CS168 Spring Assignment 9

SUNet ID(s): anikydv, preksha

Name(s): Anik Yadav, Preksha Koirala

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

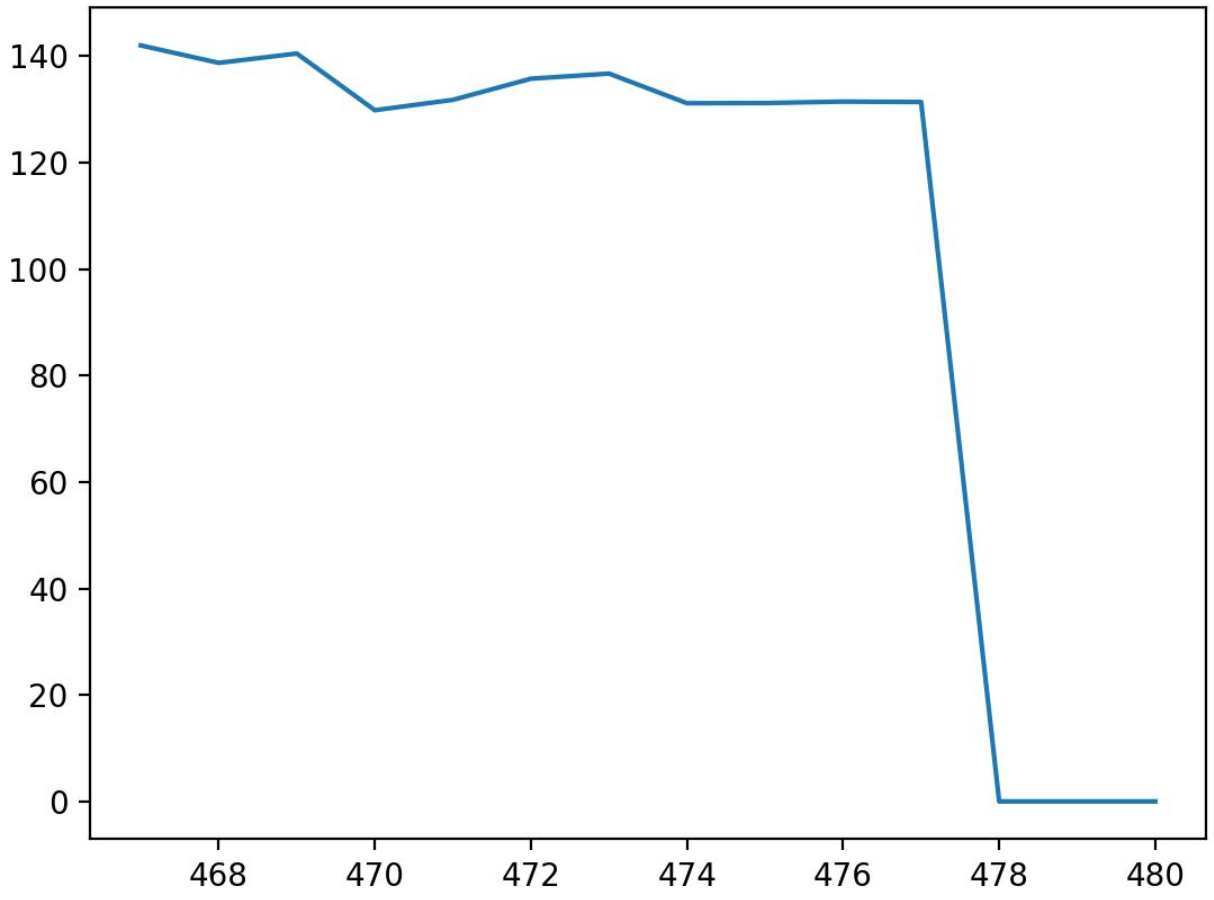
Part 1

a) 0.21

b) r* is 473

c) In appendix and p1.py in corn

d) Plot:



