CSC411H1 L0101, Fall 2018
Assignment 6

Name: JingXi, Hao
Student Number: 1000654188

Due Date: 21st November, 2018 11:59pm

I. Learning the parameters (3 marks)

In the first step, we'll learn the parameters of the model given the responsibilities, using the MAP criterion. This corresponds to the M-step of the E-M algorithm.

In lecture, we discussed the E-M algorithm in the context of maximum likelihood (ML) learning. The MAP case is only slightly different from ML: the only difference is that we add a prior probability term to the objective function in the M-step. In particular, recall that in the context of ML, the M-step maximizes the objective function:

$$\sum_{i=1}^{N}\sum_{k=1}^{K} r_k^{(i)}[logPr(z^{(i)}=k)+logp(x^{(i)}|z^{(i)}=k)], \tag{5}$$

where the $r_k^{(i)}$ are the responsibilities computed during the E-step. In the MAP formulation, we add the (log) prior probability of the parameters:

$$\sum_{i=1}^{N}\sum_{k=1}^{K} r_k^{(i)}[logPr(z^{(i)}=k)+logp(x^{(i)}|z^{(i)}=k)]+logp(\pi)+logp(\mathbf{\Theta}) \tag{6}$$

Our prior for $\mathbf{\Theta}$ is as follows: every entry is drawn independently from a beta distribution with parameters a and b. The beta distribution is discussed in Lecture 14, but here it is again for reference:

$$p(\theta_{k,j}) \propto \theta_{k,j}^{a-1}(1-\theta_{k,j})^{b-1} \tag{7}$$

Recall that $\propto$ means "proportional to". i.e., the distribution has a normalizing constant which we're ignoring because we don't need it for the M-step.

For the prior over mixing proportions π, we'll use the Dirichlet distribution, which is the conjugate prior for the multinomial distribution. It is a distribution over the **probability simplex**, i.e. the set of vectors which define a valid probability distribution. The distribution takes the form

$$p(\pi) \propto \pi_1^{a_1-1}\pi_2^{a_2-1}\cdots\pi_K^{a_K-1}. \tag{8}$$

For simplicity, we use a symmetric Dirichlet prior where all the a_k parameters are assumed to be equal. Like the beta distribution, the Dirichlet distribution has a normalizing constant which we don't

need when updating the parameters. The beta distribution is actually the special case of the Dirichlet distribution for $K = 2$. You can read more about it on Wikipedia if you're interested.

Your tasks for this part are as follows:

1. Derive the M-step update rules for π and $\mathbf{\Theta}$ by setting the partial derivatives of Eqn 6 to zero. Your final answers should have the form:

$$\pi_k \leftarrow \cdots$$
$$\theta_{k,j} \leftarrow \cdots$$

Be sure to show your steps. (There's no trick here; you've done very similar questions before.)

Solution: In order to derive the M-step update rules for π and $\mathbf{\Theta}$, we have to take partial derivatives of **Eqn (6)** with respective to π_k and $\theta_{k,j}$ and then set the partial derivatives to 0.

- First, we derive the M-step rules for π. In order to do that, we need to maximize **Eqn (6)** subject to $\sum_{k=1}^{K} \pi_k = 1$. Then, we employ Lagrange multiplier algorithm. As we can see that the relevant terms with π_k in **Eqn (6)** are $r_k^{(i)} log Pr(z^{(i)} = k)$ and $log p(\pi)$. Hence, let

$$\begin{aligned} f &= \sum_{i=1}^{N}\sum_{k=1}^{K} r_k^{(i)} log Pr(z^{(i)} = k) + log p(\pi) \\ &= \sum_{i=1}^{N}\sum_{k=1}^{K} r_k^{(i)} log \pi_k + log \prod_{k=1}^{K} \pi_k^{a_k - 1} \quad \#substitute\ Pr(z^{(i)} = k) = \pi_k\ and\ p(\pi) \propto \pi_1^{a_1-1}\pi_2^{a_2-1}\cdots\pi_K^{a_K-1} \\ &= \sum_{i=1}^{N}\sum_{k=1}^{K} r_k^{(i)} log \pi_k + \sum_{k=1}^{K}(a_k - 1) log \pi_k \end{aligned}$$

