COMP0080 Graphical Models
Assignment 2

Daniel May
ucabdd3@ucl.ac.uk

Olivier Kraft
ucabpk0@ucl.ac.uk

Daniel O'Hara
daniel.o'hara.20@ucl.ac.uk

Stephen O'Brien
stephen.o'brien.20@ucl.ac.uk

Umais Zahid
umais.zahid.16@ucl.ac.uk

December 15, 2020

Contents

1 Question 1

1.1 Systematic Encoding

The code below takes a matrix H and performs Gaussian Elimination in $\mathbb{F}_2$ which returns a matrix R of the form $[I\ P]$. It then performs column permutations to get it in the form $[P\ I]$ and returns both this matrix, $\hat{R}$, and the permutations required.

The systemic encoding G is then formed from $\hat{R}$ and $\hat{H}$ is formed by applying the column permutations.

```
import numpy as np
def swap(b):
    """
    Function to change rref(H) from [I, P] to [P I]
    """
    n = len(b)
    swaps = np.arange(b.shape[1])
    swaps[-n:] = np.arange(n)
    newb = b[:, swaps]
    swaps[:b.shape[1]-n] = np.arange(n, b.shape[1], 1)
    return newb, swaps

def rref(H):
    """
    Function to convert matrix H to reduced row echelon form in base 2
    """
    X = H.copy()
    n = len(X)
    for i in range(n-1):
        j = i
        while X[j,i] == 0:
            j += 1
            if j == n:
                print(f"Matrix is not of rank {n}")
                return None
        temp = X[i].copy()
        X[i] = X[j].copy()
        X[j] = temp.copy()
        for k in range(i+1, n):
            if X[k,i] != 0:
                X[k] = (X[k] + X[i])%2
    for i in range(n-1, 0, -1):
        for j in range(i-1, -1, -1):
            if X[j,i] != 0:
                X[j] = (X[j] + X[i]) % 2

    return X

def getG(Z):
    """
    Final function to get RREF(H), perform column permutations and
    return Hhat and G
    """
    X = Z.copy()
    r = rref(X)
    rhat, swaps = swap(r)
```

```
    n, k = X.shape
    G = np.vstack([np.eye(k-n), rhat[:, :k-n].reshape(n, k-n)])
    Hhat = Z[:, swaps]
    return Hhat, G
```

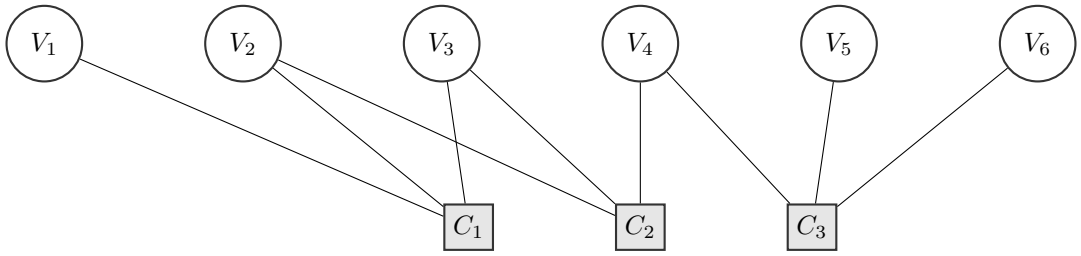
For the given H, the code returned the following $\hat{H}$ and G.

$$\hat{H} = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \end{pmatrix} \tag{1}$$

$$\tag{2}$$

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \tag{3}$$

1.2 Factor Graph



1.3 LDPC

The following code takes a parity check matrix H and a received word y, and performs Loopy Belief Propagation. It returns the decoded message attempt, whether the decoding was successful and the number of iterations required.

```
import numpy as np
H1 = np.loadtxt("H1.txt", delimiter = " ")
H = H1
y1 = np.loadtxt("y1.txt", delimiter = " ")
G = np.loadtxt("sys_G.txt", delimiter = " ")
def ftov(H, M):
    """
    Factor to variable message passing
    """
    m, n = H.shape
    fM= np.zeros((m, n))
    for i in range(m):
        o = np.where(H[i] == 1)[0]
        for j in o:
            # if H[i,j] == 1:
            result = np.prod((np.exp(M[i,o]) - 1)/(np.exp(M[i,o]) + 1))/((np.exp(M[i,j]) - 1)/(np.exp(M[i,j])
            ↪  + 1))
```

```
            fM[i, j] = np.log((1+result)/(1-result))
    return fM
def updateM(r, M, f, H):
    """
    Update Message
    """
    m, n = M.shape
    for i in range(n):
        for j in range(m):
            if H1[j, i] == 1:
                result = np.sum(f[:,i]) - f[j,i]
                M[j,i] = r[i] + result
  return M

def F(H, y, p = 0.1, maxiter = 20):
    """
    Loopy Belief Propagation
    """
    P = np.zeros(len(y))
    P[y == 1] = np.log(p) - np.log(1-p)
    P[y == 0] = - np.log(p) + np.log(1-p)
    M = np.zeros_like(H)
    n = len(H)
    for i in range(n):
        M[i][H[i]==1] = P[H[i]==1]
    success = -1
    for i in range(maxiter):
        f = ftov(H, M)
        v = P + np.sum(M, axis = 0)
        decoded = np.array(v <= 0).astype("float")
        M = updateM(P, M, f, H)
        if np.all(H.dot(decoded)%2==0):
            success = 0
            break


    return decoded, success, i

decoded, success, i = F(H1, y1)
print(success, i)
```

