Unsupervised Learning

Piyush Patil

March 10, 2017

In contrast to all of the methods we've studied so far, unsupervised learning involves finding patterns in unlabeled data. Rather than learning by example, which is the driving motivation for supervised learning, unsupervised learning is about discovering hidden structure in unstructured data.

1 Singular Value Decomposition

The spectral theorem gives us a convenient factorization for square symmetric matrices; in this section, we'll examine more general methods for factoring arbitrary, non-symmetric, non-square matrices usefully. Recall that the utility of the spectral theorem was that the factorization consisted of a diagonal matrix and two orthonormal matrices. It turns out we can factor an arbitrary $n \times d$ matrix X similarly:

$$X = UDV^\intercal = \sum_{i=1}^{d} \delta_i u_i v_i^\intercal$$

where $U = \begin{bmatrix} u_i \end{bmatrix}$ is $n \times d$, $D = \text{diag}(\delta_i)$ is a diagonal $d \times d$ matrix, and $V^\intercal = \begin{bmatrix} v_i \end{bmatrix}^\intercal$ is a $d \times d$ matrix. Analogous to the spectral theorem, the columns of U and the rows of V are orthonormal, so that $U^\intercal U = V^\intercal V = I$. We call the columns of U the *left singular vectors* of X, and the rows of $V^\intercal$ the *right singular vectors* of X. We call the δ_i's the *singular values* of X, and are guaranteed to be non-negative (in fact, though some might be zero, the number of positive singular values equals the rank of X). This is especially useful to us because when X is a data matrix, the quantity $X^\intercal X$ shows up quite a bit in closed form solutions, but is expensive to compute (it takes $O(nd^2)$ time). Given the factorization, however, we need only compute $X^\intercal X = VDU^\intercal UDV^\intercal = VD^2V^\intercal$.
All these properties stem from connections between the three matrices above: v_i is the eigenvector of $X^\intercal X$ with eigenvalue δ_i^2. We can find the k largest singular values, and their corresponding eigenvectors, in $O(ndk)$ time.

Motivation: From a machine learning perspective, the concept of singular vectors can be motivated from linear least squares regression. Viewing the rows of X as d points in n-dimensional space, consider the least squares best fit line (through the origin) for the points. Let v be a unit vector along this line, so the line is essentially uniquely determined by v. For a point x_i, $|v \cdot x_i|$ is the length of the projection of x_i onto v; in other words, it's the distance between the point x_i and the line determined by v, i.e. the least squares best fit line. Since $|Xv|^2$ is the sum of the squared such projections, the best fit line is the one determined by the v which maximizes $|Xv|^2$. We thus define the *first singular vector* of X to be

$$v_1 = \arg\max_{|v|=1} |Xv|$$

It follows that the first singular vector is precisely the one determining the least squares best fit line (through the origin) for the points in the data matrix X (we'll prove later that the v_i's defined below are exactly the ones composing the rows of $V^\intercal$ above). Let's generalize this process - suppose we want to find the best fit plane through the points. The greedy approach, which turns out to be correct in this case (by the Gram-Schmidt process), is the fix v_1 as one of the basis vectors of the plane, and search for the other one; if we again use least squares best fit, this means

$$v_2 = \arg\max_{|v|=1, v^\intercal v_1 = 0} |Xv|$$

is the second basis vector. Notice that what we've done is essentially enforce the greedy assumption that the best fit plane contains the best fit line, allowing us to use v_1 as our first basis vector, and then simply use least squares to find the second basis vector v_2 (which, by design, is both perpendicular to v_1 and minimizes the squared distances to the points by maximizing $|Xv|$). Continuing in this way, we define

$$v_i = \arg\max_{|v|=1, v^\intercal v_j = 0} |Xv| \text{ for } j < i$$

to be the i^{th} *singular vector* of X, so that $\{v_1, \cdots, v_k\}$ determines the least squares best fit k-dimensional subspace for X.

