CS168 Spring Assignment 7

SUNet ID(s): johngold jsalloum
Name(s): John Gold Justin Salloum
Collaborators: None

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

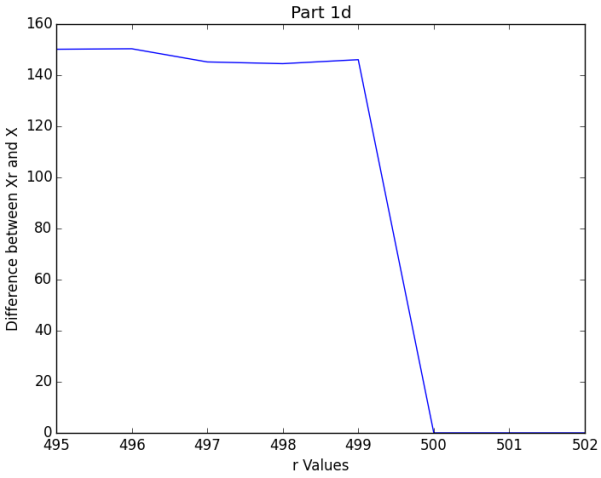
Part 1: Compressive sensing

(a) 0.21

(b)
```
def part1b():
    np.random.seed(1)
    A = np.random.normal(0,1,(1200, 1200))
    x = np.reshape(img, (1200,1))
    br = A.dot(x)
    size = 500
    print "size = ", size
    x_opt = linear_program(A, br, size)
    print np.sum(np.absolute(x_opt - x))

def linear_program(A, br, size):
    br = br[:size]
    Ar = A[:size, :]
    x = Variable(1200)
    constraints = [x >= 0,
                   Ar*x == br]
    # Form objective.
    obj = Minimize(sum_entries(x))
    # Form and solve problem.
    prob = Problem(obj, constraints)
    prob.solve()
    return x.value
```

(c) 500



(d)

Part 1: Bonus

(a) When there is an error $< .001$, the solver basically gets all entries of the original X (image/matrix) correct. However, once we hit the threshold the problem is too underdetermined and many pixels become incorrect.

Part 2: Image reconstruction

(a) 70% of pixels unknown

(b)
```
def my_tv(U):
    total_variations = 0
    for i in range(1, U.shape[0]-1):
        for j in range(1, U.shape[1]-1):
            x = U[i+1, j] - U[i, j]
            y = U[i, j+1] - U[i, j]
            total_variations += np.sqrt(x*x + y*y)
    return total_variations
```

(c) Recovered Image:

(d) Make the values of the corrupted pixels the average of the surrounding 8 pixels. If only 1% of the pixels are corrupted, this should work because we have enough data to just "smooth" over the missing data.

(e) For compressive sensing we want to maximize the sparsity of our result, which would in theory be solved by minimizing the l_0 norm, but since that is NP-Hard we minimize the l_1 norm instead, just the sum of all entries. When recovering a corrupted image we do not want to maximize sparsity, but instead we want to minimize difference between adjacent pixels. This is better solved using the l_2 norm because it penalizes the higher variation pixels (ie, the corrupted ones) vastly more the natural variations in pixel color due to the squared nature of l_2.

Part 2: Bonus

(a) Prove tv is a convex function...

Part 3: Matrix completion, revisited

(a) If we have a graph where rank(A) is on the y-axis and number of rows(A) is on the x-axis, we can show geometrically that rank is not convex. After 1 row is added, the rank is 1, and in this case, after the second row added we also increase the rank. However, imagine a 3rd row that is a combination of rows 1 and 2, now the rank is still 2 but we have 3 rows. Geometrically, the line between (1,1) and (3,2) will go under the graph at (2,2). Formally, the midpoint between x=1 and x=3 exceeds the arithmetic mean between these points ($2 > 1.5$). Therefore rank is not a convex function.

(b)
```
def part3():
    #np.random.seed(1)
    R = np.random.normal(0, 1, (25, 5))
    M = R.dot(R.T)

    known_entries = {}
    for row in range(M.shape[0]):
        for col in range(M.shape[1]):
            if(np.random.random() > .6):
                known_entries[(row, col)] = M[row, col]

    #Start the cvxpy code
    M_prime = Variable(25, 25)
    #Symmetric, positive semi-def, and matches known entries
    constraints = [M_prime == Semidef(25),
                   M_prime == M_prime.T]
    for key, value in known_entries.items():
        constraints.append(M_prime[key[0], key[1]] == value)

    # Form objective.
    obj = Minimize(trace(M_prime))
    # Form and solve problem.
    prob = Problem(obj, constraints)
    prob.solve()

    print np.linalg.norm(M - M_prime.value)
```

