CS168, Spring 2015

Mini-Project #7

Due by 11:59 PM on Tuesday, May 19th.

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.
- **New instructions:** 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/p7_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) 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. [Deliverable: 1 number.]

   (b) 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 #14, 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 1: 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.

   [Deliverable: The code, pasted into this section, using the `verbatim` environment. See instructions.]

   (c) 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.) [Deliverable: 1 number]

   (d) 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. [Deliverable: plot].

2. **Bonus questions**

   (a) Speculate on why the plot in 1(d) drops off so hard, rather than gradually declining.

- **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?

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) 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("corrupted.png"))[:,:,0]
Known = (img > 0).astype(int)
```

Display `Known`, and make sure it matches what you'd expect. What fraction of pixels are unknown? [No deliverable.]

(b) Write the function `tv(U)`, defined in the first few lines on the website above, using only the standard libraries. Make sure your `tv(img)` matches the output of `cvxpy`'s built-in `tv`. Note that you'll need `tv(img).value` to get the expected result out of `cvxpy`'s `tv`. (Hint: `img` is a matrix of type `uint8`, so you may want to recast it before doing any computations.) [Deliverable: the code.]

(c) The following code should reconstruct our mascot! Use the built in `tv` command rather than the one we wrote in (b), since special syntax is needed to define operations on `Variable`'s.

```
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()`. [Deliverable: recovered image].

(d) Say we had caught the infestation early in the season, only a random 1% of the pixels had been eaten, and there was no additional textual damage. Propose a much simpler algorithm that would have given reasonable results, and justify why you'd expect the algorithm to work well in this regime. [Deliverable: Description of algorithm, and 1-2 sentences explaining why it would work.]

(e) Why does it make sense conceptually to use the ℓ_1 norm in the objective function for compressive sensing, but to use the ℓ_2 norm for recovering a corrupted image? [Deliverable: Several sentences.]

4. **Bonus**

(a) Prove that `tv` is a convex function. There is a three-line proof if you remember enough facts about convex functions. (Hint: If $g(y)$ is a convex function, and A is an affine transformation, then $f(x) = g(A(x))$ is also a convex function. Think of x and y as vectors, and A as a matrix.)

Part 3: Matrix completion, revisited

Description: Recall our matrix completion exercise from Week 5, where we used a low-rank approximation related to the SVD. We will now solve a similar 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 to recover M:

Minimize rank(U), subject to the constraint that U 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

Minimize $\|\Lambda(U)\|_0$, subject to the constraint that U 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:

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

5. **Exercises**

   (a) Show that rank is not a convex function. [Deliverable: Short argument/proof.]

   (b) 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`
   [Deliverable: the code.]

   (c) Do the same exercise as (b), except where each entry is missing with probability 80%. Output the Frobenius norm of $M' - M$. Repeat 5 times with different random R's and missing entries. [Deliverable: 5 numbers].

   (d) The SVD computation from Week 5 returns the "best" rank-5 approximation to a matrix, in some sense. Would you expect this technique to lead to exact recovery of M from $\widehat{M}$ in the present setting? Please explain why or why not, even if you investigate this experimentally. [Deliverable: a couple of sentences.]