6.867 Machine learning and neural networks

Problem set 4

Due: November 28th, in recitation

What and how to turn in?

Turn in short written answers to the questions explicitly stated, and when requested to explain or prove. Do **not** turn in answers when requested to "think", "consider", "try" or "experiment" (except when specifically instructed).

Turn in all MATLAB code explicitly requested, or that you used to calculate requested values. It should be clear exactly what command was used to get the answer to each question.

To help the graders (including yourself...), please be neat, answer the questions briefly, and in the order they are stated. **Staple each "Problem" separately**, and be sure to write your name on the top of every page.

Density Estimation using Mixtures of Gaussians

In this problem, we will fit mixture of Gaussians models to traced handwritten digits using the EM algorithm. Most of the code and utilities that you will need will be provided.

`EM.m` fits a mixture of k Gaussian distributions to the training data. The simplest way to call this function is `[params,loglikes] = EM(X,k)`, where `params` contains the estimated parameters (mixing proportions and the means and covariances of each of the component Gaussian); `loglikes` stores the log-data likelihoods of the mixture models we produce in the course of the EM iterations. The following functions are called inside `EM.m`:

`Estep` Performs the *expectation* step: for each data point, we calculate the posterior probability that it was drawn from each of the component Gaussians.

`Mstep` Performs the *maximization* step: finds the maximum likelihood settings of the parameters assuming we already have the posterior probabilities computed in the E-step.

`mixtureLL` Calculates the log-likelihood of the data given the current setting of the mixture model parameters (component Gaussians as well as the mixing proportions).

1. Complete the function `Mstep.m` in terms of updating the means and the covariances of the Gaussian components (use the skeleton we have provided).

Our implementation of the E-step (`EstepX.m`) deviates slightly from a "straight-forward" implementation. Modifications of this type are necessary in practice to avoid numerical problems.

2. By looking through `EstepX.m`, `mixtureLLX.m` and `evalmixtureX.m` can you suggest what the problem is that these routines are trying to avoid?

In the following questions, we study a dataset of handwritten digits given in the file `pendigits.mat` (type `load pendigits` to load it into `MATLAB`). The digits were drawn with a stylus input device. The raw trajectories of the digits can be found in the files `pendigits-orig.(tra|tes|names)`. The trajectories were then scaled to normalize them, and eight equi-distant points along the trajectory were extracted from each digit sample. We will use versions of the resulting 16-dimensional dataset (two dimensions for each of the eight points). You can find more information in the file `pendigits.names`.

The provided routine `showdigit.m` can be used to plot the digits:

```
>> showdigit(pentrain8(42,:))
>> showdigit(pentrain4(42,:))
```

The green circle is the beginning of the trajectory. Each row of the data matrix `pentrain4` contains only four points, the means of every two consecutive points.

In order to visualize our estimation results, we will simplify the problem a bit by considering only the second point in `pentrain4`, given by `pentrain4(:,3:4)`. Moreover, we will restrict ourselves to just digits three, four, and five. The resutling 2219×2 matrix `second345` contains the relevant points.

3. Use `EM.m` to fit the distribution of points in `second345` as a mixture of six Gaussian distributions. Plot the resulting distribution, and the resulting components, using the routine `plotmixture2D.m` that we have provided. You can call `plotmixture2D.m`, e.g., as follows:

   ```
   >> plotmixture2D(second345,'evalmixtureX',params)
   ```

   where `params` specify the 6-component mixture model. The blue lines in the contour plot correspond to each of the component Gaussians in the mixture and the red lines provide the contours of the 6-component mixture model. At the center of each Gaussian component, you also see the numerical value of the mixing proportion allocated to that component.

   Turn in the contour plot (no need to turn in the 3D plot).

The resulting log-likelihood should be a bit over 1000, and EM should converge after a few hundred iterations.

4. Rerun EM on the same data-set a few times, looking at the resulting plots and the log-likelihoods. Explain why you do not always get close to the same distribution, and why you sometimes do get the exact same distribution (and not just a simmilar one).

