Convex Optimization Project 2

Principal Component Analysis and Surveillance Videos

Felipe González Casabianca

October 2016

Note

This work corresponds to the second project in the class of: Introduction to Convex Optimization, given in the fall of 2016 at Universidad de los Andes.

Chapter 1

Introduction

In this project we wish to analyse how Principal Component analysis can be used for surveillance software. A common task for surveillance video systems is to distinguish moving objects[1] from the background, and alert the security personnel. Although there are specific methods and open source software (in MATLAB and Python) for this challenge [1] [2] [3], we wish to implement a simple principal component analysis scheme and see how it can be used for background and moving object detections. Since the idea is to code from scratch as much as possible, we will compare our results with a classical image editing tool algorithm for background extraction, for its simplicity.

This work is based on the paper: *Robust Principal Component Analysis?*, by Emmanuel J. Candes, Xiaodong Li, Yi Ma and John Wright4.

[1] in reality, since they usually have very low frame per second rates, they need to detect emerging objects and not moving ones

Chapter 2

The Problem

2.1 Background and Moving Object Detection on Surveillance Footage

Given a surveillance video we wish to isolate the moving objects from the background, which enables the system to detect people, animals, cars etc... from the video without manually checking the footage. Recall that surveillance videos have very low frame per second rate, since they produce incredibly large amounts of hours of video. So moving objects will simply be figures that appear for a couple of frames and then disappear, making the definition of background quite immediate. For this project, since getting hold of a security footage is somewhat hard, we use a stationary camera black and white video and then lowered its frame per second rate. Although this video is technically the same as a security camera footage, is has a lot more movement since it corresponds to a crowded street corner of New York.

Specifically, the video can be found in:

`https://www.youtube.com/watch?v=uOFpRIfBaQo`

it was downloaded and split into 204 frames of 90 by 160 pixels using the tool VLC. We will see later that the size of each frame has a great impact on the algorithms performance.

(a) Frame 1

(b) Frame 168

Figure 2.1: Sample frames from the video: *Stationary 2 Minute Camera Shot New York Street Corner v2* by Nicholas Lensander

2.2 Principal Component Analysis

Given a large matrix M, we wish to decompose it in the form:

$$M = L_0 + S_0 \tag{2.1}$$

here, L_0 has low rank and S_0 is sparse. Beside the previous fact, we don't know anything else about any of this matrices. In real world applications in areas like image, video and multimedia processing, the magnitude of the samples are very large, but usually have low intrinsic dimensionality. So it is sane to assume that the a given data in this context can be represented simply a low rank matrix affected by some noise i.e the sum of a low rank matrix and a sparse and small entry perturbation matrix.

This problem is exactly the one that Principal Component Analysis (PCA) tackles. Formally speaking, PCA seeks to solve the problem:

$$\begin{aligned} &\textbf{minimize: } \|M - L\|_2 \\ &\textbf{subject to: } rank(L) \leq k \end{aligned}$$

Now, if noise is sufficiently small and i.i.d Gaussian, singular value decompositions yields an efficient solution for this problem.

But, even though this PCA is widely used, it is incredibly sensitive to large perturbation and data corruptions, two routine facts in today's modern applications. Which leads us to the ideal version of **Robust PCA**, where we allow the perturbation matrix to have randomly large entries with the only condition that they be sparse, returning to our first decomposition of M.

How can this be applied to our static video footage? We use the same technique as the authors of [4]. The idea is to convert each static image of dimension $h \times w$, into a single vector in $\mathbb{R}^{hw}$, and then stacking them as columns of a matrix. Recall that our static video has 204 frames of 90 by 160 pixels, which gives us 204 vectors in $\mathbb{R}^{14400}$ and in return, a matrix M in $\mathbb{R}^{14400\times 204}$. This

technique is easy with black and white images, since the entries of the matrix simply correspond to the gray-scale value of the pixel.

Now, notice that if there where no moving objects in our 204 frames, our matrix M would have all constant rows, since no pixel changes from frame to frame. Therefore, our matrix will be large with rank equal to one. So by lowering the matrix rank, we are extracting the constant elements of the video i.e the background and in turn, the difference of this low rank matrix with the original one, gives us the moving objects in the video.

