
Distance functions

One of the most basic and frequently-used tasks in data mining is the operation of *comparing* objects in order to assess how *dissimilar* (or *similar*) they are. For example, in recommendation systems, we want to find users who have similar taste in movies, so that we can recommend to the one movies that the other have liked.

Such a comparison of data objects is accomplished by *distance functions*. There are many different distance functions, and choosing which one to use for a particular problem setting depends on the application domain but also from the data type that we use to represent the data objects. Specific families of distance functions will be given below.

In abstract terms, let X be a space of data objects. A distance function has the type

$$d : X \times X \rightarrow \mathbb{R},$$

that is, pairs of objects in X are mapped to a real value. Intuitively, for two objects $x, y \in X$, the larger is the value of $d(x, y)$ the more dissimilar we consider the objects x and y to be.

Metrics. In most cases it is desirable to consider distance functions that satisfy certain properties. In particular, we consider intuitive properties that generalize the Euclidean distance between points in a geometric space. These are the following properties:

1. $d(x, y) \geq 0$ (non-negativity, or separation axiom)
2. $d(x, y) = 0$ if and only if $x = y$ (coincidence axiom)
3. $d(x, y) = d(y, x)$ (symmetry)
4. $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality).

A distance function d that satisfies all the above properties is called a *metric*. If d is a metric for X , the pair (X, d) , representing the object space X *equipped* with the metric d , is called a *metric space*.

Many algorithms for data-analysis tasks have been designed for general metric spaces. And many computational tasks can be performed more efficiently if the underlying distance function is a metric, instead of a function that does not satisfy the metric properties 1–4. For example, as we will say later in the course,

similarity search in metric spaces can be performed more efficiently than similarity search in non-metric spaces.

Distance functions vs. similarity functions. In our discussion we use the concepts of *distance* and *similarity* in an almost interchangeable manner. Although these two concepts aim to capture the same intuition, from a technical point-of-view they are inversely related.

A distance function $d : X \times X \rightarrow \mathbb{R}_+$ takes value 0 when two objects are identical, and it *increases* as the objects become less similar. In many cases a distance function is unbounded, i.e., it may take arbitrarily large values.

On the other hand, a similarity function, which is also defined to be of the type $s : X \times X \rightarrow \mathbb{R}_+$, is often assumed to take values in the range $[0, 1]$. In this case we assume that two identical objects have similarity value 1, and the function *decreases* as the objects become less similar. Two objects with similarity value 0 are assumed to be as dissimilar as possible.

Given a similarity function $s : X \times X \rightarrow [0, 1]$ we can define the *induced distance function* $d_s = 1 - s$, with the understanding that d_s also takes values in $[0, 1]$ and it assigns a distance value of 1 for objects that are as dissimilar as possible.

Inversely, given a distance function d that takes values in $[0, 1]$ we can induce a similarity function by $s_d = 1 - d$. If d take values larger than 1, but it is bounded, we can still induce a similarity function with appropriate *normalization*. In particular, if the function d takes values in $[0, D]$, where D is the largest possible distance between two objects, we can define the induced similarity function s_d by

$$s_d = 1 - \frac{d}{D}.$$

Another way to induce a similarity function s_d from a distance function d is by the exponential relation

$$s_d = e^{-d/\sigma^2},$$

where σ^2 is a parameter that controls the decay rate of the exponential: the larger the value of σ^2 the slower the decay. Note that in this case there is no need for d to be bounded, since s_d goes in the limit to 0, as d goes to infinity.

Next we discuss specific distance functions for different data representations.

L_p distance. We first consider the case that the underlying object space is the Euclidean space \mathbb{R}^m . An object in \mathbb{R}^m is an m -dimensional vector $\mathbf{x} = [x_1, \dots, x_m]$. Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ and a value p , the L_p distance of \mathbf{x} and \mathbf{y} is defined by

$$L_p(\mathbf{x}, \mathbf{y}) = L_p([x_1, \dots, x_m], [y_1, \dots, y_m]) = \left(\sum_{j=1}^m |x_j - y_j|^p \right)^{\frac{1}{p}}.$$

The L_p distance is also known as the *Minkowski distance*. It can be shown that for all values of p , $1 \leq p < \infty$, the L_p distance is a metric. For values $p < 1$ the L_p distance is *not* a metric.

The most common values of the parameter p and the corresponding instantiations of L_p distances are the following:

- $p = 1$ gives the L_1 distance, also known as the *Manhattan* distance or *city-block* distance.
- $p = 2$ gives the L_2 distance, known as the Euclidean distance.
- $p = \infty$ gives the L_∞ distance, known as the “*L-infinity*” distance.

It should be noted that

$$L_\infty(\mathbf{x}, \mathbf{y}) = \max_{j=1..m} |x_j - y_j|.$$

Hamming distance. We then consider distance functions for m -dimensional binary vectors, i.e., the underlying object space is the hypercube $\{0, 1\}^m$. As $\{0, 1\}^m \subseteq \mathbb{R}^m$ any L_p distance can be used. However, when all vector coordinates are 0–1, all L_p distances are equivalent (up to taking a p -th root), and thus we focus on the L_1 distance.

In the case of binary vectors the L_1 distance is also known as *Hamming* distance, and it simply counts the number of positions that two binary vectors differ. The distance is named after Richard Hamming, who introduced it in the context of *coding theory* and *information theory*, in order to quantify the error introduced in the transmission of a fixed-length binary vector through a communication channel Hamming (1950).

Sets. As we discussed earlier, one can represent sets using binary vectors. In particular, let X be a ground set of items, so that all sets we consider are subsets of X . Then, a set $S \subseteq X$ can be represented as a binary vector $\mathbf{x}(S)$ in $\{0, 1\}^{|X|}$, so that

$$x_i(S) = 1 \text{ if and only if } i \in S,$$

where $x_i(S)$ denotes the i -th coordinate of $\mathbf{x}(S)$. The vector $\mathbf{x}(S)$ is also known as the *indicator vector* of the set S .