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

Part 2

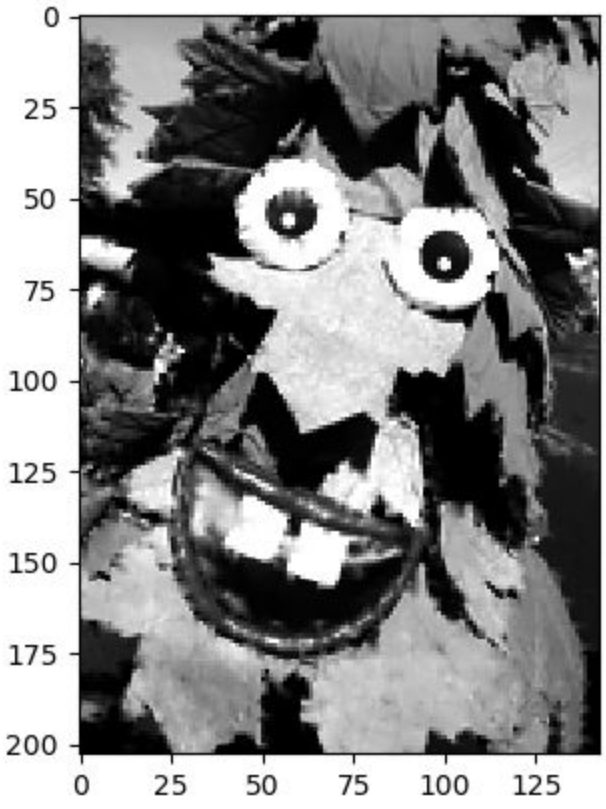
b)

The naive solution takes the average of its known neighbors which means for those pixels that don't have known neighbors, it fails to update it which shows up as white pixels.

In images, averaging works fine for areas with the similar shade of color. But when there is a stark difference between too bright and too dark shade, averaging produces blur area in the middle. (For instance this can be noticed on the front tooth of the Stanford Tree.)

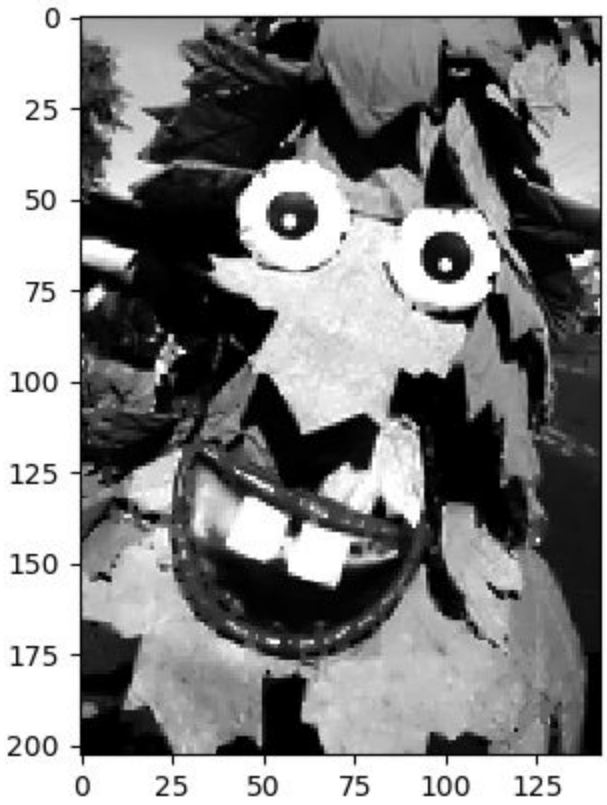
Ways to improve:

For those pixels whose nearest neighbors are all unknown, we could take average of the next nearest neighbors. Also assign weight to the known values for instance, if all its neighbors are known, it should get higher weight in order to impact the average. If less neighbors are known, it should get lower weight.



c)

This new image is less blurred and has better sharpness and clarity. Most of the noticeable differences are in between the light and dark areas. While the naive implementation added a blur at the border of two contrasting areas, this implementation is much sharper and introduces less blur.



d)

L_2 norm is better when we are dealing with non-sparse data. In this case the image was non-sparse since only few pixels were missing. L_2 norm recovers a smooth data because there are many solutions and it chooses the "average" solution among them.
In the case of L_1 norm, there are infinitely many possible solutions and it has to sweep through to find the best solution. The L_1 is not smooth and it is fit for recovering sparse data. In case of our non-sparse image, it will not necessarily produce better results and is also computationally more demanding than the L_2 norm.

Part 3

a)
Let f(X) denote the rank of a matrix X.
Let X=[1 0, 0 0] and Y=[0 0, 0 1] be two arbitrary matrices and λ =[0, 1]
f(X) = 1 and f(Y) = 1

When λ =0.5,
f(λ X + (1- λ)Y) = f([0.5 0, 0 0] + [0 0, 0 0.5]) = f([0.5 0, 0 0.5])
= 2
λ f(X) + (1- λ)f(Y) = 0.5f([0.5 0, 0 0]) + 0.5f([0 0, 0 0.5]
= 0.5*1 + 0.5*1
= 1
For all λ =[0, 1] : f(λ X + (1- λ)Y) > λ f(X) + (1- λ)f(Y)
Therefore, rank is not a convex function.

b)
Code is in appendix.

c)
Code is in appendix and the values are:
33.7276918427
26.0400503743
21.3733786896
28.2366945249
31.0168475982
21.1872428695
23.2430908818
21.1985556263
36.3230078145
34.9965140361

d)No. We cannot exactly recover M from M^ using rank 5 approximation.
SVD is useful when the data is dense (not missing) so it can find the appropriate constraints looking at the pattern of the matrix in order to recreate the missing part. But if we give data that's 60% missing in random order, SVD fails to recognize the pattern and thus cannot recreate the original matrix no matter what best rank approximation we use.

