STA414_2024_HW1_Answers

February 20, 2024

1 Q1 - Decision Theory

One successful use of probabilistic models is for building spam filters, which take in an email and take different actions depending on the likelihood that it's spam.

Imagine you are running an email service. You have a well-calibrated spam classifier that tells you the probability that a particular email is spam: $p(spam|email)$. You have five options for what to do with each email: You can list it as important email, show it to the user, put it in the other folder, put it in the spam folder, or delete it entirely.

Depending on whether or not the email really is spam, the user will suffer a different amount of wasted time for the different actions we can take, L(action, spam):

Action	Spam	Not spam
Important	30	0
Show	10	2
Other	1	5
Spam	3	50
Delete	0	100

1.1 Q1.1

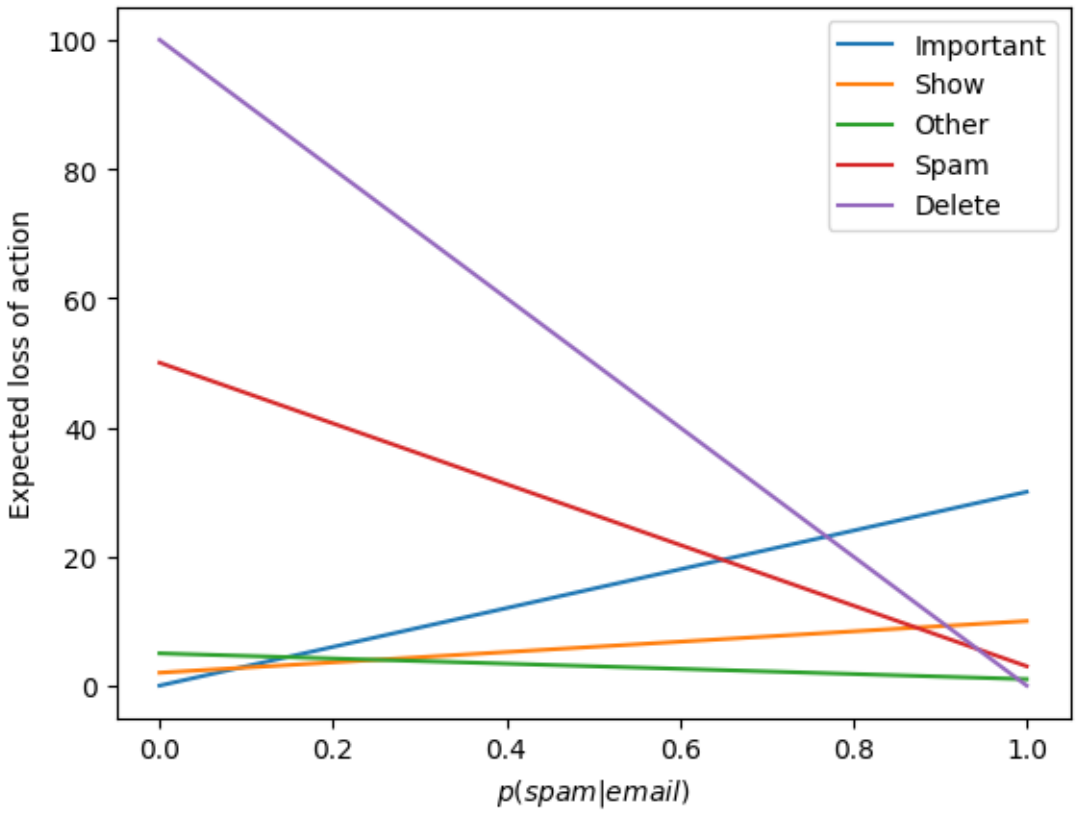
[3pts] Plot the expected wasted user time for each of the three possible actions, as a function of the probability of spam: $p(spam|email)$.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
losses = [[30, 0],[10, 2], [1, 5], [3, 50],[0, 100]]
actions_names = ['Important','Show', 'Other', 'Spam', 'Delete']
num_actions = len(losses)
def expected_loss_of_action(prob_spam, action):
    #TODO: Return expected loss over a Bernoulli random variable
    # with mean prob_spam.
    # Losses are given by the table above.
    # Delete below
    EL = losses[action][0] * prob_spam + losses[action][1] * (1 - prob_spam)
    return EL
```

```
    # Delete above

prob_range = np.linspace(0., 1., num=600)

# Make plot
for action in range(num_actions):
    plt.plot(prob_range, expected_loss_of_action(prob_range, action),␣
 ↪label=actions_names[action])

plt.xlabel('$p(spam|email)$')
plt.ylabel('Expected loss of action')
plt.legend()
```

[ ]: <matplotlib.legend.Legend at 0x7cd945b15d50>



1.2 Q1.2

[2pts] Write a function that computes the optimal action given the probability of spam.

[ ]:
```
def optimal_action(prob_spam):
    #TODO: return best action given the probability of spam.
```

```
    #Hint: np.argmin might be helpful.
    ELs = []
    for action in range(num_actions):
        ELs.append(expected_loss_of_action(prob_spam, action))
    return(np.argmin(ELs))
```

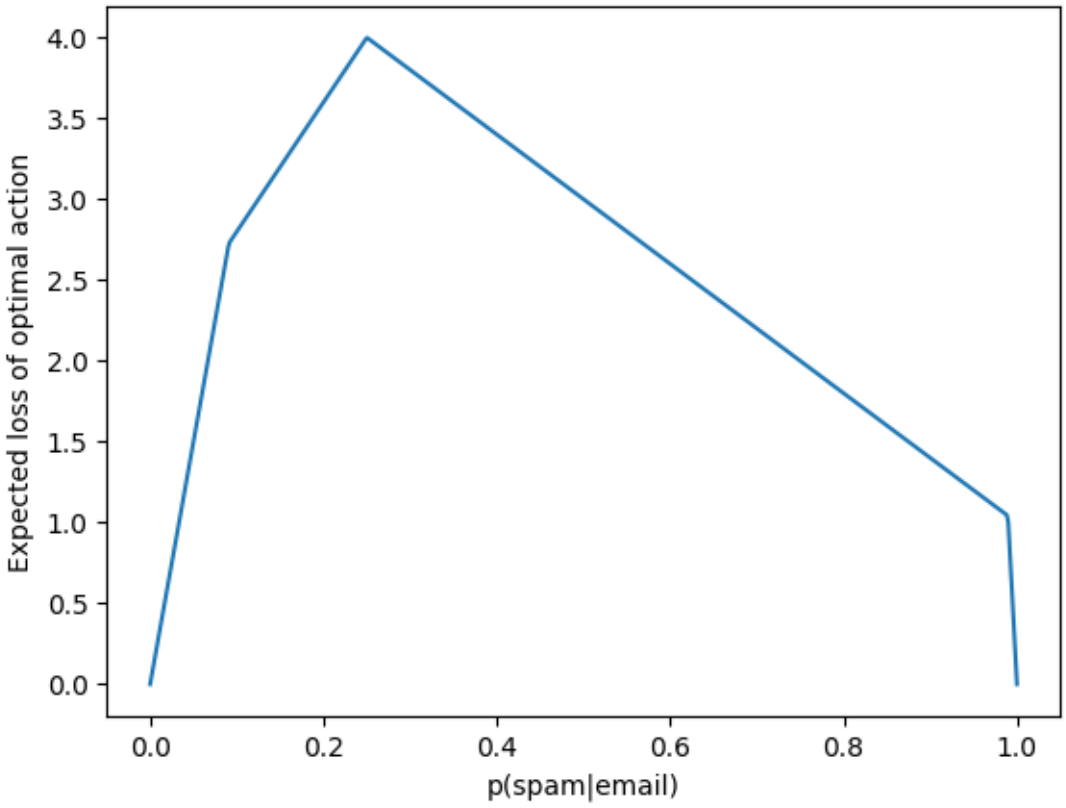
1.3 Q1.3

[4pts] Plot the expected loss of the optimal action as a function of the probability of spam.

Color the line according to the optimal action for that probability of spam.

[ ]:
```
prob_range = np.linspace(0., 1., num=600)
optimal_losses = []
optimal_actions = []
for p in prob_range:
    # TODO: Compute the optimal action and its expected loss for
    # probability of spam given by p.
    optimal_actions.append(optimal_action(p))
    optimal_losses.append(expected_loss_of_action(p, optimal_actions[-1]))
    # print(optimal_actions[-1], p)

