STAT 339: HOMEWORK 5 (CLUSTERING AND HIDDEN MARKOV MODELS)

UPDATED: DUE VIA GITHUB BY THE END OF FINALS)

Instructions. Create a directory called `hw5` in your `stat339` GitHub repo. Your main writeup should be called `hw5.pdf`, and any source code should either be in that directory, a subdirectory within it, or a "library" directory at the top level (in the case of files defining functions used in multiple assignments).

I suggest placing the definitions of any "helper" functions in a separate file which you load (in Python, `import`) from your main file.

I will access your work by cloning your repository; make sure that any file path information is written relative to your repo – don't use absolute paths on your machine, or the code won't run for me!

You may use any language you like to do this assignment — the tasks are stated in a language-neutral way — but Python is recommended.

You may also use any typesetting software to prepare your writeup, but the final document should be a PDF. LaTeXis highly encouraged.

All data files referred to in the problems below can be found at

`http://colindawson.net/data/<filename>.csv`.

Date: Last Revised: May 12, 2020.

1. Clustering with a Mixture of Normals Model

Consider a mixture of D-dimensional multivariate Normals model with K components, where K is specified in advance. The data consists of N vectors, $\mathbf{x}_1, \ldots, \mathbf{x}_N$, each with D real-valued entries, representing observable features. The unknown parameters are:

- the D-dimensional mean vectors for each cluster: $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K$
- the $D \times D$ covariance matrices for each cluster, $\boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_K$
- the mixture weights $\pi_1, \ldots, \boldsymbol{\pi}_K$

(a) Write a function called `kmeans()`, which takes the following inputs:

(i) `X=`: the $N \times D$ data matrix $\mathbf{X}$ to be clustered

(ii) `K=`: an integer (2 or larger) giving the number of clusters

(iii) `initialization=`: the name of an initialization function (which in turn can be expected to take a data matrix and a number of clusters, and perhaps other optional inputs passed via `**kwargs`),

The function should do the following:

(i) Standardize the columns of $\mathbf{X}$ as z-scores, so that Euclidean distances between points and centers are meaningful (storing the means and standard deviations so that the cluster centers can be transformed back at the end)

(ii) Call the initialization function passed to `intialization=`

(iii) Run one pass of K-means, terminating when $\mathbf{z}$ stops changing

and return a dictionary with the following entries:

(i) `"z"`: an array of cluster indicators

(ii) `"mu"`: a $K \times D$ array whose rows are cluster centers

(b) Write a plotting function, `plotClusters()`, with the following inputs:

(i) `X=`: an $N \times D$ dataset $\mathbf{X}$

(ii) `z=`: an indicator vector $\mathbf{z}$

(iii) `mu=`: a $K \times D$ mean matrix

(iv) `dims=`: a tuple of two column indices, d_1 and d_2, each between 1 and D

The function should produce a scatterplot of columns d_1 and d_2 of the data, color coded by the clusters assigned in $\mathbf{z}$, overlaying the d_1 and d_2 coordinates of the cluster centers in a distinct symbol.

(c) Test your `kmeans()` and plotting function using the iris data (complete data with labels in `iris.csv`, or if you prefer, features only in `iris_features.csv` and labels only in `iris_labels.csv`).

(d) Write a function, `clusterNormalMix()` that does clustering with a mixture of Normals model. Inputs should be the same as the inputs for the K-means function, plus some additional inputs.

(i) `method=` The algorithm to employ. Possible values should include `"kmeans"`, `"EM"`, and `"Gibbs"`.

