# cs276a PE2 Review

## Dimensionality Reduction

Assume you have your data in a (m x N) term incidence matrix X, where N is the number of documents and m is the number of terms. Thus each document is represented by a column of X. Let $X^{(i)}$ denote the i-th column of X (e.g. the vector representing document *i*).

For the purposes of the k-means algorithm, each column of X (e.g. each document) is considered to be a point in m-dimensional space. The algorithm produces sensible results because the distance between the columns of X represents the similarity between the documents. Looking at it very simply, given documents *i* and *j*, k-means uses the distance $||X^{(i)}-X^{(j)}||$ to decide whether *i* and *j* should be in the same cluster. If they are close (as compared to other documents), then they'll likely end up in the same cluster, and a human looking at the clustering will agree with this assignment, because small distance means that the documents are similar.

The running time of this algorithm is highly dependent on m, and m can be quite large. Dimensionality reduction allows us to run k-means on a (d x N) matrix R which we derive from X, where d<m, and get a clustering that is as "good" as the one we would have gotten had we run it on the original (m x N) matrix X, in much less time.

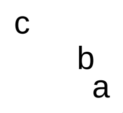
To do so we need to construct the (d x N) matrix R from the (m x N) X. The definition of matrix multiplication tells us that given any (d x m) matrix P, we can construct a (d x N) matrix simply by matrix multiplication $\overset{d\times N}{R} = \overset{d\times m}{P} * \overset{m\times N}{X}$ . Let some such P be given, and consider what happens when we run k-means on R. It will be highly dependent on P. For example if P is the zero matrix, then R is the zero matrix; for any documents *i* and *j*, whatever the distance $||X^{(i)}-X^{(j)}||$, $||R^{(i)}-R^{(j)}||=0$, and so we have a degenerate case where the k-means algorithm, looking only at R, will consider all documents the same, and will not give us a sensible answer, certainly nothing like we would get by running it on X.

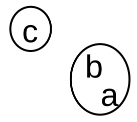
Let us now suppose we were given a P such that for any pair of documents *i* and *j*, we were told that $||X^{(i)}-X^{(j)}||=||R^{(i)}-R^{(j)}||$. In this case, our notion of similarity is completely preserved, and we can expect k-means on R to produce a clustering as "good" as the one it would produce on X. In general such a P does not exist, however there are many matrices P which will get us close, so that for any pair of documents *i* and *j*, $||X^{(i)}-X^{(j)}||\approx||R^{(i)}-R^{(j)}||$. How well the distances are preserved, will determine how sensible the clustering we get from k-means on R will be, and this is directly dependent on P.

## Constructing P

As mentioned above there are many possible matrices P that can work, and there are many methods to construct them [2]. We present a few below, they are ordered by computation speed. Each one works harder than the ones before it to construct P, but gives more assurances that the distances between documents - our notion of similarity -

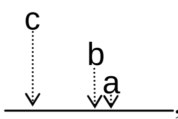
will be preserved, and thus that the clustering we get by running k-means (or some other clustering algorithm) on R (=PX) will be reasonable.

1. P is a completely random matrix, every element of the matrix, $P_{ij}$, is drawn from some zero mean, unit variance distribution.
    a. The cheapest way that can work is to simply have
    $$P_{ij} = \sqrt{3} * \begin{cases} +1 & \text{with probability } \frac{1}{6} \\ 0 & \text{with probability } \frac{2}{3} \\ -1 & \text{with probability } \frac{1}{6} \end{cases}.$$
    b. A little more expensive is to have every element of P be Gaussian distributed, $P_{ij}$=N(0,1) (in matlab do **P=randn(d, m)**).
    c. Constructing P as above is very easy, but you get few assurances that it will preserve distances between documents, though [1] suggests that it may.
2. First construct P randomly as above, and then *orthogonalize* it.
    a. P is called **orthogonal** if every row of P is perpendicular to every other row, and also every row, when interpreted as a vector, has length 1. The reason why we want to do this, is that if P is orthogonal, R=PX has a simple geometric interpretation -- R is a projection of X from a m-dimensional space, to a d-dimensional one, where the the rows of P are the d directions we've selected to project down to.
        i. Example: we have 3 documents a,b, and c, each a 2 dimensional vector: 

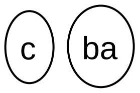
. If we cluster in the original 2 dimensions, we would expect a and b to be clustered together, and c to be in its own cluster: 

. Let us first do a dimensionality reduction to 1 dimension, d=1.

