DD2343 Assignment 2

jacmalm@kth.se

December 12, 2021

1 Dependencies in a Directed Graphical Model

1.1 In the graphical model of Figure 1, is $W_{d,n} \perp W_{d,n+1} \mid \theta_d, \beta_{1:K}$?

Yes.

1.2 In the graphical model of Figure 1, is $\theta_d \perp \theta_{d+1} \mid Z_{d,1:N}$?

No.

1.3 In the graphical model of Figure 1, is $\theta_d \perp \theta_{d+1} \mid \alpha, Z_{1:D,1:N}$?

Yes.

1.4 In the graphical model of Figure 2, is $W_{d,n} \perp W_{d,n+1} \mid \Lambda_d, \beta_{1:K}$?

No.

1.5 In the graphical model of Figure 2, is $\theta_d \perp \theta_{d+1} \mid Z_{d,1:N}, Z_{d+1,1:N}$?

No.

1.6 In the graphical model of Figure 2, is $\Lambda_d \perp \Lambda_{d+1} \mid \Phi, Z_{1:D,1:n}$?

No.

2 Likelihood of a Tree Graphical Model

2.1 Implement a dynamic programming algorithm that for a given T, Θ, β computes $p(\beta \mid T, \Theta)$

This algorithm is based on the derivation given in video 7.6b, module 7.

We start by defining a binary-tree shaped DGM, with root node A, and nodes B, C being children of A, and O_1, O_2 being both leaves and children of B.

Now, we can define a function

$$s(u,i) = p(O \cap \downarrow u \mid u = i)$$

Here O refers to the set of all observations (located on the leaves) and $\downarrow u$ denotes the set of nodes that are descendants of u.

We know that all nodes (and thus all observations) are descended from the root, thus $s(A,i)$ denotes the probability of all observations, given that the root takes on value i. We know that we can get the probability of only the observations from the conditional probability using the product rule and marginalization

$$s(A,i) = p(O \mid A = i)$$

$$p(O \mid A = i) = \frac{p(O, A = i)}{p(A = i)}$$

$$p(O) = \sum_A \frac{p(O, A = i)}{p(A = i)} p(A = i)$$

$$p(O) = \sum_i s(A,i) p(A = i)$$

Furthermore, we can define a recurrence relationship for s, using the conditional independence properties encoded in the DGM

$$s(A,i) = p(O\cap \downarrow B \mid A=i)p(O\cap \downarrow C \mid A=i)$$

"Unmarginalizing" either of these terms results in

$$p(O\cap \downarrow B \mid A=i) = \sum_j p(O\cap \downarrow B, B=j \mid A=i)$$

Using the product rule and conditional independence

$$\sum_j p(O\cap \downarrow B, B=j \mid A=i) = \sum_j p(O\cap \downarrow B \mid B=j)p(B=j \mid A=i)$$

Now, we recognize that

$$p(O\cap \downarrow B \mid B=j) = s(B,j)$$

Putting this together

$$s(A,i) = \sum_j s(B,j)p(B=j \mid A=i) \times \sum_k s(C,k)p(C=k \mid A=i)$$

We can also see that when we reach node B, who has children that are leaves, we reach our base case and the recursion ends.

$$s(B,i) = p(O_1 \mid B=i)p(O_2 \mid B=i)$$

Now, in order to calculate $p(O)$, we will need to calculate $s(A,i)$ with i taking on all K values. We also see that as the recursion becomes a layer removed from A we will calculate $s(B,j)$, whose computation does not depend on i. Thus we will do the same computations K times. As $s(B,j)$ is itself a recurrent function call with the same problem, these inefficiencies quickly add up. If we instead make use of dynamic programming, and save the result of each $s(u,i)$ that we compute in a map with (u,i) as the key, we will only need to calculate each item once. This makes our algorithm time complexity $O(Kn)$ where $n = ||nodes||$ as opposed to K^d where d is depth of the tree, a great improvement.

Code that implements this algorithm can be found in Appendix 1.

2.2 Report $p(\beta \mid T, \Theta)$ for each given data

2.2.1 Small tree

Sample: 0 Beta: [nan 2. nan 0. 2.] Likelihood: 0.009786075668602368

Sample: 1 Beta: [nan 3. nan 0. 0.] Likelihood: 0.015371111945909397

Sample: 2 Beta: [nan 1. nan 0. 0.] Likelihood: 0.02429470256988136

Sample: 3 Beta: [nan 0. nan 3. 4.] Likelihood: 0.005921848333806081

Sample: 4 Beta: [nan 3. nan 3. 3.] Likelihood: 0.016186321212555956

2.2.2 Medium tree

Sample: 0
Beta: [nan nan nan nan nan nan nan 1. 3. nan nan nan nan 4. nan nan 1. nan nan nan 0. nan nan 3. nan nan 0. nan 0. 4. nan nan 1. 0. 4. 0. 1. 1. nan 3. 3. 1. 0. 0. 2. 4. nan 1. 2. 3. 0.]
Likelihood: 1.7611947348905713e-18

Sample: 1
Beta: [nan nan nan nan nan nan nan 0. 0. nan nan nan nan 4. nan nan 3. nan nan nan 0. nan nan 3. nan nan 1. nan 3. 1. nan nan 4. 3. 1. 1. 4. 3. nan 3. 1. 0. 2. 4. 3. 2. nan 4. 4. 0. 0.]
Likelihood: 2.996933026124685e-18

Sample: 2
Beta: [nan nan nan nan nan nan nan 1. 3. nan nan nan nan 2. nan nan 2. nan nan nan 1. nan nan 4. nan nan 4. nan 4. 4. nan nan 2. 1. 2. 3. 0. 3. nan 3. 2. 2. 2. 1. 4. 2. nan 4. 3. 3. 4.]
Likelihood: 2.891411201505415e-18

Sample: 3
Beta: [nan nan nan nan nan nan nan 0. 0. nan nan nan nan 2. nan nan 4. nan nan nan 1. nan nan 3. nan nan 2. nan 4. 2. nan nan 4. 4. 2. 3. 0. 3. nan 3. 4. 1. 2. 3. 4. 2. nan 1. 0. 3. 0.]
Likelihood: 4.6788419411270006e-18

Sample: 4
Beta: [nan nan nan nan nan nan nan 4. 1. nan nan nan nan 0. nan nan 1. nan nan nan 1. nan nan 0. nan nan 1. nan 0. 1. nan nan 1. 1. 2. 4. 1. 0. nan 3. 0. 0. 3. 3. 3. 1. nan 2. 3. 3. 1.]
Likelihood: 5.664006737201378e-18

2.2.3 Large tree

Sample: 0
Likelihood: 3.63097513075208e-69
Sample: 1
Likelihood: 3.9405421986921234e-67
Sample: 2
Likelihood: 5.549061147187144e-67
Sample: 3
Likelihood: 9.89990102807915e-67
Sample: 4
Likelihood: 3.11420969368965e-72

3 Super Epicentra - Expectation-Maximization

3.1 Derive an EM algorithm for the model

The outline of the expectation maximization algorithm is given in section 9.3 in Pattern Recognition and Machine Learning, by Bishop [1]. The outline is summarized as follows:

Given a joint distribution $p(Y, Z \mid \theta)$ the goal is to maximize the likelihood $p(Y \mid \theta)$ w.r.t. θ.

