V1: Last Edited: 2024-01-25

Changes: v0 -> v1 Q2: Remove the data point that cause log(0) Q1: three -> four

- **Deadline:** Feb 5, at 23:59PM.
- **Submission:** You need to submit your solutions through Crowdmark, including all your derivations, plots, and your code. You can produce the files however you like (e.g. LATEX, Microsoft Word, etc), as long as it is readable. Points will be deducted if we have a hard time reading your solutions or understanding the structure of your code.
- **Collaboration policy**: After attempting the problems on an individual basis, you may discuss and work together on the assignment with up to two classmates. However, **you must write your own code and write up your own solutions individually and explicitly name any collaborators** at the top of the homework.

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 four options for what to do with each email: You can list it as important email, show it to the user, 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	15	0
Show	5	1
Folder	1	40
Delete	0	150

Q1.1

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

In [19]:

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

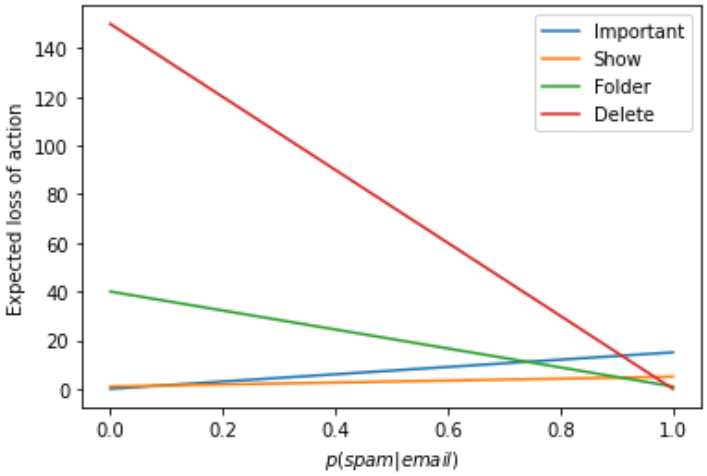
In [20]:

```
#@title
losses = [[15, 0],[5, 1], [1, 40],[0, 150]]
actions_names = ['Important','Show', 'Folder', '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.
    return losses[action][1]*(1-prob_spam)+losses[action][0]*prob_spam

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()
```

Out[20]:

```
<matplotlib.legend.Legend at 0x7fd32b133040>
```



Q1.2

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

In [21]:

```
def optimal_action(prob_spam):
    #TODO: return best action given the probability of spam.
    #Hint: np.argmin might be helpful.
    loss=[]
    for action in actions_names:
      index = actions_names.index(action)
      loss.append(expected_loss_of_action(prob_spam, index))
    return actions_names[np.argmin(loss)]
```

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.

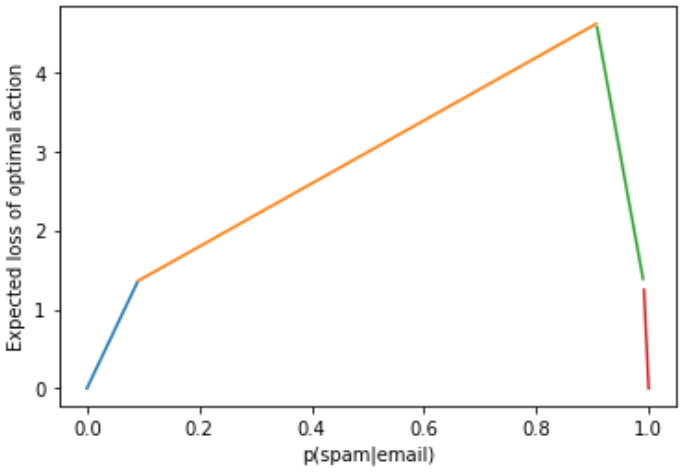
In [22]:

```
prob_range = np.linspace(0., 1., num=600)
Prob, Loss = [[], [], [], []], [[], [], [], []]#partition
optimal_losses = []
optimal_actions = []

    # TODO: Compute the optimal action and its expected loss for
    # probability of spam given by p.

#use a dictionary to combine the action with index
actions_name_dict = {'Important': 0,'Show': 1, 'Folder': 2, 'Delete': 3}

optimal_actions = [optimal_action(p) for p in prob_range]

for index in range(len(optimal_actions)):
```

```
  action_index = actions_name_dict[optimal_actions[index]]
  optimal_losses.append(expected_loss_of_action(prob_range[index], action_index))
  Prob[action_index].append(prob_range[index])
  Loss[action_index].append(optimal_losses[index])

colors = ['red', 'green', 'blue', 'black']
for inputPindex in range(len(Prob)):
  plt.plot(Prob[inputPindex], Loss[inputPindex])

plt.xlabel('p(spam|email)')
plt.ylabel('Expected loss of optimal action')



#plt.plot(prob_range, optimal_losses, color = )
```

