercluster $\{x_k, \ldots x_n\}$. The radius of this supercluster is $\sum_{j=k}^{n}(1 - j\epsilon)/2$ which is at least $(n - k - \epsilon\binom{n}{2})/2$. If we set ϵ to be some small number that is less than $1/\binom{n}{2}$, then the radius of this supercluster is

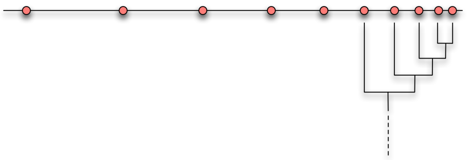


Figure 4.7: A bad example for single-link hierarchical clustering

more than $(n-k-1)/2$, and therefore the cost of the k-center clustering is at least $(n-k-1)/2$. However, by grouping the points in equal sized consecutive intervals we see that the optimal k-clustering has cost roughly $n/2k$: merely . The gap between the two costs is at least $(n-k-1)k/n$, which approaches k (for $n \gg k$). For complete linkage and average-linkage the construction is slightly more complicated, and yields a (weaker) $\log k$ bound .

This is not good news. It says that as we increase the number of clusters, the hierarchical clustering provided by any of the standard agglomerative methods will deteriorate in quality compared to the best clustering for any fixed k.

The Good News

It turns out that with a little bit of care, it is possible to produce a hierarchical clustering that *does* compare well with the best possible k-clustering for any k. We will develop this algorithm slowly via a series of refinements.

There is a trivial way to obtain a set of clusterings that are good for all k: merely compute the best k-clustering of the points for all values of k. Unfortunately, there might be no way to organize this set of clusterings into a single hierarchy. In particular, the set of cluster centers for some fixed k might not be a subset of the set of centers for some $k' < k$ (see Exercise).

This suggests another idea. Recall the FURTHESTPOINT algorithm (Algorithm 2.1) from Section 2.3. It works by constructing a sequence of centers $c_1, c_2, \ldots$ with the property that the first k centers yield a 2-approximation for k-center clustering, for all k. Let the centers of the desired k-clustering consist of $c_1, \ldots, c_k$, with all of the remaining points assigned to their nearest centers. This ensures that the centers can be organized hierarchically.

Unfortunately, even this does not yield a hierarchical clustering, because once a point is "promoted" to being a cluster center, it might "steal" a point away from another cluster.

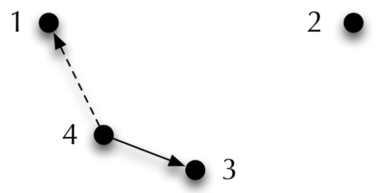


Figure 4.8: FurthestPoint picks the points in the numbered order. When 3 is promoted, 4 changes its assignment from 1 to 3.

Can we somehow retain the main idea of the clustering algorithm, and yet insist on a hierarchical solution? Let us reexamine the FurthestPoint algorithm. At the i^{th} step, we have centers $C = c_1, \ldots, c_i$, and we find a point x such that $\min_{c_i \in C} d(x, c_i)$ is maximized. This point x then becomes the new center c_{i+1}. Suppose that some center $c_\ell, \ell \leq i$ is the center closest to x. We define the *parent* $\pi(c_\ell) = c_k$. The graph consisting of the points P and the edges $(x, \pi(x))$ is a tree T_π (because each point other than the first has exactly one parent).

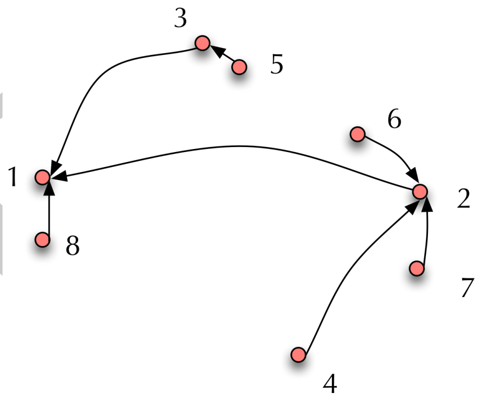


Figure 4.9: Example of the parent structure

Number all the points in the order in which FurthestPoint adds them to C. If we delete the edge $(2, \pi(2))$ from T_π we get two connected components in the tree. If we then delete $(3, \pi(3))$, we get three components. In other words, deleting the $k-1$ edges $(2, \pi(2)), (3, \pi(3)), \ldots, (k, \pi(k))$ yields a clustering with k components. Furthermore, all of these clusterings form a hierarchy because each new cluster is formed by splitting one of the previous clusters.

However, by insisting on a hierarchical solution we have lost the approximation guarantees. The clusters formed this way can connect points that are quite far from each other .

The key insight of Dasgupta and Long [2005] is that the idea of a parent function is not limited to the parents produced by FurthestPoint. Indeed, all we need to ensure a hierarchical structure is

- An ordering of the points from 1 to n
- A parent function $\pi : [n] \to [n]$ such that $\pi(i) < i$.

at which point deleting the edges $(2, \pi(2)), (3, \pi(3)), \ldots, (k, \pi(k))$ will give us the desired k-clustering.

Let us return to the FurthestPoint parent function π. As we noted, it doesn't necessarily yield a good hierarchical clustering but it does have valuable information. Let $R_i = d(i, \pi(i))$ be the distance from a point to the place where it attaches to the tree. The key property of the centers produced by this algorithm is that the value R_{k+1} is a good approximation to the optimal clustering of the points into k clusters. We will divide the points into levels based on the value of R_i. Set $R^* = R_2$, and set $L_0 = \{1\}$. We know that R^* is a good estimate of the cost of clustering the points using the (single) center in L_0.

Now consider all of the points whose attachments to the tree are sufficiently large: set $L_1 = \{i \mid R^*/2 < R_i \leq R^*\}$. Intuitively, these are all points that are quite far from their parents, and should be split off in a more refined clustering. In a similar manner, let $L_2 = \{i \mid R^*/4 < R_i \leq R^*/2\}$, and continue to define $L_3, L_4, \ldots$. By construction, every point of the input must be within distance $R^*/2^j$ of $L_0 \cup L_1 \cup \cdots \cup L_j$. Define the *level* $\ell(j)$ of a node to be the set L_j that it belongs to.

We can now construct a modified parent function π'. Consider any point i in L_j. Rather than attaching it to its closest neighbor in $\{1, 2, \ldots, i-1\}$, we attach it to its closest neighbor in L_{j-1} and define π' as the resulting parent function. By induction, the distance from i to its closest neighbor in L_{j-1} is at most $R^*/2^{j-1}$. More generally, under π',the distance of any point i to its parent is at most $R^*/2^{\ell(i)-1}$.

Consider a k-center clustering obtained by deleting the edges $(2, \pi'(2))$, $(3, \pi'(3)), \ldots, (k, \pi'(k))$. Under the original parent function π, $R_{k+1} = d(k+1, \pi(k+1))$ is a 2-approximation of the true clustering cost. In order to show that the resulting clustering is good, we must show that the distance of any remaining point from its cluster center is close to R_{k+1}.

Consider any point i. Under π', it has a chain of parents $i \to i_0 \to i_1 \ldots \to i_t$ back to some center i_t. By the triangle inequality therefore,

$$d(i, i_t) \leq \frac{R^*}{2^{\ell(i)-1}} + \frac{R^*}{2^{\ell(i_0)-1}} + \ldots + \frac{R^*}{2^{\ell(i_{t-1})-1}}$$

Because each π' points to a node in a previous level, $\ell(i) > \ell(i_0) > \ldots > \ell(i_{t-1})$. Therefore, $d(i, i_t)$ is at most $2R^*/2^{\ell(i_{t-1})-1}$. But we know that $R_{k+1} \geq R/2^{\ell(i_{t-1})}$, and so we conclude that the distance from any point to its nearest cluster center is at most $4R_{k+1}$.

Theorem 4.1. *Let $C_1, C_2, \ldots, C_K$ be a k-clustering computed by deleting edges from the tree induced by the parent function π'. Then the cost of this clustering is at most 8 times the cost of the optimal k-center clustering.*

We can improve the analysis even further by optimizing the distances between successive levels. Instead of choosing $R^* = R_2$ and defining a range between R^* and $R^*/2$, choose $R^* = \alpha R_2$, and define the range between R^* and R^*/β. By setting $\beta = e$, Euler's constant, and $\alpha = \beta^s$, where s is chosen uniformly at random between $[0, 1]$. In this case the expected quality of the algorithm improves from 8 to $2e \approx 5.44$.

