Robust Principal Component Analysis and Its Application to Digital Imagery

Jonathan D. Bruner

University of Central Oklahoma

Abstract

I have chosen to do my final project on robust principal component analysis (RPCA). The goal of this project is to explain RPCA and how to use it. First, I will explain the difference between PCA and RPCA, and explain the use of the SVD and other techniques and algorithms used in this method. As for examples, I thought to bring back the “Buddy” image we used in the homework for using SVD on images, as well as using it with video footage.

Robust Principal Component Analysis and Its Application to Digital Imagery

Suppose we have some matrix that can be represented as the sum of a low-rank approximation and a sparse matrix. Is it possible to solve each of these components individually? With Robust Principal Component Analysis (RPCA) we have found that, indeed, we can solve such problems. The foundation of RPCA is the Singular Value Decomposition (SVD) method we have discussed in class. We can build on the SVD to get Principal Component Analysis (PCA) then build on that to get our RPCA method. What makes RPCA "Robust"? The difference between PCA and RPCA is that with RPCA we can consider outliers and major corruption in our given matrix. Not only can we consider these anomalies we can use these to capture important information. Applications for RPCA are quite numerous but include Video Surveillance, Facial Recognition, Latent Semantic Indexing, Ranking and Collaborative Filtering, etc. (Candès et al.). It this project I will be focusing on the application to recovering corrupted data from images, and motion capture on video.

Relationship Between PCA and SVD

Singular Value Decomposition (SVD) lies at the heart of both Principal Component Analysis (PCA) and Robust PCA (RPCA). To further understand RPCA we will first look at the relationship between SVD and PCA using the framework provided by Jaadi. Suppose we have some standardized[1] design matrix[2] $\boldsymbol{X} \in \mathbb{R}^{m \times n}$, we can get the SVD of $\boldsymbol{X}$ by $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T$ where $\boldsymbol{U} \in \mathbb{R}^{m \times m}$, $\boldsymbol{S} \in \mathbb{R}^{m \times n}$ (diagonal), and $\boldsymbol{V} \in \mathbb{R}^{n \times n}$ with both $\boldsymbol{U}$ and $\boldsymbol{V}$ being unitary. Now for the PCA let's take the matrix $\boldsymbol{X}^T\boldsymbol{X}$ (I will explain why this matrix later) using the SVD from above we get the following equation:

$$\boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{V}\boldsymbol{S}^T\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T = \boldsymbol{V}\boldsymbol{S}^T\boldsymbol{S}\boldsymbol{V}^T$$

To further simplify we can set $\boldsymbol{D} = \boldsymbol{S}^T\boldsymbol{S}$ which is still a diagonal matrix but with the squares of the singular values, and right multiply both sides by $\boldsymbol{V}$. After doing these two steps we are left with the following equation:

$$(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{V} = \boldsymbol{V}\boldsymbol{D} \tag{1}$$

It might not be obvious but (1) summarizes the eigenvalue, eigenvector equation $(\boldsymbol{A}\boldsymbol{x} = \lambda\boldsymbol{x})$. In our case $\boldsymbol{V}$ is a matrix with all our eigenvectors, and $\boldsymbol{D}$ is a diagonal matrix with the eigenvalues on the diagonal. Now why did we choose $\boldsymbol{X}^T\boldsymbol{X}$? As it turns out $\boldsymbol{X}^T\boldsymbol{X}$ is our Sample Covariance Matrix, more specifically a scaled multiple of the Sample Covariance Matrix. These eigenpairs of our covariance matrix are known as Principal Components and these are what PCE is built upon. Our eigenvectors represent the direction of our axis that account for the most variance, and

---

[1] Calculated by $\frac{\text{value}-\text{mean}}{\text{st.deviation}}$ for each data point. This is due to PCA being sensitive to variance.
[2] See the article from Taboga, Marco (2021) detailing what a design matrix is.

our eigenvalues represent how much variance. SVD is especially nice in this regard since it automatically orders our $\boldsymbol{S}$ and $\boldsymbol{V}$ matrices in order. So, our first eigenpair is the direction with greatest variability. With these we can form a "feature vector" which is just a scaled down version of $\boldsymbol{V}$ if there are any vectors that have very little variance. This in essence is getting a low rank approximation of our data since discarding vectors in $\boldsymbol{V}$ will cause us to lose data. Now if we multiply our standardized matrix transposed $(\boldsymbol{X}^{\boldsymbol{T}})$ with our feature vector transpose $\left(\widehat{\boldsymbol{V}}^{T}\right)$, we will get our final data set.

$$\boldsymbol{Z} = \widehat{\boldsymbol{V}}^{T} * \boldsymbol{X}^{T}$$

From PCA to RPCA

Now that we understand how PCE is related to the SVD, how can we make it more robust? As previously stated, "The difference between PCA and RPCA is that with RPCA we can consider outliers and major corruption in our given matrix.", we just need to be able to separate out any outliers from our data set. The way we are going to achieve this is through an iterative algorithm called Augmented Lagrange Multiplier (ALM). However, before we can get into the algorithm let us try to understand this robustness better.