1. Choose initial parameters θ
2. Evaluate $p(Z \mid Y, \theta^{old})$
3. Evaluate θ^{new} where

$$\theta^{new} = argmax_{\theta} Q(\theta, \theta^{old})$$

and

$$Q(\theta, \theta^{old}) = \sum_Z p(Z \mid Y, \theta^{old}) ln(p(Y, Z \mid \theta))$$

4. Check for convergence, we deem the algorithm to have converged if $p(Y \mid \theta^{new}) - p(Y \mid \theta^{old}) < \varepsilon$. If not converged, repeat from step 2. Otherwise θ^{new} are our parameter values.

In the outline above, Y denotes the set of observed variables, Z denotes the set of latent variables, and θ denotes the model parameters. Thus in the context of our model $Y = \{X, S\}$, $Z = \{Z\}$, and $\theta = \{\mu, \tau, \lambda, \pi\}$.

From inspection of the DGM, and taking note of the conditional independence properties encoded by it, we can see that the joint probability distribution of a single data point $n \in N$ is given by:

$$p(X_n, S_n, Z_n \mid \theta) = \prod_k p(Z_n) p(X_n \mid Z_n) p(S_n \mid Z_n)$$

Assuming that our data points are sampled independently, the probability of observing the entire dataset, or likelihood, is given by

$$p(X, S, Z \mid \theta) = \prod_n \prod_k p(Z) p(X \mid Z) p(S \mid Z)$$

$$p(X, S, Z \mid \theta) = \prod_n \prod_k \pi_k^{z_{nk}} (N(x_n \mid \mu_k, \tau_k^{-1}) P(s_n \mid \lambda_k))^{z_{nk}}$$

Where N and P denote the Normal and Poisson distribution respectively. The log-likelihood is thus given by

$$ln(p(X, S, Z \mid \theta)) = \sum_n \sum_k z_{nk} (ln(\pi_k) + ln(N(x_n \mid \mu_k, \tau_k^{-1})) + ln(P(s_n \mid \lambda_k))$$

In practice, a problem arises due to the fact that Z is not observed. This means that we cannot directly use $p(Z \mid Y, \theta^{old})$, but we instead use its expectation w.r.t its posterior distribution. Thus we introduce

$$\gamma(z_{nk}) = \mathbb{E}(z_{nk}) = \frac{\pi_k N(x_n \mid \mu_k, \tau_k^{-1}) P(s_n \mid \lambda_k)}{\sum_j \pi_j N(x_n \mid \mu_j, \tau_j^{-1}) P(s_n \mid \lambda_j)}$$

We then want to maximize

$$\mathbb{E}_{\mathbb{Z}}(ln(p(X, S, Z \mid \theta)) = \sum_n \sum_k \gamma(z_{nk})(ln(\pi_k) + ln(N(x_n \mid \mu_k, \tau_k^{-1})) + ln(P(s_n \mid \lambda_k))$$

Comparing this expression with the one given for only a Gaussian mixture model we can see that the difference is given by the added $ln(P(\lambda_k)$ term. Thus, when we derive the expression with respect to μ and σ, this term will disappear. This means that we can use the same solutions for approximating μ_{new}, σ_{new} as given by equations 9.19 and 9.21 in [1]. An analogous argument can be made for the estimation of π.

Using the same method as in the book for maximizing λ:

$$\lambda_k^{new} = argmax_\lambda \sum_n \gamma(z_{nk})(ln(\pi_k) + ln(N(x_n \mid \mu_k, \tau_k^{-1})) + ln(P(s_n \mid \lambda_k))$$

$$0 = \frac{\partial}{\partial \lambda_k} \sum_n \gamma(z_{nk})(ln(\pi_k) + ln(N(x_n \mid \mu_k, \tau_k^{-1})) + ln(P(s_n \mid \lambda_k))$$

$$0 = \sum_n \gamma(z_{nk}) \frac{\partial}{\partial \lambda_k} (ln(\pi_k) + ln(N(x_n \mid \mu_k, \tau_k^{-1})) + ln(P(s_n \mid \lambda_k))$$

$$0 = \sum_n \gamma(z_{nk}) \frac{\partial}{\partial \lambda_k} ln(P(s_n \mid \lambda_k))$$

Note that we are able to drop the summation over k due to the fact that if we reverse order of the summation we note that all the k terms are independent equally expressed additions to a quantity we wish to maximize (save for the value of λ_k), and thus we want to maximize all of them independently.

The Poisson distribution is defined as

$$P(\lambda) := f(s_n) = \frac{\lambda^{s_n} e^{-\lambda}}{s_n!}$$

Therefore

$$ln(P(\lambda) = s_n ln(\lambda) - \lambda - ln(s_n!)$$

and

$$\frac{\partial}{\partial \lambda_k}(P(\lambda) = \frac{\partial}{\partial \lambda}(s_n ln(\lambda) - \lambda - ln(s_n!)$$

$$\frac{\partial}{\partial \lambda}(P(\lambda) = \frac{s_n}{\lambda} - 1$$

Plugging this into our derivative of the expected log-likelihood

$$0 = \sum_n \gamma(z_{nk})(\frac{s_n}{\lambda_k} - 1)$$

$$0 = \sum_n (\frac{\gamma(z_{nk}) s_n}{\lambda_k} - \gamma(z_{nk}))$$

$$\sum_n \gamma(z_{nk}) = \sum_n \frac{\gamma(z_{nk}) s_n}{\lambda_k}$$

$$\lambda_k = \frac{\sum_n \gamma(z_{nk}) s_n}{\sum_n \gamma(z_{nk})}$$

Observe that this equation is of the same form as the equation for updating mu, except that we use s_n instead of x_n.

