Programming Assignment 4: Image Completion using Mixture of Bernoullis

Deadline: Tuesday, April 4, at 11:59pm
TA: Renjie Liao (csc321ta@cs.toronto.edu)

Submission: You must submit two files through MarkUs[1]: a PDF file containing your writeup, titled `a4-writeup.pdf`, and your completed code file `mixture.py`. Your writeup must be typeset using LaTeX.

The programming assignments are individual work. See the Course Information handout[2] for detailed policies.

You should attempt all questions for this assignment. Most of them can be answered at least partially even if you were unable to finish earlier questions. If you were unable to run the experiments, please discuss what outcomes you might hypothetically expect from the experiments. If you think your computational results are incorrect, please say so; that may help you get partial credit.

Overview

In this assignment, we'll implement a probabilistic model which we can apply to the task of image completion. Basically, we observe the top half of an image of a handwritten digit, and we'd like to predict what's in the bottom half. An example is shown in Figure 1.

This assignment is meant to give you practice with the techniques covered in the lectures on probabilistic modeling. It's not as long as it looks from this handout. The solution requires about 8-10 lines of code.

Mixture of Bernoullis model

The images we'll work with are all 28×28 binary images, *i.e.* the pixels take values in $\{0, 1\}$. We ignore the spatial structure, so the images are represented as 784-dimensional binary vectors.

A mixture of Bernoullis model is like the other mixture models we've discussed in this course. Each of the mixture components consists of a collection of independent Bernoulli random variables. I.e., conditioned on the latent variable $z = k$, each pixel x_j is an independent Bernoulli random variable with parameter $\theta_{k,j}$:

$$p(\mathbf{x}^{(i)} \mid z = k) = \prod_{j=1}^{D} p(x_j^{(i)} \mid z = k) \tag{1}$$

$$= \prod_{j=1}^{D} \theta_{k,j}^{x_j^{(i)}} \left(1 - \theta_{k,j}\right)^{1 - x_j^{(i)}} \tag{2}$$

Try to understand where this formula comes from. You'll find it useful when you do the derivations.

This can be written out as the following generative process:

[1] `https://markus.teach.cs.toronto.edu/csc321-2017-01`
[2] `http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/syllabus.pdf`

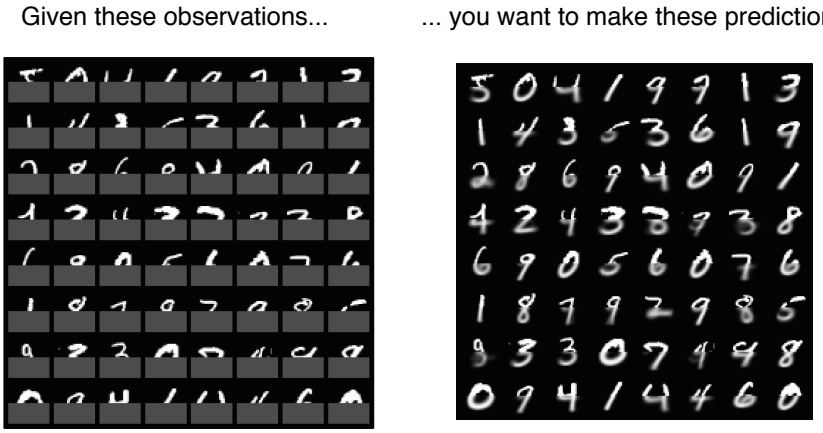

Figure 1: An example of the observed data (left) and the predictions about the missing part of the image (right).

Sample z from a multinomial distribution with parameter vector $\boldsymbol{\pi}$.

For $j = 1, \ldots, D$:

Sample x_j from a Bernoulli distribution with parameter $\theta_{k,j}$, where k is the value of z.

It can also be written mathematically as:

$$z \sim \text{Multinomial}(\boldsymbol{\pi}) \tag{3}$$

$$x_j \,|\, z = k \sim \text{Bernoulli}(\theta_{k,j}) \tag{4}$$

Summary of notation

We will refer to three dimensions in our model:

- $N = 60{,}000$, the number of training cases. The training cases are indexed by i.
- $D = 28 \times 28 = 784$, the dimension of each observation vector. The dimensions are indexed by j.
- K, the number of components. The components are indexed by k.

The inputs are represented by $\mathbf{X}$, an $N \times D$ binary matrix. In the E-step, we compute $\mathbf{R}$, the matrix of responsibilities, which is an $N \times K$ matrix. Each row gives the responsibilities for one training case.

The trainable parameters of the model, written out as vectors and matrices, are:

$$\boldsymbol{\pi} = \begin{pmatrix} \pi_1 \\ \pi_2 \\ \vdots \\ \pi_K \end{pmatrix}$$

$$\boldsymbol{\Theta} = \begin{pmatrix} \theta_{1,1} & \theta_{1,2} & \cdots & \theta_{1,D} \\ \theta_{2,1} & \theta_{2,2} & & \theta_{2,D} \\ \vdots & & \ddots & \vdots \\ \theta_{K,1} & \theta_{K,2} & \cdots & \theta_{K,D} \end{pmatrix}$$

The rows of $\boldsymbol{\Theta}$ correspond to mixture components, and columns correspond to input dimensions.