(ii) `covariance=` an argument to control how flexible the covariance matrices should be, choosing from the following options:

- `"fixed"`: No learning of covariance matrices (e.g., just set the covariances to the identity matrix in the "standardized" feature space). (If `"kmeans"` is used, this is the only valid option)
- `"isotropicShared"`: Fixing covariance matrices at a scalar multiple of the identity matrix, with the same scalar used for all clusters
- `"isotropicSeparate"`: Fixing covariance matrices at a scalar multiple of the identity, but allowing different scalars for each cluster
- `"diagonal"`: Fixing covariance matrices to be diagonal, but allowing the diagonal entries to differ both by feature and by cluster
- `"unconstrained"`: Unrestricted covariance (you do not need to implement this one in the regularized EM/sampling cases)

(iii) `prior=`: Default to `None`, since it is not used with K-means, or unregularized EM, but otherwise a dictionary specifying prior parameters. Assume a symmetric Dirichlet prior for the cluster weights, $\boldsymbol{\pi}$, independent Normal priors on the means, $\boldsymbol{\mu}_1, \ldots, bmu_K$, and independent Gamma priors on the inverse diagonal elements of the covariances, $\boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_K$. You can assume that the data is standardized, so that the same prior parameters can be used for each feature dimension; we then need only four scalar hyperparameters: α, the Dirichlet effective sample size, h_0, controlling the prior on the cluster means (i.e., assume the prior on $\boldsymbol{\mu}_k$ is $\mathcal{N}(\mathbf{0}, h_0^{-1}\mathbf{I})$), and a and b, the shape and rate parameters of the Gamma prior on the inverse diagonal elements of each $\boldsymbol{\Sigma}_k$.

(iv) `options=`: A dictionary of additional method-specific options for the algorithms. In the case of EM, this should include

- `"tolerance"`: a convergence threshold, ε, to be used when EM is specified as the algorithm, to determine when the algorithm terminates. At each iteration (after the M-step), the algorithm should compute the normalized log likelihood, which is just the log likelihood $\log p(\mathbf{X} \mid \hat{\boldsymbol{\pi}}^{(m)}, \hat{\boldsymbol{\theta}}^{(m)})$ divided by the sample size N (where the likelihood is evaluated based on density in the standardized feature space), and should terminate when the normalized log likelihood increases by less than ε.

In the case of Gibbs sampling, this should include

- `"iterations"`: total iterations
- `"outputfile"`: the path to a file to write samples to

The return value should also be a dictionary, whose structure depends on the method used.

If `method="kmeans"`, the function can just return the results of the `kmeans()` function.

If `method="EM"`, the dictionary should contain the following entries:

(i) `"params"`: a dictionary containing the final parameter estimates, with entries, `"pi"`, `"mu"`, and `"Sigma"`, which are a one-dimensional array of length K, a $K \times D$ array whose rows are the cluster means, and a $K \times D \times D$ array of cluster covariances, respectively

(ii) `"Q"` The $N \times K$ array of "soft" clustering assignments based on the final parameter estimates (essentially, do one extra "E"-step after convergence and return the $\mathbf{Q}$ matrix)

(iii) `"logLik"`: An array of normalized log likelihoods (i.e., overall log likelihood divided by sample size); one entry per iteration (for diagnostic purposes)

(iv) `"lwr"`: An array whose entries consist of the Jensen's inequality lower bound on the log likelihood computed at each iteration (also for diagnostic purposes)

If `method="Gibbs"`, the dictionary should contain

(i) `"params"`: a dictionary containing the samples for each parameter, with entries `"pi"`, `"mu"`, `"Sigma"`, and `"z"`. The `"pi"` entry should be an $M \times K$

array (where each row is an iteration), `"mu"` should be $M \times K \times D$, `"Sigma"` should be $M \times K \times D \times D$, and `"z"` should be $M \times N$.

(ii) `"Q"` An $M \times N \times K$ array of "soft" clustering assignments at each iteration, corresponding to the parameters sampled at each iteration. (You will have used these probabilities to sample $\mathbf{z}$ at each iteration)

2. Diagnostics, Debugging, and Model Selection

(a) Test the EM case of your clustering function on the iris data for a few small values of K (say between 2 and 5), plotting the log likelihood and lower bound over iterations, as well as plotting the data (two features at a time), color-coded by the most likely cluster (if you're feeling fancy you could try to use a blended color scheme that takes soft clustering into account). Comment on what effect the choice of K has on the final log likelihood.

