Chapter 10

Distance and Network Methods

When a measure becomes a target, it ceases to be a measure.

– Charles Goodhart (Goodhart's Law)

An $n \times d$ data matrix, consisting of n examples/rows each defined by d features/columns, naturally defines a set of n points in a d-dimensional geometric space. Interpreting examples as points in space provides a powerful way to think about them – like the stars in the heavens. Which stars are the closest to our sun, i.e. our nearest neighbors? Galaxies are natural groupings of stars identified by clustering the data. Which stars share the Milky Way with our sun?

There is a close connection between collections of points in space and vertices in networks. Often we build networks from geometric point sets, by connecting close pairs of points by edges. Conversely, we can build point sets from networks, by embedding the vertices in space, so that pairs of connected vertices are located near each other in the embedding.

Several of the important problems on geometric data readily generalize to network data, including nearest neighbor classification and clustering. Thus we treat both topics together in this chapter, to better exploit the synergies between them.

10.1 Measuring Distances

The most basic issue in the geometry of points p and q in d dimensions is how best to measure the distance between them. It might not be obvious that there is any issue here to speak of, since the traditional *Euclidean* metric is *obviously*

S.S. Skiena, *The Data Science Design Manual,*
Texts in Computer Science, https://doi.org/10.1007/978-3-319-55444-0_10

how you measure distances. The Euclidean matrix defines

$$d(p,q) = \sqrt{\sum_{i=1}^{d} |p_i - q_i|^2}$$

But there are other reasonable notions of distance to consider. Indeed, what is a distance metric? How does it differ from an arbitrary scoring function?

10.1.1 Distance Metrics

Distance measures most obviously differ from similarity scores, like the correlation coefficient, in their direction of growth. Distance measures get smaller as items become more similar, while the converse is true of similarity functions.

There are certain useful mathematical properties we assume of any reasonable distance measure. We say a distance measure is a *metric* if it satisfies the following properties:

- *Positivity*: $d(x,y) \geq 0$ for all x and y.
- *Identity*: $d(x,y) = 0$ if and only if $x = y$.
- *Symmetry*: $d(x,y) = d(y,x)$ for all x and y.
- *Triangle inequality*: $d(x,y) \leq d(x,z) + d(z,y)$ for all x, y, and z.

These properties are important for reasoning about data. Indeed, many algorithms work correctly only when the distance function is a metric.

The Euclidean distance is a metric, which is why these conditions seem so natural to us. However, other equally-natural similarity measures are not distance metrics:

- *Correlation coefficient*: Fails positivity because it ranges from -1 to 1. Also fails identity, as the correlation of a sequence with itself is 1.
- *Cosine similarity/dot product*: Similar to correlation coefficient, it fails positivity and identity for the same reason.
- *Travel times in a directed network*: In a world with one-way streets, the distance from x to y is not necessarily the same as the distance from y to x.
- *Cheapest airfare*: This often violates the triangle inequality, because the cheapest way to fly from x to y might well involve taking a detour through z, due to bizarre airline pricing strategies.

By contrast, it is not immediately obvious that certain well-known distance functions are metrics, such as edit distance used in string matching. Instead of making assumptions, prove or disprove each of the four basic properties, to be sure you understand what you are working with.

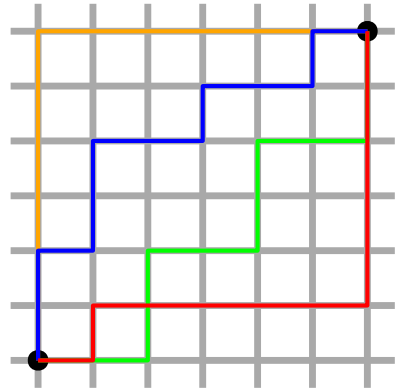


Figure 10.1: Many different paths across a grid have equal Manhattan (L_1) distance.

10.1.2 The L_k Distance Metric

The Euclidean distance is just a special case of a more general family of distance functions, known as the L_k distance metric or norm:

$$d_k(p,q) = \sqrt[k]{\sum_{i=1}^{d} |p_i - q_i|^k} = (\sum_{i=1}^{d} |p_i - q_i|^k)^{1/k}$$

The parameter k provides a way to trade off between the largest and the total dimensional differences. The value for k can be any number between 1 and ∞, with particularly popular values including:

- *Manhattan distance ($k = 1$)*: If we ignore exceptions like Broadway, all streets in Manhattan run east–west and all avenues north–south, thus defining a regular grid. The distance between two locations is then the sum of this north–south difference and the east–west difference, since tall buildings prevent any chance of shortcuts.

  Similarly, the L_1 or *Manhattan* distance is the total sum of the deviations between the dimensions. Everything is linear, so a difference of 1 in each of two dimensions is the same as a difference of 2 in only one dimension. Because we cannot take advantage of diagonal short-cuts, there are typically many possible shortest paths between two points, as shown in Figure 10.1.

- *Euclidean distance ($k = 2$)*: This is the most popular distance metric, offering more weight to the largest dimensional deviation without overwhelming the lesser dimensions.

- *Maximum component ($k = \infty$)*: As the value of k increases, smaller dimensional differences fade into irrelevance. If $a > b$, then $a^k \gg b^k$. Taking the kth root of $a^k + b^k$ approaches a as $b^k/a^k \to 0$.

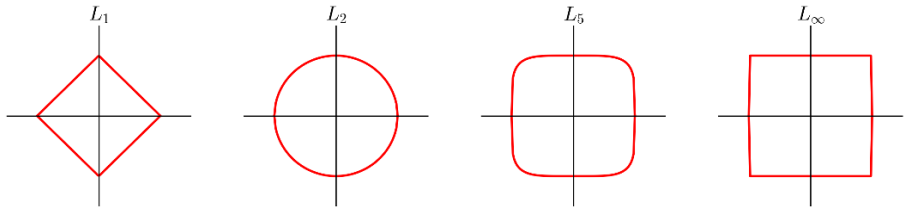


Figure 10.2: The shape of circles defining equal distances changes with k.

Consider the distance of points $p_1 = (2, 0)$ and $p_2 = (2, 1.99)$ from the origin:

- For $k = 1$, the distances are 2 and 3.99, respectively.
- For $k = 2$, they are 2 and 2.82136.
- For $k = 1000$, they are 2 and 2.00001.
- For $k = \infty$, they are 2 and 2.

The L_∞ metric returns the largest single dimensional difference as the distance.

We are comfortable with Euclidean distance because we live in a Euclidean world. We believe in the truth of the Pythagorean theorem, that the sides of a right triangle obey the relationship that $a^2 + b^2 = c^2$. In the world of L_k distances, the Pythagorean theorem would be $a^k + b^k = c^k$.

We are similarly comfortable with the notion that circles are round. Recall that a circle is defined as the collection of points which are at a distance r from an origin point p. Change the definition of distance, and you change the shape of a circle.

The shape of an L_k "circle" governs which points are equal neighbors about a center point p. Figure 10.2 illustrates how the shape evolves with k. Under Manhattan distance ($k = 1$), the circle looks like a diamond. For $k = 2$, it is the round object we are familiar with. For $k = \infty$, this circle stretches out to an axis-oriented box.

There is smooth transition from the diamond to the box as we vary $1 \leq k \leq \infty$. Selecting the value of k is equivalent to choosing which circle best fits our domain model. The distinctions here become particularly important in higher dimensional spaces: do we care about deviations in all dimensions, or primarily the biggest ones?

Take-Home Lesson: Selecting the right value of k can have a significant effect on the meaningfulness of your distance function, particularly in high-dimensional spaces.

Taking the kth root of the sum of kth-power terms is necessary for the resulting "distance" values to satisfy the metric property. However, in many applications we will be only using the distances for comparison: testing whether $d(x,p) \leq d(x,q)$ as opposed to using the values in formulas or isolation.

Because we take the absolute value of each dimensional distance before raising it to the kth power, the summation within the distance function always yields a positive value. The kth root/power function is *monotonic*, meaning that for $x, y, k \geq 0$

$$(x > y) \rightarrow (x^k > y^k).$$

Thus the order of distance comparison is unchanged if we do not take the kth root of the summation. Avoiding the kth root calculation saves time, which can prove non-trivial when many distance computations are performed, as in nearest neighbor search.

10.1.3 Working in Higher Dimensions

I personally have no geometric sense about higher-dimensional spaces, anything where $d > 3$. Usually, the best we can do is to think about higher-dimensional geometries through linear algebra: the equations which govern our understanding of two/three-dimensional geometries readily generalize for arbitrary d, and that is just the way things work.

We can develop some intuition about working with a higher-dimensional data set through *projection* methods, which reduce the dimensionality to levels we can understand. It is often helpful to visualize the two-dimensional projections of the data by ignoring the other $d-2$ dimensions entirely, and instead study dot plots of dimensional pairs. Through dimension reduction methods like principle component analysis (see Section 8.5.2), we can combine highly correlated features to produce a cleaner representation. Of course, some details are lost in the process: whether it is noise or nuance depends upon your interpretation.

It should be clear that as we increase the number of dimensions in our data set, we are implicitly saying that each dimension is a less important part of the whole. In measuring the distance between two points in feature space, understand that large d means that there are more ways for points to be close (or far) from each other: we can imagine them being almost identical along all dimensions but one.

This makes the choice of distance metric most important in high-dimensional data spaces. Of course, we can always stick with L_2 distance, which is a safe and standard choice. But if we want to reward points for being close on many dimensions, we prefer a metric leaning more towards L_1. If instead things are similar when there are no single fields of gross dissimilarity, we perhaps should be interested in something closer to L_∞.

One way to think about this is whether we are more concerned about random added noise to our features, or exceptional events leading to large artifacts. L_1 is undesirable in the former case, because the metric will add up the noise from all dimensions in the distance. But artifacts make L_∞ suspect, because a

substantial error in any single column will come to dominate the entire distance calculation.

Take-Home Lesson: Use your freedom to select the best distance metric. Evaluate how well different functions work to tease out the similarity of items in your data set.

10.1.4 Dimensional Egalitarianism

The L_k distance metrics all implicitly weigh each dimension equally. It doesn't have to be this way. Sometimes we come to a problem with a domain-specific understanding that certain features are more important for similarity than others. We can encode this information using a coefficient c_i to specify a different weight to each dimension:

$$d_k(p,q) = \sqrt[k]{\sum_{i=1}^{d} c_i |p_i - q_i|^k} = (\sum_{i=1}^{d} c_i |p_i - q_i|^k)^{1/k}$$

We can view the traditional L_k distance as a special case of this more general formula, where $c_i = 1$ for $1 \leq i \leq d$. This dimension-weighted distance still satisfies the metric properties.

If you have ground-truth data about the desired distance between certain pairs of points, then you can use linear regression to fit the coefficients c_i to best match your training set. But, generally speaking, dimension-weighted distance is often not a great idea. Unless you have a genuine reason to know that certain dimensions are more important than others, you are simply encoding your biases into the distance formula.

But much more serious biases creep in if you do not normalize your variables before computing distances. Suppose we have a choice of reporting a distance in either meters or kilometers. The contribution of a 30 meter difference in the distance function will either be $30^2 = 900$ or $0.03^{=}0.0009$, literally a *million-fold* difference in weight.

The correct approach is to normalize the values of each dimension by Z-scores *before* computing your distance. Replace each value x_i by its Z-score $z = (x - \mu_i)/\sigma_i$, where μ_i is the mean value of dimension i and σ_i its standard deviation. Now the expected value of x_i is zero for all dimensions, and the spread is tightly controlled if they were normally distributed to start with. More stringent efforts must be taken if a particular dimension is, say, power law distributed. Review Section 4.3 on normalization for relevant techniques, like first hitting it with a logarithm before computing the Z-score.

Take-Home Lesson: The most common use of dimension-weighted distance metrics is as a kludge to mask the fact that you didn't properly normalize your data. Don't fall into this trap. Replace the original values by Z-scores before computing distances, to ensure that all dimensions contribute equally to the result.

10.1.5 Points vs. Vectors

Vectors and points are both defined by arrays of numbers, but they are conceptually different beasts for representing items in feature space. Vectors decouple direction from magnitude, and so can be thought of as defining points on the surface of a unit sphere.

To see why this is important, consider the problem of identifying the nearest documents from word–topic counts. Suppose we have partitioned the vocabulary of English into n different subsets based on topics, so each vocabulary word sits in exactly one of the topics. We can represent each article A as a bag of words, as a point p in n-dimensional space where p_i equals the number of words appearing in article A that come from topic i.

If we want a long article on football to be close to a short article on football, the magnitude of this vector cannot matter, only its direction. Without normalization for length, all the tiny tweet-length documents will bunch up near the origin, instead of clustering semantically in topic space as we desire.

Norms are measures of vector magnitude, essentially distance functions involving only one point, because the second is taken to be the origin. Vectors are essentially normalized points, where we divide the value of each dimension of p by its L_2-norm $L_2(p)$, which is the distance between p and the origin O:

$$L_2(p) = \sqrt{\sum_{i=1}^{n} p_i^2}$$

After such normalization, the length of each vector will be 1, turning it into a point on the unit sphere about the origin.

