Japan Advanced Institute of Science and Technology

Homework 13

Optimization in Information Theory

- I232 Information Theory 2021 -

By PRASETYO, Yusuf

2110148

June 9, 2021

Contents

1 Number 1

The code will be implemented in python, more specifically Jupyter Notebook as follows:

Arimoto-Blahut Algorithm for Capacity

```python
import numpy as np
import matplotlib.pyplot as plt
import math
```

```python
def p_exp(e,x):
    """
    Practical exponentiation
    returns 1 if given 0 to the power 0
    """
    if e == 0 and x == 0:
        return 1
    else:
        return e**x
```

```python
def entropy_1(px):
    """
    Entropy count of one variable vector
    Input: probability
    Output: entropy in nats
    """
    Hx = []
    for i in range(len(px)):
        if px[i] == 0:
            Hx.append(0)
        else:
            Hx.append(-px[i]*np.log(px[i]))
    return np.sum(Hx)
```

```python
def entropy_2(pxy):
    """
    Entropy count of two variable matrix
    Input: joint probability
    Output: joint entropy in nats
    """
    Hxy = []
    for i in range(len(pxy)):
        for j in range(len(pxy[0])):
            if pxy[i][j] == 0:
                Hxy.append(0)
            else:
                Hxy.append(-pxy[i][j]*np.log(pxy[i][j]))
    return np.sum(Hxy)
```

```python
def capacity(pxy, entropy_2, entropy_1):
    """
    Capacity count of two variables
    Input: joint distribution
    Output: mutual information in bits
    """
    Hxy = entropy_2(pxy)
    Hy = entropy_1(np.sum(pxy, axis=0))
    Hx = entropy_1(np.sum(pxy, axis=1))
    return (Hx+Hy-Hxy)/np.log(2)
```

```python
def gen_pxgy(pygx, px, sx, sy):
    """
    Generation of optimum px-given-y
    Input: channel (pygx) and input distribution (assumed as constant)
    Output: optimum px-given-y
    """
    pxgy = np.zeros((sy,sx))
    for i in range(sy):
        mx = 0
        for j in range(sx):
            mx += px[j]*pygx[j][i]
        for j in range(sx):
            pxgy[i][j] = px[j]*pygx[j][i]/mx
    return pxgy
```

```python
def gen_npx(pygx, pxgy, sx, sy, p_exp):
    """
    Generation of optimum input distribution
    Input: channel (pygx) and px-given-y
    Output: optimum px
    """
    npx = []
    for i in range(sx):
        tx = 1
        for j in range(sy):
            tx = tx * p_exp(pxgy[j][i],pygx[i][j])
        npx.append(tx)
    npx = npx/np.sum(npx)
    return npx
```

```python
def gen_pxy(pygx, px, sx, sy):
    """
    Generation of joint probability
    Input: channel and input distribution
    Output: joint probability pxy
    """
    pxy = np.zeros((sx,sy))
    for i in range(sx):
        for j in range(sy):
            pxy[i][j] = pygx[i][j]*px[i]
    return pxy
```

```python
def AB_Capacity(pygx, capacity, entropy_1, entropy_2, gen_pxgy, gen_npx, gen_pxy, p_exp):
    """
    Arimoto-Blahut algorithm for searching capacity of given Channel Pygx
    Uses uniform distribution as first guess
    Stops when input distribution is converged
    Input: channel (pygx)
    Output: optimum input distribution and capacity of channel
    """
    sx = len(pygx)
    sy = len(pygx[0])
    px = np.zeros(sx)
    npx = []
    for i in range(sx):
        npx.append(1/sx)
    while np.linalg.norm(npx-px) > 10**-6:
        px = npx
        pxgy = gen_pxgy(pygx, px, sx, sy)
        npx = gen_npx(pygx, pxgy, sx, sy, p_exp)
    px = npx
    print("Optimum channel usage is:", px)
    pxy = gen_pxy(pygx, px, sx, sy)
    cap = capacity(pxy, entropy_2, entropy_1)
    print("Maximum capacity is:", cap)
    return cap
```

```python
# pygx = np.array([[0.75, 0.25, 0],
#                 [0, 1, 0],
#                 [0, 0.5, 0.5]])
pygx = np.array([[1, 2, 3, 4, 5, 6, 7, 8, 9],
                [5, 2, 8, 1, 3, 6, 4, 7, 9],
                [5, 8, 2, 9, 7, 4, 6, 3, 1]])/45
Capacity = AB_Capacity(pygx, capacity, entropy_1, entropy_2, gen_pxgy, gen_npx, gen_pxy, p_exp)
```

```
Optimum channel usage is: [0.00194499 0.49857312 0.49948189]
Maximum capacity is: 0.21262939018889393
```

Capacity of the channel is $C \approx 0.2126$.

