Metric embeddings and similarities

In this note, we will talk about mappings from a latent space (e.g. raw data) to a domain which respect a notion of distance (or similarity). There are several different settings, but we try to take a unified point of view. These differences come down to the level of measurement https://en.wikipedia.org/wiki/Level_of_measurement : ordinal if we can only distinguish between same versus different, nominal if we have a numerical notion of distance.

Multi-Dimensional Scaling https://en.wikipedia.org/wiki/Multidimensional_scaling

Is another approach to embedding, it is an example of nonlinear dimensionality reduction https://en.wikipedia.org/wiki/Nonlinear_dimensionality_reduction We are given the matrix of distances $d(x_i, x_j)$ between points, but not the points, and we seek to find an imbedding of the points which respects the distance. Note that if we had the imbedding, given by the matrix X whose rows are the points, then there is a connection with the Singular Value Decomposition of the symmetric positive definite matrix $D_{ij} = d(x_i, x_j)^2$

PCA and Kernel PCA

Principal Component Analysis take a covariance matrix $C = \frac{1}{m}\sum_i x_i x_i^T$ and computes the projection onto the first k eigenvectors. kernel PCA.[31] It is a combination of Principal component analysis and the kernel trick, KPCA begins by computing the covariance matrix of the data after being transformed into a higher-dimensional space $C = \frac{1}{m}\sum_i \Phi(x_i)\Phi(x_i)^T$

It then projects the transformed data onto the first *k* eigenvectors of that matrix, just like PCA.

General nonlinear maps

Suppose, instead of using linear maps, PCA, or kernels, we want to find a nonlinear mapping (e.g. neural network), to find the embedding. Now we have issues: (i) no longer have a formula for the map (ii) need to find a loss to train, (iii) want to make sure that it doesn't overfit. Let's talk about (i) and (ii) first.

We can ignore the parameters, and just see what the map, $\Phi : \mathcal{X} \to \mathbb{R}^d$ should satisfy.

1. (distance) The first thing is that it should (approximately) respect the distance.

   $(1-\epsilon)d(x,y)^2 \leq d(\Phi(x),\Phi(y))^2 \leq (1+\epsilon)d(x,y)^2$

   The middle term is the Euclidean distance

2. (distance) We can train it using pairs of data x, y and asking that $d_X = d(x,y)$ be close to $d_{\mathcal{F}} = d(\Phi(x),\Phi(y))^2$. So for example a loss of the form

   $\sum_{x_i,x_j} L(d(x,y), d(\Phi(x),\Phi(y)))$

where L is your favorite loss which makes the two terms equal. For example, we could have $\sum_{x_i,x_j}(d(x,y)-d(\Phi(x),\Phi(y)))^2$. In this case, if we think the numbers are accurate, the error is measured in relative terms (i.e percentage error), so a relative loss might be useful.

However, we may not really have $d(x,y)$, instead we may have a cost or discrepancy: a function which is increasing in distance, but may not scale correctly. This is the case where we are basing our distance on some kind of cost or discrepancy: higher cost means bigger distance, but not proportionally.

$$c(x,y)\le c(x_2,y_2) \implies d(\Phi(x),\Phi(y))^2 \le d(\Phi(x_1),\Phi(y_2))^2$$

In the extreme case, we just have similar and different examples (nominal). In this case, it makes sense to normalize the points, as follows.

Similarity versus distance

Look for a mapping f from data to feature space. The feature space can be $\mathbb{R}^d$, but it is sometimes convenient to normalize the vector to have unit length.

$$f:\mathcal{X}\to F=S^d=\{x\in\mathbb{R}^d \mid \|x\|_2=1\}$$

In this case, the distance in feature space simplifies, since the vectors are unit norm.

$$d^2(x,y)=\|x-y\|^2=x^2-2x\cdot y+y^2=1-2x\cdot y$$

The corresponding similarity is $s(x,y)=x\cdot y$, so $d(x,y)^2=1-2*s(x,y)$.

Now similar points have $s(x,y)\approx 1$ and dissimilar points have $s(x,y)\approx -1$

What is the loss for similarity?

Now, given any pair of points, x,y we just have an ordinal ± 1 for similar or dissimilar. So we train the similarity to be ± 1 accordingly. Make $\Phi(x)$ a unit vector and the loss for x,y to make $\Phi(x)\cdot\Phi(y)=sim(x,y)=\pm 1$. Thus summing over all pairs of points, we have

$$\frac{1}{C_1}\sum_{x\sim y}\|\Phi(x)-\Phi(y)\|^2-\frac{1}{C_2}\sum_{x\nsim y}\|\Phi(x)-\Phi(y)\|^2$$

where we also nomalize over the number of pairs in each case.

