CS168, Spring 2017

Mini-Project #9

Due by 11:59 PM on Wednesday, June 7th.

Instructions

- You can work individually or with one partner. If you work in a pair, both partners will receive the same grade.
- If you've written code to solve a certain part of a problem, or if the part explicitly asks you to implement an algorithm, you must also include the code in your pdf submission. See the problem parts below for instructions on where in your writeup to put the code.
- Make sure plots you submit are easy to read at a normal zoom level.
- Detailed submission instruction can be found on the course website (`http://cs168.stanford.edu`) under the "Coursework - Assignment" section. If you work in pairs, only one member should submit all of the relevant files.
- a) Use 12pt or higher font for your writeup. b) Code marked as "Deliverable" gets pasted into the relevant section, rather than into the appendix (though feel free to put it in both). Keep variable names consistent with those used in the problem statement, and with general conventions. No need to include import statements and other scaffolding, if it is clear from context. Also, please use the `verbatim` environment to paste code in LaTeX from now on, rather than the listings package:

```
def example():
    print "Your code should be formatted like this."
```

- **Reminder:** No late assignments will be accepted, but we will drop your lowest mini-project grade when calculating your final grade.

Part 0: Prelude

This assignment will be centered around linear and convex optimization. We will be using the python `cvxpy` package for convex programming, written by students at Stanford. Installation instructions are at `http://www.cvxpy.org/en/latest/install/index.html`. If you have a partner using a unix-based operating system (GNU/Linux, OS X), we highly recommend using their machine, as installation will be considerably simpler. You can also install `cvxpy` into your corn/farmshare account with `pip install --user cvxpy`.

You are still welcome to use whatever programs you are comfortable with, though our instructions will only be in python. If needed, everything can be done in matlab using the `cvx` package.

Part 1: Compressive sensing

Description

Download and unpack `http://web.stanford.edu/class/cs168/p9_images.tar`. We will be using `wonderland-tree.png`, a 40×30 pixel section of a tree from the queen's garden in Alice and Wonderland. For convenience, it is also presented as a 2D binary array in `wonderland-tree.txt`, with 0 corresponding to black and 1 corresponding to white.

The goal of this section is to see compressive sensing in action, and also to get a bit of practice using a linear programming solver.

1. **Exercises**

   (a) (2 points) Let n be the total number of pixels, and let k be the total number of 1s. What is k/n? Recall that compressive sensing only works when the image is sparse, so we're hoping k/n is much less than 1.

   (b) (6 points) Fix a 1200×1200 matrix A of independently chosen $\mathcal{N}(0, 1)$ Gaussians. Let A_r denote the first r rows of A. Let x be the image `wonderland-tree` as a vector of 1200 pixels. Our compressed image is going to be $b_r = A_r x$.

   Based on Lecture #18, write a linear program that recovers an approximation x_r to x from b_r, and verify that $x_{600} = x$ up to numerical precision (i.e., the recovery is exact).

   [Hint: You'll get better results if you add constraints like $x \geq 0$ into your linear program. The staff implementation takes 5-20 seconds on a laptop, depending on the implementation, so if your program is taking ten minutes you likely have an error.]

   The example code in the *Vectors and Matrices* section at `http://www.cvxpy.org/en/latest/tutorial/intro/index.html` has all the `cvxpy` commands you should need.

   (c) (4 points) Let r^* be the smallest r such that $\|x - x_r\|_1 < .001$ (i.e., $x = x_r$, up to numerical precision errors). Find r^*.

   [Hint 1: Do not use `numpy.allclose()` with the default parameters; the numerical errors here are larger. Hint 2: Use binary search.]

   (d) (3 points) Plot $\|x_i - x\|_1$ for $i = [r^* - 10, r^* - 9, r^* - 8, \ldots, r^* - 1, r^*, r^* + 1, r^* + 2]$. You should see a sharp drop-off.

2. **Bonus question**

   (a) (Extra credit) Speculate 2-3 sentences on why the plot in 1(d) drops off so quickly, rather than gradually decline.

- **Check your understanding** (Do not submit.)

   (a) How would we change the recovery algorithm if `wonderland-tree` were the sum of k rows of the Fourier matrix, rather than the sum of k standard basis vectors?

   (b) Why did we use rows of a fixed A, rather than generating a new A each time?

   (c) Find a dense matrix A of rank r^* for which your recovery algorithm from 1(b) would not work.

   (d) Suppose you only needed to compress this one image, as opposed to coming up with a compression matrix that works for all sparse images. How would you do it?

Deliverables: 1 number (k/n) for part (a). Linear program code for part (b). Value of r^* and code for part (c). Plots for part (d). Discussion for bonus (points awarded for correctness).

Part 2: Image reconstruction

Description

Our friend the Stanford Tree needs your help! Caterpillars are eating it away. You can see the damage in `corrupted.png`, and a picture of the healthy tree in `stanford-tree.png`.

The images are 203×143 pixels. We will be following the example at `http://nbviewer.ipython.org/github/cvxgrp/cvxpy/blob/master/examples/notebooks/WWW/tv_inpainting.ipynb`; feel free to use it (and anything else linked from `cvxpy.org`) as a reference.

The goal of this section is to get a bit more comfortable with `cvxpy`, to see an awesome application of convex programming, and to provide some example code you can reference for Part 3.