3.2 Implement your EM algorithm

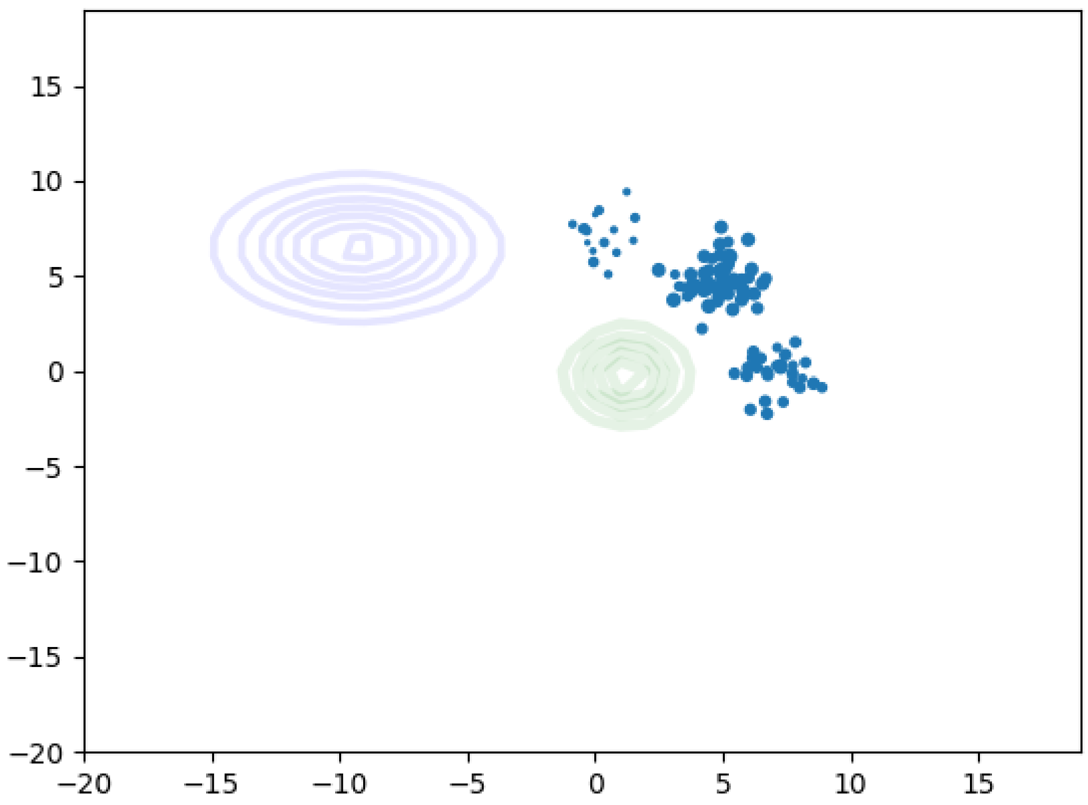
Appendix 2.

3.3 Apply it to the data provided, give an account of the success and provide visualizations for a couple examples. Repeat it for a few different choices of K.

I used the directions for plotting the contours of the K different models as provided on the course webpage. Thus I plotted contours for the different Normal distributions, with linewidth signifying the "strength" of the super epicentra, as given by the Poisson parameter.

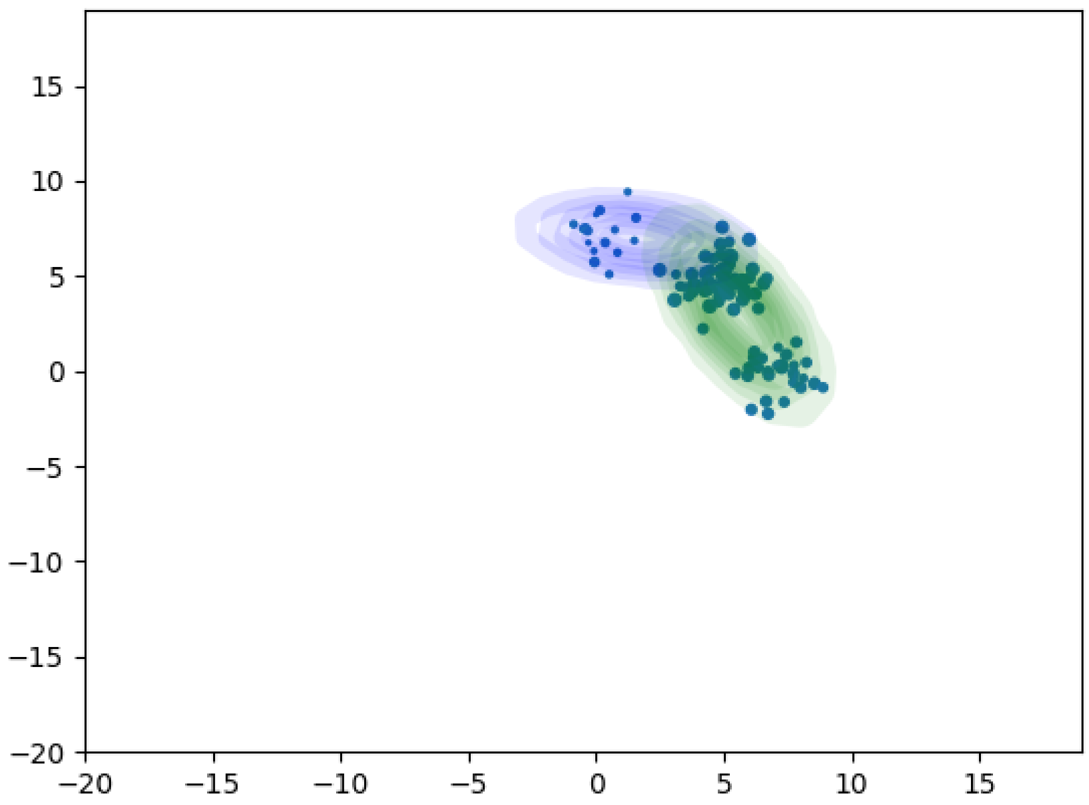
I superimposed this plot onto a scatter plot of our data points, where the size of the data point is given by that data points S value.

I began by fitting the data points with $K = 2$. The models initially looked like this:

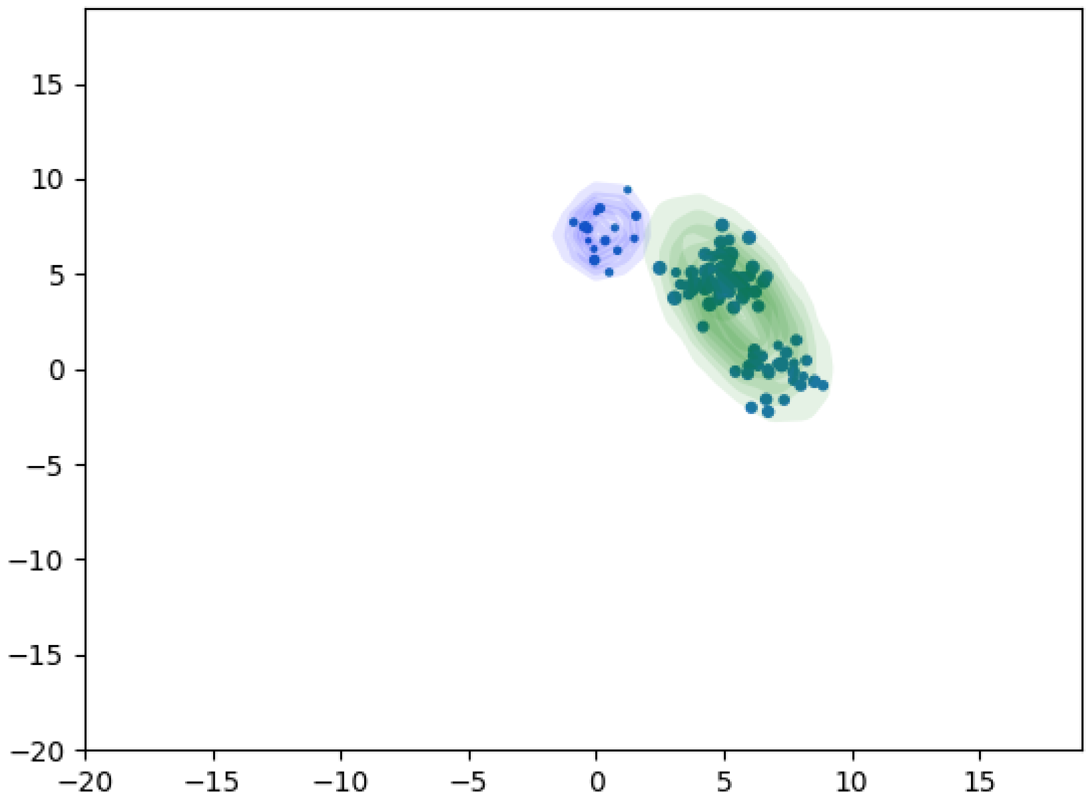


Optically, we can determine that there are 3 clusters formed by the data points, and as expected our initial clusters do not seem well placed at all.