Intuition: From a high level, the singular value decomposition is a main result in operator theory, and is a powerful statement on arbitrary operators between two different spaces. It essentially states that with respect to suitably chosen bases, any linear operator can be expressed as a diagonal matrix; moreover, the bases in question are quite amenable - they are orthonormal. Geometrically, the singular value decomposition states that hyperspheres in the domain are transformed into ellipsoids in the co-domain. Because the operator might not be injective, the dimension of the ellipsoid is at most the dimension of the sphere, so there's some "collapsing" along some axes, and distortion along others, but these changes are the only thing the operator does, nothing more. Because this applies to any arbitrary linear operator, the theorem is a major one in operator theory. The singular vectors are simply the bases orthonormalizing the domain and co-domain of the linear operator. This can also be seen with the original derivation of SVD, which comes from the eigenvectors and eigenvalues of the Hermitian matrix

$$\begin{bmatrix} O & A \\ A^\intercal & O \end{bmatrix}$$

where O is a zero matrix of appropriate size to make the above matrix square. The eigenvectors and eigenvalues are the singular vectors and values, respectively.
Suppose A is a square matrix. If $A = U\Sigma V^\intercal$ then U diagonalizes $AA^\intercal$ (so it's columns are the (orthonormal) eigenvectors of $AA^\intercal$) and V diagonalizes $A^\intercal A$ (so it's columns are the (orthonormal) eigenvectors of $A^\intercal A$). The squared singular values are the eigenvalues of $A^\intercal A$. This is clear from the decomposition:

$$AA^\intercal = (U\Sigma V^\intercal)(U\Sigma V^\intercal)^\intercal = U\Sigma V^\intercal V\Sigma^\intercal U^\intercal = U\Sigma^2 U^{-1}$$

$$A^\intercal A = (U\Sigma V^\intercal)^\intercal(U\Sigma V^\intercal) = V\Sigma^\intercal U^\intercal U\Sigma V^\intercal = V\Sigma^2 V^{-1}$$

Thus, U and V are simply taken from applying the spectral theorem to $AA^\intercal$ and $A^\intercal A$, which are both guaranteed to be symmetric. This reinforces the intuition that singular value decomposition is a generalization of the spectral theorem.
Geometrically, since orthonormal matrices preserve the length of a vector under multiplication, we can view the singular value decomposition as breaking the action of A, as a linear operator, down into a rotation, followed by a scaling of the axes, followed by another rotation. Moreover, because there are only as many non-zero singular values as the rank of A, the singular value decomposition can be written

$$A = \sum_{i=1}^{r} \sigma_i \cdot u_i v_i^\intercal \text{ where } r = \text{rank}(A)$$

i.e. the sum of r rank one matrices. If we want to find the best rank k approximation to A, we can simply truncate the sum by taking only the terms corresponding to the k largest singular values. Though we've been working with square A so far, the full decomposition is a generalization.

2 Principal Components Analysis

Principal component analysis, or PCA, is an unsupervised method for dimensionality reduction of data. Dimensionality reduction in general is a field which tries to many unavoidable real world problems in dealing with data. It's very often the case that when dealing with data describing the dynamics of an unknown system, although the data is sufficient to provide a complete description of the system, it's dimensionality is often much higher than necessary. Because we don't know the underlying structure of the system, we often capture data in far too many dimensions than necessary, which not only imposes computational constraints but also obscures the underlying structure and patterns contained in the system. Moreover, we will almost always have to deal with some level of random noise which throws variance into the data. By reducing the dimensionality by exploiting hidden structure in the data, we can deal with both of these problems much more easily.

From a high level, the idea is to, given sample points in $\mathbb{R}^d$, find k orthogonal directions which capture the most variation in the data. More specifically, in PCA we compute a basis to express the data in which hopefully will highlight the underlying patterns in the data, filtering out the noise and eliminating redundant dimensions. The basis is computed in a greedy manner - we start by choosing the direction which captures the most variation in the data (so that projecting down onto this dimension will minimize information loss), and henceforth iteratively choose a direction which maximizes the variation in the data under the constraint of being orthogonal to all the presently chosen directions, thereby forming an orthogonal basis.

