HW10

April 2, 2021

We will examine an application of Compressed Sensing in the form of audio signal recovery. I will note that the provided Python functionality is imported throught the utility package seen below.

[1]:
```
from optimization.methods import proximalNGD
import utility as utils

import cvxpy as cvx
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Problem 1

We begin by solving the following Basis Pursuit problem for some arbitrary matrix $A = \Psi D^T$ and vector b.

$$\begin{aligned} \min_x \quad & ||x||_1 \\ \text{s.t.} \quad & \Psi D^T x = b \end{aligned}$$

We note that this is the original formulation of the Compressed Sensing problem, and we will call the solution x_{BP} as we will use it later.

We take the elements of A and b to be distributed as a standard normal.

[2]:
```
#Set up A and b
rng = np.random.default_rng()

n = 20
m = 10

A = rng.standard_normal((m,n))
b = rng.standard_normal(m)

print(f'A shape: {A.shape}, b shape: {b.shape}')
```

```
A shape: (10, 20), b shape: (10,)
```

Now we solve using CVXPY.

[3]:
```
#Solve with cvxpy
x = cvx.Variable(n)
obj = cvx.Minimize(cvx.norm(x, 1))
con = [A@x==b]
prob = cvx.Problem(obj, con)

prob.solve()

x_BP = x.value

print(f'Minimum: {prob.value}')
```

```
Minimum: 2.441317157527109
```

Not too much to look at yet, and mostly I am looking at the minimum to be sure we got an answer.

Problem 2

Next, we consider a smooth least-squares variant of the basis pursuit problem we solved above.

$$\begin{aligned} \min_x \quad & \frac{1}{2}||\Psi D^T x - b||_2^2 \\ \text{s.t.} \quad & ||x||_1 \leq \tau \end{aligned}$$

We take $\tau = ||x_{BP}||_1$ (the solution from earlier) and apply a proximal Nesterov accelerated gradient descent method. The data $A = \Psi D^T$ and b are the same as in the previous problem.

[4]:
```
tau_BP = prob.value

f = lambda x, A, b: (1/2)*np.sum((A@x-b)**2)
gf = lambda x, A, b: A.T@(A@x-b)

x0 = np.ones(n)
step0 = np.linalg.norm(A, 2)**-2
```

[5]:
```
x_LS = proximalNGD(x0, f, gf, prox=utils.project_l1, step=step0,
                   args=(A,b), prox_args=(tau_BP,), maxItr=3000)
```

[6]:
```
print('Solution Difference: {}'.format(np.linalg.norm(x_BP-x_LS)))
```

```
Solution Difference: 0.002529397137169277
```

The solutions aren't exactly the same, but we are at least getting about two digits of accuracy. Although I will note that depending on the data A and b this accuracy can vary. In all cases however we are getting similar enough answers to move forward, and we can decrease our tolerance when it matters.

Problem 3

Now we assume that we do not know that correct value of τ, but we do know that we want the smallest value such that $\sigma(\tau) = ||Ax_\tau - b||_2 = 0$. Here A is defined as before and x_τ is the solution of the minimization problem above. Thus we can apply an iterative root finding process such as Newton's method as an outer loop over a proximal descent for finding x_τ.

[7]:
```
'''
Modified Newton root finding
'''
def newton(t0, x0, f, df, fargs=(), maxiter=100, atol=1e-5, rtol=1e-8):
    for i in range(maxiter):
        #Compute new t
        f_val, x_t = f(t0, x0, *fargs)
        t1 = t0 - f_val/df(t0, x_t, *fargs)

        #Check convergence
        if np.abs(t1-t0) < atol + rtol*np.abs(t0):
            return t1, x_t

        #Update
        t0 = t1
        x0 = x_t

    print('Maximum iterations exceeded without meeting tolerance!')

    return t1, x_t
```

Below we define $\sigma(\tau)$ and $\sigma'(\tau)$ as per homework 9.

[8]:
```
#Define how to find x given t
def xt(t, x0, A, b):
    return proximalNGD(x0, f, gf, prox=utils.project_l1, step=step0,
                       args=(A,b), prox_args=(t,), maxItr=10000)

#Define sigma and its derivative
def s(t, x0, A, b):
    x_t = xt(t, x0, A, b)

    return np.linalg.norm(A@x_t-b), x_t

def ds(t, x_t, A, b):
    #First compute z_t
    z_t = b - A@x_t

    #First compute nu_t
    if np.all(z_t==0):
        #Self-destruct
        rng = np.random.default_rng()
        y = rng.standard_normal(z_t.shape)
```

```
        nu_t = -y/np.linalg.norm(y,2)

    else:
        nu_t = -z_t/np.linalg.norm(z_t,2)

    #Now compute l_t
    return -np.linalg.norm(A.T@nu_t, np.inf)
```

We start with an initial guess of $\tau = 1$ and run our double loop method.

[9]:
```
#Find t and then get solution
t0 = 1 #Keine anhung
tau_N, x_N = newton(t0, x0, s, ds, fargs=(A, b))
```

[10]:
```
#Compare
print('Tau Difference: {}'.format(np.abs(tau_BP-tau_N)))