plt.xlabel('p(spam|email)')
plt.ylabel('Expected loss of optimal action')
plt.plot(prob_range, optimal_losses)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7cd94413e4a0>]
```



1.4 Q1.4

[4pts] For exactly which range of the probabilities of an email being spam should we set it as important?

And based on user feedback, they what less email to be marked as important. How to change L(action=important, spam=True) so that only if $p(spam|email) < 0.01$, email would be marked as important?

Find the exact answer by hand using algebra.

[Type up your derivation here]

Your answer:

$30p < 10p + 2(1-p) \quad \Rightarrow \quad p < 1/11$

$30p < 3p + 50(1-p) \quad \Rightarrow \quad p < 50/77$

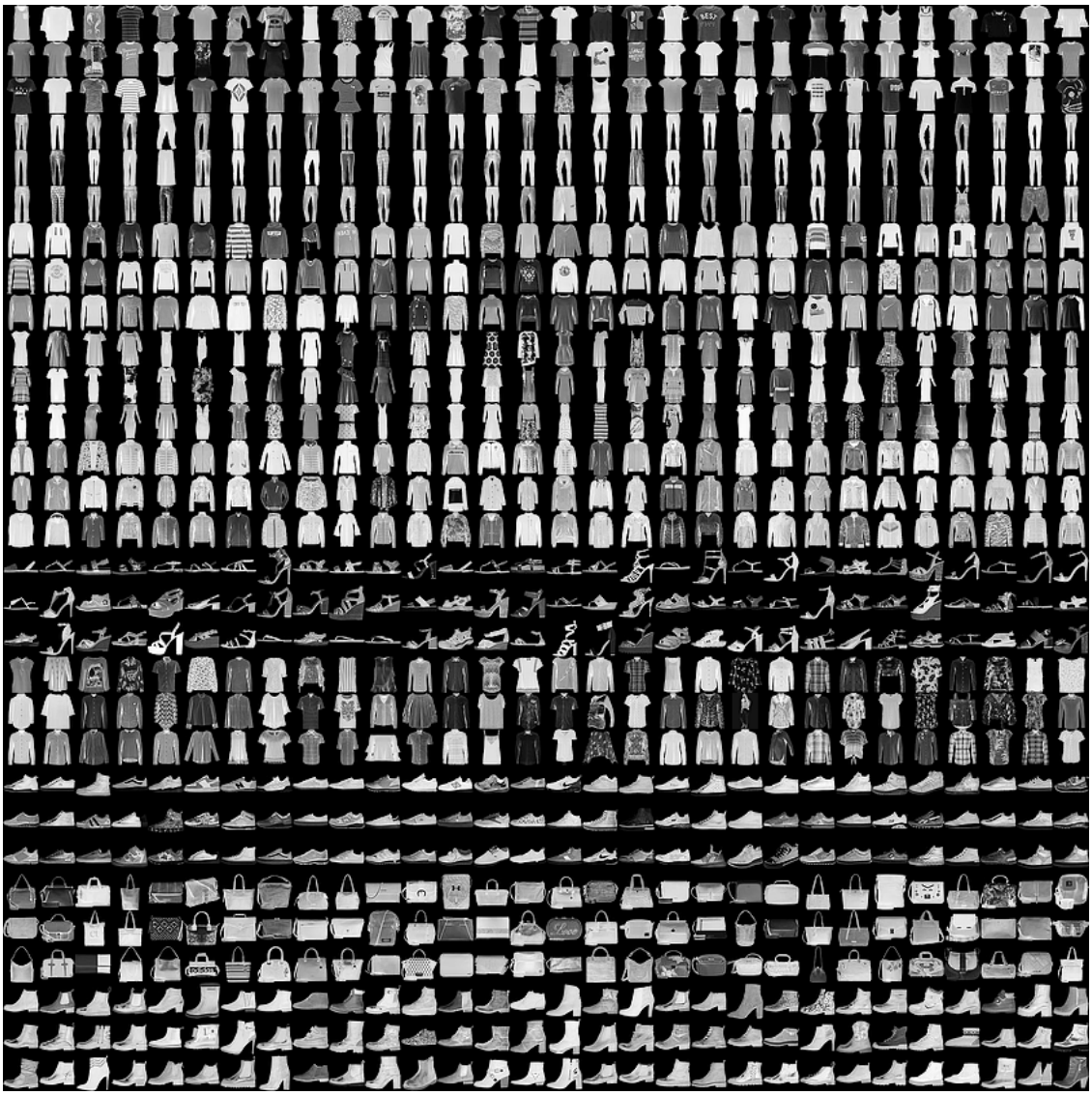
$30p < p + 5(1-p) \quad \Rightarrow \quad p < 5/34$

$30p < 100(1-p) \quad \Rightarrow \quad p < 10/13$

$p < 1/11$

$ L_I p < 10p + 2(1-p) p < 2/(L_I-8) = 1/100 L_I=208$

2 Q2 - Naïve Bayes, A Generative Model



In this question, we'll fit a Bernoulli Naïve Bayes model to the fashion MNIST dataset, and use this model for making predictions and generating new images from the same distribution. Fasion MNIST is a dataset of 28x28 images of items of clothing.

We represent each image by a vector $x^{(i)} \in \{0,1\}^D$, where 0 and 1 represent white and black pixels respectively, and $D = 784$. Each class label $c^{(i)}$ is a different item of clothing, which in the code is represented by a K=10-dimensional one-hot vector.

The Bernoulli Naïve Bayes model parameterized by θ and π defines the following joint probability of x and c,

$$p(x, c|\theta, \pi) = p(c|\pi)p(x|c, \theta) = p(c|\pi)\prod_{j=1}^{D} p(x_j|c, \theta),$$

where $x_j|c, \theta \sim \text{Bernoulli}(\theta_{jc})$, i.e. $p(x_j|c, \theta) = \theta_{jc}^{x_j}(1-\theta_{jc})^{1-x_j}$, and $c|\pi$ follows a simple categorical distribution, i.e. $p(c|\pi) = \pi_c$.

We begin by learning the parameters θ and π.

First The following code will download and prepare the training and test sets.

[ ]:
```
import numpy as np
import os
import gzip
import struct
import array
import matplotlib.pyplot as plt
import matplotlib.image
from urllib.request import urlretrieve

def download(url, filename):
    if not os.path.exists('data'):
        os.makedirs('data')
    out_file = os.path.join('data', filename)
    if not os.path.isfile(out_file):
        urlretrieve(url, out_file)


def fashion_mnist():
    base_url = 'http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/'

    def parse_labels(filename):
        with gzip.open(filename, 'rb') as fh:
            magic, num_data = struct.unpack(">II", fh.read(8))
            return np.array(array.array("B", fh.read()), dtype=np.uint8)

    def parse_images(filename):
        with gzip.open(filename, 'rb') as fh:
            magic, num_data, rows, cols = struct.unpack(">IIII", fh.read(16))
            return np.array(array.array("B", fh.read()), dtype=np.uint8).
 ↪reshape(num_data, rows, cols)

    for filename in ['train-images-idx3-ubyte.gz',
                     'train-labels-idx1-ubyte.gz',
                     't10k-images-idx3-ubyte.gz',
                     't10k-labels-idx1-ubyte.gz']:
        download(base_url + filename, filename)

    train_images = parse_images('data/train-images-idx3-ubyte.gz')
    train_labels = parse_labels('data/train-labels-idx1-ubyte.gz')
    test_images = parse_images('data/t10k-images-idx3-ubyte.gz')
    test_labels = parse_labels('data/t10k-labels-idx1-ubyte.gz')