Out[22]:

```
Text(0, 0.5, 'Expected loss of optimal action')
```



Q1.4

[4pts] For exactly which range of the probabilities of an email being spam should we delete an email?

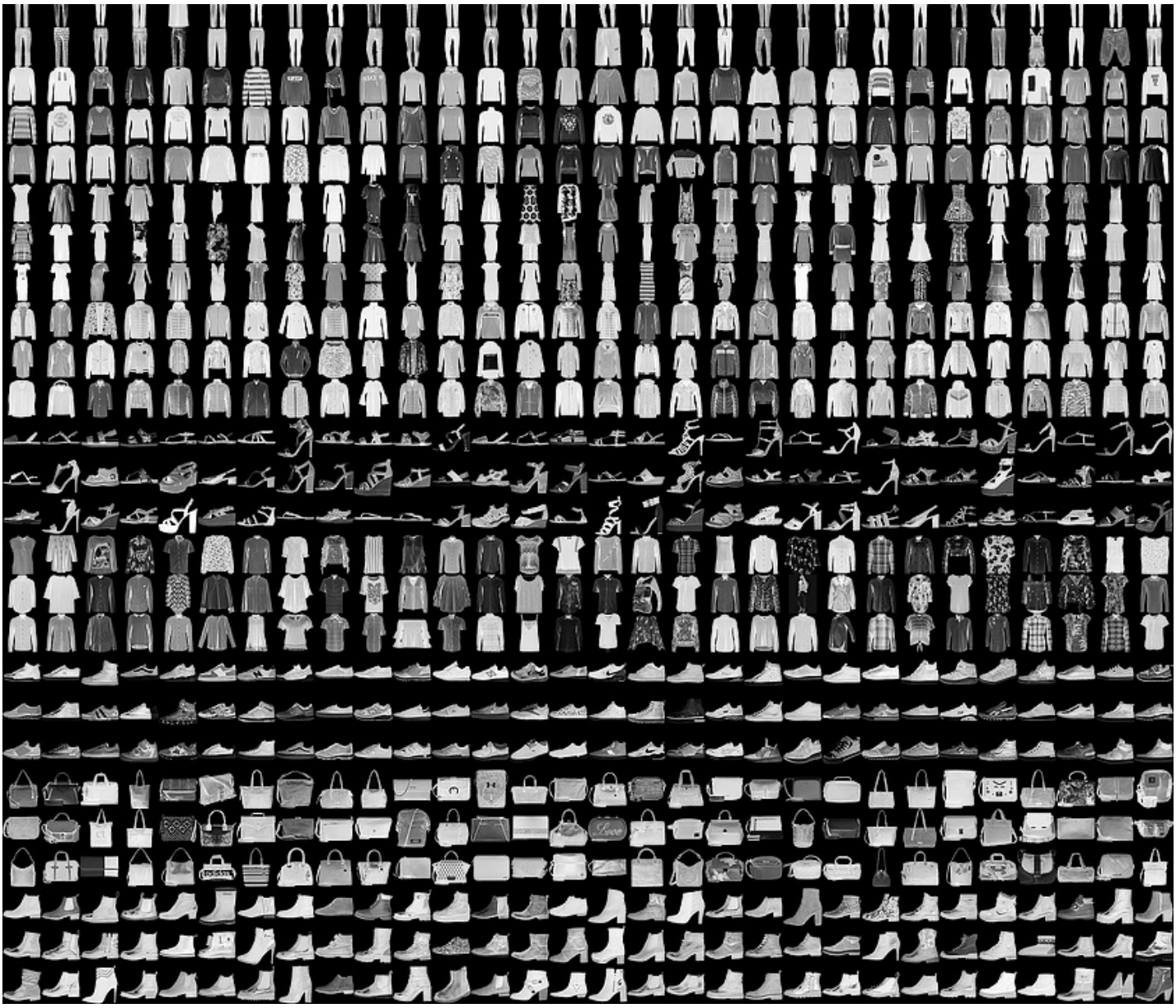
Find the exact answer by hand using algebra.