We have several possible distance metrics to use in comparing pairs of vectors. The first class is defined by the L_k metrics, including Euclidean distance. This works because points on the surface of a sphere are still points in space. But we can perhaps more meaningfully consider the distance between two vectors in terms of the angle defined between them. We have seen that the *cosine similarity* between two points p and q is their dot product divided by their L_2-norms:

$$\cos(p, q) = \frac{p \cdot q}{||p||\ ||q||}$$

For previously normalized vectors, these norms equal 1, so all that matters is the dot product.

The cosine function here is a similarity function, *not* a distance measure, because larger values mean higher similarity. Defining a *cosine distance* as $1 - |\cos(p, q)|$ does yield distance measure that satisfies three of the metric properties, all but the triangle inequality. A true distance metric follows from

angular distance, where

$$d(p,q) = 1 - \frac{\arccos(\cos(p,q))}{\pi}$$

Here arccos() is the inverse cosine function $\cos^{-1}()$, and π is the largest angle range in radians.

10.1.6 Distances between Probability Distributions

Recall the Kolmogorov-Smirnov test (Section 5.3.3), which enabled us to determine whether two sets of samples were likely drawn from the same underlying probability distribution.

This suggests that we often need a way to compare a pair of distributions and determine a measure of similarity or distance between them. A typical application comes in measuring how closely one distribution approximates another, providing a way to identify the best of a set of possible models.

The distance measures that have been described for points could, in principle, be applied to measure the similarity of two probability distributions P and Q over a given discrete variable range R.

Suppose that R can take on any of exactly d possible values, say $R = \{r_1, \ldots, r_d\}$. Let p_i (q_i) denote the probability that $X = r_i$ under distribution P (Q). Since P and Q are both probability distributions, we know that

$$\sum_{i=1}^{d} p_i = \sum_{i=1}^{d} q_i = 1$$

The spectrum of p_i and q_i values for $1 \leq i \leq d$ can be thought of as d-dimensional points representing P and Q, whose distance could be computed using the Euclidean metric.

Still, there are more specialized measures, which do a better job of assessing the similarity of probability distributions. They are based on the information-theoretic notion of *entropy*, which defines a measure of uncertainty for the value of a sample drawn from the distribution. This makes the concept mildly analogous to variance.

The entropy $H(P)$ of a probability distribution P is given by

$$H(P) = \sum_{i=1}^{d} p_i \log_2(\frac{1}{p_i}) = -\sum_{i=1}^{d} p_i \log_2(p_i).$$

Like distance, entropy is always a non-negative quantity. The two sums above differ only in how they achieve it. Because p_i is a probability, it is generally less than 1, and hence $\log(p_i)$ is generally negative. Thus either taking the reciprocal of the probabilities before taking the log or negating each term suffices to make $H(P) \geq 0$ for all P.

Entropy is a measure of uncertainty. Consider the distribution where $p_1 = 1$ and $p_i = 0$, for $2 \leq i \leq d$. This is like tossing a totally loaded die, so

despite having d sides there is no uncertainty about the outcome. Sure enough, $H(P) = 0$, because either p_i or $\log_2(1)$ zeros out every term in the summation. Now consider the distribution where $q_i = 1/d$ for $1 \leq i \leq d$. This represents fair dice roll, the maximally uncertain distribution where $H(Q) = \log_2(d)$ bits.

The flip side of uncertainty is information. The entropy $H(P)$ corresponds to how much information you learn after a sample from P is revealed. You learn nothing when someone tells you something you already know.

The standard distance measures on probability distributions are based on entropy and information theory. The *Kullback-Leibler* (KL) divergence measures the uncertainty gained or information lost when replacing distribution P with Q. Specifically,

$$KL(P||Q) = \sum_{i=1}^{d} p_i \log_2 \frac{p_i}{q_i}$$

Suppose $P = Q$. Then nothing should be gained or lost, and $KL(P, P) = 0$ because $\lg(1) = 0$. But the worse a replacement Q is for P, the larger $KL(P||Q)$ gets, blowing up to ∞ when $p_i > q_i = 0$.

The KL divergence resembles a distance measure, but is not a metric, because it is not symmetric ($KL(P||Q) \neq KL(Q||P)$) and does not satisfies the triangle inequality. However, it forms the basis of the *Jensen-Shannon divergence* $JS(P, Q)$:

$$JS(P, Q) = \frac{1}{2} KL(P||M) + \frac{1}{2} KL(Q||M)$$

where the distribution M is the average of P and Q, i.e. $m_i = (p_i + q_i)/2$.

$JS(P, Q)$ is clearly symmetric while preserving the other properties of KL divergence. Further $\sqrt{JS(P, Q)}$ magically satisfies the triangle inequality, turning it into a true metric. This is the right function to use for measuring the distance between probability distributions.

10.2 Nearest Neighbor Classification

Distance functions grant us the ability to identify which points are closest to a given target. This provides great power, and is the engine behind *nearest neighbor classification*. Given a set of labeled training examples, we seek the training example which is most similar to an unlabeled point p, and then take the class label for p from its nearest labeled neighbor.

The idea here is simple. We use the nearest labeled neighbor to a given query point q as its representative. If we are dealing with a classification problem, we will assign q the same label as it nearest neighbor(s). If we are dealing with a regression problem, assign q the mean/median value of its nearest neighbor(s). These forecasts are readily defensible assuming (1) the feature space coherently captures the properties of the elements in question, and (2) the distance function meaningfully recognizes similar rows/points when they are encountered.

The Bible exhorts us to love thy neighbor. There are three big advantages to nearest neighbor methods for classification:

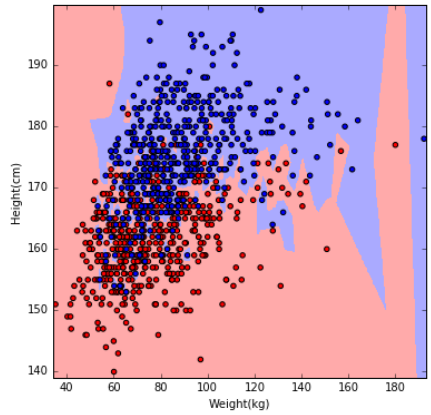


Figure 10.3: The decision boundary of nearest-neighbor classifiers can be non-linear.

- *Simplicity*: Nearest neighbor methods are not rocket science; there is no math here more intimidating than a distance metric. This is important, because it means we can know exactly what is going on and avoid being the victim of bugs or misconceptions.
- *Interpretability*: Studying the nearest-neighbors of a given query point q explains exactly why the classifier made the decision it did. If you disagree with this outcome, you can systematically debug things. Were the neighboring points incorrectly labeled? Did your distance function fail to pick out the items which were the logical peer group for q?
- *Non-linearity*: Nearest neighbor classifiers have decision boundaries which are piecewise-linear, but can crinkle arbitrarily following the training example herd, as shown in Figure 10.3. From calculus we know that piecewise-linear functions approach smooth curves once the pieces get small enough. Thus nearest neighbor classifiers enable us to realize very complicated decision boundaries, indeed surfaces so complex that they have no concise representation.

There are several aspects to building effective nearest neighbor classifiers, including technical issues related to robustness and efficiency. But foremost is learning to appreciate the power of analogy. We discuss these issues in the sections below.

10.2.1 Seeking Good Analogies

Certain intellectual disciplines rest on the power of analogies. Lawyers don't reason from laws directly as much as they rely on precedents: the results of

previously decided cases by respected jurists. The right decision for the current case (I win or I lose) is a function of which prior cases can be demonstrated to be most fundamentally similar to the matter at hand.

Similarly, much of medical practice rests on experience. The old country doctor thinks back to her previous patients to recall cases with similar symptoms to yours that managed to survive, and then gives you the same stuff she gave them. My current physician (Dr. Learner) is now in his eighties, but I trust him ahead of all those young whipper-snappers relying only on the latest stuff taught in medical school.

Getting the greatest benefits from nearest neighbor methods involves learning to respect analogical reasoning. What is the right way to predict the price of a house? We can describe each property in terms of features like the area of the lot and the number of bedrooms, and assign each a dollar weight to be added together via linear regression. Or we can look for "comps," seeking comparable properties in similar neighborhoods, and forecast a price similar to what we see. The second approach is analogical reasoning.

I encourage you to get hold of a data set where you have domain knowledge and interest, and do some experiments with finding nearest neighbors. One resource that always inspires me is `http://www.baseball-reference.com`, which reports the ten nearest neighbors for each player, based on their statistics to date. I find these analogies amazingly evocative: the identified players often fill similar roles and styles which should not be explicitly captured by the statistics. Yet somehow they are.

Try to do this with another domain you care about: books, movies, music, or whatever. Come to feel the power of nearest neighbor methods, and analogies.

> *Take-Home Lesson*: Identifying the ten nearest neighbors to points you know about provides an excellent way to understand the strengths and limitations of a given data set. Visualizing such analogies should be your first step in dealing with any high-dimensional data set.

10.2.2 k-Nearest Neighbors

To classify a given query point q, nearest neighbor methods return the label of q', the closest labeled point to q. This is a reasonable hypothesis, assuming that similarity in feature space implies similarity in label space. However, this classification is based on exactly one training example, which should give us pause.

More robust classification or interpolation follows from voting over multiple close neighbors. Suppose we find the k points closest to our query, where k is typically some value ranging from 3 to 50 depending upon the size of n. The arrangement of the labeled points coupled with the choice of k carves the feature space into regions, with all the points in a particular given region assigned the same label.

Consider Figure 10.4, which attempts to build a gender classifier from data

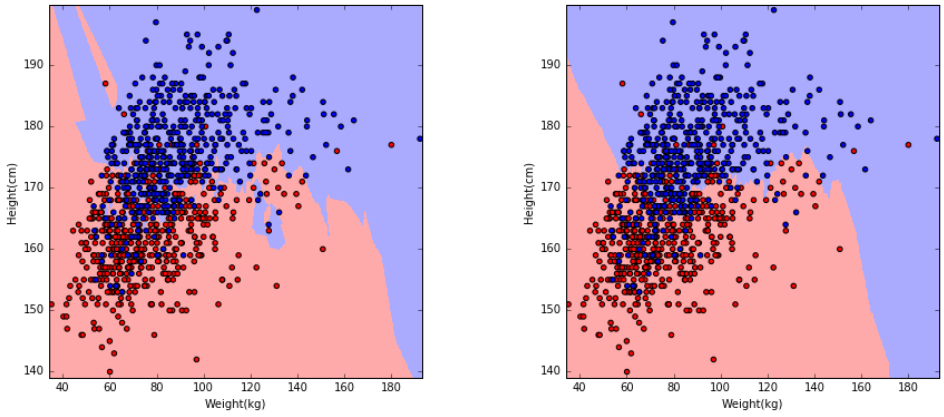


Figure 10.4: The effect of k on the decision boundary for gender classification using k-NN. Compare $k = 3$ (left) and $k = 10$ (right) with $k = 1$ in Figure 10.3.

on height and weight. Generally speaking, women are shorter and lighter than men, but there are many exceptions, particularly near the decision boundary. As shown in Figure 10.4, increasing k tends to produce larger regions with smoother boundaries, representing more robust decisions. However, the larger we make k, the more generic our decisions are. Choosing $k = n$ is simply another name for the majority classifier, where we assign each point the most common label regardless of its individual features.

The right way to set k is to assign a fraction of labeled training examples as an evaluation set, and then experiment with different values of the parameter k to see where the best performance is achieved. These evaluation values can then be thrown back into the training/target set, once k has been selected.

For a binary classification problem, we want k to be an odd number, so the decision never comes out to be a tie. Generally speaking, the difference between the number of positive and negative votes can be interpreted as a measure of our confidence in the decision.

There are potential asymmetries concerning geometric nearest neighbors. Every point has a nearest neighbor, but for outlier points these nearest neighbors may not be particular close. These outlier points in fact can have an outsized role in classification, defining the nearest neighbor to a huge volume of feature space. However, if you picked your training examples properly this should be largely uninhabited territory, a region in feature space where points rarely occur.

The idea of nearest neighbor classification can be generalized to function interpolation, by averaging the values of the k nearest points. This is presumably done by real-estate websites like `www.zillow.com`, to predict housing prices from nearest neighbors. Such averaging schemes can be generalized by non-uniform

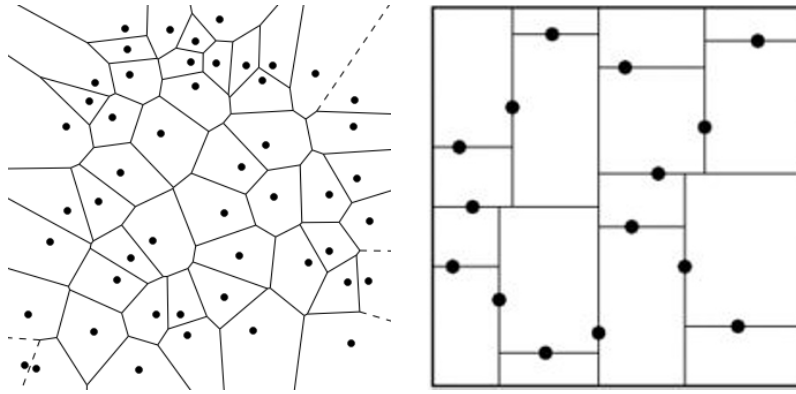


Figure 10.5: Data structures for nearest neighbor search include Voronoi diagrams (left) and *kd*-trees (right).

weights, valuing points differently according to distance rank or magnitude. Similar ideas work for all classification methods.