Part 1: Learning the parameters (4 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. This is discussed in detail in Section 5.2 of the Lecture 18 notes, and you should read that section carefully before starting this part. 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:

$$\sum_{i=1}^{N} \sum_{k=1}^{K} r_k^{(i)} \left[\log \Pr(z^{(i)} = k) + \log p(\mathbf{x}^{(i)} \mid z^{(i)} = k) \right], \tag{5}$$

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:

$$\sum_{i=1}^{N} \sum_{k=1}^{K} r_k^{(i)} \left[\log \Pr(z^{(i)} = k) + \log p(\mathbf{x}^{(i)} \mid z^{(i)} = k) \right] \; + \; \log p(\boldsymbol{\pi}) + \log p(\boldsymbol{\Theta}) \tag{6}$$

Our prior for $\boldsymbol{\Theta}$ is as follows: every entry is drawn independently from a beta distribution with parameters a and b. The beta distribution is discussed in Section 3.1 of the Lecture 17 notes, 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 $\boldsymbol{\pi}$, 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.[3] The distribution takes the form

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

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

[3] I.e., they must be nonnegative and sum to 1.

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.[4]

Your tasks for this part are as follows:

1. **(2 marks)** Derive the M-step update rules for $\boldsymbol{\Theta}$ and $\boldsymbol{\pi}$ 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. You may want to refer to Example 5 from the Lecture 17 notes, and Example 4 from the Lecture 18 notes.

   Note: both of these optimization problems involve inequality constraints, i.e. $0 \leq \theta_{k,j} \leq 1$ and $\pi_k \geq 0$. You can ignore the inequality constraints, since they will never be tight. On the other hand, the optimization for $\boldsymbol{\pi}$ involves a normalization constraint $\pi_1 + \cdots + \pi_K = 1$. You can deal with this using Lagrange multipliers, similarly to Example 4 from Lecture 18. Alternatively, you can define $\pi_K = 1 - \pi_1 - \cdots - \pi_{K-1}$ and then solve the unconstrained optimization problem for $\pi_1, \ldots, \pi_{K-1}$. Either approach will work, but the first one is slightly shorter.

2. **(2 marks)** 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.[5]

   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.

3. **(0 marks)** The function `learn_from_labels` learns the parameters of the model from the *labeled* MNIST images. The values of the latent variables are chosen based on the digit class labels, i.e. the latent variable $z^{(i)}$ is set to k if the ith training case is an example of digit class k. In terms of the code, this means the matrix $\mathbf{R}$ of responsibilities has a 1 in the (i, k) entry if the ith image is of class k, and 0 otherwise.

   Run `learn_from_labels` to train the model. It will show you the learned components (i.e. rows of $\boldsymbol{\Theta}$) and print the training and test log-likelihoods. *You do not need to submit anything for this part. It is only for your own satisfaction.*

Part 2: Posterior inference (3 marks)

Now we derive the posterior probability distribution $p(z \mid \mathbf{x}_{\text{obs}})$, where $\mathbf{x}_{\text{obs}}$ denotes the subset of the pixels which are observed. In the implementation, we will represent partial observations in terms

[4] `http://en.wikipedia.org/wiki/Dirichlet_distribution`

[5] How this check works is beyond the scope of the class. Essentially, each step of E-M can be shown to optimize a particular objective function which is a lower bound on the log-likelihood. We can check the E-M steps by verifying that each update actually maximizes the lower bound. You can read more about this in Neal and Hinton, 1998, "A view of the EM algorithm that justifies incremental, sparse, and other variants."

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

$$\Pr(z = k \mid \mathbf{x}) = \cdots \tag{9}$$

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

   Hints: For this derivation, you probably want to express the observation probabilities in the form of Eqn 2. You may also wish to refer to Example 1 from the Lecture 18 notes.

2. **(1 mark)** 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 $\log p(z, \mathbf{x})$ 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.

3. **(1 mark)** 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`. You may wish to refer to Example 2 from the Lecture 18 notes.

   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.

4. **(0 marks)** Run the function `train_with_em`, which trains the mixture model using E-M. It plots the log-likelihood as a function of the number of steps.[6] You can watch how the mixture components change during training.[7] It also shows the model's image completions after every step. You can watch how they improve over the course of training. At the very end, it outputs the training and test log-likelihoods. The final model for this part should be much better than the one from Part 1. *You do not need to submit anything for this part. It's only for your own satisfaction.*

[6] Observe that it uses a log scale for the number of E-M steps. This is always a good idea, since it can be difficult to tell if the training has leveled off using a linear scale. You wouldn't know if it's stopped improving or is just improving very slowly.

[7] It's likely that 5-10 of the mixture components will "die out" during training. In general, this is something we would try to avoid using better initializations and/or priors, but in the context of this assignment it's the normal behavior.

Part 3: 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. **(1 mark)** In the code, the default parameters for the beta prior over $\boldsymbol{\Theta}$ 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? You may want to re-read Section 2.1 of the Lecture 17 notes.*

2. **(1 mark)** 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?

3. **(1 mark)** 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?