Input distribution $p^*_{\mathsf{X}} \approx [0.00194499, 0.49857312, 0.49948189]$ (numerical precision matters).

2 Number 2

(a) We will use this code for implementing Arimoto-Blahut algorithm:

Arimoto-Blahut Algorithm for Rate-Distortion

```
import numpy as np
import matplotlib.pyplot as plt
import math
```

```
def p_exp(e,x):
    """
    Practical exponentiation
    returns 1 if given 0 to the power 0
    """
    if e == 0 and x == 0:
        return 1
    else:
        return e**x
```

```
def entropy_1(px):
    """
    Entropy count of one variable vector
    Input: probability
    Output: entropy in nats
    """
    Hx = []
    for i in range(len(px)):
        if px[i] == 0:
            Hx.append(0)
        else:
            Hx.append(-px[i]*np.log(px[i]))
    return np.sum(Hx)
```

```
def entropy_2(pxy):
    """
    Entropy count of two variable matrix
    Input: joint probability
    Output: joint entropy in nats
    """
    Hxy = []
    for i in range(len(pxy)):
        for j in range(len(pxy[0])):
            if pxy[i][j] == 0:
                Hxy.append(0)
            else:
                Hxy.append(-pxy[i][j]*np.log(pxy[i][j]))
    return np.sum(Hxy)
```

```
def rate(pxy, entropy_2, entropy_1):
    """
    Capacity count of two variables
    Input: joint distribution
    Output: mutual information in bits
    """
    Hxy = entropy_2(pxy)
    Hy = entropy_1(np.sum(pxy, axis=0))
    Hx = entropy_1(np.sum(pxy, axis=1))
    return (Hx+Hy-Hxy)/np.log(2)
```

```
def gen_pygx(py, dxxh, lam, sx, sy):
    """
    Generation of optimum channel
    Input: output probability, distortion matrix, and lambda value
    Output: optimum channel
    """
    pygx = np.zeros((sx, sy))
    for i in range(sx):
        my = 0
        for j in range(sy):
            my += py[j]*np.exp(lam*dxxh[i][j])
        for j in range(sy):
            pygx[i][j] = py[j]*np.exp(lam*dxxh[i][j])/my
    return pygx
```

```
def gen_npy(px, pygx, sy, sx):
    """
    Generation of optimum output distribution
    Input: channel (pygx) and px
    Output: optimum py
    """
    npy = []
    for i in range(sy):
        pyc = 0
        for j in range(sx):
            pyc += px[j]*pygx[j][i]
        npy.append(pyc)
    npy = npy/np.sum(npy)
    return npy
```

```python
def gen_pxy(pygx, px, sx, sy):
    """
    Generation of joint probability
    Input: channel and input distribution
    Output: joint probability pxy
    """
    pxy = np.zeros((sx,sy))
    for i in range(sx):
        for j in range(sy):
            pxy[i][j] = pygx[i][j]*px[i]
    return pxy
```

```python
def gen_D(pxy, dxxh):
    """
    Calculation of distortion
    Input: joint probability and distortion matrix
    Output: distortion value"""
    return np.sum(pxy*dxxh)
```

