Home Assigment 4, CMPE 252, Section 01, Fall 2023.

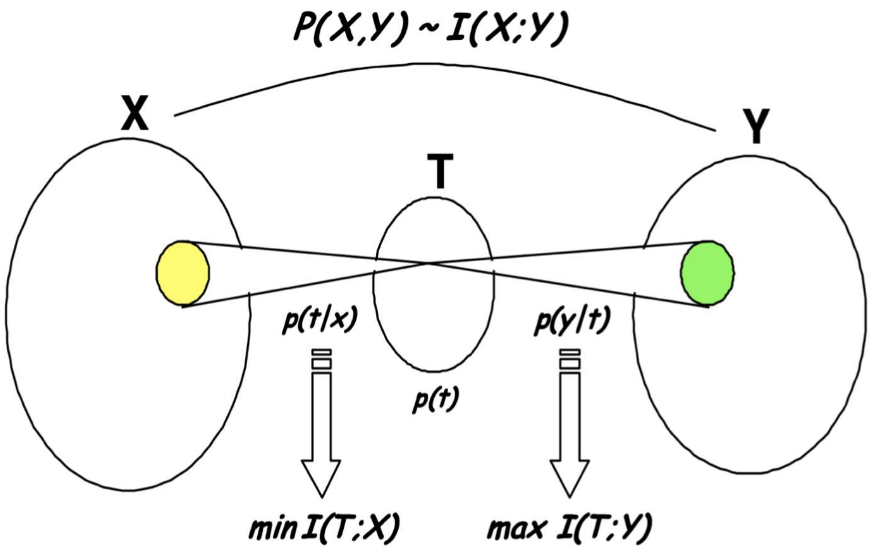
In this assigment you will get experience with the information bootleneck method, and its efficient implementation using broadcasting.

It is a team assigment. You can discuss your solutions but do not to share your code between the teams.

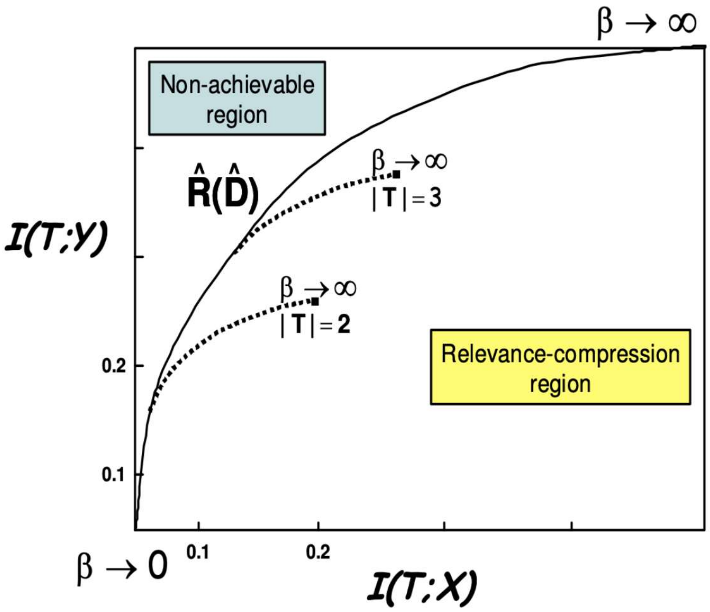
What to submit in Canvas: a working notebook with the full solution, and its corresponding PDF in two separate files (not a zip file).

Due: Dec 6, 11:59PM

Information Bottleneck



Information Bottleneck Curve



Information Bottleneck Self-Consistent equations

INIT

$$p(x, y) \leftarrow \text{ given}$$
$$p(t \mid x) \leftarrow \text{random init}$$
$$p(t) \leftarrow \sum_x p(t \mid x)p(x)$$
$$p(y \mid t) \leftarrow \frac{1}{p(t)} \sum_x p(t \mid x)p(x, y)$$

DO UNTIL CONVERGENCE

$$p(t \mid x) \leftarrow \frac{p(t)\exp\left(-\beta D_{kl}[p(Y \mid x)||p(Y \mid t)]\right)}{Z}$$
$$p(t) \leftarrow \sum_x p(t \mid x)p(x)$$
$$p(y \mid t) \leftarrow \frac{1}{p(t)} \sum_x p(t \mid x)p(x, y)$$

RETURN

$$p^*(t \mid x)$$

In [1]:

```
import numpy as np
from matplotlib import pyplot as plt
import math
from scipy.special import rel_entr
```

