Clustering with Gradient Descent

compiled by Alvin Wan from Professor Benjamin Recht's lecture

1 Performance

We explored SVD. Note that in practice, you should *not* compute XX^T, as this squares all singular values, leading to higher instability. The built-in `scipy.linalg.svd` has the same runtime as an explicit computation $O(dn^2 + d^3)$ but modifies the algorithm so that the results are more stable.

We also explored k-means clustering. Again, our objective is $\textsc{Minimize}_{\mu_1,\ldots\mu_T} \sum_{i=1} \textsc{Min}_{1\leq j\leq k} \|X_i - \mu_{j_i}\|^2$. Note that Lloyd's algorithm might find local minima, and the complexity per iteration is $O(ndk + nd)$.

Finally, we explored spectral clustering, where our goal was to $\textsc{Minimize} cut(V_1, V_2)$ s.t. $|V_1| = |V_2|$. This is the equivalent of $\textsc{Minimize}\frac{1}{4}v^T Lv$ s.t. $v_i \in \{-1, 1\}, 1^T v = 0$. The solution to our latter form can be approximated using the solution to $\textsc{Minimize}\frac{1}{4}v^T Lv$ s.t. $\|v\| = \sqrt{n}, 1^T v = 0$. This always gives a response but it only gives a good answer if there actually exists clusters in your data. The solution to this is the second eigenvalue, with an $L \times n$ matrix, meaning the runtime is $O(n^3)$.

In this note, we will see how gradient descent can be applied to each of these problems, so that we get iterative algorithms for each of these.

2 SVD with Gradient Descent

In SVD, our goal is to factor X into $U\Lambda V^T$. Recall that an analogous objective

$$\textsc{Minimize}_{A\in\mathbb{R}^{r\times d}, B\in\mathbb{R}^{r\times n}} \|X - A^T B\|_F^2$$

, where the above objective has the solutions $A = \Lambda^{1/2}U^T, B = \Lambda^{1/2}V^T$. Let us rewrite the objective function using the column vectors of A and column vectors of B.

$$A = \left[a_1, \ldots, a_d\right], B = \left[b_1, \ldots b_n\right]$$
$$\text{Minimize} \sum_{i=1}^{d} \sum_{j=1}^{n} (X_{ij} - a_i^T b_j)^2$$

Take the gradient of the objective to obtain the following, w.r.t. a and w.r.t. b.

$$\nabla_a \{(X_{ij} - a_i^T b_j)^2\} = -(X_{ij} - a_i^T b_j) b_j$$
$$\nabla_b \{(X_{ij} - a_i^T b_j)^2\} = -(X_{ij} - a_i^T b_j) a_i$$

We can fit this to gradient descent. Fix step size $t \neq 0$ and initialize A, B.

1. Sample Y_{ij}
2. Set $e = X_{ij} - a_i^T b_j$.
3. $a_i \leftarrow a_i + teb_j, b_j \leftarrow b_j + tea_i$.
4. Repeat.

Computing e, a_i, and b_j all take $O(r)$, so the complexity per iteration is $O(r)$. There are nd total entries, but interestingly, $O((n+d)r)$ iterations often will suffice. Turns out the proof is quite complicated, so we will omit it.

What if most entries are missing? We can run SGD on the fully-observed entries to achieve *matrix completion*.

3 K-means Clustering

Our algorithm is even simpler.

1. Pick i.
2. Find μ_j closest to x_i.
3. $\mu_j = (1-t)\mu_j + tx_i$.

As it turns out, $\nabla \textsc{Min}_i f_i(x) = \nabla f_i(x)$ In other words, the gradient of the minimum $f_i(x)$ is the equivalent of the gradient at a randomly-selected f_i.

4 Spectral Clustering

Our goal is to rid of our constraints, else the efficiency of SGD decreases. Run gradient descent. Recall that

$$L_{ij} = \begin{cases} \sum_j a_{ij} & i = j \\ -a_{ij} & \text{o.w.} \end{cases}$$

1. Initialize v s.t. $1^T v = 0$, so that $v = randn(n), v \leftarrow v - (1^T v)1$.
2. project: $v \leftarrow \sqrt{n}\frac{v}{\|v\|}$.
3. gradient step: $v \leftarrow v - \frac{t}{2}Lv$.

This is called the **projected gradient algorithm**[1]. In short, we project onto the unit ball. Take a gradient descent, and then repeat. The complexity is the number of nonzero entries in L. If L is extremely sparse, the weight update for projected gradient is extremely fast. On the other hand, an extremely dense graph will see a runtime approaching $O(n^2)$, and a graph where the number of neighbors for any node is bounded by a constant k, then the runtime can be $O(n)$.

How do you choose t? Decrease t until you no longer have `nans`. How do you choose stopping criteria? Run for 1-200 epochs. After a number of epochs, set $t \leftarrow \beta t$ where $\beta < 1$. Common values are $\beta = 0.9, 0.8, 0.1$.

5 Non-Negative Matrix Factorization

We have familiar algorithms for this problem. For each weight update, simply take the maximum of 0 with the new value. For example, we could modify the iterative algorithm for SVD to be the following:

1. Sample Y_{ij}

[1]To see a derivation of the projected gradient algorithm, see http://www.stats.ox.ac.uk/ lienart/blog_opti_pgd.html

2. Set $e = X_{ij} - a_i^T b_j$.
3. $a_i \leftarrow \textsc{Max}(a_i + teb_j, 0), b_j \leftarrow \textsc{Max}(b_j + tea_i, 0)$.
4. Repeat.

Our objective function for this problem is formally the following.

$$\textsc{Min}\|X - A^T B\| \text{ s.t. } A \geq 0, B \geq 0$$

Why would we want non-negative A, B? Consider a term-document matrix. We note that this matrix is effectively a bag-of-words model, where each entry is the number of occurrences of a particular term in a particular document. These entries are strictly non-negative, and likewise, A, B should not be negative: A represents the topics, and B represents the weights, how much of each topic is in each document. As it turns out, this problem is NP-hard.

6 Page Rank (Optional)

Create a new matrix H, where

$$H_{ij} = \begin{cases} 1 & i \text{ links to } j \\ 0 & \text{o.w.} \end{cases}$$

We can take the out-degree n_i and create $\hat{H_{ij}} = \frac{1}{n_i} H_{ij}$. Take $\vec{p}$ to be the probability of landing on a page. The eigenvector $p = \hat{H}p$ gives us the most popular webpages on the internet.

$$\textsc{Maximize} p^T \hat{H} p \text{ s.t. } p \geq$$

As it turns out, the largest eigenvalue of H is 1, by the Perron-Frobenius theorem. Projected gradient descent allows us to find this vector in linear time.