Plot the log-likelihood at each EM iteration (this is returned as the second output value of `EM.m`). Notice (sometimes) the step-like behavior of the log-likelihood: the log-likelihood is almost unchanged throughout many iterations, and then increases dramatically after a few additional iterations. You can use the history of mixture parameters (returned as the third output value of `EM.m`) to see how the mixture distribution itself changes in relation to the log-likelihood. It takes time to resolve how to best allocate the mixture components, especially if (some of) the Gaussian components are only slightly different from each other in the current model.

Initialization

The EM algorithm describes how to *update* the mixture parameters in order to improve the log-likelihood of the data. This search process must start somewhere. An optional parameter to `EM.m` can be used to control the way the mixture parameters are initialized.

5. The provided routine `initSame.m` sets all mixture components to have zero mean, unit (spherical) covariances, and equal mixing proportions. Try using it to fit `second345`. How many iterations does it take for EM to converge? Why? Describe what happens at each iteration (you might want to set the `histfreq` parameter to one, and plot the mixture at each iteration).

6. What would happen if the means and covariances were all equal, but the initial mixing proportions were chosen randomly? If the means and mixing proportions were equal but the initial covariance matrices were chosen randomly?

The default initialization routine used by `EM.m` is `initRandPoints.m`, which initializes the means to random data points, and sets the covariance matrices to diagonal matrices with the same overall variance parameter computed from the overall covariance of the data.

Data anomalies and regularization

The actual data was given as integers in the range 0..100 in each coordinate. The matrix `pentrain8`, and its derivatives `pentrain4` and `second345` represent the original data scaled to [0,1] with uniform noise added.

The matrix `cleanpentrain8` contains the 0..100 integer valued data.

7. Try estimating a 5-component mixture model to the first point `cleanpentrain8(:,1:2)`. You can also try the rest of the time points `cleanpentrain8(:,2t-1:2t)`. Why does EM fail? What happens ? Hint: plot the distribution using

   ```
   >> plot(cleanpentrain8(:,1),cleanpentrain8(:,2),'.')
   ```

8. Would we be able to resolve the above problem by introducing regularization to the M-step? Regularization would modify how we update the covariance matrix of each of the component Gaussians in the M-step. Let $\hat{n}_i$ is the number of "soft" counts assigned to component i in the E-step and $\hat{\Sigma}_i$ the corresponding (unregularized) estimate of the covariance matrix. We can now imagine having an additional set of n' points with covariance S (the prior covariance matrix) also assigned to component i. These additional points would change the covariance update according to

   $$\hat{\Sigma}'_i = (\frac{\hat{n}_i}{\hat{n}_i + n'})\,\hat{\Sigma}_i + (\frac{n'}{\hat{n}_i + n'})\,S \tag{1}$$

   where $\hat{\Sigma}'_i$ is the new regularized estimate of the covariance matrix for component i. (We do not expect you to try this out but answer the question based on what you think would happen).

Unsupervised Learning Using Mixture Models

We continue to use the trajectories in an unsupervised manner, i.e., not paying attention to the accompanying labels. We can use the mixture models to try to partition the set of input samples (trajectories) into *clusters*. We will also assess how the emerging cluster identities will correlate with the actual labels.

The first question we have to answer is how many Gaussian components we should use in the mixture model. How could we decide the issue? We could proceed similarly to the case of setting the kernel width parameter in non-parametric density estimation, i.e., via cross-validation. The problem here, however, is that estimating any single mixture model with say 5 components already takes a fair amount of computing time and repeating this process n times (n is the number of training examples) does not appear very attractive. Instead, we will use a simple approximate criterion or *Bayesian information criterion* to find the appropriate number of clusters. This involves adding a complexity penalty to the log-data likelihoods. In other words, if $LL(k)$ is the log-likelihood that our k-component mixture model assigns to the training data after training, we find k that maximizes

$$score(k) = LL(k) - \frac{1}{2} D_k \log(n) \tag{2}$$

where D_k denotes the number of parameters in the k-component mixture model. For a Gaussian mixture mode, $D_k = k * (d + d * (d+1)/2) + k - 1$, where d is the dimensionality of the examples (e.g., 2). Note that the penalty is a function of the number of parameters and the number of training examples n.

9. Use different numbers of mixture components with `second345` and select the "optimum" number of components. Use `mixtureModSel.m` to plot the scores for $k = 1, \ldots, 8$. Return the plot. Your choice of k may change if you re-estimate the mixture models (make sure you understand why).

