This homework is due February 22, at Noon.

1. Homework process and study group

Who else did you work with on this homework? List names and student ID’s. (In case of hw party, you can also just describe the group.) How did you work on this homework?

Solution: I worked on this homework with...

I first worked by myself for 2 hours, but got stuck on Problem 5 so I went to office hours on...

Then I went to homework party for a few hours, where I finished the homework.

2. Lecture Attendance

This question is a student trust-based system for giving credit to those who attend lecture and help the course out by doing so. Lying on this (or any other part of the homework) constitutes academic dishonesty, and more importantly than any academic sanctions, lying would damage your honor and integrity. Be honest. You will carry your honor and integrity with you for the rest of your life, and they are way more important than your GPA.

Did you attend live lecture this week? (the week you were working on this homework) What was your favorite part? Was anything unclear? Answer for each of the subparts below. If you only watched on YouTube, write that for partial credit.

(a) Monday lecture
(b) Wednesday lecture
(c) Friday lecture

Solution: Full credit for attending live lecture and giving a comment (what you liked best, what was unclear) about that lecture. 8 points for attending live lecture but giving no comment. 5 points for watching on YouTube and giving a comment. 2 points for just watching on YouTube. 0 points for blank or not watching lecture at all.

3. The Moore-Penrose Pseudoinverse for “Fat” Matrices

Say we have a set of linear equations described as $A\vec{x} = \vec{y}$. If $A$ is invertible, we know that the solution is $\vec{x} = A^{-1}\vec{y}$. However, what if $A$ is not a square matrix? In 16A, you saw how this problem could be approached for tall matrices $A$ where it really wasn’t possible to find a solution that exactly matches all the measurements. The Linear Least-Squares solution gives us a reasonable answer that asks for the “best” match in terms of reducing the norm of the error vector.

This problem deals with the other case — when the matrix $A$ is short and fat. In this case, there are generally going to be lots of possible solutions — so which should we choose? Why? We will walk you through the Moore-Penrose Pseudoinverse that generalizes the idea of the matrix inverse and is derived from the singular value decomposition. You have already seen these ideas in the MIMO wireless problem in the last homework, this time, we will do it more generally.
(a) Say you have the following matrix.

\[
A = \begin{bmatrix}
1 & 1 & 1 \\
1 & -1 & 1 \\
\end{bmatrix}
\]

Calculate the SVD decomposition of \( A \). That is to say, calculate \( U, \Sigma, V \) such that,

\[
A = U \Sigma V^T
\]

What are the dimensions of \( U, \Sigma \) and \( V \)?

**Note.** Do NOT use a computer to calculate the SVD. You may be asked to solve similar questions on your own in the exam.

**Solution:**

First, note that, \( A^T A = \begin{bmatrix} 2 & 0 & 2 \\ 0 & 2 & 0 \\ 2 & 0 & 2 \end{bmatrix} \). Since the first and last column are identical, it has two linearly dependent columns and thus we can always find a non-zero solution to the following equation:

\[
A^T A \vec{v} = \vec{0}
\]

which means that the matrix has an eigenvalue of 0. It is clear that a vector that solves the above equation is \([1, 0, -1]^T\), which can be normalized to be \(\frac{1}{\sqrt{2}}[1, 0, -1]^T\). Since the other eigenvectors need to be orthogonal to it, some natural candidates include \([0, 1, 0]^T\), \([1, 0, 1]^T\), and so on. We can check and verify that they are indeed eigenvectors, corresponding to eigenvalues of 2 and 4, respectively. Therefore, we have \( \lambda_0 = 4, \lambda_1 = 2, \lambda_2 = 0 \) as the eigenvalues, and the corresponding eigenvectors \( \vec{v}_0 = \frac{1}{\sqrt{2}}[1, 0, 1]^T, \vec{v}_1 = [0, 1, 0]^T, \vec{v}_2 = \frac{1}{\sqrt{2}}[1, 0, -1]^T \) respectively. Then, noting that the singular values are the square roots of the eigenvalues, we get,

\[
\Sigma = \begin{bmatrix} 2 & 0 & 0 \\ 0 & \sqrt{2} & 0 \end{bmatrix}, \quad V = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & 1 & 0 \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{bmatrix}
\]

We can then solve for \( U \) by noting that

\[
U \Sigma = [2\vec{u}_0 \quad \sqrt{2}\vec{u}_1 \quad \vec{0}] = AV = \begin{bmatrix} \sqrt{2} & 1 & 0 \\ \sqrt{2} & -1 & 0 \end{bmatrix}
\]