(b) Implement J-fold cross-validation to select the best value of K for the EM case, using normalized log likelihood as the error metric. Note: You should be able to call your code from HW1, passing `clusterNormalMix` with `method="EM"` as the training function, and a function calculating the normalized log likelihood as the "error" metric. Your code should plot the normalized training set and validation set log likelihoods (normalize by sample size so that training and validation likelihood are on the same scale), for each K.

(c) Does your cross-validation procedure select a value of K which is close to the true number of iris species in the data (i.e., $K = 3$)?

(d) To assess convergence of your Gibbs sampler, plot the sampled values of a few coordinates of the individual means, as well as the determinants of the covariance matrices, over iterations.

(e) Using only the iterations following an assessment that the Gibbs chain has more or less converged, estimate the log of the posterior predictive density of a validation set, by averaging over the likelihoods computed for each sampled parameter

set. I.e., estimate

$$\begin{aligned}
\log p(\mathbf{X}_{\text{validation}} \mid \mathbf{X}_{\text{train}}, K) &= \log \int \int p(\mathbf{X}_{\text{validation}} \mid \boldsymbol{\pi}, \boldsymbol{\theta}) p(\boldsymbol{\pi}, \boldsymbol{\theta} \mid \mathbf{X}_{\text{train}}) \; d\boldsymbol{\pi} d\boldsymbol{\theta} \\
&= \log \mathbb{E}_{p(\boldsymbol{\pi}, \boldsymbol{\theta} \mid \mathbf{X}_{\text{train}})} \left[p(\mathbf{X}_{\text{validation}} \mid \boldsymbol{\pi}, \boldsymbol{\theta}) \right] \\
&\approx \log \left\{ \frac{1}{M - M_{min}} \sum_{m=M_{min}+1}^{M} p(\mathbf{X}_{\text{validation}} \mid \boldsymbol{\pi}^{(m)}, \boldsymbol{\theta}^{(m)}) \right\} \\
&= \log \left\{ \frac{1}{M - M_{min}} \sum_{m=M_{min}+1}^{M} \prod_{n=1}^{N_{\text{validation}}} p(\mathbf{x}_{\text{validation},n} \mid \boldsymbol{\pi}^{(m)}, \boldsymbol{\theta}^{(m)}) \right\}
\end{aligned}$$

where M is the total number of iterations and M_{min} is chosen to be an iteration that is large enough that the sampler can be said to have converged by that point.

For numerical stability, you will want to make use of the following identities for computing the log sums of small probabilities (which will likely produce numerical underflow if calculated naively):

$$\begin{aligned}
\log(\sum_{m=1}^{M} \prod_{n=1}^{N} p_{mn}) &= \log(\sum_{m=1}^{M} \exp\{\sum_{n=1}^{N} \log(p_{mn})\}) \\
&= \log(\sum_{m=1}^{M} \exp\{C + \sum_{n=1}^{N} \log(p_{mn})\} \exp\{-C\}\}) \\
&= -C + \log(\sum_{m=1}^{M} \exp(C + \sum_{n=1}^{N} \log(p_{mn})))
\end{aligned}$$

where C can be any number but is chosen to make the terms being exponentiated not too far from zero (for example, since log probabilities are negative, we might choose $C = \min_m \sum_{n=1}^{N} \log(p_{mn})$).

3. Applying clustering to cancer microarray data

This problem should not require any new implementation; just applying what you did in the previous problem to a more interesting dataset.

The file `nci60_reduced.csv` comes from a dataset in which cell lines from 64 cancerous tumors were analyzed using a "microarray", which measures the degree to which particular genes are expressed in the sample. In the original data, 6830 gene expression measurements were taken from every cell line. However, we do not want to cluster data with this many dimensions using the simple techniques we have learned; so I have preprocessed the data using Principle Components Analysis (PCA) to reduce the number of dimensions to 4.