1. First we pick a direction to reduce to and project down:



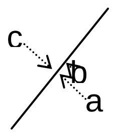
, getting

, and doing the clustering we get


. We've just expressed geometrically what R=PX means, in this case we picked [1 0] as our projection direction, so P=[1 0], if we denote by ($a_x$, $a_y$) the coordinates of a, similarly for b and c, then we can write the above as

$$\underset{R}{\begin{bmatrix} a_x & b_x & c_x \end{bmatrix}} = \underset{P}{\begin{bmatrix} 1 & 0 \end{bmatrix}} \underset{X}{\begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \end{bmatrix}}$$

2. We were lucky in picking a projection direction which preserved our clustering; let's see what happens when we are not so lucky.

   a. With a different direction:
   

   , we get the projection

   , and with everything on top of each other, when we do k-means for 2 clusters we get random results.

ii. In higher dimensions, everything proceeds similarly, except we pick more than one direction to project down to - we pick d of them. If the directions we pick are not perpendicular to each other, in other words P is not orthogonal, as with the first scheme when P is completely random, then the geometric interpretation of what we've done is not only project down to a d-dimensional space, but also stretch things in various directions; no reason to do that if we can avoid it.

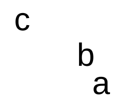
b. There are many algorithms to orthogonalize a matrix, such as Gram-Schmidt and Householder transform (in matlab do `P=orth(P')'`)

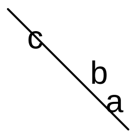
   i. Note that the hashes (“ **'** “) are important! They transpose the matrix; if you wanted to orthogonalize the columns of P, you would just say `P=orth(P)`, but we want the rows, so we first transpose P -- `P=P’`, which makes its rows into columns, orthogonalize, and then transpose back to get them as rows.

3. In the last method, we will construct a P that is optimal, meaning that R=PX will preserve distances between documents better than any other possible (d x m) P. We use a *singular value decomposition* (SVD) to find matrices U, S, and V such that get $X=USV^T$, we set $P=\{\text{first d columns of U}\}^T$, and compute R=PX as before (in matlab do `[U,S,V]=svd(X); P=U(:,1:d)'`).

   a. SVD is expensive to compute. Can use iterative methods to only get the first d columns of U that we actually end up using (in matlab do `[U,S,V]=svds(X,d); P=U’`); even so, still expensive.

   b. Why does it work?

      i. Consider our earlier example 

, the best projection direction would be the one that best preserves the distances between points, this one: 

. This is precisely the one we find with the SVD method, it is the eigenvector of $AA^T$ corresponding to the largest eigenvalue; U contains these eigenvectors in its columns, sorted by eigenvalue, so when we take the first d columns, we are taking the eigenvectors corresponding to the biggest d eigenvalues.

## A word on SVD

For any matrix X, SVD gives us a decomposition of X into three matrices U, S (often called $\Sigma$ ), and V such that $X=USV^T$. All three matrices are very specific and special for

X (see lecture 15). Because S is diagonal (eg $S = \begin{pmatrix} \sigma_1 & & & & \\ & \sigma_1 & & & 0 \\ & & \ddots & & \\ & & & \sigma_r & \\ & 0 & & & 0 \end{pmatrix}$), if we perform the multiplication symbolically, we get $X = \sum_i \sigma_i u_i v_i^T$ , where $u_i$ is the i-th column of U, and $v_i$ is the i-th column of V. Every $u_i v_i^T$ is a (m x N) matrix just like X, so

$$X = \sigma_1 \begin{bmatrix} & & \\ & u_1 v_1^T & \\ & & \end{bmatrix} + \sigma_2 \begin{bmatrix} & & \\ & u_2 v_2^T & \\ & & \end{bmatrix} + \sigma_3 \begin{bmatrix} & & \\ & u_3 v_3^T & \\ & & \end{bmatrix} + \cdots + \sigma_m \begin{bmatrix} & & \\ & u_m v_m^T & \\ & & \end{bmatrix}.$$