This algorithm gives us a hierarchy that supplies good k-center clusterings. But what about if we wanted a hierarchy of k-median clusterings? The recipe we used asks for three ingredients:

- An algorithm that can produce a sequence of centers $c_1, c_2, \ldots$ such that the set $c_1, \ldots, c_k$ is a good clustering for all k.
- An initial parent function π that establishes baseline for cluster costs.
- A way to construct a parent function π' such that the lengths of chains from points to their cluster center are bounded.

5 Visualizing High Dimensional Data

Always project

With apologies to E. M. Forster

Thus far, we have avoided discussing how we actually *visualize* clusterings. Obviously, when our data lies in a two- or three-dimensional space, it is easy to plot the points and use color-coding to visualize clusters, but we cannot do this if we have clusters in (say) ten-dimensional space.

If we believe that clustering is a powerful *exploratory* tool, then this is a problem. If I can't visualize the results of clustering, then it will be difficult to validate the computation, or easily determine where outliers might be lurking, or even understand the shape of the clusters.

So the (high) dimensionality of data makes it difficult to look at clusterings. But high dimensionality is a problem in many other ways. Most importantly, data in high dimensions suffers from *"the curse of dimensionality"*. This is a poetic way of observing that most computations involving geometry take time exponential in the dimension. This is a serious problem considering that a typical data set one might wish to cluster might have millions of points that lie in a hundred-dimensional space.

There are other problems that manifest themselves in high dimensions. While we won't be able to go into this in any detail (), data might lie in an ambient high dimensional space, but actually only populate a lower-dimensional subspace. For example, we might have objects that are described by three coordinates, but actually all lie on a straight line. This *intrinsic* dimensionality of the data is smaller, but we cannot exploit it because we cannot "see" this structure in the ambient space.

5.1 Principal Component Analysis

Points in high dimensions are hard to visualize and manipulate. We would like to *reduce* the dimensionality of the data. But wouldn't we lose useful information in doing so?

Imagine a set of points (depicted in Figure 5.1) in two dimensions that capture some *noisy* linear relationship between two variables. The idea behind principal component analysis (PCA) is to transform the data so that the relationship can be easily identified. In other words, we'd like to transform the data so that the picture looks more like Figure 5.2. The advantage of do-

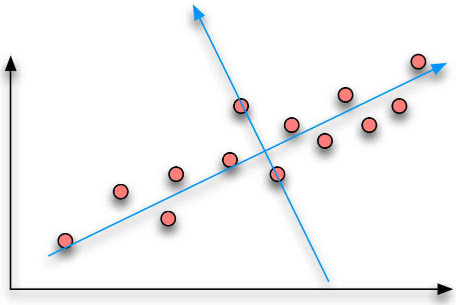


Figure 5.1: Directions of variance

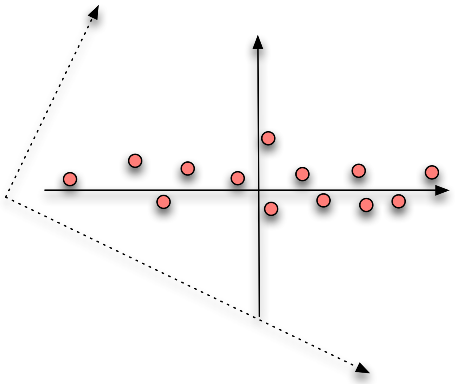


Figure 5.2: Directions of variance: figure rotated and scaled from Figure 5.1

ing this is two-fold. Firstly, we've identified the key dimension that carries the signal in the data, and secondly we've made the different dimensions independent of each other – essentially *decorrelating* the dimensions. This first aspect is important because it allows us to find a good approximation to the actual noisy representation; the second is important because with independent features we actually have a better geometric representation of the data in a Euclidean space.

In order to do this, PCA starts with a simple idea: find the direction along which there is maximum variation in the data. Intuitively, this is the direction that holds most information about the data. In Figure 5.1, this is the line marked in blue. Let us now formalize this idea.

We are given points $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$. We will treat each point x_i as a $d \times 1$ vector $\boldsymbol{x}_i$, and the set X as a $d \times n$ matrix $\mathbf{X}$. Let $\boldsymbol{\mu} = \frac{1}{n} \sum \boldsymbol{x}_i$ be the centroid of X. Fix a direction $\boldsymbol{u}$, where $\|\boldsymbol{u}\| = 1$. The projection of any point $\boldsymbol{x}$ along $\boldsymbol{u}$ can be written as $\boldsymbol{x}^\top \boldsymbol{u}$, and so we wish to find a direction $\boldsymbol{u}$ such that

$$\sum (\boldsymbol{x}^\top \boldsymbol{u} - \boldsymbol{\mu}^\top \boldsymbol{u})^2$$

is maximized. Note that by subtracting the centroid $\boldsymbol{\mu}$ from each point $\boldsymbol{x}$ we are computing a translation-invariant quantity that does not depend on where the points are—if we did not do this we could record an artificially high variance with points that just happen to be far from the origin.

Another way to think about this is that without loss of generality we can set $\boldsymbol{\mu} = 0$ by translating X. We can also express the (scalar) quantity $(\boldsymbol{x}^\top \boldsymbol{u})^2$ as the vector norm $\|\boldsymbol{x}^\top \boldsymbol{u}\|^2$. Noting that we can write $\|\boldsymbol{v}\|^2$ as $\boldsymbol{v}^\top \boldsymbol{v}$, our goal is now to compute

$$\max_{\boldsymbol{u}, \|u\|=1} \sum \boldsymbol{u}^\top \boldsymbol{x}_i \boldsymbol{x}_i^\top \boldsymbol{u}$$

We can further simplify this expression by noting that the $d \times d$ matrix $\frac{1}{n-1} \sum \boldsymbol{x}_i \boldsymbol{x}_i^\top = \frac{1}{n-1} \mathbf{X}\mathbf{X}^\top$ is a constant, called the *covariance matrix*. Denoting it by $\mathbf{C}$, and dropping the constant $n - 1$ that doesn't affect the optimization, the expression we obtain is

$$\max_{\boldsymbol{u}, \|u\|=1} \boldsymbol{u}^\top \mathbf{C} \boldsymbol{u}$$

But this is one definition of the eigenvector of $\mathbf{C}$ with the largest eigenvalue![1]

Denote this by $\boldsymbol{u}_1$ and let the corresponding eigenvalue be λ_1. If we project the points onto $\boldsymbol{u}_1$, the "coordinate" of each point is the projection $\frac{1}{\sqrt{\lambda_1}} \boldsymbol{x}_i^\top \boldsymbol{u}_1$.

Of course we would like an embedding of the points into more than one dimension. Since $\mathbf{C}$ is symmetric, we can express it in the form

$$\mathbf{C} = \mathbf{U} \boldsymbol{\Lambda} \mathbf{U}^\top$$

where $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_d)$ is the diagonal matrix of the eigenvalues of $\mathbf{C}$ in decreasing order and $\mathbf{U} = (\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_d)$ is the (orthonormal) matrix of eigenvectors. As before, the j^{th} coordinate of a point $\boldsymbol{x}$ in this embedding is the projection $\frac{1}{\sqrt{\lambda_j}} \boldsymbol{x}^\top \boldsymbol{u}_j$. More compactly, let $\tilde{\mathbf{X}} = \boldsymbol{\Lambda}^{-1/2} \mathbf{U}_k^\top \mathbf{X}$, where $\mathbf{U}_k = (\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k)$. Then $\mathbf{U}_k$ represents the *principal components* of $\boldsymbol{X}$, and $\tilde{\mathbf{X}}$ is the k-dimensional representation of $\boldsymbol{X}$. We summarize the entire procedure in Algorithm 5.1.

[1] This is a consequence of the fact that $\mathbf{C}$ is symmetric, which allows us to apply the well-known min-max (or variational) theorem from linear algebra. .

Algorithm 5.1 Principal Component Analysis

```
Input: $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\} \subset \mathbb{R}^d$. Integer $1 \le k < d$.
  $\boldsymbol{\mu} = \frac{1}{n} \sum \boldsymbol{x}_i$
  for all $i$ do {Center points}
    $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i - \boldsymbol{\mu}$
  $\mathbf{C} \leftarrow \frac{1}{n} \mathbf{X}\mathbf{X}^\top$
  $(\boldsymbol{u}_i, \lambda_i) \leftarrow$ top $k$ eigenvalues and eigenvectors of $\mathbf{C}$.
  $\boldsymbol{\Lambda} \leftarrow \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_k)$
  $\mathbf{U}_k = (\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k)$
  $\tilde{\mathbf{X}} \leftarrow \boldsymbol{\Lambda}^{-1/2} \mathbf{U}_k^\top \mathbf{X}$
```

The resulting set of points $\tilde{\mathbf{X}}$ is "maximally informative", but we can say even more about how it preserves the structure of $\mathbf{X}$. Consider the $n \times n$ *similarity matrix* associated with $\boldsymbol{X}$:

$$\mathbf{S} = \mathbf{X}^\top \mathbf{X}$$

i.e where each entry $\mathbf{S}_{ij} = \boldsymbol{x}_i^\top \boldsymbol{x}_j$. Then it is not hard to show that

$$\tilde{\mathbf{X}} = \arg \min_{\mathbf{X} \text{ of rank } k} \|\mathbf{X}^\top \mathbf{X} - \mathbf{S}\|_F$$

where $\|\mathbf{A}\|_F = \sum_{i,j} a_{ij}^2$ is the Frobenius norm of a matrix[2]. In other words, $\tilde{\mathbf{X}}$ preserves the similarities among points of X the best among all possible k-dimensional representations of X.