Since sets and binary vectors are equivalent representations, we can measure distance between sets using any distance function designed for binary vectors, such as, the Hamming distance.

On the other hand, the Hamming distance treats 0’s and 1’s in a symmetric manner, meaning that the positions that the vectors agree are counted equally, independent on whether there is an agreement on a 1 bit or a 0 bit.

In most applications, however, the roles of 1 and 0 are not symmetric. 1 is typically used to indicate the presence of an item, and 0 is used to indicate absence. Furthermore, the ground set X is typically very large, while most of the observation sets S are fairly small compared to X , resulting in sparse data. In such a setting, an agreement on a 1 bit (presence of an item) is much more important than agreement on a 0 bit (absence of an item).

To take into account this asymmetry between 0’s and 1’s, a number of different measures for sets are commonly used. Most such measures are defined in terms of a similarity function, rather than a distance function, but as we have already

discussed, it is possible to obtain a distance function from a given similarity measure.

The most common set similarity functions are the following:

$$\text{cosine similarity} \quad \cos(A, B) = \frac{|A \cap B|}{\sqrt{|A|} \sqrt{|B|}} = \frac{\mathbf{x}(A) \cdot \mathbf{x}(B)}{\|\mathbf{x}(A)\| \|\mathbf{x}(B)\|},$$

$$\text{dot-product similarity} \quad \text{dot}(A, B) = |A \cap B| = \mathbf{x}(A) \cdot \mathbf{x}(B),$$

$$\text{Jaccard coefficient} \quad J(A, B) = \frac{|A \cap B|}{|A \cup B|},$$

for sets $A, B \subseteq X$, with corresponding indicator vectors $\mathbf{x}(A)$ and $\mathbf{x}(B)$, and where $|A|$ denotes the cardinality of a set A , $\|\mathbf{x}\|$ denotes the L_2 norm of a vector \mathbf{x} , and $\mathbf{x} \cdot \mathbf{y}$ denotes the dot-product of two vectors \mathbf{x} and \mathbf{y} .

Strings. As there is not a natural representation of strings by vectors, a new concept is needed in order to measure distances between strings. Consider the following example, which is meant to represent fragments of two DNA sequences.

```
g a t t a c a
a g a t t a c c
```

A first observation is that the lengths of these two strings is different, so we need a distance function that can handle strings with different lengths. A second observation is that if we compare the two strings character-by-character, according to the alignment shown above there are many differences, but if one shifts the first string by one position to the right, the number of different characters in the two strings decreases significantly.

These two observations, motivate the notion of *edit distance* between two strings, which is defined as the *minimum number of operations* required to transform one string into the other. We typically consider three types of operations

- *insertion*: one character is added in one string, at any position,
- *deletion*: one character is deleted from one string, at any position,
- *substitution*: one character in one string is substituted by another character.

In the above example, the edit distance between the two strings can be obtained by the following sequence of operations

```
g a t t a c a
```

add 'a' at the beginning of string

```
a g a t t a c a
```

substitute 'a' with 'c' at the end of string

```
a g a t t a c c
```

so the edit distance between the two strings is 2 (one addition and one substitution).

A common variant of the edit distance is the *longest common subsequence* distance (LCS), in which the only operations are addition and deletion. Another common extension is to assign different costs for the different operations. This can be done at a detailed level, where the operation costs depend on the characters that are added, deleted, or substituted. In this case, the edit distance is defined with respect to costs w_{ins} , w_{del} , and w_{sub} , where $w_{\text{ins}}(c)$ is the cost of adding character c to a string, $w_{\text{del}}(c)$ is the cost of deleting character c from a string, $w_{\text{sub}}(c, d)$ is the cost of substituting character c with character d .

Note that the edit distance is defined as the *minimum* number of operations required to transform one string into the other. Computing this number is not a trivial computational task. Fortunately, the computation can be performed in polynomial time by an algorithm that is based on *dynamic programming*.

Consider two strings x and y , of lengths n and m , respectively. We write $x[i]$ to refer to the i -th character of x , $1 \leq i \leq n$ and $x[i..j]$ to refer to the *substring* of x that starts at position i and ends at position j , $1 \leq i \leq j \leq n$.


To compute the edit distance we use a *dynamic-programming table* D of dimension $(n+1) \times (m+1)$. The $D[i, j]$ entry of the table records the edit distance of the strings $x[1..i]$ and $y[1..j]$, while the entries $D[i, 0]$ and $D[0, j]$, for $1 \leq i \leq n$ and $1 \leq j \leq m$, record the edit distance of the *empty string* with $x[1..i]$ and with $y[1..j]$, respectively.

The entries of the table D are computed using the following dynamic-programming equations:

$$\begin{aligned} D[0, 0] &= 0, \\ D[i, 0] &= D[i-1, 0] + w_{\text{del}}(x[i]), \quad \text{for } 1 \leq i \leq n, \\ D[0, j] &= D[0, j-1] + w_{\text{ins}}(y[j]), \quad \text{for } 1 \leq j \leq m, \\ D[i, j] &= \min \left\{ \begin{array}{l} D[i-1, j] + w_{\text{del}}(x[i]), \\ D[i, j-1] + w_{\text{ins}}(y[j]), \\ D[i-1, j-1] + w_{\text{sub}}(x[i], y[j]) \end{array} \right\}, \quad \text{for } 1 \leq i \leq n \text{ and } 1 \leq j \leq m. \end{aligned}$$

The table D can be computed in a row-wise order. Each entry of D can be computed in constant time, as we need to access at most three entries that have been computed previously, so the total time for edit-distance computation is $\mathcal{O}(nm)$.

The space required for the computation is also $\mathcal{O}(nm)$ (the size of table D), however, as we can fill the table D row by row, only the last two rows of D are needed, at any given time. By this observation we can reduce the space required by the algorithm to $\mathcal{O}(\min\{n, m\})$.