Note that there may be more clusters than than there are different types of digits. Some of the digits may have sub-types or our assumption that the clusters look like Gaussians may be invalid and we need more than one Gaussian to account for any single real cluster in the data.

Now, given the "best" mixture model, we would like to correlate the cluster identities with the available labels. We can do this to find out how well we could capture relevant structure in the data without actually knowing anything about the labels.

10. The labels are provided in `labels345`. Write a simple matlab routine to get hard assignments of training examples to clusters. You'll probably want to use `mixtureScaledPiX` for this purpose. Use `correlate.m` to correlate the cluster identities with the available labels `labels345`. Return your MATLAB code and the correlation matrix. Briefly discuss whether the approach was successful, i.e., whether we could associate the clusters with specific labels.

Using Mixture Models for Classification

We will now try to use the Gaussian mixture models in a classification setting. We can estimate class-conditional density models (mixture models) separately for each class and classify any new test example on the basis of which class-conditional model assigns the highest likelihood to the example.

11. Using the provided `fitAllDigits.m`, fit a 3-component Gaussian mixture model to each of the digits in `pentrain8,trainlabels`.

12. Complete the routine `classify.m` that takes as input the examples to be classified and a cell-array of mixture parameters (as returned by `fitAllDigits.m`) to return the maximum likelihood assignment of labels to the examples. Provide the code for `classify.m`.

13. Classify the `pentrain8` and `pentest8`, and use `correlate.m` to compare the results to the true labelings in `trainlabels` and `testlabels`. Turn in the two correlation matrices.

For comparison, you probably want to repeat the above for 1-component Gaussian mixtures. You could also use the model selection routine we have provided to find a different number of mixture components for each class-conditional density. Increasing the number of components per class would eventually force you to use regularized estimates (see above).

Markov and Hidden Markov Models

In this problem, we will use Markov and Hidden Markov models to identify the language of written sentences. For simplicity our representation of text will include only 27 symbols—the 26 letters of the Latin alphabet, and the space symbol. Any accented letter is represented as a non-accented letter, none-Latin letters are converted to their closest Latin letters, and punctuation is removed. This representation naturally looses quite a bit of information compared to the original ASCII text. This 'handicap' is in part intentional so that the classification task would be a bit more challenging.

Most of the MATLAB code you will need here will be given. You will find the following routines useful (here and perhaps in some of your projects as well):

`readlines.m` Reads a named text file, returning a cell array of the lines in the file. To get line `i` of cell-array `lines` returned from, e.g., `lines = readlines('cnn.eng')`, use `lines`$\{$`i`$\}$.

`text2stream.m` Converts a string (a line of text) into a row vector of numbers in the range $\{1, \ldots, 27\}$, according to the representation discussed above. So, for example, `numberline = text2stream(lines1)` would convert the first line of text from `lines` into a row vector of numbers. The conversion of the full output of `readlines` would have to be done line by line.

`count.m` Given text in a row vector representation and a width k, the function computes the count of all k-grams in the array. In other words, the function returns a k-dimensional array representing the number of times each configuration of k successive letters occurs in the text.

`totalcount.m` This function allows you to compute the accumulated counts from each of the lines of text returned by `readlines`. Use this function to find the training counts for the different languages.

Language identification using Markov models

Here we will construct a language classifier by using Markov models as class-conditional distributions. In other words, we will separately train a Markov model to represent each of the chosen languages: English, Spanish, Italian and German. The training data is given in the files `cnn.eng, cnn.spa, cnn.ita, cnn.ger`, which contain several news articles (same articles in different languages), one article per line.

We will first try a simple independent (zeroth-order Markov) model. Under this model, each successive symbol in text is chosen independently of other symbols. The language is in this case identified based only on its letter frequencies.

1. Write a function `naiveLL(stream,count1)` which takes a 1-count (frequency of letters returned by `count.m`) and evaluates the log-likelihood of the text stream (row vector of numbers) under the independent (zeroth-order Markov) model.

Extract the total 1-counts from the language training sets described above. Before proceeding, let's quickly check your function `naiveLL`. If you evaluate the log-likelihood of `'This is an example sentence'` using the English 1-counts from `cnn.eng`, you'll get -76.5690, while the Spanish log-likelihood of the same sentence is -77.2706.