and

$$g = \sum_{k=1}^{K} \pi_k - 1$$

Then, there exists a Lagrange multiplier λ such that $\partial \mathcal{L} / \partial \pi_k = 0$, where $\mathcal{L} = f - \lambda g$.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \pi_k} &= \frac{\partial}{\partial \pi_k}[\sum_{i=1}^{N}\sum_{k=1}^{K} r_k^{(i)} log \pi_k + \sum_{k=1}^{K}(a_k - 1) log \pi_k - \lambda(\sum_{k=1}^{K} \pi_k - 1)] \\ &= \frac{\sum_{i=1}^{N} r_k^{(i)}}{\pi_k} + \frac{a_k - 1}{\pi_k} - \lambda \end{aligned} \tag{9}$$

Then, we set **Eqn (9)** to 0, where we obtain that

$$\frac{\sum_{i=1}^{N} r_k^{(i)}}{\pi_k} + \frac{a_k - 1}{\pi_k} - \lambda = 0$$
$$\frac{\sum_{i=1}^{N} r_k^{(i)}}{\pi_k} + \frac{a_k - 1}{\pi_k} = \lambda$$
$$\frac{\sum_{i=1}^{N} r_k^{(i)} + a_k - 1}{\pi_k} = \lambda \tag{10}$$
$$\pi_k = \frac{\sum_{i=1}^{N} r_k^{(i)} + a_k - 1}{\lambda}$$
$$\pi_k = \frac{a_k - 1 + \sum_{i=1}^{N} r_k^{(i)}}{\lambda}$$

Since $\sum_{k=1}^{K} \pi_k = 1$, therefore, by substitution **Eqn (10)**, we have that

$$\sum_{k=1}^{K} \frac{\sum_{i=1}^{N} r_k^{(i)} + a_k - 1}{\lambda} = 1$$
$$\lambda = \sum_{k=1}^{K} [\sum_{i=1}^{N} r_k^{(i)} + a_k - 1] \tag{11}$$
$$\lambda = \sum_{k=1}^{K} \sum_{i=1}^{N} r_k^{(i)} + \sum_{k=1}^{K} a_k - K$$

Therefore, by substitution **Eqn (11)**, we find that

$$\pi_k \leftarrow \frac{a_k - 1 + \sum_{i=1}^{N} r_k^{(i)}}{\sum_{k'=1}^{K} \sum_{i=1}^{N} r_{k'}^{(i)} + \sum_{k'=1}^{K} a_{k'} - K}$$

- Then, derive the M-step rules for $\mathbf{\Theta}$. In order to do that, we need to maximize **Eqn (6)**, which is to take partial derivative with respect to $\theta k, j$ and set it to 0. As we can see that the relevant terms with $\theta k, j$ in **Eqn (6)** are $r_k^{(i)} logp(x^{(i)}|z^{(i)} = k)$ and $logp(\mathbf{\Theta})$. Hence, let

$$\begin{aligned}
\mathcal{L} &= \sum_{i=1}^{N} \sum_{k=1}^{K} r_k^{(i)} logp(x^{(i)}|z^{(i)} = k) + logp(\mathbf{\Theta}) \\
&= \sum_{i=1}^{N} \sum_{k=1}^{K} r_k^{(i)} log \prod_{j=1}^{D} \theta_{k,j}^{x_j^{(i)}} (1 - \theta_{k,j})^{1 - x_j^{(i)}} + log \prod_{k=1}^{K} \prod_{j=1}^{D} \theta_{k,j}^{a-1} (1 - \theta_{k,j})^{b-1} \quad \#substitute\ p(x^{(i)}|z^{(i)}=k)\ and\ p(\mathbf{\Theta}) \\
&= \sum_{i=1}^{N} \sum_{k=1}^{K} r_k^{(i)} [\sum_{j=1}^{D} x_j^{(i)} log(\theta_{k,j}) + (1 - x_j^{(i)}) log(1 - \theta_{k,j})] + \sum_{k=1}^{D} \sum_{j=1}^{D} [(a-1) log\theta_{k,j} + (b-1) log(1 - \theta_{k,j})]
\end{aligned}$$

