Homework 2

Zachary DeStefano, 15247592
CS 274B: Spring 2016

Problem 1:

Part A,B

Here is the code I used to produce the mutual information

```
#Part A
D = np.genfromtxt('data/data.txt',delimiter=None)
loc = np.genfromtxt('data/locations.txt',delimiter=None)
m,n = D.shape # m = 2760 data points, n=30 dimensional

#Part B

#Find phat for each variable
posXcount = np.sum(D,axis=0)
probXj = np.zeros((n,2))
probXj[:, 1] = np.divide(posXcount,m)
probXj[:, 0] = 1-probXj[:,1]

#Find phat(x_j,x_k) for each pair of variables
probXjk = np.zeros((n,n,2,2))
for i in range(n):
    for j in range(n):
        for k in range(m):
            probXjk[i, j, D[k, i], D[k, j]] += 1
probXjk = np.divide(probXjk,m)

#Compute mutual information
mutualInfo = np.zeros((n,n))
for i in range(n):
    for j in range(n):
        curTerm = 0
        for k1 in range(2):
            for k2 in range(2):
                denomTerm = probXj[i,k1]*probXj[j,k2]
                jointTerm = probXjk[i, j, k1, k2]
                if denomTerm>0 and jointTerm>0:
                    curTerm += jointTerm*np.log(jointTerm/denomTerm)
        mutualInfo[i, j] = curTerm

#print results
print mutualInfo[0:5,0:5]
```

Here are the first 5 rows and columns of mutual information

```
[[ 0.60870002  0.24584358  0.30645904  0.17521613  0.24642811]
 [ 0.24584358  0.66081843  0.28479996  0.21222061  0.29414659]
 [ 0.30645904  0.28479996  0.64609051  0.21486084  0.27873285]
 [ 0.17521613  0.21222061  0.21486084  0.68239551  0.23489376]
 [ 0.24642811  0.29414659  0.27873285  0.23489376  0.67286449]]
```

Part C

For the algorithm, I added edges in descending order by weight if they did not make a cycle. To compute this, I first initialized the edge information arrays:

```
edges = np.zeros(n*(n-1)/2)
edgeNodeID = np.zeros((n*(n-1)/2,2))
edgeInd = 0
for i in range(n):
    for j in range(i+1,n):
        edges[edgeInd] = mutualInfo[i, j]
        edgeNodeID[edgeInd,:] = [i, j]
        edgeInd+=1

sortedEdges = np.sort(edges)[::-1]
sortedEdgeID = np.argsort(edges)[::-1]
```

I made a function *nodesHavePath* that tells whether an edge will form a cycle

```
def nodesHavePath(adjMatrix,node0,node1):

    n,m = adjMatrix.shape
    adjMatrix = adjMatrix + np.transpose(adjMatrix)
    adjPower = np.copy(adjMatrix)
    adjTotal = np.copy(adjMatrix)
    for i in range(n):
        adjPower = np.dot(adjMatrix, adjPower)
        adjTotal += adjPower

    return (adjTotal[node0,node1]>0)
```

I then computed the adjacency matrix for the Chow-Liu Tree:

```
adjMatrix = np.zeros((n,n))
for curI in range(len(sortedEdges)):
    curEdgeID = sortedEdgeID[curI]
    curNode0 = edgeNodeID[curEdgeID, 0]
    curNode1 = edgeNodeID[curEdgeID, 1]
    if not nodesHavePath(adjMatrix,curNode0,curNode1):
        adjMatrix[curNode0,curNode1]=1
```

To print the edges, I made an adjacency list that does not repeat edges. Here is the code that makes the list and prints out the results

```
def getAdjList(adjMatrix):
    mm,xx = adjMatrix.shape
    visited = np.zeros((mm))
    adjList = []
    listVertices = []
    for i in range(mm):
        visited[i] = 1
        adjVertices = np.where(adjMatrix[i,:]>0)
        verticesAdd = []
        for j in adjVertices[0]:
            if visited[j] <= 0:
                verticesAdd.append(j)
        if len(verticesAdd) > 0:
            adjList.append(verticesAdd)
            listVertices.append(i)
    return listVertices,adjList

listVertices,adjList = getAdjList(adjMatrix)
for i in range(len(listVertices)):
    print listVertices[i],adjList[i]
```

Here is the printed result

```
0 [2]
1 [4]
2 [16, 17]
3 [5, 12, 29]
4 [6]
5 [6]
7 [10, 13]
8 [13]
9 [10]
10 [14]
11 [12, 14]
13 [15]
17 [18, 20]
18 [19]
20 [27]
21 [24, 25, 27]
22 [23, 26, 28]
27 [29]
28 [29]
```