    # Remove the data point that cause log(0)
    remove = (14926, 20348, 36487, 45128, 50945, 51163, 55023)
```

```
    train_images = np.delete(train_images,remove, axis=0)
    train_labels = np.delete(train_labels, remove, axis=0)
    return train_images, train_labels, test_images[:1000], test_labels[:1000]


def load_fashion_mnist():
    partial_flatten = lambda x: np.reshape(x, (x.shape[0], np.prod(x.shape[1:
 ↪])))
    one_hot = lambda x, k: np.array(x[:, None] == np.arange(k)[None, :],␣
 ↪dtype=int)
    train_images, train_labels, test_images, test_labels = fashion_mnist()
    train_images = (partial_flatten(train_images) / 255.0).astype(float)
    test_images = (partial_flatten(test_images) / 255.0).astype(float)
    train_images_binarized = (train_images > 0.5).astype(float)
    test_images_binarized = (test_images > 0.5).astype(float)
    train_labels = one_hot(train_labels, 10)
    test_labels = one_hot(test_labels, 10)
    N_data = train_images.shape[0]

    return N_data, train_images, train_images_binarized, train_labels,␣
 ↪test_images, test_images_binarized, test_labels
```

2.1 Q2.1

[4pts] Derive the expression for the Maximum Likelihood Estimator (MLE) of θ and π.

[Type up your derivation here]

Your answer:

[Type up your derivation here]

Your answer:

We begin by finding the log-likelihood of the data given θ and π. Let $\mathcal{D} = \{(x^{(1)}, c^{(1)}), ..., (x^{(n)}, c^{(n)})\}$ denote our training set. Then

$$\begin{aligned}
\ell(\theta, \pi; \mathcal{D}) &= \log \prod_{i=1}^{n} p(x^{(i)}, c^{(i)} | \theta, \pi) \\
&= \sum_{i=1}^{n} \log p(x^{(i)}, c^{(i)} | \theta, \pi) \\
&= \sum_{i=1}^{n} \log p(c^{(i)} | \pi) + \sum_{i=1}^{n} \sum_{j=1}^{784} \log p(x_j^{(i)} | c^{(i)}, \theta) \\
&= \sum_{i=1}^{n} \log \pi_{c^{(i)}} + \sum_{i=1}^{n} \sum_{j=1}^{784} x_j^{(i)} \log \theta_{jc^{(i)}} + (1 - x_j^{(i)}) \log(1 - \theta_{jc^{(i)}})
\end{aligned}$$

Notice that in order to find the pair $(\hat{\theta}_{\text{MLE}}, \hat{\pi}_{\text{MLE}})$ that maximizes the log-likelihood, we can maximize each term separately. Moreover, we have the constraint $\sum_c \pi_c = 1$ for π. Using the method

of Lagrange multipliers for maximizing the first term, we have for each $0 \leq t \leq 9$,

$$\frac{\partial}{\partial c_t}\left\{\sum_{i=1}^{n} \log \pi_{c^{(i)}} + \lambda(\sum_{j=0}^{9} \pi_j - 1)\right\} = 0 \implies \hat{\pi}_t = \frac{-\sum_{i=1}^{n} 1(c^{(i)} = t)}{\lambda}$$

where $1(c^{(i)} = t)$ is the indicator function. Combined with the condition $\sum_t \pi_t = 1$, we have $\lambda = -n$, therefore

$$\hat{\pi}_t = \frac{\sum_{i=1}^{n} 1(c^{(i)} = t)}{n}.$$

In other words, $\hat{\pi}_t$ is simply the percentage of the images labeled t. Similarly, maximizing the second term results in

$$\begin{aligned}
&\frac{\partial}{\partial \theta_{jt}}\left\{\sum_{i=1}^{n}\sum_{j=1}^{784} x_j^{(i)} \log \theta_{jc^{(i)}} + (1 - x_j^{(i)}) \log(1 - \theta_{jc^{(i)}})\right\} = 0 \\
&\implies \sum_{i=1}^{n} \frac{x_j^{(i)} 1(c^{(i)} = t)}{\hat{\theta}_{jt}} - \sum_{i=1}^{n} \frac{(1 - x_j^{(i)}) 1(c^{(i)} = t)}{1 - \hat{\theta}_{jt}} = 0 \\
&\implies \hat{\theta}_{jt} = \frac{\sum_{i=1}^{n} x_j^{(i)} 1(c^{(i)} = t)}{\sum_{i=1}^{n} 1(c^{(i)} = t)}.
\end{aligned}$$

Again, $\hat{\theta}_{jt}$ is just the mean of pixel j in class t.

2.2 Q2.2

[5pts] Using the MLE for this data, many entries of θ will be estimated to be 0, which seems extreme. So we look for another estimation method.

Assume the prior distribution of θ is such that the entries are i.i.d. and drawn from Beta(α, α). Derive the Maximum A Posteriori (MAP) estimator for θ (it has a simple final form). You can return the MLE for π in your implementation. From now on, we will work with this estimator.

[Type up your derivation here]

Your answer:

[ ]:
```
def train_map_estimator(train_images, train_labels, alpha):
    """ Inputs:
        train_images (N_samples x N_features)
        train_labels (N_samples x N_classes)
        alpha (float)
        Returns the MAP estimator theta_est (N_features x N_classes) and the MLE
        estimator pi_est (N_classes)"""

    # YOU NEED TO WRITE THIS PART
    pi_est = np.mean(train_labels, axis=0)
    num_features = train_images.shape[1]
    theta_est = (train_images.T.dot(train_labels) + (alpha - 1.)) / \
        np.tile(np.sum(train_labels, axis=0) + (2 * alpha - 2.), (num_features,␣
↪1))
```

```
    return theta_est, pi_est
```

2.3 Q2.3

a) [4pts] Derive an expression for the class log-likelihood $\log p(c|x, \theta, \pi)$ for a single image. Then, complete the implementation of the following functions. Recall that our prediction rule is to choose the class that maximizes the above log-likelihood, and accuracy is defined as the fraction of samples that are correctly predicted.

Report the average log-likelihood $\frac{1}{N}\sum_{i=1}^{N} \log p(c^{(i)}|x^{(i)}, \hat{\theta}, \hat{\pi})$ (where N is the number of samples) on the training test, as well the training and test errors. Use a value of $\alpha = 2$.

[Type up your derivation here]

Your answer:

[ ]:
```
def log_likelihood(images, theta, pi):
    """ Inputs: images (N_samples x N_features), theta, pi
        Returns the matrix 'log_like' of loglikehoods over the input images␣
↪where
        log_like[i,c] = log p (c |x^(i), theta, pi) using the estimators theta␣
↪and pi.
        log_like is a matrix of (N_samples x N_classes)
    Note that log likelihood is not only for c^(i), it is for all possible c's.
↪"""