From the above equation, we can see that \( \vec{u}_0 = [\sqrt{2}\frac{1}{\sqrt{2}}, \frac{\sqrt{2}}{\sqrt{2}}]^T \), and \( \vec{u}_1 = [\sqrt{2}\frac{1}{\sqrt{2}}, -\frac{\sqrt{2}}{\sqrt{2}}]^T \). This gives us,

\[
U = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}
\]

So the SVD decomposition is

\[
A = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & 1 & 0 \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{bmatrix}
\]
(b) Let us think about what the SVD does. Let us look at matrix $A$ acting on some vector $\vec{x}$ to give the result $\vec{y}$. We have,

$$A\vec{x} = U\Sigma V^T \vec{x} = \vec{y}$$

Observe that $V^T \vec{x}$ rotates the vector, $\Sigma$ scales it and $U$ rotates it again. We will try to "reverse" these operations one at a time and then put them together.

If $U$ “rotates” the vector $(\Sigma V^T) \vec{x}$, what operator can we derive that will undo the rotation?

**Solution:** By orthonormality, we know that $U^T U = UU^T = I$. Therefore, $U^T$ undoes the rotation.

(c) Derive an matrix that will "unscale", or undo the effect of $\Sigma$ where it is possible to undo. Recall that $\Sigma$ has the same dimensions as $A$. Ignore any division by zeros (that is to say, let it stay zero).

**Solution:** If you observe the equation:

$$\Sigma \vec{x} = \vec{y}, \quad (2)$$

you can see that $\sigma_i x_i = y_i$ for $i = 0, \ldots, m-1$, which means that to obtain $x_i$ from $y_i$, we need to multiply $y_i$ by $\frac{1}{\sigma_i}$. For any $i > m-1$, the information in $x_i$ is lost by multiplying with 0. Therefore, the reasonable guess for $x_i$ is 0 in this case. That’s why we padded 0s in the bottom of $\tilde{\Sigma}$ given below:

If $\Sigma = \begin{bmatrix} \sigma_0 & 0 & 0 & 0 & 0 & \ldots & 0 \\ 0 & \sigma_1 & 0 & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & 0 & \sigma_{m-1} & 0 & \ldots & 0 \end{bmatrix}$

then $\tilde{\Sigma} = \begin{bmatrix} \frac{1}{\sigma_0} & 0 & \ldots & 0 \\ 0 & \frac{1}{\sigma_1} & \ldots & 0 \\ \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & \frac{1}{\sigma_{m-1}} \\ 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & 0 \end{bmatrix}$

(d) Derive an operator that would "unrotate" by $V^T$.

**Solution:** By orthonormality, we know that $V^T V = VV^T = I$. Therefore, $V$ undoes the rotation.

(e) Try to use this idea of "unrotating" and "unscaling" to derive an "inverse" (which we will use $A^\dagger$ to denote). That is to say,

$$\vec{x} = A^\dagger \vec{y}$$

The reason why the word inverse is in quotes (or why this is called a pseudo-inverse) is because we’re ignoring the "divisions" by zero.

**Solution:** We can use the unrotation and unscaling matrices we derived above to "undo" the effect of $A$ and get the required solution. Of course, nothing can possibly be done for the information that was destroyed by the nullspace of $A$ — there is no way to recover any component of the true $\vec{x}$ that was in the nullspace of $A$. However, we can get back everything else.

$$\vec{y} = A \vec{x} = U\Sigma V^T \vec{x}$$

Unrotating by $U$

$$\tilde{\Sigma} U^T \vec{y} = \Sigma V^T \vec{x}$$

Unscaling by $\tilde{\Sigma}$

$$V \tilde{\Sigma} U^T \vec{y} = \vec{x}$$

Unrotating by $V$

Therefore, we have $A^\dagger = V \tilde{\Sigma} U^T$, where $\tilde{\Sigma}$ is given in (c).