Recall that SVD sorts the sigmas by magnitude, so $\sigma_i \geq \sigma_j$ when $i \geq j$ , in other words we have

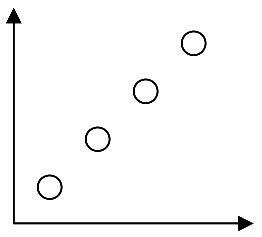
$$X = \boldsymbol{\sigma_1} \begin{bmatrix} & & \\ & u_1 v_1^T & \\ & & \end{bmatrix} + \boldsymbol{\sigma_2} \begin{bmatrix} & & \\ & u_2 v_2^T & \\ & & \end{bmatrix} + \boldsymbol{\sigma_3} \begin{bmatrix} & & \\ & u_3 v_3^T & \\ & & \end{bmatrix} + \cdots + {\scriptstyle \sigma_m} \begin{bmatrix} & & \\ & u_m v_m^T & \\ & & \end{bmatrix}$$

We see that the last terms of this sum are probably not significant compared with the first ones, and we can ignore them (though in the worst case we have $\sigma_1 = \sigma_2 = \cdots = \sigma_m$ , and then we can't ignore them). This is precisely what LSI does, for example when we compute the approximation for k=2, we have

$$X_2 = \boldsymbol{\sigma_1} \begin{bmatrix} & & \\ & u_1 v_1^T & \\ & & \end{bmatrix} + \boldsymbol{\sigma_2} \begin{bmatrix} & & \\ & u_2 v_2^T & \\ & & \end{bmatrix}.$$

**Matrix Rank**

Let X represent a set of points as before, and suppose the points look like this:



then X will look something like $\begin{pmatrix} 0.5 & 1.5 & 2 & 2.5 \\ 0.5 & 1.5 & 2 & 2.5 \end{pmatrix}$. Even though X represents the points in two dimensions, we see that all the points lie on a line. This means that our dataset, even though we've represented it in two dimensions, is one dimensional, and we say that the row **rank** of X is 1. In a similar way, in higher dimensions, X may have m rows, but the data is really d-dimensional (eg the row rank of X is d), in which case we are better off projecting the data to d-dimensions with SVD; we lose no information and we have fewer numbers to deal with. Note that if we don't project with SVD, but do so randomly, then we may still lose information - in the example above if we project down to a line perpendicular to the line the points lie on, then we end up with all the points being the same and so we've lost all distance information; using SVD guarantees that we will project down to the line the points lie on.

## Matlab

We suggest you consider using Matlab in your project, it is one of the best packages for numerical computation and is easy to learn (see [3] for tutorial). All of the more sophisticated methods of constructing P, at least one of which you should try involve some amount of numerical computation. Java has some native linear algebra libraries such as [4], but they may turn out to be too slow for our purposes. If you do decide to use Matlab, you'll need to write your data out to files, and then read them into Matlab for processing (do `help fscanf` in the matlab command window to see how to do file IO).

[1] D. Achlioptas. Database-friendly random projections. In *Proc. ACM Symp. on the Principles of Database Systems*, pages 274-281, 2001.

[2] Bingham, E. and Mannila, H. Random Projection in Dimensionality Reduction: Applications to Image and Text Data. *7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2001),* San Francisco, CA, USA, August 26-29, 2001. pp. 245-250.

[3] http://web.mit.edu/afs/.athena/astaff/project/logos/olh/Math/Matlab/Matlab.html

[4] http://math.nist.gov/javanumerics/jama/