For the given word, y, the algorithm took 8 epochs to converge and was successful. The decoded output is printed below.

0100100001100001011100000111000001111001001000000100100001101111011011000110100101100100011000010
1111001011100110010000100100000010001000110110101101001011101000111001001111001001001100100010001
1000010111011001101001011001000010000000111010001010010000110011111111110000110000110000110101100
1101010101111111100000000110011000000110010101001111110101100000010011111010101100110010100111010
0000100111000000011000000110011000001001100110011001110011111111000000110011011001011001011010010
1100110100101100000101000110110111100001111110011111010011001011111110010011001101010100110001111
0001101100011010011001011010101111100101100110111111000101110011111000000000101011011000000001100 1
1111101100110100100111010010110100011001100111100101011001001101010011111011001101010001111111010
1010000001011001000010101001010111110110001111000000010101101100001111110011000000110110111100110
1010110100111111111000011000110110100010101101110010010011100110101101011100100101010101111011101
1100011111111101101100101100111 0

1.4 Original Message

```
def decode(signal):
    signal = signal.reshape(len(signal)//8, 8).astype("int")
    message = ""
    for character in signal[:31]:
        message += chr(int(np.array_str(character)[1:-1].replace(" ", ""), 2))
    return message
decode(decoded)
```

Happy Holidays! Dmitry&David :)

1.5 Empirical Study

```
ps = np.arange(0.125, 0.19, 0.0025)
print(len(ps))
ntrials = 10
results = np.zeros((len(ps), ntrials, 3))
successes = np.zeros(len(ps))
for i, p in enumerate(ps):
    for j in range(ntrials):
        original = np.random.randint(0, 2, (252,))
        corrupted = G.dot(original)
        corrupti = np.random.randint(0, 1000, int(1000*p))
        corrupted[corrupti] += 1
        corrupted = corrupted % 2
        decoded, success, niter = F(H1, corrupted, p, 20)
        print(success, niter, p)
        results[i, j] = [p, success, niter]
        successes[i] += (1+success)
        print(successes[i])
    print(f"{p}: {successes[i]}")
```

We investigated values for noise, p, from 0.01 to 0.2 by running 100 tests for each value and recording the number of successes. For any value above this, the algorithm never converged within 20 iterations. As shown in Figure 1, the algorithm always converges for $p < 0.13$. Success rate then decreases rapidly, until it reaches 0 around $p = 0.18$.

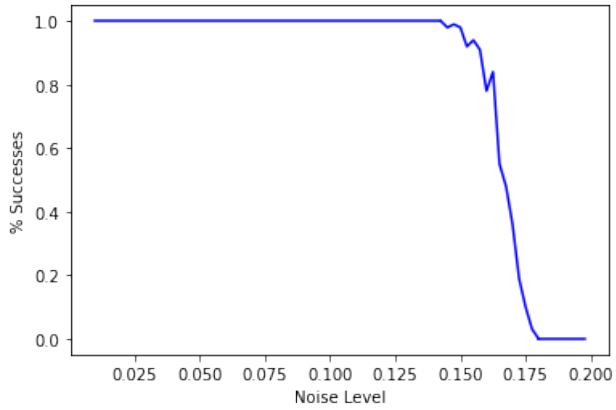
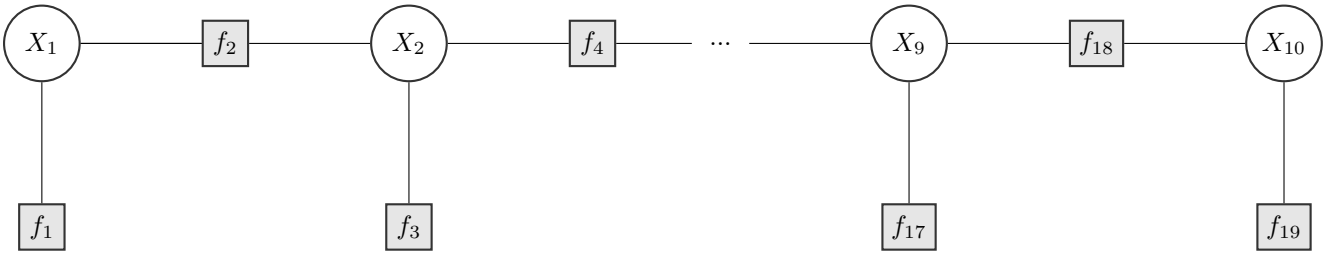


