Assignment 3
CS260B: Algorithmic Machine Learning, Spring 2022
Due: May 11, 10PM

Guidelines for submitting the solutions:

- The assignments need to be submitted on Gradescope. Make sure you follow all the instructions - they are simple enough that exceptions will not be accepted.
- Start each problem or sub-problem on a separate page even if it means having a lot of whitespace and write/type in large font.
- The solutions need to be submitted by 10 PM on the due date. No late submissions will be accepted.
- Please adhere to the code of conduct outlined on the class page.

1. The goal of this exercise is to partially show one of the remarkable properties of SVD that we discussed in class. Let X be a $n \times d$ matrix and let $X = U\Sigma V^T$ be its singular-value decomposition where Σ is a $r \times r$ diagonal matrix with $\Sigma_{11} \geq \Sigma_{22} \geq \ldots \geq \Sigma_{rr}$. Let $\sigma_i = \Sigma_{ii}$. Show that $\max_{v:\|v\|=1} \|Xv\| \leq \sigma_1$.[1] [4 points]

   [Hint: Write a vector v in an orthonormal basis that contains the columns of V, i.e., write $v = \sum_i \alpha_i v_i$. What is Xv given this expansion? Use this to compute $\|Xv\|^2$ and bound it.]

   Soluton. Let $v_1, v_2, \ldots, v_r$ be the singular vectors of M. We can extend them to $v_{r+1}, \ldots, v_d$ to form an orthnormal basis in case $r < d$. Now, write $v = \sum_i \alpha_i v_i$ in this orthnormal basis. So, $Mv = \sum_{i=1}^r \alpha_i \sigma_i u_i$ as we saw in class. Now, $\|Mv\|^2 = \sum_{i=1}^r \alpha_i^2 \sigma_i^2$ (as u_i's are orthonormal). Therefore, $\|Mv\|^2 \leq \sigma_1^2 (\sum_{i=1}^r \alpha_i^2) \leq \sigma_1^2 \|v\|^2$, as required.

2. In class we showed that the span of the first two right singular vectors gives a best-fit subspace of dimension 2. Generalize the argument to higher k. That is, show that for any X, the span of the first k right singular vectors gives a solution to the best-fit subspace problem for dimension k. You have to use this without using any properties of the SVD (even those we stated in class). [4 points]

   [Hint: Use induction. Suppose the claim is true for $k-1$ and prove it for k. Recall that in the argument for $k = 2$, we had to choose an orthonormal basis w_1, w_2 for an optimal subspace S_* carefully so that w_2 was perpendicular to v_1. You can use a generalization of this to higher k; you don't have to prove this last fact.]

   Solution. 2. The proof is Theorem 3.1 in the book by Blum, Hopcroft and Kannan.

[1] Provide a complete proof without circularly referring to the two main theorems we stated about SVD!

3. In class we saw a way to compute the singular vector corresponding to the largest singular value without computing the entire SVD. What if you had to compute the smallest singular vector? Describe a way to compute the smallest right singular vector of a matrix X by a reduction. [3 points]

   You don't have to prove anything just an idea about what to do will suffice.

   [Hint: It might be to best to work with the matrix $Y = X^T X$. If X had SVD $X = U\Sigma V^T$, then $Y = V\Sigma^2 V^T$ and the eigenvalues of Y are exactly the squares of the singular values of X. Perform a suitable shift on Y to get the smallest singular vector of X.]

   Solution. Let's say you first compute the largest singular value of X using the power iteration. Call this σ_1. Now, consider the matrix $Y' = 2\sigma_1 I - Y$. (The factor 2 is arbitrary really.). As we discussed in class, $Y' = 2\sigma_1 I - V\Sigma^2 V^T = V(2\sigma_1 I - \Sigma^2)V^T$. So the singular vectors of Y' from largest to smallest are the smallest to largest singular vectors of Y (and hence of X). We can now use power iteration to compute the largest singular vector of Y' to get the smallest singular vector of X.

4. The goal of this exercise is to implement the singular value projection idea we saw in class.

   Let X be a unknown $n \times d$ matrix of rank at most k (a small number). We are given as input $((X_{ij} : (i,j) \in O))$ for some subset of entries $O \subseteq [n] \times [d]$. Your goal is to figure out the missing entries in X.