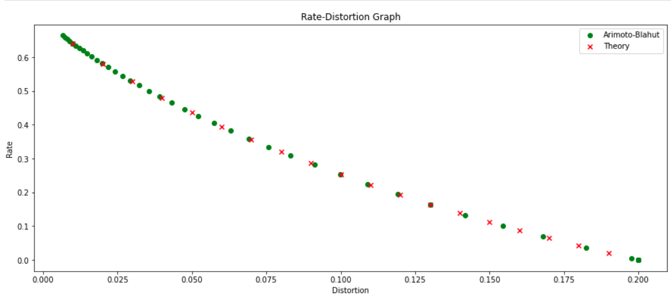
```python
def AB_Rate(px, dxxh, lams, gen_pygx, gen_npy, gen_pxy, rate, gen_D, entropy_2, entropy_1):
    """
    Arimoto-Blahut algorithm for searching rate distortion of given input and distortion matrix
    Uses uniform distribution as first guess
    Stops when output distribution is converged
    Input: input distribution and distortion matrix
    Output: rate-distortion pairs
    """
    sx = len(dxxh)
    sy = len(dxxh[0])
    rates = []
    Ds = []
    npy = []
    for i in range(sy):
        npy.append(1/sy)
    for k in range(len(lams)):
        py = np.zeros(sy)
        while np.linalg.norm(npy-py) > 10**-6:
            py = npy
            pygx = gen_pygx(py, dxxh, lams[k], sx, sy)
            npy = gen_npy(px, pygx, sy, sx)
        py = npy
        pxy = gen_pxy(pygx, px, sx, sy)
        rat = rate(pxy, entropy_2, entropy_1)
        D = gen_D(pxy, dxxh)
        rates.append(rat)
        Ds.append(D)
    return rates, Ds
```

```python
px = np.array([0.2, 0.8])
dxxh = np.array([[0, 1],
                 [1, 0]])
lams = np.arange(-5, 0, 0.1)
R, D = AB_Rate(px, dxxh, lams, gen_pygx, gen_npy, gen_pxy, rate, gen_D, entropy_2, entropy_1)
```

```python
def h(p):
    """
    Binary entropy function
    Input: one of two probability in the binary probability distribution
    Output: binary entropy of input probability
    """
    hp = -p*np.log(p)-(1-p)*np.log(1-p)
    return hp/np.log(2)
def maximum(a,b):
    """
    Choose maximum number from two inputs
    """
    if a < b:
        return b
    else:
        return a
```

```python
d = np.arange(0.01, 0.2, 0.01)
Rt = []
for i in range(len(d)):
    Rt.append(maximum(h(0.8)-h(d[i]),0))
```

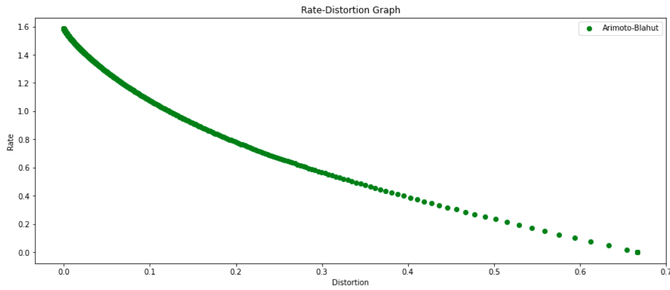
```python
plt.figure(figsize=(15,6))
plt.scatter(D,R, color ="green", label="Arimoto-Blahut", marker='o')
plt.scatter(d,Rt, color="red", label="Theory", marker='x')
plt.title("Rate-Distortion Graph")
plt.xlabel("Distortion")
plt.ylabel('Rate')
plt.legend()
plt.show()
```



(b) Using the same code for given input distribution and distortion matrix, we get:

```
px = np.array([1/3, 1/3, 1/3])
dxxh = np.array([[0, 1, 4],
                 [1, 0, 1],
                 [4, 1, 0]])
lams = np.arange(-20, 0, 0.01)
R, D = AB_Rate(px, dxxh, lams, gen_pygx, gen_npy, gen_pxy, rate, gen_D, entropy_2, entropy_1)
```

```
plt.figure(figsize=(15,6))
plt.scatter(D,R, color ="green", label="Arimoto-Blahut", marker='o')
plt.title("Rate-Distortion Graph")
plt.xlabel("Distortion")
plt.ylabel('Rate')
plt.legend()
plt.show()
```




```
print("Our best approximation of R(0) is:", R[0])
print("with D equal to:", D[0])
print("Theoretically, R(0) is log(3) =", np.log(3)/np.log(2))
```

```
Our best approximation of R(0) is: 1.584962417459905
with D equal to: 2.748204821421369e-09
Theoretically, R(0) is log(3) = 1.5849625007211563
```

```
print("Our calculated D* is:", D[R.index(min(R))])
print("with R equal to:", min(R), ", approximately 0")
```

```
Our calculated D* is: 0.6666666667542175
with R equal to: 1.7298503120600557e-14 , approximately 0
```

The theoretical value of $R(0)$ is $\log_2(3) \approx 1.584$. We can find it by vector source coding theorem that states

$$H(\mathsf{X}) \leq R \leq H(\mathsf{X}) + \epsilon \tag{1}$$

with ϵ can be made arbitrarily small positive number for source coding without error.

We can achieve $R(D^*) = 0$ for $D^* = 2/3$. The scheme to achieve it is by letting $\hat{\mathsf{X}} = 1$. Then, D^* comes from $p_{\mathsf{X}}(0)d(0,1) + p_{\mathsf{X}}(2)d(2,1) = 2/3$.