Figure 1: Empirical Study of Success % as a function of noise level.

2 Question 2

2.1 Exact inference

For this first method, we treat each column of the lattice as a single variable with 2^{10} potential states. We can then represent the probability distribution of the Markov Random Field through the following factor graph:



2.1.1 Interpretation of factor graph

Variable nodes The variable X_t ($t \in [1, 10]$) represents all variables from the t^{th} column of the original Markov random field. Since each of the original variables has two potential values, we obtain 2^{10} potential states for X_t.
For example, using -1 and 1 as the two potential values for the original variables, X_t may take the state: [-1, 1, 1, -1, 1, -1, -1, 1, 1, -1].

Factors We have two types of factors:

- Factors associated with a single variable (f with odd subscripts in the graph above):
  These factors reflect the probability distribution of X_t based on whether two neighbouring nodes of the original column are identical or different. They can be represented as vectors of length 2^{10}. Each entry is equal to the product of 9 'sub-factors', which each correspond to one pair of neighbouring values.
  For example, the factor value associated with the state mentioned above would be $e^{3\beta}$ since only 3 pairs of neighbouring values are equal. By contrast, the factor value for a uniform state with 10 identical values would be $e^{9\beta}$.

- Factors between two variables (f with even subscripts in the graph above):
  These factors correspond to the horizontal edges in the original graph. Since each variable can take 2^{10} states, the factors can be represented as matrices of dimension $2^{10} \times 2^{10}$. Each entry of the matrix is equal to the product of 10 'sub-factors', which each correspond to one pair of values that share the same index in two neighbouring columns.
  For example, if X_t is in the state [-1, 1, 1, -1, 1, -1, -1, 1, 1, -1], and X_{t+1} is in the state [-1, **-1**, 1, **1**, 1, -1, -1, 1, **-1**, -1], the associated factor value will be $e^{7\beta}$, since for 7 out of 10 indices, we have the same value in X_t and X_{t+1} (the differences are marked in bold characters).

- All factor matrices with even subscripts ($f_2, ...f_{18}$) are equal, though the individual factors can take different values depending on their neighbours. The same is true for factor vectors with odd subscripts ($f_1, ...f_{19}$).

2.1.2 Message passing

Since we are interested in the marginal probability distribution of X_{10}, we begin the message passing from the left-hand side.

$\mu_{f_1 \to X_1}$ Since f_1 is an extremal factor, its value is set to the factor.

$\mu_{X_1 \to f_2}$ There is only one incoming message to X_1, therefore, we have:

$$\mu_{X_1 \to f_2}(X_1) = \mu_{f_1 \to X_1}(X_1)$$

$\mu_{f_2 \to X_2}$ We multiply the incoming message $\mu_{X_1 \to f_2}$ with f_2 itself, and then marginalise over X_1:

$$\mu_{f_2 \to X_2} = \sum_{X_1} f_2(X_1, X_2) \mu_{X_1 \to f_2}(X_1)$$

$\mu_{X_2 \to f_4}$ We multiply the two incoming messages:

$$\mu_{X_2 \to f_4} = \mu_{f_2 \to X_2} \cdot \mu_{f_3 \to X_2}$$

Remaining messages We can continue the same reasoning up until X_{10} to eventually obtain the marginal distribution of X_{10}:

$$\begin{aligned} p(X_{10}) &= \mu_{f_{18} \to X_{10}} \cdot \mu_{f_{19} \to X_{10}} \\ &= \left[\sum_{X_9} f_{18}(X_9, X_{10}) \mu_{X_9 \to f_{18}}(X_9) \right] \cdot \mu_{f_{19} \to X_{10}} \end{aligned}$$

Once we have the marginal distribution of X_{10}, we can determine the distribution of the first and last element of the state by marginalising over all the other elements.

2.1.3 Computation

The code to compute this result can be seen below.