Since we perform SVD on M_hat, we will never be able to recover the full matrix M with rank-5 approximation. In fact, even with full rank-25 approximation, the best that SVD can recover is the sparse matrix M_hat itself.

Some of the values of Frobenius norm we got using SVD are: 48.7227009827, 65.3343388281, 41.3888334043

Appendix

Code for Part 1

```
#-----------------------------------1 B-----------------------------------

import numpy as np
from scipy import ndimage
from cvxpy import *
import math
import matplotlib.pyplot as plt

mean, var = 0, 1
m = 1200
A = np.random.normal(loc =0, scale =1, size=(m, m))

text_file = open("p9_images/wonderland-tree.txt", "r")
# 0 is black, 1 is white
lines = text_file.readlines()
x = []
for line in lines:
   for i in line.strip('\n'):
      x.append(int(i))
r = 600
Ar = A[0:r]
br = np.dot(Ar, x)


xbar = Variable(m)
objective = Minimize(norm(xbar, 1))
constraints = [0 <= xbar, xbar <= 1, Ar*xbar - br == 0]
prob = Problem(objective, constraints)
prob.solve()
print xbar.value

print "norm of xbar is ", np.linalg.norm(xbar.value, ord=1)
print "norm of x is ", np.linalg.norm(x, ord=1)


#-----------------------------------1 C-----------------------------------------



def findOptimal(r):
   Ar = A[0:r]
   br = np.dot(Ar, x)
   xbar = Variable(m)
   objective = Minimize(norm(xbar, 1))
   constraints = [0 <= xbar, xbar <= 1, Ar*xbar - br == 0]
   prob = Problem(objective, constraints)
   prob.solve()

   norm_xbar = np.linalg.norm(xbar.value, ord=1)
   norm_x = np.linalg.norm(x, ord=1)
```

```
    diff = [math.fabs(x[i] - xbar.value[i]) for i in xrange(len(x))]
    norm_diff = np.linalg.norm(diff, ord=1)
    print "norm diff ", norm_diff
    return norm_diff

def binarySearch(rs, start, end, R):
    r = R
    if start > end:
        print "so the least r is ", r
        return r
    mid = (start + end)/2
    candidate = rs[mid]
    print "candidate ", candidate
    if findOptimal(candidate) <= 0.001:
        r = candidate
        return binarySearch(rs, start, mid-1, r)
    elif findOptimal(candidate) > 0.001:
        return binarySearch(rs, mid+1, end, r)
rs = [i for i in xrange(600)]
least_r = binarySearch(rs, 0, len(rs)-1, -1)
print "least r is ", least_r


#-----------------------------------1 D-----------------------------------------------

r_sts = [i for i in xrange(least_r-11, least_r+3)]
norm_diffs = []
for r in r_sts:
    norm_diff = findOptimal(r)
    norm_diffs.append(norm_diff)
plt.plot(r_sts, norm_diffs)
plt.show()
```