    # YOU NEED TO WRITE THIS PART
    N = images.shape[0]
    log_joint = np.array([np.sum(np.tile(np.log(theta[:, c]), (N, 1)) * images␣
↪+ np.tile(np.log(1 - theta[:, c]), (N, 1)) * (1 - images), axis=1) + np.
↪log(pi[c]) for c in range(len(pi))])
    log_like = log_joint - np.tile(np.log(np.sum(np.exp(log_joint), axis=0)),␣
↪(len(pi), 1))

    return log_like.T


def accuracy(log_like, labels):
    """ Inputs: matrix of log likelihoods and 1-of-K labels (N_samples x␣
↪N_classes)
    Returns the accuracy based on predictions from log likelihood values"""

    # YOU NEED TO WRITE THIS PART
    predictions = np.argmax(log_like, axis=1)
    return float(np.sum(predictions == np.argmax(labels, axis=1))) /␣
↪len(predictions)
```

```
N_data, train_images, train_images_binarized, train_labels, test_images,␣
 ↪test_images_binarized, test_labels = load_fashion_mnist()

theta_est, pi_est = train_map_estimator(train_images_binarized, train_labels,␣
 ↪alpha=2.)

loglike_train = log_likelihood(train_images_binarized, theta_est, pi_est)
avg_loglike = np.sum(loglike_train * train_labels) / N_data
train_accuracy = accuracy(loglike_train, train_labels)
loglike_test = log_likelihood(test_images_binarized, theta_est, pi_est)
test_accuracy = accuracy(loglike_test, test_labels)

print(f"Average log-likelihood for MAP with alpha = 2 is {avg_loglike:.3f}")
print(f"Training accuracy for MAP with alpha = 2 is {train_accuracy:.3f}")
print(f"Test accuracy for MAP with alpha = 2 is {test_accuracy:.3f}")
```

```
Average log-likelihood for MAP with alpha = 2 is -34.231
Training accuracy for MAP with alpha = 2 is 0.651
Test accuracy for MAP with alpha = 2 is 0.638
```

b) [2pts] Now compute the MAP estimators using $\alpha = 1$. Then rerun the code for computing the log-likelihoods and accuracy.

What do you observe? - comment on whether it was important or not to use the MAP (with $\alpha > 1$). Based on your previous derivation, what does $\alpha = 1$ correspond to?

(Note: You do not need to report the average log-likelihoods or accuracy in this part.)

[ ]:
```
theta_alpha1_est, pi_est = train_map_estimator(train_images_binarized,␣
 ↪train_labels, alpha=1.)

loglike_train = log_likelihood(train_images_binarized, theta_alpha1_est, pi_est)
avg_loglike = np.sum(loglike_train * train_labels) / N_data
train_accuracy = accuracy(loglike_train, train_labels)
loglike_test = log_likelihood(test_images_binarized, theta_alpha1_est, pi_est)
test_accuracy = accuracy(loglike_test, test_labels)

print(f"Average log-likelihood for MAP with alpha = 1 is {avg_loglike:.3f}")
print(f"Training accuracy for MAP with alpha = 1 is {train_accuracy:.3f}")
print(f"Test accuracy for MAP with alpha = 1 is {test_accuracy:.3f}")
```

```
<ipython-input-7-1444158f2320>:10: RuntimeWarning: divide by zero encountered in
log
  log_joint = np.array([np.sum(np.tile(np.log(theta[:, c]), (N, 1)) * images +
np.tile(np.log(1 - theta[:, c]), (N, 1)) * (1 - images), axis=1) + np.log(pi[c])
for c in range(len(pi))])
<ipython-input-7-1444158f2320>:10: RuntimeWarning: invalid value encountered in
multiply
  log_joint = np.array([np.sum(np.tile(np.log(theta[:, c]), (N, 1)) * images +
```

```
np.tile(np.log(1 - theta[:, c]), (N, 1)) * (1 - images), axis=1) + np.log(pi[c])
for c in range(len(pi))])
```

```
Average log-likelihood for MAP with alpha = 1 is nan
Training accuracy for MAP with alpha = 1 is 0.100
Test accuracy for MAP with alpha = 1 is 0.107
```

[Type up your answer here.]

2.4 Q2.4

[2pts] Given this model's assumptions, is it always true that any two pixels x_i and x_j with $i \neq j$ are independent: - when conditioned on c? - after marginalizing over c? - when unconditioned on c?

Provide brief justification for your answers.

[Type up your answer here]

Your answer: - yes - no - no

2.5 Q2.5

[3pts] - How many parameters need to be estimated in the Bernoulli Naive Bayes model? - How many parameters need to be estimated, if we remove the Naive Bayes assumption?

Briefly justify your answers.

[Type up your answer here]

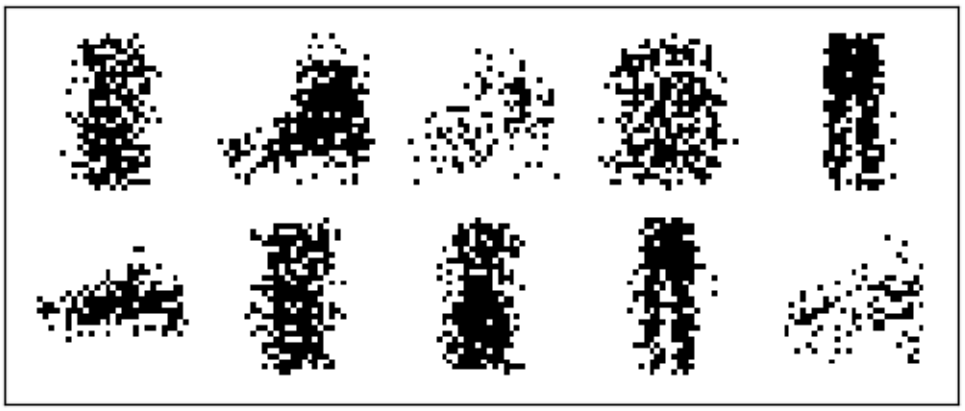
- $KD + K - 1$
- $K \cdot 2^D - 1$

2.6 Q2.6

[4pts] Since we have a generative model for our data, we can do more than just prediction. Randomly sample and plot 10 images from the learned distribution using the MAP estimates. (Hint: You first need to sample the class c, and then sample pixels conditioned on c.)

[ ]:
```
def image_sampler(theta, pi, num_images):
    """ Inputs: parameters theta and pi, and number of images to sample
    Returns the sampled images (N_images x N_features)"""

    # YOU NEED TO WRITE THIS PART
    classes = np.random.choice(len(pi), num_images, p=pi)
    pixel_probs = theta[:, classes]
    sampled_images = np.random.binomial(1, pixel_probs)
    return sampled_images.T


def plot_images(images, ims_per_row=5, padding=5, image_dimensions=(28, 28),
                cmap=matplotlib.cm.binary, vmin=0., vmax=1.):
    """Images should be a (N_images x pixels) matrix."""
```

```
    fig = plt.figure(1)
    fig.clf()
    ax = fig.add_subplot(111)