(f) Use $A^\dagger$ to solve for $\vec{x}$ in the following systems of equations.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix} \vec{x} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$
Solution: From the above, we have the solution given by:

\[
\vec{x} = A^\dagger \vec{y} = V \tilde{\Sigma} U^T \vec{y}
\]

\[
\begin{bmatrix}
\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\
0 & 1 & 0 \\
\frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}}
\end{bmatrix}
\begin{bmatrix}
\frac{1}{2} & 0 \\
0 & \frac{\sqrt{2}}{2} \\
\frac{1}{\sqrt{2}} & 0
\end{bmatrix}
\begin{bmatrix}
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}
\end{bmatrix}
\begin{bmatrix}
2 \\
4
\end{bmatrix}
\]

Therefore, the solution to the system of equations is:

\[
\vec{x} = \begin{bmatrix}
\frac{3}{2} \\
-1 \\
\frac{3}{2}
\end{bmatrix}
\]

(g) (optional) Now we will see why this matrix is a useful proxy for the matrix inverse in such circumstances. Show that the solution given by the Moore-Penrose Psuedoinverse satisfies the minimality property that if \( \vec{\hat{x}} \) is the psuedo-inverse solution to \( A \vec{x} = \vec{y} \), then \( \| \vec{\hat{x}} \| \leq \| \vec{z} \| \) for all other vectors \( \vec{z} \) satisfying \( A \vec{z} = \vec{y} \).

*Hint: look at the vectors involved in the V basis. Think about the relevant nullspace and how it is connected to all this.*

This minimality property is useful in both control applications (as you will see in the next problem) and in communications applications (as you saw in the MIMO wireless problem on the last HW).

Solution: Since \( \vec{\hat{x}} \) is the pseudo-inverse solution, we know that,

\[
\vec{\hat{x}} = V \tilde{\Sigma} U^T \vec{y}
\]

Let us write down what \( \vec{\hat{x}} \) is with respect to the columns of \( V \). Let there be \( k \) non-zero singular values. The following expression comes from expanding the matrix multiplication.

\[
\vec{\hat{x}} \bigg|_V = V^T \vec{\hat{x}}
\]

\[
= V^T A^\dagger \vec{y} = V^T V \tilde{\Sigma} U^T \vec{y} = \tilde{\Sigma} U^T \vec{y}
\]

\[
= \begin{bmatrix}
\langle \vec{y}, \vec{u}_0 \rangle / \sigma_0 , \\
\langle \vec{y}, \vec{u}_1 \rangle / \sigma_1 , \\
\vdots \\
\langle \vec{y}, \vec{u}_{k-1} \rangle / \sigma_{k-1} , \\
0 , \\
\ldots , \\
0
\end{bmatrix}^T
\]

The \( n - k \) zeros at the end come from the fact that there are only \( k \) non-zero singular values. Therefore, by construction, \( \vec{\hat{x}} \) is a linear combination of the first \( k \) columns of \( V \).

Since any other \( \vec{z} \) is also a solution to the original problem, we have

\[
A \vec{z} = U \Sigma V^T \vec{z} = U \Sigma \vec{z}_V = \vec{y},
\]

where \( \vec{z}_V \) is the projection of \( \vec{z} \) in the \( V \) basis. Using the idea of “unscaleing” for the first \( k \) elements (where the unscaleing is clearly invertible) and “unrotation” after that, we see that the first \( k \) elements of \( \vec{z} \big|_V \) must be identical to those first \( k \) elements of \( \vec{x} \big|_V \).
However, since the information for the last \( n - k \) elements of \( \mathbf{z}_V \) is lost by multiplying 0s, any values \( \alpha_i \) there are unconstrained as weights on the last part of the \( V \) basis — namely the weights on the basis for the nullspace of \( A \). Therefore,

\[
\mathbf{z}_V = \left[ \frac{\langle \mathbf{y}, \mathbf{u}_0 \rangle}{\sigma_0}, \frac{\langle \mathbf{y}, \mathbf{u}_1 \rangle}{\sigma_1}, \ldots, \frac{\langle \mathbf{y}, \mathbf{u}_{k-1} \rangle}{\sigma_{k-1}}, \alpha_k, \alpha_{k+1}, \ldots, \alpha_{n-1} \right]^T.
\]

Now, since the columns of \( V \) are orthonormal, observe that,

\[
||\mathbf{x}||^2 = \sum_{i=0}^{k-1} \left| \frac{\langle \mathbf{y}, \mathbf{u}_i \rangle}{\sigma_i} \right|^2
\]

and that,

\[
||\mathbf{z}||^2 = \sum_{i=0}^{k-1} \left| \frac{\langle \mathbf{y}, \mathbf{u}_i \rangle}{\sigma_i} \right|^2 + \sum_{i=k}^{n-1} |\alpha_i|^2
\]

Therefore,

\[
||\mathbf{z}||^2 = ||\mathbf{x}||^2 + \sum_{i=k}^{n-1} |\alpha_i|^2
\]

This tells us that,

\[
||\mathbf{z}|| \geq ||\mathbf{x}||
\]

4. Inverse Kinematics

Inverse Kinematics is critical in robotics, control, and computer graphics applications. We need to be able to go backward from what we want to have happen in the real (or virtual) world to how to set parameters.

