L16: Approximate PCA

Recall that PCA is the process to find the most dominant *directions* in an $(n \times d)$ matrix P (or n points in $\mathbb{R}^d$). We explored before using the SVD

$$[U, S, V] = \texttt{svd}(P)$$

where $U = [u_1, \ldots, u_n]$, $S = \mathsf{diag}(\sigma_1, \ldots, \sigma_d)$, and $V = [v_1, \ldots, v_d]$. Then $P = USV^T$ and in particular $P = \sum_{i=1}^{d} \sigma_i u_i v_i^T$. To approximate P we just use the first k components to find $P_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T = U_k S_k V_k^T$ where $U_k = [u_1, \ldots, u_k]$, $S_k = \mathsf{diag}(\sigma_1, \ldots, \sigma_k)$, and $V_k = [v_1, \ldots, v_k]^T$. The the vectors v_i (starting with smaller indexes) provide the best subspace representation of P.

But, although SVD has been *heavily* optimized on data sets that fit in memory (via LAPACK, found in Matlab, and just about every other language), it can sometimes be improved. Here we highlight two of these ways:

- to provide better interpretability of each v_i.
- to be more efficient on enormous scale, and in a stream.

The SVD takes $O(\min\{nd^2, d^2 n\})$ time to compute.

16.1 Column Sampling

The goal is to approximate P up to the accuracy of P_k. But in P_k the directions v_i are *linear combinations of features*.

- What is a linear combination of genes?
- What is a linear combination of typical grocery purchases?

Instead our goal is to choose V so that the columns of V are also columns of P.

For each column of $p^j \in P$, set $w_j = \|p_j\|^2$. Then select $t = (1/\varepsilon)^2 k \log k \log(1/\delta)$ columns of P, each proportional to w_j. Let the concatenation of these columns be C.

These t columns will jointly act in place of V_k. However since V was orthogonal, then the columns $v_i, v_j \in V_k$ were orthogonal. This is not the case for C, we need to orthogonalize C. Let $J_C = C(C^T C)^{-1} C^T$ be the projection matrix for C, so that $P_C = J_C P$ describes the *projection* of P onto the subspace of the directions spanned by C. Now

$$\|P - P_C\|_F \leq \|P - P_k\|_F + \varepsilon \|P\|_F$$

with probability at least $1 - \delta$.

- *Why did we not just choose the t columns of P with the largest w_j values?*
  Some may point along the same "direction" and would be repetitive. This should remind you of the choice to run k-means++ versus the Gonzalez algorithm for greedy point-assignment clustering.
- *Why did we not factor out the directions we already picked?*
  We could, but this allows us to run this in a streaming setting. (See next approach)

- *Can we get a better error bound?*
  Yes. First take SVD and then set weight $w_j = \|J_{U_k} p^j\|^2$ where $J_{U_k} = U_k(U_k^T U_k)^{-1} U_k^T$ projects a vector onto the first k singular vectors. Then the bounds are

  $$\|P - P_C\|_F \leq (1+\varepsilon)\|P - P_k\|_F.$$

  But this requires us to first take the SVD, so its is harder to do in a stream.

- *Can we also sample rows this way?*
  Yes. All tricks can be run on P^T the same way. And, both approaches can be combined. This is known as the CUR-decomposition of P.

A significant downside of these column sampling approaches is that the $(1/\varepsilon^2)$ coefficient can be quite large for a small error tolerance. If $\varepsilon = 0.01$, meaning 1% error, then this coefficient alone is $10{,}000$. In practice, the results may be better, but for guarantees, this may only work on very enormous matrices.

Note that the analysis explaining why this works can be seen, roughly, as a combination of the Coupon Collectors bound, saying we need to sample $k \ln k$ to hit all large directions, and then a Chernoff-Hoeffding bound (yield the $1/\varepsilon^2$ coefficient) to say we repeated this enough to even out the directions.

16.2 Frequent Directions

Another efficient solution is provided by using a Misra-Gries trick. It is called Frequent Directions (Liberty 2012).