Here is the code I used to produce the graph drawing of the Chow-Liu Tree

```
graph2 = nx.Graph()
graph2.add_nodes_from(range(n))
for i in range(len(listVertices)):
                for j in adjList[i]:
                                graph2.add_edge(listVertices[i],j)
loc2 = np.zeros(loc.shape)
loc2[:, 0] = loc[:, 1]
loc2[:, 1] = loc[:, 0]
nx.draw_networkx(graph2, node_color='c', pos=loc2)
plt.title('Weather Station Locations with Chow-Liu Tree')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.show()
```

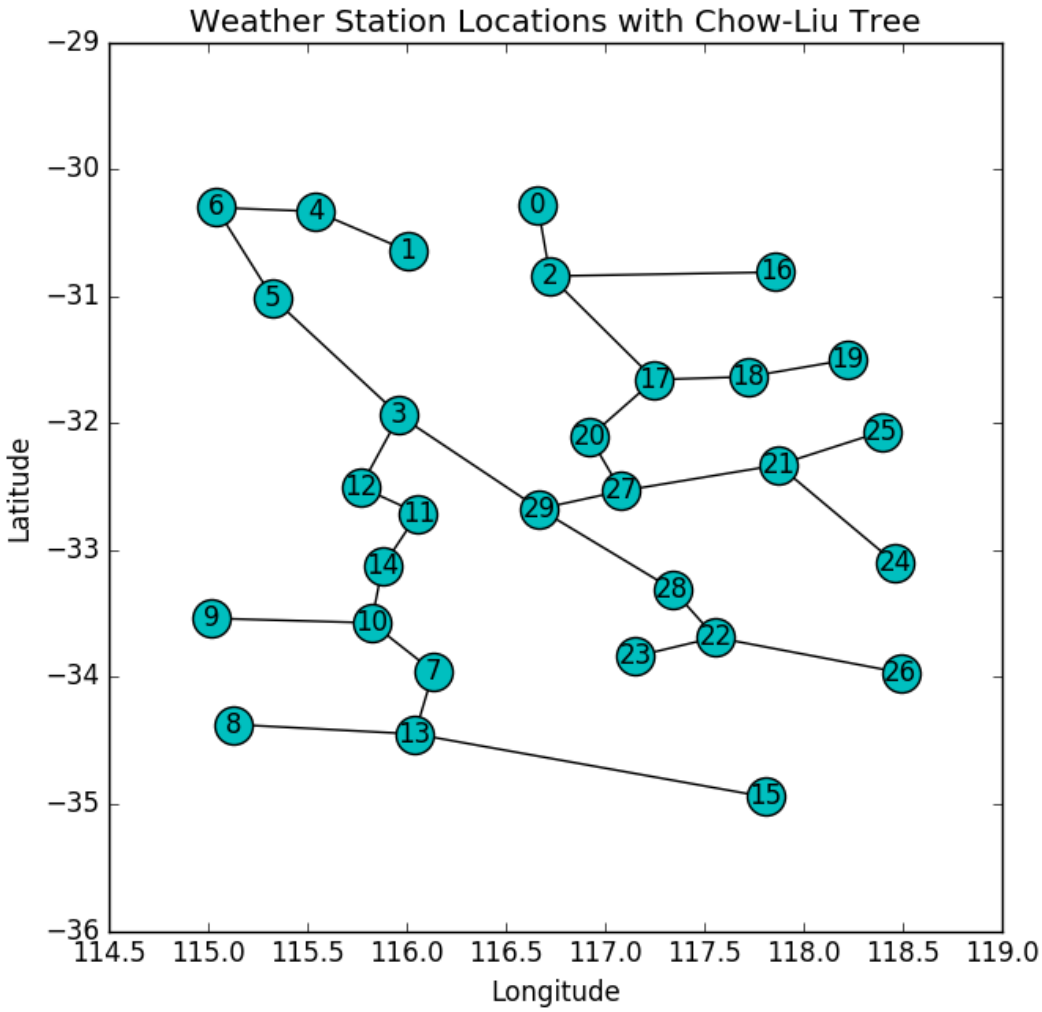


Figure 1: Chow-Liu Tree as computed in Part C

Part D

To calculate the likelihood, I used the method in Slide 6 of *Structure Learning in Bayes Nets* where you add together the entropy and mutual information. Here is the code:

```
#look at diagnoal of mutual info matrix for entropy
mutualInfoEntropy = np.zeros(n)
for i in range(n):
    mutualInfoEntropy[i] = mutualInfo[i,i]
entropyPart = mutualInfoEntropy.sum()

mutualInfoPart = 0
for jj in range(len(listVertices)):
    curJind = listVertices[jj]
    for kk in adjList[jj]:
        mutualInfoPart += mutualInfo[curJind,kk]

loglike = mutualInfoPart - entropyPart

print 'Log Likelihood:'
print loglike
```