    N_images = images.shape[0]
    N_rows = np.int32(np.ceil(float(N_images) / ims_per_row))
    pad_value = vmin
    concat_images = np.full(((image_dimensions[0] + padding) * N_rows + padding,
                             (image_dimensions[1] + padding) * ims_per_row +␣
↪padding), pad_value)
    for i in range(N_images):
        cur_image = np.reshape(images[i, :], image_dimensions)
        row_ix = i // ims_per_row
        col_ix = i % ims_per_row
        row_start = padding + (padding + image_dimensions[0]) * row_ix
        col_start = padding + (padding + image_dimensions[1]) * col_ix
        concat_images[row_start: row_start + image_dimensions[0],
                      col_start: col_start + image_dimensions[1]] = cur_image
        cax = ax.matshow(concat_images, cmap=cmap, vmin=vmin, vmax=vmax)
        plt.xticks(np.array([]))
        plt.yticks(np.array([]))

    plt.plot()


sampled_images = image_sampler(theta_est, pi_est, 10)
plot_images(sampled_images)
```



2.7 Q2.7

[5pts] One of the advantages of generative models is that they can handle missing data, or be used to answer different sorts of questions about the model. Assume we have only observed some

pixels of the image. Let $x_E = \{x_p : \text{pixel p is observed}\}$. Derive an expression for $p(x_j|x_E, \theta, \pi)$, the conditional probability of an unobserved pixel j given the observed pixels and distribution parameters. (Hint: You have to marginalize over c.)

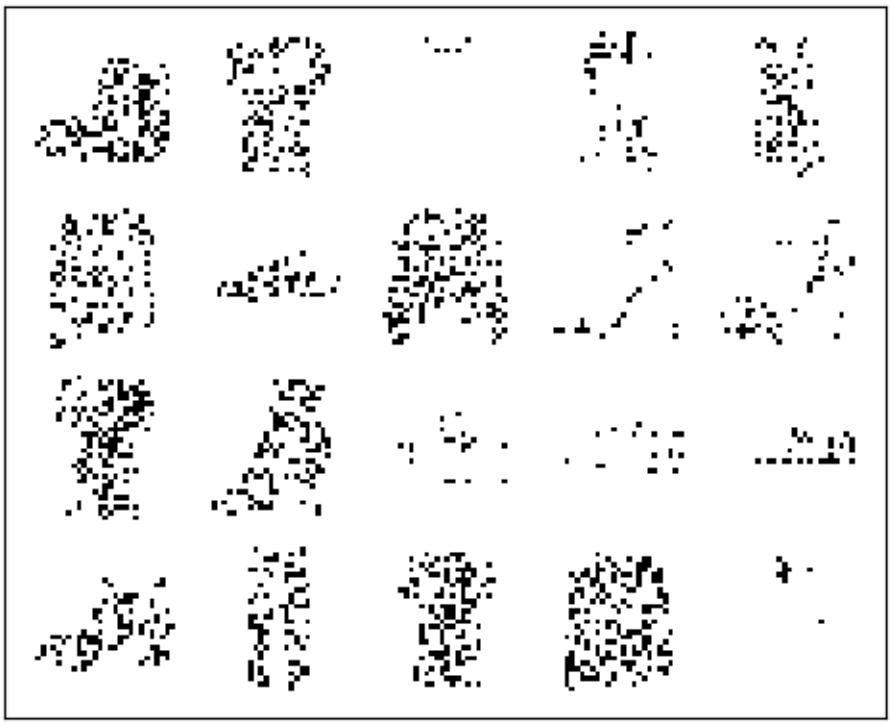
[Type up your derivation here]

Your answer:

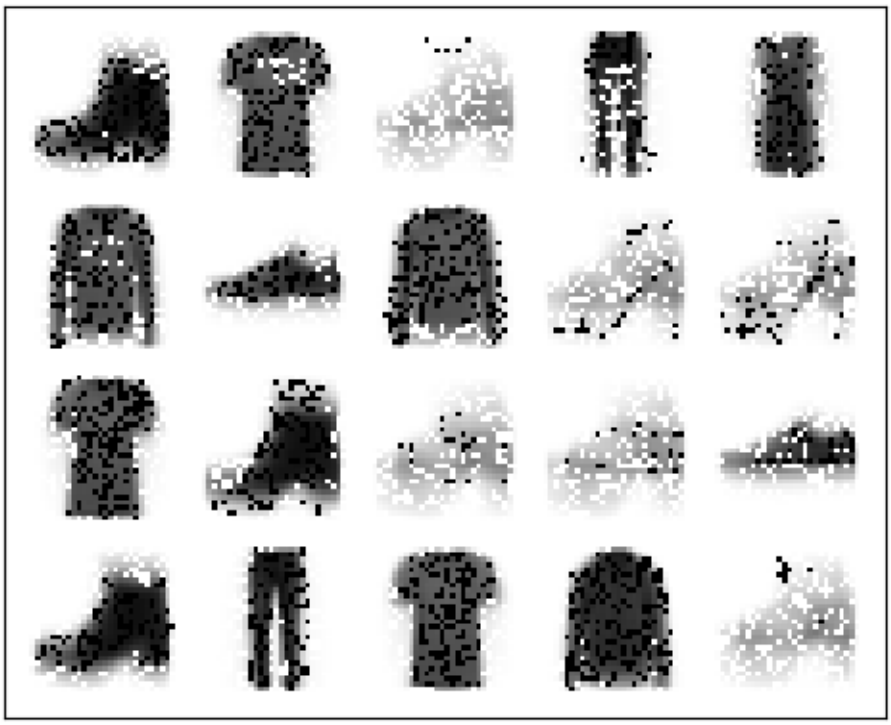
2.8 Q2.8

a) [5pts] We assume that only 30% of the pixels are observed. For the first 30 images in the training set, plot the images when the unobserved pixels are left as white, as well as the same images when the unobserved pixels are filled with the marginal probability of the pixel being 1 given the observed pixels, i.e. the value of the unobserved pixel j is $p(x_j = 1|x_E, \theta, \pi)$.

[ ]:
```
def probabilistic_imputer(theta, pi, original_images, is_observed):
    """Inputs: parameters theta and pi, original_images (N_images x N_features),
        and is_observed which has the same shape as original_images, with a value
        1. in every observed entry and 0. in every unobserved entry.
    Returns the new images where unobserved pixels are replaced by their
    conditional probability"""

    # YOU NEED TO WRITE THIS PART
    N = original_images.shape[0]
    log_joint = np.array([np.sum(np.tile(np.log(theta[:, c]), (N, 1)) * (original_images * is_observed) \
                                 + np.tile(np.log(1 - theta[:, c]), (N, 1)) * ((1 - original_images) * is_observed), axis=1) \
                          + np.log(pi[c]) for c in range(len(pi))])
    log_like = log_joint - np.tile(np.log(np.sum(np.exp(log_joint), axis=0)), (len(pi), 1))

    classes = np.array([np.random.choice(len(pi), 1, p=np.exp(log_like[:,i]))[0] for i in range(N)])
    imputed_images = (1 - is_observed) * theta[:, classes].T + is_observed * original_images
    return imputed_images