[Type up your derivation here]

Your answer:

By definition of optimal action, we need to find the action which minimize the loss. Then, if we need to delete an email, we must make sure that deleting action is the optimal action, which means E(Loss|Delete) $\leq$ E(Loss|Important), E(Loss|Show) and E(Loss|Folder).\ E(Loss|Imortant) = 15 *P(spam|email)+ 0* (1-P(spam|email)) = 15P(spam|email)\ E(Loss|Show) = 5 *P(spam|email)+ 1* (1-P(spam|email)) = 1 + 4P(spam|email)\ E(Loss|Foler) = 1 *P(spam|email)+ 40* (1-P(spam|email))=40 - 39P(spam|email)\ E(Loss|Delete) = 0 *P(spam|email)+ 150* (1-P(spam|email))=150 - 150P(spam|email)\ Now solve E(Loss|Delete) $\leq$ E(Loss|Important):\ 150 - 150P(spam|email) $\leq$ 15P(spam|email), then we will have 0.909 $\leq$ P(spam|email)\ Now solve E(Loss|Delete) $\leq$ E(Loss|Show):\ 150 - 150P(spam|email) $\leq$ 1 + 4P(spam|email), then we will have 0.968 $\leq$ P(spam|email)\ Now solve E(Loss|Delete) $\leq$ E(Loss|Foler):\ 150 - 150P(spam|email) $\leq$ 40 - 39P(spam|email), then we will have 0.991 $\leq$ P(spam|email)\ Therefore, we find the range of probabilities of an email being spam is [0.991,1]

Q2 - Naïve Bayes, A Generative Model



In this question, we'll fit a Naïve Bayes model to the fashion MNIST dataset, and use this model for making predictions and generating new images from the same distribution. MNIST is a dataset of 28x28 black-and-white images of items of clothing. We represent each image by a vector $x^{(i)} \in \{0,1\}^{784}$, where 0 and 1 represent white and black pixels respectively. Each class label $c^{(i)}$ is a different item of clothing, which in the code is represented by a 10-dimensional one-hot vector.

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

$$\begin{aligned} p(x, c|\theta, \pi) &= p(c|\pi)p(x|c, \theta) \\ &= p(c|\pi) \prod_{j=1}^{784} p(x_j|c, \theta), \end{aligned}$$

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

We begin by learning the parameters θ and π. The following code will download and prepare the training and test sets.

In [23]:

```
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_dat
a, 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 > .5).astype(float)
    test_images = (partial_flatten(test_images) / 255.0 > .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_labels, test_images, test_labels
```

Q2.1

[2pts] Derive the expression for the Maximum Likelihood Estimator (MLE) of θ and π.

[Type up your derivation here]

Your answer:

2.1

Suppose we have N objects $x^{(i)}$ with labels $c^{(i)}$,

The likelihood function can be represented as

$$L(\theta, \pi) = \prod_{i=1}^{n} P(x^{(i)}, c^{(i)} \mid \theta, \pi)$$

Then the log-likelihood function is

$$\ell(\theta, \pi) = \log(L) = \sum_{i=1}^{N} \log P(x^{(i)}, c^{(i)} \mid \theta, \pi)$$

$$= \sum_{i=1}^{N} \left[\log \left(P(c^{(i)} \mid \pi) \prod_{j=1}^{784} P(x_j^{(i)} \mid c^{(i)}, \theta) \right) \right]$$

