CS168: The Modern Algorithmic Toolbox
Lecture #8: PCA and the Power Iteration Method

Tim Roughgarden & Gregory Valiant

April 15, 2015

This lecture began with an extended recap of Lecture 7. Recall that if we have a matrix X, whose rows represent the data points, then we can find an orthonormal matrix (rows/columns are orthogonal to eachother, and have norm 1) Q and a diagonal matrix D, s.t. $X^tX = Q^tDQ$. Given such a decomposition, the rows of Q are the eigenvectors of X^tX, with associated eigenvalues given by the corresponding entries of the diagonal matrix D. The decomposition $X^tX = Q^tDQ$ is unique, up to permuting the diagonal entries of D (and the corresponding permutation of the rows of Q). Henceforth we will assume that we choose the representation in which the entries of D are ordered in decreasing order from the upper left, to lower right.

Recall that there are several uses of PCA:

- As a tool for dimension reduction: e.g. we want to represent high-dimensional data as k-dimensional data, one natural choice is to pick the k dimensional subspace that capture as much of the lengths of the datapoints as possible. This is the space spanned by the top k principal components. Note that this subspace is dependent on how we scale the data (as opposed to the random projection dimension reduction JL-machinery we saw in Lecture 4).

- As a tool for de-noising data: i.e. the hope is that the data is actually low dimensional, but with noise. By keeping the low-dimensional structure, we are removing the unstructured noise.

- As a tool for revealing structure in the data. For example, the European Genomes example I gave (see the links from the course page), shows that the top two principal components of genetic snp data corresponds to geographic latitude/longitude.

1 The Power Iteration Method

How does one compute the top k principal components? There are two main approaches: 1) an explicit computation of the decomposition $X^tX = Q^tDQ$, and 2) the power iteration method. The first approach, which can be done symbolically, is basically only feasible for

small matrices (and involves computing roots of the "characteristic polynomial" of the matrix X^tX. The power iteration method is what is typically used in practice (especially in settings where we only want the top few principal components), and gives some additional intuition as to what the principal components actually mean.

The main intuition behind the power method is that if one regards the matrix $M = X^tX$ as a function that maps the unit sphere to an ellipse, the longest axis of the ellipse corresponds to the top principal components. Given that the top principal components corresponds to the direction in which multiplication by M stretches the vector the most, it is natural to hope that if we start with a random vector v, and keep applying M over and over, we will end up having stretched v so much in the direction of M's top principal component, that the image of v will essentially lie almost entirely in the direction of this top principal component. This is the power iteration algorithm. We now restate it formally:

Algorithm 1
POWER ITERATION

Given matrix $M = X^tX$:

- Select random unit vector v_0
- For $i = 1, 2, \ldots$, set $v_i = M^iv_0$. If $v_i/||v_i|| \approx v_{i-1}/||v_{i-1}||$, then return $v_i/||v_i||$.

Often, rather than computing $Mv_0, M^2v_0, M^3v_0, \ldots, M^4v_0, M^5v_0$ one instead uses the repeated squaring "trick" and computes $Mv_0, M^2v_0, M^4v_0, M^8v_0, \ldots$.

To see why the algorithm works, we first establish that if $M = Q^tDQ$, then $M^i = Q^tD^iQ$—namely that M^i has the same orientation (i.e. eigenvectors) as M, but all of the eigenvalues are raised to the ith power (and are hence exaggerated—e.g. if $\lambda_1 > 2\lambda_2$, then $\lambda_1^{10} > 1000\lambda_2^{10}$).

Claim 1.1 *If $M = Q^tDQ$, then $M^i = Q^tD^iQ$.*

Proof: We prove this via induction on i. The base case, $i = 1$ is immediate. Assuming that the statement holds for some specific $i \geq 1$, consider the following:

$$M^{i+1} = M^i \cdot M = Q^tD^iQQ^tDQ = Q^tD^iDQ = Q^tD^iQ,$$

where we used our induction hypothesis to get the second equality, and the third equality followed from the orthogonality of Q, which allowed us to replace QQ^t by the identity matrix. ■

We can now quantify the performance of the power iteration algorithm:

Theorem 1.2 *With probability at least* $1/2$ *over the choice of* v_0*, for and* $t \geq 1$*,*

$$|\langle M^t v_0, w_1 \rangle| \geq 1 - 2\sqrt{d} \left(\frac{\lambda_2}{\lambda_1} \right)^t,$$

where w_1 *is the top eigenvector of* M*, with eigenvalue* λ_1*, and* λ_2 *is the second-largest eigenvalue of* M*. In particular, if* $t > 10 \frac{\log d}{\log \frac{\lambda_1}{\lambda_2}}$ *then* $|\langle v_t, w_1 \rangle| > 1 - 2e^{-20} > 0.99999$*, namely* v_t *is essentially pointing in the same direction as* w_1*.*

Proof: Let $w_1, w_2, \ldots,$ represent the eigenvectors of M, with associated eigenvalues $\lambda_1, \ldots$. We can consider the representation of our random vector v_0 in the (unknown) basis defined by $w_1, w_2, \ldots$. Namely $v_0 = c_1 w_1 + c_2 w_2 + \ldots$. We claim that, with probability at least $1/2$, $|c_1| > \frac{1}{2\sqrt{d}}$. This is a simple fact about random vectors, and follows from the fact that we can choose a random unit vector by selecting each coordinate independently from a Gaussian of variance $1/d$, then normalizing (by a factor that will be very close to 1).