10.2.3 Finding Nearest Neighbors

Perhaps the biggest limitation of nearest neighbor classification methods is their runtime cost. Comparing a query point q in d dimensions against n such training points is most obviously done by performing n explicit distance comparisons, at a cost of $O(nd)$. With thousands or even millions of training points available, this search can introduce a notable lag into any classification system.

One approach to speeding up the search involves the use of geometric data structures. Popular choices include:

- *Voronoi diagrams*: For a set of target points, we would like to partition the space around them into cells such that each cell contains exactly one target point. Further, we want each cell's target point to be the nearest target neighbor for all locations in the cell. Such a partition is called a *Voronoi diagram*, and is illustrated in Figure 10.5 (left).

  The boundaries of Voronoi diagrams are defined by the perpendicular bisectors between pairs of points (a, b). Each bisector cuts the space in half: one half containing a and the other containing b, such that all points on a's half are closer to a than b, and visa versa.

  Voronoi diagrams are a wonderful tool for thinking about data, and have many nice properties. Efficient algorithms for building them and searching them exist, particularly in two dimensions. However, these procedures rapidly become more complex as the dimensionality increases, making them generally impractical beyond two or three dimensions.

- *Grid indexes*: We can carve up space into d-dimensional boxes, by dividing the range of each dimension into r intervals or buckets. For example,

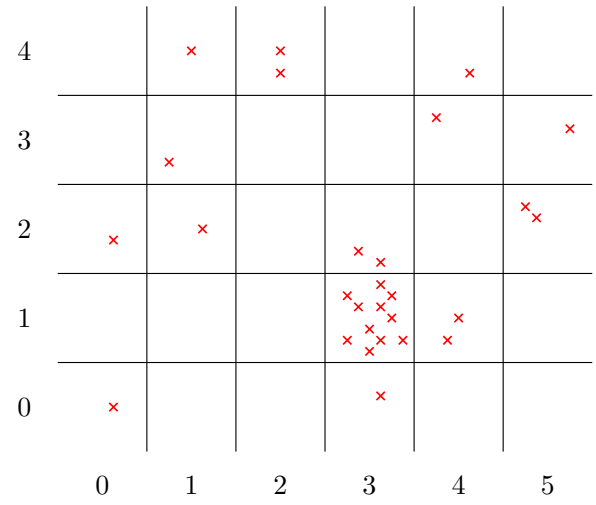


Figure 10.6: A grid index data structure provides fast access to nearest neighbors when the points are uniformly distributed, but can be inefficient when points in certain regions are densely clustered.

consider a two-dimensional space where each axis was a probability, thus ranging from 0 to 1. This range can be divided into r equal-sized intervals, such that the ith interval ranges between $[(i-1)/r, i/r]$.

These intervals define a regular grid over the space, so we can associate each of the training points with the grid cell where it belongs. Search now becomes the problem of identifying the right grid cell for point q through array lookup or binary search, and then comparing q against all the points in this cell to identify the nearest neighbor.

Such grid indexes can be effective, but there are potential problems. First, the training points might not be uniformly distributed, and many cells might be empties, as in Figure 10.6. Establishing a non-uniform grid might lead to a more balanced arrangement, but makes it harder to quickly find the cell containing q. But there is also no guarantee that the nearest neighbor of q actually lives within the same cell as q, particularly if q lies very close to the cell's boundary. This means we must search neighboring cells as well, to ensure we find the absolute nearest neighbor.

- *Kd-trees*: There are a large class of tree-based data structures which partition space using a hierarchy of divisions that facilitates search. Starting from an arbitrary dimension as the root, each node in the *kd*-tree defines median line/plane that splits the points equally according to that dimension. The construction recurs on each side using a different dimension, and so on until the region defined by a node contains just one training point.

This construction hierarchy is ideally suited to support search. Starting at the root, we test whether the query point q is to the left or right of the median line/plane. This identifies which side q lies on, and hence which side of the tree to recur on. The search time is $\log n$, since we split the point set in half with each step down the tree.

There are a variety of such space-partition search tree structures available, with one or more likely implemented in your favorite programming language's function library. Some offer faster search times on problems like nearest neighbor, with perhaps a trade-off of accuracy for speed.

Although these techniques can indeed speed nearest neighbor search in modest numbers of dimensions (say $2 \leq d \leq 10$), they get less effective as the dimensionality increases. The reason is that the number of ways that two points can be close to each other increases rapidly with the dimensionality, making it harder to cut away regions which have no chance of containing the nearest neighbor to q. Deterministic nearest neighbor search eventually reduces to linear search, for high-enough dimensionality data.

10.2.4 Locality Sensitive Hashing

To achieve faster running times, we must abandon the idea of finding the exact nearest neighbor, and settle for a good guess. We want to batch up nearby points into buckets by similarity, and quickly find the most appropriate bucket B for our query point q. By only computing the distance between q and the points in the bucket, we save search time when $|B| \ll n$.

This was the basic idea behind the grid index, described in the previous section, but the search structures become unwieldy and unbalanced in practice. A better approach is based on hashing.

Locality sensitive hashing (LSH) is defined by a hash function $h(p)$ that takes a point or vector as input and produces a number or code as output such that it is likely that $h(a) = h(b)$ if a and b are close to each other, and $h(a) \neq h(b)$ if they are far apart.

Such locality sensitive hash functions readily serve the same role as the grid index, without the fuss. We can simply maintain a table of points bucketed by this one-dimensional hash value, and then look up potential matches for query point q by searching for $h(q)$.

How can we build such locality sensitive hash functions? The idea is easiest to understand at first when restricting to vectors instead of points. Recall that sets of d-dimensional vectors can be thought of as points on the surface of a sphere, meaning a circle when $d = 2$.

Let us consider an arbitrary line l_1 through the origin of this circle, which cuts the circle in half, as in Figure 10.7. Indeed, we can randomly select l_1 by simply picking a random angle $0 \leq \theta_1 < 2\pi$. This angle defines the slope of a line passing through the origin O, and together θ_1 and O completely specify l_1. If randomly chosen, l_1 should grossly partition the vectors, putting about half of them on the left and the remainder on the right.

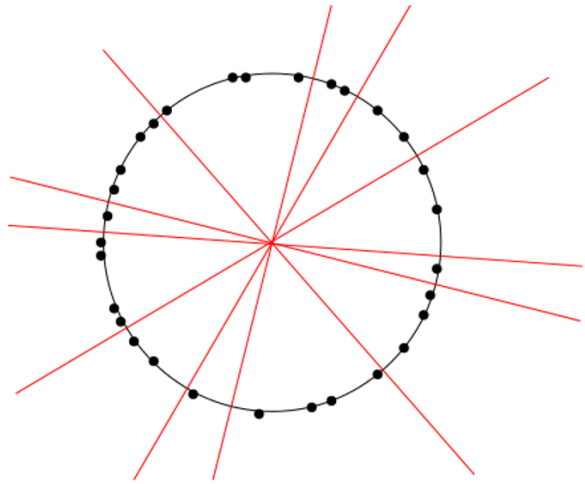


Figure 10.7: Nearby points on the circle generally lie on the same side of random lines through the origin. Locality-sensitive hash codes for each point can be composed as a sequence of sidedness tests (left or right) for any specific sequence of lines.

Now add a second random divider l_2, which should share the same properties. This then partitions all the vectors among four regions, $\{LL, LR, RL, RR\}$, defined by their status relative to these dividers l_1 and l_2.

The nearest neighbor of any vector v should lie in the same region as v, unless we got unlucky and either l_1 or l_2 separated them. But the probability $p(v_1, v_2)$ that both v_1 and v_2 are on the same side of l depends upon the angle between v_1 and v_2. Specifically $p(v_1, v_2) = 1 - \theta(v_1, v_2)/\pi$.

Thus we can compute the exact probability that near neighbors are preserved for n points and m random planes. The pattern of L and R over these m planes defines an m-bit locality-sensitive hash code $h(v)$ for any vector v. As we move beyond the two planes of our example to longer codes, the expected number of points in each bucket drops to $n/2^m$, albeit with an increased risk that one of the m planes separates a vector from its true nearest neighbor.

Note that this approach can easily be generalized beyond two dimensions. Let the hyperplane be defined by its normal vector r, which is perpendicular in direction to the plane. The sign of $s = v \cdot r$ determines which side a query vector v lies on. Recall that the dot product of two orthogonal vectors is 0, so $s = 0$ if v lies exactly on the separating plane. Further, s is positive if v is above this plane, and negative if v is below it. Thus the ith hyperplane contributes exactly one bit to the hash code, where $h_i(q) = 0$ iff $v \cdot r_i \leq 0$.

Such functions can be generalized beyond vectors to arbitrary point sets. Further, their precision can be improved by building multiple sets of code words for each item, involving different sets of random hyperplanes. So long as q shares at least one codeword with its true nearest neighbor, we will eventually

encounter a bucket containing both of these points.

Note that LSH has exactly the opposite goal from traditional hash functions used for cryptographic applications or to manage hash tables. Traditional hash functions seek to ensure that pairs of similar items result in wildly different hash values, so we can recognize changes and utilize the full range of the table. In contrast, LSH wants similar items to recieve the exact same hash code, so we can recognize similarity by collision. With LSH, nearest neighbors belong in the same bucket.

Locality sensitive hashing has other applications in data science, beyond nearest neighbor search. Perhaps the most important is constructing compressed feature representations from complicated objects, say video or music streams. LSH codes constructed from intervals of these streams define numerical values potentially suitable as features for pattern matching or model building.

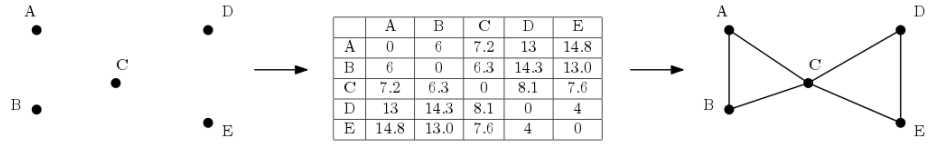
10.3 Graphs, Networks, and Distances

A *graph* $G = (V, E)$ is defined on a set of *vertices* V, and contains a set of *edges* E of ordered or unordered pairs of vertices from V. In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are directly connected by roads/edges. In analyzing human interactions, the vertices typically represent people, with edges connecting pairs of related souls.

Many other modern data sets are naturally modeled in terms of graphs or networks:

- *The Worldwide Web (WWW)*: Here there is a vertex in the graph for each webpage, with a directed edge (x, y) if webpage x contains a hyperlink to webpage y.
- *Product/customer networks*: These arise in any company that has many customers and types of products: be it Amazon, Netflix, or even the corner grocery store. There are two types of vertices: one set for customers and another for products. Edge (x, y) denotes a product y purchased by customer x.
- *Genetic networks*: Here the vertices represent the different genes/proteins in a particular organisms. Think of this as a parts list for the beast. Edge (x, y) denotes that there are interactions between parts x and y. Perhaps gene x regulates gene y, or proteins x and y bind together to make a larger complex. Such interaction networks encode considerable information about how the underlying system works.

Graphs and point sets are closely related objects. Both are composed of discrete entities (points or vertices) representing items in a set. Both of them encode important notions of distance and relationships, either near–far or connected–independent. Point sets can be meaningfully represented by graphs, and graphs by point sets.



	A	B	C	D	E
A	0	6	7.2	13	14.8
B	6	0	6.3	14.3	13.0
C	7.2	6.3	0	8.1	7.6
D	13	14.3	8.1	0	4
E	14.8	13.0	7.6	4	0

Figure 10.8: The pairwise distances between a set of points in space (left) define a complete weighted graph (center). Thresholding by a distance cutoff removes all long edges, leaving a sparse graph that captures the structure of the points (right).

10.3.1 Weighted Graphs and Induced Networks

The edges in graphs capture *binary relations*, where each edge (x, y) represents that there is a relationship between x and y. The existence of this relationship is sometimes all there is to know about it, as in the connection between webpages or the fact of someone having purchased a particular product.

But there is often an inherent measure of the strength or closeness of the relationship. Certainly we see it in road networks: each road segment has a length or travel time, which is essential to know for finding the best route to drive between two points. We say that a graph is *weighted* if every edge has a numerical value associated with it.