After 1 iteration however, the clusters are already quite well fitted to the data, with the exception being that the middle cluster is ”split” between the two distributions.

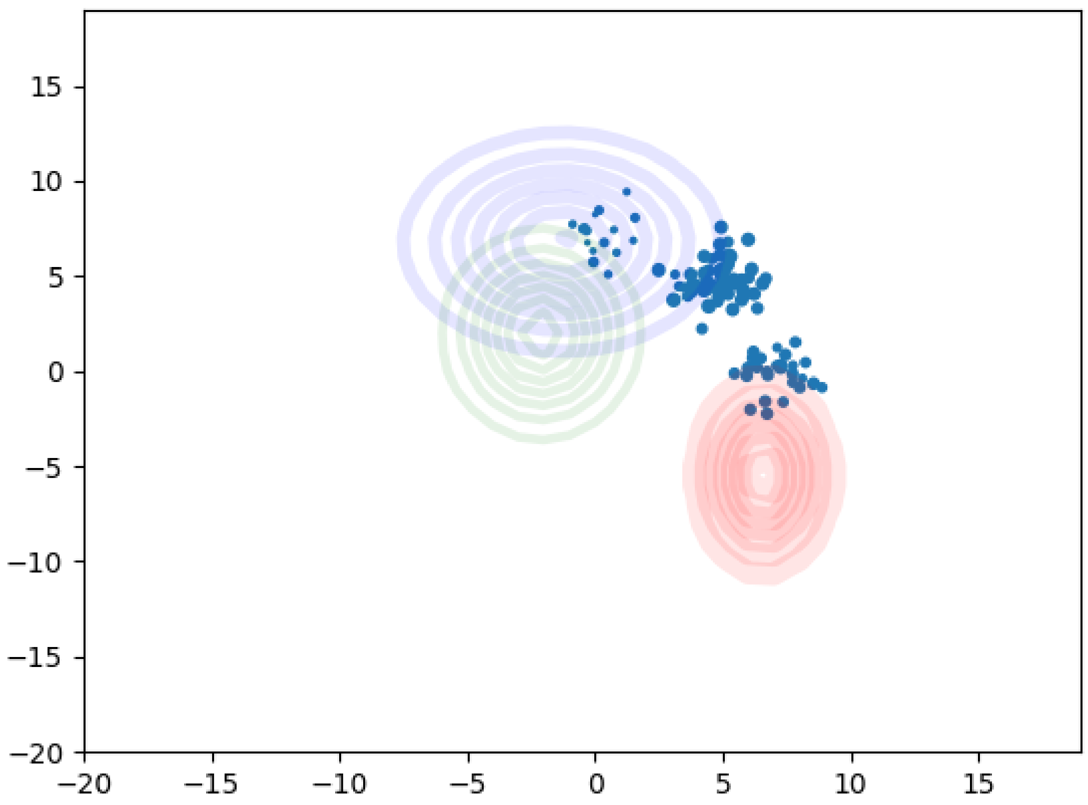


And by the end, the clusters are about as well fitted as they can be

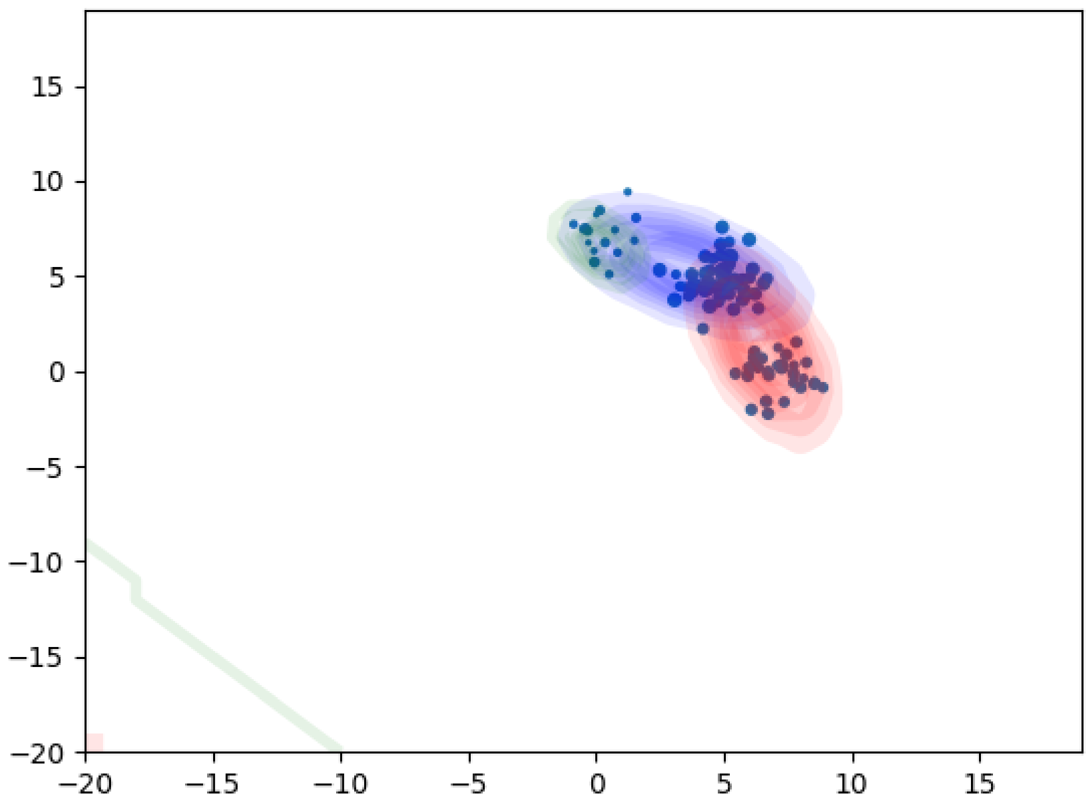


We can see that the algorithm attempts to make up for the deficiency we have in setting $K = 2$ by making the variance of the of the second distribution larger, so that it encompasses both of the clusters not attributed to the first distribution, that also happen to be closer together.