Suppose we have a matrix $\boldsymbol{M} \in \mathbb{R}^{m \times n}$, that can be decomposed as follows:

$$\boldsymbol{M} = \boldsymbol{L}_0 + \boldsymbol{S}_0 \tag{2}$$

where $\boldsymbol{L}_0$ is a low rank approximation of our matrix $\boldsymbol{M}$, and $\boldsymbol{S}_0$ is a sparse matrix. This alone is not enough to solve the system, as there are too many unknowns. We must first make some assumptions about our matrices that are mentioned in the work by Candès et al. 2011. First, we must assume that $\boldsymbol{L}_0$ is not sparse. This is referenced as the "Incoherence Condition". We will write our Reduced Singular Value Decomposition of $\boldsymbol{L}_0 \in \mathbb{R}^{m \times n}$ as

$$\boldsymbol{L}_0 = \boldsymbol{U\Sigma V}^* = \textstyle\sum_{i=1}^{r} \sigma_i u_i v_i^* \tag{3}$$

where r is the rank of matrix $\boldsymbol{M}$, $\sigma_1, \dots, \sigma_r$ are the positive singular values, and $\boldsymbol{U} = [\boldsymbol{u}_1, \dots, \boldsymbol{u}_r], \boldsymbol{V} = [\boldsymbol{v}_1, \dots, \boldsymbol{v}_r]$ are the left- and right-singular vectors. We can then define the "Incoherence Conditions" as such:

$$\max_i \|\boldsymbol{U}^* \boldsymbol{e_i}\|_2^2 \leq \frac{\mu r}{m}, \quad \max_i \|\boldsymbol{V}^* \boldsymbol{e_i}\|_2^2 \leq \frac{\mu r}{n}, \quad \text{and} \quad \|\boldsymbol{UV}^*\|_\infty \leq \sqrt{\frac{\mu r}{mn}}. \tag{4}$$

Here $\|\boldsymbol{M}\|_\infty = \max_{i,j} \left|\boldsymbol{M}_{ij}\right|$, μ is our incoherence parameter, and $\boldsymbol{e}_i$ is a standard basis in the direction of i. Take, $\|\boldsymbol{V}^* \boldsymbol{e_i}\|^2$, what does this equation represent? We know that $\boldsymbol{e}_i$ is a standard basis, so only one value is 1 while the rest are zeros. This lets us isolate individual columns of

$\boldsymbol{V}^*$ (or rows of $\boldsymbol{V}$), and take the 2-norm of it; and since we want to maximize this what we are wanting is that we do not want all our data in $\boldsymbol{V}$ located in only a few columns. We want the data to be spread across all columns of $\boldsymbol{V}$. According to Candès et al. this asserts that for small values of μ, the singular vectors are spread out. The second assumption is that our sparse matrix cannot be low rank. To remedy this issue, we will assume the sparsity pattern of the sparse matrix is selected uniformly at random. With these two assumptions we can now create our RPCA. On top of these two assumptions, we must also reformat our original question to make sure it is solvable. Instead of solving $\boldsymbol{M} = \boldsymbol{L}_0 + \boldsymbol{S}_0$ we will instead solve the following problem:

$$\min_{\boldsymbol{L}_0, \boldsymbol{S}_0} (\operatorname{rank}(\boldsymbol{L_0}) + ||\boldsymbol{S_0}||_0) \text{ subject to } \boldsymbol{L}_0 + \boldsymbol{S}_0 = \boldsymbol{M} \tag{5}$$

where $||\boldsymbol{S_0}||_0$ represents the number of nonzero entries in $\boldsymbol{S}_0$. If we think back to (2) the whole point is to decompose $\boldsymbol{M}$ into the sum of a low-rank matrix and a sparse matrix. In (5) you can see we are trying to minimize the rank of $\boldsymbol{L_0}$, and minimize the number of nonzero entries in $\boldsymbol{S_0}$. This now gives us the behavior we want, however there is still no guarantee that this can be solved. Candès et al. addresses this by employing convex relaxation. Next, we will use convex relaxation to get this into a form that can be guaranteed to have a solution. This is because convex problems guarantee a global minimum/maximum and not just local minima/maxima.