We set out two goals for a dimensionality reduction procedure. The first was to identify the key dimensions that capture signal in the data. The second was to "decorrelate" the data by ensuring that individual dimensions of the resulting data representation were orthogonal. It can easily be shown (see the Exercises) that if the covariance matrix of the resulting point set $\tilde{\mathbf{X}}$ is diagonal, which means that we have decorrelated all the variables successfully.

Computational Issues and the SVD

Computing the principal components of a set of points is straightforward, involving computing the eigenvectors of the covariance matrix $\mathbf{C}$. However, C is a $d \times d$ matrix that must be explicitly computed, and if d is large, this

[2] The Frobenius norm of a matrix is the matrix equivalent of the ℓ_2 norm of a vector.

could be very expensive. An easier way to compute the desired projection is to use the *singular value decomposition* of $\mathbf{X}$.

Any $d \times n$ matrix $\mathbf{X}$ of rank r admits a representation in the form

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$$

where

- $\mathbf{\Sigma}$ is an $r \times r$ diagonal matrix of *singular values*
- $\mathbf{U}$ is a $d \times r$ orthonormal matrix of *left singular vectors*
- $\mathbf{V}$ is an $n \times r$ orthonormal matrix of *right singular vectors*.

We can then write $\mathbf{X}\mathbf{X}^\top = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^\top$. Comparing this to C, we can see that the projection vectors we need are precisely the first k left singular vectors $\mathbf{U}_k$ of the singular value decomposition. Moreover, the resulting matrix $\tilde{\mathbf{X}} = \mathbf{\Sigma}\mathbf{V_k}^\top$.

5.2 Multidimensional scaling

PCA starts with a set of points in a Euclidean space and projects them down a lower dimensional space that (approximately) preserves the structure of the set. But what if we don't even have that starting set?

Consider the setting of Chapter 2. Data is presented to us as points in a metric space and so all we have is a matrix of distances between points. We now have two problems. First, we need a way to *embed* the points in some Euclidean space (of any dimension) in a way that distances between points are approximately preserved. Second, we need to find a way to represent these points in a low dimensional space for easy visualization.

The second task is easy: after all, that's exactly what PCA is for. But what about the first task? This is where multidimensional scaling (MDS) comes in. It allows us to create a *realization* of a matrix of distances as the distances between points.

To understand the difficulty of doing this, let us consider a simple example. Suppose we are given the following distance matrix between three points

$$D = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

It is easy to see that this can be realized by three points that form an equilateral triangle of unit side length. But suppose we now add another point that is equidistance from all of these, resulting in the following distance matrix

$$D = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Now it is a little harder, and some thought will convince you that there is no way to embed these points in the plane so that all distances are maintained exactly [3]. In general, an arbitrary metric will *not* embed nicely in a Euclidean space, and so we need to consider approximate forms of representation.

The approach that MDS takes is that it does not actually try to preserve distances when doing an embedding. Rather, it does the following three things:

(a) It constructs a *similarity* matrix from the original set of distances

(b) It then finds an explicit set of points whose inner products match this similarity matrix as closely as possible, via a PCA-like approach.

Let the input $n \times n$ distance matrix be D. MDS starts by assuming that these distances come from some Euclidean space. In other words, it assumes that there exists some set of points $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ such that

$$D_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|$$

Notice that such a representation is not unique: we can add any vector $\boldsymbol{v}$ to all the points and obtain the same distance matrix. So we will assume that this set of points is centered at the origin:

$$\sum \boldsymbol{x}_i = \mathbf{0}$$

Our goal is now to recover the $\boldsymbol{x}_i$ from the D_{ij}. As it turns out, this is not that easy. Rather, we'll ask for something more modest: that we can recover the matrix of *similarities* $\mathbf{S} = \mathbf{X}^\top \mathbf{X}$, where each entry $S_{ij} = \langle \boldsymbol{x}_i^\top \boldsymbol{x}_j \rangle$.

Expanding D_{ij},

$$\begin{aligned} D_{ij}^2 &= \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2 \\ &= \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{x}_j\|^2 - 2\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle \\ &= \langle \boldsymbol{x}_i, \boldsymbol{x}_i \rangle + \langle \boldsymbol{x}_j, \boldsymbol{x}_j \rangle - 2\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle \\ &= S_{ii} + S_{jj} - 2S_{ij} \end{aligned}$$

[3] If we go one dimension higher, this becomes easy.

Our goal is to express S_{ij} as a function of D_{ij}^2. To this end, let us introduce the notation $a_{\cdot,j} \triangleq \sum_i a_{ij}$ and the notation $a_{i,\cdot} \triangleq \sum_j a_{ij}$. Similarly, we will use $a_{\cdot,\cdot} \triangleq \sum_i \sum_j a_{ij}$.

$$\begin{aligned}
\sum_j D_{ij}^2 = D_{i,\cdot}^2 &= \sum_j \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{x}_j\|^2 - 2\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle \\
&= n\|\boldsymbol{x}_i\|^2 + \sum_j \|\boldsymbol{x}_j\|^2 - 2\langle \boldsymbol{x}_i, \sum_j \boldsymbol{x}_j \rangle \\
&= n\|\boldsymbol{x}_i\|^2 + \sum_j \|\boldsymbol{x}_j\|^2 \\
&= nS_{ii} + \sum_j \|\boldsymbol{x}_j\|^2
\end{aligned}$$

because $\sum_j \boldsymbol{x}_j = 0$. A similar calculation yields

$$D_{\cdot,j}^2 = \sum_i \|\boldsymbol{x}_i\|^2 + nS_{jj}$$

and a further summation yields

$$D_{\cdot,\cdot}^2 = 2n \sum_l \|\boldsymbol{x}_l\|^2$$

We can now write

$$S_{ij} = -\frac{1}{2}(D_{ij}^2 - \frac{D_{i,\cdot}^2}{n} - \frac{D_{\cdot,j}^2}{n} + \frac{D_{\cdot,\cdot}}{n^2})$$

There is a convenient matrix representation of this expression. Let

$$\mathbf{H} = \mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^\top$$

Post-multiplying a matrix $\mathbf{M}$ by $\mathbf{H}$ has the effect of replacing each element by its value less the average of its row. Pre-multiplying does the same thing for columns. Noting that $\frac{D_{i,\cdot}^2}{n}$ is merely the row average of the matrix of squared distance values (and $\frac{D_{\cdot,j}^2}{n}$ is the column average), we can rewrite $\mathbf{S}$ more compactly as

$$\mathbf{S} = -\frac{1}{2}\mathbf{H}\mathbf{D}^{(2)}\mathbf{H}$$

where $\mathbf{D}^{(2)}$ is the matrix of squared distance values D_{ij}^2. [4]

Now that we have a similarity-based representation $\mathbf{S}$ of the input distances, our goal is find a collection of n points in k dimensions whose similarity matrix is as close to $\mathbf{S}$ as possible. Let the $k \times n$ matrix $\tilde{\mathbf{Y}}$ represent these points. Then

$$\tilde{\mathbf{Y}} = \arg\min \|\mathbf{Y}^\top \mathbf{Y} - \mathbf{S}\|_F$$

Using the same technique as in Section 5.1, the desired $\tilde{\mathbf{Y}}$ is given by:

$$\tilde{\mathbf{Y}} = \mathbf{\Sigma}^{1/2}\mathbf{V}_k$$

where $\mathbf{S} = \mathbf{V}\Sigma\mathbf{V}^\top$ and as usual $\mathbf{V}_k$ denotes the first k columns of $\mathbf{V}$ assuming that all eigenvalues in Σ are sorted in descending order.

Notice that in order to compute $\mathbf{\Sigma}^{1/2}$, it must be the case that $\mathbf{S}$ is positive semidefinite. This is guaranteed to be true by construction if the original distances were Euclidean. However, in general this might not be true, in which case $\mathbf{S}$ might have negative eigenvalues. In this case, the typical practice is to take as many eigenvectors as needed that correspond to positive eigenvalues.