Since each class label $c^{(i)}$ is a different item of clothing, and we have 10 different types,

$$P(c^{(i)} = c \mid \pi) = \prod_{c=0}^{9} \pi_c^{\mathbb{1}\{c^{(i)} = c\}}$$

Then, deriving the log-likelihood function.

$$\ell(\theta, \pi) = \sum_{i=1}^{N} \left[\log P(c^{(i)} \mid \pi) + \log \left(\prod_{j=1}^{784} P(x_j^{(i)} \mid c^{(i)}, \theta) \right) \right]$$

$$= \sum_{i=1}^{N} \left[\mathbb{1}\{c^{(i)} = c\} \sum_{c=0}^{9} \log \pi_c + \sum_{j=1}^{784} \left[x_j^{(i)} \log \theta_{jc} + (1 - x_j^{(i)}) \log(1 - \theta_{jc}) \right] \right]$$

We know the MLE of π and θ are separable, so we can compute MLE of π_c and θ_{jc} independently.

MLE of θ_{jc}: before taking the derivative respect to θ_{jc}, we need to understand that the value of θ is determined by $j \in [1, 784]$ (pixels, interger) and $C \in [0, 9]$ (class, integer only). to make the expression of θ_{jc} more rigorious, I'll use a new variable k to represent the pixel.

$$\frac{\partial}{\partial \theta_{jc}} \ell = \sum_{i=1}^{N} \sum_{k=1}^{784} \frac{\partial}{\partial \theta_{jc}} \left[x_k^{(i)} \log \theta_{kc^{(i)}} + (1 - x_k^{(i)}) \log(1 - \theta_{kc^{(i)}}) \right]$$

when $k \neq j \Rightarrow$ this pixel doesn't contribute to parameter θ.

when $k = j$, the MLE is valid with $\mathbb{1}\{c^{(i)} = c\}$

$$= \frac{\sum_{i=1}^{N} \left[(x_j^{(i)} - \theta_{jc}^{(i)}) \ \mathbb{1}\{c^{(i)} = c\} \right]}{\sum_{i=1}^{N} \left[\theta_{jc^{(i)}} (1 - \theta_{jc^{(i)}}) \ \mathbb{1}\{c^{(i)} = c\} \right]} = 0$$

Therefore, we solve above get $\hat{\theta}_{jc} = \dfrac{\sum_{i=1}^{N} \left[x_j^{(i)} \mathbb{1}\{c^{(i)} = c\} \right]}{\sum_{i=1}^{N} \mathbb{1}\{c^{(i)} = c\}}$

MLE of π:

we still a new variable k, k has range $0 \sim 9$, represents class.

$$\frac{\partial}{\partial \pi_c} \ell = \sum_{i=1}^{N} \sum_{k=0}^{9} \frac{\partial}{\partial \pi_c} \left[\mathbb{1}(c^{(i)} = k) \log \pi_k \right]$$

$$= \sum_{i=1}^{N} \sum_{k=0}^{9} \frac{\partial}{\partial \pi_c} \left[\mathbb{1}\{c^{(i)} = k\} \log \pi_k + 1 - \sum_{k=0}^{9} \pi_k \right] \quad \text{since} \sum_{\text{all classes}} \pi_k = 1$$

when $k \neq c$, the derivative will be 0 because we take derivative respect to 0

$k = c$, $\frac{\partial}{\partial \pi_k} \pi_k = 1$

so $\frac{\partial}{\partial \pi_c} \ell = \sum_{i=1}^{N} \left[\frac{\mathbb{1}\{c^{(i)} = c\}}{\pi_c} - 1 \right] = 0 \Rightarrow \hat{\pi}_c = \dfrac{\sum_{i=1}^{N} \mathbb{1}\{c^{(i)} = c\}}{N}$

Q2.2

[4pts] 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 $\mathrm{Beta}(2, 2)$. 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:

2.2

Since the prior given is Beta(2,2), so the PDF of prior will be

$$f(\theta) = \frac{\theta(1-\theta)}{6}$$

In MAP, we're supposed to optimize $P(x, c|\theta, \pi) f(\theta)$, and we have the likelihood function from previous question.