$$\min_{\boldsymbol{L}_0, \boldsymbol{S}_0} (||\boldsymbol{L_0}||_* + \lambda_0 ||\boldsymbol{S_0}||_1) \text{ subject to } \boldsymbol{L}_0 + \boldsymbol{S}_0 = \boldsymbol{M} \tag{6}$$

where $||\boldsymbol{L_0}||_*$ is the nuclear norm, or the sum of the singular values, and λ_0 is a scalar parameter. Candès et al. goes into detail about optimizing this λ_0 parameter but it turns out that $\lambda_0 = \frac{1}{\sqrt{n_{(1)}}}$ where $n_{(1)} = \max(m, n)$ is universal and always returns the correct result. Therefore, our final optimization problem that we are solving is:

$$\min_{\boldsymbol{L}_0, \boldsymbol{S}_0} \left(||\boldsymbol{L_0}||_* + \frac{1}{\sqrt{n_{(1)}}} ||\boldsymbol{S_0}||_1 \right) \text{ subject to } \boldsymbol{L}_0 + \boldsymbol{S}_0 = \boldsymbol{M}. \tag{7}$$

The algorithm Candès et al. chose to use and the one I have used for my examples is the Augmented Lagrange Multiplier algorithm (ALM). Using equation (13) from (Lin et al., 2010) we can set up the ALM function as such:

$$\mathcal{L}(\boldsymbol{L}_0, \boldsymbol{S}_0, \boldsymbol{Y}, \mu) = \|\boldsymbol{L}_0\|_* + \frac{1}{\sqrt{n_{(1)}}}\|\boldsymbol{S}_0\|_1 + \langle \boldsymbol{Y}, \boldsymbol{M} - \boldsymbol{L}_0 - \boldsymbol{S}_0 \rangle + \frac{\mu}{2}\|\boldsymbol{M} - \boldsymbol{L}_0 - \boldsymbol{S}_0\|_F^2 \quad (8)$$

where $\boldsymbol{Y}$ is our Lagrange Multiplier Matrix, and μ is a chosen incoherence parameter. Candès et al. and Lin et al. both go into more detail about the formulation of the ALM algorithm, but there are a few key points to bring up. One key to this algorithm is the shrinkage operator ($\mathcal{S}_\tau$), defined as $\mathcal{S}_\tau[x] = \operatorname{sign}(x)\max(|x| - \tau, 0)$. The second is the singular value thresholding operator ($\mathcal{D}_\tau$), defined as $\mathcal{D}_\tau[\boldsymbol{X}] = \boldsymbol{U}\mathcal{S}_\tau[\boldsymbol{\Sigma}]\boldsymbol{V}^*$ where $\boldsymbol{X} = \boldsymbol{U\Sigma V}^*$ is any singular value decomposition. Candès et al. goes on to prove that rather than having to solve a sequence of convex problems, it is more efficient to use the shrinkage operator on a matrix by applying it to each element, as well as applying the singular value thresholding operator to a matrix and minimizing it. Below are the minimizations used in the algorithm,

$$\arg\min_{\boldsymbol{S}} \ell(\boldsymbol{L}, \boldsymbol{S}, \boldsymbol{Y}) = \mathcal{S}_{\lambda/\mu}(\boldsymbol{M} - \boldsymbol{L} + \mu^{-1}\boldsymbol{Y}) \quad (9)$$

$$\arg\min_{\boldsymbol{L}} \ell(\boldsymbol{L}, \boldsymbol{S}, \boldsymbol{Y}) = \mathcal{D}_{1/\mu}(\boldsymbol{M} - \boldsymbol{S} + \mu^{-1}\boldsymbol{Y}) \quad (10)$$

Thus, we can minimize ℓ with respect to $\boldsymbol{L}$, fixing $\boldsymbol{S}$ (10), then minimize ℓ with respect to $\boldsymbol{S}$, fixing $\boldsymbol{L}$ (9), then updating the Lagrange Multiplier ($\boldsymbol{Y}$) based on the residual. Below is Algorithm 1 from Candès et al. that summarizes those steps.