We begin by creating an array that consists of all potential states of each of the cluster variables X.

```
import numpy as np
n = 10
x_states = np.zeros((2**n, n))
values = [-1,1]
i=0
for a_1 in values:
    for a_2 in values:
        for a_3 in values:
            for a_4 in values:
                for a_5 in values:
                    for a_6 in values:
                        for a_7 in values:
                            for a_8 in values:
                                for a_9 in values:
                                    for a_10 in values:
                                        x_states[i,:] = [a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9,a_10]
                                        i += 1
```

The message passing procedure described above requires the multiplication of rapidly increasing numbers. Since the objective here is not to calculate the normalising constant, we can normalise the variable-to-factor messages at the stage of every variable. This does not alter the result when we marginalise over the relevant variable to compute the next factor-to-variable message.

```
beta_values = [0.01,1,4]

for beta in beta_values:
```

```
    # Computing f_1
    f_1 = np.zeros(len(x_states))
    for i in range(len(x_states)):
        identities = 0
        # determine how many neighbouring elements are identical
        for j in range(x_states.shape[1]-1):
            if x_states[i,j] == x_states[i,j+1]:
                identities += 1
        f_1[i] = np.exp(beta*identities)

    # Computing message from f_2 to X_2
    f_2_to_x_2 = np.zeros(len(x_states))
    for i in range(len(x_states)):
        for j in range(len(x_states)):
            identities = 0
            for k in range(x_states.shape[1]):
                if x_states[i,k] == x_states[j,k]:
                    identities += 1
            f_2_to_x_2[i] += f_1[j]*np.exp(beta*identities)

    # Computing message from X_2 to f_4, multiplying incoming factor messages
    x_2_to_f_4 = np.multiply(f_1,f_2_to_x_2)
    # normalising message
    x_2_to_f_4 = x_2_to_f_4/sum(x_2_to_f_4)

    message_from_previous_x = np.copy(x_2_to_f_4)

    # subsequent messages up to X_10

    for v in range(3,10+1):
        message_from_joint_factor = np.zeros(len(x_states))
        for i in range(len(x_states)):
            for j in range(len(x_states)):
                identities = 0
                for k in range(x_states.shape[1]):
                    if x_states[i,k] == x_states[j,k]:
                        identities += 1
                message_from_joint_factor[i] += message_from_previous_x[j]*np.exp(beta*identities)
        message_from_previous_x = np.multiply(f_1,message_from_joint_factor)
        message_from_previous_x = message_from_previous_x/sum(message_from_previous_x)

    # determine which of the states involve the same value for the first and last elements of X
    indices = np.where(x_states[:,0]==x_states[:,-1])
    prob_same = 0
    # compute the marginal probability that x_{1,10} = x_{10,10}
    for i in indices:
        prob_same += message_from_previous_x[i]

    print(f'For beta = {beta}, the probability that x_{1,10} = x_{10,10} is equal to {sum(prob_same)}.')
```

2.1.4 Result

Using the approach described above, we obtain the following results:

Table 1: Joint probability distribution for $x_{1,10}$ and $x_{10,10}$ (up to 3 decimals)

β	$\mathrm{P}(x_{1,10} = x_{10,10})$	$\mathrm{P}(x_{1,10} \neq x_{10,10})$
4	0.999	0.001
1	0.561	0.439
0.01	0.500	0.500

2.2 Mean Field Approximation

We begin by recalling the joint probability distribution over all variables, as follows:

$$P(x) = Z^{-1} \prod_{i>j} \phi(x_i, x_j) \tag{4}$$

With Z^{-1} as the normalisation factor, and $\phi(x_i, x_j)$ defined as follows

$$\phi(x_i, x_j) = e^{\beta \mathbb{I}(x_i = x_j)} \tag{5}$$

Where $\mathbb{I}(x_i = x_j)$ is the indicator function, i.e. non-zero, for when $x_i = x_j$

2.2.1 Approximate distribution $q(x)$

The mean-field approximation relies on the assumption that the joint distribution of our variables can be factorised into the product of individual distributions. That is to say that the approximate distribution ($q(x)$) factorises as follows:

$$q(x) = \prod_i q(x_i)$$

The objective is then to find the $q(x)$ which maximises the Evidence Lower Bound (ELBO), which in the absence of observed variables is equivalent to the negative Kullback-Leibler divergence between the approximate distribution and the true distribution. The ELBO is thus defined as the following:

$$ELBO = \mathbb{E}_{q(x)} \left[\log(p(x)\right] - \mathbb{E}_{q(x)} \left[\log(q(x)\right] \tag{6}$$