Robust PCA has more applications [4] to image and video analysis, including light enhancement, face recognition, ranking and collaborative filtering etc.. but this particular application gives very good results.

2.3 Background Extraction with Image Editing Tools

Ever wonder how they take those pictures of the Taj Mahal or Mecca in mid day without a single tourist? Well, they don't. Of course with in this days you can Photoshop your way out of anything, but there is a simple trick (which a lot of basic image editing tools use e.g Pixelmator [5]) to achieve this. You simply take a picture every 5 seconds using a tripod (so your camera doesn't move) and then merge them with a simple image editing tool. What this option does, is simply detects the color that appears the most for each pixel[1] and selects it for the merged photo. So, if you where patient enough and the tourist are moving a lot, it is probable that the most common color for each pixel corresponds to the actual building, instead of the t-shirt of a random person.

We will use this technique with the static camera video, and compare it with the results of PCA.

[1] Can be the mean or the mode

Chapter 3

The Approach

Recall that the problem we wish to address is the decomposition of a given matrix M in the form:

$$M = L + S$$

where L has low rank and S is sparse. At first site this clearly seems challenging, specially when S can have arbitrary large entries.

Nevertheless, the authors of [4], show that in fact it can be solved specifically using *tractable* convex optimization. For notation, let:

$$\|M\|_* = \sum_i \sigma_i(M)$$

be the sum of all the singular values of M (the nuclear norm), and let:

$$\|M\|_1 = \sum_{ij} |M_{ij}|$$

be the $l_1 - norm$.

Now, Candes et.al show that, under a couple of conditions, solving the *Principal Component Pursuit*:

$$\begin{array}{c} \textbf{minimize: } \|L\|_* + \lambda\|S\|_1 \\ \textbf{subject to: } L + S = M \end{array}$$

recovers the low rank and sparse matrix we wanted, at a similar cost as running classic PCA.

In reality, the problem of principal component pursuit for small matrices can be solved easily using commercial software (by interior point) [6], but when matrices exceed the 100×100 dimension, this tools don't appear to be a good

option.

For larger matrices there are several methods including: reduction of singular values (to minimize the nuclear norm), Accelerated Proximal Gradient (APG) and Augmented Lagrangian Multiplier (ALM). The authors use the latter since, in their experience, it is faster and more accurate than the other two.

ALM uses the augmented Lagrangian, that is for this Principal Component Pursuit:

$$l(L, S, Y) = \|L\|_* + \lambda\|S\|_1 + \langle Y, M - L - S\rangle + \frac{\mu}{2}\|M - L - S\|_F^2$$

Where $\|M\|_F$ is the Frobenius norm. This problems can be solved iterating like this:

Algorithm 1 ALM Algorithm

```
1: procedure RunALM()
2:     L_0, S_0, Y_0 ← 0
3:     while not converged do
4:         (L_k, S_k) ← arg min_{L,S} l(L, S, Y_k)
5:         Y_{k+1} ← Y_k + μ(M − L_k − S_k)
       return L_k, S_k
```

Nevertheless, as the authors noticed, it is not necessary to solve a convex problem in each iteration. Let:

$$Shrink_\tau(M)_{ij} = sign(M_{ij}) \max\left(|M_{ij}| - \tau, 0\right)$$

$$SVT_\tau(X) = U\, Shrink_\tau(\Sigma)\, V^*$$

where $X = U\Sigma V^*$ (any singular value decomposition). So, the authors notice that:

$$arg\min_S l(L, S, Y) = Shrink_{\lambda\mu^{-1}}(M - L + \mu^{-1}Y)$$

$$arg\min_L l(L, S, Y) = SVT_{\mu^{-1}}(M - S + \mu^{-1}Y)$$

With this results, the previous algorithm can be simplified as:

Algorithm 2 ALM Algorithm (Optimized)

procedure RunALM()
    $L_0, S_0, Y_0 \leftarrow 0$
    while not converged **do**
        $L_{k+1} \leftarrow SVT_{\mu^{-1}}(M - S_k + \mu^{-1}Y_k)$
        $S_{k+1} \leftarrow Shrink_{\lambda\mu^{-1}}(M - L_{k+1} + \mu^{-1}Y)$
        $Y_{k+1} \leftarrow Y_k + \mu(M - L_k - S_k)$
    return L_k, S_k

This algorithm is the one used for the experiments of this project.