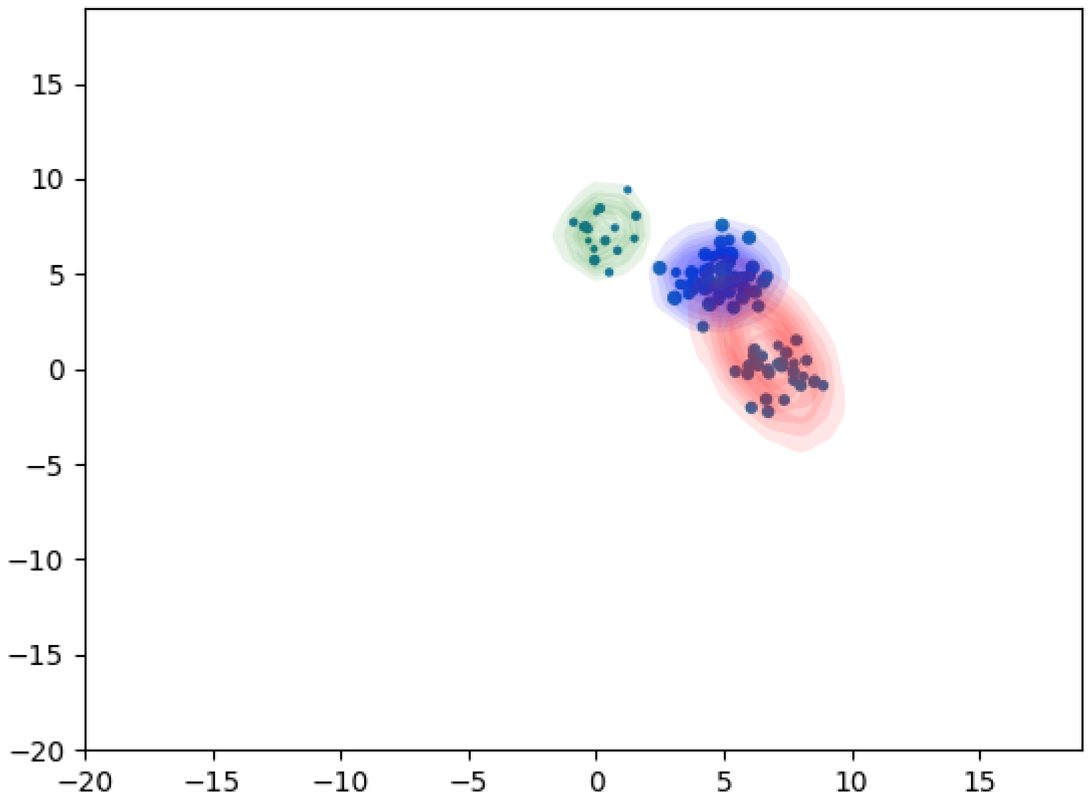
If we set $K = 3$ and try again, we see quite good results, as expected from our observations of the previous experiment.