In [2]:
```
def DKL(A, B):
    """
    Kulback-Leibler divergence D(A||B)
    :param A: pa(x), target distribution
    :param B: pb(x), other distribution
    :return: component-wise DKL(pa(x) || pb(x)), which is a tensor of the same dimen
    :        each entry in the tensor is Ai * ln(Ai / Bi), which means the i-th comp
    :        this code structure, to return the component-wise Dkl, rather than sum_
    :        simplifies the update equations and the calculation of MI.
    :        you will use it in the calculation of mutual information and in the upd
    """
    epsilon = 1e-10
    A = np.clip(A, epsilon, 1 - epsilon)
    B = np.clip(B, epsilon, 1 - epsilon)


    # Calculate KL divergence component-wise
    d = A * np.log(A / B)

    # Handling numerical issues with inf/nan
    d[np.isnan(d)] = 0
    d[np.isinf(d)] = 0

    return d
```

In [3]:
```
def I(pA, pB, pAB):
    """mutual information I(X,Y) = DKL(P(x,y) || Px x Py)
    :param pA: A - p(a): marginal probability of X
    :param pB: B - p(b): marginal probability of Y
    :param pAB: p(a,b): joint probability of X and Y"""


    # your code (use your DKL function above)
    mutual_info = np.sum(DKL(pAB, pA * pB))

    return mutual_info
```

In [4]:
```
def entropy(p):
    """entropy of a discrete distribution"""
    ### example for the vectorized calculation of the entropy ###
    return -np.sum(p * np.log(p))
```

In [5]:
```
def make_probs(*dims):
    XY = np.random.rand(*dims)
    # normalize to a probability
    XY = XY / XY.sum()
    return XY
```

In [6]:
```
def iterative_information_bottleneck(
        Xdim, Ydim, Mmax, Mmin,
        n_iters=100, # iteration of the equations until convergence
        n_tries=3  , # number of trials from random initial conditions with n_iters
        n_betas=100, # number of different beta values to create the IB curve
        beta_min=0.1,#   the above results in 30000 itertions
        beta_max=100):

    # working with probabilities as tensors makes it easier to
    # read/write code and lets us utilize numpy broadcasting.
    # e.g, in this function we are working with discrete random
    # variables X, Y, and T. We can represent all probabilities as
    # tensors, where each dimension represents a random variable.
```

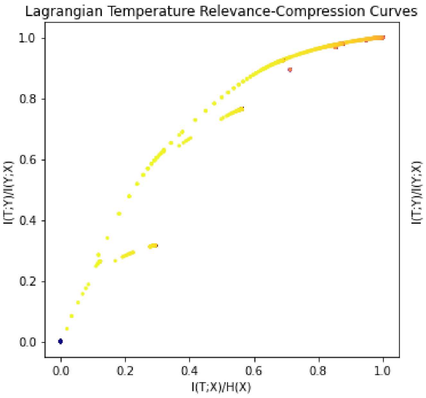
```
# e.g. p(x,y) is a tensor of shape (Xdim, Ydim, 1)
pXY = make_probs(Xdim, Ydim, 1)
# p(x) is a tensor of shape (Xdim, 1, 1)
pX = np.sum(pXY, axis=1, keepdims=True)/np.sum(pXY)
#print(pX.shape)
pY = np.sum(pXY, axis=0, keepdims=True)/np.sum(pXY)
#print("true")

# p(y | x) is a tensor of shape (Xdim, Ydim, 1), calculate it from pXY and Px
pY_X = pXY / pX

hX = entropy(pX)

target_MI = I(pX, pY, pXY)
print("The MI between generated X and Y is:", target_MI)
print("The entropy of X is:", hX)


# place hiolder for the Lagrangian = I_TXs - beta I_TYs
Ls = np.zeros((Mmax - Mmin+1, n_betas))
#epsilon = 1e-8
# relevance I(T;Y)
I_TYs = np.zeros((Mmax - Mmin+1, n_betas))
#epsilon = 1e-8
# compression I(T;X)
I_TXs = np.zeros((Mmax - Mmin+1, n_betas))

# betas
betas = np.zeros((Mmax - Mmin+1, n_betas))
# change the cardinality of the features, starting from |T| == |X|m and decreasi
for m, M in enumerate(range(Mmax, Mmin-1, -2)):
    for i, beta in enumerate(np.linspace(beta_min, beta_max, n_betas)[::-1]):

        L = np.inf
        I_TX = np.nan
        I_TY = np.nan
        for _ in range(n_tries):


            pTX = make_probs(Xdim, 1, M)

            pT_X = pTX / np.sum(pTX, axis=2, keepdims=True)  # Normalize to get 

            pT = np.sum(pT_X * pX, axis=0, keepdims=True)  # Calculate p(t) usin




            pY_T = np.sum(pY_X * (pT_X * pX/ pT) , axis=0, keepdims=True)

            for _ in range(n_iters):



                unnormalized_pT_X = pT * np.exp((-1) * beta * np.sum(DKL(pY_X, p


                sum_unnormalized_pT_X = np.sum(unnormalized_pT_X, axis=2, keepdi



                pT_X = unnormalized_pT_X / sum_unnormalized_pT_X
```