Code for Part 2

```
from PIL import Image
from numpy import array
import matplotlib.pyplot as plt

img = array(Image.open("corrupted.png"), dtype=int)[:,:,0]
Known = (img > 0).astype(int)

naive_img = array(img)

for i in xrange(203):
  for j in xrange(143):
    if naive_img[i][j] == 0:
      summ = 0
      num_neigh = 0

      # update spots with avg(known neighbors)
      if (i==0 and j==0):
        summ = (naive_img[i][j+1] + naive_img[i+1][j])
        if naive_img[i][j+1] > 0:
          num_neigh+=1
        if naive_img[i+1][j] > 0:
          num_neigh+=1
      elif (i==0 and j==142):
        summ = (naive_img[i][j-1] + naive_img[i+1][j])
        if naive_img[i][j-1] > 0:
          num_neigh+=1
        if naive_img[i+1][j] > 0:
          num_neigh+=1
      elif (i==202 and j==0):
        summ = (naive_img[i][j+1] + naive_img[i-1][j])
        if naive_img[i][j+1] > 0:
          num_neigh+=1
        if naive_img[i-1][j] > 0:
          num_neigh+=1
      elif (i==202 and j==142):
        summ = (naive_img[i][j-1] + naive_img[i-1][j])
        if naive_img[i][j-1] > 0:
          num_neigh+=1
        if naive_img[i-1][j] > 0:
          num_neigh+=1

      elif (i>0 and i<202 and j>0 and j<142):
        summ = (naive_img[i][j-1] + naive_img[i][j+1] + naive_img[i-1][j] + naive_img[i+1][j])
        if naive_img[i][j-1] > 0:
          num_neigh+=1
        if naive_img[i][j+1] > 0:
          num_neigh+=1
        if naive_img[i-1][j] > 0:
          num_neigh+=1
        if naive_img[i+1][j] > 0:
          num_neigh+=1
```

```
    else:
      if (i==0 and j<142 and j>0):
        summ = (naive_img[i][j-1] + naive_img[i][j+1] + naive_img[i+1][j])
        if naive_img[i][j-1] > 0:
          num_neigh+=1
        if naive_img[i][j+1] > 0:
          num_neigh+=1
        if naive_img[i+1][j] > 0:
          num_neigh+=1

      if (j==0 and i<202 and i>0):
        summ = (naive_img[i][j+1] + naive_img[i-1][j] + naive_img[i+1][j])
        if naive_img[i][j+1] > 0:
          num_neigh+=1
        if naive_img[i-1][j] > 0:
          num_neigh+=1
        if naive_img[i+1][j] > 0:
          num_neigh+=1

      if (i==202 and j<142 and j>0):
        summ = (naive_img[i][j-1] + naive_img[i][j+1] + naive_img[i-1][j])
        if naive_img[i][j-1] > 0:
          num_neigh+=1
        if naive_img[i][j+1] > 0:
          num_neigh+=1
        if naive_img[i-1][j] > 0:
          num_neigh+=1

      if (j==142 and i<202 and i>0):
        summ = (naive_img[i][j-1] + naive_img[i-1][j] + naive_img[i+1][j])
        if naive_img[i][j-1] > 0:
          num_neigh+=1
        if naive_img[i-1][j] > 0:
          num_neigh+=1
        if naive_img[i+1][j] > 0:
          num_neigh+=1

    if num_neigh > 0:
      naive_img[i][j] = ((summ*1.0)/num_neigh)

plt.imshow(naive_img)
plt.gray()
plt.savefig('recovered_stanford_tree_2b.png')
plt.show()

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()
# recovered image is now in U.value
plt.imshow(U.value)
plt.gray()
```

```
plt.savefig('recovered_stanford_tree_2c.png')
plt.show()
```

Code for Part 3:

```
----------------------------------------------------------------3 B----------------------------------------------------------------

import numpy as np
from scipy import ndimage
from cvxpy import *
import math

m = 25
n = 5
R = np.random.normal(loc =0, scale =1, size=(m, n))
M = R.dot(R.T)


P = 0.6
M_hat = np.zeros(shape=(m,m))
masked = np.zeros(shape=(m,m))
for i in xrange(m):
   for j in xrange(m):
      masked[i][j] = np.random.choice([0, 1], p=[P, 1-P])

M_hat = mul_elemwise(masked, M)

U = Semidef(m)
objective = Minimize(trace(U))
constraints = [mul_elemwise(masked, U) == M_hat]
prob = Problem(objective, constraints)
prob.solve()
print "the prob status is ", prob.status

diff = M - U.value
print "diff and then norm is ", np.linalg.norm(diff, 'fro')


----------------------------------------------------------------------------------3 C----------------------------------------------------------------------------------

import numpy as np
from scipy import ndimage
from cvxpy import *
import math

m = 25
n = 5
for ind in xrange(10):
   R = np.random.normal(loc =0, scale =1, size=(m, n))
   M = R.dot(R.T)
   P = 0.8
   M_hat = np.zeros(shape=(m,m))
   masked = np.zeros(shape=(m,m))


   for i in xrange(m):
```

```
    for j in xrange(m):
        masked[i][j] = np.random.choice([0, 1], p=[P, 1-P])
M_hat = mul_elemwise(masked, M)
U = Semidef(m)
objective = Minimize(trace(U))
constraints = [mul_elemwise(masked, U) == M_hat]
prob = Problem(objective, constraints)
prob.solve()
print "the prob status is ", prob.status
#print U.value

diff = M - U.value
print "diff and then norm is ", np.linalg.norm(diff, 'fro')
```