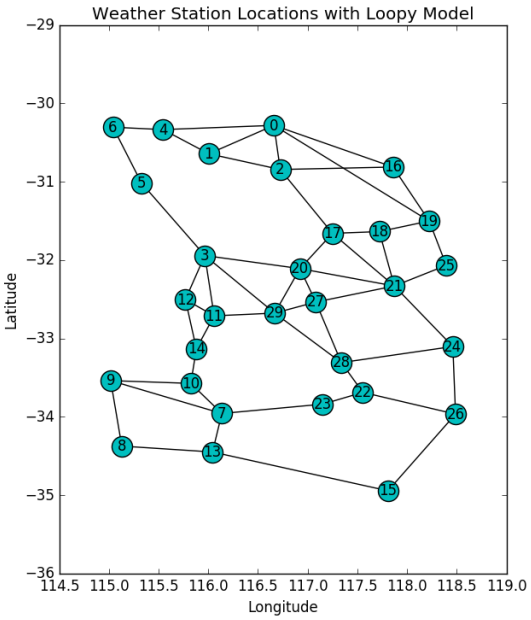
Here is the result

```
Log Likelihood:
-11.0986143278
```

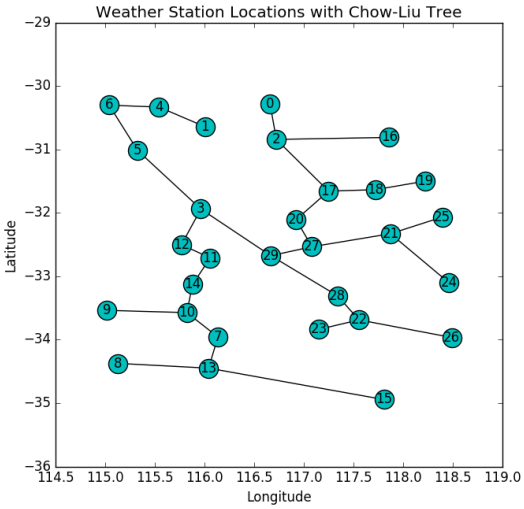
Problem 2:

Part A

Here is the loopy model compared with the Chow-Liu Tree Model. The loopy model seems more logical because it resembles a real-world network slightly better.



(a) Loopy Model



(b) Chow-Liu Tree Model

Figure 2: Loopy Model and Chow-Liu Tree Model side-by-side

Here is the code I used to generate the loopy model graph

```
edges = np.genfromtxt('data/edges.txt')
loc = np.genfromtxt('data/locations.txt',delimiter=None)

graph2 = nx.Graph()
graph2.add_nodes_from(range(n))
for edgeI in edges:
    graph2.add_edge(edgeI[0], edgeI[1])
loc2 = np.zeros(loc.shape)
loc2[:,0] = loc[:,1]
loc2[:,1] = loc[:,0]
nx.draw_networkx(graph2, node_color='c', pos=loc2)
plt.title('Weather Station Locations with Loopy Model')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.show()
```

Part B

Here is the code for the empirical marginal probabilities

```
probXjk = np.zeros((n,n,2,2))
for i in range(n):
    for j in range(n):
        for k in range(m):
            probXjk[i, j, D[k, i], D[k, j]] += 1
probXjk = np.divide(probXjk,m)

edgeMarginals = np.zeros((nEdges,2,2))
for ee in range(nEdges):
    node0 = edges[ee,0]
    node1 = edges[ee,1]
    edgeMarginals[ee,:,:] = probXjk[node0,node1,:,:]
print edgeMarginals[0:5]
```

Here are the first 5 empirical marginal probabilities

```
[[[ 0.59130435  0.11123188]
  [ 0.03514493  0.26231884]]

 [[ 0.62391304  0.07862319]
  [ 0.02826087  0.2692029 ]]

 [[ 0.57318841  0.12934783]
  [ 0.02717391  0.27028986]]

 [[ 0.66014493  0.0423913 ]
  [ 0.07427536  0.22318841]]

 [[ 0.65543478  0.04710145]
  [ 0.09637681  0.20108696]]]
```

Part C

The log partition function stayed constant with a value of 20.7944154168.
To run IPF, I first initialized the model:

```
edges = np.genfromtxt('data/edges.txt')
nEdges,cc = edges.shape

#initializes the factors given the loopy model graph we have
gmNodes = [gm.Var(i,2) for i in range(nEdges)]
probNodes = [gm.Var(i,2) for i in range(nEdges)]
gmFactors = []
probFactors = []
for ee in range(nEdges):
    jj = int(edges[ee,0])
    kk = int(edges[ee,1])
    gmFactors.append(gm.Factor([gmNodes[jj], gmNodes[kk]], 1.0))
    probFactors.append(gm.Factor([probNodes[jj], probNodes[kk]], 1.0))

    #fills the table with probabilities
    inputFactor = np.matrix(np.ones((2, 2)))
    gmFactors[ee].table = inputFactor
    probFactors[ee].table = probXjk[jj, kk, :, :]
```