Suppose you have a robotic arm composed of several rotating joints.

The lengths \( r_i \) of the arm are fixed, but you can control the arm by specifying the amount of rotation \( \theta_i \) for each joint. If we have an arm with four joints, it can be parameterized by:

\[
\mathbf{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}.
\]
Suppose further that we have some target $\vec{t} \in \mathbb{R}^2$, which represents a point in the 2D space, and we would like for the end of the arm, called the end effector, to reach to the target. We have some function $\vec{f}(\vec{\theta})$ that rotates each joint of the arm according to the input and returns the position of the end effector. Figure [1] shows a visualization of an arm rotated by $\vec{\theta}$. To make the arm reach for the target $\vec{t}$, we want to find where the function

$$\vec{g}(\vec{\theta}) = \vec{f}(\vec{\theta}) - \vec{t}$$

is equal to $\vec{0}$. To accomplish this, we use Newton’s method for solving potentially nonlinear equations.

You might have seen Newton’s method in your calculus course in the 1-d case. It follows an iterative approach where you have a real function $g$ of a single parameter $\theta$ and we want to find a $\vec{\theta}$ so that $g(\vec{\theta}) = 0$. We take the current guess $\theta^{(i)}$ and then approximate the function $g$ by the best linear approximation in terms of $\theta$. This approximation is $g(\theta^{(i)}) + g'(\theta^{(i)})(\theta - \theta^{(i)})$. We then ask for the $\theta$ that sets this approximation to zero, and this is our next guess. Doing this involves finding an increment $\theta - \theta^{(i)}$ so that when we transform it by the derivative $g'(\theta^{(i)})$, it gives us $-g(\theta^{(i)})$. This requires finding the “inverse” of the derivative. The iteration goes like this: guess a configuration $\theta$, compute the function there and the first-order approximation to the function, update to a new configuration $\theta$ based on that approximation, and then repeat computing the function and the first-order approximation.

This is reminiscent of the k-means iteration where we guess centers, compute clusters based on the centers, and then recompute centers based on those clusters and iterate. And like k-means, there are issues with local minima, etc. As a technicality, in practical Newton’s iterations we often add a step-size parameter that makes us move in the direction that the first-order approximation to the function suggests, but not all the way. (The counterpart in k-means would be to move the cloud centers, but not all the way to the averages.) This is done because the derivative of the function there might be very different there than it is here.

While you might have seen Newton’s method as described above in your calculus courses, you might not have seen the vector-generalization of it. It follows exactly the same spirit. The first-order approximation to the vector valued function $\vec{g}(\vec{\theta})$ at $\vec{\theta}^{(i)}$ is now $\vec{g}(\vec{\theta}^{(i)}) + \vec{J}_g(\vec{\theta}^{(i)})(\vec{\theta} - \vec{\theta}^{(i)})$ where $\vec{J}_g(\vec{\theta})$ is the Jacobian matrix of the function $\vec{g}(\vec{\theta})$. For this problem, we will be using a robotic arm with 4 joints in a 2-dimensional space. Therefore, the Jacobian of $\vec{g}(\vec{\theta})$ will be a 2x4 matrix, and it is computed by calculating the partial derivatives of $\vec{g}(\vec{\theta})$:

$$\vec{J}_g = \begin{bmatrix} \frac{\partial g_x(\vec{\theta})}{\partial \theta^1} & \frac{\partial g_x(\vec{\theta})}{\partial \theta^2} & \frac{\partial g_x(\vec{\theta})}{\partial \theta^3} & \frac{\partial g_x(\vec{\theta})}{\partial \theta^4} \\ \frac{\partial g_y(\vec{\theta})}{\partial \theta^1} & \frac{\partial g_y(\vec{\theta})}{\partial \theta^2} & \frac{\partial g_y(\vec{\theta})}{\partial \theta^3} & \frac{\partial g_y(\vec{\theta})}{\partial \theta^4} \end{bmatrix}.$$

In this notation, we use $\vec{g}(\vec{\theta}) = [g_x(\vec{\theta}) \quad g_y(\vec{\theta})]^T$ where $g_x(\vec{\theta})$ is the x coordinate of the end effector and $g_y(\vec{\theta})$ is the y coordinate in our 2D space. There is nothing mysterious about this, if you think about it, this matrix of partial derivatives (a partial derivative is just a regular derivative with respect to a particular variable, treating all the other variables as constants) is the natural candidate for how small changes in the angles $\theta_i$ impact the position of the end-effector. Numerically, they are computed by literally taking a small difference and seeing what happens.