And this weight is often (but not always) naturally interpreted as a distance. Indeed one can interpret a data set of n points in space as a complete weighted graph on n vertices, where the weight of edge (x, y) is the geometric distance between points x and y in space. For many applications, this graph encodes all the relevant information about the points.

Graphs are most naturally represented by $n \times n$ *adjacency matrices*. Define a non-edge symbol x. Matrix M represents graph $G = (V, E)$ when $M[i, j] \neq x$ if and only if vertices $i, j \in V$ are connected by an edge $(i, j) \in E$. For unweighted networks, typically the edge symbol is 1 while $x = 0$. For distance weighted graphs, the weight of edge (i, j) is the cost of travel between them, so setting $x = \infty$ denotes the lack of any direct connection between i and j.

This matrix representation for networks has considerable power, because we can introduce all our tools from linear algebra to work with them. Unfortunately, it comes with a cost, because it can be hopelessly expensive to store $n \times n$ matrices once networks get beyond a few hundred vertices. There are more efficient ways to store large *sparse graphs*, with many vertices but relatively few pairs connected by edges. I will not discuss the details of graph algorithms here, but refer you with confidence to my book *The Algorithm Design Manual* [Ski08] for you to learn more.

Pictures of graphs/networks are often made by assigning each vertex a point in the plane, and drawing lines between these vertex-points to represent edges. Such *node-ink diagrams* are immensely valuable to visualize the structure of

the networks you are working with. They can be algorithmically constructed using *force-directed layout*, where edges act like springs to bring adjacent pairs of vertices close together, and non-adjacent vertices repel each other.

Such drawings establish the connection between graph structures and point positions. An *embedding* is a point representation of the vertices of a graph that captures some aspect of its structure. Performing a feature compression like eigenvalue or singular value decomposition (see Section 8.5) on the the adjacency matrix of a graph produces a lower-dimensional representation that serves as a point representation of each vertex. Other approaches to graph embeddings include DeepWalk, to be discussed in Section 11.6.3.

Take-Home Lesson: Point sets can be meaningfully represented by graphs/distance matrices, and graphs/distance matrices meaningfully represented by point sets (embeddings).

Geometric graphs defined by the distances between points are representative of a class of graphs I will call *induced networks*, where the edges are defined in a mechanical way from some external data source. This is a common source of networks in data science, so it is important to keep an eye out for ways that your data set might be turned into a graph.

Distance or similarity functions are commonly used to construct networks on sets of items. Typically we are interested in edges connecting each vertex to its k closest/most similar vertices. We get a sparse graph by keeping k modest, say $k \approx 10$, meaning that it can be easily worked with even for large values of n.

But there are other types of induced networks. Typical would be to connect vertices x and y whenever they have a meaningful attribute in common. For example, we can construct an induced social network on people from their resumes, linking any two people who worked at the same company or attended the same school in a similar period. Such networks tend to have a blocky structure, where there are large subsets of vertices forming fully connected cliques. After all, if x graduated from the same college as y, and y graduated from the same college as z, then this implies that (x, z) must also be an edge in the graph.

10.3.2 Talking About Graphs

There is a vocabulary about graphs that is important to know for working with them. Talking the talk is an important part of walking the walk. Several fundamental properties of graphs impact what they represent, and how we can use them. Thus the first step in any graph problem is determining the flavors of the graphs you are dealing with:

- *Undirected vs. Directed*: A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E. If not, we say that the graph is *directed*. Road networks *between* cities are typically undirected, since any large road has lanes going in both directions. Street networks *within* cities are almost

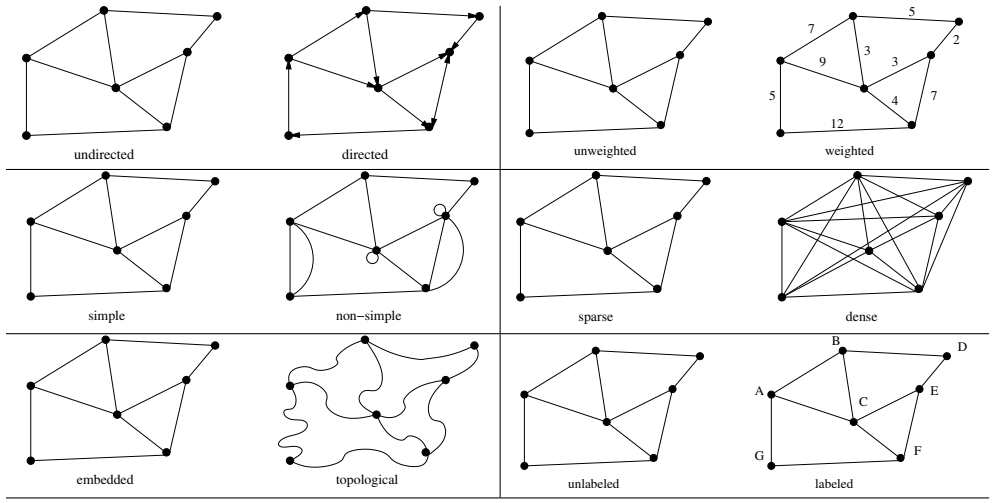


Figure 10.9: Important properties/flavors of graphs

always directed, because there are at least a few one-way streets lurking somewhere. Webpage graphs are typically directed, because the link from page x to page y need not be reciprocated.

- *Weighted vs. Unweighted*: As discussed in Section 10.3.1, each edge (or vertex) in a *weighted* graph G is assigned a numerical value, or weight. The edges of a road network graph might be weighted with their length, drive-time, or speed limit, depending upon the application. In *unweighted* graphs, there is no cost distinction between various edges and vertices.

  Distance graphs are inherently weighted, while social/web networks are generally unweighted. The difference determines whether the feature vectors associated with vertices are 0/1 or numerical values of importance, which may have to be normalized.

- *Simple vs. Non-simple*: Certain types of edges complicate the task of working with graphs. A *self-loop* is an edge (x, x), involving only one vertex. An edge (x, y) is a *multiedge* if it occurs more than once in the graph.

  Both of these structures require special care in preprocessing for feature generation. Hence any graph that avoids them is called *simple*. We often seek to remove both self-loops and multiedges at the beginning of analysis.

- *Sparse vs. Dense*: Graphs are *sparse* when only a small fraction of the total possible vertex pairs ($\binom{n}{2}$ for a simple, undirected graph on n vertices) actually have edges defined between them. Graphs where a large fraction of the vertex pairs define edges are called *dense*. There is no official boundary between what is called sparse and what is called dense,

but typically dense graphs have a quadratic number of edges, while sparse graphs are linear in size.

Sparse graphs are usually sparse for application-specific reasons. Road networks must be sparse graphs because of road junctions. The most ghastly intersection I've ever heard of was the endpoint of only nine different roads. k-nearest neighbor graphs have vertex degrees of exactly k. Sparse graphs make possible much more space efficient representations than adjacency matrices, allowing the representation of much larger networks.

- *Embedded vs. Topological* – A graph is *embedded* if the vertices and edges are assigned geometric positions. Thus, any drawing of a graph is an *embedding*, which may or may not have algorithmic significance.

  Occasionally, the structure of a graph is completely defined by the geometry of its embedding, as we have seen in the definition of the distance graph where the weights are defined by the Euclidean distance between each pair of points. Low-dimensional representations of adjacency matrices by SVD also qualify as embeddings, point representations that capture much of the connectivity information of the graph.

- *Labeled vs. Unlabeled* – Each vertex is assigned a unique name or identifier in a *labeled* graph to distinguish it from all other vertices. In *unlabeled* graphs no such distinctions are made.

  Graphs arising in data science applications are often naturally and meaningfully labeled, such as city names in a transportation network. These are useful as identifiers for representative examples, and also to provide linkages to external data sources where appropriate.

10.3.3 Graph Theory

Graph theory is an important area of mathematics which deals with the fundamental properties of networks and how to compute them. Most computer science students get exposed to graph theory through their courses in discrete structures or algorithms.

The classical algorithms for finding shortest paths, connected components, spanning trees, cuts, matchings and topological sorting can be applied to any reasonable graph. However, I have not seen these tools applied as generally in data science as I think they should be. One reason is that the graphs in data science tend to be very large, limiting the complexity of what can be done with them. But a lot is simply myopia: people do not see that a distance or similarity matrix is really just a graph than can take advantage of other tools.

I take the opportunity here to review the connections of these fundamental problems to data science, and encourage the interested reader to deepen their understanding through my algorithm book [Ski08].

- *Shortest paths*: For a distance "matrix" m, the value of $m[i,j]$ should reflect the minimum length path between vertices i and j. Note that independent estimates of pairwise distance are often inconsistent, and do not necessarily satisfy the triangle inequality. But when $m'[i,j]$ reflects the *shortest path distance* from i to j in any matrix m it *must* satisfy the metric properties. This may well present a better matrix for analysis than the original.

- *Connected components*: Each disjoint piece of a graph is called a *connected component*. Identifying whether your graph consists of a single component or multiple pieces is important. First, any algorithms you run will achieve better performance if you deal with the components independently. Separate components can be independent for sound reasons, e.g. there is no road crossing between the United States and Europe because of an ocean. But separate components might indicate trouble, such as processing artifacts or insufficient connectivity to work with.

- *Minimum spanning trees*: A spanning tree is the minimal set of edges linking all the vertices in a graph, essentially a proof that the graph is connected. The minimum weight spanning tree serves as the sparsest possible representation of the structure of the graph, making it useful for visualization. Indeed, we will show that minimum spanning trees have an important role in clustering algorithms in Section 10.5.

- *Edge cuts*: A cluster in a graph is defined by a subset of vertices c, with the property that (a) there is considerable similarity between pairs of vertices within c, and (b) there is weak connectivity between vertices in c and out of c. The edges (x,y) where $x \in c$ and $y \notin c$ define a cut separating the cluster from the rest of the graph, making finding such cuts an important aspect of cluster analysis.

- *Matchings*: Marrying off each vertex with a similar, loyal partner can be useful in many ways. Interesting types of comparisons become possible after such a *matching*. For example, looking at all close pairs that differ in one attribute (say gender) might shed light on how that variable impacts a particular outcome variable (think income or lifespan). Matchings also provide ways to reduce the effective size of a network. By replacing each matched pair with a vertex representing its centroid, we can construct a graph with half the vertices, but still representative of the whole.

- *Topological sorting*: Ranking problems (recall Chapter 4) impose a pecking order on a collection of items according to some merit criteria. *Topological sorting* ranks the vertices of a directed acyclic graph (DAG) so edge (i,j) implies that i ranks above j in the pecking order. Given a collection of observed constraints of the form "i should rank above j," topological sorting defines an item-order consistent with these observations.

10.4 PageRank

It is often valuable to categorize the relative importance of vertices in a graph. Perhaps the simplest notion is based on vertex *degree*, the number of edges connecting vertex v to the rest of the graph. The more connected a vertex is, the more important it probably is.

The degree of vertex v makes a good feature to represent the item associated with v. But even better is *PageRank* [BP98], the original secret sauce behind Google's search engine. PageRank ignores the textual content of webpages, to focus only on the structure of the hyperlinks between pages. The more important pages (vertices) should have higher in-degree than lesser pages, for sure. But the importance of the pages that link to you also matter. Having a large stable of contacts recommending you for a job is great, but it is even better when one of them is currently serving as the President of the United States.

PageRank is best understood in the context of random walks along a network. Suppose we start from an arbitrary vertex and then randomly select an outgoing link uniformly from the set of possibilities. Now repeat the process from here, jumping to a random neighbor of our current location at each step. The PageRank of vertex v is a measure of probability that, starting from a random vertex, you will arrive at v after a long series of such random steps. The basic formula for the PageRank of v ($PR(v)$) is:

$$PR_j(v) = \sum_{(u,v)\in E} \frac{PR_{j-1}(u)}{out - degree(u)}$$

This is a recursive formula, with j as the iteration number. We initialize $PR_0(v_i) = 1/n$ for each vertex v_i in the network, where $1 \leq i \leq n$. The PageRank values will change in each iteration, but converge surprisingly quickly to stable values. For undirected graphs, this probability is essentially the same as each vertex's in-degree, but much more interesting things happen with directed graphs.

In essence, PageRank relies on the idea that if all roads lead to Rome, Rome must be a pretty important place. It is the paths *to* your page that counts. This is what makes PageRank hard to game: other people must link to your webpage, and whatever shouting you do about yourself is irrelevant.