And then using our specification of the joint probability distribution

$$= -\mathbb{E}_{q(x)} \left[\log(Z)\right] + \mathbb{E}_{q(x)} \left[\log(\prod_{i>j} e^{\beta \mathbb{I}(x_i = x_j)})\right] - \mathbb{E}_{q(x)} \left[\log(q(x))\right] \tag{7}$$

$$= -\log(Z) + \mathbb{E}_{q(x)} \left[\sum_{i>j} \beta \mathbb{I}(x_i = x_j)\right] - \mathbb{E}_{q(x)} \left[\log(q(x))\right] \tag{8}$$

Using the linearity of expectation

$$= -\log(Z) + \sum_{i>j} \mathbb{E}_{q(x)} \left[\beta \mathbb{I}(x_i = x_j)\right] - \mathbb{E}_{q(x)} \left[\log(q(x))\right] \tag{9}$$

We can then minimise equation 9 with respect to q(x) using coordinate ascent. To do so, we iterate over the variables that q is dependent on, and minimise with respect to each of these variables while keeping the rest fixed. Note that coordinate ascent is guaranteed to find a local (but not necessarily global) maxima.

$$q_i^{t+1} = \underset{q_i}{\operatorname{argmax}} \left[\sum_{i>j} \mathbb{E}_{q(x)} \left[\beta \mathbb{I}(x_i = x_j) \right] - \mathbb{E}_{q(x)} \left[\log(q(x)) \right] \right] \tag{10}$$

We then isolate only the terms that are dependent on $q_i(x_i)$. $\text{ne}(x_i)$ is the set of neighbours of x_i.

$$= \underset{q_i}{\operatorname{argmax}} \left[\sum_{j \in \text{ne}(x_i)} \mathbb{E}_{q(x)} \left[\beta \mathbb{I}(x_i = x_j) \right] - \mathbb{E}_{q_i(x_i)} \left[\log(q_i(x_i)) \right] \right] \tag{11}$$

Since $q_i(x_i)$ is a discrete binary distribution, we can define it in terms of a single parameter μ_i

$$= \underset{\mu_i}{\operatorname{argmax}} \left[\sum_{j \in \text{ne}(x_i)} \left[\beta \mu_i \mu_j + \beta (1 - \mu_i)(1 - \mu_j) \right] - \mathbb{E}_{q_i(x_i)} \left[\log(q_i(x_i)) \right] \right] \tag{12}$$

Where the expectation terms with x_i not equal to x_j have disappeared due to the indicator function

$$\tag{13}$$

We can then proceed to find the maxima of this function by differentiating with respect to μ_i and setting to 0. This gives us the following.

$$\log \mu_i - \log (1 - \mu_i) = \left[\sum_{j \in \text{ne}(x_i)} \beta (2\mu_j - 1) \right] \tag{14}$$

$$\log \frac{\mu_i}{(1 - \mu_i)} = \left[\sum_{j \in \text{ne}(x_i)} \beta (2\mu_j - 1) \right] \tag{15}$$

Rearranging for μ_i gives us

$$\mu_i = \frac{\exp \left[\sum_{j \in \text{ne}(x_i)} \beta (2\mu_j - 1) \right]}{1 + \exp \left[\sum_{j \in \text{ne}(x_i)} \beta (2\mu_j - 1) \right]} \tag{16}$$

The joint probability of $P(x_{1,10}, x_{10,10})$ can then be approximated as the product of our approximate probabilities $q(x_{1,10})q(x_{10,10})$

The obvious interpretation to this result is that when our β parameter goes to 0 (i.e. temperature goes to infinity), each particle will have equal probability of being in either spin state ($\mu_i = 0.5$), and thus every particle will be effectively independent of each other. Alternatively, if our β goes to infinity, each particle will be in the same state with a probability close to 1.

2.2.2 Computation

The code to compute this result can be seen below.

The function get_neighbours returns the indices of the connected neighbours of a particular node.

```
1  def get_neighbours(x, y, shape=(10,10)):
2      edges_list = []
3      if x != 0:
4          edges_list.append((x-1, y))
5      if x != (shape[0]-1):
6          edges_list.append((x+1, y))
```

```
    if y != 0:
        edges_list.append((x, y-1))
    if y != (shape[1]-1):
        edges_list.append((x, y+1))

    return tuple(zip(*edges_list))
```