ALGORITHM 1: (Principal Component Pursuitby Alternating Directions [Lin et al. 2009a; Yuan and Yang 2009])

initialize: $S_0 = Y_0 = 0$, $\mu > 0$.
while not converged **do**
    compute $L_{k+1} = \mathcal{D}_{1/\mu}(M - S_k + \mu^{-1}Y_k)$;
    compute $S_{k+1} = \mathcal{S}_{\lambda/\mu}(M - L_{k+1} + \mu^{-1}Y_k)$;
    compute $Y_{k+1} = Y_k + \mu(M - L_{k+1} - S_{k+1})$;
end while
output: L, S.

Application

As discussed, the RPCA makes its distinction with the detection/removal of outliers and corrupted data, but how might these be useful to us. I have chosen two examples that use these in opposite ways. First is an example of a single still image of a person's face. We can decompose this image into a low rank approximation of a human face and a sparse matrix. In this case the sparse matrix could be shadows, scars, glasses, facial hair, etc. Using RPCA we can take out the sparse component and be left with just the image of the person's face. This is powerful when it comes to facial recognition because things like shadows, facial hair, etc. can interfere with a computer being able to recognize a face; however, if we can strip those components off, we are just left with the face of the person. Our second example is a matrix with multiple superimposed images from a security camera. Again, we can decompose this matrix, however this time the low rank approximation turns out to be the background (or stationary objects) and our sparse components can capture movement. Depending on what is wanted we can break a video down and only see the background (low rank approximation) or we can see the foreground that is moving (sparse component). These are two examples, one where we want to throw out the sparse component and one where the sparse component is what we want.

Example 1: Single Image Recovery

Using RPCA and the Augmented Lagrange Multiplier algorithm I want to take an image that has been corrupted and try to recover the original. The way I will go about doing this is starting with the original, uncorrupted, image and creating 5 new images that are corruptions of this image. I will then vectorize these images to combine into an $m \times 5$ matrix where m is the number of pixels in the image. Once I have this matrix of vectorized images, I can then pass this matrix into the ALM algorithm (See Appendix A for algorithm details) that will output the Low Rank Matrix, which will hopefully be the recovered original image, and the Sparse Matrix, which will contain the contaminants of the image.

As for the corruption, all corruption generated will be random to prevent ungeneralizable results. I will also show 2 forms of corruption so that we can see how this algorithm handles different forms of corruption. The first will be "block corruption" and the second will be "noise corruption". Figure 1 below shows the uncorrupted image I am using in these examples.

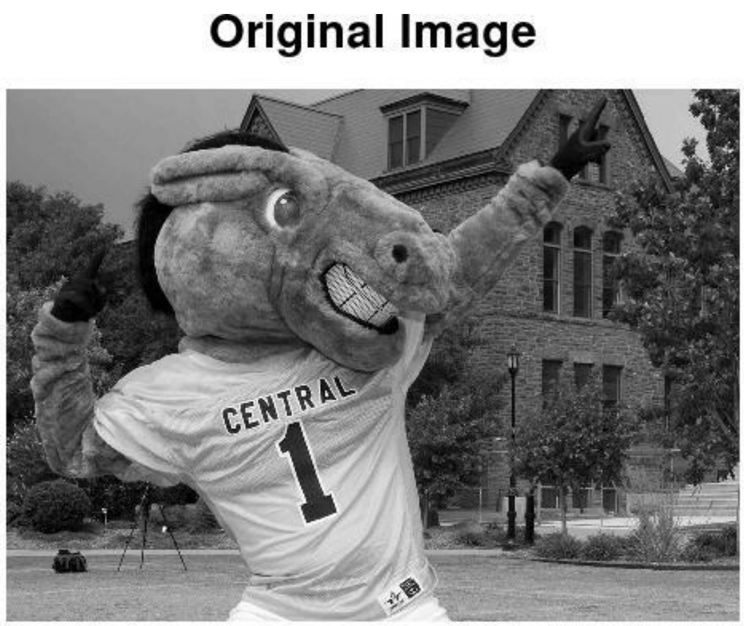

Figure 1

Block Corruption. The first example is a chunk of data being randomly removed from the sample. I am creating a random sized square located in a random spot in the image, and inside of this square I am setting everything to “1” (in gray scale this shows as pure white). Figures 2-5 show the randomized “block corruption” added to the original image. Figures 6-9 show the resulting Low Rank image that was generated by the algorithm, as you can see it was able to successfully identify the corruption and replace it with the true values. Figures 10-12 show the targeted image that needed repair; Figure 10 is the corrupted image, Figure 11 is the resulting Low Rank image, and Figure 12 is the Sparse image that the algorithm identified. An interesting note is that the Sparse image looks to be a color inverted representation of the “true” image in the location of the corruption. When thinking through this algorithm I expected the Sparse image to be the corruption itself, or in other words I expected it to just be a white square.