5.3 Nonlinear embeddings and t-SNE*

MDS and PCA are *linear* operators. In each case, we take a linear transformation of the input (distances or points) in order to find the desired structure. If the inherent structure of the points is not linear, these methods will not work.

Consider a set of points that lie on a spiral centered at the origin. While these points lie in an underlying one-dimensional curve, no linear embedding will be able to lay them out on a straight line or even identify the lower-dimensional structure.

There are a variety of *non-linear* dimensionality reduction tools that start with a collection of points and attempt to find a low-dimensional representation. Intuitively, all of these methods assume that the points lie on a low-dimensional surface (a *manifold* and attempt to find a "stretched-out" representation of these manifold.

[4] We use this notation to distinguish the matrix from D^2, which would denote normal matrix multiplication.

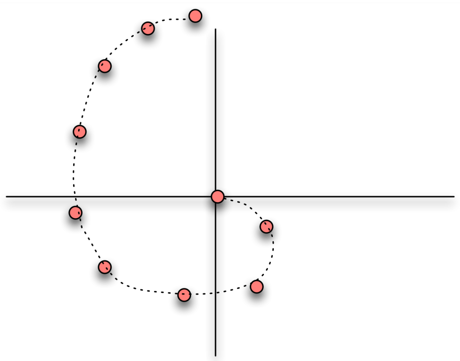


Figure 5.3: A two-dimensional spiral

One such method is called t-stochastic neighbor embedding (t-SNE) **?**. It bears a strong resemblance to the linear methods we've seen thus far, in that it proceeds by constructing a similarity matrix associated with the input points and then attempts to find a set of points in a lower-dimensional space that has the same similarity structure.

The challenge with all such methods is determining this similarity. The proximity of two points in a Euclidean space might have no bearing on their similarity on the underlying surface (see Figure 5.4).

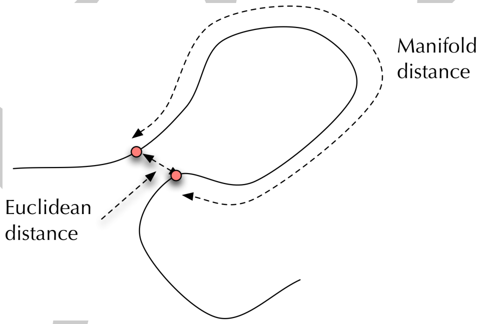


Figure 5.4: Why Euclidean distance and manifold distance are unrelated

We need more information on how points might be related to each other. In t-SNE, this is done by assuming that each point is the center of a probability distribution. Let the input points be $x_1, \ldots, x_n \in \mathbb{R}^d$. The similarity between x_i and x_j can then be estimated by asking how likely it is that a x_j could be drawn from a distribution centered at x_i. Let $p_i(y)$ be a probability distribution with mean at x_i, *relative to the other points*. This quantity be written as

$$p_{j|i} = \frac{p_i(x_j)}{\sum_{k \neq j} p_i(x_k)}$$

To make this symmetric, we can ask the same question the other way around, obtaining the quantity $p_{i|j}$. Taking a normalized[5] average, we set

$$p_{ij} = \begin{cases} \frac{p_{i|j}+p_{j|i}}{2n} & i \neq j \\ 0 & i = j \end{cases}$$

The matrix $\mathbf{P} = (p_{ij})$ is the associated similarity matrix for the points. Assume that the desired low-dimensional representation consists of the points $y_1, \ldots, y_n \in \mathbb{R}^k$. We can similarly associate a probability distribution q_j with each y_j, and construct a similarity matrix $\mathbf{Q} = q_{ij}$ as before.

We would like the two matrices to be close to each other. We can think of the p_{ij} as probabilities in a joint distribution: indeed, $\sum_{i,j} p_{ij} = 1$. One way to compare two probability distributions is to use the *Kullback-Leibler divergence*:

$$d_{\mathrm{KL}}(\mathbf{P}, \mathbf{Q}) = \sum_{ij} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

And so the problem of nonlinear dimensionality reduction reduces to the following minimization:

$$\min_{y_1, \ldots, y_n} \sum_{ij} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

for which local minima can be found via gradient descent as long as we have closed-form expressions for the p_i, q_i.

And what of these p_i, q_i? The distribution p_i is typically set to be an isotropic Gaussian:

$$p_i(x) \propto \exp(-\frac{\|x_i - x\|^2}{\sigma_i^2})$$

In the original paper on stochastic neighbor embeddings**?**, the q_i were also set to be Gaussians. However, these provided an unsuitable fit for the data[6] and in t-SNE, a heavy-tailed Cauchy (or Student's t-) distribution is used instead:

[5] The division by n is a technical detail that ensures that each quantity is sufficiently large, and is not important for the development of the basic framework.

[6] See the discussion in **?**.

$$q_i(y) \propto \frac{1}{1 + \|y_i - y\|^2}$$

6 Graph Clustering

Let it flow, let it flow, let it flow.

In the clustering problems we considered thus far, the distance (or similarity) between two objects is supplied directly as part of the problem statement. But suppose you would like to cluster friends into friend groups—some are family, some of people you grew up with, others are colleagues at work, and so on. What distance function should you use? You can, of course, try to define a notion of distance between a pair of friends, and then attempt to use one of the clustering objectives we've already discussed.

But this ignores an important consideration: that the distance is something that should *emerge* from the relationships between entities, rather than being imposed externally. In other words, the data here is naturally represented as nodes in a graph, with an edge capturing a specific relationship (for example, friendship) between nodes. In fact, when we talk of a *social network*, whether it be from Facebook, Twitter, or LinkedIn, this is the graph we are referring to.

Our goal is now to define a notion of a cluster on a graph where an edge represents some notion of proximity. One approach is to say that two friends, Alice and Dean are in the same cluster if there is a chain of pairwise relationships (i.e a path) between them. That is if Alice is friends with Bob, and Bob is friends with Charlotte, and Charlotte is friends with Dean, then we will put Alice and Dean into the same cluster.

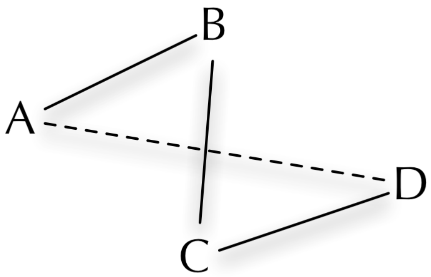


Figure 6.1: Example of a connected component as a cluster

Recall that a *connected component* of a graph is a (maximal) set of vertices that all have paths to each other. In effect then, defining a cluster in terms of chains means that each connected component of a graph is a cluster.

As it turns out, while this clustering objective is natural, it is not very useful. Large social graphs tends to have giant connected components that don't really represent a common set of interests. Moreover, friendships are not usually transitive: Alice and Bob might both be friends with Charlotte, but might not know each other. Placing all three of them in a single cluster would then not be meaningful [1]. Indeed, research suggests that social networks like Facebook have single digit average diameters.

Suppose we tried a more conservative strategy to define clusters. Let us require that for a set of nodes to form a cluster, all of them should have direct relationships with each other. Translating this to the language of graphs, we are requiring that the set of nodes that form a clustering induce a *clique* (a fully connected set of vertices).

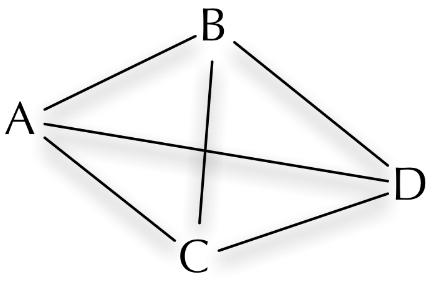


Figure 6.2: Example of a clique as a cluster

Of course we have now swung from being overly liberal in the definition of a cluster to being overly conservative. Most clusters that we would find in a typical graph under this definition would be very small, because even missing one edge could render a cluster invalid.

While neither of the two notions above appear to characterize a cluster in a graph very well, they do capture important aspects of a good cluster. Clusters based on connected components capture the idea of *completeness* – that everyone has the chance to be part of a cluster. Clusters based on cliques embody the idea of *cohesion* – that a cluster should be internally more well-connected than the rest of the graph.