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

max_epochs = 100
shape = (10,10)

for beta in [4,1,0.01]:
    q_array = np.zeros(shape)
    for epoch in range(max_epochs):
        for x in range(shape[0]):
            for y in range(shape[1]):
                neighbours = get_neighbours(x,y, shape = shape)
                log_odds = beta*np.sum(2*q_array[neighbours]-1)
                q_array[x,y] = np.exp(log_odds)/(1+np.exp(log_odds))

    plt.figure()
    sns.heatmap(q_array)
    plt.savefig(f"q Array - Beta: {beta}.png")
    joint_prob = q_array[0,9]*q_array[9,9] + (1-q_array[0,9])*(1-q_array[9,9])
    print(q_array)
    print(f"Beta: {beta}")
    print(f"Probability of x_top = x_bottom: {joint_prob}")
    print(f"Probability of x_top != x_bottom: {1-joint_prob}")
```

2.2.3 Result

Using the approach described above, we obtain the following results:

Note: The results below represent one local minima that the algorithm may converge to. Depending on the choice of initialization for our approximate distribution q, we may also converge to other local minima.

Table 2: Joint probability distribution for $x_{1,10}$ and $x_{10,10}$

β	$\mathrm{P}(x_{1,10} = x_{10,10})$	$\mathrm{P}(x_{1,10} \neq x_{10,10})$
4	0.999	0.001
1	0.739	0.261
0.01	0.500	0.500

We observe good correspondence between exact inference and the mean field approximation for $\beta = 4$ and $\beta = 0.01$, but not for $\beta = 1$. This is likely due to the strength of our assumption, namely that the joint distribution over our variables factorises into a product of independent distributions.

Heatmaps of our final predicted array of probabilities can be seen below. Each square represents a node i in our factor graph, with the colour corresponding to it's approximate probability μ_i.

Note: The colour scales of our 3 heatmaps are adjusted to best show any differences within each array and thus are not directly comparable to each other. Importantly, while the corner distributions of $\beta = 4$ appear to show a large difference in probability with the non-corner distributions, this difference is in fact quite small ($\tilde{0}.0001$).

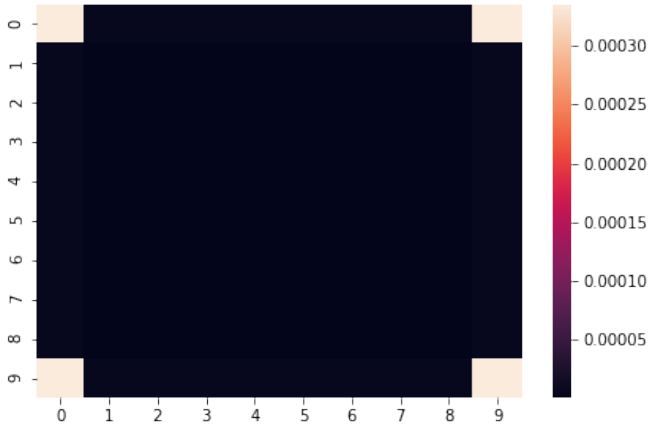


Figure 2: Array of μ values for $\beta = 4$

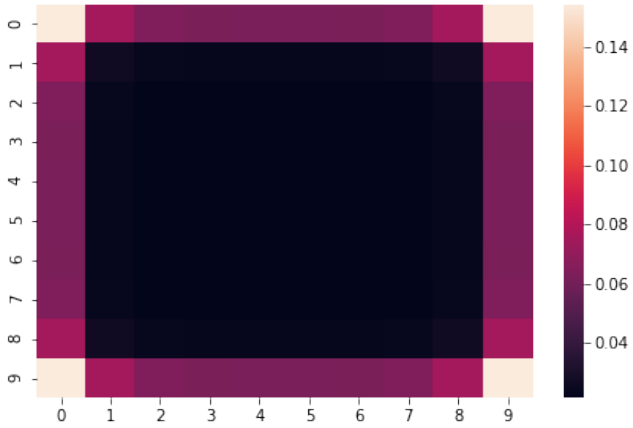


Figure 3: Array of μ values for $\beta = 1$

2.3 Gibbs Sampling

Our final method will use Gibbs sampling to approximate the distribution of interest.

The joint distribution of our 10×10 lattice is given by

$$P_x(x) = \frac{1}{Z} \prod_{i>j} \phi(x_i, x_j),$$

with node potentials $\phi(x_i, x_j) = e^{\beta \mathbb{I}[x_i = x_j]}$, where i and j are neighbours. Computing this distribution would require Z, which is intractable.

Sampling methods allow us to approximate a distribution $p(x)$ by drawing samples $X = \{x^1, ..., x^L\}$ from it, and then computing quantities of interest, such as expectations, using these.

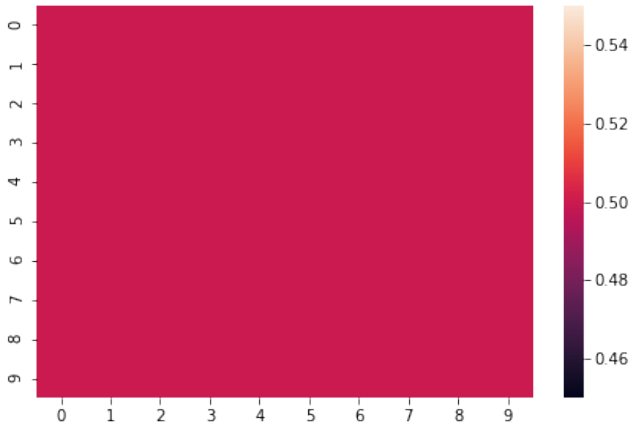


Figure 4: Array of μ values for $\beta = 0.01$

Two of the most basic sampling methods are importance and rejection sampling, but these are particularly difficult and costly as the complexity of the problem increases, such as for high-dimensional distributions, motivating the use of Markov chain Monte Carlo (MCMC) methods.

Suppose we wish to sample from a distribution $p(x)$. Beginning with an initial sample x^1, the idea behind MCMC methods is to recursively generate the samples $x^1, ..., x^L$ using a conditional transition distribution $T(x \to x') = p(x'|x)$ where each consecutive sample depends on the one before it. By constructing a Markov chain where $p(x)$ is the stationary distribution, the samples will eventually converge to a state in which it is as if they were drawn from $p(x)$:

$$p_\infty(x') = \sum_x p_\infty(x)p(x'|x).$$

We therefore need to find a transition distribution which has $p(x)$ as its long-run, stationary distribution. A sufficient condition for the existence of the stationary distribution $p(x)$ is *detailed balance*, which is the property that for all x and x', the probability of being in state x and transitioning to x' is equal to the probability of being in state x' and transitioning to x, or

