CS-E4600 — Lecture notes #2
Distance functions

Aristides Gionis

September 16, 2018

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

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

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 $d : X \times X \to \mathbb{R}_+$ 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 $s : X \times X \to \mathbb{R}_+$, 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 $s : X \times X \to [0, 1]$ 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 $\mathbf{x} = [x_1, \ldots, x_m]$. 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

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

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 [1].

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 $S \in X$ 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:

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

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

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

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 c.

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.

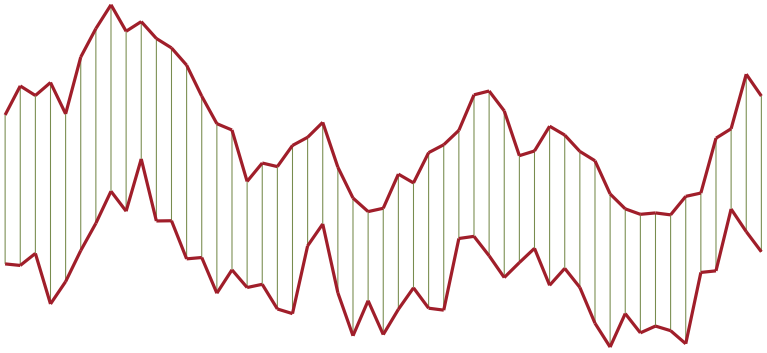

Figure 1: An example illustrating the definition of the Euclidean distance between two time-series.

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

$$D[0,0] = 0,$$

$$D[i,0] = D[i-1,0] + w_{\text{del}}(x[k]), \quad \text{for } 1 \le i \le n,$$

$$D[0,j] = D[0,j-1] + w_{\text{ins}}(y[k]), \quad \text{for } j \le i \le m,$$

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

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 $\mathcal{O}(\min\{n,m\})$.

Time-series. Consider that we want to compare two time-series $\mathbf{t} = \langle v_1, \ldots, v_m \rangle$ and $\mathbf{r} = \langle u_1, \ldots, u_m \rangle$, 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 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*

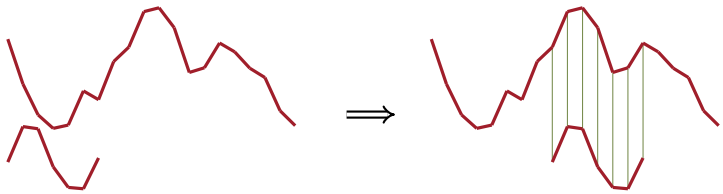

Figure 2: An example illustrating the definition of the Euclidean distance between two time-series of different length. On the left, the two time-series are aligned with respect to their left-most endpoint. On the right, the shortest time-series is translated to a position that gives the minimum distance with a matching-length subsequence of the longest time-series.

each time-series so that they have *mean* equal to 0 and *standard deviation* equal to 1. If $\mathbf{t} = \langle v_1, \ldots, v_m \rangle$ is the original time-series, then it is normalized to $\mathbf{t}' = \langle v'_1, \ldots, v'_m \rangle$, 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 $\mathbf{t} = \langle v_1, \ldots, v_n \rangle$ and $\mathbf{r} = \langle u_1, \ldots, u_m \rangle$ be the two time-series, and without loss of generality assume that $n > m$. The Euclidean distance between $\mathbf{t}$ 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 2.

The example in Figure 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 $\mathbf{t} = \langle v_1, \ldots, v_n \rangle$ and $\mathbf{r} = \langle u_1, \ldots, u_m \rangle$, 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

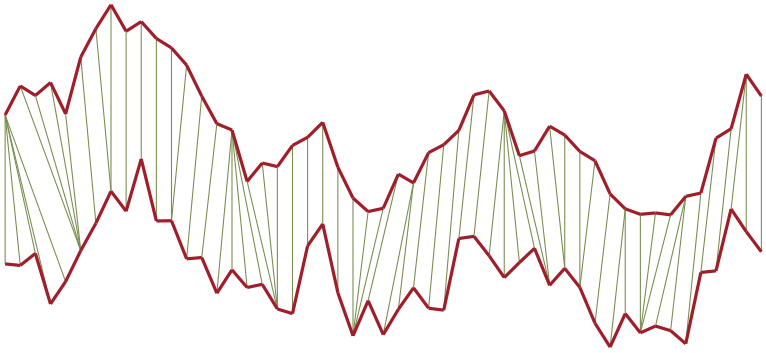

Figure 3: An example illustrating the definition of the dynamic-time warping (DTW) distance between two time-series. Time can be locally warped and advance with different speed in the two time-series.