After initialization, this is the code I used to run IPF:

```
sumElim = lambda F,Xlist: F.sum(Xlist)   # helper function for eliminate

numIter=15
totalEnt = numIter*nEdges
logLikeIter = np.zeros(numIter)
logLikeAll = np.zeros(totalEnt)
arrInd = 0
for iterI in range(numIter):
    print 'Now computing Iteration: ',iterI
    for ee in range(nEdges):
        # computes the likelihood of the current probabilistic model
        curLog = 0
        probModel = gm.GraphModel(probFactors)
        for ptNum in range(m):
            curLog += probModel.logValue(D[ptNum, :])
        curLog = -curLog / m
        logLikeAll[arrInd] = curLog
        arrInd += 1

        #current edge
        jj = int(edges[ee, 0])
        kk = int(edges[ee, 1])

        #does variable elimination to get p_jk value
        currentFactors = copy.deepcopy(gmFactors)
        curModel = gm.GraphModel(currentFactors)
        pri = [1.0 for Xi in currentFactors]
        pri[jj], pri[kk] = 2.0, 2.0
        order = gm.eliminationOrder(curModel,orderMethod='minfill',priority=pri)[0]
        curModel.eliminate(order[:-2], sumElim)  # eliminate all but last two
        curP = curModel.joint()
        curLnZ = np.log(curP.sum())
        print 'lnZ: ', curLnZ
        curP /= curP.sum()

        #update the current f_jk value
        currentFij = gmFactors[ee].table
        probRatio = np.matrix(np.divide(probXjk[jj,kk,:,:],curP.table))
        newFij = np.multiply(currentFij,probRatio)
        gmFactors[ee].table = newFij

        #update the probabilistic model
        newFijNorm = newFij/newFij.sum()
        probFactors[ee].table = newFijNorm

    logLikeIter[iterI] = curLog
```

Here is the negative log-likelihood after each iteration (starting from 0):

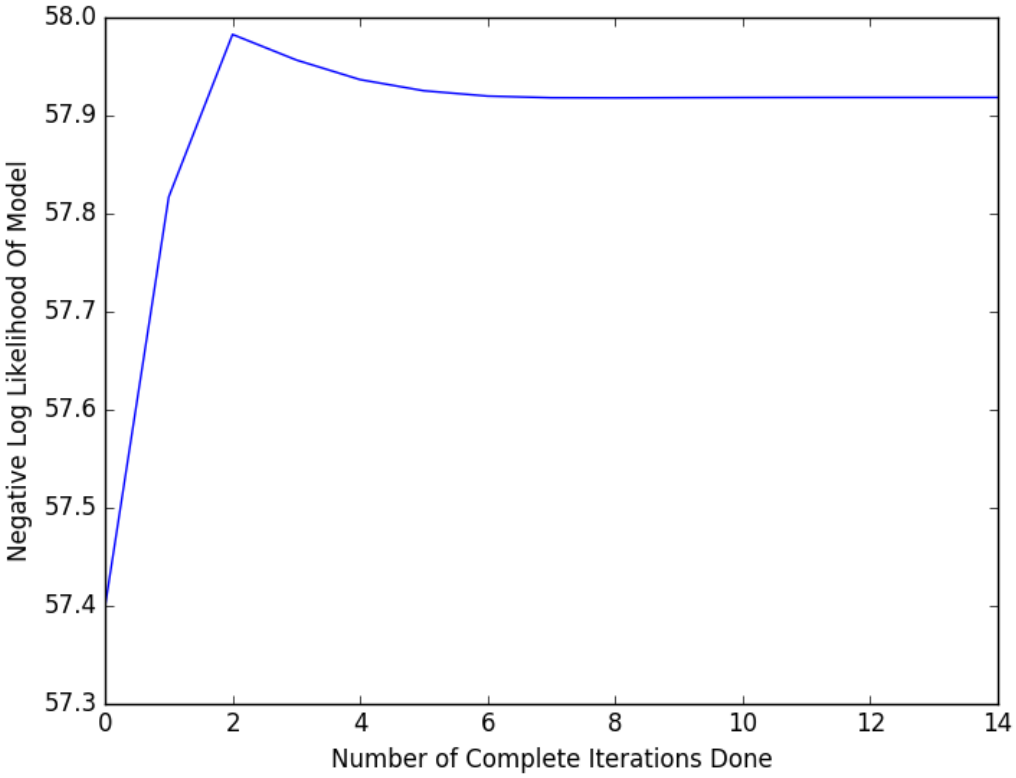


Figure 3: Neg LL after running IPF

It finally converged to a value of 57.91773114