To define a single objective we need to balance cohesion and completeness. We first define it for a single cluster. Let $G = (V, E)$ be a graph, with

[1] We note in passing that relationships are also not necessarily symmetric: consider for example the relationship "A follows B on Twitter". We can model asymmetric relationships with directed edges in a graph, but defining natural clusters becomes much harder.

nodes V and edges E. Suppose we have identified a group of people, $S \subseteq V$ as forming a natural friend cluster. For technical reasons, assume that the group comprises less than half the graph i.e that $|S| < |V|/2$. We define the cost of the cluster S as:

$$\phi(S) = \frac{|\{(i,j) \in E | i \in S, j \in V \setminus S\}|}{|\{(i,j) \in E | i \in S\}|}.$$

Examining $\phi(S)$ closer, we see that the numerator counts the number of friend relationships where one friend is inside the group, S, and the other outside the group. On the other hand, the denominator counts the total number of friendship relationships by those inside the group. Thus a complete cluster will have a small numerator and a cohesive cluster will have a large denominator, which together reduce $\phi(S)$.

There are many ways to convert this intuition into a formal clustering objective. For example, we might wish to find a partition $C_1, C_2, \ldots, C_k$ where $\max_i \phi(C_i)$ is minimized. Or we might fix a threshold λ and require that for all clusters $C_i, \phi(C_i) \leq \lambda$. While these problems are intractable in general, they lend themselves to a natural "peeling" strategy:

(a) Find a subset S with minimum $\phi(S)$

(b) Remove ("peel off") the elements of S from V and repeat.

This greedy heuristic reduces the general problem of finding a high quality clustering to finding a high-quality *cluster*. It also reveals a fascinating connection to random walks which we explore next.

Random Walks

A brute force approach to finding a cluster with small $\phi(S)$ would be to try all possible subsets S of V. Since there are $2^{|V|}$ such subsets, this approach does not scale. Instead, let us determine what properties might yield a cluster with low $\phi(S)$.

Consider the following process. We start at some node $v \in V$ in the graph. We look at all of the edges incident on the node, choose one of them uniformly at random, and follow it to some node u. At the next time step we repeat the process at u. This process is known as a *random walk* in the graph.

Suppose we start a random walk inside a set S with high $\phi(S)$. Since the number of edges leaving S is large (compared to the total number of edges inside S) the random walk is likely to leave S, a phenomonen we refer to as *escaping* S. On the other hand, if we start a random walk inside a set S with low $\phi(S)$ then the random walk is likely to stay inside the set. The quantity ϕ is often referred to as the *conductance*[2]—when $\phi(S)$ is small, a random walk that enters S has a hard time escaping, therefore S has *low conductance*. On the other hand when $\phi(S)$ is large, we say that S has high conductance. See for example, the sets in Figure 6.3.

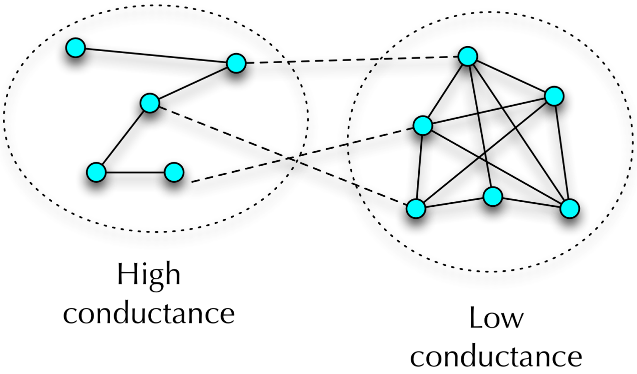


Figure 6.3: An example of high and low conductance subsets in a graph

We can make the argument above even more precise. Suppose we pick some set $S \subset V$ with conductance $\phi(S)$. We first pick a node $v \in S$ with probability proportional to its degree, d_v, and then start a random walk from v. Then the probability that this random walk has not left S after t steps is at least $1 - t \cdot \phi(S)$.

Let $vol(S) = |\{(i,j) \in E | i \in S\}| = \sum_{v \in S} d_v$ be the sum of the degrees of vertices in S, also known as the *volume* of S. In order to prove this statement we consider two facts. First, we bound the probability that a non-escaped random walk is at some node $u \in S$ is at most $d_u/vol(S)$. The proof follows by induction. Before the walk begins, the statement is true, since we chose the starting vertex with probability exactly $d_u/vol(S)$.

Now suppose the statement holds after t rounds. If the random walk is at node v after $t+1$ steps, that means after t steps it was at some node u, which is a neighbor of v. The probability that it was at node u is at most $d_u/vol(S)$ by the inductive hypothesis. Since the walk choses one outgoing edge of u at random, the probability that it transitioned to v is $1/d_u$. Therefore, the

[2] This is by analogy with electrical circuits – we can think of the random walk as the motion of electrons in a network of unit resistances.

total probability that a random walk ends up at v is at most:

$$\sum_{u:(u,v)\in E} \frac{d_u}{vol(S)} \cdot \frac{1}{d_u} = \sum_{u:(u,v)\in E} \frac{1}{vol(S)} = \frac{d_v}{vol(S)}$$

Now consider the probability that a random walk escapes S at time t. If the walk is at some node v, and there are ℓ edges from v leaving S, then the walk escapes with probability ℓ/d_v. Combined with the fact that the walk is at node v with probability at most $d_v/vol(S)$, the total chance that the walk escapes from v is at most $\ell/vol(S)$. Taking the sum over all nodes, we have:

$$\sum_{v\in S} \frac{|\{(v,w) \in E | W \notin S\}|}{d_v} \cdot \frac{d_v}{vol(S)} = \frac{|\{(v,w) \in E | V \in S, W \notin S\}|}{vol(S)} = \phi(S)$$

Therefore at every step the walk escapes with probability at most $\phi(S)$, and after t steps it has not escaped with probability at most $1 - t\phi(S)$. We note that a more refined analysis can give an even tighter bound on the probability of staying, showing that it is at least $(1 - \phi(S)/2)^t$.

Algorithms

The relationship between conductance and random walks suggests a strategy to find a cluster S to peel away. Let us assume that we know a vertex $v \in V$ that belongs to this cluster (this assumption is usually removed by trying all vertices). If S has low conductance $\phi(S)$ then we would expect that a random walk starting at v will visit most (or all) of the vertices in S before "escaping" to the rest of G. One idea is therefore to run a random walk starting at v for T time steps (for some time T), and define S as the set of vertices visited.

The formal relationship between conductance and random walks tells us that if we pick T to be on the order of $1/\phi$, then we can guarantee that the walk has not yet escaped S. However, we don't know ϕ ahead of time. What we will do instead, is to pick the length of the walk at random as well, with a bias towards shorter walks.

Suppose we pick the length of the walk, from a geometric distribution. That is probability that $T = k$ is distributed proportionally to $(1 - \alpha)^k$ for some $0 < \alpha < 1$. This obviously biases T towards shorter walks, but also has another natural interpretation.

Consider a random walk ' where starting from v, the walk can *teleport* back to v at any point with probability α. Formally, this merely means that at a node of degree d, each neighbor is visited with probability $(1 -$

$\alpha)/d$, and v is visited with probability α. Note that this is equivalent to first picking T from the geometric distribution is above, and then running the walk for T steps before resetting. This walk is very well known. In the resulting stationary distribution, the probability of being at any node u is called the *personalized pagerank* of u. Intuitively, the parameter α ensures that the effective length of a path is roughly $1/\alpha$ in expectation. This means that nodes closer to v will naturally appear more frequently in the walk, which is a desirable outcome[3].

Now that we have a random walk that can find nodes close to v, how do we quantify closeness (and therefore membership in the cluster)? For a node u, let $q(u)$ denote the probability that personalized pagerank walk ends at u. We are interested in the value of $q(u)$ *relative* to the probability of ending up at u without restarts (which from remarks above we know to be the degree $d(u)$). Motivated by this, let $q'(u) = q(u)/d(u)$ be the normalized personalized pagerank. Order the nodes in order of decreasing $q'(u)$ i. $q'(u_1) \geq q'(u_2) \geq \ldots \geq q'(u_n)$. Our intuition dictates that u_1 is more likely to be in the same cluster as v than u_2, and u_2 is more likely to be in the cluster than u_3. Thus we only need to pick a threshold q^* and place all nodes u with $q'(u) \geq q^*$ in the same cluster as v, and all other nodes in a different cluster.

Here we just try all possible values for q^*, and check which one yields a set with the lowest conductance. Known as a *sweep cut*, this is equivalent to checking the conductance of $\{u_1\}, \{u_1, u_2\}, \{u_1, u_2, u_3\}$, and so on. Thus instead of looking for $2^{|V|-1}$ possible clusters, we only consider the $|V| - 1$ clusters.