There are several tweaks one can make to this basic PageRank formula to make the results more interesting. We can allow the walk to jump to an arbitrary vertex (instead of a linked neighbor) to allow faster diffusion in the network. Let p be the probability of following a link in the next step, also known as the *damping factor*. Then

$$PR_j(v) = \sum_{(u,v)\in E} \frac{(1-p)}{n} + p\frac{PR_{j-1}(u)}{\text{out-degree}(u)}$$

where n is the number of vertices in the graph. Other enhancements involve making changes to the network itself. By adding edges from every vertex to a

	PageRank PR1 (all pages)
1	Napoleon
2	George W. Bush
3	Carl Linnaeus
4	Jesus
5	Barack Obama
6	Aristotle
7	William Shakespeare
8	Elizabeth II
9	Adolf Hitler
10	Bill Clinton

	PageRank PR2 (only people)
1	George W. Bush
2	Bill Clinton
3	William Shakespeare
4	Ronald Reagan
5	Adolf Hitler
6	Barack Obama
7	Napoleon
8	Richard Nixon
9	Franklin D. Roosevelt
10	Elizabeth II

Table 10.1: Historical individuals with the highest PageRank in the 2010 English Wikipedia, drawn over the full Wikipedia graph (left) and when restricted to links to other people (right).

single super-vertex, we ensure that random walks cannot get trapped in some small corner of the network. Self-loops and parallel edges (multiedges) can be deleted to avoid biases from repetition.

There is also a linear algebraic interpretation of PageRank. Let M be a matrix of vertex-vertex transition probabilities, so M_{ij} is the probability that our next step from i will be to j. Clearly $M_{ij} = 1/\text{out-degree}(i)$ if there is a directed edge from i to j, and zero if otherwise. The jth round estimate for the PageRank vector PR_j can be computed as

$$PR_j = M \cdot PR_{j-1}$$

After this estimate converges, $PR = M \cdot PR$, or $\lambda U = MU$ where $\lambda = 1$ and U represents the PageRank vector. This is the defining equation for eigenvalues, so the $n \times 1$ vector of PageRank values turns out to be the principle eigenvector of the transition probability matrix defined by the links. Thus iterative methods for computing eigenvectors and fast matrix multiplication leads to efficient PageRank computations.

How well does PageRank work at smoking out the most central vertices? To provide some intuition, we ran PageRank on the link network from the English edition of Wikipedia, focusing on the pages associated with people. Table 10.1 (left) lists the ten historical figures with the highest PageRank.

These high-PageRank figures are all readily recognized as very significant people. The least familiar among them is probably *Carl Linnaeus (1707–1778)* [46], biology's "father of taxonomy" whose Linnaean system (Genus species; e.g., *Homo sapiens*) is used to classify all life on earth. He was a great scientist, but why is he *so* very highly regarded by PageRank? The Wikipedia pages of all the plant and animal species he first classified link back to him, so thousands of life forms contribute prominent paths to his page.

The Linnaeus example points out a possible weakness of PageRank: do we really want plants and other inanimate objects voting on who the most promi-

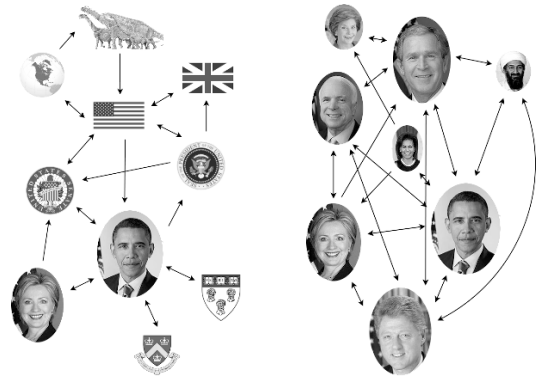


Figure 10.10: PageRank graphs for *Barack Obama*, over all Wikipedia pages (left) and when restricted only to people (right)

nent people are? Figure 10.10 (left) shows a subset of the full PageRank graph for *Barack Obama* [91]. Note, for example, that although the links associated with Obama seem very reasonable, we can still reach Obama in only two clicks from the Wikipedia page for Dinosaurs. Should extinct beasts contribute to the President's centrality?

Adding and deleting sets of edges from a given network gives rise to different networks, some of which better reveal underlying significance using PageRank. Suppose we compute PageRank (denoted PR2) using only the Wikipedia edges linking people. This computation would ignore any contribution from places, organizations, and lower organisms. Figure 10.10 (right) shows a sample of the PageRank graph for *Barack Obama* [91] when we restrict it to only people.

PageRank on this graph favors a slightly different cohort of people, shown in Table 10.1 (right). Jesus, Linnaeus and *Aristotle (384–322 B.C.)* [8] are now gone, replaced by three recent U.S. presidents – who clearly have direct connections from many important people. So which version of PageRank is better, PR1 or PR2? Both seem to capture reasonable notions of centrality with a substantial but not overwhelming correlation (0.68), so both make sense as potential features in a data set.

10.5 Clustering

Clustering is the problem of grouping points by similarity. Often items come from a small number of logical "sources" or "explanations", and clustering is a good way to reveal these origins. Consider what would happen if an alien species were to come across height and weight data for a large number of humans. They would presumably figure out that there seem to be two clusters representing distinct populations, one consistently bigger than the other. If the aliens were really on the ball, they might call these populations 'men" and "women". Indeed, the two height-weight clusters in Figure 10.11 are both highly

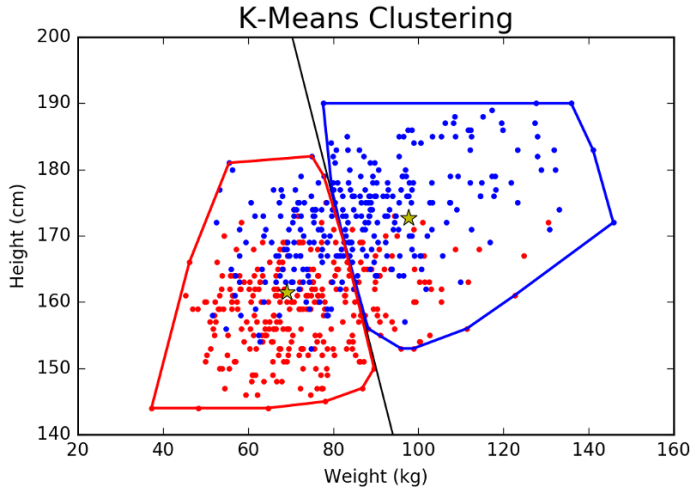


Figure 10.11: Clustering people in weight-height space, using 2-means clustering. The left cluster contains 240 women and 112 men, while the right cluster contains 174 men to 54 women. Compare this to the logistic regression classifier trained on this same data set, in Figure 9.17.

concentrated in one particular gender.

Patterns on a two-dimensional dot plot are generally fairly easy to see, but we often deal with higher-dimensional data that humans cannot effectively visualize. Now we need algorithms to find these patterns for us. Clustering is perhaps the first thing to do with any interesting data set. Applications include:

- *Hypothesis development*: Learning that there appear to be (say) four distinct populations represented in your data set should spark the question as to why they are there. If these clusters are compact and well-separated enough, there *has* to be a reason and it is your business to find it. Once you have assigned each element a cluster label, you can study multiple representatives of the same cluster to figure out what they have in common, or look at pairs of items from different clusters and identify why they are different.

- *Modeling over smaller subsets of data*: Data sets often contain a very large number of rows (n) relative to the number of feature columns (m): think the taxi cab data of 80 million trips with ten recorded fields per trip. Clustering provides a logical way to partition a large single set of records in a (say) a hundred distinct subsets each ordered by similarity. Each of these clusters still contains more than enough records to fit a forecasting model on, and the resulting model may be more accurate on this restricted class of items then a general model trained over all items. Making a forecast now involves identifying the appropriate cluster your

query item q belongs to, via a nearest neighbor search, and then using the appropriate model for that cluster to make the call on q.

- *Data reduction*: Dealing with millions or billions of records can be overwhelming, for processing or visualization. Consider the computational cost of identifying the nearest neighbor to a given query point, or trying to understand a dot plot with a million points. One technique is to cluster the points by similarity, and then appoint the *centroid* of each cluster to represent the entire cluster. Such nearest neighbor models can be quite robust because you are reporting the consensus label of the cluster, and it comes with a natural measure of confidence: the accuracy of this consensus over the full cluster.

- *Outlier detection*: Certain items resulting from any data collection procedure will be unlike all the others. Perhaps they reflect data entry errors or bad measurements. Perhaps they signal lies or other misconduct. Or maybe they result from the unexpected mixture of populations, a few strange apples potentially spoiling the entire basket.

  Outlier detection is the problem of ridding a data set of discordant items, so the remainder better reflects the desired population. Clustering is a useful first step to find outliers. The cluster elements furthest from their assigned cluster center don't really fit well there, but also don't fit better anywhere else. This makes them candidates to be outliers. Since invaders from another population would tend to cluster themselves together, we may well cast suspicions on small clusters whose centers lie unusually far from all the other cluster centers.

Clustering is an inherently ill-defined problem, since proper clusters depend upon context and the eye of the beholder. Look at Figure 10.12. How many different clusters do you see there? Some see three, other see nine, and others vote for pretty much any number in between.

How many clusters you see depends somewhat upon how many clusters you want to see. People can be clustered into two groups, the *lumpers* and the *splitters*, depending upon their inclination to make fine distinctions. Splitters look at dogs, and see poodles, terriers, and cocker spaniels. Lumpers look at dogs, and see mammals. Splitters draw more exciting conclusions, while lumpers are less likely to overfit their data. Which mindset is most appropriate depends upon your task.

Many different clustering algorithms have been developed, and we will review the most prominent methods (k-means, agglomerative clustering, and spectral clustering) in the sections below. But it is easy to get too caught up in the differences between methods. If your data exhibits strong-enough clusters, any method is going to find something like it. But when an algorithm returns clusters with very poor coherence, usually your data set is more to blame than the algorithm itself.

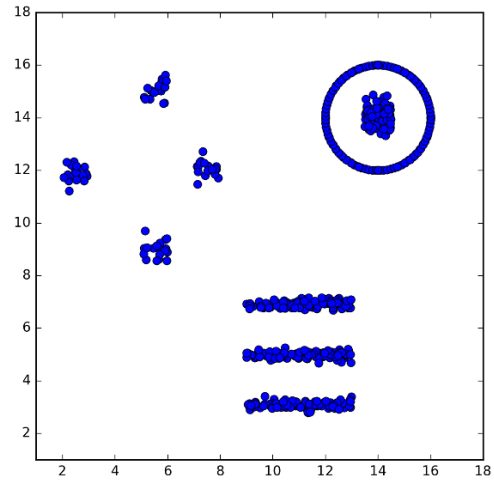


Figure 10.12: How many clusters do you see here?

Take-Home Lesson: Make sure you are using a distance metric which accurately reflects the similarities that you are looking to find. The specific choice of clustering algorithm usually proves much less important than the similarity/distance measure which underlies it.

10.5.1 k-means Clustering

We have been somewhat lax in defining exactly *what* a clustering algorithm should return as an answer. One possibility is to label each point with the name of the cluster that it is in. If there are k clusters, these labels can be the integers 1 through k, where labeling point p with i means it is in the ith cluster. An equivalent output representation might be k separate lists of points, where list i represents all the points in the ith cluster.

But a more abstract notion reports the *center point* of each cluster. Typically we think of natural clusters as compact, Gaussian-like regions, where there is an ideal center defining the location where the points "should" be. Given the set of these centers, clustering the points becomes easy: simply assign each point p to the center point C_i closest to it. The ith cluster consists of all points whose nearest center is C_i.

k-means clustering is a fast, simple-to-understand, and generally effective approach to clustering. It starts by making a guess as to where the cluster centers might be, evaluates the quality of these centers, and then refines them to make better center estimates.

K-means clustering

Select k points as initial cluster centers $C_1, \ldots, C_k$.
Repeat until convergence {
  For $1 \leq i \leq n$, map point p_i to its nearest cluster center C_j

  Compute centroid C'_j of the points nearest C_j, for $1 \leq j \leq k$

  For all $1 \leq j \leq k$, set $C_j = C'_j$

}

Figure 10.13: Pseudocode for the k-means clustering algorithm.

The algorithm starts by assuming that there will be exactly k clusters in the data, and then proceeds to pick initial centers for each cluster. Perhaps this means randomly selecting k points from the set of n points S and calling them centers, or selecting k random points from the bounding box of S. Now test each of the n points against all k of the centers, and assign each point in S to its nearest current center. We can now compute a better estimate of the center of each cluster, as the centroid of the points assigned to it. Repeat until the cluster assignments are sufficiently stable, presumably when they have not changed since the previous generation. Figure 10.13 provides pseudocode of this k-means procedure.

Figure 10.14 presents an animation of k-means in action. The initial guesses for the cluster centers are truly bad, and the initial assignments of points to centers splits the real clusters instead of respecting them. But the situation rapidly improves, with the centroids drifting into positions that separates the points in the desired way. Note that the k-means procedure does not necessarily terminate with the best possible set of k centers, only at a locally-optimal solution that provides a logical stopping point. It is a good idea to repeat the entire procedure several times with different random initializations and accept the best clustering found over all. The *mean squared error* is the sum of squares of the distance between each point P_i and its center C_j, divided by the number of points n. The better of two clusterings can be identified as having lower mean squared error, or some other reasonable error statistic.

Centers or Centroids?