 **Note:** for time-series: euclidean distance or other L_p distance, normalization before applying the distance, distance on time-series subsequences, DTW, LCSS, EDR and ERP

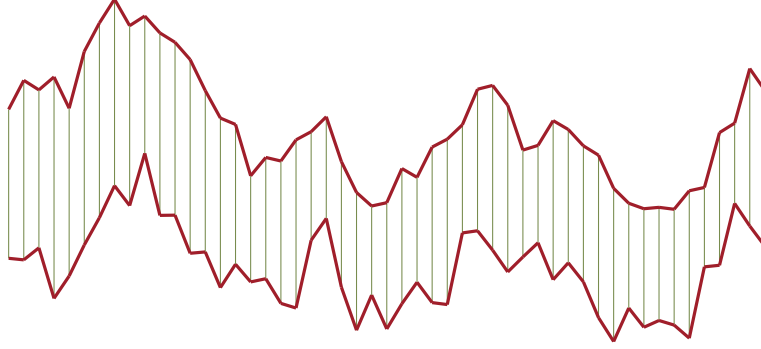


Figure 3.1 An example illustrating the definition of the Euclidean distance between two time-series.

Time-series. Consider that we want to compare two time-series $\mathbf{t} = \langle v_1, \dots, v_m \rangle$ and $\mathbf{r} = \langle u_1, \dots, u_m \rangle$, where we first assume that the two time-series have equal length m . A simple way to perform such a comparison is to ignore the temporal ordering and treat the two time-series as vectors of dimension m . Any L_p distance can then be applied. A popular choice is the Euclidean distance

$$L_2(\mathbf{t}, \mathbf{r}) = \left(\sum_{j=1}^m |v_j - u_j|^2 \right)^{\frac{1}{2}}.$$

An example illustrating the definition of the Euclidean distance between two time-series is shown in Figure 3.1.

Often we need to compare time-series of different units, or at different scales. For instance, imagine that for the purposes of a financial analysis one wants to compare inflation vs. unemployment over a certain time period. In this case we are more interested on the relative fluctuation of the two time-series rather than the absolute magnitude of their values. To perform such a comparison, we need to *normalize* the two time-series. Such a normalization can be done by *translating* and *scaling* each time-series so that they have *mean* equal to 0 and *standard deviation* equal to 1. If $\mathbf{t} = \langle v_1, \dots, v_m \rangle$ is the original time-series, then it is normalized to $\mathbf{t}' = \langle v'_1, \dots, v'_m \rangle$, where

$$v'_i = \frac{v_i - \mu}{\sigma}, \text{ with } \mu = \frac{1}{m} \sum_{j=1}^m v_j \text{ and } \sigma^2 = \frac{1}{m} \sum_{j=1}^m (v_j - \mu)^2.$$

One limitation of the Euclidean distance, is that it is required that the two time-series have the same length. However, the measure can be generalized easily to the case of non-equal length time-series, by considering the best time offset that one time-series (the shortest) becomes a subsequence of the other (the longest). More concretely, let $\mathbf{t} = \langle v_1, \dots, v_n \rangle$ and $\mathbf{r} = \langle u_1, \dots, u_m \rangle$ be the two time-series, and without loss of generality assume that $n > m$. The Euclidean distance between \mathbf{t}

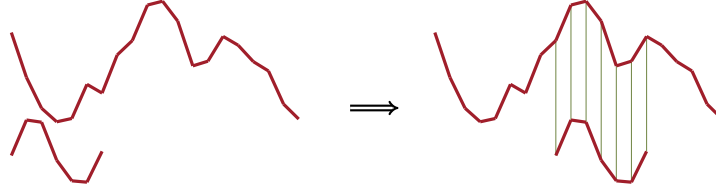


Figure 3.2 An example illustrating the definition of the Euclidean distance between two time-series of different length. On the left, the two time-series are aligned with respect to their left-most endpoint. On the right, the shortest time-series is translated to a position that gives the minimum distance with a matching-length subsequence of the longest time-series.

and \mathbf{r} can now be defined as

$$L_2(\mathbf{t}, \mathbf{r}) = \min_{k=0}^{n-m} \left(\sum_{j=1}^m |v_{j+k} - u_j|^2 \right)^{\frac{1}{2}}.$$

An example illustrating the definition of the Euclidean distance between two time-series of different length is shown in Figure 3.2.

The example in Figure 3.2 can also be used to illustrate one *limitation* of the Euclidean distance as a measure of comparing time-series. This is that while the measure provides a certain amount of flexibility, namely, normalizing the mean and variance or translating one time-series along the time-axis so that it best matches the other, it assumes that the two time-series vary with the same “speed.” In the Figure this is illustrated by the fact that all vertical lines that show the matching between the time points of the two time-series are *parallel*. However, in many applications it is meaningful to allow the time-series to vary with different speed. A good example is in the field of *speech analysis*, where a time-series represents the signal produced by a person’s speech. In this case, two people (or the same person in different situations) may speak at different speeds, or pronounce different phonemes at different speeds, or use different intonations.

The *dynamic-time warping* (DTW) distance has been proposed to address the issues discussed above and provide additional flexibility in terms of varying speed and locally varying magnitude of the time-series. The idea of the (DTW) distance is that it allows the time to be locally “warped,” i.e., advance with different speed.

More concretely, consider again two time-series $\mathbf{t} = \langle v_1, \dots, v_n \rangle$ and $\mathbf{r} = \langle u_1, \dots, u_m \rangle$, not necessarily of the same length. DTW tries to obtain an optimal matching between points of \mathbf{t} with points of \mathbf{r} . Consider for a moment that point v_i of \mathbf{t} should be matched with the point u_j of \mathbf{r} . Such a matching should contribute error $|v_i - u_j|^2$ in the overall DTW calculation. Furthermore, after matching v_i with u_j , there are three options:

- (i) time advances by one unit in \mathbf{t} and by one unit in \mathbf{r} , yielding a matching of points v_{i+1} with u_{j+1} ;

- (ii) time advances by one unit in \mathbf{t} but it does not advance in \mathbf{r} , yielding a matching of points v_{i+1} with u_j ;
- (iii) time advances by zero units in \mathbf{t} and by one unit in \mathbf{r} , yielding a matching of points v_i with u_{j+1} .