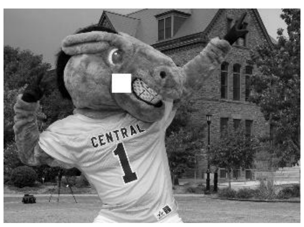

Figure 2: Corrupted Image B

Figure 3: Corrupted Image C

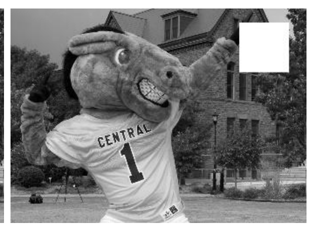

Figure 4: Corrupted Image D

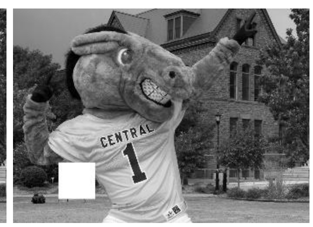

Figure 5: Corrupted Image E

Figure 6: Low Rank Image B

Figure 7: Low Rank Image C

Figure 8: Low Rank Image D

Figure 9: Low Rank Image E

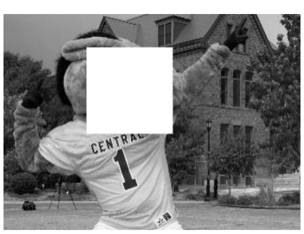

Figure 10: Corrupted Target Image M

Figure 11: Low Rank Target Image M


Figure 12: Sparse Target Image M

Noise Corruption. My next example I wanted to look at corruption that looked more like noise. For this I created a sparse matrix of the size of my images and perturbed the pixels in those locations. In effect, I could leave parts of the image untouched and only perturb a set number of pixels. I was able to accomplish this by adjusting the percentage of pixels touched. The following are the corruption levels for each image: Image B (Figure 13) is 15% corrupted, Image C (Figure 14) is 30% corrupted, Image D (Figure 15) is 45% corrupted, Image E (Figure 16) is 60% corrupted, and the Target Image M (Figure 21) is 75% corrupted. The detail of corruption is hard to see, but if you click into these images, it becomes clear that the low rank images recover a much clearer image. From the Sparse Image we can see that there was a substantial amount of corruption in Image M.


Figure 13: Corrupted Image B

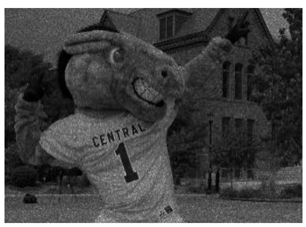
Figure 14: Corrupted Image C

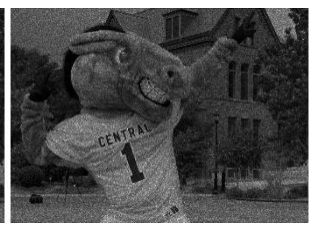
Figure 15: Corrupted Image D

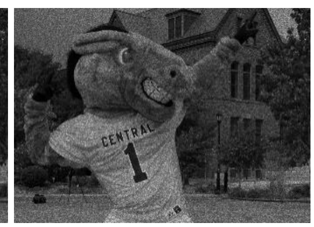
Figure 16: Corrupted Image E


Figure 17: Low Rank Image B


Figure 18: Low Rank Image C


Figure 19: Low Rank Image D


Figure 20: Low Rank Image E


Figure 21: Corrupted Target Image M


Figure 22: Low Rank Target Image M


Figure 23: Sparse Target Image M

As we can see if you have a set of images that are corrupt, we can identify the outliers, or corruption, and remove it to recover a much clearer image. It is important to note that the more corruption you have and the fewer images you have, you might not be able to recover a perfect image. When testing, I tried some extreme corruption levels and even though it was making the image much clearer, there was still substantial corruption. So, this is not an end all be all to get perfectly recovered images every time.