$$p(x')T(x' \to x) = p(x)T(x \to x').$$

It is also important that the Markov chain converges to $p(x)$ uniquely, regardless of where it started (the initial sample), for which a sufficient condition is *irreducibility*: that it is possible to reach any state from any other state, and that it is aperiodic, i.e. it does not return to any particular state at fixed intervals. In other words, for all x and x', and some k steps,

$$T^k(x \to x') > 0.$$

As discussed, for our problem, it is difficult to compute the joint distribution exactly. However, it is simple to compute a conditional distribution for each node which conditions only on its neighbours, making the problem well-suited to Gibbs sampling, a particular MCMC method. If we have a multivariate distribution $p(x)$ and some initial joint state x^1, we condition for a single variable $p(x_i)$ using

$$p(x) = p(x_i|x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)p(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n),$$

Given the values of $x_1^1, ..., x_{i-1}^1, x_{i+1}^1, ..., x_n^1$ from the initial state, we can therefore draw a new sample x_i^2 using

$$p(x_i|x_1^1, ..., x_{i-1}^1, x_{i+1}^1, ..., x_n^1) \equiv p(x_i|x_{\setminus i}),$$

and so on to sample a large number of times. In fact, it is only necessary to condition on the Markov blanket of x_i (for our problem, the neighbours of a node), as

$$p(x_i|x_{\setminus i}) = \frac{1}{Z} p(x_i|\text{pa}(x_i)) \prod_{j \in \text{ch}(i)} p(x_j|\text{pa}(x_j)).$$

We can construct a Markov chain with a transition distribution $x^{l+1} \sim q(x^{l+1}|x^l)$, and repeatedly draw samples by randomly choosing a single variable x_i to update at a time, and keeping all the others the same as in each previous sample, so that the transition distribution is

$$q(x^{l+1}|x^l, i) = p(x_i^{l+1}|x_{\setminus i}^l) \prod_{j \neq i} \delta(x_j^{l+1}, x_j^l),$$

where $q(i) > 0$ and $\sum_i q(i) = 1$.

If the equilibrium distribution of $q(x^{l+1}|x^l, i)$ is $p(x)$ and it satisfies detailed balance and irreducibility, then it will converge to $p(x)$ in the long-run (similar for discrete case):

$$\begin{aligned}
\sum_x q(x'|x)p(x) &= \sum_i q(i) \int_x q(x'|x, i)p(x) \\
&= \sum_i q(i) \int_x \prod_{j \neq i} \delta(x'_j, x_j) p(x'_i|x_{\setminus i} p(x_i|x_{\setminus i}) \\
&= \sum_i q(i) \int_{x_i} p(x'_i|x'_{\setminus i}) p(x_i, x'_{\setminus i}) \\
&= \sum_i q(i) p(x'_i|x'_{\setminus i}) p(x'_{\setminus i}) \\
&= \sum_i q(i) p(x') \\
&= p(x').
\end{aligned}$$

As the chain may start in an initial state that is not representative of the desired distribution which is reached when the number of iterations $n \to \infty$, it is common to use *burn-in* to discard initial samples. Additionally, our samples are not independent, since each successive sample only updates a single variable, so we can *subsample* by taking every kth sample that we generate and discarding the others, so that they are less correlated.

For our problem, we can make use of the lattice structure, and the dependence of each node on its neighbours, to implement Gibbs sampling. We can iterate through each node (in order or at random), and compute a simple, tractable conditional on its neighbouring nodes only, in order to generate consecutive samples, using a structured model:

$$p(x_i|x_{\setminus i}) \propto \prod_{j \in \text{ne}(i)} \phi(x_i, x_j) = \prod_{j \in \text{ne}(i)} e^{\beta \mathbb{I}[x_i = x_j]} \tag{17}$$

For our initial sample x^1, we randomly generate a 10×10 lattice with values in $\{-1, 1\}$. To determine the subsequent samples, we then randomly iterate through each of the nodes in the lattice, over thousands of runs, using Equation 17 to compute the transition probability.

2.3.1 Computation

We choose to sample 25,000 times, with a burn-in period of 5,000 iterations, and take every 10^{th} sample after this point, discarding all others. We repeat the process for $\beta = 0.01, 1, 4$.