No other matching options are permitted. In particular, time *cannot* advance by *two or more units*, as this would leave some points of one of the two time-series unmatched, and also, time *cannot* advance by a *negative* increment, as this would imply that time moves backwards.

Furthermore, we require that v_1 should match u_1 and v_n should match u_m , so as to enforce that every point in each time-series is matched with at least one point in the other.

An example illustrating the definition of the dynamic-time warping (DTW) distance between two time-series is shown in Figure 3.3.

To compute the best matching between points of the two time-series, and thus, their DTW distance, we use the same dynamic-programming technique as with the edit distance. As before, we use a *dynamic-programming table* D of dimension $(n + 1) \times (m + 1)$. The $D[i, j]$ entry of the table records the DTW distance for matching the time-series $\mathbf{t}[1..i]$ and $\mathbf{r}[1..j]$, that is, up to points i and j , respectively. The entries of the table D are computed using dynamic-programming:

$$\begin{aligned}
 D[0, 0] &= 0, \\
 D[i, 0] &= +\infty, \quad \text{for } 1 \leq i \leq n, \\
 D[0, j] &= +\infty, \quad \text{for } 1 \leq j \leq m, \\
 D[i, j] &= |v_i - u_j|^2 + \min \left\{ \begin{array}{l} D[i - 1, j], \\ D[i, j - 1], \\ D[i - 1, j - 1] \end{array} \right\}, \quad \text{for } 1 \leq i \leq n \text{ and } 1 \leq j \leq m.
 \end{aligned}$$

The DTW between the two time-series \mathbf{t} and \mathbf{r} is then given by

$$\text{DTW}(\mathbf{t}, \mathbf{r}) = D[n, m]^{\frac{1}{2}}.$$

As with edit distance, each entry of D can be computed in constant time, implying that the overall running-time for computing the DTW distance is $\mathcal{O}(nm)$.

As defined above, the DTW distance permits the two time-series to be warped with arbitrarily different speeds. We sometimes want to avoid having such large differences in relative speed. One way to address this consideration is by adding a *locality constraint*. In this case, we require that two points v_i and u_j can be matched only if $|i - j| \leq w$, where w is a user-specified parameter. Let us call w -DTW the variant of DTW with such a locality constraint. The modification dynamic-programming algorithm presented above can be easily modified to compute the w -DTW distance.

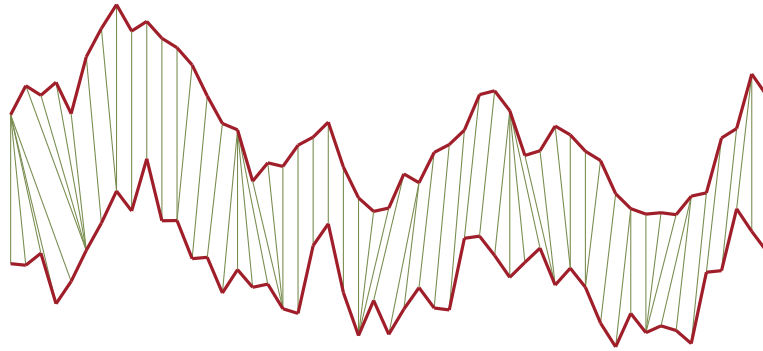


Figure 3.3 An example illustrating the definition of the dynamic-time warping (DTW) distance between two time-series. Time can be locally warped and advance with different speed in the two time-series.


Distance function embeddings

Part II

Clustering

The goal of clustering is to identify groups of similar objects (e.g., multidimensional points) and put them in the same cluster.

Although the clustering literature is rich, we will focus here on two types of clustering problems: (a) *partition-based clusterings* and (b) *hierarchical clusterings*. The output of a partition-based clustering is to *partition* a collection of objects into a (usually prespecified) number of clusters. A partition basically means that every object will belong to *exactly* one cluster. The output of a hierarchical clustering is a *nested* hierarchy of clusterings of the input collection. That is, if the input collection consists of n objects, then the output of the hierarchical clustering is a hierarchy of n clusterings such that the clustering of level i is a *generalization* of the clustering of level $(i + 1)$, where by generalization we mean that at level i two clusters of the clustering of level $(i + 1)$ were merged. Note that at level 1 we have one single cluster containing all objects and at level n we have the *singleton* clustering, where all objects are in a cluster by themselves.

 **Note:** There should be more that we could write for introduction to clustering. Mostly a lot of references :) –ET

Representatives of a metric space

Definition 5.1 (REPRESENTATIVE) Assume a set of objects $X = \{x_1, \dots, x_n\}$ from some universe U and a distance function $D : U \times U \rightarrow \mathbb{R}_0^+$. The *representative* of X with respect to D is the point $x^* \in U$ such that:

$$x^* = \arg \min_{x \in U} \sum_{i=1}^n D(x_i, x).$$

Proposition 5.2 Consider a set of points $X = \{x_1, \dots, x_n\}$, with $x_i \in U$ and a distance function $D : U \times U \rightarrow \mathbb{R}_0^+$. Now let

$$\hat{x} = \arg \min_{x \in X} \sum_{i=1}^n D(x_i, x).$$

If $D()$ is a metric, then it holds that:

$$\sum_{i=1}^n D(x_i, \hat{x}) \leq 2 \sum_{i=1}^n D(x_i, x^*),$$

where x^* is the representative of X .