Example 2: Video Surveillance Motion Capture

Those were good examples of being able to identify outliers and remove them, but what if we want to leverage outliers and make use of them. For this example, instead of a still image I want to look at some security footage. If you recall in the previous examples, we vectorized each image and combined them together to create a single matrix of all the images, well in this case it will work the same however instead of different corruptions of the same image, each column will be a frame in the video. The below GIF (Giphycat, 2017) has 40 frames (Figure 24), therefore the matrix I will be using will be an $m \times 40$ matrix where m is the number of pixels in the video. Unlike before where we have the same image repeated just with different corruption, we have different images each time, so what does the Low Rank Image and Sparse Image represent? For this example, the Low Rank Image represents the background of the video; these are things in the video that are stationary. The Sparse Image represents the foreground of the video; these are things that are moving. You can think of outliers in this case as pixels that are changing values throughout the different frames. If you have something in the video that is stationary, the pixels representing that object will stay consistent, not necessarily exact, throughout the video. Below you can see the still background of the Low Rank GIF (Figure 25) and the captured motion of the Sparse GIF (Figure 26).

Figure 24: Original GIF

Figure 25: Low Rank GIF

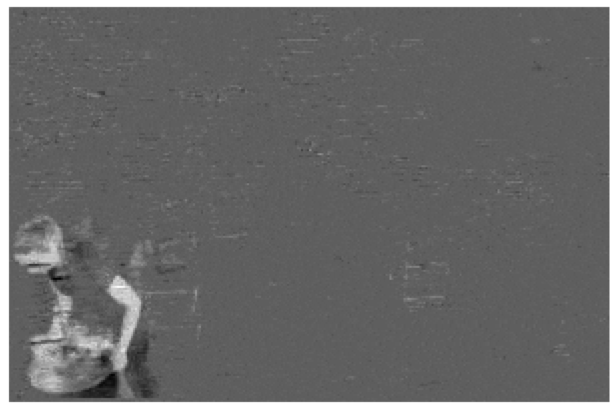

Figure 26: Sparse GIF

Conclusion

We see the results of RPCA from these examples, but how well did it perform these examples. For the first example ("Block Corruption"), it took the algorithm 373 iterations to hit the tolerance provided $\left(1 \times 10^{-7}\right)$ and that took 103.944 seconds (or 1 minute 43 seconds), with an average of 0.279 seconds per iteration. For the second example ("Noise Corruption"), it took the algorithm 1,722 iterations that took 507.732 seconds (or 8 minutes 27 seconds), with an average of 0.295 seconds per iteration. For the third example ("Surveillance Video"), it took the algorithm 8,420 iterations that took 2,329.72 seconds (or 38 minutes 49 seconds), with an average of 0.277 seconds per iteration. As you can see the more complex the problem is, the longer it will take to solve this algorithm, however this algorithm performed quite well since this algorithm averaged 0.284 seconds per iteration.

References

Candès, Emmanuel, et al. “Robust Principal Component Analysis?” Journal of the ACM 58, no. 3, Association for Computing Machinery, May 2011, doi:10.1145/1970392.

Lin, Zhouchen, Minming Chen, and Yi Ma. "The augmented lagrange multiplier method for exact recovery of corrupted low-rank matrices." arXiv preprint arXiv:1009.5055 (2010).

Jaadi, Zakaria. “Principal Component Analysis (PCA) Explained | Built In.” Built In https://builtin.com/data-science/step-step-explanation-principal-componentanalysis.

(Jaadi)

Gfycat. “Security Camera Gifs Ef GIF | Gfycat.” Gfycat, 12 May 2017, https://gfycat.com/calculatingajarelver.

Taboga, Marco (2021). "Design matrix", Lectures on probability theory and mathematical statistics. Kindle Direct Publishing. Online appendix. https://www.statlect.com/glossary/design-matrix.

APPENDIX A

Augmented Lagrange Multiplier Algorithm

ALM.m

```
function [L, S] = ALM(Z)
%{
    This function houses the Augmented Lagrange Multiplier Algorithm
    INPUT:
      Matrix Z
        This matrix is a combination of vectorized matrices of multiple "noisy" imagesc

    OUTPUT:
      Low Rank Matrix (L)
      Sparse Matrix (S)
%}

%{
  Declare variables used in the algorithm
    [m, n]: Size of Original Matrix
    mu: Augmented Lagrangian Parameter (Incoherence Parameter)
    lambda: Regularization Parameter
    delta: Termination Parameter
    L: Low Rank Approximation of Orignial Matrix (Defaulted as empty)
    S: Sparse Matrix containing the outliers of the original Matrix (Defaulted as empty)
    Y: Lagrangian Multiplier (Defaulted as empty)
    Count: Number of iterations the algorithm needs to hit the tolerance threshold
%}
  [m, n] = size(Z);
  %mu = (m * n) / (4 * norm(M,1));
  mu = (m * n) / (4 * sum(abs(Z(:))));
  lambda = 1 / sqrt((max(m,n)));
  %delta = 10^-7;
  delta = 1e-7;
  L = S = Y = zeros(m,n);
  Count = 0;

  % Algorithm 1 from Candes et al.
  while norm(Z - L - S,'fro') > (delta * norm(Z,'fro'))
    L = Thresholding((Z - S + ((mu^-1) * Y)),(1 / mu));
    S = Shrinkage((Z - L + ((mu^-1) * Y)),(lambda / mu));
    Y = Y + (mu * (Z - L - S));
    Count = Count + 1
  endwhile
```