(a) Using a mixture of Normals, estimating parameters using EM, and using 10-fold cross-validation, find a suitable number of clusters for this data, and plot the first two dimensions of the data, labeling by highest probabilty cluster for your final choice of K.

(b) The diagnosed cancer types are listed in the file `nci60_labels.csv`. Investigate the extent to which your clusters line up with the human-provided catgories. (There is no one correct way to measure this; be creative)

4. A Text Autocorrect Model

This exercise is based on Example 23.5 in BRML.

Consider using an HMM as a model of typed sequences, in which the intended key is usually pressed, but with some probability a neighboring key is pressed instead.

We will consider sequences consisting exclusively of lower case letters and spaces, so there are 27 total states and 27 total observable symbols. For simplicity, encode symbols as integers: $a = 0, b = 1, \ldots, z = 25, <\text{space}> = 26$.

The "neighboring key" model defines an emission matrix, $\mathbf{B}$, where $\mathbf{B}_{kj} = p(x_t = j \mid z_t = k)$, where $\mathbf{B}_{kj}$ is large when $j = k$, moderate when the keys indexed by j and k are nearby on the (QWERTY layout) keyboard, and small when they are distant.

The transition matrix, $\mathbf{A}$ has entries $\mathbf{A}_{k'k} = p(z_t = k \mid z_{t-1} = k')$, and is constructed using the statistics of English words.

Heatmap images of these two matrices are shown in Figs. 1 and 2, respectively, where white is 1.0 and black is 0.0.

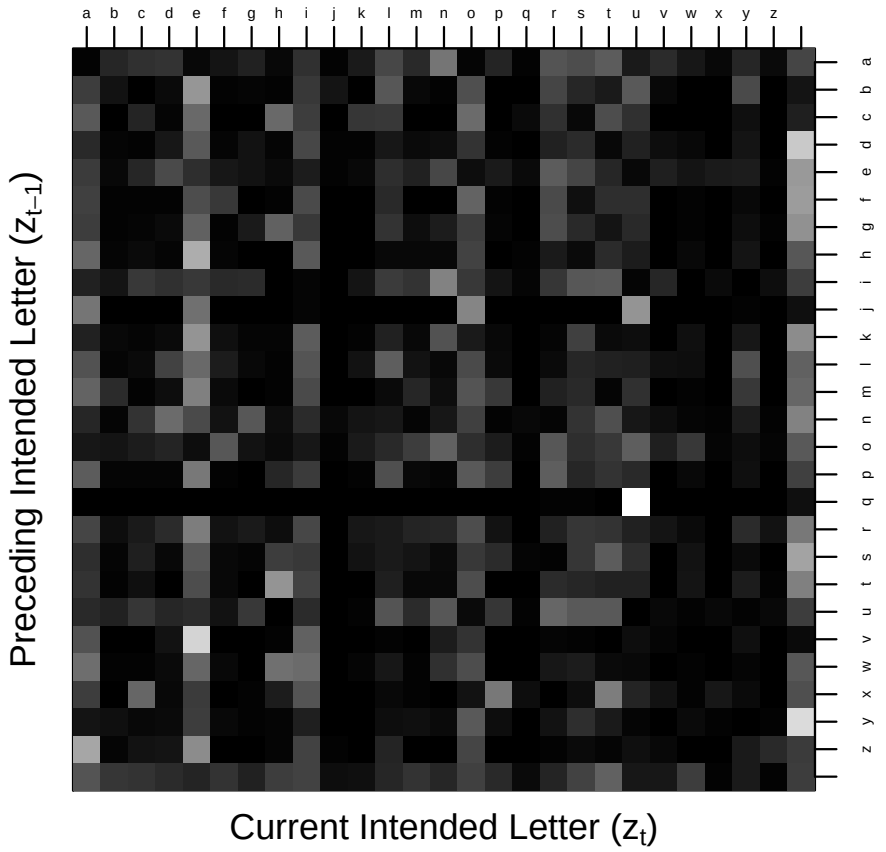

FIGURE 1. Transition matrix $\mathbf{A}$ for the typo HMM model. Color represents $p(z_t \mid z_{t-1})$ on a grayscale with black as 0 and white as 1