Proof Assume $x' \in X$ such that $x' = \arg \min_{x \in X} D(x, x^*)$. That is, let x' be the point in X that is the closest to the representative x^* of X . Now for this point we have the following:

$$\sum_{i=1}^n D(x_i, \hat{x}) \leq \sum_{i=1}^n D(x_i, x'). \quad (5.1)$$

This is due to the fact that \hat{x} is the point in X that minimizes the sum of distances of all other points to it. Now we are ready to prove our claim as follows.

$$\sum_{i=1}^n D(x_i, \hat{x}) \leq \sum_{i=1}^n D(x_i, x') \quad (\text{from Equation (5.1)}) \quad (5.2)$$

$$\leq \sum_{i=1}^n [D(x', x^*) + D(x^*, x_i)] \quad (\text{from triangle inequality}) \quad (5.3)$$

$$= \sum_{i=1}^n D(x', x^*) + \sum_{i=1}^n D(x^*, x_i) \quad (5.4)$$

$$\leq \sum_{i=1}^n D(x_i, x^*) + \sum_{i=1}^n D(x^*, x_i) \quad (5.5)$$

$$= 2 \sum_{i=1}^n D(x_i, x^*). \quad (5.6)$$

□

Partition-based clustering

6.1 The k -Means clustering problem

In k -means, we assume that the distance between two points $x, y \in \mathbb{R}^d$ is given is the *squared-euclidean distance* between the points. That is, for the next of this section we will assume that:

$$D(x, y) = \|x - y\|_2^2.$$

An interesting fact about the representative of X with respect to the squared euclidean distance function is that the following:

Proposition 6.1 *For a set of points $X = \{x_1, \dots, x_n\}$ with $x_i \in \mathbb{R}^d$ and d being the squared euclidean distance, the representative x^* of X with respect to D is the mean of X . That is,*

$$x^* = \frac{1}{n} \sum_{i=1}^n x_i.$$

The proof of this proposition is left as an exercise.

Problem 6.2 (k -MEANS) Given a set of points $X = \{x_1, \dots, x_n\}$ with $x_i \in \mathbb{R}^d$ and an integer k , find a partition \mathbb{P} of X into k parts $\mathbb{P} = \{C_1, \dots, C_k\}$ such that if μ_i is the representative of C_i for $i = 1, \dots, k$, then

$$\text{kMeans-Cost}(\mathbb{P}) = \sum_{i=1}^k \sum_{x \in X_i} D(x, \mu_i) = \sum_{i=1}^k \sum_{x \in X_i} \|x - \mu_i\|_2^2 \quad (6.1)$$

is minimized.

Proposition 6.3 *The k -MEANS is NP-Complete.*

 **Note:** We need the reference for the above. *-ET*

6.2 Algorithms for the k -Means problem

We first start by establishing that although the k -MEANS problem is NP-Complete, when the input data points are 1-dimensional, then the problem can be solved optimally in polynomial time.

Proposition 6.4 *The k -MEANS problem is solvable in polynomial time when $d = 1$.*

We prove this proposition by designing a dynamic-programming algorithm for k -MEANS, when $d = 1$. In this case, of course the input points can be sorted, in time $\mathcal{O}(n \log n)$. Now assume that x_1, x_2, \dots, x_n is the sorted sequence of points, i.e., $x_i \leq x_{i+1}$ for $i \in \{1, \dots, n-1\}$. A key observation that we will exploit in the design of our algorithm is that for $d = 1$ the clusters will consist of consecutive points on this order.

We define $\text{UnitCost}(i, j)$, with $i \leq j$, to be the cost of representing all points $x_i, x_{i+1}, \dots, x_{j-1}, x_j$ with their representative

$$\mu_{ij} = \frac{1}{j - i + 1} \sum_{\ell=i}^j x_\ell.$$

That is,

$$\text{UnitCost}(i, j) = \sum_{\ell=i}^j D(x_\ell, \mu_{ij}) = \sum_{\ell=i}^j \|x_\ell - \mu_{ij}\|_2^2.$$

Note that one can naively compute $\text{UnitCost}(i, j)$ for all pairs $i \leq j$ in time $\mathcal{O}(n^3)$, as there are $\mathcal{O}(n^2)$ such pairs and linear time is required for each computation.

Now, if $C(i, k')$ is the cost of partitioning points x_1, x_2, \dots, x_i into k' clusters – computed as in Equation (6.1) – then, the optimal solution to the 1-dimensional k -MEANS problem is given by the following dynamic programming recursion:

$$C(i, k') = \min_{k'-1 \leq j \leq i-1} \{C(j, k' - 1) + \text{UnitCost}(j + 1, i)\} \quad (6.2)$$

When the values of $\text{UnitCost}()$ are pre-computed for all pairs, then evaluating Equation (6.2) takes time $\mathcal{O}(n^2 k)$. Taking the preprocessing steps into consideration, the total running time of the algorithm is $\mathcal{O}(n \log n + n^3 + n^2 k)$, i.e., $\mathcal{O}(n^3)$ and thus polynomial.

In the exercises, we ask you to modify this algorithm so that, with appropriate bookkeeping, its running time becomes $\mathcal{O}(n^2 k)$.

When $d > 1$, then the problem is NP-Complete and therefore we design polynomial approximation algorithms for solving it. The most popular algorithm for solving the k -MEANS is the **k-means** algorithm.

The k-means algorithm: **k-means** has been voted as one of the most popular algorithms in data mining. The pseudocode for the **k-means** is shown in Algorithm 1.

Algorithm 1 The pseudocode of the of the **k-means** algorithm.

Input: n points $X = \{x_1, \dots, x_n\}$, k , distance metric D .

Output: A partition of X into k clusters $\mathbb{P} = \{C_1, \dots, C_k\}$.