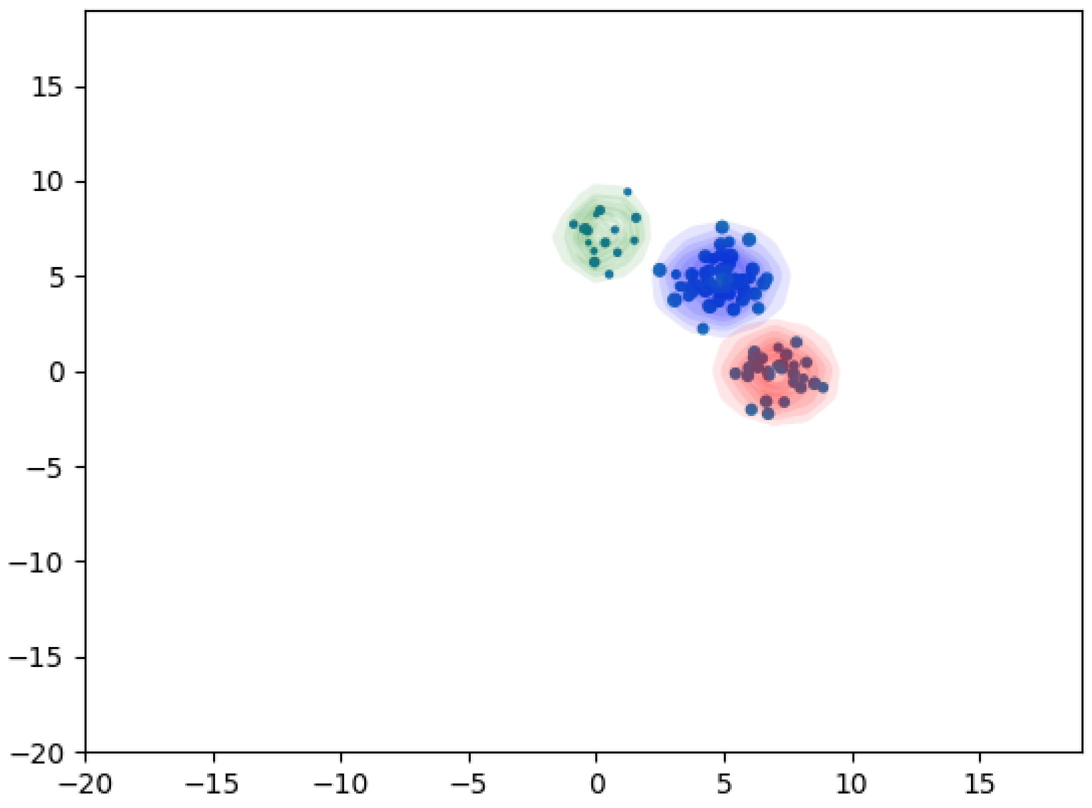


15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15



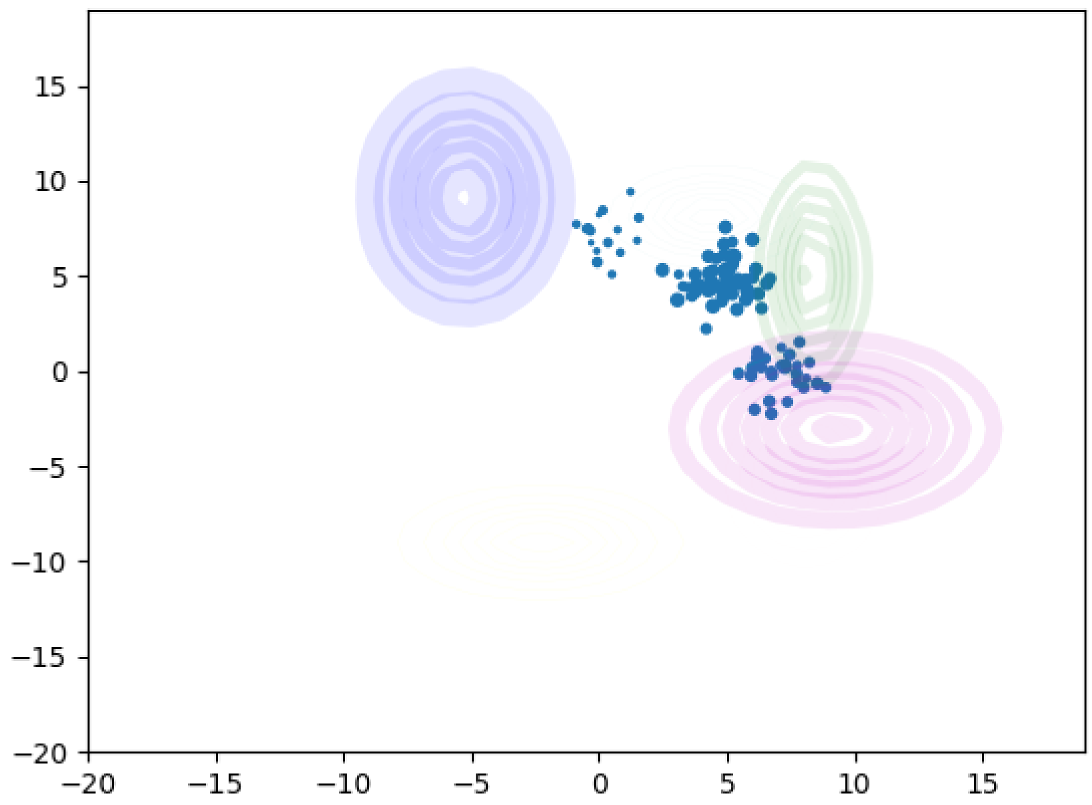
15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15

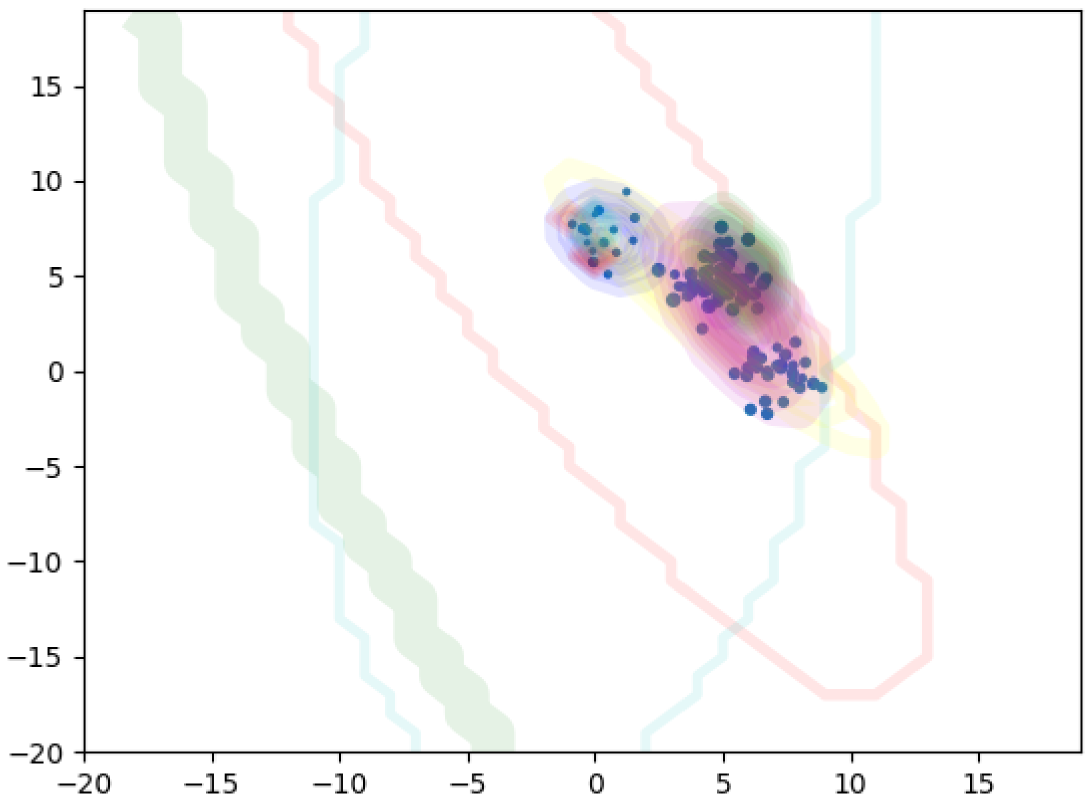

15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15

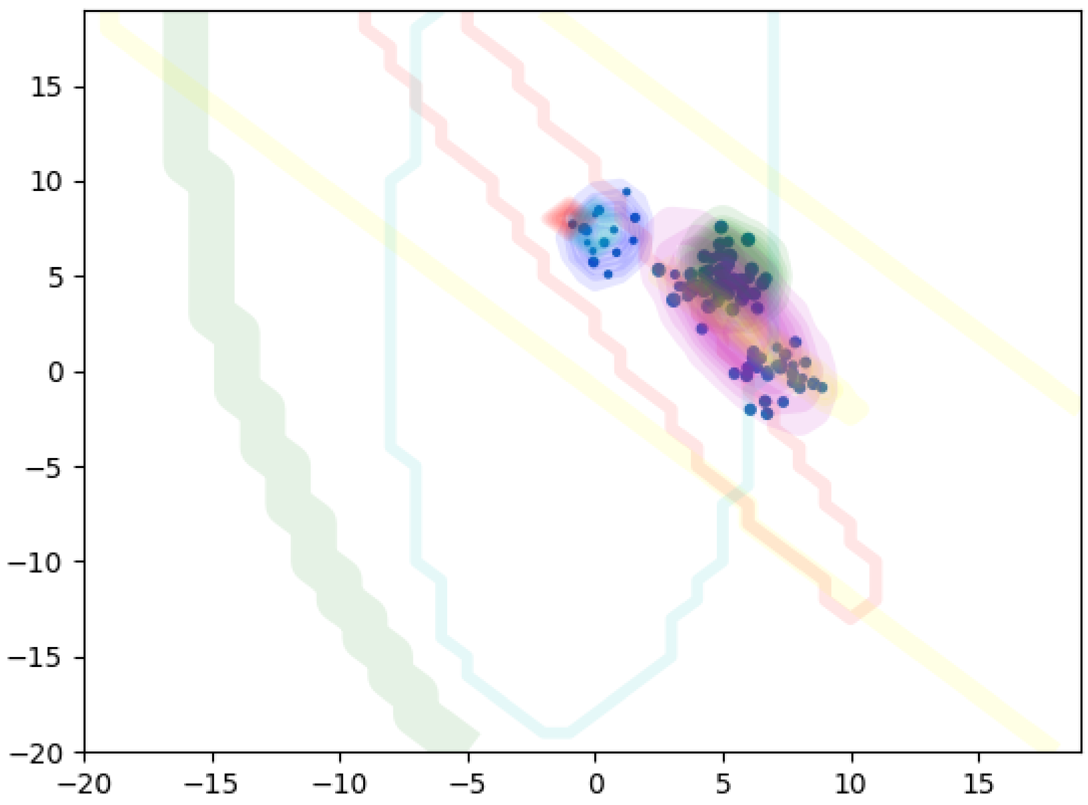


By the end, the three distributions are well fitted to the three clusters, and the line widths seem correlated to that clusters average size of data points.