Remarkably, we can prove that this simple algorithm achieves a nearly optimal solution **?**

Theorem 6.1. *For an undirected connected graph $G = (V, E)$ on n nodes, let et $S^* \subset V$ be the set with the lowest conductance, and ϕ^* be the conductance of S^*. Then for $\alpha \geq 16\phi^*$, the above algorithm finds a set with conductance ϕ such that:*

$$\phi = O(\sqrt{\phi^* \log |S^*|}) = O(\sqrt{\phi^* \log n}).$$

In practice, computing the exact personalized pagerank of a node is computationally intensive. However, what we can do instead is to compute the empirical distribution by starting many nodes at v, letting them run for a number of timesteps and recording where they end up. These empirical

[3] If, instead of returning to v, we had the walk jump to a random node in the graph G, the stationary distribution represents the global pagerank of the node u.

personalized pagerank vectors can be used instead during the sweep cut, and still yield provably good approximations to the global lowest conductance cut.

The above algorithm gives us a way to find a single good cluster. A natural approach for graph clustering is then to remove this cluster from the graph, and recurse on the remainder as necessary. While there are no formal guarantees for this heuristic, it is a practically effective way to find clusters.

Directed Graphs

The personalized pagerank based algorithm above is specific to undirected graphs. However, a very similar approach can be used when the graph itself is strongly connected, that is, there is a directed path from every node u to every other node v.

The main difference from the undirected case is we no longer use the regular random walk as the basis of our analysis. Instead, consider the lazy random walk W, which, when moving from node u, with probability 1/2 stays at u, and with the remaining probability picks one of the outgoing edges at random.

We compute two stationary distributions. Let π_W be the stationary distribution of W, and $\pi_W(u)$ its value at node u. Since the graph is strongly connected, this distribution is well defined. We then compute the personalized pagerank equivalent, that is the stationary distribution of a walk that at every time step returns to a given node v with probability α, and with probability $(1-\alpha)$ follows W. For a node u, let $q(u)$ be the probability that this walk ends up at u.

As in the undirected case, we first order the vertices by their value of $\frac{q(u)}{\pi_W(u)}$ and then perform a sweep cut over the resulting sets, evaluating the conductance of the two vertices with the highest values, then the top three vertices, and so on. Once again we can prove that the best set found by this method is approximately optimal.

7 Correlation Clustering

The enemy of my enemy is my friend

Ancient Proverb

In the last chapter we learnt how to formulate a clustering problem when the only information we have on our data is the relationship between items, expressed in the form of a graph. This form of relationship information is *positive*: we can either encode a connection between two entities in the form of an edge, or we can use the absence of an edge as a lack of connection.

But suppose we wanted to express both positive and negative connections between entities? For example, when clustering fruit we may consider that lemons and bananas should belong to the same cluster because both are yellow, and lemons and limes should belong to the same cluster because both are citrus fruits, but then insist that bananas and limes be in different clusters. When given inconsistent information, our goal is either to maximize the total number of agreements, or, equivalently, minimize the total number of disagreements. In the example above, putting all three fruit in the same cluster has two agreements and one disagreement. Putting lemons and limes together, and bananas separate also leads to two agreements: (i) lemons and limes are in the same cluster, (ii) limes and bananas are in different clusters, but also has one disagreement: lemons and bananas are in different clusters. We call the class of clustering questions arising out of this formulation *correlation clustering*.

A pleasant property of correlation clustering is that we never specify the number of clusters that we are looking for. Instead, the correct number of clusters emerges naturally. For example, if all of the items are similar to each other, the optimum solution returns all of the elements in one cluster, if all are dissimilar from each other, then each point is its own (singleton) cluster. This is in direct contrast to other clustering methods which ask for the number of clusters, k, as the input to the problem.

An instance of correlation clustering is then a graph $G = (V, E^+ \cup E^-)$, where the vertices, V represent the set of items to be clustered, E^+ denote pairs of items that should be placed in the same cluster, and E^- represent pairs that should be placed in different clusters.

Objective Function

There are two obvious ways to evaluate the results of correlation clustering. The first is to maximize agreements: count all of the positive edges within each cluster and all of the negative edges across clusters. The second is to minimize disagreements: count all of the positive edges across clusters, and negative edges within each cluster. A moment's reflection shows that the two are the same, the sum of the two objectives is always equal to the number of edges, $|E^+| + |E^-|$.

Since the sum is always fixed, the optimal solution will simultaneously maximize agreements and minimize disagreements. However, the same does not hold for approximately optimal solutions. Indeed a solution that is within a constant factor on one metric, can be very far from optimum on the other. To see this, consider the following simple algorithm for maximizing agreement.

Let S be a clustering that places each vertex into its own cluster, and T be a clustering that places all vertices into one cluster.

Lemma 7.1. *Either S or T give a 2-approximation to maximizing agreement.*

Proof. By placing all vertices into separate clusters, S is consistent with all of the negative edges, E^-. On the other hand, by placing all vertices into the same cluster, T is consistent with all of the positive edges, E^+. Since the optimum solution value is at most $|E^+| + |E^-|$, one of these solutions agrees achieves at least half of the optimum objective (S if there are more negative edges than positive ones, and T otherwise.) □

While the above proof shows that it is easy to find a 2-approximation to maximizing agreement, it says nothing about minimizing disagreements. Indeed, even it is easy to come up with situations where the minimum number of disagreements is 0, yet the above algorithm will not find it !

Minimizing Disagreements

Since it is easy to find approximately optimal solutions that maximize agreement, we instead focus on minimizing disagreements. We begin with the full information version, where for every pair of nodes we are told whether they should be in the same cluster, or in a different cluster. Equivalently, every pair of nodes (v_i, v_j) is either in E^+ or E^-, and the graph G is a complete graph.

Given such an instance, how do we find a good clustering? It turns out that one of the simplest algorithms is also one of the best. The RandomPivot algorithm proceeds by selecting one vertex, called the pivot, uniformly at random. It then adds it, and all of the similar points to a cluster, and recurses on the leftovers. We show it formally in Algorithm 7.1.

Algorithm 7.1 RandomPivot(V, E^+, E^-)

```
If $G$ is empty, return
Select a random pivot $v_i \in V$
Set $C = \{v_i\}$, $V' = \emptyset$
for all $v_j \in V$ do
  if $(v_i, v_j) \in E^+$ then
    Add $v_j$ to $C$
  else
    Add $v_j$ to $R$
Let $G' = (R, E^+[R], E^-[R])$
return $C$, RandomPivot$(G')$
```

Observe that since every pair of vertices is either in E^+ or E^-, each node is either similar to the pivot (in which case it is added to the cluster), or dissimilar, in which case it is added to the remaining set R, and is then handled in the recursive call. For ease of notation, we denote by $E^+[R]$ and $E^-[R]$ the set of edges in E^+ and E^- restricted to having both of their endpoints in R.

It is perhaps surprising that such a simple algorithm returns a nearly optimal solution. Specifically:

Theorem 7.1. *When the objective is to minimize the number of disagreements, the solution returned by the* RandomPivot *algorithm is a 3-approximation in expectation.*

How do we go about proving such a statement? Consider the situation with three items, A, B and C and constraints that A and B and A and C should be in the same cluster, and B and C should be in different clusters. Looking at it in a graph sense, we have a triangle where two of the edges are positive, and one is negative, let's call such a triangle, a *bad* triangle. Any solution, including the optimum, must make at least one mistake when clustering A, B and C. On this simple example, the algorithm RandomPivot will also make one mistake, no matter which vertex is chosen first.

Can we use the number of bad triangles as a lower bound on the number of mistakes any algorithm must make? Not quite: consider the example in Figure 7.1, here both ABD and CBD are bad triangles, but since they share the BD edge, a solution can make one mistake overall, but break both triangles (for example, putting all points in one cluster). Thus simply counting the number of bad triangles is not enough, we must ensure that each edge is counted at most once. One way to do this is to assign a weight to each bad triangle, and make sure that for any edge, the total weight of triangles this edge is incident on is at most 1. Consider again the example in Figure 7.1. Here we can assign a weight of $1/2$ to each of the three bad triangles, ABD, BCD, and ACD. The total weight of all bad triangles is $3/2$, thus any solution must have a cost of $\lceil 3/2 \rceil = 2$.

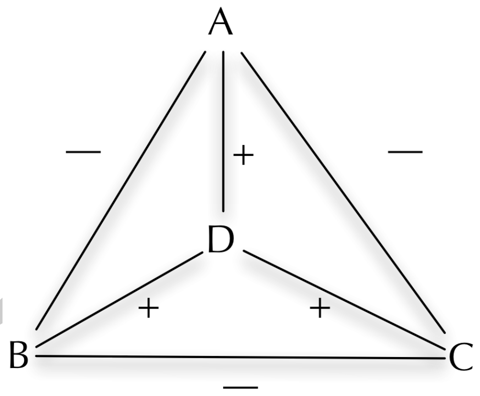


Figure 7.1: Bad triangles

Let T be the set of bad triangles, and w_t be the weights such that $\sum_{t \in T: e \in t} w_t \leq 1$ for all edges; such an assignment is called a packing. Then any solution, including the optimum must have a cost at least $\sum_{t \in T} w_t$.