The Newton algorithm in this case is an iterative method that gives us successively better estimates for our vector $\vec{\theta}$. If we start with some guess $\vec{\theta}^{(i)}$, then the next guess is given by

$$\vec{\theta}^{(i+1)} = \vec{\theta}^{(i)} - \eta \vec{J}_g^{-1}(\vec{\theta}^{(i)})\vec{g}(\vec{\theta}^{(i)})$$

(7)

where $\eta$ is adjusted to determine how large of a step we make between $\vec{\theta}^{(i)}$ and $\vec{\theta}^{(i+1)}$. Notice that we need to invert the Jacobian matrix of first-partial-derivatives, and this matrix is not square. It is in fact a fat matrix. Fortunately, another problem on this homework has helped us understand how to invert fat matrices. The minimality property of the psuedoinverse is useful because we would rather take small steps than big ones.
And when tracking a moving reference, we’d like to have the joint angles change in a minimal way rather than in some very convoluted fashion.

The following problem will guide you step-by-step through the implementation of the pseudoinverse. The three steps of the algorithm is first, compute the SVD of the input matrix. Next, we compute \( \Sigma^{-1} \) by inverting each singular value \( \sigma_i \). Finally, we compute the pseudoinverse by multiplying the matrices together.

There are three test cases that you can use to determine if your pseudoinverse function works correctly. In the first case, the arm is able to reach the target, and the end of the arm will be touching the target. In the second case, the arm should be pointing in a straight line towards the blue circle. The last case is the same as the second with the addition that a singular value will be very close to zero to test your pseudoinverse function’s ability to handle small singular values. There is also an animated test case that will move the target in and out of the reach of the arm. Verify that the arm follows the target correctly and points towards the target when it is out of reach.

(a) In the “pseudoinverse” function, compute the SVD of the input matrix \( A \) by using the appropriate NumPy function.

Solution: See the IPython notebook for the solution.

(b) To save space, the NumPy algorithm returns the matrix \( \Sigma \) as a one-dimensional array of the singular values. Use this vector to compute the diagonal entries of \( \Sigma^{-1} \). In the computation, you will need to threshold the singular values and invert only the singular values above the threshold to avoid dividing by zero or a very small number.

Solution: See the IPython notebook for the solution.

(c) We now have all of the parts to compute the pseudoinverse of \( A \). Add this computation to the function.

Solution: See the IPython notebook for the solution.

5. Low Rank Approximation of a Matrix

In this question we will study the so called “low rank approximation” problem. As the name implies, consider an arbitrary matrix \( A \in \mathbb{R}^{m \times n} \), with \( m \geq n \). we are interested in finding another matrix \( B \) having specified lower rank \( k \), such that \( B \) is “closest” to \( A \), i.e.,

\[
\begin{align*}
\min_{B} & \| A - B \|_{\text{some norm}} \\
\text{subject to} & \quad \text{rank}(B) \leq k
\end{align*}
\]

This problem goes to the heart of how we use the SVD for dimensionality reduction and to look at data. If we view a data matrix as a collection of columns where each of the columns is a different data point, then a rank-\( k \) approximation to that matrix is a collection of columns all of which represent points that are all on a \( k \)-dimensional subspace. This discovery of hidden subspace structure is what finding low-rank approximations is about.

To understand this problem, we have to first think about what it might mean to approximate a matrix and we use the natural matrix norm to do this.

(a) A simple way to define matrix norm is to treat a matrix \( A \) as a “long vector” composed by concatenating its columns. i.e., we can think of \( A \) as

\[
A = [\vec{a}_0, \vec{a}_1, \cdots, \vec{a}_{n-1}] \Rightarrow \begin{bmatrix} \vec{a}_0 \\ \vec{a}_1 \\ \vdots \\ \vec{a}_{n-1} \end{bmatrix}
\]
With this intuition, the so called Frobenius norm for a matrix is defined as.

\[ \|A\|_F = \sqrt{\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} |a_{ij}|^2} \]  

(11)

Show that the same formula for Frobenius norm can be obtained if we view matrix \( A \) as a long vector composed by row concatenation.