num_features = train_images_binarized.shape[1]
is_observed = np.random.binomial(1, p=0.3, size=(20, num_features))
plot_images(train_images_binarized[:20] * is_observed)
```

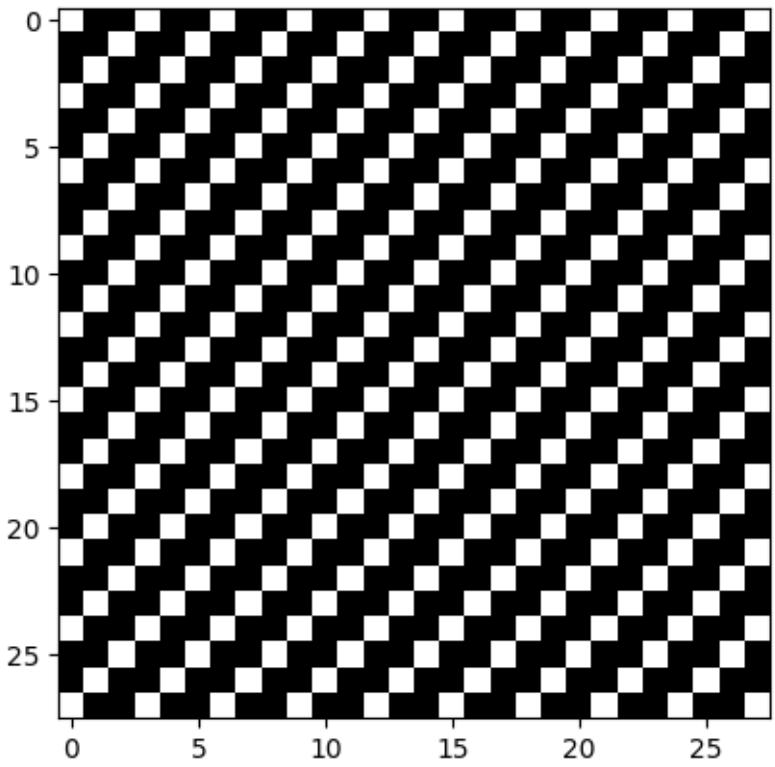


[ ]:
```
imputed_images = probabilistic_imputer(theta_est, pi_est,␣
↪train_images_binarized[:20], is_observed)
plot_images(imputed_images)
```



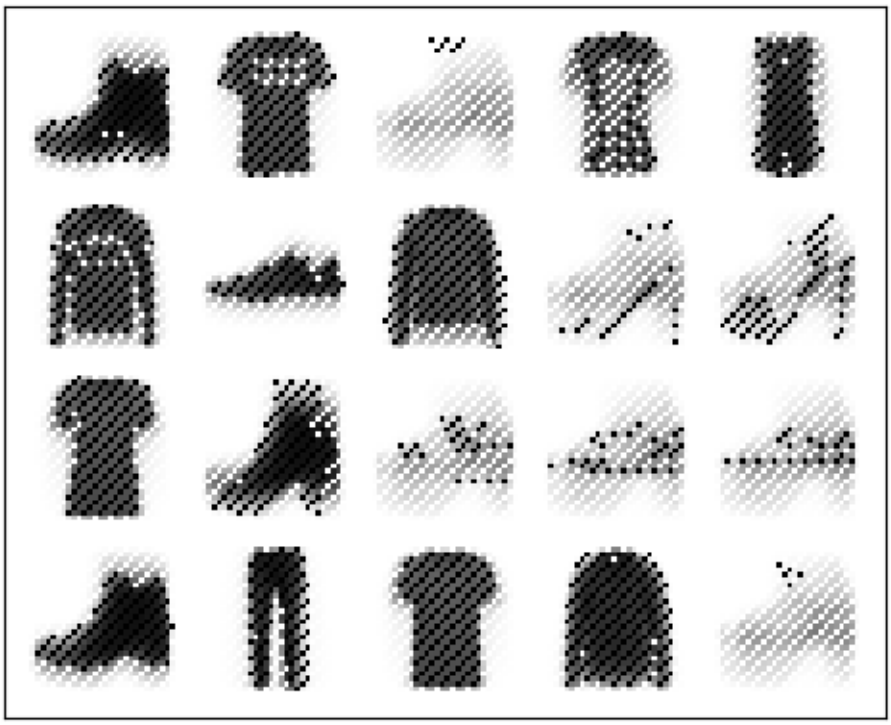
b) [2pts] Now suppose instead of choosing the 30% observed pixels at random, we constructed a grid with roughly evenly spaced observed pixels, as follows:

[ ]:
```
h, w = 28, 28
p = 0.3

num_pixels = h * w
num_indices = int(num_pixels * p)

step_size = num_pixels // num_indices

indices = [(i // w, i % w) for i in range(0, num_pixels, step_size)]
flattened_indices = [row * w + col for row, col in indices]

one_hot_indices = np.zeros(num_pixels, dtype=int)
for index in flattened_indices:
    one_hot_indices[index] = 1

one_hot_matrix = one_hot_indices.reshape(h, w)

import matplotlib.pyplot as plt
plt.imshow(one_hot_matrix, cmap='gray')
plt.show()
```



Next we impute as before, except using this grid of observed indices.

[ ]:
```
repeated_one_hot_indices = np.tile(one_hot_indices, (20, 1))
imputed_images = probabilistic_imputer(theta_est, pi_est,␣
  ↪train_images_binarized[:20], repeated_one_hot_indices)
plot_images(imputed_images)
```



Compare the resulting images you found when using the probabilistic_imputer in part 2.8 a) (random 30% observed) and 2.8 (evenly spaced 30% observed).

What do you find? Why might one of the two resulting samples of images be better than the other? (Hint: consider what the spatial properties of naturally occuring images are.)
You can use the following ground truth images to help inform your answer.

[ ]:
```
plot_images(train_images[:20])
```

2.9 Q2.9

We now consider the Gaussian Naïve Bayes model, parameterized by μ, Σ, π, to model the pixel data. Thus we revert back to representing each data sample using a continuous range of values, i.e. $x^{(i)} \in \mathbb{R}^D$, and we no longer use the binarized version of the data samples. (Note that in reality $x^{(i)} \in [0,1]^D$, but for the purposes of this question we will ignore this.) Here the joint probability distribution of x and c is given by

$$p(x, c|\mu, \Sigma, \pi) = p(c|\pi)p(x|c, \mu, \Sigma) = p(c|\pi)p(x|c, \mu, \Sigma),$$

where $p(x|c, \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^D |\Sigma_c|}} \exp\left(-\frac{1}{2}(x-\mu_c)^T \Sigma_c^{-1}(x-\mu_c)\right)$. As before $c|\pi$ follows a simple categorical distribution, i.e. $p(c|\pi) = \pi_c$.

[1pts] Recall in general for the normal distribution, $\Sigma_c \in \mathbb{R}^{D,D}$. What special form does Σ_c take, in context of the Naive Bayes assumption?

[Type your answer here.]

[7pts] Derive the maximum likelihood estimates of μ_c, Σ_c, π_c for all $c \in \{1, \ldots, 10\}$, then fill in the code block below with the implementation.
(Hint: it is normal if you use a single for loop over the computation of each Σ_c)

[ ]:
```
def train_gnb_mle_estimator(train_images, train_labels, epsilon=1e-6):
    """ Inputs:
        train_images (N_samples x N_features)
```

```
        train_labels (N_samples x N_classes)
        Returns the MLE estimators mu_est, sigma_est, pi_est.
    """
    # YOU NEED TO WRITE THIS PART
    N, D = train_images.shape
    K = train_labels.shape[1]

    # pi_est
    class_counts = np.sum(train_labels, axis=0)
    pi_est = class_counts / N

    # mu_est
    mu_est = np.dot(train_labels.T, train_images) / class_counts[:, np.newaxis]

    # sigma_est
    sigma_est = np.zeros((K, D, D))
    for k in range(0, K):
        x_k = train_images[train_labels[:, k] == 1]
        x_k_sub_mu_k = x_k - mu_est[k]

        squares = x_k_sub_mu_k ** 2

        # Note to marker: it is ok if they do not apply Bessel's correction here
        # i.e. (np.sum(train_labels[:, k])) instead of (np.sum(train_labels[:,␣
↪k]) - 1) is fine
        variances = np.sum(squares, axis=0) / (np.sum(train_labels[:, k]))

        sigma_est[k, :, :] = np.diag(variances)

    return mu_est, sigma_est, pi_est
```

2.10 Q2.10

[6pts] Similar to before, derive an expression for the class log-likelihood $\log p(c|x, \mu, \Sigma, \pi)$ for a single image. Then, complete the implementation for computing the log_likelihoods.

As before report the average log-likelihood $\frac{1}{N}\sum_{i=1}^{N} \log p(c^{(i)}|x^{(i)}, \hat{\theta}, \hat{\pi})$ (where N is the number of samples) on the training test, as well the training and test errors. Use the accuracy function you implemented earlier.

Note: Here because we did not find the MAP estimators for μ, Σ we use a technique called variance smoothing, which adds a small value ϵ to Σ in the implementation, before computing the log-likelihoods. The subsequent part of this question comments on what one might expect if they used the MAP estimators.

[ ]:
```
def gnb_log_likelihood(images, mu, sigma, pi, epsilon):
    sigma = sigma + epsilon
```