Then, we compute the partial derivative of $\mathcal{L}$ with respect to $\theta k, j$, where we have that

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \theta k, j} &= \sum_{i=1}^{N} r_k^{(i)} \left(\frac{x_j^{(i)}}{\theta_{k,j}} + \frac{(-1)(1-x_j^{(i)})}{1-\theta_{k,j}}\right) + \frac{a-1}{\theta_{k,j}} + \frac{(-1)(b-1)}{1-\theta_{k,j}} \\
&= \sum_{i=1}^{N} r_k^{(i)} \frac{x_j^{(i)} - \theta_{k,j}}{\theta_{k,j}(1-\theta_{k,j})} + \frac{a - a\theta_{k,j} - 1 + \theta_{k,j} + \theta_{k,j} - b\theta_{k,j}}{\theta_{k,j}(1-\theta_{k,j})} \\
&= \sum_{i=1}^{N} r_k^{(i)} \frac{x_j^{(i)} - \theta_{k,j}}{\theta_{k,j}(1-\theta_{k,j})} + \frac{(2-a-b)\theta_{k,j} + a - 1}{\theta_{k,j}(1-\theta_{k,j})} \\
&= \frac{\sum_{i=1}^{N} r_k^{(i)}(x_j^{(i)} - \theta_{k,j})}{\theta_{k,j}(1-\theta_{k,j})} + \frac{(2-a-b)\theta_{k,j} + a - 1}{\theta_{k,j}(1-\theta_{k,j})}
\end{aligned} \tag{12}
$$

Then, we set **Eqn (12)** to be 0, we have that

$$
\begin{aligned}
\frac{\sum_{i=1}^{N} r_k^{(i)}(x_j^{(i)} - \theta_{k,j})}{\theta_{k,j}(1-\theta_{k,j})} + \frac{(2-a-b)\theta_{k,j} + a - 1}{\theta_{k,j}(1-\theta_{k,j})} &= 0 \\
\sum_{i=1}^{N} r_k^{(i)}(x_j^{(i)} - \theta_{k,j}) + (2-a-b)\theta_{k,j} + a - 1 &= 0 \\
\sum_{i=1}^{N} r_k^{(i)} x_j^{(i)} - \sum_{i=1}^{N} r_k^{(i)} \theta_{k,j} + (2-a-b)\theta_{k,j} + a - 1 &= 0 \\
\theta_{k,j} &= \frac{\sum_{i=1}^{N} r_k^{(i)} x_j^{(i)} + a - 1}{\sum_{i=1}^{N} r_k^{(i)} + a + b - 2} \\
\theta_{k,j} &= \frac{a - 1 + \sum_{i=1}^{N} r_k^{(i)} x_j^{(i)}}{a + b - 2 + \sum_{i=1}^{N} r_k^{(i)}}
\end{aligned}
$$

Hence, we obtain that

$$
\theta_{k,j} \leftarrow \frac{a - 1 + \sum_{i=1}^{N} r_k^{(i)} x_j^{(i)}}{a + b - 2 + \sum_{i=1}^{N} r_k^{(i)}}
$$

2. Take these formulas and use them to implement the functions Model.update_pi and Model.update_theta in mixture.py. Each one should be implemented in terms of NumPy matrix and vector operations. Each one requires only a few lines of code, and should not involve any for loops.