$$\log\left(P(x|\theta) f(\theta)\right) = \left(\sum_{i=1}^{N} \sum_{c=0}^{9} \mathbb{1}\{C^{(i)}=c\} \log \pi_c + \sum_{i=1}^{N} \sum_{j=1}^{784} \left[x_j^{(i)} \log \theta_{jc} + (1-x_j^{(i)}) \log(1-\theta_{jc}) \right] \right) + \log\left(\frac{\theta(1-\theta)}{6} \right)$$

$$= \left(\sum_{i=1}^{N} \sum_{c=0}^{9} \mathbb{1}\{C^{(i)}=c\} \log \pi_c + \sum_{i=1}^{N} \sum_{j=1}^{784} \left[x_j^{(i)} \log \theta_{jc} + (1-x_j^{(i)}) \log(1-\theta_{jc}) \right] \right) + \log(\theta) + \log(1-\theta) - \log(6)$$

Then take the derivative respect to θ, and use part of result from previous question.

$$\sum_{i=1}^{N} \left(\frac{x_j^{(i)}}{\theta_{jc}} - \frac{1-x_j^{(i)}}{1-\theta_{jc}} \right) \mathbb{1}\{C^{(i)}=c\} + \frac{1}{\theta} - \frac{1}{1-\theta} = 0$$

$$\sum_{i=1}^{N} \frac{x_j^{(i)} - \theta_{jc}}{\theta_{jc}(1-\theta_{jc})} \mathbb{1}\{C^{(i)}=c\} + \frac{1}{\theta_{jc}} - \frac{1}{1-\theta_{jc}} = 0$$

Solving for θ, we have $\hat{\theta}_{map} = \dfrac{\sum_{i=1}^{N} \left[x_j^{(i)} \mathbb{1}\{C^{(i)}=c\} \right] + 1}{\sum_{i=1}^{N} \left[\mathbb{1}\{C^{(i)}=c\} \right] + 2}$

In [24]:

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

```
        estimator pi_est (N_classes)"""
    #use the MLE of pi: number of correct lable / number of samples
    pi_est = np.mean(train_labels, 0)
    #use the result above
    theta_est = (1 + np.matmul(train_images.T, train_labels)) / (2 + np.sum(train_labels
, 0))
    return theta_est, pi_est
```

Q2.3

[5pts] 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.

[Type up your derivation here]

Your answer:

2.3

By Bayes' rule,

$$P(c|x,\theta,\pi) = \frac{P(c,x|\theta,\pi)}{P(x|\theta,\pi)}$$

Now take log to $P(c|x,\theta,\pi)$, we have

$$\log P(c|x,\theta,\pi) = \log \frac{P(c,x|\theta,\pi)}{P(x|\theta,\pi)}$$

$$= \log P(c,x|\theta,\pi) - \log P(x|\theta,\pi)$$

$$= \log\left[P(c|\pi)\,P(x|c,\theta)\right] - \log P(x|\theta,\pi)$$

$$= \log P(c|\pi) + \log P(x|c,\theta) - \log P(x|\theta,\pi)$$

where $P(c|\pi) = \pi_c$,

$$\log P(x|c,\theta) = \prod_{j=1}^{784} P(x_j|c,\theta) = \sum_{j=1}^{784}\left[x_j \log\theta_{jc} + (1-x_j)\log(1-\theta_{jc})\right]$$

$$P(x|\theta,\pi) = \sum_{c=0}^{9} P(c|\pi)\,P(x|c,\theta)$$

For the following function log_likelihood :