```
    # YOU NEED TO WRITE THIS PART
    K = pi.shape[0]
    N, D = images.shape

    log_likelihoods = np.zeros((N, K))
    for k in range(K):
        x_sub_mu_k = images - mu[k]
        inv_sigma_k = 1.0 / np.diag(sigma[k])

        t1 = -0.5 * D * np.log(2. * np.pi) - 0.5 * np.sum(np.log(np.
 ↪diag(sigma[k])))
        t2 = -0.5 * np.sum(x_sub_mu_k ** 2 * inv_sigma_k, axis=1)
        t3 = np.log(pi[k])
        log_likelihoods[:, k] = t1 + t2 + t3

    return log_likelihoods


N_data, train_images, _, train_labels, test_images, _, test_labels =␣
 ↪load_fashion_mnist()
mu_est, sigma_est, pi_est = train_gnb_mle_estimator(train_images, train_labels)

epsilon = 1e-05
loglike_train = gnb_log_likelihood(train_images, mu_est, sigma_est, pi_est,␣
 ↪epsilon)
avg_loglike = np.sum(loglike_train * train_labels) / N_data
train_accuracy = accuracy(loglike_train, train_labels)
loglike_test = gnb_log_likelihood(test_images, mu_est, sigma_est, pi_est,␣
 ↪epsilon)
test_accuracy = accuracy(loglike_test, test_labels)

print(f"Average log-likelihood for MLE (with variance smoothing) is␣
 ↪{avg_loglike:.3f}")
print(f"Training accuracy for MLE (with variance smoothing) is {train_accuracy:.
 ↪3f}")
print(f"Test accuracy for MLE (with variance smoothing) is {test_accuracy:.3f}")
```

```
-2953.008409026207
-3959.9729748013506
-2894.5411770320593
-3734.0503162049718
-3294.3992475578316
-3365.155403860341
-2680.914076179266
-4577.517390525711
-2272.461184207915
-3485.1792318183875
```