```python
import numpy as np

def generate_sample(n):
  return np.random.choice([-1, 1], (n, n))

def get_neighbours(i, j, shape=(10, 10)):
  neighbours = [[i+1, j], [i-1, j], [i, j+1], [i, j-1]]
  valid_neighbours = list(filter(lambda pair : all([0 <= index <= shape[0]-1 for index in pair]),
  ↪  neighbours))
  return valid_neighbours

def conditional(beta, sample, i, j):
  neighbours = get_neighbours(i, j, sample.shape)
  energy = sum([sample[i, j] == sample[k, l] for (k, l) in neighbours])
  return np.exp(- beta * energy)

def gibbs_sampler(n, beta, runs=5000, burn_in=1000, nth=10):

  # Initialise
  sample = generate_sample(n)
  samples = [np.copy(sample)]

  for _ in range(runs):
    for _ in range(sample.size):
      # Get a random index
      (i, j) = np.unravel_index(np.random.randint(sample.size), sample.shape)

      p = conditional(beta, sample, i, j)
      # Decide whether to flip index
      if np.random.rand() < p:
        sample[i, j] *= -1

    samples.append(np.copy(sample))

  # Discard first burn_in samples, and take every nth from then on
  return samples[burn_in::nth]


for beta in [0.01, 1, 4]:

  # Run Gibbs sampler on Ising model
  samples = gibbs_sampler(n=10, beta=beta, runs=25000, burn_in=5000, nth=10)

  num_samples = len(samples)

  # Compute joint probability
  count_00, count_01, count_10, count_11 = 0, 0, 0, 0

  for sample in samples:
    if sample[0, 9] == -1 and sample[9, 9] == -1:
      count_00 += 1
    elif sample[0, 9] == -1 and sample[9, 9] == 1:
      count_01 += 1
    elif sample[0, 9] == 1 and sample[9, 9] == -1:
      count_10 += 1
    elif sample[0, 9] == 1 and sample[9, 9] == 1:
```

```
        count_11 += 1

    print(f"Beta: {beta}")
    print("* Probability of x_{1,10} = -1, x_{10, 10} = -1:", count_00/num_samples)
    print("* Probability of x_{1,10} = -1, x_{10, 10} = 1:", count_01/num_samples)
    print("* Probability of x_{1,10} = 1, x_{10, 10} = -1:", count_10/num_samples)
    print("* Probability of x_{1,10} = 1, x_{10, 10} = 1:", count_11/num_samples)
```

2.3.2 Results

Using the approach described above, we obtain the following results:

Table 3: Joint probability distribution for $\beta = 0.01$

	$P(x_{1,10} = -1)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = -1)$	0.257	0.253
$P(x_{10,10} = 1)$	0.245	0.245

Table 4: Joint probability distribution for $\beta = 1$

	$P(x_{1,10} = -1)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = -1)$	0.234	0.230
$P(x_{10,10} = 1)$	0.214	0.322

Table 5: Joint probability distribution for $\beta = 4$

	$P(x_{1,10} = -1)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = -1)$	0	0
$P(x_{10,10} = 1)$	0.001	0.999

Note: We report the results for a single run, but we find that re-running the Gibbs sampler with different initializations of the lattice, we may converge to different distributions.