There are at least two possible criteria for computing a new estimate for the center point as a function of the set S' of points assigned to it. The *centroid* C of a point set is computed by taking the average value of each dimension. For

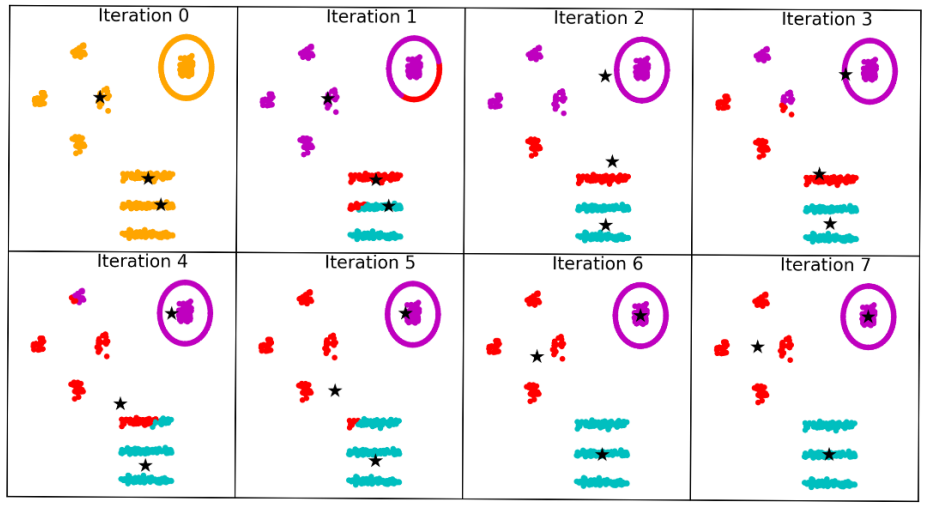


Figure 10.14: The iterations of k-means (for $k = 3$) as it converges on a stable and accurate clustering. Fully seven iterations are needed, because of the unfortunate placement of the three initial cluster centers near the logical center.

the dth dimension,

$$C_d = \frac{1}{|S'|} \sum_{p \in S'} p[d]$$

The centroid serves as the *center of mass* of S', the place where the vectors defined through this point sum to zero. This balance criteria defines a natural and unique center for any S'. Speed of computation is another nice thing about using the centroid. For n d-dimensional points in S', this takes $O(nd)$ time, meaning linear in the input size of the points.

For numerical data points, using the centroid over an appropriate L_k metric (like Euclidean distance) should work just fine. However, centroids are not well defined when clustering data records with non-numerical attributes, like categorical data. What is the centroid of 7 blonds, 2 red-heads, and 6 gray-haired people? We have discussed how to construct meaningful distance functions over categorical records. The problem here is not so much measuring similarity, as constructing a representative center.

There is a natural solution, sometimes called the *k-mediods* algorithm. Suppose instead of the centroid we define the centermost point C in S' to be the cluster representative. This is the point which minimizes the sum of distances

to all other points in the cluster:

$$C = \arg\min_{c \in S'} \sum_{i=1}^{n} d(c, p_i)$$

An advantage of using a centerpoint to define the cluster is that it gives the cluster a potential name and identity, assuming the input points correspond to items with identifiable names.

Using the centermost input example as center means we can run k-means so long as we have a meaningful distance function. Further, we don't lose very much precision by picking the centermost point instead of the centroid. Indeed, the sum of distances through the centermost point is at most twice that of the centroid, on numerical examples where the centroid can be computed. The big win of the centroid is that it can be computed faster than the centermost vertex, by a factor of n.

Using center vertices to represent clusters permits one to extend k-means naturally to graphs and networks. For weighted graphs, it is natural to employ a shortest path algorithm to construct a matrix D such that $D[i, j]$ is the length of the shortest path in the graph from vertex i to vertex j. Once D is constructed, k-means can proceed by reading the distances off this matrix, instead of calling a distance function. For unweighted graphs, a linear time algorithm like breadth-first search can be efficiently used to compute graph distances on demand.

How Many Clusters?

Inherent to the interpretation of k-means clustering is the idea of a *mixture model*. Instead of all our observed data coming from a single source, we presume that our data is coming from k different populations or sources. Each source is generating points to be like its center, but with some degree of variation or error. The question of how many clusters a data set has is fundamental: how many different populations were drawn upon when selecting the sample?

The first step in the k-means algorithm is to initialize k, the number of clusters in the given data set. Sometimes we have a preconception of how many clusters we want to see: perhaps two or three for balance or visualization, or maybe 100 or 1000 as a proxy for "many" when partitioning a large input file into smaller sets for separate modeling.

But generally speaking this is a problem, because the "right" number of clusters is usually unknown. Indeed, the primary reason to cluster in the first place is our limited understanding of the structure of the data set.

The easiest way to find the right k is to try them all, and then pick the best one. Starting from $k = 2$ to as high as you feel you have time for, perform k-means and evaluate the resulting clustering according to the mean squared error (MSE) of the points from their centers. Plotting this yields an error curve, as shown in Figure 10.16. The error curve for random centers is also provided.

Both error curves show the MSE of points from their centers decreasing as we allow more and more cluster centers. But the wrong interpretation would

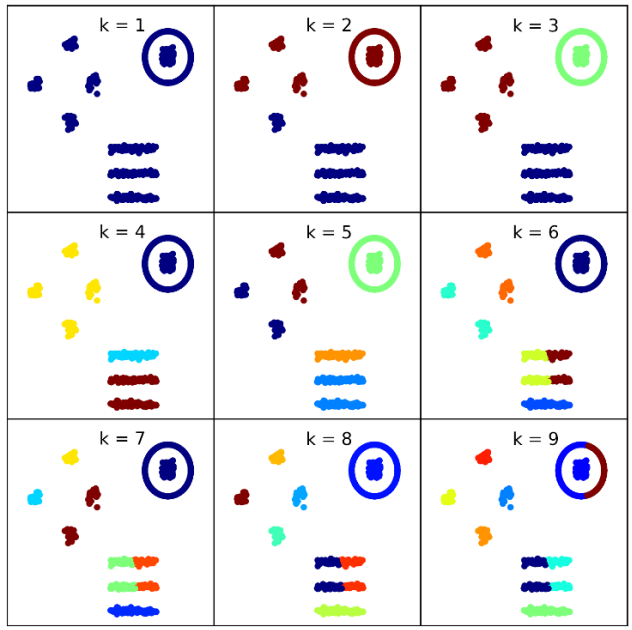


Figure 10.15: Running k-means for $k = 1$ to $k = 9$. The "right" clustering is found for $k = 3$, but the algorithm is unable to properly distinguish between nested circular clusters and long thin clusters for large k.

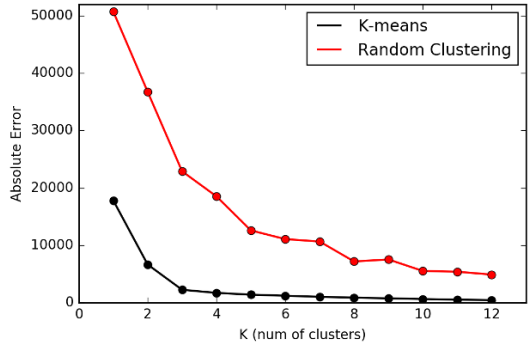


Figure 10.16: The error curve for k-means clustering on the point set of Figure 10.12, showing a bend in the elbow reflecting the three major clusters in the data. The error curve for random cluster centers is shown for comparison.

be to suggest we need k as large as possible, because the MSE *should* decrease when allowing more centers. Indeed, inserting a new center at a random position r into a previous k-means solution can only decrease the mean squared error, by happening to land closer to a few of the input points than their previous center. This carves out a new cluster around r, but presumably an even better clustering would have been found running by k-means from scratch on $(k+1)$ centers.

What we seek from the error curve in Figure 10.16 is the value k where the *rate* of decline decreases, because we have exceeded number of true sources, and so each additional center is acting like a random point in the previous discussion. The error curve should look something like an arm in typing position: it slopes down rapidly from shoulder to elbow, and then slower from the elbow to the wrist. We want k to be located exactly at the elbow. This point might be easier to identify when compared to a similar MSE error plot for random centers, since the relative rate of error reduction for random centers should be analogous to what we see past the elbow. The slow downward drift is telling us the extra clusters are not doing anything special for us.

Each new cluster center adds d parameters to the model, where d is the dimensionality of the point set. Occam's razor tells us that the simplest model is best, which is the philosophical basis for using the bend in the elbow to select k. There are formal criterias of merit which incorporate both the number of parameters and the prediction error to evaluate models, such as the *Akaike information criterion* (AIC). However, in practice you should feel confident in making a reasonable choice for k based on the shape of the error curve.

Expectation Maximization

The k-means algorithm is the most prominent example of a class of learning algorithms based on *expectation maximization* (EM). The details require more formal statistics than I am prepared to delve into here, but the principle can be observed in the two logical steps of the k-means algorithm: (a) assigning points to the estimated cluster center that is closest to them, and (b) using these point assignments to improve the estimate of the cluster center. The assignment operation is the expectation or *E-step* of the algorithm, while the centroid computation is the parameter maximization or *M-step*.

The names "expectation" and "maximization" have no particular resonance to me in terms of the k-means algorithm. However, the general form of an iterative parameter-fitting algorithm which improves parameters in rounds based on the errors of the previous models does seem a sensible thing to do. For example, perhaps we might have partially labeled classification data, where there are relatively few training examples confidently assigned to the correct class. We can build classifiers based on these training examples, and use them to assign the unlabeled points to candidate classes. This presumably defines larger training sets, so we should be able to fit a better model for each class. Now reassigning the points and iterating again should converge on a better model.

Figure 10.17: Agglomerative clustering of gene expression data.

10.5.2 Agglomerative Clustering

Many sources of data are generated from a process defined by an underlying hierarchy or taxonomy. Often this is the result of an evolutionary process: In the Beginning there was one thing, which repeatedly bifurcated to create a rich universe of items. All animals and plant species are the result of an evolutionary process, and so are human languages and cultural/ethic groupings. To a lesser but still real extent, so are products like movies and books. This book can be described as a "Data Science Textbook", which is an emerging sub-genre split off from "Computer Science Textbook", which logically goes back to "Engineering Textbook", to "Textbook", to "Non-Fiction" eventually back to the original source: perhaps "Book".

Ideally, in the course of clustering items we will reconstruct these evolutionary histories. This goal is explicit in *agglomerative clustering*, a collection of bottom-up methods that repeatedly merge the two nearest clusters into a bigger super-cluster, defining a rooted tree whose leaves are the individual items and whose root defines the universe.

Figure 10.17 illustrates agglomerative clustering applied to gene expression data. Here each column represents a particular gene, and each row the results of an experiment measuring how active each gene was in a particular condition. As an analogy, say each of the columns represented different people, and one particular row assessed their spirits right after an election. Fans of the winning party would be more excited than usual (green), while voters for the losing team would be depressed (red). Most of the rest of the world wouldn't care (black). As it is with people, so it is with genes: different things turn them on and off, and analyzing gene expression data can reveal what makes them tick.

So how do we read Figure 10.17? By inspection it is clear that there are blocks of columns which all behave similarly, getting turned on and turned off in similar conditions. The discovery of these blocks are reflected in the tree above the matrix: regions of great similarity are associated with small branchings. Each node of the tree represents the merging of two clusters. The height of the node is proportional to the distance between the two clusters being merged. The taller the edge, the more dodgy the notion that these clusters *should* be merged. The columns of the matrix have been permuted to reflect this tree

organization, enabling us to visualize hundreds of genes quantified in fourteen dimensions (with each row defining a distinct dimension).

Biological clusterings are often associated with such *dendograms* or *phylogenic trees*, because they are the result of an evolutionary process. Indeed, the clusters of similar gene expression behavior seen here are the results of the organism evolving a new function, that changes the response of certain genes to a particular condition.

Using Agglomerative Trees

Agglomerative clustering returns a tree on top of the groupings of items. After cutting the longest edges in this tree, what remains are the disjoint groups of items produced by clustering algorithms like k-means. But this tree is a marvelous thing, with powers well beyond the item partitioning:

- *Organization of clusters and subclusters*: Each internal node in the tree defines a particular cluster, comprised of all the leaf-node elements below it. But the tree describes hierarchy among these clusters, from the most refined/specific clusters near the leaves to the most general clusters near the root. Ideally, nodes of a tree define nameable concepts: natural groupings that a domain expert could explain if asked. These various levels of granularity are important, because they define structural concepts we might not have noticed prior to doing clustering.

- *Visualization of the clustering process*: A drawing of this agglomeration tree tells us a lot about the clustering process, particularly if the drawing reflects the cost of each merging step. Ideally there will be very long edges near the root of the tree, showing that the highest-level clusters are well separated and belong in distinct groupings. We can tell if the groupings are balanced, or whether the high level groupings are of substantially different sizes. Long chains of merging small clusters into a big cluster is generally a bad sign, although the choice of merging criteria (to be discussed below) can bias the shape of the tree. Outliers show up nicely on a phylogenic tree, as singleton elements or small clusters that connect near the root through long edges.