More precisely, let X by an $n \times d$ centered (so the mean is zero) data matrix. Let w be a unit vector. The projection of a vector x onto w is given by $(x^\intercal w)w$, and geometrically corresponds to the operation of throwing away the spatial information contained in x and retaining only the information in the direction of w, similar to how a shadow is a two dimensional projection of a three dimensional object, created by discarding information about the object which is orthogonal to the shadow. The idea here is that we're going to choose the "best" (in a sense that will be made precise soon) direction, determined by w, allowing us to analyze our data in one dimensional space. To preserve more information than simply one

dimension, we can instead project x down to a subspace (the line determined by w is of course merely a one dimensional subspace); given a span $S = \{w_1, \cdots, w_s\}$, we compute the projection of x onto the subspace as

$$\text{proj}_S(x) = \sum_{i=1}^{s} (x^\intercal w_i) w_i$$

The question remains of how to choose S optimally. Intuitively, our projection shouldn't bunch up the data into the same spot, as this would lose a lot of information about the original data. Instead, we want to keep the data as spread out as possible after projecting. In other words, we want to maximize the variance. Thus, we choose our first basis vector w_1 with

$$w_1 = \arg\max_{||w||=1} \left(\frac{1}{n} \sum_{i=1}^{n} ||\text{proj}_w(x_i) - \mu|| \right) = \left(\frac{1}{n} \sum_{i=1}^{n} ||x_i^\intercal w - \mathbf{0}|| \right) = \arg\max_{||w||=1} \left(\sum_{i=1}^{n} x_i^\intercal w \right)$$

Or, in terms of the data matrix,

$$w_1 = \arg\max_{||w||=1} ||Xw||^2 = \arg\max_{||w||=1} w^\intercal X^\intercal X w = \arg\max_{||w||=1} \frac{w^\intercal X^\intercal X w}{w^\intercal w}$$

The last equality holds since w is a unit vector; our motivation for writing it this way is that this quantity is known as a *Rayleigh quotient* (since $X^\intercal X$ is square, symmetric, and positive semi-definite). The reason for introducing the concept of a Rayleigh quotient is that the Rayleigh quotient of a symmetric, square matrix M with respect to a non-zero vector x, defined as

$$R(M, x) = \frac{x^\intercal M x}{x^\intercal x}$$

is used in a very popular and fast eigenvalue algorithm known as Rayleigh quotient iteration, which iteratively approximates the eigenvalues of M given eigenvector approximation x. It follows that the best direction for us to choose for w_1 is the unit eigenvector of $X^\intercal X$ corresponding to the largest eigenvalue. If we continue the process, we'll find that in general the best choice for w_k is the unit eigenvector corresponding to the k^{th} largest eigenvalue of $X^\intercal X$. Notice that these are precisely the first k singular vectors of X.

(Theorem) Optimal PCA vectors: *The basis for the k-dimensional subspace maximizing variance over X is given by the first k singular vectors of X.*

We can interpret this as similar to fitting a linear subspace to our data, in the same way we did so in linear least squares regression. They key difference here is that in least squares, while we minimzied the distance between data points and our linear subspace, the "distance" was always taken to be a "vertical" projection (that is, the distance was computed along an axis), whereas in PCA we fit a linear subspace and minimize the squared distance to the data points, but we compute a true Euclidean distance, by taking not the vertical projection but rather the shortest projection, which is of course orthogonal to our linear subspace.

3 Clustering

Cluster analysis is a general field which aims to classify data into a set of groups or clusters, with the constraint that points in the same cluster are more similar to each other than they are to points in other clusters, for some notion of similarity. This can be useful in discovering patterns among the data based on which data points tend to cluster together, and is similar to nearest neighbors when applied to classification. There are many ways to cluster data - some common methods used in practice are connectivity (a.k.a. heirarchical) models, which tries to build clusters recursively out of smaller clusters in a hierarchical approach, centroid models, which characterize a cluster by some notion of a center vector (e.g. in k-means clustering, this is the literal arithmetic mean vector) describing the vectors in that cluster, distribution models, where clusters are modeled as statistical distributions, density models, where clusters are formed based on the density of the data, etc.

3.1 k-Means Clustering

One very common clustering method, and the most common centroid clustering model, is k-means clustering, which essentially tries to cluster the data points into k clusters which are each characterized by the arithmetic mean of the vectors in that cluster. The optimal clustering scheme is for every vector x to belong to the cluster whose mean is closer (in Euclidean distance) to x than any other cluster's mean. The result is a Voronoi diagram in which clusters form cells - vectors in a cell are closer to the cell's mean than to any other cell's mean.