To analyze the performance of RandomPivot, let Z_t be the event that one of the vertices of t is chosen as a pivot, when all three vertices are part of the same recursive call. We denote by $p_t = Pr[Z_t]$ the probability of this happening. Then the expected number of mistakes made by the algorithm is exactly $\sum_t p_t$ (we make a mistake every time we pivot on one of the vertices of the bad triangle). What we will show is that setting $w_t = p_t/3$ satisfies the conditions of the packing. Thus the cost of any solution is at least one third of the cost of the algorithm's solution.

To show that the weights $p_t/3$ satisfy the packing constraints, we must show that for any edge e, $\sum_{t \in T: e \in t} p_t/3 \leq 1$. Consider an edge e whose constraint is violated. It must be that it is part of some bad triangle, and the vertex opposite this edge was chosen as a pivot. Let $e_T = \{t_1, t_2, \ldots, t_k\}$ be the set of bad triangles that e is present in. For every such bad triangle t, the probability that the vertex opposite is chosen is 1/3, since every vertex is chosen equally likely. Moreover, if the edge is broken due to pivoting on a vertex in some triangle t_i, it cannot be broken again by pivoting on a vertex in another triangle t_j. Formally, the events of an edge e being broken by pivoting on an opposing vertex are independent. Therefore, $\sum_{t\ e_T} p_t/3 \leq 1$, and $w_t = p_t/3$ represents a feasible packing.

Weighted Instances

So far we have assumed that the world is black and white, that is every pair of points is either similar or dissimilar. However, often times, the world is gray, and a pair of points tends to be similar or dissimilar. Formally, for every pair of nodes v_i, v_j we assign a similarity score w_{ij}^+ and a dissimilarity score of w_{ij}^-. The goal is now to maximize the weighted agreement or minimize the weighted disagreement. The two formulations are equivalent if we insist that for every pair of nodes $w_{ij}^+ + w_{ij}^- = 1$, in other words, any two nodes that are p-similar are $1 - p$-dissimilar.

To solve the weighted version, a simple approach is to look at the majority vote for each pair of points. If $w_{ij}^+ > w_{ij}^-$ add this pair to E^+, otherwise, add it to E^-. It is remarkable that in this situation the RandomPivot algorithm is in expectation a 5-approximation algorithm to minimizing the weighted disagreement!

Incomplete Information

The RandomPivot algorithm crucially relies on the fact that all of the points are either similar or dissimilar. However, what if we don't hold strong opinions on the similarity of a pair of items. One can of course simply assign a random direction to this edge (this is equivalent to putting a weight of 1/2 in the weighted instance. This would transform the problem into a full information instance. However, from the point of view of approximation, such a transformation actually changes the value of the objective function, therefore there is no guarantee that the final solution is reasonable with respect to the original set of constraints.

Instead, such a scenario calls for a different approach. Consider again the graph view of the correlation clustering problem. We are given a set of nodes, and some edges are positive, denoting that the pair of nodes should be in the same cluster, while others are negative. Our goal is to cut the graph into a number of pieces to minimize disagreements.

We can transform the problem of correlation clustering directly into the a cut problem on graphs. Specifically, we will make use of the *multiway cut* formulation. In the MultiCut problem we are given a (potentially weighted) graph, and a set of pairs of nodes, called terminals, (s_i, t_i). The goal is to remove as few edges as possible so that for all paris, s_i and t_i lie in different connected components.

This problem feels similar to correlation clustering. In some sense, we are replacing the notion of dissimilar edges with pairs of nodes that must be separated. The difference of course, comes from the fact that correlation clustering is more lax: we are allowed to put dissimilar nodes into the same cluster, it simply increases our objective. On the other hand the multi cut problem is strict, any solution *must* separate all pairs of terminals. It turns out this distinction does not change the complexity of the problem, and we can reduce every correlation clustering instance to an instance of weighted multicut with the optimal solutions having the same value.

Specifically, for each negative edge (v_i, v_j) with an auxiliary vertex v_{ij} and replace the edge with an edge (v_i, v_{ij}) of the same weight. Moreover, we add (v_j, v_{ij}) to the set of terminals that must be separated. We then consider the multi cut problem on the graph G' consisting of all positive edges and all auxiliary edges. Note that if there were no positive edges connecting v_i to v_j then in G' the terminals v_j and v_{ij} are not connected. On the other hand, if there were positive edges, we can separate the terminals by either cutting the auxiliary edge, or cutting one of the positive edges. The former may be cheaper, but only disconnects v_{ij}, whereas the latter may simultaneously disconnect multiple terminals.

One can show that this construction is tight, that is for a correlation clustering with objective W there exists a multi cut in the new graph G' of weight W and vice versa. The multicut problem is NP-complete in general (only the version with one or two pairs of terminals is solvable in polynomial time), however, there are efficient $\log n$-approximation algorithms for this problem. Moreover, for more restricted versions of the problem (for example when the underlying graph G is a tree, or is planar) there are more efficient and better approximation algorithms.

8 Spectral Clustering

> *One cannot alter a condition with the same mind set that created it in the first place.*
>
> *Albert Einstein*

> *Ceci n'est pas une pipe*
>
> *René Magritte*

We have now seen two ways to cluster data presented to us in the form of relationships (positive or negative). If we merely have information about linkage, we can use a graph clustering framework as described in Chapter 6 and if we have both positive and negative linkage information we could perform correlation clustering as described in Chapter 7.

A third approach leads us to spectral clustering, which is one of the most popular approaches to clustering. Just like before, it takes as input a set of objects V and a *similarity* function $w\colon V \times V \to \mathbb{R}$, and groups them so that similar objects are clustered together.

The interesting thing about spectral clustering though is that *it's not actually a clustering technique*! Rather, it's a way to change the space the data lies in from some arbitrary space to a Euclidean space while approximately preserving the similarities given by w. Once this embedding is done, any Euclidean clustering algorithm (such as k-means) can be used to cluster the points.

8.1 Formal Objective

To motivate how spectral clustering works, consider the simpler situation when we wish to divide a set of objects into two clusters. For convenience, we will assume that the set $V = [n] = \{1, \ldots, n\}$ and we use $w_{ij} = w(i, j)$ to represent the similarity between two objects.

We encode the desired clustering as a function $f\colon [n] \to \{-1, 1\}$, so that the two clusters correspond to points that f maps to 1 and to -1. Again, we will use the shorthand $f_i = f(i)$. Abusing notation slightly, we will also interpret f as a vector $(f_1, f_2, \ldots, f_n)$.

Our goal is to place points that are similar in the same cluster. Note that any two points i, j are in the same cluster if and only if $f_i - f_j = 0$, and belong to different clusters if $f_i - f_j \in \{-2, +2\}$, or equivalently if $(f_i - f_j)^2 = 4$. For every pair of points i, j that is not clustered together we pay a cost w_{ij}. Since $(f_i - f_j)^2$ is either 0 or 4, we can write the total cost incurred by any clustering as

$$c(f) = \sum_{i,j} \frac{w_{ij}}{4} (f_i - f_j)^2.$$

Minimizing $c(f)$ is easy. If we set f to be the constant vector $(1, 1, \ldots, 1)$ then $c(f) = 0$. However, this is not a very interesting solution. It corresponds to placing all of the objects in a single cluster. This solution captures the literal interpretation of the cost function but not its spirit [1].

We must add additional constraints to the problem in order to obtain more interesting solutions. One way to do it is by requiring both clusters to be of exactly the same size. Since all points are assigned either $+1$ or -1 this can be encoded by requiring that $\sum_i f_i = 0$.

Our clustering problem now looks like this:

$$\begin{aligned} \min \quad & c(f) = {}^1\!/_4 \sum_{i,j} w_{ij}(f_i - f_j)^2 \\ \text{Subject to:} \quad & \sum_i f_i = 0 \\ \forall i \quad & f_i \in \{-1, +1\} \end{aligned}$$

A relaxed optimization

This problem is NP-hard. We can make the problem easier by allowing f to vary smoothly between $[-1, 1]$ instead of requiring it to be exactly $+1$ or -1. A solution to this *relaxed* problem is a mapping of the points to locations on the interval $[-1, +1]$ in a way that minimizes the cost. In essence, we have constructed an *embedding* of the data onto the interval $[-1, +1]$ in such a way that any clustering algorithm, such as k-means or k-median, can now be used to find the desired 2-partition.

How do we obtain such an embedding ? The simplest approach would be to replace the constraint

$$f_i \in \{-1, +1\}$$

[1] This is a common pitfall when formalizing a notion of clustering. Modeling an inherently fuzzy problem with precise mathematics often creates a disconnect between what you want and what you get.

by

$$-1 \leq f_i \leq 1$$

But this doesn't work ! Once again, we can get a trivial solution by setting $f_i \leftarrow 0$.