- *Natural measure of cluster distance*: An interesting property of any tree T is that there is exactly one path in T between any two nodes x and y. Each internal vertex in an agglomerative clustering tree has a weight associated with it, the cost of merging together the two subtrees below it. We can compute a "cluster distance" between any two leaves by the sum of the merger costs on the path between them. If the tree is good, this can be more meaningful than the Euclidean distance between the records associated with x and y.

- *Efficient classification of new items*: One important application for clustering is classification. Suppose have agglomeratively clustered the prod-

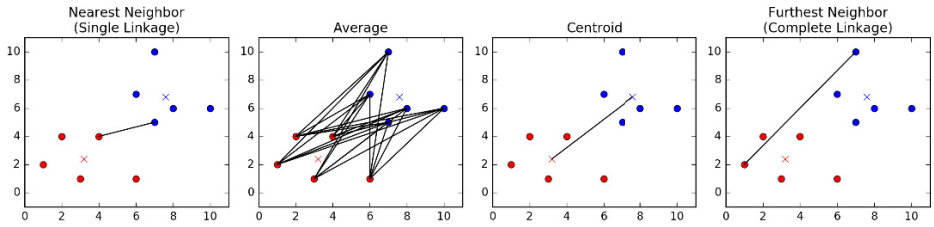


Figure 10.18: Four distance measures for identifying the nearest pair of clusters.

ucts in a store, to build a taxonomy of clusters. Now a new part comes along. What category should it be classified under?

For k-means, each of the c clusters are categorized by their centroid, so classifying a new item q reduces to computing the distance between q and all c centroids to identify the nearest cluster. A hierarchical tree provides a potentially faster method. Suppose we have precomputed the centroids of all the leaves on the left and right subtrees beneath each node. Identifying the right position in the hierarchy for a new item q starts by comparing q to the centroids of the root's left and right subtrees. The nearest of the two centroids to q defines the appropriate side of the tree, so we resume the search there one level down. This search takes time proportional to the height of the tree, instead of the number of leaves. This is typically an improvement from n to $\log n$, which is much better.

Understand that binary merging trees can be drawn in many different ways that reflect exactly the same structure, because there is no inherent notion of which is the left child and which is the right child. This means that there are 2^{n-1} distinct permutations of the n leaves possible, by flipping the direction of any subset of the $n-1$ internal nodes in the tree. Realize this when trying to read such a taxonomy: two items which look far away in left-right order might well have been neighbors had this flipping been done in a different way. And the rightmost node of the left subtree might be next to the leftmost node in the right subtree, even though they are really quite far apart in the taxonomy.

Building Agglomerative Cluster Trees

The basic agglomerative clustering algorithm is simple enough to be described in two sentences. Initially, each item is assigned to its own cluster. Merge the two closest clusters into one by putting a root over them, and repeat until only one cluster remains.

All that remains is to specify how to compute the distance between clusters. When the clusters contain single items, the answer is easy: use your favorite distance metric like L_2. But there are several reasonable answers for the distance

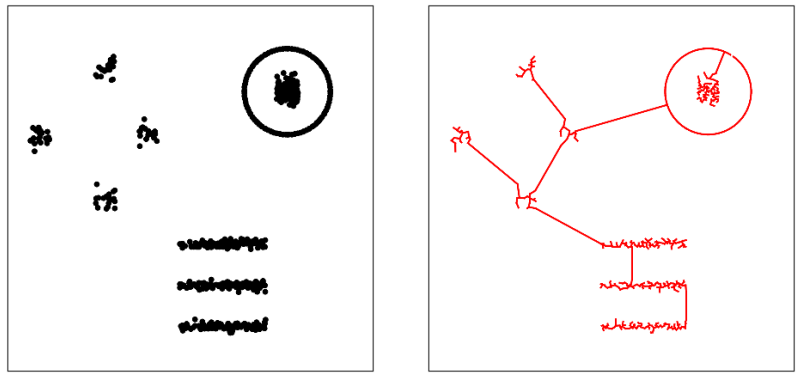


Figure 10.19: Single linkage clustering is equivalent to finding the minimum spanning tree of a network.

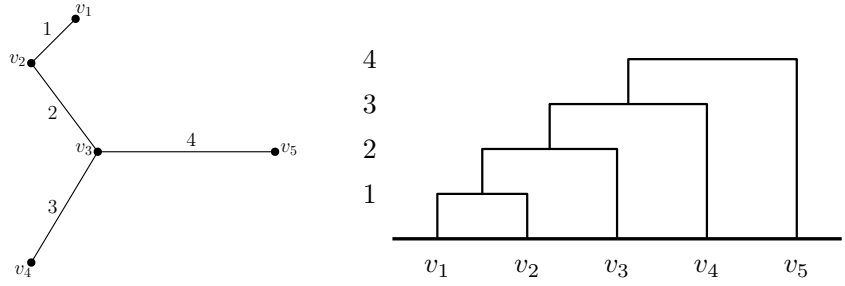


Figure 10.20: Kruskal's algorithm for minimum spanning tree is indeed single-linkage agglomerative clustering, as shown by the cluster tree on right.

between two non-trivial clusters, which lead to different trees on the same input, and can have a profound impact on the shape of the resulting clusters. The leading candidates, illustrated in Figure 10.18, are:

- *Nearest neighbor (single link)*: Here the distance between clusters C_1 and C_2 is defined by the closest pair of points spanning them:

$$d(C_1, C_2) = \min_{x \in C_1, y \in C_2} ||x - y||$$

  Using this metric is called *single link clustering*, because the decision to merge is based solely on the single closest link between the clusters.

  The *minimum spanning tree* of a graph G is tree drawn from the edges of G connecting all vertices at lowest total cost. Agglomerative clustering with the single link criteria is essentially the same as Kruskal's algorithm, which creates the *minimum spanning tree* (MST) of a graph by repeatedly adding the lowest weight edge remaining which does not create a cycle in the emerging tree.

The connection between the MST (with n nodes and $n-1$ edges) and the cluster tree (with n leaves, $n-1$ internal nodes, and $2n-2$ edges) is somewhat subtle: the order of insertion edges in the MST from smallest to largest describes the order of merging in the cluster tree, as shown in Figure 10.20.

The Platonic ideal of clusters are as compact circular regions, which generally radiate out from centroids, as in k-means clustering. By contrast, single-link clustering tends to create relatively long, skinny clusters, because the merging decision is based only on the nearness of boundary points. Single link clustering is fast, but tends to be error prone, as outlier points can easily suck two well-defined clusters together.

- *Average link*: Here we compute distance between all pairs of cluster-spanning points, and average them for a more robust merging criteria than single-link:

$$d(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{x \in C_1} \sum_{y \in C_2} ||x - y||$$

This will tend to avoid the skinny clusters of single-link, but at a greater computational cost. The straightforward implementation of average link clustering is $O(n^3)$, because each of the n merges will potentially require touching $O(n^2)$ edges to recompute the nearest remaining cluster. This is n times slower than single link clustering, which can be implemented in $O(n^2)$ time.

- *Nearest centroid*: Here we maintain the centroid of each cluster, and merge the cluster-pair with the closest centroids. This has two main advantages. First, it tends to produce clusters similar to average link, because outlier points in a cluster get overwhelmed as the cluster size (number of points) increases. Second, it is much faster to compare the centroids of the two clusters than test all $|C_1||C_2|$ point-pairs in the simplest implementation. Of course, centroids can only be computed for records with all numerical values, but the algorithm can be adapted to use the centermost point in each cluster (medioid) as a representative in the general case.

- *Furthest link*: Here the cost of merging two clusters is the farthest pair of points between them:

$$d(C_1, C_2) = \max_{x \in C_1, y \in C_2} ||x - y||$$

This sounds like madness, but this is the criteria which works hardest to keep clusters round, by penalizing mergers with distant outlier elements.

Which of these is best? As always in this business, it depends. For very large data sets, we are most concerned with using the fastest algorithms , which are typically single linkage or nearest centroid with appropriate data structures. For small to modest-sized data sets, we are most concerned with quality, making more robust methods attractive.

10.5.3 Comparing Clusterings

It is a common practice to try several clustering algorithms on the same data set, and use the one which looks best for our purposes. The clusterings produced by two different algorithms should be fairly similar if both algorithms are doing reasonable things, but it is often of interest to measure exactly how similar they are. This means we need to define a similarity or distance measure on clusterings.

Every cluster is defined by a subset of items, be they points or records. The *Jaccard similarity* $J(s_1, s_2)$ of sets s_1 and s_2 is defined as the ratio of their intersection and union:

$$J(s_1, s_2) = \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|}$$

Because the intersection of two sets is always no bigger than the union of their elements, $0 \leq J(s_1, s_2) \leq 1$. Jaccard similarity is a generally useful measure to know about, for example, in comparing the similarity of the k nearest neighbors of a point under two different distance metrics, or how often the top elements by one criteria match the top elements by a different metric.

This similarity measure can be turned into a proper distance metric $d(s_1, s_2)$ called the *Jaccard distance*, where

$$d(s_1, s_2) = 1 - J(s_1, s_2)$$

This distance function only takes on values between 0 and 1, but satisfies all of the properties of a metric, including the triangle inequality.

Each clustering is described by a partition of the universal set, and may have many parts. The *Rand index* is a natural measure of similarity between two clusterings c_1 and c_2. If the clusterings are compatible, then any pair of items in the same subset of c_1 should be in the same subset of c_2, and any pairs in different clusters of c_1 should be separated in c_2. The Rand index counts the number of such consistent pairs of items, and divides it by the total number of pairs $\binom{n}{2}$ to create a ratio from 0 to 1, where 1 denotes identical clusterings.

10.5.4 Similarity Graphs and Cut-Based Clustering

Recall our initial discussion of clustering, where I asked asked how many clusters you saw in the point set repeated in Figure 10.21. To come up with the reasonable answer of nine clusters, your internal clustering algorithm had to manage tricks like classifying a ring around a central blob as two distinct clusters, and avoid merging two lines that move suspiciously close to each other. k-means doesn't have a chance of doing this, as shown in Figure 10.21 (left), because it always seeks circular clusters and is happy to split long stringy clusters. Of the agglomerative clustering procedures, only single-link with exactly the right threshold might have a chance to do the right thing, but it is easily fooled into merging two clusters by a single close point pair.

Clusters are not always round. Recognizing those that are not requires a high-enough *density* of points that are sufficiently *contiguous* that we are not

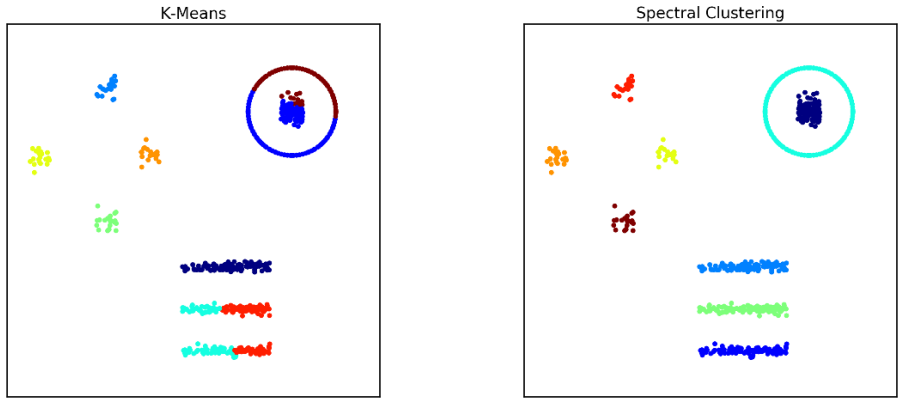


Figure 10.21: The results of k-means (left) and cut-based spectral clustering (right) on our 9-cluster example. Spectral clustering correctly finds connected clusters here that k-means cannot.

tempted to cut a cluster in two. We seek clusters that are *connected* in an appropriate similarity graph.

An $n \times n$ *similarity matrix* S scores how much alike each pair of elements p_i and p_j are. Similarity is essentially the inverse of distance: when p_i is close to p_j, then the item associated with p_i must be similar to that of p_j. It is natural to measure similarity on a scale from 0 to 1, where 0 represents completely different and 1 means identical. This can be realized by making $S[i,j]$ an inverse exponential function of distance, regulated by a parameter β:

$$S[i,j] = e^{-\beta||p_i - p_j||}$$

This works because $e^0 = 1$ and $e^{-x} = 1/e^x \to 0$ as $x \to \infty$.

A *similarity graph* has a weighted edge (i,j) between each pair of vertices i and j reflecting the similar of p_i and p_j. This is exactly the similarity matrix described above. However, we can make this graph sparse by setting all small terms ($S[i,j] \leq t$ for some threshold t) to zero. This greatly reduces the number of edges in the graph. We can even turn it into an unweighted graph by setting the weight to 1 for all $S[i,j] > t$.