   To help you check your solution, we have provided the function checking.check_m_step. If this check passes, you're probably in good shape.

   To convince us of the correctness of your implementation, **please include the output of running** mixture.print_part_1_values(). Note that we also require you to submit mixture.py through MarkUs.

   Solution:

   Please see the detailed code implementation in the file, **mixture.py**.

We show the output of running mixture.print_part_1_values() below.

```
>>> print_part_1_values()
pi[0] 0.08499999999999992
pi[1] 0.12999999999999987
theta[0, 239] 0.6427106227106232
theta[3, 298] 0.46573612495845823
```

II. Posterior inference (3 marks)

Now we derive the posterior probability distribution $p(z|\mathbf{x}_{obs})$, where $\mathbf{x}_{obs}$ denotes the subset of the pixels which are observed. In the implementation, we will represent partial observations in terms of variables m_j^i, where $m_j^i = 1$ if the jth pixel of the ith image is observed, and 0 otherwise. In the implementation, we organize the m_j^i's into a matrix $\mathbf{M}$ which is the same size as $\mathbf{X}$.

1. Derive the rule for computing the posterior probability distribution $p(z|\mathbf{x})$. Your final answer should look something like

$$Pr(z = k|\mathbf{x}) = \cdots$$

where the ellipsis represents something you could actually implement. Note that the image may be only partially observed.

Solution:

In this question, we need to derive the rule for computing the posterior probability distribution $p(z = k|\mathbf{x}_{obs})$. By Bayes' rule, we have that

$$\begin{aligned}
p(z = k|\mathbf{x}_{obs}) &= \frac{p(\mathbf{x}_{obs}|z = k)p(z = k)}{p(\mathbf{x}_{obs})} \\
&= \frac{p(\mathbf{x}_{obs}|z = k)p(z = k)}{\sum_{k'=1}^{K} p(\mathbf{x}_{obs}|z = k')p(z = k')} \\
&= \frac{p(z = k)\prod_{i=1}^{N} p(\mathbf{x}_{obs}^{(i)}|z = k)}{\sum_{k'=1}^{K} p(z = k')\prod_{i=1}^{N} p(\mathbf{x}_{obs}^{(i)}|z = k')} \\
&= \frac{\pi_k \prod_{i=1}^{N} p(\mathbf{x}_{obs}^{(i)}|z = k)}{\sum_{k'=1}^{K} \pi_{k'}\prod_{i=1}^{N} p(\mathbf{x}_{obs}^{(i)}|z = k')} \quad \#since\ p(z = k) = \pi_k \\
&= \frac{\pi_k \prod_{i=1}^{N}\prod_{j=1}^{D}[\theta_{k,j}^{x_j^{(i)}}(1-\theta_{k,j})^{1-x_j^{(i)}}]^{m_j^{(i)}}}{\sum_{k'=1}^{K}[\pi_{k'}\prod_{i=1}^{N}\prod_{j=1}^{D}[\theta_{k',j}^{x_j^{(i)}}(1-\theta_{k',j})^{1-x_j^{(i)}}]^{m_j^{(i)}}]} \quad \#where\ m_j^{(i)}\ is\ partial\ observation
\end{aligned}$$

Then, we have derived the rule for computing the posterior probability distribution.

2. Implement the method Model.compute_posterior using your solution to the previous question. While your answer to Question 1 was probably given in terms of probabilities, we do the computations in terms of log probabilities for numerical stability. We've already filled in part of the implementation, so your job is to compute $logp(z, \mathbf{x})$. as described in the method's doc string.

   Your implementation should use NumPy matrix and vector operations, rather than a for loop. *Hint: There are two lines in Model.log_likelihood which are almost a solution to this question. You can reuse these lines as part of the solution, except you'll need to modify them to deal with partial observations.*

   To help you check your solution, we've provided the function checking.check_e_step. Note that this check only covers the case where the image is fully observed, so it doesn't fully verify your solution to this part.