- 1: Pick a set of k cluster centers $\{c_1, \dots, c_k\}$
 - 2: **repeat**
 - 3: Let cluster $C_i = \{x \in X \mid i = \arg \min_{j=1\dots k} D(x, c_j)\}$
 - 4: **for** $i = 1, \dots, k$ **do**
 - 5: $c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$
 - 6: **until** Convergence
-

Advantages and disadvantages of k-means: One of the advantages of the **k-means** algorithm is that it finds a local optimum and it often (but not always) converges quickly. Thus, the algorithm is rather efficient and it is widely used in practice.

The main disadvantage of **k-means** is that the choice of the initial points can have a large influence in the results. Moreover **k-means** tends to find spherical clusters, it is very sensitive to outliers and clusters with different densities and sizes may confuse the algorithm.

One solution to the instability caused by the random initialization has on the output of **k-means** is to repeat the execution of the algorithm many times for different random initial points. Although such an approach may help, the solution may still be bad. Another solution that has been repeatedly shown to work well in practice is the one that picks the initial points c_1, c_2, \dots, c_k to be as distant to each other as possible. This idea has been formalized in the **k-means++** algorithm. The main advantage of this algorithm, first proposed by Arthur and Vassilvitskii Arthur and Vassilvitskii (2007), is that it comes with provable approximation guarantees.

The k-means++ algorithm: The only difference between **k-means++** and the classical **k-means** described in Algorithm 1 is in the initialization phase. The **k-means++** algorithm chooses the first center uniformly at random. Then, it chooses the next center with probability proportional to $D^2(x)$, where $D(x)$ is the distance between point x and the nearest center selected so far.¹ The iteration phase of the **k-means++** is identical to the iteration phase of **k-means** shown in Algorithm 1. The main result proved by Arthur and Vassilvitskii is summarized in the following theorem.

Theorem 6.5 (Arthur and Vassilvitskii (2007)) *The **k-means++** is an $\mathcal{O}(\log k)$ -approximation algorithm in expectation.*

Although the full proof of this result can be found in the original paper, we highlight here the main steps of their proof, so that we develop the appropriate intuition. First, we note that the approximation guarantee expressed in the

¹ Recall that for **k-MEANS** the distance between two points is measured using the squared L_2 -norm of the difference between the points.

above theorem comes from the first iteration (i.e., the initialization). Subsequent iterations can only improve the cost.

On a high level, the proof of Arthur and Vassilvitskii proceeds in two steps. In the first step, they show that if \mathbb{P}^* is the optimal partition of the input points for the k -MEANS and \mathbb{P} is the clustering output by **k-means++**, then if the initialization step of **k-means++** picks one point from every one of the optimal clusters then the expected cost of \mathbb{P} is at most 8 times the cost of \mathbb{P}^* . That is,

$$\text{kMeans-Cost}(\mathbb{P}) \leq 8\text{kMeans-Cost}(\mathbb{P}^*).$$

In the second step of the proof, they observe that if no points from a cluster in \mathbb{P}^* are picked, then this cluster probably does not contribute too much in the overall error. Using these two observations, they then construct an inductive proof that shows the $\mathcal{O}(\log k)$ -factor approximation.

6.3 The k -Median problem

In the k -MEDIAN, we assume that the distance between two points $x, y \in \mathbb{R}^d$ is given by any distance function $D()$ that is a metric; for example, the L_1 or L_2 norms of the differences between vectors x and y are the most common choices for such distance functions. Given this, the k -MEDIAN is defined as follows.

Problem 6.6 (k -MEDIAN) Given a set of points $X = \{x_1, \dots, x_n\}$ with $x_i \in \mathbb{R}^d$ and an integer k , find a partition \mathbb{P} of X into k parts $\mathbb{P} = \{C_1, \dots, C_k\}$ such that if μ_i is the representative of C_i for $i = 1, \dots, k$, then

$$\text{kMedian-Cost}(\mathbb{P}) = \sum_{i=1}^k \sum_{x \in X_i} D(x, \mu_i) \quad (6.3)$$

is minimized.

As we discussed above, instantiations of $D(x, \mu_i)$ can be: (a) $D(x, \mu_i) = L_1(x, \mu_i)$ or (b) $D(x, \mu_i) = L_2(x, \mu_i)$.

Proposition 6.7 *The k -MEDIAN is NP-Complete.*

 **Note:** Find the original citation for this –ET

6.4 Algorithms for the k -Median problem

As with the k -MEANS, when the data are 1-dimensional the k -MEDIAN can be solved optimally in polynomial time. Thus, we have the following proposition, the proof of which we leave as an exercise.

Proposition 6.8 *The k -MEDIAN problem is solvable in polynomial time when $d = 1$.*

When $d > 1$, the k -MEDIAN problem is NP-Complete and therefore we devise polynomial-time approximation algorithms or heuristics. Below, we describe Local- k Median approximation-algorithm and a very popular heuristic known as the k -centroids.

The Local- k Median algorithm: There are multiple variants of the local-search approximation algorithm we present here in the literature Arya et al. (2004); Charikar and Guha (2005); Charikar et al. (2002). We present here the solution by Arya et al. Arya et al. (2004), which gives a constant-approximation factor. The algorithm is rather easy to describe and implement. In this paragraph we will only state the approximation factors of the algorithm and we will direct the interested reader to the original paper for the analysis.

The high-level idea of the Local- k Median algorithm is given by the pseudocode shown in Algorithm 2.

Algorithm 2 The high-level pseudocode of the Local- k Median algorithm.

Input: n points $X = \{x_1, \dots, x_n\}$, k , distance metric D .

Output: A partition of X into k clusters $\mathbb{P} = \{C_1, \dots, C_k\}$.

