CS168, Spring 2019

Mini-Project #9

Due by 11:59 PM on Thursday, June 6th.

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. Even though you may feel that you are just following instructions in this assignment at times, one of the messages we want you to take away is that convex programming is a very powerful tool, and there are packages out there that you can simply use to solve your problem.

We will be using the python `cvxpy` package for convex programming, written by students at Stanford. Installation instructions are at `https://www.cvxpy.org/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 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) (4 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) (9 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 using `numpy.allclose()` 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) (7 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) (5 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.

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).

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` and to see an awesome application of convex programming.

2. **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) (8 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 (up to 4) known neighbors' 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) (8 points) The following code should reconstruct our mascot! (You may want to look up what the function `tv` does here: `http://nbviewer.jupyter.org/github/cvxgrp/cvxpy/blob/master/examples/notebooks/WWW/tv_inpainting.ipynb`)

```
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) (9 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).