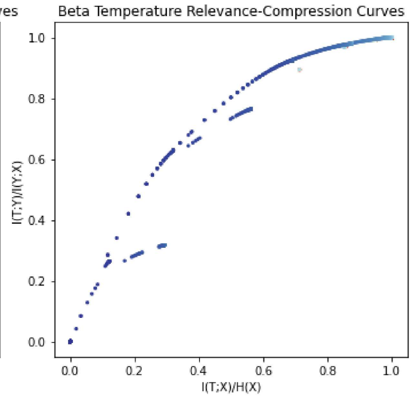
```
                    pT = np.sum(pT_X * pX, axis=0, keepdims=True)  # Recalculate p(t


                    pY_T = np.sum(pY_X * (pT_X * pX / pT), axis=0, keepdims=True)
                    #print(pY_T.shape)
                    I_TX_ = I(pT, pX, pT_X * pX)
                    I_TY_ = I(pT, pY, pY_T * pT)

                    L_ = I_TX_ - I_TY_ * beta   # calculate the objective of IB

                    if L_ < L: # find find the minimum within n_iters. we need it be
                        L = L_
                        I_TX = I_TX_
                        I_TY = I_TY_

            # save minimum L, corresponding beta, and mutual information tx and ty
            Ls[m, i] = L
            I_TXs[m, i] = I_TX
            I_TYs[m, i] = I_TY
            betas[m, i] = beta

    fig, axs = plt.subplots(1, 3, figsize=(15, 5))

    # relevance-compression curves
    axs[0].set_title("Lagrangian Temperature Relevance-Compression Curves")
    axs[0].set_xlabel("I(T;X)/H(X)")
    axs[0].set_ylabel("I(T;Y)/I(Y;X)")
    for i, (itx, ity, ls) in enumerate(zip(I_TXs, I_TYs, Ls)):
        axs[0].scatter(itx / hX, ity / target_MI, label=f"M:{Mmax - i}", s=5, c=ls,

    axs[1].set_title("Beta Temperature Relevance-Compression Curves")
    axs[1].set_xlabel("I(T;X)/H(X)")
    axs[1].set_ylabel("I(T;Y)/I(Y;X)")
    for i, (itx, ity, bs) in enumerate(zip(I_TXs, I_TYs, betas)):
        axs[1].scatter(itx / hX, ity / target_MI, label=f"M:{Mmax - i}", s=5, c=bs,

    axs[2].set_title("Relevance-Compression Curves")
    axs[2].set_xlabel("I(T;X)/H(X)")
    axs[2].set_ylabel("I(T;Y)/I(Y;X)")
    for i, (itx, ity) in enumerate(zip(I_TXs, I_TYs)):
        axs[2].scatter(itx / hX, ity / target_MI, label=f"M:{Mmax - i}", s=5)
    axs[2].legend()

    fig.tight_layout()
    plt.show()
```

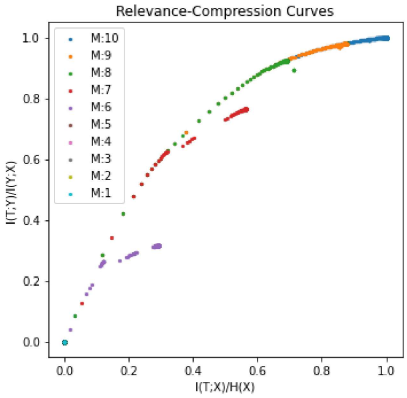
In [7]:

```
if __name__ == '__main__':
    iterative_information_bottleneck(
        Xdim=10,
        Ydim=5,
        Mmax=10, # the same cardinality as in the original X
        Mmin=1, # everything is collapsed to a single cluster
        n_iters=100,
        n_tries=100,
        n_betas=1000,
        beta_min=0.1,
        beta_max=1000
    )
```

```
The MI between generated X and Y is: 0.10646865253266291
The entropy of X is: 2.2506336580637005
```


Lagrangian Temperature Relevance-Compression Curves
I(T;Y)/I(Y;X)
I(T;X)/H(X)
0.0
0.2
0.4
0.6
0.8
1.0


Beta Temperature Relevance-Compression Curves
I(T;Y)/I(Y;X)
I(T;X)/H(X)
0.0
0.2
0.4
0.6
0.8
1.0


Relevance-Compression Curves
M:10
M:9
M:8
M:7
M:6
M:5
M:4
M:3
M:2
M:1
I(T;Y)/I(Y;X)
I(T;X)/H(X)
0.0
0.2
0.4
0.6
0.8
1.0