```
-2953.008409026207
-3959.9729748013506
-2894.5411770320593
-3734.0503162049718
-3294.3992475578316
-3365.155403860341
-2680.914076179266
-4577.517390525711
-2272.461184207915
-3485.1792318183875
Average log-likelihood for MLE (with variance smoothing) is 570.808
Training accuracy for MLE (with variance smoothing) is 0.642
Test accuracy for MLE (with variance smoothing) is 0.643
```

Using a small value of ϵ is a crude way of circumventing the problem of using the MLE directly.

The MAP for the Gaussian Naive Bayes can be obtained with the help of the conjugate prior for the multivariate normal distribution, which is the normal-inverse-Wishart. We will not derive the MAP for the Gaussian Naive Bayes here - but one can still appreciate the benefit of doing so by drawing a comparison to what was found for the Bernoulli Naive Bayes, when $\alpha = 2$ was used as compared with $\alpha = 1$.

3 Q3 - Bayes Ball Algorithm

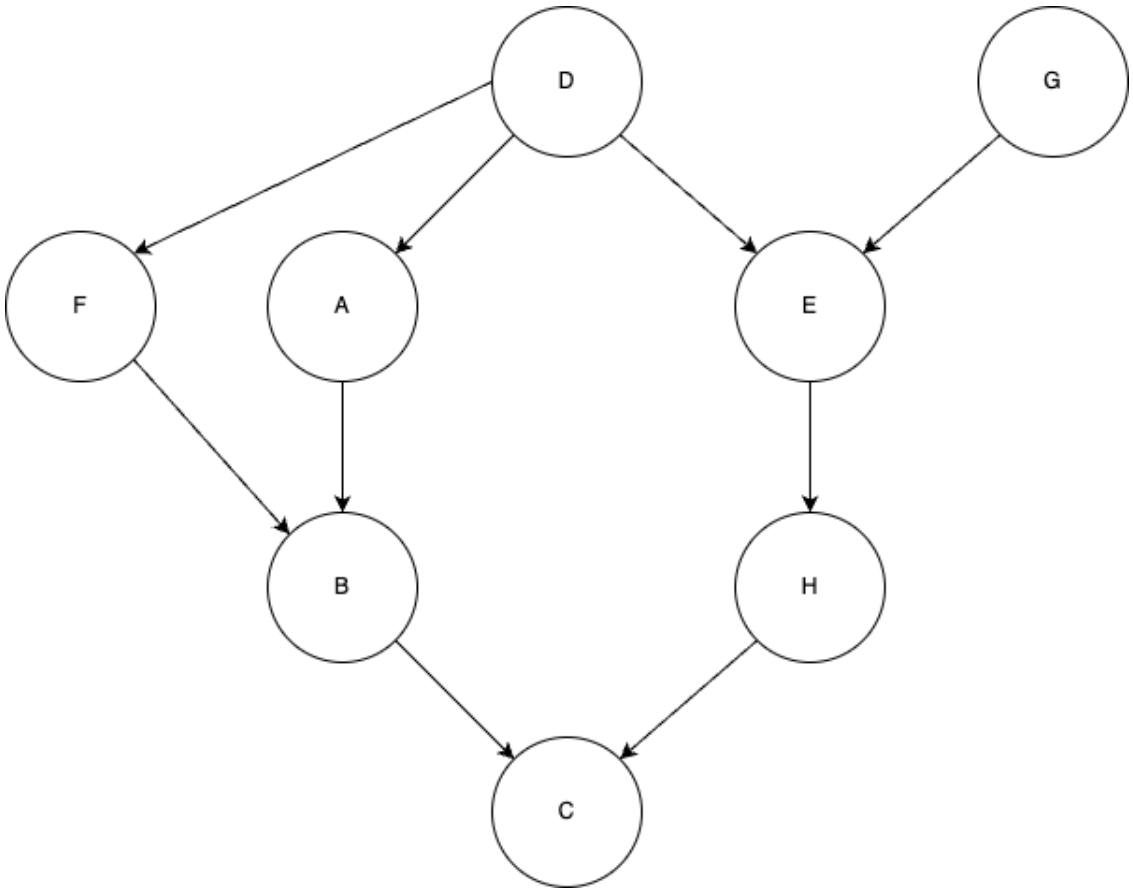
3.1 Q3.1

[4pts]

Consider the following directed acyclic graph. Use the Bayes ball algorithm to compute all the nodes that are independent of C given

i: $\{B, D\}$,

ii: $\{H\}$.



[Type up your answer below]

Answer:

i: $\{A, F\}$

ii: $\emptyset$

3.2 Q3.2

[3pts] For the graph shown above, show using the factorization of the joint probabilities whether D is independent of H given E. You may suppose that the domain of the variable G is **G**.

Hint: $P(E|D) = _\{G\} P(E|D,G) P(G) $.

[Type up your answer below]

Answer:

Note that in this DAG model

$$P(D, G, E, H) = P(D)P(G)P(E|D, G)P(H|E).$$

This can be argued, for example, by summing over all possible values of C, B, F, A.

$$P(H|E,D) = \frac{\sum_G P(D)P(G)P(E|D,G)P(H|E)}{\sum_{G,H} P(D)P(G)P(E|D,G)P(H|E)} = \frac{\sum_G P(G)P(E|D,G)P(H|E)}{\sum_G P(G)P(E|D,G)} = P(H|E).$$

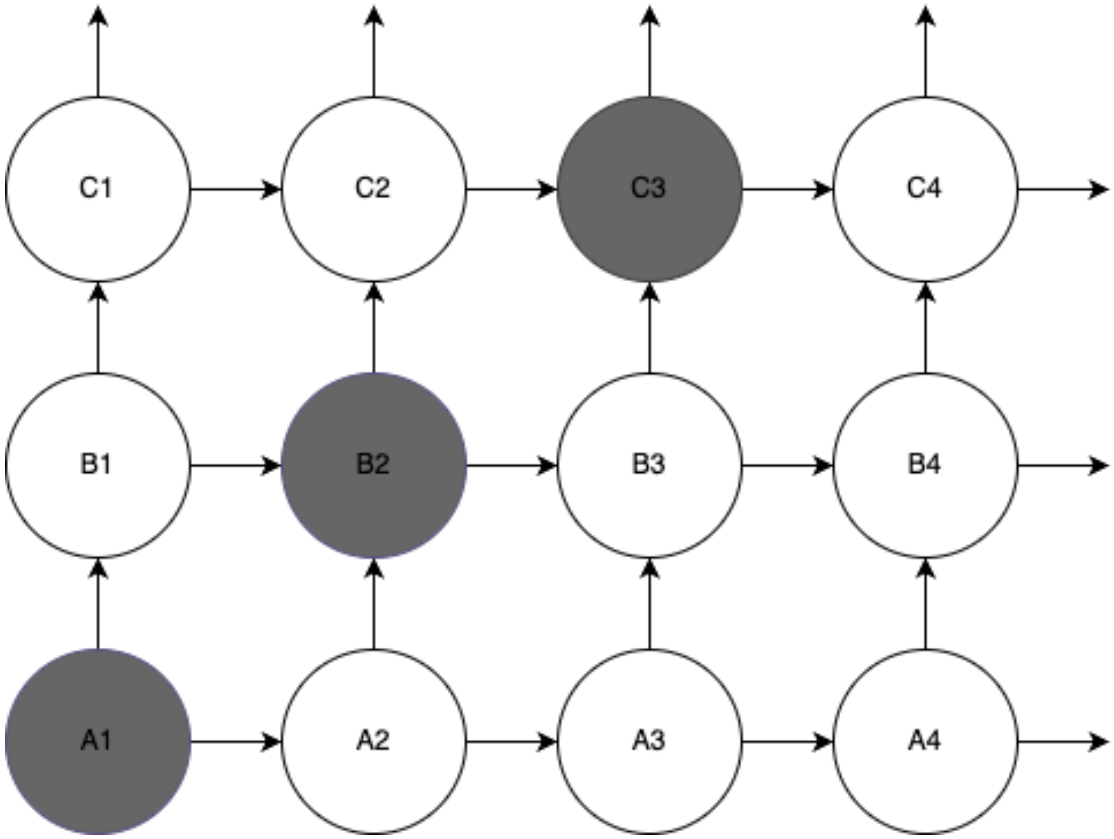
Therefore H is conditionally independent of D.

Note: There was no need to use the hint.

3.3 Q3.3

[3pts] Consider the following lattice structure with the diagonal nodes shaded. You may assume that it extends arbitrarily far upwards and also to the right.

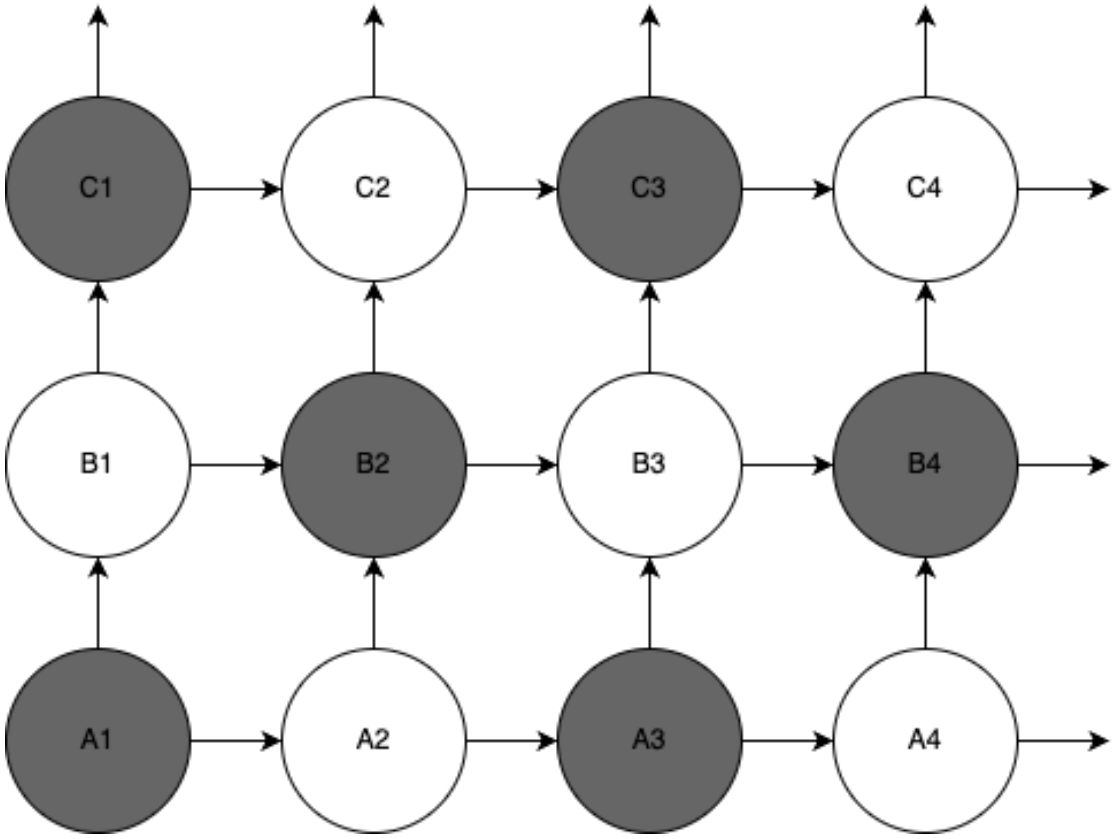
Conditioned on the shaded nodes, what are the set of all nodes independent of C_2? Justify your answer.



[Type up your answer below] All nodes are conditionally dependent on C_2. On the top/left side of the lattice, all nodes are either parents, children, or "sister" nodes that do not pass through any shaded nodes. We can cross over to the bottom/right side to B_3 since they are both parents of a conditional node C_3, and as a result we can get any node on the bottom side as well.

3.4 Q3.4

[3pts] Consider the following checkboard variant of the lattice from the previous question. Assume it extends indefinitely to the right and upwards (not shown in the diagram).



Condition on the shaded nodes. What are the set of all nodes that are conditionally independent of the variable B_3? (You may also state the set conditionally dependent on B_3 if it is easier).

As a result, what can be concluded about the "shape" of the set of conditionally independent nodes for an arbitrary unshaded node?

[Type up your answer below]

The conditionally dependent set is $\{D_1, C_2, A_4\}$ given the shaded nodes. The conditionally independent nodes will be the complement of this set.

For a general node, the conditionally dependent set will look like the diagonal along the direction $\searrow$.