Thresholding.m

```
function D_O = Thresholding(A,t)
%{
    This function handles the Shrinkage Operation from equation 5.3 in Candes et al.

    This function takes in a Matrix A and operator t and outputs a Matrix D_O

    This function also calls another function Shrinkage that outputs a Matrix S_O
    that is the same size as the input matrix
%}

  [U,Sigma,V] = svd(A, 'econ');
  D_O = U * Shrinkage(Sigma,t) * V';
```

Shrinkage.m

```
function S_O = Shrinkage(A,t)
%{
    This function handles the Shrinkage Operation from equation 5.2 in Candes et al.

    This function takes in a Matrix A and an operator t and outputs a Matrix S_O
%}

  S_O = sign(A) .* max(abs(A) - t,0);
```

FinalProject_main.m

```
% load necessary packages
pkg load image
pkg load communications
clear
%{
  Load the Buddy Image and Plot it in a window
  This code was taken from Homework 6
%}
% Load image to approximate
load('Buddy_Images/buddyfig.mat','buddyfig'); % Figure adapted from
https://twitter.com/UCOBronchos/status/1273249045822267401/photo/1
A = buddyfig;

% Plot the original image
figure;
imshow(mat2gray(A));
title('Original Image','fontsize',14,'interpreter','latex')

%{
  Add in some Gaussian White Noise using the built in function awgn(Signal, Signal-Noise ratio)
  https://octave.sourceforge.io/communications/function/awgn.html

  must run "pkg load communications" before this function can work

  I have different M's as I am playing with how to "Mess up" the data
%}

[x, y] = size(A);

%{
  1%, 5%, 10%, 25%, Target of 50%: 674 iterations
  10%, 25%, 40%, 55%, Target of 75%: 1696 iterations

  1722 iterations / 507.732 seconds
%}

B = A .* (sprand(x,y,0.15)+ 1);
C = A .* (sprand(x,y,0.30)+ 1);
D = A .* (sprand(x,y,0.45)+ 1);
E = A .* (sprand(x,y,0.60)+ 1);
M = A .* (sprand(x,y,0.75)+ 1);


%{
  Didn't get a very good Low Rank Approximation but it was a lot better

B = awgn(A, 15, 'measured');
C = awgn(A, 10, 'measured');
D = awgn(A, 5, 'measured');
E = awgn(A, 1, 'measured');
M = awgn(A, 0.5, 'measured');
%}

%{
  373 iterations / 103.944 seconds

B = mat2gray(A);
randBx = randi([1,x]);
randBy = randi([1,y]);
randB2 = randi([50,200]);
for i = randBy:min(randBy+randB2,y)
  for j = randBx:min(randBx+randB2,x)
    B(j,i) = 1;
  endfor
endfor

C = mat2gray(A);
```

```
randCx = randi([1,x]);
randCy = randi([1,y]);
randC2 = randi([50,200]);
for i = randCy:min(randCy+randC2,y)
  for j = randCx:min(randCx+randC2,x)
    C(j,i) = 1;
  endfor
endfor

D = mat2gray(A);
randDx = randi([1,x]);
randDy = randi([1,y]);
randD2 = randi([50,200]);
for i = randDy:min(randDy+randD2,y)
  for j = randDx:min(randDx+randD2,x)
    D(j,i) = 1;
  endfor
endfor

E = mat2gray(A);
randEx = randi([1,x]);
randEy = randi([1,y]);
randE2 = randi([50,200]);
for i = randEy:min(randEy+randE2,y)
  for j = randEx:min(randEx+randE2,x)
    E(j,i) = 1;
  endfor
endfor

M = mat2gray(A);
randMx = randi([80,500]);
randMy = randi([150,550]);
randM2 = randi([100,300]);
for i = randMy:min(randMy+randM2,y)
  for j = randMx:min(randMx+randM2,x)
    M(j,i) = 1;
  endfor
endfor
%}
% Plot the original image with noise
figure;
imshow(mat2gray(B));
title('Corrupted Image B','fontsize',14,'interpreter','latex')
imwrite(mat2gray(B),'Buddy_Images/Noise_Corruption/Corrupted_ImageB.jpg')

figure;
imshow(mat2gray(C));
title('Corrupted Image C','fontsize',14,'interpreter','latex')
imwrite(mat2gray(C),'Buddy_Images/Noise_Corruption/Corrupted_ImageC.jpg')

figure;
imshow(mat2gray(D));
title('Corrupted Image D','fontsize',14,'interpreter','latex')
imwrite(mat2gray(D),'Buddy_Images/Noise_Corruption/Corrupted_ImageD.jpg')

figure;
imshow(mat2gray(E));
title('Corrupted Image E','fontsize',14,'interpreter','latex')
imwrite(mat2gray(E),'Buddy_Images/Noise_Corruption/Corrupted_ImageE.jpg')

figure;
imshow(mat2gray(M));
title('Target Corrupted Image M','fontsize',14,'interpreter','latex')
imwrite(mat2gray(M),'Buddy_Images/Noise_Corruption/Corrupted_ImageM.jpg')


Z = [B(:), C(:), D(:), E(:), M(:)];

tic();
[L, S] = ALM(Z);
toc()
```

```
B = reshape(L(:,1), x, y);
C = reshape(L(:,2), x, y);
D = reshape(L(:,3), x, y);
E = reshape(L(:,4), x, y);
L = reshape(L(:,5), x, y);
S = reshape(S(:,5), x, y);

figure;
imshow(mat2gray(B));
title('Low Rank Image B','fontsize',14,'interpreter','latex')
imwrite(mat2gray(B),'Buddy_Images/Noise_Corruption/Fixed_ImageB.jpg')

figure;
imshow(mat2gray(C));
title('Low Rank Image C','fontsize',14,'interpreter','latex')
imwrite(mat2gray(C),'Buddy_Images/Noise_Corruption/Fixed_ImageC.jpg')

figure;
imshow(mat2gray(D));
title('Low Rank Image D','fontsize',14,'interpreter','latex')
imwrite(mat2gray(D),'Buddy_Images/Noise_Corruption/Fixed_ImageD.jpg')

figure;
imshow(mat2gray(E));
title('Low Rank Image E','fontsize',14,'interpreter','latex')
imwrite(mat2gray(E),'Buddy_Images/Noise_Corruption/Fixed_ImageE.jpg')

figure;
imshow(mat2gray(L));
title('Low Rank Target Image M','fontsize',14,'interpreter','latex')
imwrite(mat2gray(L),'Buddy_Images/Noise_Corruption/LowRank_ImageM.jpg')

figure;
imshow(mat2gray(S));
title('Sparse Target Image M','fontsize',14,'interpreter','latex')
imwrite(mat2gray(S),'Buddy_Images/Noise_Corruption/Sparse_ImageM.jpg')
```