3. **Exercises**

   (a) *Do not submit* The following code is the mask we will use to separate the good pixels from the corrupted ones.

```
from PIL import Image
from numpy import array
img = array(Image.open("images/corrupted.png"), dtype=int)[:,:,0]
Known = (img > 0).astype(int)
```

   Display `Known`, and make sure it matches what you'd expect. What fraction of pixels are unknown?

   Note: You can use plt.imshow(img) on python to display the image. Display images in subsequent parts using grayscale (plt.gray())

   (b) (5 points) We will first explore a naive solution for image reconstruction. For every pixel in img that is 0 (unknown), replace it with the average of its known neighbor's pixels. If there are no known neighbors, then keep the pixel value of 0. Submit your recovered image and a 2-3 sentence explanation for why the naive solution performs poorly and how you might improve it.

   (c) (5 points) The following code should reconstruct our mascot!

```
from cvxpy import Variable, Minimize, Problem, mul_elemwise, tv
U = Variable(*img.shape)
obj = Minimize(tv(U))
constraints = [mul_elemwise(Known, U) == mul_elemwise(Known, img)]
prob = Problem(obj, constraints)
prob.solve(verbose=True, solver=SCS)
# recovered image is now in U.value
```

   If you haven't installed SCS, you can just use `prob.solve()`. Submit your recovered image and a 1-2 sentence comparison between this new image and the naive solution's image.

   (d) (5 points) Give a 2-3 sentence explanation for why it makes sense, conceptually, to use the ℓ_2 norm (as opposed to the ℓ_1 norm) for recovering a corrupted image?

Deliverables: No deliverables for part (a). Image, description and code for part (b). Image and description for part (c). 2-3 sentence explanation for part (d).

Part 3: Matrix completion, revisited

Description: Recall our matrix completion problem (we discussed an SVD-based solution in Lecture #9). In this exercise you will solve this problem using nuclear norm minimization.

Let R be a 25×5 matrix, where each entry is chosen independently from the Gaussian distribution $\mathcal{N}(0,1)$. Let $M = RR^T$. Check that your M is a 25×25 matrix. Note that M has rank at most 5.

Let $\widehat{M}$ denote M with each entry "missing" with probability 0.6. Our goal is to recover M from $\widehat{M}$.

Let us take on faith (it can be proved, though it's not easy) that, with high probability, M is the only matrix that both matches the non-missing entries of $\widehat{M}$ and has rank at most 5. This is similar in spirit to the compressive sensing situation where there is only a single k-sparse x such that $Ax = b$.

We wish we could solve the following optimization problem to recover M:

$$\text{Minimize rank}(U)\text{, subject to the constraint that } U \text{ matches the known entries of } \widehat{M}.$$

Let $\Lambda(U)$ denote the eigenvalues of a matrix U. Recall that for a symmetric matrix, rank(U) equals the number of non-zero entries in $\Lambda(U)$, or in other words, the ℓ_0 norm of $\Lambda(U)$. So we can restate this as

$$\text{Minimize } \|\Lambda(U)\|_0\text{, subject to the constraint that } U \text{ is symmetric and matches the known entries of } \widehat{M}.$$

As with compressive sensing, minimizing the ℓ_0 norm is NP-hard, but under certain restrictions, minimizing the ℓ_1 norm returns the exact same solution. So, filled with hope and wonder, we relax our optimization problem to the following:

$$\text{Minimize } \|\Lambda(U)\|_1, \text{ subject to the constraints that:}$$
$$U \text{ is symmetric and matches the known entries of } \widehat{M}, \text{ and } \Lambda(U) \text{ is non-negative.}$$

Basic linear algebra implies that $\Lambda(U)$ is indeed non-negative for every matrix of the form RR^T, so this constraint is satisfied by the desired answer M. Basic linear algebra also implies that this problem can be restated as

Minimize trace(U) — i.e., the sum of the diagonal entries — subject to the constraints that: U is symmetric and positive semidefinite (i.e., of the form BB^T for some matrix B) and also matches the known entries of $\widehat{M}$.

This problem is an example of *nuclear norm* or *trace norm minimization*. We now implement this minimization problem in `cvxpy`.

4. **Exercises**

   (a) (4 points) Show that rank is not a convex function.

   (b) (6 points) Recover a matrix M' from $\widehat{M}$ using nuclear norm minimization, and verify that $M' \approx M$. You may need functions from the following two pages:
   `http://www.cvxpy.org/en/latest/tutorial/advanced/index.html#semidefinite-matrices`
   `http://www.cvxpy.org/en/latest/tutorial/functions/index.html`

   (c) (4 points) Do the same exercise as (b), except where each entry is missing with probability 80%. Output the Frobenius norm of $M' - M$. Repeat 10 times with different random R's and missing entries.

   (d) (6 points) The SVD computation from Lecture #9 returns the "best" rank-5 approximation to a matrix, in some sense. Would you expect this technique to lead to the exact recovery of M from $\widehat{M}$ in the present setting (assume 60% missing)? You may want to investigate this question experimentally. Explain in 3-4 sentences why or why not.

Deliverables: Argument/proof for part (a). Code for part (b). 10 values for part (c). 3-4 sentence explanation for part (d).