**Solution:** Let \( A = \begin{bmatrix} \vec{a}_0^T \\ \vdots \\ \vec{a}_{n-1}^T \end{bmatrix} \), where \( \vec{a}_i^T \) is the \( i \)-th row for \( i = 0, \ldots, m - 1 \). The Frobenius norm is then given by 

\[ \|A\|_F = \sqrt{\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} |a_{ij}|^2} = \sqrt{\sum_{i=0}^{n-1} \|\vec{a}_i\|^2}. \]

(b) This norm is related to a natural inner-product for matrices, where we treat the matrix as one big long vector. So \( \langle C, D \rangle = \sum_{i=0}^{m-1} (c_i^T \vec{a}_i) = \sum_{i=0}^{m-1} c_i d_i = \sum_{i=0}^{m-1} c_i^T \vec{d}_i \).

Define the trace of a square matrix as the sum of its diagonal terms, i.e., trace(S) = \( \sum_{i=0}^{m-1} s_{ii} \). Show that \( \langle C, D \rangle = \text{trace}(C^T D) = \text{trace}(CD^T) \).

This in turn establishes that 

\[ \|A\|_F = \sqrt{\text{trace}(A^T A)} = \sqrt{\text{trace}(AA^T)}. \]

**Solution:** The \( k^{th} \) diagonal element of \( C^T D \) is \( c_k^T \vec{d}_k = \sum_{i=0}^{m-1} c_{ki} d_{kj} \). Summing up all diagonal terms we get trace\((C^T D) = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} c_{ki} d_{kj} = \langle C, D \rangle \). Similarly considering matrix \( C \) and \( D \) as a collection of row vectors \( \vec{c}_j \) and \( \vec{d}_j \), we have trace\((CD^T) = \sum_{j=0}^{m-1} c_j^T \vec{d}_j \). Summing up diagonal terms we get trace\((CD^T) = \langle C, D \rangle \).

(c) Using either the “long vector” intuition or the “trace trick”, show that Frobenius norm is invariant under orthonormal transformation from either side, i.e., \( \|UA\|_F = \|A\|_F \) and \( \|AV\|_F = \|A\|_F \) for any \( UU^T = I \) and \( VV^T = I \).

**Solution:** Because \( A = [\vec{a}_0, \vec{a}_1, \ldots, \vec{a}_{n-1}] \), and \( UA = [U\vec{a}_0, U\vec{a}_1, \ldots, U\vec{a}_{n-1}] \). We see that each column vector is undergoing an orthonormal transformation. Since \( \|UA\|^2 = \vec{a}^T UU^T \vec{a} = \|\vec{a}\|^2 \), we have that treating \( A \) as a long vector implies that Frobenius norm is invariant under orthonormal transformation. Another way to see this is to use the relation that you just showed between Frobenius norm and trace: \( \|A\|_F^2 = \text{trace}(A^T A) \). Then with orthonormal transformation we have

\[ \|UA\|_F^2 = \text{trace}((UA)^T (UA)) = \text{trace}(A^T U U^T UA) = \text{trace}(A^T A) = \|A\|_F^2 \]

Similarly for \( V \), we see:

\[ \|AV\|_F^2 = \text{trace}((AV)(AV)^T) = \text{trace}(AVV^T A^T) = \text{trace}(AA^T) = \|A\|_F^2 \]

(d) Use SVD and the above invariance to show that \( \|A\|_F = \sqrt{\sum_{i=0}^{\min(m-1,n-1)} \sigma_i^2} \) (Recall that a \( m \times n \) matrix can have at most \( \min(m, n) \) nonzero singular values), where \( \sigma_i \)'s are the singular values of \( A \).

