CS 5785 – Applied Machine Learning – Lec. 8

Prof. Serge Belongie, Cornell Tech
Scribe: Jocelyn Kong, Jeff Ponnor, Harrison Gregg, Ashwin Ramanathan, Elena Zhizhimontova

Sept. 24, 2018

1 Computing Eigenvectors of Large Matrices

Given a matrix $\mathbf{B}$ and a vector a, the *Rayleigh quotient* is defined as:

$$\frac{a^\top \mathbf{B} a}{a^\top a}$$

In the numerator we have the quadratic form $a^\top \mathbf{B} a$ and in the denominator we have $a^\top a$, which is simply $\|a\|^2$, the squared norm of a. Dividing out the norm has the effect of normalizing the expression, so without loss of generality, we can think of a as having unit norm. Now consider the set of all possible vectors a: the choice of a that maximizes the Rayleigh quotient is the leading eigenvector of $\mathbf{B}$. The value attained at the maximum is the corresponding eigenvalue. This leading eigenvector represents the dimension along which the data exhibits the most spread. For successive eigenvectors, we repeat the maximization procedure on a with the constraint that they be orthogonal to the previous eigenvectors.

What happens if you want to do the SVD on really large matrices? In many cases of practical interest, we only need a relatively small set of leading eigenvectors/singular vectors. We observed this, for example, in the clown image from the previous lecture, for which the singular values drop off dramatically. Going from one row to the next in this image, the pixel values don't change very much.

1.1 Power Iteration

Suppose we have $\mathbf{B} = \mathbf{X}^\top \mathbf{X}$ and we want the leading eigenvector. Let $\mathbf{\Lambda} = \mathbf{D}^2$ and $\mathbf{B} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top$. Pick a random vector a_k and iterate on

$$a_{k+1} = \frac{\mathbf{B} a_k}{\|\mathbf{B} a_k\|}$$

In general, multiplying a matrix $\mathbf{A}$ by a vector a does two things to that vector: it rotates and scales it. If you choose a to be an eigenvector of $\mathbf{A}$, something special happens: $\mathbf{A}$ just scales the vector by a constant. If you repeat the iteration defined above, you will eventually converge to an eigenvector. Why does this work? Effectively, we are multiplying a_k by higher and higher powers

of $\mathbf{B}$. Observe what happens when we raise $\mathbf{B}$ to a high power:

$$\begin{aligned}\mathbf{B}^n &= (\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top)^n \\ &= (\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top)(\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top)\cdots(\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top) \\ &= \mathbf{V}\mathbf{\Lambda}^n\mathbf{V}^\top\end{aligned}$$

In other words, the exponent n moves inside the diagonalization to sit on $\mathbf{\Lambda}$, the diagonal matrix of eigenvalues ($\lambda_1, \lambda_2, \ldots$ in descending order). Now suppose we divide all the eigenvalues by λ_1. This will make the first entry 1 and all the rest less than 1. If you raise all these numbers to the n^{th} power, the other values will eventually get driven to 0 if there's a gap between λ_1 and the rest of the eigenvalues. After many iterations, we get a rank 1 matrix $v_1 v_1^\top$, from which we can easily extract the leading eigenvector.

What causes trouble for power iteration? If the gap between the first eigenvalue and the next ones is really small, it will take a very long time to drive the next eigenvalue to 0. The pathological case would be a gap of zero, i.e., repeated eigenvalues. This is called a subspace and power iteration will break. To avoid problems such as repeated eigenvalues and also to exploit properties of large matrices that are sparse, we use tools such as inverse iteration or the Lanczos Method, which is well-suited to Map-Reduce implementation. Inverse iteration was developed as a followup to power iteration that can handle subspaces of size k. A sparse matrix is a matrix with lots of zeros which could represent, for example, the vast quantities of movies or songs users have not rated on sites such as Netflix or Pandora that make extensive use of recommender systems. (It's not realistic to assume users on such sites can rate every item).