- 1: Pick a random set of k cluster centers $S = \{c_1, \dots, c_k\}$ such that $c_i \in X$
 - 2: \mathbb{P}_S : partition induced by assigning every x_i to its closest point in S .
 - 3: **repeat**
 - 4: Find S' “similar” to S
 - 5: **if** $k\text{Median-Cost}(\mathbb{P}_{S'}) < k\text{Median-Cost}(\mathbb{P}_S)$ **then**
 - 6: $S = S'$
 - 7: **until** Convergence
-

Initially, a random set of k points from X are picked as the cluster centers; every other point is assigned to its closest center. Therefore, for a fixed set of centers S , we define as \mathbb{P}_S to be the partition induced by assigning every point in X to its closest center in S . If this new S' induces a partition with smaller $k\text{Median-Cost}$, then it is adopted and the algorithm proceeds until convergence. The notion of closeness between S and S' is defined in terms of swaps of points from $X \setminus S$ to S and vice versa. We make this notion clear below.

Local- k Median with single swaps: A swap occurs in S if a center from $s \in S$ is evicted and a new center $s' \in X \setminus S$ takes its place. Therefore, the Local- k Median with a single swap simply moves from S to S' by performing a single random swap that leads to a partition with smaller cost. Arya et al. (2004) have the following result:

Proposition 6.9 (Arya et al. (2004)) *If \mathbb{P}_S is partition output by the Local- k Median with single swaps and \mathbb{P}_{S^*} the optimal partition for the k -MEDIAN problem, then*

$$k\text{Median-Cost}(\mathbb{P}_S) \leq 5 \times k\text{Median-Cost}(\mathbb{P}_{S^*}).$$

Local- k Median with p -swaps: The single-swap algorithm can be generalized by considering multiple swaps simultaneously. In this case, at most p centers can

be chosen to be removed from S and p new centers will be added to replace them. Again, Arya et al. (2004) prove the following result:

Proposition 6.10 (Arya et al. (2004)) *If \mathbb{P}_{S_p} is partition output by the **Local- k Median** with p swaps and \mathbb{P}_{S^*} the optimal partition for the k -MEDIAN problem, then*

$$k\text{Median-Cost}(\mathbb{P}_{S_p}) \leq (3 + 2/p) \times k\text{Median-Cost}(\mathbb{P}_{S^*}).$$

The k -centroids algorithm: k -centroids is a variant of the k -means algorithm shown in Algorithm 1. In fact, k -centroids can be obtained by k -means if we make two changes in the latter. First, in line 1 instead of assigning random centers to the k clusters we assign as centers points already in the dataset; this can be implemented either by selecting k random points from X , or – even better – by using the initialization method used by k -means++. The second change is in line 5; there instead of computing the representative c_i of cluster C_i by taking the average of all the points assigned to this cluster, we now find c_i as:

$$c_i = \arg \min_x \sum_{y \in C_i} D(x, y). \quad (6.4)$$

In the cases where the cluster representative cannot be computed efficiently in polynomial time, then we find c_i as:

$$c_i = \arg \min_{x \in C_i} \sum_{y \in C_i} D(x, y).$$

That is, the cluster center c_i is a point from X such that $c_i \in C_i$ and it also minimizes the sum of the distances to every other point in C_i .

From the computational point of view, the k -centroids algorithm is much more expensive than k -means. The reason for that is that finding the centroid c_i of *every* cluster C_i in *every* iteration using Equation (6.4) induces at least an extra factor of $\mathcal{O}(n)$ in the running time of the algorithm. Therefore, the k -centroids algorithm is not very efficient in practice.

6.5 The k -Center problem

The k -CENTER, which we define below, is defined for any distance function $D()$. However, many of the proofs of the approximations of the algorithms are going to be restricted to cases where $D()$ is a metric.

Problem 6.11 (k -CENTER) Given a set of points $X = \{x_1, \dots, x_n\}$ with $x_i \in \mathbb{R}^d$ and an integer k , find a partition \mathbb{P} of X into k parts $\mathbb{P} = \{C_1, \dots, C_k\}$ such that

$$k\text{Center-Cost}(\mathbb{P}) = \max_{i \in \{1, \dots, k\}} \max_{x, x' \in C_i} D(x, x')$$

is minimized.

Proposition 6.12 *The k -CENTER is NP-Complete.*

 **Note:** We need the reference for the above. –ET

Proposition 6.13 *The k -CENTER problem is solvable in polynomial time when $d = 1$.*

Again we prove this proposition, rather informally, by giving a polynomial-time algorithm for the k -CENTER when $d = 1$. As we did for the corresponding case in the k -MEANS, our algorithm first sorts the points in increasing order. Time $\mathcal{O}(n \log n)$ is required for that. Now let x_1, x_2, \dots, x_n be the sorted sequence of 1-dimensional points. The observation we made for the k -MEANS also holds for the k -CENTER problem with $d = 1$. That is, in this case as well the clusters will consist of consecutive points on this order.

Now we start by observing the following fact: Assume an oracle that tells us that the cost of the optimal solution to the k -CENTER problem for this instance is equal to τ . With this knowledge, we are able to actually *find* the optimal solution as follows: start with point x_1 and traverse the points in increasing order until you meet the first point x_i for which $D(x_1, x_i) > \tau$; in that case set all points x_1, \dots, x_{i-1} in the same cluster. Start a new cluster at point x_i and proceed in a similar fashion to form the second cluster and so on. The result of this process is a partition of the points into ℓ clusters $C_1 \dots C_\ell$ such that

$$\text{kCenter-Cost}(C_1, \dots, C_\ell) \leq \tau.$$

However, this algorithm does not guarantee to us that the number of clusters ℓ will necessarily be equal to the required number of clusters k .

The second observation that will lead us to our algorithm is that there is an anti-monotonic relationship between τ and ℓ . That is, as τ increases the number of clusters reported by the above procedure decreases and vice versa.

The final observation is that there are at most $\mathcal{O}(n^2)$ distinct values that τ can take, as there are at most $\mathcal{O}(n^2)$ clusters consisting of consecutive points in our data.