Given that $|c_1| > 1/2$,

$$|\langle M^t v_0, w_1 \rangle| = \frac{c_1 \lambda_1^t}{\sqrt{\sum_{i=1}^d (\lambda_i^2 c_i)^2}} \geq \frac{c_1 \lambda_1^t}{\sqrt{c_1^2 \lambda_1^{2t} + d\lambda_2^{2t}}} \geq \frac{c_1 \lambda_1^t}{c_1 \lambda_1^t + \lambda_2^t \sqrt{d}} \geq 1 - 2\sqrt{d} \left(\frac{\lambda_2}{\lambda_1} \right)^t.$$

■

Note that the "with probability 1/2" statement can be replaced by "with probability at least $1 - 1/2^{100}$ by repeating the above algorithm 100 times (for independent choices of v_0), and outputting the recovered vector v that maximizes $||Mv||$.

The above theorem shows that the number of power iterations we require scales as $\frac{\log d}{\log \frac{\lambda_1}{\lambda_2}}$. If λ_1/λ_2 is large, then we are in excellent shape. If $\lambda_1 \approx \lambda_2$, then the algorithm might take a long time (or might never) find w_1. Note that if $\lambda_1 = \lambda_2$, then w_1 and w_2 are not uniquely defined—-any pair of orthonormal vectors that lie in this 2-d space will be eigenvectors with eigenvalue $\lambda_1 = \lambda_2$. Note that in this case, the power iteration algorithm will simply return a vector that lies in this subspace, which is the correct thing to do.

1.1 Finding Lots of Principal Components

The power iteration algorithm finds the top principal component. If we want to find the top k, we can do the following:

1. Find the top component, w_1, using power iteration.
2. Project our data orthogonally to w_1, by, for each datapoint x, replacing it with $x - w_1 \langle x, w_1 \rangle$.
3. Recurse by finding the top $k-1$ principal components of the the new data (in which the direction w_1 has been removed).

The correctness of this greedy algorithm follows immediately from the fact that the k dimensional subspace that maximizes the norms of the projections of a data matrix X contains the $k-1$ dimensional subspace that maximizes the norms of the projections.

1.2 Eigen-spectrum

How do you know how many principal components to find? The simple answer is that you don't. In general, it is worth computing a lot of them, considering the eigen-spectrum—namely the set of eigenvalues. Often, if you ploting the eigenvalues, you will see that after some point, they are all fairly small (i.e. your data might have 200 dimensions, but after the first 50 eigenvalues, the rest are all tiny). Looking at this plot might give you some heuristic sense of how to choose the number of components so as to maximize the signal of your data, while preserving the low-dimensionality.

2 Failure Cases

When and why does PCA fail?

1. You messed up the scaling/normalizing. PCA is sensitive to different scalings/normalizations of the coordinates. Much of the artistry of getting good results from PCA involves choosing an appropriate scaling for the difference data coordinates.

2. Non-linear structure. PCA is all about finding linear structure in your data. If your data has some low-dimensional, but non-linear structure, i.e. you have two data coordinates, x, y, with $y \approx x^2$, then PCA will not find this. [Note: of course you can always add extra non-linear dimensions to your data, i.e. add coordinates x^2, y^2, xy, etc., and then PCA this higher dimensional data....you can even do this in a kernelized manner....if you haven't seen the kernel trick in another class, don't worry about this last comment.]

3. Non-orthogonal structure. It is nice to have coordinates that are interpretable. Even if your data is essentially linear, the fact that the principal components are all orthogonal will often mean that after the top 3 or 4 components, it will be almost impossible to interpret the meanings of the components. This is because, say, the 5th component is FORCED to be orthogonal to the top 4 components, which is a rather artificial constraint to put on an interpretable feature.

3 Power iteration, principal components, and Markov Chains

I didn't have time to mention this in class, but wanted to just make a note of this because someone asked after class. (Don't worry, this won't be on the exam.)

You might recall the expression $M^t \cdot v$ from Lecture 6 on Markov Chains. In that lecture, M was the transition matrix of a Markov Chain, v was an initial state (or distribution over states), and $\lim_{t\to\infty} M^t v = \pi$, which satisfies $M^t\pi = \pi$ is the stationary distribution for the Markov Chain, which is the limiting distribution, provided such a distribution exists (recall the two conditions for the fundamental theorem of Markov chains: every state must be eventually reachable from every other state, and the chain cannot be "periodic"). After today's lecture, it should be clear that π is the top eigenvector of the transition matrix M, and running MCMC (or explicitly calculating $M^t v$) are both essentially applying the power iteration method to find this limiting "stationary" distribution.

In this lecture $M = X^t X$ is symmetric, whereas a typical transition matrix for a Markov Chain might not be symmetric. The main difference is that for a transition matrix M that is not symmetric, we will have negative eigenvalues, and the interpretation/intuition gets a bit more complicated, though many of the ideas still apply. We will talk more about what happens when one applies PCA to transition matrices in the last week of the course.