2 Kernel Smoothing

Kernel smoothing is an idea from statistics in which we use weighted averages to smooth noisy, irregularly spaced training data. When you have a finite sample of data as depicted in Figure 1, you may have reason to believe there's an underlying smooth signal for which your finite sample represents. Kernel smoothing helps you obtain a continuum, if you want to look up a function value at a point where you didn't sample any data, kernel smoothing can give you an answer.

Suppose you want to smooth noisy sensor measurements that are a function of time. One method is k nearest neighbor smoothing:

$$\widehat{f}(x) = Ave\,(y_i | x_i \in N_k(x))$$

This is an estimate of the regression function $E(Y|X = x)$ where $N_k(x)$ is the set of k points closest to x and Ave computes the mean. This is what's happening in the Figure 1(left). In other words, the k-NN smoother estimates $\widehat{f}(x_0)$ at some arbitrary value x_0 using the average value of the k nearest neighbors of x_0. In Figure 1, the blue curve represents the ground truth curve from which the noisy samples were obtained. The result of the k-NN smoother is shown by the green line in the plot on the left. This method, though simple, generally yields a jumpy discontinuous curve, which is undesirable. As the window moves, one point is added and one point is left out, which leads to the jumpy behavior. The window is depicted by the yellow region in the left image in Figure 3.

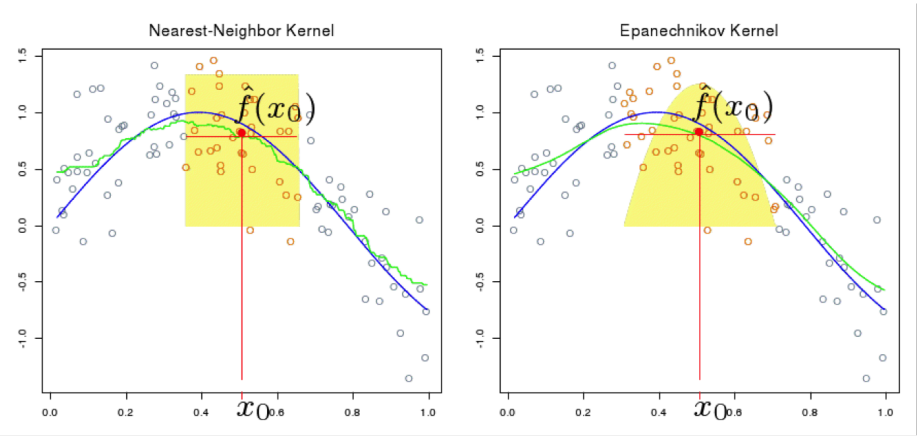

FIGURE 6.1

Figure 1: [Kernel Smoothing]

Kernel smoothers address this problem by using a weighting function that dies off smoothly toward the edges. Thus, the smoothing function pays more attention to data points closer to the target point and less attention to data points further away. We express this as follows:

$$\widehat{f}(x_0) = \frac{\sum_{i=1}^{N} K_\lambda(x_0, x_i) y_i}{\sum_{i=1}^{N} K_\lambda(x_0, x_i)}$$

$$K_\lambda(x_0, x) = D\left(\frac{|x - x_0|}{\lambda}\right)$$

$$D(t) = \begin{cases} \frac{3}{4}(1 - t^2), & \text{if } |t| \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

In this case, we have chosen $D(\cdot)$ to be an *Epanechnikov kernel*, which has a quadratic shape within a fixed interval. In this case, you pick a particular width, λ, and you perform a weighted average of the y values in the corresponding interval, paying more attention to the data near the center. Weighting the y values lets points closest to the center have the greatest influence, instead of all points having the same impact, as in the k-NN smoother. This results in a smoother curve, shown in green on the right of Figure 1. The yellow part in the right image in Figure 3 depicts the chosen window of this fancy weighted average.

Note, this isn't necessarily any more accurate than what we obtained from the k-NN smoother, though it may be qualitatively better. What matters is its utility in real world applications, i.e., its ability to predict subsequent data.

This idea of kernel smoothing can also be applied to density estimation, which is discussed in the next lecture.