What about the multiclass version

We can write the multiclass version of the that loss, but summing the loss above, but again over each class.

$$\sum_{classes}\left(\frac{1}{C_1}\sum_{x\sim y}\|\Phi(x)-\Phi(y)\|^2-\frac{1}{C_2}\sum_{x\nsim y}\|\Phi(x)-\Phi(y)\|^2\right)$$

What if you want to classify ?

Once you have a good embedding, how do you use it to classify? In this case, we simply want to convert the distances to score functions (as in usual classification). Now the score functions are over all pairs of points (or pair in a minibatch) $x_i x_j$. so the score are the similarity $\Phi(x_i) \cdot \Phi(x_j)$. We will omit the Φ in this section. Then we do softmax to convert scores to probabilities:

$$softmax(x_k, x_l) = \frac{\exp(x_k x_l)}{\sum_{i.j} \exp(x_i x_j)}$$

In the special case, where we have x_1 is similar to x_2, and the rest are different (as is the case for contrastive learning), we get the contrastive learning loss

$$L(x_1, x_2, x_3, \ldots, x_N) = -\log\left(\frac{\exp(x_1 x_2)}{\sum_{i.j} \exp(x_i x_j)}\right)$$

Summary : Interpretation of contrastive loss

I think the right way to look at it:

- The real loss for the imbedding, when you have distances, is

$\sum_{x_i, x_j} (d(x,y) - d(\Phi(x), \Phi(y)))^2$

- if we only have similarities, then the loss is the one with signs above. $\frac{1}{C_1} \sum_{x \sim y} \|\Phi(x) - \Phi(y)\|^2 - \frac{1}{C_2} \sum_{x \nsim y} \|\Phi(x) - \Phi(y)\|^2$
- If now, we are going to use the imbedding for classification, then we can skip ahead to

$$L(x_1, x_2, x_3, \ldots, x_N) = -\log\left(\frac{\exp(x_1 x_2)}{\sum_{i.j} \exp(x_i x_j)}\right)$$

which combines the imbedding with the fact that only the first pair is similar.

Appendix. Background math : metric imbeddings and generalization

It's natural to assume that data has a lower dimensional representation - we want to be able to throw away "nuisance" features, and keep the relevant ones.   We can mathematically define this by imagining the following setting

Imagine you take a bunch of points x_i, randomly sampled from a low dimensinonal unit box $[0,1]^d$. Then you embedd them in a higher dimensional space (say by just appending zeros to random coordinates), and add a tiny amount of random noise.   Suppose you forget the imbedding mapping. Can you recover the original points? The answer is basically yes, with high probability.

The Johnson-Lindenstrauss lemma not only proves that we can do this, but it also gives an algorithm, https://en.wikipedia.org/wiki/Johnson%E2%80%93Lindenstrauss_lemma. By taking projections with random vectors, we can map the points in the big space back down to a smaller space, and recover the distance up to a small factor. The formula is

$$(1-\epsilon)\|Px - Py\|^2 \leq \|x - y\|^2 \leq (1+\epsilon)\|Px - Py\|^2$$

The small factor ϵ depends on the number of data points. The matrix P is the projection matrix built from taking a number of random vectors. In high dimension, random unit vectors are nearly orthogonal, so it is nearly just the dot products x, η_i for each η_i

In fact, this result allows us to choose the dimenion of the target space. There is a trade-off: to low dimension means we may lose detail, and too high means we are not compressing the data. In the example we gave, we know the optimal dimension, but this is often not the case.

This result tells us that we can do the same when we have data, and a distance $d(x,y)$: find the imbedding into a lower dimensional feature space which preserves the distance.

Generalization

The result above says that certain kind of map used for the metric imbedding will generalize, in the sense that it capture the distance within a certain error. This result can also be extended to Kernels without too much difficultly (mainly notation).

If we want to get the result to work with nonlinear maps (e.g. networks), it won't work without an extra ingredient. Because without assumptions on the map, e.g. could just learn pairs of distances, and be zero everwhere else.

However, the regularization should be designed to make the map less oscillatory, or closer to linear. In this case it should be more likely to generalize (but may only have a rigorous proof in the kernel case)