FinalProject_Video.m

```
% load necessary packages
pkg load image
pkg load communications
clear
%{
  Load the images from the video into separate variables
%}

files = glob('SurveillanceGIF/*.jpg');
files = strrep(files, '\', '/');
for i = 1:numel(files)
  [~, name] = fileparts(files{i});
  eval(sprintf('%s = imread("%s");', name, files{i}));
endfor

%{
  turn each frame into a column vector of Matrix Z
%}

[x, y] = size(frame_00);
Z = [];

%{
  For 20 frames: 5274 iterations
  For 40 frames: 8420 iterations / 2329.72 seconds
%}

for i = 0:39
  frame_name = sprintf('frame_%s',substr(sprintf('00%d',i),-2));
  eval(sprintf('Z = [Z, %s(:)];', frame_name));
endfor

%{
  Was getting the following error so added the line below

  "error: xfrobnorm: wrong type argument 'uint8 matrix'"
%}
Z = double(Z);

tic();
[L, S] = ALM(Z);
toc()

for i = 0:39
  frame_name = sprintf('frame_%s',substr(sprintf('00%d',i),-2));

  eval(sprintf('%s = reshape(L(:,i+1),x,y);', frame_name));
  eval(sprintf('imwrite(mat2gray(%s),"LowRank_Frames/%s.jpg");', frame_name, frame_name));

  eval(sprintf('%s = reshape(S(:,i+1),x,y);', frame_name));
  eval(sprintf('imwrite(mat2gray(%s),"Sparse_Frames/%s.jpg");', frame_name, frame_name));
endfor
```