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.

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:

$$D[0,0] = 0,$$

$$D[i,0] = +\infty, \quad \text{for } 1 \le i \le n,$$

$$D[0,j] = +\infty, \quad \text{for } 1 \le j \le m,$$

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

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

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

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.

Point sets. Next we discuss distance functions for comparing *point sets*, which we sometimes call *point clouds*. First we assume that single points are represented in some underlying metric space X, and distances between pairs of points are evaluated using a distance function $d : X \times X \to \mathbb{R}$. We assume that (X, d) is a metric space. For simplicity, we may think that X is the m-dimension Euclidean space $\mathbb{R}^m$, and the distance function d is the Euclidean L_2 norm, but this is not really necessary. Our definitions hold for point sets in any metric space (X, d).

A point set $A = \{a_1, \ldots, a_k\}$ is a subset of X containing k points. Two point sets A and B may contain a different number of points.

One possible way to define a distance between point sets is via the *Hausdorff distance*. This distance function has been proposed in the area of pattern recognition for comparing point distributions in images [2], and it has applications in computer vision and computer graphics, as well as in more theoretical areas such as computational geometry.

Given two point sets A and B, the Hausdorff distance from A to B is defined as

$$d_H(A, B) = \max_{a \in A} \min_{b \in B} d(a, b).$$

In other words, for every point in A we consider its closest point in B, and among all these closest-point distances we take the largest. To motivate this definition imagine, for example, that we want to examine whether cinemas and shopping malls are distributed in a similar manner in a city. For a given city we represent all its cinemas as a point set, and all its shopping malls as another point set, and we compute the worst-case distance from each cinema to its nearest shopping mall.

One can observe that the distance function d_H is not symmetric and it does not satisfy the identity property. Indeed, in the previous example, consider a city in which all cinemas are located in some shopping mall, while there are some shopping malls without a cinema. In that case, the Hausdorff distance from cinemas to shopping malls would be 0, while the Hausdorff distance from shopping malls to cinemas would be strictly greater than zero.

These drawbacks can be resolved by defining a symmetric version of Hausdorff distance:

$$D_H(A, B) = \max\{d_H(A, B), d_H(B, A)\}.$$

It can be shown that the symmetric Hausdorff distance is a metric (proof left as exercise).

During the computation of the Hausdorff distance each point in the first point set is "matched" to some point in the second point set. The resulting matching is not one-to-one, as two distinct points in the first point set may have the same nearest neighbor in the second point set. In some applications, however, it is desirable to have distances that correspond to a one-to-one matching. In pattern recognition, for example, the sets of points under consideration may represent different features of geometric objects, and to compare two geometric objects we need to match all their features one to one.

To address this issue we define an alternative distance function between point sets based on matchings, which we call *bottleneck* distance. The drawback is that the bottleneck distance is defined over point sets that have the same number of points.

Given two point sets $A = \{a_1, \ldots, a_k\}$ and $B = \{b_1, \ldots, b_k\}$ with k points each, we define S_k to be the set of *all permutations* of k elements. Recall that a permutation is a *bijection* (a function that is *one-to-one* and *onto*) of the set $\{1, 2, \ldots, k\}$ to itself. We use symbols like σ or π to denote permutations in S_k. The bottleneck distance $d_M(A, B)$ of the two k-sets A and B is then defined as

$$d_M(A, B) = \min_{\sigma \in S_k} \max_{1 \leq i \leq k} d(a_i, b_{\sigma(i)}).$$

One can again show that the bottleneck distance is a metric (proof left as exercise).

Exercises

1. Prove or disprove: the string edit distance is a metric.

2. Prove or disprove: the DTW distance is a metric.

3. Show that the Hausdorff and the bottleneck distance functions are metrics.

4. Provide efficient algorithms to compute the Hausdorff and bottleneck distance functions. Analyze the running time of the proposed algorithms.

References

[1] R. W. Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[2] D. P. Huttenlocher, G. A. Klanderman, and W. J. Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on pattern analysis and machine intelligence*, 15(9):850–863, 1993.