The stopping criteria and choice of λ and μ for the experiments will be the same as the authors:

$$\lambda = \sqrt{numrows(M)} = \sqrt{160 * 90} = \sqrt{14400}$$

$$\mu = \frac{numrows(M) * numcols(M)}{4\|M\|_1} = \frac{160 * 90 * 204}{4}$$

and the algorithm stops when:

$$\frac{\|M - L - S\|_F}{\|M\|_F} \leq 10^{-9}$$

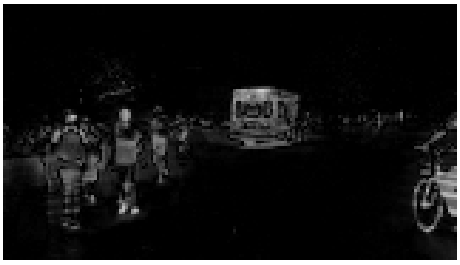
Chapter 4

Results

The following images compare the different methods, when applied to the static camera video, specifically speaking: frames 28 and 195.

(a) Original Frame 28

(b) L

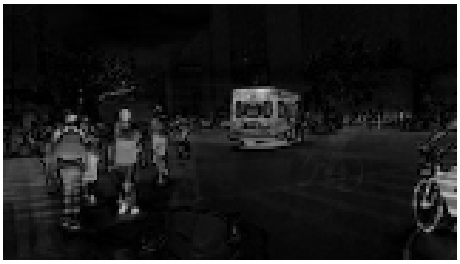


(c) S

Figure 4.1: Frame 28 after **PCA**

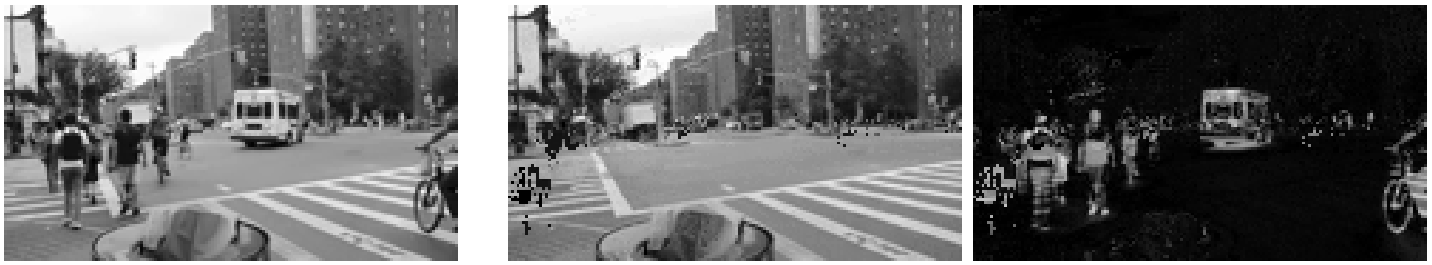
(a) Original Frame 28

(b) L



(c) S

Figure 4.2: Frame 28 after extracting the **Mean** pixel



(a) Original Frame 28 (b) L (c) S

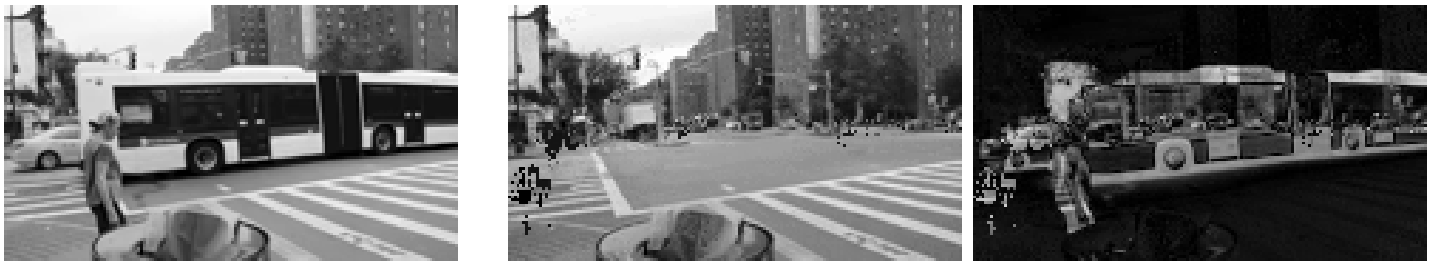
Figure 4.3: Frame 28 after extracting the **Mode** pixel