More formally, given n data points that we wish to partition into k clusters, suppose we assign each input vector x_i a cluster $y_i \in \{1, \cdots, k\}$. Then cluster i has mean

$$\mu_i = \frac{1}{n_i} \sum_{y_j = i} x_i \text{ where } n_i = \sum_{y_j = i} 1 = \text{ number of points in cluster } i$$

We wish to find the optimal clustering, defined as the cluster partitioning which minimizes the loss

$$L = \sum_{i=1}^{k} \sum_{y_j = i} |x_j - \mu_i|^2$$

In other words, we want to minimize the total (squared) distances between each point and its cluster's mean.
This is a very natural and intuitive way to partition data points into k clusters, and is similar to nearest neighbors in that we simply put data points into the "closest" cluster. Unfortunately, finding the optimal clustering has been shown to be NP-hard, it's difficult to do significantly better than the brute force approach of simply trying every possible partition of clusters, which takes $O(nk^n)$ time. In practice, there are fast heuristic algorithms that can be used to obtain approximate solutions (the main downside is that these algorithms tend to get stuck in local optima). The following algorithm is the most common heuristic approximation algorithm for k-means clustering.

Lloyd's Algorithm: The idea behind Lloyd's algorithm is to iteratively update which clusters vectors are assigned to, based on the cluster with the closest mean. Because re-assigning vectors to clusters based on cluster means will change the cluster means in the next iteration, the algorithm will continue iterating until it stabilizes, i.e. until no vectors are re-assigned to a different cluster than they were in in the previous iteration. Thus, the algorithm initializes with a some set of initial clusters $\{S_1, \cdots, S_k\}$ (so that S_i's are disjoint sets which fully partition the full set of data) with corresponding means $m_1, \cdots m_k$, and consists of two steps:

1. *Assignment*: First, we re-assign the vectors based on the means.
   procedure ASSIGN($m_1, \cdots, m_k$):
   Clear the existing clusters $S_1, \cdots, S_k$
   for data point x_i **do**
   $c \leftarrow \arg\min_{1 \le i \le k}(||x_i - m_i||^2)$
   Add x_i to S_c
   end for
   return $S_1, \cdots, S_k$
   end procedure

2. *Update*: Next, after re-assigning vectors to their new clusters, we re-compute the cluster means for the next iteration.
   procedure UPDATE($S_1, \cdots, S_k$)
   for $i \in \{1, \cdots, k\}$ **do**
   $m_i \leftarrow |S_i|^{-1} \sum_{x \in S_i} x$
   end for
   return $m_1, \cdots, m_k$
   end procedure

The full algorithm is to simply go back and forth between assignment and updating until we converge and don't re-assign at all (i.e. when the update assignment function returns the same sets). This algorithm has the pitfall of getting stuck in local minima, however, which is unsurprising given that the similarity of its approach with hill climbing. However, it tends to be very fast in practice.

Euclidean distance isn't always the best measure for similarity; k-medoids clustering is a method which generalizes k-means clustering by allowing for any arbitrary distance metric between the points. Instead of mean points, sample points which minimize the distance metric to every other point in its cluster are used.

3.2 Hierarchical Clustering

This is a dynamic programming style approach to clustering which has the advantage that it doesn't need a pre-determined number of clusters. This is helpful since knowing how many clusters to cluster data in isn't always feasible in practice. Hierarchical clustering essentially produces a cluster tree, in which nodes are sample points and sub-trees are in the cluster determined by their parent. Thus, clusters have sub-clusters, and so on. The tree can be produced from a top down approach, in which we start with a cluster containing the entire dataset and iteratively split clusters into sub-clusters, or a bottom up approach, in which every point starts in its own cluster, and clusters are iteratively fused into parent clusters.