2. Write a short function `naiveC` which takes a stream, and several 1-counts corresponding to different languages, and finds the maximum-likelihood language for the stream. You could assume, e.g., that the 1-counts are stored in an array, where each column corresponds to a specific language. The format of the labels should be in correspondence with the `test_labels` described below.

The files `song.eng, song.spa, song.ita, song.ger` contain additional text in the four languages. We will use these as the test set. For your convenience, we have provided you with script `generate_test.m` :

```
test_sentences = [ readlines('song.eng') ; ...
   readlines('song.ger') ; ...
   readlines('song.spa') ; ...
   readlines('song.ita') ] ;
test_labels = [ ones(17,1) ; ones(17,1)*2 ; ones(17,1)*3 ; ones(17,1)*4 ]
```

In order to study the performance of the classifier as a function of the length of test strings, we will classify all prefixes of the lines in the test files. The provided routine `testC.m` calculates the success probability of the classification, for each prefix length, over all the streams or strings in a given cell-array. You can call this function, e.g., as follows

```
successprobs = testC(test_sentences,test_labels,'naiveC',count1s)}
```

where `count1s` provides the array of training counts that your function `naiveC` should accept as an input.

3. Plot the success probability as a function of the length of the string. What is the approximate number of symbols that we need to correctly assign new piece of text to one of the four languages?

In order to incorporate second order statistics, we will now move on to modeling the languages with first-order Markov models.

4. Write a function `markovLL(stream,count2)` which returns the log-likelihood of a stream under a first-order Markov model of the language with the specified 2-count. For the initial probabilities, you can use 1-counts calculated from the 2-counts.

Quick check: The English log-likelihood of `'This is an example sentence'` is -60.6718, while its Spanish log-likelihood is -63.2288. We are again assuming that you are using the training sets described above to extract the 2-counts for the different languages.

5. Write a corresponding function `markovC.m` that classifies a stream based on Markov models for various languages, specified by their 2-counts.

6. Try to classify the sentence `'Why is this an abnormal English sentence'`. What is its likelihood under a Markov model for each of the languages ? Which language does it get classified as ? Why does it not get classified as English ?

When estimating discrete probability models, we often need to regularize the parameter estimates to avoid concluding that any configuration (e.g., two successive characters) have probability zero unless such the configuration appeared in the limited training set. Let $\{\theta_i\}$, where $\sum_{i=1}^m \theta_i = 1$, define a discrete probability distribution over m elements $i \in \{1, \ldots, m\}$. For the purposes of estimation, we treat θ_i as parameters. Given now a training set summarized in terms of the number of occurrences of each element i, i.e., $\hat{n}_i$, the maximum likelihood estimate of $\{\theta_i\}$ would be

$$\hat{\theta}_i = \frac{\hat{n}_i}{\sum_{j=1}^m \hat{n}_j} \tag{3}$$

This is zero for all elements i that did not occur in the training set, i.e., when $\hat{n}_i = 0$. To avoid this problem, we introduce a prior distribution over the parameters $\{\theta_i\}$:

$$P(\theta) = \frac{1}{Z} \prod_{i=1}^m \theta_i^{\alpha_i} \tag{4}$$

where Z is a normalization constant and $\alpha_i \geq 0$'s are known as *pseudo-counts.* This prior, known as a *Dirichlet* distribution, assigns the highest probability to

$$\tilde{\theta}_i = \frac{\alpha_i}{\sum_{j=1}^m \alpha_j} \tag{5}$$

Thus α_i's behave as if they were additional counts from some prior (imaginary) training set.

We can combine the prior with the maximum likelihood criterion by maximizing instead the penalized log-likelihood of the data (expressed here in terms of the training set counts

$\hat{n}_i$):

$$
\begin{aligned}
J(\theta) &= \overbrace{\sum_{i=1}^{m} \hat{n}_i \log \theta_i}^{\text{log-probability of data}} + \overbrace{\log P(\theta)}^{\text{log-prior}} && (6)\\
&= \sum_{i=1}^{m} \hat{n}_i \log \theta_i + \sum_{i=1}^{m} \alpha_i \log \theta_i + \text{constant} && (7)\\
&= \sum_{i=1}^{m} (\hat{n}_i + \alpha_i) \log \theta_i + \text{constant} && (8)
\end{aligned}
$$

The maximizing $\{\theta_i\}$ is now

$$\hat{\theta}_i = \frac{\hat{n}_i + \alpha_i}{\sum_{j=1}^{m} (\hat{n}_j + \alpha_j)} \tag{9}$$

which will be non-zero whenever $\alpha_i > 0$ for all $i = 1, \ldots, m$. Setting $\alpha_i = 1/m$ would correspond to having a single prior observation distributed uniformly among the possible elements $i \in \{1, \ldots, m\}$. Setting $\alpha_i = 1$, on the other hand, would mean that we had m prior observations, observing each element i exactly once.