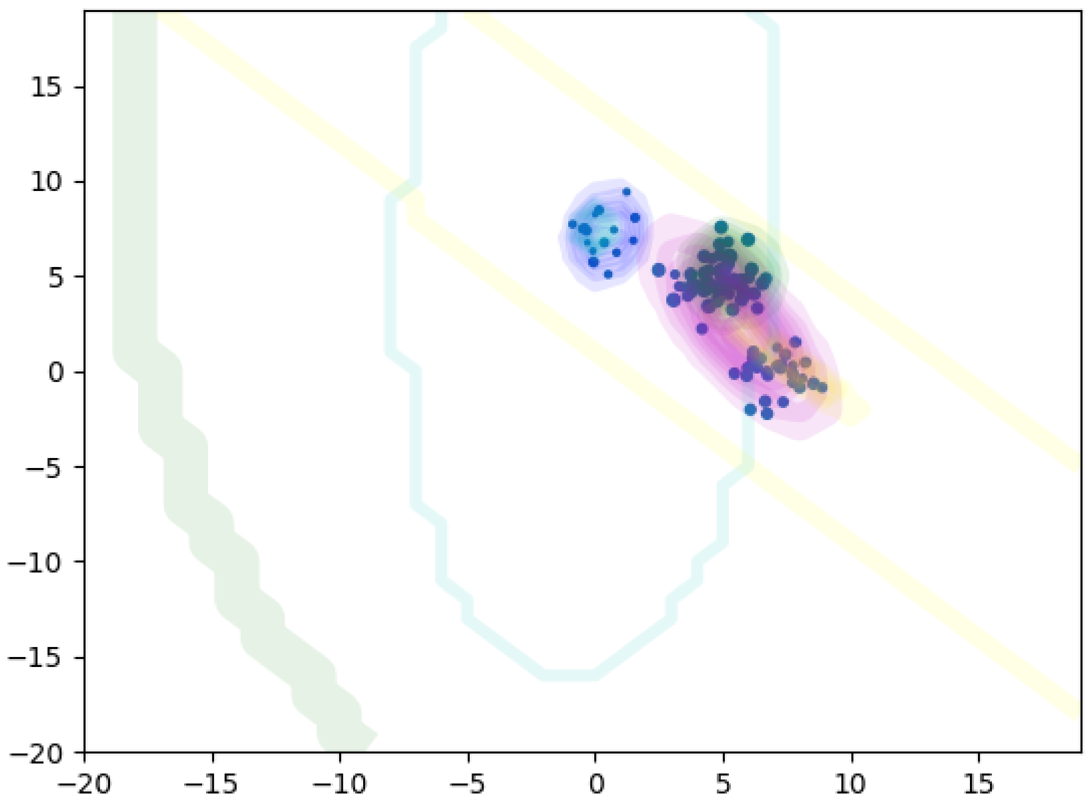
An interesting thing happens when we set $K = 6$. Of course, there is a lot more chaos, and some distributions end up ”sharing” the clusters.


15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15


15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15



15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15



However, if we look at the difference between the second last and last iteration, we can see that one distribution has disappeared. Also, I noticed that when I ran the code for $K = 6$, seemingly randomly I would get an error about not being able to create the Normal distributions with our updated parameters.

When I began debugging, I noticed that when this would happen, we attempted to instantiate a Normal distribution with certain variance values being equal to zero, and scipy would throw an error. Passing in "allow_singular=True" to the constructor of the Normal distribution fixed this.

This explains why in the images, one distribution disappears. I believe this is caused by the "vanishing variance" problem where if the mean of one cluster is exactly equal to one of the data points, setting its variance to zero contributes an infinite term to our log likelihood calculation, as mentioned in the book [1].

4 Appendix

4.1 Appendix 1

```
""" This file is created as a suggested solution template for question 2.2 in DD24

    We encourage you to keep the function templates.
    However this is not a "must" and you can code however you like.
    You can write helper functions etc however you want.

    If you want, you can use the class structures provided to you (Node and Tree c
    file), and modify them as needed. In addition to the data files given to you,
    test your algorithm with your own simulated data for various cases and analyse
    For those who do not want to use the provided structures, we also saved the pr
    format. Let us know if you face any problems.

    Also, we are aware that the file names and their extensions are not well-forme
    (i.e example_tree_mixture.pkl_samples.txt). We wanted to keep the template cod
    You can change the file names however you want.

    For this task, we gave you three different trees (q2_2_small_tree, q2_2_medium
    Each tree has 5 samples (the inner nodes' values are masked with np.nan).
    We want you to calculate the likelihoods of each given sample and report it.

    Note:   The alphabet "K" is K={0,1,2,3,4}.

    Note:   A VERY COMMON MISTAKE is to use incorrect order of nodes' values in CP
            theta is a list of lists, whose shape is approximately (num_nodes, K, 
            For instance, if node "v" has a parent "u", then p(v=Zv | u=Zu) = thet

            If you ever doubt your useage of theta, you can double check this marg
            \sum_{k=1}^K p(v = k | u=Zu) = 1
"""

import numpy as np
import itertools
from Tree import Tree
from Tree import Node
import copy

K = 5

DYNAMICU_PROGRAMMINGGU = {}


def calculate_likelihood(tree_topology, theta, beta):
    """
    This function calculates the likelihood of a sample of leaves.
    :param: tree_topology: A tree topology. Type: numpy array. Dimensions: (num_no
    :param: theta: CPD of the tree. Type: list of numpy arrays. Dimensions (approx
    :param: beta: A list of node assignments. Type: numpy array. Dimensions: (num_
                Note: Inner nodes are assigned to np.nan. The leaves have values i
    :return: likelihood: The likelihood of beta. Type: float.

    This is a suggested template. You don't have to use it.
    """
    _, leaves = partition_leaves(beta)
    likelihood = 0
    DYNAMICU_PROGRAMMINGGU.clear()
    for i in range(K):
        likelihood += s(0, i, theta, tree_topology, leaves, beta) * theta[0][i]
    return likelihood

# returns the probability of all observations underneath node, given that node tak
def s(node, i, theta, tree_topology, leaves, beta):
    key = (node, i)
    if key in DYNAMICU_PROGRAMMINGGU:
        return DYNAMICU_PROGRAMMINGGU[key]

    left_child, right_child = get_children(node, tree_topology)
    left_recursion = 0
    right_recursion = 0
    #base case
    if left_child in leaves:
        left_recursion = theta[left_child][i][int(beta[left_child])]
    else:
        for j in range(K):
            left_recursion += s(left_child, j, theta, tree_topology, leaves, beta)

    if right_child in leaves:
        right_recursion = theta[right_child][i][int(beta[right_child])]
    else:
        for k in range(K):
            right_recursion += s(right_child, k, theta, tree_topology, leaves, bet
    return_val = left_recursion * right_recursion
    DYNAMICU_PROGRAMMINGGU[key] = return_val
    return left_recursion * right_recursion


def get_children(node, tree_topology):
    children = []
    for index, element in enumerate(tree_topology[node::]):
        if element == int(node):
            children.append(index + node)
    return children


def partition_leaves(beta):
    leaves = []
    latent_vertices = []
    for index, node in enumerate(beta):
        if np.isnan(node):
            latent_vertices.append(index)
        else:
            leaves.append(index)
    return latent_vertices, leaves


def main():
    print("Hello␣World!")
    print("This␣file␣is␣the␣solution␣template␣for␣question␣2.2.")

    print("\n1.␣Load␣tree␣data␣from␣file␣and␣print␣it\n")

    filename = "data/q2_2_small_tree.pkl"  # "data/q2_2_medium_tree.pkl", "data/q2
    print("filename:␣", filename)

    t = Tree()
    #t.create_random_binary_tree(seed_val=0, k=2, num_nodes=4)
    t.load_tree(filename)
    t.print()
    t.sample_tree(num_samples=10000, seed_val=0)
    print("K of the tree: ", t.k, "\talphabet: ", np.arange(t.k))

    print("\n2. Calculate likelihood of each FILTERED sample\n")
    # These filtered samples already available in the tree object.
    # Alternatively, if you want, you can load them from corresponding .txt or .np

    for sample_idx in range(t.num_samples):
        beta = t.filtered_samples[sample_idx]
        print("\n\tSample: ", sample_idx, "\tBeta: ", beta)
        theta = copy.deepcopy(t.get_theta_array())
        calculated_likelihood = calculate_likelihood(t.get_topology_array(), theta
        print("\tLikelihood: ", calculated_likelihood)
        sample_likelihood = estimate_likelihood_from_samples(t, beta)
        print("\tSample Likelihood: ", sample_likelihood)




def estimate_likelihood_from_samples(tree, beta):
    count = 0
    filtered_samples = tree.filtered_samples
    for sample in filtered_samples:
        if np.array_equal(sample, beta, equal_nan=True):
            count += 1
    return count / tree.num_samples


if __name__ == "__main__":
    main()
```