When it comes to top down clustering, there are many different approaches to finding the optimal division, some similar to decision tree splitting algorithms. In practice, bottom up clustering tends to perform much better when the dataset is a set of points, as opposed to a graph. With bottom up clustering, we often take the greedy approach of defining a distance function d' between clusters and fusing the two clusters which minimize the function. Given a distance function d over points in $\mathbb{R}^n$, common cluster distance functions are

1. Complete linkage: $d'(X,Y) = \max_{x\in X, y\in Y} d(x,y)$
2. Single linkage: $d'(X,Y) = \min_{x\in X, y\in Y} d(x,y)$
3. Average linkage: $d'(X,Y) = (|X|\cdot|Y|)^{-1}\sum_{x\in X}\sum_{y\in Y} d(x,y)$
4. Centroid linkage: $d'(X,Y) = d(\mu_X, \mu_Y)$ where μ_X, μ_Y are the means of X and Y, respectively.

4 Spectral Graph Clustering

So far, we've looked at datasets which consist of a set of points. Another way to formulate clustering problems is in the language of graph theory. Suppose we have a weighted graph $G = (V, E)$ where

$$w_{i,j} = \begin{cases} \text{weight of edge between node } i \text{ and node } j, & \text{if } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$

We wish to cut G into disjoint subgraphs G_1, G_2 of similar sizes, but without too cutting too much edge weight. One way to formalize this notion is to cut G into G_1, G_2 in such a way as to minimize the cut ratio, given by

$$\frac{\text{cut}(G_1,G_2)}{|V_1|\cdot|V_2|} \text{ where } \text{cut}(G_1,G_2) = \text{ total weight of cut edges}$$

This way, we want to simultaneously maximize the product of the new vertex sets, which will occur at an even split, and minimize the weight of the cut.

4.1 Transforming to an Optimization Problem

By introducing some notation, we can cast this combinatorial graph problem as an algebraic one. Letting $n = |V|$and defining $y \in \mathbb{R}^n$ to be an indicator vector given by

$$y_i = \begin{cases} 1, & \text{if node } i \in V_1 \\ -1, & \text{otherwise} \end{cases}$$

Then it follows that

$$\frac{w_{i,j}}{4}(y_i - y_j)^2 = \begin{cases} w_{i,j}, & \text{if } (i,j) \text{ was a cut edge} \\ 0, & \text{otherwise} \end{cases}$$

We can use this formulation to give a closed form for the cut ratio:

$$\text{cut}(G_1,G_2) = \sum_{(i,j)\in E} \frac{w_{i,j}}{4}(y_i - y_j)^2 = -\frac{1}{2}\underbrace{\sum_{(i,j)\in E} w_{i,j}y_iy_j}_{\text{diagonal terms}} + \frac{1}{4}\underbrace{\sum_{i=1}^{n} y_i \sum_{j\neq i} w_{i,j}}_{\text{off-diagonal terms}} = \frac{1}{4}y^\intercal L y \text{ where } L_{i,j} = \begin{cases} -w_{i,j}, & \text{if } i \neq j \\ \deg(v_i), & \text{otherwise} \end{cases}$$

where $w_{i,j}$ is zero if the vertices aren't adjacent and $\deg(v_i)$ = degree of vertex $i = \sum_{k\neq i} w_{i,k}$. The matrix $L \in \mathbb{R}^{n\times n}$ is symmetric, and is known as the *Laplacian matrix* for G. The Laplacian matrix of a graph is essentially a matrix representation of the graph, and is structured in such a way as to be useful in computing certain quantities about the graph, such as the number of spanning trees. The standard, more intuitive, definition of the Laplacian matrix L is the difference of the degree and adjacency matrices:

$$L = D - A \text{ where } D = \text{diag}(\deg(v_i), 1 \le i \le n) \text{ and } A_{i,j} = \begin{cases} w_{i,j}, & \text{if } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$

so D and A are the degree and adjacency matrices of G, respectively. The above shows that minimizing the weight of a cut is equivalent to minimizing the quadratic form $y^\intercal L y$, reducing the problem to one of matrix algebra. Moreover, if there are no negative weights, L is guaranteed to be positive semi-definite, which means $yLy^\intercal \ge 0$.
Notice that in the degenerate case where we don't make a cut at all, so that $y_i = 1, \forall i$, then since the cut has no weight,

clearly $\text{cut}(G_1, G_2) = 0$, which, assuming no edge weights are negative, gives the minimum value of zero for $yLy^\intercal$. Thus, the vector of all ones, $\mathbf{1}$, is an eigenvector of L, with eigenvalue zero. It turns out that if G is connected, this is the only eigenvalue; otherwise, L has one zero eigenvalue for every connected component of G.