(c) 16.67, 20.84, 28.36, 24.45, 21.47

(d) No we do not expect this to work. We are dealing with a lot more missing entries in this scenario than we were in week 5 (60% or 80% compared to 10%). This makes is a lot harder for SVD to recover the entries, because although our true matrix is only of rank 5, it's still too hard to "fill in the blanks" because there are too many possibilities. Also, due to experimental results SVD didn't work well compared to NNM.

```
#p7.py
import scipy
import scipy.misc
import scipy.linalg
import numpy as np
#import matplotlib.pyplot as plt
#import matplotlib.cm as cm
import sys
import os
from cvxpy import *
from PIL import Image
from numpy import array
import copy

def main():
    if(sys.argv[1] == "1"):
        part1()
    if(sys.argv[1] == "2"):
        part2()
    if(sys.argv[1] == "3"):
        part3()

#Matrix completion
def part3():
    #np.random.seed(1) #For sanity/checking
    R = np.random.normal(0, 1, (25, 5))
    M = R.dot(R.T)

    known_entries = {}
    for row in range(M.shape[0]):
        for col in range(M.shape[1]):
            if(np.random.random() > .8):
                known_entries[(row, col)] = M[row, col]

    #Experimentation for part d
    if(sys.argv[2] == "d"):
        M_with_zeros = np.zeros_like(M)
        for key, value in known_entries.items():
            M_with_zeros[key[0], key[1]] = value

        U, s, Vh = scipy.linalg.svd(M_with_zeros)
        k = 10
```

```
        SVD_M = U[:, :k].dot(np.diag(s[:k])).dot(Vh[:k, :])
        print np.linalg.norm(M - SVD_M)


    #Start the cvxpy code
    M_prime = Variable(25, 25)
    #Symmetric, positive semi-def, and matches known entries
    constraints = [M_prime == Semidef(25),
                   M_prime == M_prime.T]
    for key, value in known_entries.items():
        constraints.append(M_prime[key[0], key[1]] == value)
    # Form objective.
    obj = Minimize(trace(M_prime))
    # Form and solve problem.
    prob = Problem(obj, constraints)
    prob.solve()

    print np.linalg.norm(M - M_prime.value)


#Recover the corrupted image!
def part2():
    img = np.array(Image.open("images/corrupted.png"), dtype=int)[:,:,0]
    Known = (img > 0).astype(int)
    print np.sum(Known)/float((Known.shape[0]*Known.shape[1]))

    print my_tv(img)
    print tv(img).value
    raw_input("")
    #From the assignment handout
    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
    recovered_img = np.array(U.value)
    scipy.misc.imsave("recovered_img.png", recovered_img)

#Use info defined by websie on handout
def my_tv(U):
    total_variations = 0
```

```
    for i in range(1, U.shape[0]-1):
        for j in range(1, U.shape[1]-1):
            x = U[i+1, j] - U[i, j]
            y = U[i, j+1] - U[i, j]
            total_variations += np.sqrt(x*x + y*y)
    return total_variations




def part1():
    img = np.genfromtxt("images/wonderland-tree.txt", delimiter=1)
    n = img.shape[0] * img.shape[1]
    k = np.sum(img)
    print k/n
    np.random.seed(1)
    A = np.random.normal(0,1,(1200, 1200))
    x = np.reshape(img, (1200,1))
    #print A
    #print x
    br = A.dot(x)
    size = 600
    print "size = ", size
    x_opt = linear_program(A, br, size)
    print np.sum(np.absolute(x_opt - x))
    print np.sum(np.absolute(x_opt - x)) < .001

    #Find size
    xis = []
    yis = []

    for i in range(-5, 3, 1):
        print i
        r = size + i
        x_opt = linear_program(A, br, r)
        xis.append(r)
        yis.append(np.sum(np.absolute(x_opt - x)))
    print xis
    print yis

    #plt.plot(xis, yis)
    #plt.ylabel('Difference')
```

```
    #plt.xlabel('R')
    #plt.title('Part 1d')
    #plt.show()

#THis is where the CXWY code goes
def linear_program(A, br, size):
    br = br[:size]
    Ar = A[:size, :]
    x = Variable(1200)
    constraints = [x >= 0,
                   Ar*x == br]
    # Form objective.
    obj = Minimize(sum_entries(x))
    # Form and solve problem.
    prob = Problem(obj, constraints)
    prob.solve()
    return x.value


if __name__ == "__main__":
    main()
```