**Solution:** We can write \( A = U \Sigma V^T \) and use the invariance property we just proved. We get

\[ \|A\|_F = \|U \Sigma V^T\|_F = \|\Sigma V^T\|_F = \|\Sigma\|_F = \sqrt{\sum_{i=0}^{\min(m-1,n-1)} \sigma_i^2} \]
(e) A useful lemma for you to show is: Given rank one matrices $D_1, D_2, \ldots, D_k$, their sum $\sum_{i=1}^k D_k$ has rank at most $k$. (Hint: recall that the rank of a matrix is the dimension of the span of its columns)

**Solution:** Take a non-zero column vector $\vec{d}_1$ from $D_1$ and $\vec{d}_2$ from $D_2$, then because the matrices have rank one, column vectors of $D_1$ are either a zero vector, or a multiple of $\vec{d}_1$. Note that a zero vector can be considered as $0\vec{d}_1$. Similarly for $D_2$ and $\vec{d}_2$.

Thus the all column vectors of $D_1 + D_2$ can be expressed as linear combinations of $\vec{d}_1$ and $\vec{d}_2$. Thus we must have rank($D_1 + D_2$) $\leq 2$.

For the general case with the sum of $k$ rank one matrices, we can take a non-zero column vector $\vec{d}_i$ from $D_i$, for $i = 1, \ldots, k$, then columns of $D_i$ are either a zero vector, or a multiple of $\vec{d}_i$. Therefore, all column vectors of $D_1 + D_2 + \cdots + D_k$ can be expressed as linear combinations of $\vec{d}_1, \ldots, \vec{d}_k$. Thus we must have rank($D_1 + D_2 + \cdots + D_k$) $\leq k$.

(f) Consider the special case of square $A$ matrices of size $n \times n$ that are already diagonal. Further suppose that the diagonal of $A$ is non-negative and sorted so that it has $a_{0,0} \geq a_{1,1} \geq \cdots \geq a_{n-1,n-1} \geq 0$ down the diagonal.

You can assume that the best rank-at-most-$k$ approximation to a diagonal $A$ is itself going to be a diagonal matrix $B$ with at most $k$ nonzero terms in it.

Under that assumption, show that $B$ has $a_{0,0}, a_{1,1}, \ldots, a_{k-1,k-1}, 0, \ldots, 0$ down the diagonal.

**Solution:** Under those assumptions, we have

$$\|A - B\|_F^2 = \sum_{i=0}^{n-1} |a_{i,i} - b_{i,i}|^2$$

Realize that the diagonal matrix $B$ can only have rank at most $k$, meaning that at most $k$ of its diagonal terms can be non-zero. This means that at most $k$ of the terms in the above summation can be zero. Since the $a_{i,i}$ are in descending order, we can see that the smallest residual we can get is $\sum_{i=k}^{n-1} |a_{i,i} - b_{i,i}|^2$, obtained by choosing $b_{i,i} = a_{i,i}$ for $i = 0, \cdots, k-1$, and 0 elsewhere. This minimizes the $\sum_{i=1}^{n-1} |a_{i,i} - b_{i,i}|^2$ by cancelling out $k$ largest diagonal terms of $A$.

If you want to be more formal, we can proceed by contradiction. Suppose that all the best solutions, instead of completely canceling one of the terms in the first $k$ entries in the diagonal, left at least of of them alone and instead acted on some other term. Choose the minimal such optimal choice — the one with the fewest elements of the first $k$ entries of the diagonal uncancelled.

The best we can do on the other term is to cancel it. However, because of the descending nature of the diagonal terms, we know that this will cost us $a_{i,i}^2$ for some $i < k$ and gain us only $a_{j,j}^2$ for some $j \geq k$.

So this is a either a strictly worse choice and hence not optimal, or it is as good in which case it is not minimal in terms of having the fewest number of entries in the first $k$ that are not cancelled. Either way, we have a contradiction.

Consequently, we have shown the desired format of $B$ as an optimal choice.

(g) (optional) Show that for the previous problem, that indeed the best rank-at-most-$k$ approximation to a diagonal matrix must itself be diagonal. This fact is intuitively clear, but requires proof. (Hint: A rank-$k$ matrix is a matrix whose column space is spanned by $k$ vectors. This means that a rank-at-most-$k$ matrix that approximates $A$ can be viewed as the product of an $n$-rows by $k$-column matrix $B_0$ premultiplying a $k$-rows by $n$-column matrix $B_\ell$. The best approximation is one that minimizes the sum of the squared differences between the columns of $A$ and the corresponding column of $B_\ell B_\ell^T$. This means that it satisfies a least-squares-type problem. See if you can show by induction that for every $k$, the best choice of basis vectors for $B_\ell$ are the first $k$ standard basis vectors. Start with the base case $k = 1$. Then assume it is true for $k \leq \ell$ and consider $\ell + 1$. )
(h) Now please solve (c) with Frobenius norm, you may want to use the results you developed so far in earlier parts of this problem.