   Solution:
   Please see the detailed code implementation in the file, **mixture.py**.

3. Implement the method Model.posterior_predictive_means, which computes the posterior predictive means of the missing pixels given the observed ones. *Hint: this requires only two very short lines of code, one of which is a call to Model.compute_posterior.*

   To convince us of the correctness of the implementation for this part and the previous part, **please include the output of running** mixture.print_part_2_values(). Note that we also require you to submit mixture.py through MarkUs.

   Solution:
   Please see the detailed code implementation in the file, **mixture.py**.

   We show the output of running mixture.print_part_2_values() below.

```
>>> print_part_2_values()
R[0, 2] 0.1748895149211729
R[1, 0] 0.6885376761092292
P[0, 183] 0.6516151998131037
P[2, 628] 0.4740801724913301
```

III. Conceptual questions (3 marks)

This section asks you to reflect on the learned model. We tell you the outcomes of the experiments, so that you can **do this part independently of the first 2**. Each question can be answered in a few sentences.

1. In the code, the default parameters for the beta prior over Θ were $a = b = 2$. If we instead used $a = b = 1$ (which corresponds to a uniform distribution), the MAP learning algorithm would

have the problem that it might assign zero probability to images in the test set. Why might this happen? *Hint: what happens if a pixel is always 0 in the training set, but 1 in the test image?*

Solution:

Based on the derivation of $\theta_{k,j}$ from Part 1 Question 1, we obtain that

$$\theta_{k,j} \leftarrow \frac{a - 1 + \sum_{i=1}^{N} r_k^{(i)} x_j^{(i)}}{a + b - 2 + \sum_{i=1}^{N} r_k^{(i)}}$$

Therefore, we can see that if we have $a = b = 1$, then this formula for $\theta_{k,j}$ would be

$$\theta_{k,j} \leftarrow \frac{\sum_{i=1}^{N} r_k^{(i)} x_j^{(i)}}{\sum_{i=1}^{N} r_k^{(i)}}$$

Thus, in this case, if we have a pixel always be 0 in the training set, then this pixel would end with $\theta_{k,j} = 0$ during training process. Thus, this certain pixel in the test image would be assigned with a probability of zero even though it has a value of 1. Therefore, if we use a uniform distribution, then the MAP learning algorithm would cause the problem that it might assign zero probability to images in the test set, which implies we may end with a bad model trained and would not produce good estimation.

2. The model from Part 2 gets significantly higher average log probabilities on both the training and test sets, compared with the model from Part 1. This is counterintuitive, since the Part 1 model has access to additional information: labels which are part of a true causal explanation of the data (i.e. what digit someone was trying to write). Why do you think the Part 2 model still does better?

   Solution:

   I think Part 2 model still does better than the Part 1 model does because Part 2 model contains 100 classes and Part 1 model only has 10 classes. In this case, the model is given top half of the image and should figure out the digit one the image. This task is different with the one to figure out which number shown in the image by given the whole image. Thus, 10 classes are definitely not enough to train a good model in this case since people have distinct handwriting habits. Hence, I think Part 2 model still does better.

3. The function print_log_probs_by_digit_class computes the average log-probabilities for different digit classes in both the training and test sets. In both cases, images of 1's are assigned far higher log-probability than images of 8's. Does this mean the model thinks 1's are far more common than 8's? i.e., If you sample from its distribution, will it generate far more 1's than 8's? Why or why not?

   Solution:

   This does NOT mean that the model thinks 1's are far more common than 8's. The log-probability value is a "confidence" value. The larger the value of log-probability, the more confidence the

model is on its prediction. Thus, both models are more confident on their predictions on 1 than their predictions on 8. This makes sense since 1 is a number that is very similar for its top half part and its bottom half part, but for 8, only given top half part, it may be classified as other digit.