   For any matrix $Y \in n \times d$, define $L(Y) = \sum_{(i,j)\in O}(X_{ij} - Y_{ij})^2$. (The loss function is computable as we see the entries in Y).

   (a) Describe how to compute the gradient of L with respect to Y. [1 point].

   (b) You can use the code from edstem for the following exercise. Generate a random rank k $n \times d$ matrix where $k = 5$, $n = 1000$, $d = 500$. Note that technically, writing down the matrix completely will need you to write down $500,000$ entries!

   Generate a random subset O of $[n] \times [d]$ of density $p = .1$. One way to do this is in python to generate a random $n \times d$ matrix with entries in the range $(0, 1)$ and set all entries bigger than p to 0 and those less than p to 1.

   Now use SVP algorithm as discussed in class for a couple of step-sizes and plot the cost function of each iteration. Your submission must be (a) A screenshot of your code for computing the projected gradient and (b) The final plot of the error. [3 points]

   (See the workspace on power iteration for helper code on generating the indices of observed entries. You have to implement the cost function, the gradient function, and call an internal SVD computation to compute the top 5 SVD (e.g., `scipy.sparse.linalg.svds(Y, k = 5)`) for each projection step. Note that you can use the earlier code we had for demoing GD here except you have to change the input handle for gradient to the projected one.)

   You might have noticed that the above method converges to the true unknown X impressively even with only 10% of entries visible. However, you can in fact do much better in terms of implementation (and you need to when computing at the scale of the Netflix dataset). Lets say X_t is the current low-rank matrix approximation candidate after t'th iteration. Can you think of ways to speed up the computation of the best rank k approximation to $Y_t = X_t - \eta\nabla L(X_t)$? Indeed writing down Y_t entirely itself will be prohibitive when you have $n = 500000$ and $d = 17000$. Instead, you can store just the low-rank factorization of X_t. Also note that $\nabla L(X_t)$ will be a sparse matrix (only 1% of its entries are non-zero). You can use these to properties to compute matrix-vector products of Y_t faster than even writing down the entire matrix. Indeed, we had to use such implementations in our paper from 2009.

Let us investigate what speed-up you might gain by the above approach by looking at a simpler setting in the next problem.

5. Let n be a large number (say $n = 10,000$) and k a small number (say 20). Let G be a sparse $n \times n$ matrix with density p (say 0.01). MATLAB, NUMPY (and other numerical analysis software) can exploit the sparsity of X to compute the top singular vector quite efficiently. Next, let U be a $n \times k$ matrix U and let $X = UU^T$. Computing the first right singular vector of X directly on MATLAB would be quite slow. But, the first right singular vector of X is the same as the first left singular vector of U which you can compute much more efficiently.

   Now, suppose we had to compute the first singular vector of $Z = X + G$ (This is similar to the situation in SVP). Doing so directly would be quite expensive. But we can use power iteration faster as for any vector v, we can compute $Zv = U(U^T v) + Gv$ without ever having to even write down Z (but only keeping U, G).

   You can use MATLAB, R, Numpy/Scipy, or Julia for this exercise. Generate a random $n \times n$ sparse matrix G with density p for $n = 10,000$, $p = 0.01$. Generate a random $n \times k$ matrix U. Compute $Z = UU^T + G$.

   Next, use the built-in command to compute the top singular vector and time the operation. Implement the power iteration (as in the above step) to compute the top singular vector of Z (say for $10, 20, 30, ..., 100$ iterations) and time the algorithm. Is there a difference in speed?

   Your submission should be the following items. [3 points]

   (a) A table or figure showing the run-times of the in-built command and the run-times for different number of iterations obtained by averaging over 10 runs (as X, U are random - it is better to consider multiple runs).

   (b) A figure plotting the number of iterations (on the x-axis) with the error (what power iteration computes and the true answer as computed by the built-in command) on the y-axis.

Solution. The main idea is to apply power iteration as matrix-vector products with Z can be computed with writing down Z fully: $Zv = U(U^T v) + Gv$. So here is one way to do it:

- Choose $v = v_0$ to be a random unit vector.
- For $t = 1, \ldots, T$:
  1. Set $u = U(U^T v) + Gv$ (you have to implement multiplication it in this order).
  2. Set $v = U(U^T u) + G^T u$.
  3. Set $v = v/\|v\|$.