**Solution:** Consider the SVD for $A$, i.e., $A = U\Sigma V^T$. For any $B$, we can write $B = UDV^T$ for $D = U^T BV$. Now we have

$$\|A - B\|_F^2 = \|U\Sigma V^T - UDV^T\|_F^2 = \|\Sigma - D\|_F^2$$

by the invariance of the norm to orthogonal transformation.

Note that $\Sigma$ is diagonal, and from a previous part we showed that the optimal solution $D$ is just a diagonal matrix with first $k$ singular values. Hence we have the optimal choice of $B$ is

$$B^* = U\hat{\Sigma} V^T = \sum_{i=0}^{k-1} \sigma_i \hat{u}_i \hat{v}_i^T$$

where $\hat{\Sigma}$ contains the largest $k$ singular value of $A$, and all the other entries are zeros. Moreover, with the little lemma we just proved, the rank of $B^*$ is no greater than $k$. This $B^*$ solves the low-rank minimization problem and this justifies using the SVD to do PCA.

(i) To specify an arbitrary $m \times n$ matrix, we have to choose $m \times n$ independent elements (entries). In other words, an arbitrary $m \times n$ matrix has $m \times n$ degree of freedom. Consider the SVD again, how much information (independent elements/degree of freedom) do we have to know to specify a rank $r$ matrix? How is low-rank approximation a kind of lossy compression for a matrix?

**Solution:** With SVD of any rank $r$ matrix $X_{m \times n} = U\Sigma V^T$, we know that we need $r$ number of $\hat{u}_i$s, $r$ number of $\hat{v}_i$s, and $r$ non-zero entries as singular values in $\Sigma$. For $\hat{u}_0$, since it is normalized, we need $m - 1$ independent elements (the last is forced by normalization). For the next $\hat{u}_1$, it needs to be normal and orthogonal to $\hat{u}_0$, so there are $m - 2$ independent elements. Similarly, for $\hat{u}_k$, for $k < r$, it needs $m - 1 - k$ number of independent elements. Therefore, in the $U$ matrix, we need $(m - 1) + (m - 2) + \cdots + (m - r) = mr - r(r + 1)/2$ number of independent elements. Similarly for $V$, we need $(n - 1) + (n - 2) + \cdots + (n - r) = nr - r(r + 1)/2$ independent elements. Hence, together with the $r$ singular values in $\Sigma$, we get $(m+n)r - r^2$ elements to specify the low-rank approximation. Hence the rank constraint could be viewed as an information constraint, and low rank approximation is to find the best matrix that approximate the original one with limited degree of freedom.

6. **k-means**

In this problem we will guide you, step-by-step through the implementation of the $k$-means algorithm. You will then run your algorithm on the hand drawn digits dataset and evaluate its performance.

Open the iPython file for the clustering with $k$-means problem.

(a) Implement the `label_dataset` function.

(b) Implement the `update_cluster_centers` function.

(c) Implement the `compute_distortion` function.

(d) Implement the `k_means` function.

Run your algorithm on the hand drawn digits dataset.

(e) What is the error rate if we start with the first 10 points as the initial cluster centers?

**Solution:** The error rate as reported\(^1\) is $\approx 0.2087$, which is the error of misclassification of digits using $k$-means under this initialization scheme.

---

\(^1\)Note that there was an early version of the iPython notebook that had a bug computing the error rate. In this case, the error rate was $\approx 0.31$. 

EECS 16B, Spring 2016, Homework 4
(f) What is the error rate if we pick the best run out of 50?

**Solution:** It would depend on the random choice of initialization points.

In most cases, the error rate improves, but it can slightly deteriorate. Values between \( \approx 0.2059 \) to \( \approx 0.21 \) are common.

The reason for the small improvement, is that the first 10 data points in are not random. These are actually the digits 0...9. It is very hard to randomly start from some place better than such a great starting point.

However, even when having to compete with a great starting point, repeated random initialization of the starting point still manages to improve the error rate in most cases.

7. Your Own Problem

Write your own problem related to this week’s material and solve it. You may still work in groups to brainstorm problems, but each student must submit a unique problem. What is the problem? How to formulate it? How to solve it? What is the solution?

**Contributors:**

- Siddharth Iyer.
- Stephen Bailey.
- Yuxun Zhou.
- Baruch Sterin.
- Shane Barratt.