We will consider a matrix P one row (one point p_j) at a time. We will maintain a matrix Q that is $\ell \times d$, that is it only has ℓ rows (directions).

We start with the first ℓ dimension p_j of P as Q. Then on each new dimension, we concatenate $Q_+ = \begin{bmatrix} Q \\ p_j \end{bmatrix}$. We set $[U, S, V] = \texttt{svd}(Q_+)$. Now examine $S = \mathsf{diag}(\sigma_1, \ldots, \sigma_{\ell+1})$, which is an $(\ell+1)$-square diagonal matrix. If $\sigma_{\ell+1} = 0$ (then p_j is in the subspace of Q), do nothing. Otherwise subtract $\hat{s} = \sigma_{\ell+1}^2$ from each (squared) entry in S, that is $\sigma_i' = \sqrt{\sigma_i^2 - \hat{s}}$ and in general $S' = \mathsf{diag}(\sqrt{\sigma_1^2 - \hat{s}}, \sqrt{\sigma_2^2 - \hat{s}}, \ldots, \sqrt{\sigma_\ell^2 - \hat{s}}, 0)$.

Now we set $Q = S'V^T$. Notice, that since S' only has non-zero elements in the first ℓ entries on the diagonal, then Q is at most rank ℓ and we can then treat V^T and Q as if the $(\ell+1)$th column does not exist.

Algorithm 16.2.1 Frequent Directions

Set Q all zeros $(\ell \times d)$ matrix.
for rows (i.e. points) $p_j \in P$ **do**
  Set $Q_+ = \begin{bmatrix} Q \\ p_j \end{bmatrix}$
  $[U, S, V] = \texttt{svd}(Q_+)$
  Set $\hat{s} = \sigma_{\ell+1}^2$ (the $(\ell+1)$ entry of S).
  Set $S' = \mathsf{diag}\left(\sqrt{\sigma_1^2 - \hat{s}}, \sqrt{\sigma_2^2 - \hat{s}}, \ldots, \sqrt{\sigma_\ell^2 - \hat{s}}, 0\right)$.
  Set $Q = S'V^T$ (keeping only the first ℓ rows, that last one should be all 0s).
return Q

The result of Algorithm 16.2.1 is a matrix Q such that for any (direction) unit vector $x \in \mathbb{R}^d$

$$0 \leq \|Px\|^2 - \|Qx\|^2 \leq 2\|A\|_F^2/\ell.$$

So setting $\ell = 2/\varepsilon$, then in any direction in $\mathbb{R}^d$, the squared mass in that direction is preserved up to $\varepsilon \|P\|_F^2$ (that is, ε times the total squared mass). Recall that $\|P\|_F = \sqrt{\sum_{p_i \in P} \|p_i\|^2} = \sqrt{\sum_{i \in [n]} \sum_{j \in [d]} P_{i,j}^2}$.

- *Why does this work?*
  Just like with Misra-Greis, when something mass is deleted from one, counter it is deleted from all ℓ counters, and none can be negative. So here when one direction is decreased its (squared) mass, all ℓ directions (with non-zero squared mass) are decreased. So no direction can have more than ε fraction of the total squared mass decreased from it.

  Finally, since squared mass can be summed independently along any set of orthogonal directions, we can subtract each of them without affecting others.

- *Why do we use the* `svd`*?*
  Because we can choose any orthogonal basis, we find the one that has the smallest $\hat{s}$ value to decrease by. This is what the `svd` gives us.

- *Did we **need** to use the* `svd`*? (its expensive, right)?*
  Well, we only need to run it on a set matrix of size $\ell \times d$, so $d\ell^2$ might not be too bad ... although this is repeated n times. We can decrease the total runtime to $O(nd\ell)$ with some tricks.

  But maybe we don't need it. The same rough analysis goes through with any basis. Instead we could only consider directions along actual points. This may not give as good of bounds in practice, but may have other advantages in sparsity and runtime.

- *What happened to U in the* `svd` *output?*
  The matrix U just related the main directions to each of the n points (rows) in P. But we don't want to keep around the space for this. In this application, we only care about the directions or subspace that best represents the points.