π_c is given by MLE $\hat{\pi}_c$,

θ_{jc} is given by MAP estimator $\hat{\theta}_{MAP_{jc}}$

x_j is given by images

In [25]:

```
from numpy.core.fromnumeric import argmax

from numpy.ma.core import log
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
    p_c_given_pi = pi
    log_p_c_given_pi = np.log(pi)
    log_p_x_given_c_theta = np.matmul(images,np.log(theta))+(np.matmul(1-images,np.log(1
-theta)))
    log_p_c_x_given_theta_pi = log_p_c_given_pi + log_p_x_given_c_theta

    p_x_given_c_theta = np.exp(log_p_x_given_c_theta)
    p_x_given_theta_pi = np.matmul(p_x_given_c_theta, p_c_given_pi)
    log_p_x_given_theta_pi = np.log(p_x_given_theta_pi).reshape(-1, 1)


    log_like = log_p_c_x_given_theta_pi - log_p_x_given_theta_pi
    return log_like




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
    prediction = np.argmax(log_like, axis =1)
    actual = np.argmax(labels, axis = 1)
    accuracy = np.mean(prediction == actual)
    return accuracy







N_data, train_images, train_labels, test_images, test_labels = load_fashion_mnist()
theta_est, pi_est = train_map_estimator(train_images, train_labels)

loglike_train = log_likelihood(train_images, 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, theta_est, pi_est)
test_accuracy = accuracy(loglike_test, test_labels)

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

```
Average log-likelihood for MAP is -34.231
Training accuracy for MAP is 0.651
Test accuracy for MAP is 0.638
```

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 indenepdent given c? How about after marginalizing over c? Explain your answer.

[Type up your answer here]

Your answer:

Naive Bayes model is built on the assumption of conditional independence.

If we have 2 pixels x_i and x_j, $i \neq j$,

x_i and x_j are independent given $C = c$.

So I agree with $p(x_i, x_j \mid c) = p(x_i \mid c)\, p(x_j \mid c)$

But x_i and x_j are marginal independent if $p(x_i, x_j) = p(x_i)\, p(x_j)$

Now we marginalize over C, C is the class from 0 to 9

$$p(x_i, x_j) = \sum_{C=0}^{9} p(x_i, x_j, c)$$

$$= \sum_{C=0}^{9} p(x_i, x_j \mid c)\, p(c)$$

$$= \sum_{C=0}^{9} p(x_i \mid c)\, p(x_j \mid c)\, p(c) \quad \text{by conditional independence}$$

we notice that

$$p(x_i)\, p(x_j) = \sum_{C=0}^{9} p(x_i, c) \sum_{C=0}^{9} p(x_j, c)$$

$$= \sum_{C=0}^{9} p(x_i \mid c)\, p(c) \sum_{C=0}^{9} p(x_j \mid c)\, p(c) \quad \text{by Bayes' Rule}$$

$$\neq \sum_{C=0}^{9} p(x_i \mid c)\, p(x_j \mid c)\, p(c) = p(x_i, x_j)$$

Therefore we conclude that

$$p(x_i, x_j) \neq p(x_i)\, p(x_j)$$

So x_i x_j are not marginally independent

Q2.5

[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.)

In [26]:

```
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

    # Get number of features for each image
    N_features = theta.shape[0]

    # Sample the class for each image using the class probabilities 'pi'
    sampled_classes = np.random.choice(pi.shape[0], size=num_images, p=pi)

    # Get the probabilities of each pixel being 1 for the sampled classes
    probs = theta[:, sampled_classes]

    # Generate the binary values for each pixel of each image using the pixel probabiliti
es
    sampled_images = np.random.binomial(1, probs)