The transition and emission matrices, $\mathbf{A}$ and $\mathbf{B}$, are available in the files

- `typing_transition_matrix.csv` and
- `typing_emission_matrix.csv`

Assume for simplicity that the first letter intended is uniformly chosen from the 26 non-space keys.

(a) Implement a forward-filtering and backward-sampling algorithm to obtain samples from the posterior distribution of the latent state sequence $\mathbf{z}$ given the observed sequence $\mathbf{x}$.

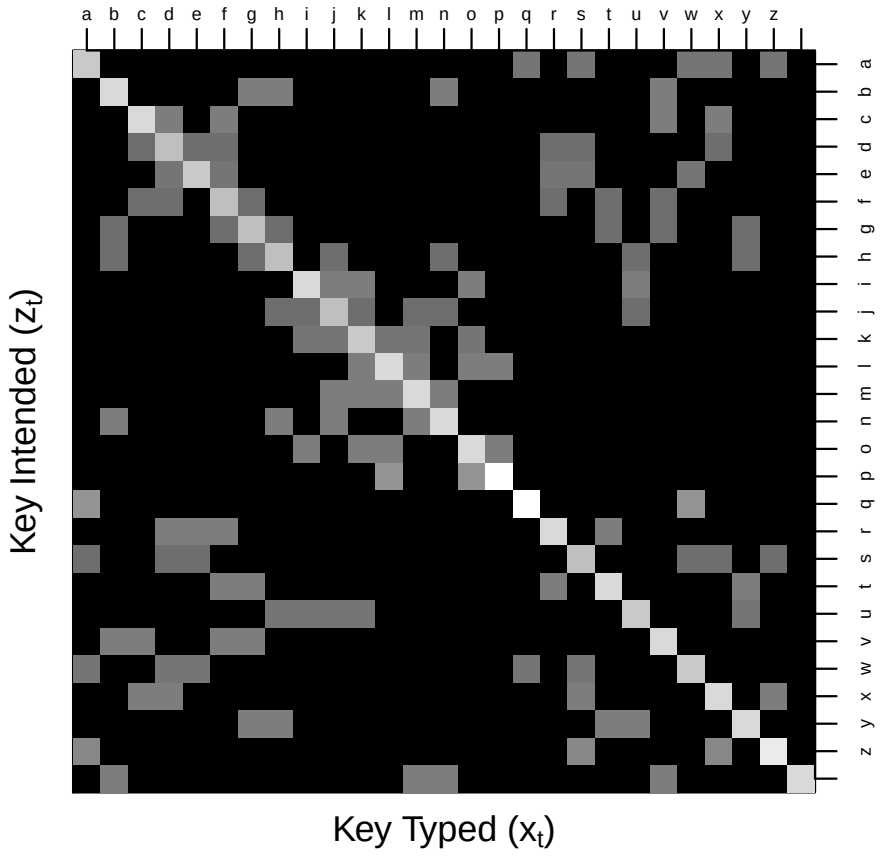

FIGURE 2. Emission matrix $\mathbf{B}$ for the typo HMM model. . Color represents $p(x_t \mid z_t)$ on a grayscale with black as 0 and white as 1