#Need to extract solution somehow
print('Solution Difference: {}'.format(np.linalg.norm(x_BP-x_N)))
```

```
Tau Difference: 0.001092308578965806
Solution Difference: 0.00018733792330638739
```

It looks like we are converging to the same solution x_{BP} and the same τ. The method converged to the requested tolerance, so I would assume that we can decrease this tolerance to get more digits of accuracy, although we may need more iterations.

Problem 4

Finally, we will apply the same method as in Problem 3 to the task of audio recovery for the Handel audio signal, which is imported and can be played below. Our overall approach is to let $A = \Psi D^T$ and $b = \Psi(y)$ and solve for x using the Newton/Proximal descent method above. Then we recover the approximate signal via $\hat{y} = D^T(x)$.

[11]:
```
#Load audio signal
import pickle
from IPython.display import Audio

y, Fs = pickle.load(open('handel.pkl', 'rb'))

y = y.ravel()
Fs = float(Fs[0][0])

Audio(y, rate=Fs)
```

[11]:
```
<IPython.lib.display.Audio object>
```

We begin by defining the linear operator Ψ and its transpose. The forward operator's action is to sample from a vector according to a previously determined random process. To do this we sample inter-arrival times according to a geometric distribution with parameter $p = 1/4$. This is a discrete distribution on $\{1, 2, ...\}$ that gives us the number of index steps until we extract a sample. The

transpose operator's action is to crudely up-sample by inserting zeros for any indices that weren't sampled.

[87]:
```
#Functionality for Psi linear operator
class PsiOperator():
    def __init__(self, p, shape):
        rng = np.random.default_rng()

        #Sample n geometric r.v. (worst case is all are taken)
        samples = rng.geometric(p, shape)

        #Now sum over rows to get inter-arrival times
        #Subtract one to get indices instead of steps
        #Only take indices that are valid
        samples = np.cumsum(samples) - 1
        self.samples = samples[samples < shape[0]]
        self.shape = shape

    #Take every fourth sample from x
    def __call__(self, x):
        #Extract these indices from the data
        return x[self.samples]

    #Fill in up to shape of x with zeros
    def T(self, x):
        out = np.zeros(self.shape)
        out[self.samples] = x

        return out

#DCT operators
from utility import forwardShortTimeDCT as D
from utility import adjointShortTimeDCT as D_T

class AOperator():
    def __init__(self, psi):
        self.psi = psi
        self.win = np.sin(np.pi*(np.arange(1,1024+1)+0.5)/1024)

    def __call__(self, x):
        return self.psi(D_T(x, self.win, self.psi.shape[0]))

    def T(self, x):
        return D(self.psi.T(x), self.win)
```

In the code below we take care of re-defining some of our previous functions so that they will work with the new setup. I probably could have set it all up in a smarter way to begin with, but this will do for now.

[95]:
```
#Instantiate our operators
psi = PsiOperator(1/4, y.shape)
A = AOperator(psi)

#Data
b = psi(y)

#Define our objective function and its derivative
f = lambda x, A, b: (1/2)*np.sum((A(x)-b)**2)
gf = lambda x, A, b: A.T(A(x)-b)

#Define how to find x given t
def xt(t, x0, A, b):
    return proximalNGD(x0, f, gf, prox=utils.project_l1, step=.1,
                       args=(A,b), prox_args=(t,), maxItr=100000)

#Define our sigma and its derivative
def s(t, x0, A, b):
    x_t = xt(t, x0, A, b)

    return np.linalg.norm(A(x_t)-b), x_t

def ds(t, x_t, A, b):
    #First compute z_t
    z_t = b - A(x_t)

    #First compute nu_t
    if np.all(z_t==0):
        #Self-destruct
        rng = np.random.default_rng()
        y = rng.standard_normal(z_t.shape)
        nu_t = -y/np.linalg.norm(y,2)

    else:
        nu_t = -z_t/np.linalg.norm(z_t,2)

    #Now compute l_t
    return -np.linalg.norm(A.T(nu_t), np.inf)
```

Now we run our double loop method and then recover the signal.

[96]:
```
#Find t and then get solution
t0 = 1 #Keine anhung
# x0 = np.ones((2*(y.shape[0]),))
x0 = np.ones((147456,))
tau_H, x_H = newton(t0, x0, s, ds, fargs=(A, b))
```

[97]:
```
#Recover x
y_h = D_T(x_H, np.sin(np.pi*(np.arange(1,1024+1)+0.5)/1024))
```

```
Audio(y_h, rate=Fs)
```

[97]: <IPython.lib.display.Audio object>

Wow, I am thoroughly impressed at how well that worked. It is only upon listening to the original signal that I realize the quality degradation, but the reconstruction is quite good. I would say the most noticeable difference is how garbled the background strings become. In the original they are quite distinct, but in the reconstruction it is hard to make them out.