Given the above observations, we design the following algorithm: First sort all possible $\mathcal{O}(n^2)$ distinct values of τ defined by $D(x_i, x_j)$ for all $i \in \{1, \dots, n\}$ and $j \in \{x_{i+1}, x_n\}$ in increasing order. This step will require time $\mathcal{O}(n^2 \log n)$.

Given this, we can now perform a binary search on the values of τ , by applying the process we described above for each such value and reporting the smallest value of τ for which the number of clusters ℓ formed by the above procedure is equal to the required number of clusters k . The running time of this binary search involves running the linear-time procedure we described above for any fixed value of τ ; there are going to be $\mathcal{O}(\log n^2) = \mathcal{O}(\log n)$ possible values of τ to be tried leading to a total running time (excluding the sorting step) of $\mathcal{O}(n \log n)$.

6.5.1 The Farthest algorithm

The idea of the **Farthest** algorithm (described by Gonzalez Gonzalez (1985) and Hochbaum and Smoys Hochbaum and Shmoys (1985)) is the following: start with a random point and put it in set S . Then choose the point that is the furthest from S , add it to S and proceed similarly until the cardinality of set S is equal

to k . Note that we use the following definition of the distance of a point to a set: $D(x, S) = \min_{y \in S} D(x, y)$; i.e., the distance of a point to a set is the minimum distance between the point and any element from the set. The **Farthest** algorithm considers the points in S as the “centers” of the k clusters. Every other point $x \in X \setminus S$ is assigned to the cluster with the closest center to x .

Algorithm 3 The pseudocode of the of the **Farthest** algorithm.

Input: n points $X = \{x_1, \dots, x_n\}$, k , distance metric D .
Output: A partition of X into k clusters $\mathbb{P} = \{C_1, \dots, C_k\}$.

- 1: Pick a random point from X and label it as point 1
- 2: **for** $i = 2, \dots, k$ **do**
- 3: Find a point $x = \arg \max_{x' \in X \setminus S} D(x', S)$ and label it i
- 4: $S = S \cup \{x\}$
- 5: **for** $x \in X \setminus S$ **do**
- 6: $\pi(x) = \arg \min_{i=1, \dots, k} D(x, i)$
- 7: $C_{\pi(x)} = C_{\pi(x)} \cup \{x\}$

In order to analyze the **Farthest** algorithm we need to introduce a procedure of traversing the data points in X , which Hochbaum and Smoys Hochbaum and Shmoys (1985) call the *furthest traversal* procedure shown in Algorithm ??.

Algorithm 4 The pseudocode of the of the **Farthest-Traversal** procedure.

Input: n points $X = \{x_1, \dots, x_n\}$, distance metric D .
Output: A labeling of the n data points and an assignment π of every point to a parent.

- 1: Pick a random point from X and label it as point 1
- 2: **for** $i = 2, \dots, n$ **do**
- 3: Find an unlabeled point furthest from $\{1, 2, \dots, i-1\}$, and label it as i ;
 use the standard distance from a point s to a set S : $D(x, S) = \min_{y \in S} D(x, y)$.
- 4: Let $\pi(i) = \arg \min_{j < i} D(i, j)$
- 5: Let $R_i = D(i, \pi(i))$

Starting with n points in a metric space, the **Farthest-Traversal** numbers the n points as follows: for each point i , the closest neighbor of i among points $1, \dots, i-1$ is described as the *parent* of i , $\pi(i)$. Then, R_i is the distance between point i and its parent. That is:

$$R_i = D(i, \pi(i)) = D(i, \{1, 2, \dots, i-1\}).$$

Note that the **Farthest** algorithm uses points $1, 2, \dots, k$ as centers for the k -clustering it reports. For the rest of the discussion we refer to this clustering as \mathbb{P}_k .

Lemma 6.14 *Let's adopt the convention that $R_1 = \infty$ and $R_{n+1} = 0$. Then we have the following:*

1. For all i , $\pi(i) < i$
2. $R_1 \geq R_2 \geq R_3 \geq \dots \geq R_n$.
3. For all k , $k\text{Center-Cost}(\mathbb{P}_k) = R_{k+1}$.

Proof Due to the way that **Farthest-Traversal** operates we know that for any i and for all $j > i$,

$$\begin{aligned}
 R_j &= d(j, \pi(i)) \\
 &= d(j, \{1, 2, \dots, j-1\}) \\
 &\leq d(j, \{1, 2, \dots, i-1\}) \\
 &\leq d(i, \{1, 2, \dots, i-1\}) \\
 &= R_i.
 \end{aligned}$$

This immediately gives us that $R_1 \geq R_2 \geq R_3 \geq \dots \geq R_n$. In order to see that for all k , $k\text{Center-Cost}(\mathbb{P}_k) = R_{k+1}$, we notice that for all $i > k$,

$$d(i, \{1, 2, \dots, k\}) \leq d(k+1, \{1, 2, \dots, k\}) = R_{k+1}.$$

□

The above lemma allows us now to prove the following:

Proposition 6.15 *The **Farthest** algorithm is a 2-approximation algorithm for the k -CENTER problem.*

Proof Let S_1, S_2, \dots, S_k denote the clusters of the optimal k -clustering. Suppose that one of these clusters contains two or more of the point $\{1, 2, \dots, k\}$ – which are used as centers in the **Farthest** algorithm. These points are at distance at least R_k from each other. Therefore, the clusters must have radius at least $\frac{1}{2}R_k \geq \frac{1}{2}R_{k+1} = \frac{1}{2}k\text{Center-Cost}(\mathbb{P}_k)$.

On the other hand, if each optimal cluster S_j contains exactly one of the points $\{1, \dots, k\}$, then (for any j) every point in S_j is within $2 \times \text{radius}(S_j)$ of $\{1, \dots, k\}$. Therefore,

$$k\text{Center-Cost}(\mathbb{P}_k) = R_{k+1} = d(k+1, \{1, \dots, k\}) \leq 2 \times \max_j \text{radius}(S_j).$$

□