Recall (or anticipate, if you are reading this before we go over this in class) that forward filtering iteratively computes the posterior distribution for z_t given the observations $x_1, \ldots, x_{t-1}$ (which we abbreviate as $\mathbf{x}_{1:t-1}$). This is obtained using the recursion relation:

$$p(z_t = k, \mathbf{x}_{1:t}) = \sum_{k'=1}^{K} p(z_{t-1} = k', \mathbf{x}_{1:t-1}) p(z_t = k \mid z_{t-1} = k') p(x_t \mid z_t = k)$$

If $\mathbf{m}_t$ is defined as the forward "message" vector whose kth entry is $p(z_t = k \mid \mathbf{x}_{1:t})$, and $\mathbf{A}$ and $\mathbf{B}$ are the transition and emission distributions as described above, then we have

$$\mathbf{m}_t = \mathbf{A}^\top \mathbf{m}_{t-1} \odot \mathbf{B}_{\cdot, x_t}$$

where $\odot$ is the elementwise product, and $\mathbf{B}_{\cdot,j}$ is the jth column of $\mathbf{B}$.

The initial message vector $\mathbf{m}_1$ is obtained by multiplying each initial probability $p(z_1 = k)$ by the likelihood $p(x_1 \mid z_1 = k)$.

Note that although the equations above yield joint probabilities, the probabilities obtained will be getting exponentially smaller as we accumulate terms that we are multiplying together. For numerical stability (to avoid underflow), it is a good idea to rescale $\mathbf{m}_t$ by a constant at each iteration.

Since we need to normalize eventually to obtain distributions over just the z_t values so we can sample them, this rescaling does not affect the posterior probabilities.

Once we have computed $\mathbf{m}_t$ for each $t = 1, \ldots, T$, then we can sample a sequence iteratively going backwards from $T, \ldots, 1$. First sample z_T from the distribution obtained by normalizing $\mathbf{m}_T$, and then, conditioning on what has already been sampled, compute

$$\begin{aligned} p(z_t = k, z_{t+1} = k, \mathbf{x}_{1:T}) &= p(z_t = k, x_{1:t}) p(z_{t+1} = k' \mid z_t = k) p(\mathbf{x}_{t+1:T} \mid z_{t+1} = k') \\ &\propto m_{t,k} \mathbf{A}_{kk'} \end{aligned}$$

where the last term in the first line can be dropped since it is constant with respect to z_t.

As a vector, the distribution we are sampling from is proportional to

$$\mathbf{m}_t \odot \mathbf{A}_{\cdot k}$$

(where again, $\mathbf{A}_{\cdot,k}$ represents the kth column of $\mathbf{A}$.

(b) Apply your algorithm to sample a few thousand possible intended sequences given the observed sequence `kezrninh`.

You will likely find it convenient to precompute a matrix of likelihoods:

Let $b^*_{tk} = p(x_t \mid z_t = k)$, since the x_t are fixed.

(c) Many of the possible sequences will consist of letter combinations that do not form real English words. The file `brit-a-z.txt` is a dictionary of British English (this was obtained from Barber, who works at a British university).

I have provided a Python module, `text_utils.py` (in the same location as the data) that defines four functions:

(i) `dict_from_file()`: Creates a Python dictionary object out of a text file with one entry per line, where the keys are the words and the values are simply 1s.

Note that all entries are lowercase; entries with capital letters will not be found.

(ii) `decode_int_list_to_string()`: Converts a list of integers, each between 0 and 26 and converts `0` to `a`, `1` to `b`, etc., and `26` to a space, and returns the corresponding string.

(iii) `encode_string_to_int_list()`: Does the opposite

(iv) `check_validity()`: Takes a string and a dictionary as returned by the first function, and returns `True` or `False` according to whether all space separated substrings appear in the dictionary.

Use this code to discard sampled sequences that contain unknown or nonsense words. Note that if your code returns integer sequences, you will need to convert these into strings first.

(d) Show (by running your code) that the most likely intended sequence for the observed sequence `kezrninh`, as measured by the sequence most often sampled by your algorithm that passes the validity check, is `learning`.