    # Transpose the sampled images to have 'num_images' rows and 'N_features' columns
    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)
```

Q2.6

[4pts] 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 $x_E = \{x_p$: 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.)

2.6 we have x_E (observed), x_j (unobserved)

$$P(x_j \mid x_E, \theta, \pi) = \frac{P(x_j, x_E \mid \theta, \pi)}{P(x_E \mid \theta, \pi)}$$

$$= \frac{\sum_{c=0}^{9} P(x_j, x_E, c \mid \theta, \pi)}{\sum_{c=0}^{9} P(x_E, c \mid \theta, \pi)} \quad \text{marginalize over } c$$

$$= \frac{\sum_{c=0}^{9} P(x_j, x_E \mid c, \theta) P(c \mid \pi)}{\sum_{c=0}^{9} P(x_E \mid c, \theta) P(c \mid \pi)} \quad \text{since } P(x, c \mid \theta, \pi) = P(c \mid \pi) P(x \mid c, \theta)$$

$$= \frac{\sum_{c=0}^{9} P(x_j \mid c, \theta) P(x_E \mid c, \theta) P(c \mid \pi)}{\sum_{c=0}^{9} P(x_E \mid c, \theta) P(c \mid \pi)} \quad \text{by conditional independence}$$

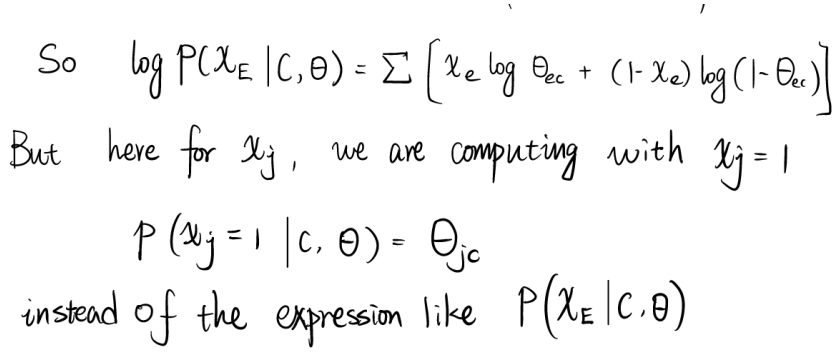
where

$$P(c \mid \pi) = \pi_c$$

Since $x_E = \{x_p :$ pixel p is observed$\}$

we can express $P(x_E \mid c, \theta) = \prod P(x_e \mid c, \theta)$

$$= \prod \left(\theta_{ec}^{x_e} + (1 - \theta_{ec})^{1 - x_e} \right)$$



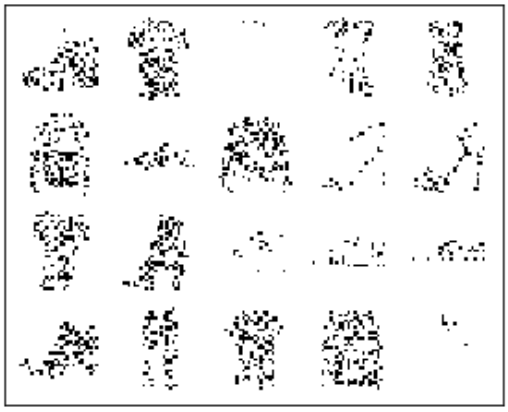
Q2.7

[4pts] We assume that only 30% of the pixels are observed. For the first 20 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 $p(x_j = 1|x_E, \theta, \pi)$.

In [27]:

```
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"""
    # We compute his function from the expression in previous
    # Set up all values I need
    p_c_given_pi = pi
    N_images = original_images.shape[0]
    N_features = original_images.shape[1]
    imputed_images = np.array(original_images)

    for image in range(N_images):
      # Find x_e, dim = N_obs x 1
      x_e = original_images[image][is_observed[image] == 1]
      # Find theta_ec, dim = N_obs x c, for the observed pixels
      theta_ec = theta[is_observed[image] == 1]
      log_p_xe_c_theta = np.matmul(np.log(theta_ec.T), x_e) + np.matmul(np.log(1 - theta
_ec.T), 1 - x_e)
      p_xe_given_c_theta = np.exp(log_p_xe_c_theta) # Nobs x c
      # Calculate the denominator
      p_xe_given_theta_pi = np.sum(p_xe_given_c_theta * p_c_given_pi)
      # Find probability of xj given c and theta, which = theta_jc as we claimed in previ
ous
      theta_jc = theta[is_observed[image] == 0] # Dim = N_unobs x c
      # Calculate the numerator, notice that p_xe_given_c_theta *  theta_jc will
      # yield (N_unobs,1), so we use sum() with respect to axis = 1
      p_xj_xe_given_theta_pi = np.sum(p_xe_given_c_theta *  theta_jc * pi, axis = 1)
      # Calculate the probability of xj given xe, theta, pi
      p_x_j = p_xj_xe_given_theta_pi / p_xe_given_theta_pi
      imputed_images[image, is_observed[image] == 0] = p_x_j


    # Return the imputed images
    return imputed_images



num_features = train_images.shape[1]
```

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



In [28]:

```
imputed_images = probabilistic_imputer(theta_est, pi_est, train_images[:20], is_observed
)
plot_images(imputed_images)
```