4.2 Appendix 2

```
import numpy as np
import random
from scipy.stats import multivariate_normal as normal_distribution
from scipy.stats import poisson as poisson_distribution
import matplotlib.pyplot as plt

K = 6


def main():
    X, S = read_in_data('X.txt', 'S.txt')
    estimate_parameters(X, S)



def estimate_parameters(X_data, S_data):
    pi, normals, poissons = initialize_models()
    visualize_data_and_models(X_data, S_data, normals, poissons)
    log_likelihood = calculate_log_likelihood(X_data, S_data, pi, normals, poisson
    next_likelihood = 0
    while next_likelihood > log_likelihood:
        if next_likelihood != 0:
            log_likelihood = next_likelihood
        gammas = calculate_gammas(X_data, S_data, pi, normals, poissons)
        pi, normals, poissons = re_estimate_parameters(X_data, S_data, gammas)
        next_likelihood = calculate_log_likelihood(X_data, S_data, pi, normals, po
        visualize_data_and_models(X_data, S_data, normals, poissons)
    return pi, normals, poissons


def visualize_data_and_models(X_data, S_data, normals, poissons):
    col = ['blue', 'green', 'red', 'yellow', 'm', 'c']
    x, y = np.mgrid[-20:20, -20:20]
    pos = np.dstack((x, y))
    for k in range(K):
        Z = normals[k].pdf(pos)
        plt.contour(x, y, Z, colors=col[k], linewidths=poissons[k].mean(), alpha=0
    plt.scatter(X_data[:, 0], X_data[:, 1], s=S_data)
    plt.show()


def calculate_log_likelihood(X_data, S_data, pi, normals, poissons):
    N = len(X_data)
    log_likelihood = 0
    for n in range(N):
        nth_addition = 0
        for k in range(K):
            nth_addition += pi[k] * normals[k].pdf(X_data[n]) * poissons[k].pmf(S_
        nth_addition = np.log(nth_addition)
        log_likelihood += nth_addition
    return log_likelihood


def re_estimate_parameters(X_data, S_data, gammas):
    N = len(X_data)
    mus = np.zeros((K, 2))
    sigmas = np.zeros((K, 2, 2))
    poisson_params = np.zeros(K)
    pi = np.zeros(K)
    normals = []
    poissons = []
    for k in range(K):
        for n in range(N):
            # store values of n_k into pi_k
            pi[k] += gammas[k][n]
            mus[k] += gammas[k][n] * X_data[n]
            poisson_params[k] += gammas[k][n] * S_data[n]
        mus[k] /= pi[k]
        poisson_params /= pi[k]
        for n in range(N):
            # numpy fuckry needed in order to transpose
            temp = np.zeros((1, 2))
            temp[0] = X_data[n] - mus[k]
            sigmas[k] += gammas[k][n] * np.matmul(np.transpose(temp), temp)
        sigmas[k] /= pi[k]
        normal = normal_distribution(mean=mus[k], cov=sigmas[k], allow_singular=Tr
        normals.append(normal)
        poisson = poisson_distribution(poisson_params[k])
        poissons.append(poisson)
        pi[k] /= N
    return pi, normals, poissons


def calculate_gammas(X_data, S_data, pi, normals, poissons):
    N = len(X_data)
    gammas = np.zeros((K, N))
    for k in range(K):
        for n in range(N):
            nominator = pi[k] * normals[k].pdf(X_data[n]) * poissons[k].pmf(S_data
            denominator = 0
            for j in range(K):
                denominator += pi[j] * normals[j].pdf(X_data[n]) * poissons[j].pmf
            gammas[k][n] = nominator / denominator
    return gammas


def initialize_models():
    pi = np.zeros(K)
    normals = []
    poissons = []
    for k in range(K):
        pi[k] = random.uniform(0, 1)
        # initialize poisson distributions
        poisson_param = random.uniform(0, 10)
        poisson = poisson_distribution(poisson_param)
        poissons.append(poisson)
        # initialize normal distributions
        mu = np.zeros(2)
        sigma = np.zeros((2, 2))
        for i in range(2):
            mu[i] = random.uniform(-10, 10)
            sigma[i][i] = random.uniform(1, 10)
        normal = normal_distribution(mean=mu, cov=sigma)
        normals.append(normal)
        # enforce probabilities adding up to 1
        sum = np.sum(pi)
        pi /= sum
    return pi, normals, poissons


def read_in_data(X_file, S_file):
    X = []
    S = []
    with open(X_file) as f:
        for data_point in f:
            data_point = data_point.rstrip()
            coord1, coord2 = data_point.split(sep=' ')
            X.append([float(coord1), float(coord2)])
    with open(S_file) as f:
        for data_point in f:
            data_point = data_point.rstrip()
            S.append(float(data_point))
    assert len(X) == len(S)
    return np.array(X), np.array(S)


main()
```

References

[1] *Pattern Recognition and Machine Learning.* Springer, 2006.