Cuts in Graphs

Real clusters in similarity graphs have the appearance of being dense regions which are only loosely connected to the rest of graph. A cluster C has a weight which is a function of the edges within the cluster:

$$W(C) = \sum_{x \in C} \sum_{y \in C} S[i,j]$$

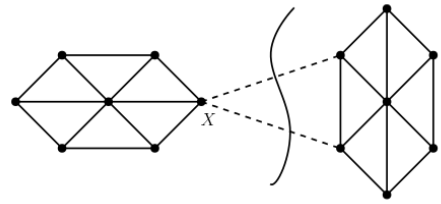


Figure 10.22: Low weight cuts in similarity graphs identify natural clusters.

The edges connecting C to the rest of the graph define a *cut*, meaning the set of edges who have one vertex in C and the other in the rest of the graph $(V - C)$. The weight of this cut $W'(C)$ is defined:

$$W'(C) = \sum_{x \in C} \sum_{y \in V - C} S[i, j]$$

Ideally clusters will have a high weight $W(C)$ but a small cut $W'(C)$, as shown in Figure 10.22. The *conductance* of cluster C is the ratio of cut weight over internal weight $(W'(C)/W(C))$, with better clusters having lower conductance.

Finding low conductance clusters is a challenge. Help comes, surprisingly, from linear algebra. The similarity matrix S is a symmetric matrix, meaning that it has an eigenvalue decomposition as discussed in Section 8.5. We saw that the leading eigenvector results in a blocky approximation to S, with the contribution of additional eigenvectors gradually improving the approximation. Dropping the smallest eigenvectors removes either details or noise, depending upon the interpretation.

Note that the ideal similarity matrix *is* a blocky matrix, because within each cluster we expect a dense connection of highly-similar pairs, with little cross talk to vertices of other clusters. This suggests using the eigenvectors of S to define robust features to cluster the vertices on. Performing k-means clustering on this transformed feature space will recover good clusters.

This approach is called *spectral clustering*. We construct an appropriately normalized similarity matrix called the *Laplacian*, where $L = D - S$ and D is the degree-weighted identity matrix, so $D[i, i] = \sum_j S[i, j]$. The k most important eigenvectors of L define an $n \times k$ feature matrix. Curiously, the most valuable eigenvectors for clustering here turn out to have the *smallest* non-zero eigenvalues, due to special properties of the Laplacian matrix. Performing k-means clustering in this feature space generates highly connected clusters.

Take-Home Lesson: What is the right clustering algorithm to use for your data? There are many possibilities to consider, but your most important decisions are:

- What is the right distance function to use?
- What are you doing to properly normalize your variables?
- Do your clusters look sensible to you when appropriately visualized? Understand that clustering is never perfect, because the algorithm can't read your mind. But is it good enough?

10.6 War Story: Cluster Bombing

My host at the research labs of a major media/tech company during my sabbatical was Amanda Stent, the leader of their natural language processing (NLP) group. She is exceptionally efficient, excessively polite, and generally imperturbable. But with enough provocation she can get her dander up, and I heard the exasperation in her voice when she muttered "Product people!"

Part of her mission at the lab was to interface with company product groups which needed expertise in language technologies. The offenders here were with the news product, responsible for showing users recent articles of interest to them. The article clustering module was an important part of this effort, because it grouped together all articles written about the same story/event. Users did not want to read ten different articles about the same baseball game or Internet meme. Showing users repeated stories from a single article cluster proved highly annoying, and chased them away from our site.

But article clustering only helps when the clusters themselves are accurate.

"This is the third time they have come to me complaining about the clustering. They never give me specific examples of what is wrong, just complaints that the clusters are not good enough. They keep sending me links to postings they find on Stack Overflow about new clustering algorithms, and ask if we should be using these instead."

I agreed to talk to them for her.

First, I made sure that the product people understood that clustering is an ill-defined problem, and that no matter what algorithm they used, there were going to be occasional mistakes that they would have to live with. This didn't mean that there wasn't any room for improvement over their current clustering algorithm, but that they would have to temper any dreams of perfection.

Second, I told them that we could not hope to fix the problem until we were given clear examples of what exactly was going wrong. I asked them for twenty examples of article pairs which were co-clustered by the algorithm, but should not have been. And another twenty examples of article pairs which naturally belonged in the same cluster, yet this similarity was not recognized by the algorithm.

This had the desired effect. They readily agreed that my requests were sensible, and necessary to diagnose the problem. They told me they would get right on top of it. But this required work from their end, and everyone is busy with too many things. So I never heard back from them again, and was left to spend the rest of my sabbatical in a productive peace.

Months later, Amanda told me she had again spoken with the product people. Someone had discovered that their clustering module was only using the words from the headlines as features, and ignoring the entire contents of the actual article. There was nothing wrong with the algorithm per-se, only with the features, and it worked much better soon as it was given a richer feature set.

What are the morals of this tale? A man has got to know his limitations, and so does a clustering algorithm. Go to Google News right now, and carefully study the article clusters. If you have a discerning eye you will find several small errors, and maybe something really embarrassing. But the more amazing thing is how well this works in the big picture, that you can produce an informative, non-redundant news feed algorithmically from thousands of different sources. Effective clustering is never perfect, but can be immensely valuable.

The second moral is that feature engineering and distance functions matter in clustering much more than the specific algorithmic approach. Those product people dreamed of a high-powered algorithm which would solve all their problems, yet were only clustering on the headlines. Headlines are designed to attract attention, not explain the story. The best newspaper headlines in history, such as "Headless Body Found in Topless Bar" and "Ford to City, Drop Dead" would be impossible to link to more sober ledes associated with the same stories.

10.7 Chapter Notes

Distance computations are a basis of the field of computational geometry, the study of algorithms and data structures for manipulating point sets. Excellent introductions to computational geometry include [O'R01, dBvKOS00].

Samet [Sam06] is the best reference on kd-trees and other spatial data structures for nearest neighbor search. All major (and many minor) variants are developed in substantial detail. A shorter survey [Sam05] is also available. Indyk [Ind04] ably surveys recent results in approximate nearest neighbor search in high dimensions, based on random projection methods.

Graph theory is the study of the abstract properties of graphs, with West [Wes00] serving as an excellent introduction. Networks represent empirical connections between real-world entities for encoding information about them. Easley and Kleinberg [EK10] discuss the foundations of a science of networks in society.

Clustering, also known as *cluster analysis*, is a classical topic in statistics and computer science. Representative treatments include Everitt et al. [ELLS11] and James et al. [JWHT13].

10.8 Exercises

Distance Metrics

10-1. *[3]* Prove that Euclidean distance is in fact a metric.

10-2. *[5]* Prove that L_p distance is a metric, for all $p \geq 1$.

10-3. *[5]* Prove that dimension-weighted L_p distance is a metric, for all $p \geq 1$.

10-4. *[3]* Experiment with data to convince yourself that (a) cosine distance is not a true distance metric, and that (b) angular distance is a distance metric.

10-5. *[5]* Prove that edit distance on text strings defines a metric.

10-6. *[8]* Show that the expected distance between two points chosen uniformly and independently from a line of length 1 is 1/3. Establish convincing upper and lower bounds on this expected distance for partial credit.

Nearest Neighbor Classification

10-7. *[3]* What is the maximum number of nearest neighbors that a given point p can have in two dimensions, assuming the possibility of ties?

10-8. *[5]* Following up on the previous question, what is the maximum number of different points that can have a given point p as its nearest neighbor, again in two dimensions?

10-9. *[3]* Construct a two-class point set on $n \geq 10$ points in two dimensions, where every point would be misclassified according to its nearest neighbor.

10-10. *[5]* Repeat the previous question, but where we now classify each point according to its three nearest neighbors ($k = 3$).

10-11. *[5]* Suppose a two-class, $k = 1$ nearest-neighbor classifier is trained with at least three positive points and at least three negative points.

(a) Might it possible this classifier could label all new examples as positive?

(b) What if $k = 3$?

Networks

10-12. *[3]* Give explanations for what the nodes with the largest in-degree and out-degree might be in the following graphs:

(a) The telephone graph, where edge (x, y) means x calls y.

(b) The Twitter graph, where edge (x, y) means x follows y.

10-13. *[3]* Power law distributions on vertex degree in networks usually result from *preferential attachment*, a mechanism by which new edges are more likely to connect to nodes of high degree. For each of the following graphs, suggest what their vertex degree distribution is, and if they are power law distributed describe what the preferential attachment mechanism might be.

(a) Social networks like Facebook or Instagram.

(b) Sites on the World Wide Web (WWW).

(c) Road networks connecting cities.

(d) Product/customer networks like Amazon or Netflix.

10-14. *[5]* For each of the following graph-theoretic properties, give an example of a real-world network that satisfies the property, and a second network which does not.

(a) Directed vs. undirected.

(b) Weighted vs. unweighted.

(c) Simple vs. non-simple.

(d) Sparse vs. dense.

(e) Embedded vs. topological.

(f) Labeled vs. unlabeled.

10-15. *[3]* Prove that in any simple graph, there are always an even number of vertices with odd vertex degree.

10-16. *[3]* Implement a simple version of the PageRank algorithm, and test it on your favorite network. Which vertices get highlighted as most central?

Clustering

10-17. *[5]* For a data set with points at positions $(4, 10)$, $(7, 10)$ $(4, 8)$, $(6, 8)$, $(3, 4)$, $(2, 2)$, $(5, 2)$, $(9, 3)$, $(12, 3)$, $(11, 4)$, $(10, 5)$, and $(12, 6)$, show the clustering that results from

(a) Single-linkage clustering

(b) Average-linkage clustering

(c) Furthest-neighbor (complete linkage) clustering.

10-18. *[3]* For each of the following *The Quant Shop* prediction challenges, propose available data that might make it feasible to employ nearest-neighbor/analogical methods to the task:

(a) *Miss Universe.*

(b) *Movie gross.*

(c) *Baby weight.*

(d) *Art auction price.*

(e) *White Christmas.*

(f) *Football champions.*

(g) *Ghoul pool.*

(h) *Gold/oil prices.*

10-19. *[3]* Perform k-means clustering manually on the following points, for $k = 2$:

$$S = \{(1, 4), (1, 3), (0, 4), (5, 1), (6, 2), (4, 0)\}$$

Plot the points and the final clusters.

10-20. *[5]* Implement two versions of a simple k-means algorithm: one of which uses numerical centroids as centers, the other of which restricts centers to be input points from the data set. Then experiment. Which algorithm converges faster on average? Which algorithm produces clusterings with lower absolute and mean-squared error, and by how much?

10-21. *[5]* Suppose s_1 and s_2 are randomly selected subsets from a universal set with n items. What is the expected value of the Jaccard similarity $J(s_1, s_2)$?

10-22. *[5]* Identify a data set on entities where you have some sense of natural clusters which should emerge, be it on people, universities, companies, or movies. Cluster it by one or more algorithms, perhaps k-means and agglomerative clustering. Then evaluate the resulting clusters based on your knowledge of the domain. Did they do a good job? What things did it get wrong? Can you explain why the algorithm did not reconstruct what was in your head?

10-23. *[5]* Assume that we are trying to cluster $n = 10$ points in one dimension, where point p_i has a position of $x = i$. What is the agglomerative clustering tree for these points under

(a) Single-link clustering

(b) Average-link clustering

(c) Complete-link/furthest-neighbor clustering

10-24. *[5]* Assume that we are trying to cluster $n = 10$ points in one dimension, where point p_i has a position of $x = 2^i$. What is the agglomerative clustering tree for these points under

(a) Single-link clustering

(b) Average-link clustering

(c) Complete-link/furthest-neighbor clustering

Implementation Projects

10-25. *[5]* Do experiments studying the impact of merging criteria (single-link, centroid, average-link, furthest link) on the properties of the resulting cluster tree. Which leads to the tallest trees? The most balanced? How do their running times compare? Which method produces results most consistent with k-means clustering?

10-26. *[5]* Experiment with the performance of different algorithms/data structures for finding the nearest neighbor of a query point q among n points in d dimensions. What is the maximum d for which each method remains viable? How much faster are heuristic methods based on LSH than methods that guarantee the exact nearest neighbor, at what loss of accuracy?

Interview Questions

10-27. *[5]* What is curse of dimensionality? How does it affect distance and similarity measures?

10-28. *[5]* What is clustering? Describe an example algorithm that performs clustering. How can we know whether it produced decent clusters on our data set?

10-29. *[5]* How might we be able to estimate the right number of clusters to use with a given data set?

10-30. *[5]* What is the difference between unsupervised and supervised learning?

10-31. *[5]* How can you deal with correlated features in your data set by reducing the dimensionality of the data.

10-32. *[5]* Explain what a local optimum is. Why is it important in k-means clustering?

Kaggle Challenges

10-33. Which people are most influential in a given social network?
`https://www.kaggle.com/c/predict-who-is-more-influential-in-a-social-network`

10-34. Who is destined to become friends in an online social network?
`https://www.kaggle.com/c/socialNetwork`

10-35. Predict which product a consumer is most likely to buy.
`https://www.kaggle.com/c/coupon-purchase-prediction`