(a) Original Frame 195 (b) L (c) S

Figure 4.4: Frame 195 after **PCA**

(a) Original Frame 195 (b) L (c) S

Figure 4.5: Frame 195 after extracting the **Mean** pixel



(a) Original Frame 195 (b) L (c) S

Figure 4.6: Frame 195 after extracting the **Mode** pixel

As for frame 28, PCA and the mean pixel both produce good methods, but the superiority of the mean pixel over PCA on the 195 frame is evident. Clearly extracting the mean and mode are simple computational procedure (a few seconds each), unlike the PCA procedure that took over 4000 iterations and 13 hours to compute.

Even with the analytic solution of the authors, the algorithm still needs to write the matrix in its singular value decomposition, which according to the Python's Numpy developer page has a time complexity of:

$$O(nm^2 + n^2m) \tag{4.1}$$

for a matrix of dimensions $n \times m$

Chapter 5

Discussion

Cleary, if we compare the mentioned techniques efficiency wise, the mean (and even the mode) produce acceptable results within a fraction of the time. Nevertheless, the mean is still very sensible to the objects that stay for a long time in the video (like the bicycle will at the lower left corner in the video). Even though the results for PCA aren't very good for certain frames (specially frame 195), they serve as better starting point for the mean pixel algorithm. Since most of the objects have already been removed and those that haven't, appear somewhat as ghost it seams reasonable that if we calculate the mean pixel using the L frames obtained after executing PCA, a better discrimination will be achieved. Indeed, the separation is close to perfect:

(a) Original Frame 28 (b) L (c) S

Figure 5.1: Frame 28 after applying **PCA** and then selecting the **Mean** value for each Pixel

(a) Original Frame 195 (b) L (c) S

Figure 5.2: Frame 195 after applying **PCA** and then selecting the **Mean** value for each Pixel

So even though PCA imposes computational limitations, in combination with this other simple algorithms, gives amazing results, distinguishing completely the background from the moving objects.

The Code

The code for this project was developed in Python 2.7. We include the PCA script for completion, but the entire code (with the resulting images and video) can be found in:

`https://github.com/minigonche/robust_pca`

PCA Source Code

```
#Python scrip for the robust PCA
#For division purpuses
from __future__ import division
#Imports Numpy for matrix manipulation
import numpy as np
#Imports the image handler
import image_interpreter as ii
#Imports OS for iteration inside the folder
import os

#Defines the shinkrage operator for matrixes
def shrinkage(X, tao):
        '''
        Parameters
        ----------
        X : numpy.matrix
                The matrix to be operated on
        tao : float
                A number for the shrinkage value
        ----------
        Return
        shrink_matrix : np.matrix
                A numpy matrix where each coordenate has been shrinked
        '''

        #Defines the function coordinate by coordinate
        def shrink(x, tao):
                return np.sign(x)*max(np.absolute(x) - tao, 0)

        v_shrink = np.vectorize(shrink)
        return v_shrink(X,tao)

#Defines the singular value tresholding
```

```
def singular_value_tresholding(X, tao):
        '''
        Parameters
        ----------
        X : numpy.matrix
                The matrix to be operated on
        tao : float
                A number for the shrinkage value
        ----------
        Return
        shrink_matrix : np.matrix
                A numpy matrix where operated over the singular
                value tresholding
        '''
        U, s, V = np.linalg.svd(X, full_matrices=False)
        s = np.diag(s)
        s = shrinkage(s, tao)

        response = np.dot(U,s)
        response = np.dot(response,V)
        return(response)


def run_PCA(input_folder, output_folder, dimensions, eps, max_ite = 3000):
        '''
        input_folder : String
                The folder name where the images will be imported
        output_folder : String
                The folder where the images will be saved
        dimensions : (int, int)
                A 2 tuple containing the dimensions     of the output images (hight, width)
        eps : float
                epsylon for convergence
        '''

        M = ii.folder_to_matrix(input_folder)

        mu = (M.shape[0]*M.shape[1])/(4*np.linalg.norm(M,1))
        mu_1 = 1/mu

        lamb = 1/np.sqrt(M.shape[0])

        L = np.matrix(np.zeros(M.shape))
        S = np.matrix(np.zeros(M.shape))
        Y = np.matrix(np.zeros(M.shape))

        counter = 0

        partial = 1


        tol = eps +1

        try:
                while( tol > eps):

                        L = singular_value_tresholding( M - S + mu_1*Y, mu_1)
```

```
                S = shrinkage( M - L + mu_1*Y, lamb*mu_1)
                Y = Y + mu*(M - L - S)

                tol = np.linalg.norm(M - (L + S), 'fro')/np.linalg.norm(M,'fro')

                counter = counter + 1

                print(str(S.max()) + ' ' + str(S.min()))
                print(tol)
                print(counter)

                if(counter > max_ite):

                        out_temp = output_folder + '_' + 'partial_' + str(partial)

                        if not os.path.exists(out_temp):
                                os.makedirs(out_temp)

                        m_out = out_temp + '/' + 'M'
                        l_out = out_temp + '/' + 'L'
                        s_out = out_temp + '/' + 'S'
                        #Creates the output subfolderers
                        if not os.path.exists(m_out):
                                os.makedirs(m_out)

                        if not os.path.exists(l_out):
                                os.makedirs(l_out)

                        if not os.path.exists(s_out):
                                os.makedirs(s_out)

                        print('Saving Partial Images...')
                        ii.matrix_image_to_frames(np.absolute(M), dimensions, m_out ,
                        ii.matrix_image_to_frames(np.absolute(L), dimensions, l_out ,
                        ii.matrix_image_to_frames(np.absolute(S), dimensions, s_out ,
                        print('ok')
                        partial = partial + 1
                        counter = 0

finally:


        #Creates the output folder
        if not os.path.exists(output_folder):
                os.makedirs(output_folder)

        m_out = output_folder + '/' + 'M'
        l_out = output_folder + '/' + 'L'
        s_out = output_folder + '/' + 'S'
        #Creates the output subfolderers
        if not os.path.exists(m_out):
                os.makedirs(m_out)

        if not os.path.exists(l_out):
                os.makedirs(l_out)

        if not os.path.exists(s_out):
```

```
                os.makedirs(s_out)

            print('Saving images...')
            ii.matrix_image_to_frames(np.absolute(M), dimensions, m_out , 'matrix_')
            ii.matrix_image_to_frames(np.absolute(L), dimensions, l_out , 'low_rank_')
            ii.matrix_image_to_frames(np.absolute(S), dimensions, s_out , 'sparse_')
            print('ok')

if __name__ == "__main__":

    run_PCA('frames_all', 'frames_out', (90,160), 1/(10**9),1000)
```

Bibliography

[1] Pakorn KaewTraKulPong and Richard Bowden. An improved adaptive background mixture model for real-time tracking with shadow detection. In *Video-based surveillance systems*, pages 135–144. Springer, 2002.

[2] Alexander Mordvintsev and Abid K. Opencv. https://opencv-python-tutroals.readthedocs.io/en/latest/, 2016.

[3] The Mathworks Inc. Motion-based multiple object tracking. https://www.mathworks.com/help/vision/examples/motion-based-multiple-object-tracking.html, 2016.

[4] Emmanuel J Candès, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *Journal of the ACM (JACM)*, 58(3):11, 2011.

[5] Pixelmator Team. Pixelmator 3.6 cordillera. http://www.pixelmator.com/mac/, 2016.

[6] Michael Grant, Stephen Boyd, and Yinyu Ye. Cvx: Matlab software for disciplined convex programming, 2008.