Recall that while we want to minimize the weight of the cut, we also want to maximize the "equality" of the cut, formulated as the product of the vertex sets of the cut graphs (which is of course maximal at a square). Optimizing this constraint first, we can force our cut to allocate $\frac{n}{2}$ vertices to each of G_1, G_2, and then optimize over which cut to make. Given this formulation of the graph clustering problem, we can reduce what was once a combinatorial graph problem to the following optimization problem:

$$\min_{y \in \mathbb{R}^n} y^\intercal L y \text{ subject to } \forall i : y_i \in \{-1, 1\} \text{ and } \sum_{i=1}^{n} y_i = 0$$

The first constraint, called the *binary constraint*, encodes our definition of y as an indicator vector, while the second constraint, called the *balance constraint*, forces the vertex allocation to be symmetric as desired. Unfortunately, this problem is also NP-hard. One useful approximation used in practice is to relax the binary constraint. Of course, we wan't simply eliminate the constraint, as the trivial solution $y = \mathbf{0}$ would always be selected. One way to relax the constraint is to visualize the binary constraint as forcing y to be a vertex on the n-dimensional unit hypercube, so that we can relax the constraint by allowing y to be any point on the n-dimensional hypersphere (of radius $\sqrt{n}$) instead. Solving optimization problems with continuous constraints is often much easier than solving those with discrete constraints; we can temporarily allow for non-integral vertex allocations in y, find a solution, and round the vertex values to ± 1 afterwards, hoping that the resulting solution is still close to optimal.
Thus, the binary constraint now becomes $y^\intercal y = n$, forcing y to lie on the n-dimensional hypersphere of radius $\sqrt{n}$. Notice that the new optimization problem is equivalent to

$$\min_{y \in \mathbb{R}^n} \frac{y^\intercal L y}{y^\intercal y} \text{ subject to } \sum_{i=1}^{n} y_i = 0$$

which is to say, minimize the Rayleigh quotient of L and y subject to the constraint that $\mathbf{1}^\intercal y = 0$. Recall from our discussion on the Rayleigh quotient in the PCA section that the quantity is minimized by the eigenvector corresponding to the smallest eigenvalue. Since L has a zero eigenvalue, corresponding to $\mathbf{1}$, this is the minimizer, but this violates our balance constraint. The next best optimizer for a Rayleigh quotient is the eigenvector corresponding to the second smallest eigenvalue, known as the *Fiedler vector*, and we can simply round its components with a sweep cut to ± 1 to enforce the balance constraint.

4.2 Vertex Masses

In some applications, we may want to use a different definition of "graph equality" when partitioning the graph than simply the number of vertices in each sub-graph. Instead, we often want to prioritize certain vertices over others, similar to adding vertex weights in addition to edge weights. We can generalize the problem by introducing vertex masses, so that the cut ratio becomes

$$\frac{cut(G_1, G_2)}{\text{mass}(G_1) \cdot \text{mass}(G_2)} \text{ where } \text{mass}(G) = \sum_{i=1}^{n} \text{mass}(v_i)$$

Previously, every vertex had unit mass, so the mass of a graph was simply the size of its vertex set, but now we may assign certain vertices more mass than others. We can modify the corresponding optimization problem by introducing a mass matrix $M \in \mathbb{R}^{n \times n}$ for G, given by $M = \text{diag}(\text{mass}(v_i), 1 \leq i \leq n)$. Then the balance constraint becomes

$$\mathbf{1}^\intercal M y = 0$$

and the binary constraint becomes

$$y^\intercal M y = \text{mass}(G) = \text{Tr}(M)$$

Intuitively, the binary constraint relaxes the hypersphere constraint by allowing y to be anywhere on an axis-aligned n-dimensional ellipsoid. Whereas the solution to the previous optimization problem was the Fiedler vector of L, we now want the Fiedler vector of the generalized eigensystem

$$Lv = \lambda M v, \text{ for } v \in \mathbb{R}^n$$

Most algorithms for quickly computing the eigenvalues for symmetric matrices can be easily adapted to solve the above generalized system.

5 Latent Semantic Indexing