In order to prevent this from happening, we must insist that the procedure make full use of the available range for f. One way to ensure this is to place a lower bound on the sum of the magnitude of the coordinates of f. When we required $f_i \in \{-1, +1\}$, then we knew that $\sum_i |f_i| = \sum_i f_i^2 = n$. We can use the same constraint in this relaxed setting.

We can now allow f_i to range freely between $-\infty$ and $+\infty$, without worrying about the all zero solution, since it is no longer feasible. The constraint $\sum_i f_i^2 = n$ ensures that the individual components cannot get too big or too small. Rewriting, and dropping the factor of 1/4 since it doesn't affect the optimization, we get:

$$\begin{aligned} \text{min} \quad & c(f) = \sum_{i,j} w_{ij}(f_i - f_j)^2 \\ \text{Subject to:} \quad & \sum_i f_i = 0 \\ & \sum_i f_i^2 = n \end{aligned}$$

Matrix Form

Finding the solution to the above problem looks like a daunting challenge. However, it is very closely related to a classical problem in linear algebra: finding the eigenvectors and eigenvalues for a matrix.

To explore the connection, we must transform the above optimization problem into matrix form. First consider the objective $c(f)$. We can expand the quadratic form and rewrite it as $\sum_{i,j} \ell_{i,j} f_i f_j$ where the coefficients $\ell_{i,j}$ are:

$$\ell_{ij}(n) = \begin{cases} -w_{ij} & \text{if } i \neq j \\ \sum_k w_{ik} & \text{if } i = j \end{cases}$$

If we let L be the matrix with value ℓ_{ij} in the position (i, j), then we can rewrite the objective as:

$$c(f) = \sum_{i,j} w_{ij}(f_i - f_j)^2 = \sum_{ij} \ell_{ij} f_i f_j = f^\top L f.$$

To rewrite the constraints in matrix form, note that if f is a vector then $\sum_i f_i^2 = \|f\|^2 = f^\top f$ is just the squared length of the vector. This leads the following optimization:

$$\begin{array}{rl} \min & f^\top L f \\ \text{Subject to:} & \sum_i f_i = 0 \\ & f^\top f = n \end{array}$$

We are almost done. Recall that the constraint $f^\top f = n$ came from a desire to put a lower bound on the magnitude of the coordinates of f. However, the exact value is not important — if we know that $f^\top f = c$ for some constant c, then we can scale all entries by $\sqrt{n/c}$ to satisfy this constraint. In this case the value of the solution will also increase by n/c. Thus we are looking to minimize the ratio of the two constraints, which leads to a simpler formulation:

$$\min \frac{f^\top L f}{f^\top f} \text{ subject to } \sum_i f_i = 0.$$

Without the balance constraint, $\sum_i f_i = 0$, the solution is the *eigenvector corresponding to the smallest eigenvalue of L*. It is easy to verify that this eigenvector is the all 1s vector $v_1 = (1, 1, \ldots, 1)$, which corresponds to the unbalanced solution we had discarded earlier !

The constraint $\sum f_i = 0$ can be rewritten as $f^\top \mathbf{1} = 0$ which in turn can be written as $f^\top v_1 = 0$. Now the balance constraint can be interpreted geometrically, as we are looking for a vector f that is perpendicular to the all ones vector, or equivalently to the first eigenvector. The resulting balanced optimization is then:

$$\min_{f^\top v_1 = 0} \frac{f^\top L f}{f^\top f}$$

The solution to this is the eigenvector corresponding to the *second smallest* eigenvector of L.

Constructing A Solution

Now that we have a vector of (relaxed) assignments f, we need to recover a 2-clustering. The easiest approach is to replace f_i by $\text{sign}(f_i)$ putting those points with positive f_i into one cluster, and negative f_i to another. This can also be viewed as fixing a boundary at 0 and dividing the points into clusters on either side of the boundary.

But there is nothing particularly special about a boundary at 0. In fact, we could imagine replacing each object by the 1-dimensional point (f_i), and then clustering the points optimally into two clusters by moving from left to right and trying all possible splits. In effect, we have embedded the objects in a (1-dimensional) *feature space* and are clustering the resulting points. In fact, if we retain the all 1s eigenvector, then we've performed a mapping into a *two*-dimensional coordinate system where object i is represented by $(1, f_i)$.

There is no fundamental reason to stop at the second eigenvector. In general, let us fix a parameter d. We compute the matrix L as before, and assemble the eigenvectors $V = (v_1, \ldots, v_d)$ corresponding to the eigenvalues $\lambda_1 = 0 \le \lambda_2 \le \lambda_3 \cdots \le \lambda_d$. Recall that all of the eigenvectors are orthogonal to each other, so this corresponds to an embedding of f into d-dimensional space. Object i maps to the i^{th} row of V (i.e the vector of i^{th} coordinates of each eigenvector). Given this embedding we can cluster these vectors using any clustering algorithm, such as 2-means.

Beyond 2-clustering

Note that the embedding of points described above is oblivious to the number of clusters we are looking for. Whether we are looking for two, three, or many more clusters the embedding is the same, and it is the final clustering algorithm, working on points in Euclidean space that determines the actual partition into k sets.

This is the key insight of spectral "clustering". Given an arbitrary similarity matrix between points, we find an embedding of these points into d-dimensional Euclidean space that is in some sense the best possible such embedding. Then we can cluster these points using any of the myriad of clustering algorithms we already discussed. Spectral clustering is not a clustering algorithm at all: it's a feature transformation of the input!

8.2 *Graph Laplacians

The matrix L is known as the *Laplacian* of the graph, and has rich connections to graph theory and topology. The transformation of a graph to a collection of points in a Euclidean space via the Laplacian is a powerful tool for going from pairwise similarities between items (encoded in a combinatorial structure of a graph) to a geometry of the space.

Let us start with a simple example. Consider a graph consisting of an n-vertex cycle. If we construct the Laplacian matrix L for this graph, then the diagonal entries are all 2, and assuming the rows are ordered as the nodes in the cycle, then the other nonzero entries are two diagonals of -1s above and below the main diagonal.

Suppose we assign a scalar value x_i to each vertex i, and multiply the Laplacian matrix L by this vector $\boldsymbol{x} = (x_1, \dots, x_n)$, we get $\boldsymbol{y} = L\boldsymbol{x}$ where

$$y_i = 2x_i - x_{i+1} - x_{i-1}$$

which can be rewritten as

$$y_i = (x_i - x_{i-1}) - (x_{i+1} - x_i)$$

This is essentially the second derivative of x viewed as a function over $[1 \dots n]$. In other words, the Laplacian matrix acts as a discrete version of the second derivative, $\Delta = \frac{d^2}{dr^2}$. Imagine now that we let $n \to \infty$. In the limit, the Laplacian matrix becomes the second derivative operator on the line.

On the other hand, if we the graph was a grid instead of a line, a similar calculation shows that in the limit, the Laplacian matrix becomes the Laplacian operator $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. Notice what happened here: when the graph was a line, the limiting behavior of the Laplacian matrix was that of the second derivative on the line. When the graph was a grid, the limiting behavior was that of the second derivative in two dimensions.

In general, suppose that we have a manifold M, and let V be a sample of points form M, with points that are reasonably close connected by edges, and weighted accordingly. Then the Laplacian matrix of this graph represents the geometry of the manifold (formally it tends in the limit to the Laplace-Beltrami operator associated with M).

Distances

The connection between a graph Laplacian and the Laplace-Beltrami operator explains the shape we're expecting to see when we analyze the Laplacian of a graph. But the spectral analysis of the Laplacian gives us an actual embedding of vertices in a vector space. What do the resulting distances represent?

The Laplace-Beltrami operator comes about in the analysis of heat flow on a surface. Roughly speaking, the rate of heat decay at a point over time is controlled by the shape of the manifold near that point. This interpretation has a natural discrete analogy: that of flows and cuts. Parts of the graph that are loosely connected are “far” from the perspective of heat flow, because the lack of connectivity means that it would take heat longer to flow between the two points. And it is this distance that the mapping produced by the Laplacian captures. In particular, if we model the heat flow as a random walk, then the commute time between two vertices corresponds to the Euclidean distance between the corresponding feature vectors produced from a spectral analysis of the Laplacian.

To summarize, the spectral interpretation of similarities is based on two modeling assumptions:

(a) That the similarities represent a graph that can be viewed as capturing the shape of a manifold.

(b) That the “distance” between two vertices can be captured in terms of the time taken for heat to flow from one vertex to the other, rather than some kind of geodesic distance.