7. Add pseudocounts (one for each configuration) and reclassify the test sentence. What are the likelihoods now. Which language does the sentence get classified as ?

8. Use `testC.m` to test the performance of Markov-based classification (with the corrected counts) on the test set. Plot the correct classification probability as a function of the text length. Compare the classification performance to that of `naiveC.m`. (Turn in both plots).

Hidden Markov Models

We will now turn to a slightly more interesting problem of language segmentation: given a mixed-language text, we would like to identify the segments written in different languages.

Consider the following simple approach: we will first find the character statistics for each language by computing the 1-counts from the available training sets as before. We will then go through the new text character by character, classifying each character to the most likely language (language whose independent model assigns the highest likelihood to the character). In other words, we would use `naiveC` to classify each character in the new text. To incorporate higher-order statistics, we could train a Markov model for each language (as before), and again assign the characters in the text to the most likely language.

9. Why would we expect the resulting segmentation not to agree with the true segmentation? What would the resulting segmentation look like? What is the critical piece of information we are not using in this approach ?

We would rather model the multi-lingual text with a hidden Markov model. For simplicity, we will focus on segmenting text written in only two languages.

10. Suggest how a hidden Markov model can solve the problem you identified above. Provide an annotated transition diagram (heavy lines for larger transition probabilties) and desribe what the output probabilities should be capturing.

For some setting of the parameters, this HMM probably degenerates to the independent model. Make sure you understand when this happens.

Now, load the example text from the file `segment.dat`. The text, mixed German and Spanish, is given in the variable `gerspa` as numbers in the range 1..27 (as before). You can use the provided function `stream2text.m` to view the numbers as letters.

The routine `[hmm,ll] = esthmm(data,hmm)` uses EM to find the maximum likelihood parameters of a hidden Markov model, for a given output sequence(s). As discussed earlier, the EM algorithm has to start its search with some parameter setting. The `hmm` input argument to `esthmm` provides this initial parameter setting.

The routine `hmm = newhmm(numstates,numout)` creates random HMM parameters for an HMM with `numstates` hidden states, and `numout` output symbols. An optional third argument controls the "non-uniformity" of the parameters.

Note: The orientation of the matrices `Po` and `Pt` is different from those in recitation: `Pt(i,j)`$= P(state_t = j|state_{t-1} = i)$ and `Po(i,a)`$= P(O_t = a|state_t = i)$.

With two different initializations, you will probably find two different HMMs. It is a good idea to run the EM algorithm multiple times, starting from slightly different initial settings. Every such run can potentially lead to a different result. Make sure you understand how you would choose among the alternative solutions.

11. Estimate a two-state hidden Markov model with two different types of initial settings of the state transition probabilities. First, set the transition probabilities close to uniform values. In the second approach, make the "self-transitions" quite a bit more likely than the transitions to different states. Which initialization leads to a better model?

Now try to segment `gerspa` into German and Spanish, using a two-state hidden Markov model. You can then use the routine `viterbi(sequence,hmm)` to examine the most likely (maximum a posteriori probability or MAP) state sequence. The correct segmentation of `gerspa` is given in `gerspa_lang`.

12. Examine the difference between your segmentation and the correct one. Do the errors make sense?

13. Use the provided routine `hmmposteriors(sequence,hmm)` to calculate the per character posterior probabilities over the states. These are the $\gamma_t(i)$ probabilities described in lectures (t gives the character position in text and i specifies the state). Plot these probabilities as a function of the character position in the text sequence and turn in the plot. You might want to re-scale the axis using `axis[0 3000 0 1.1]`.

14. Find the sequence of maximum a posteriori probability states. That is, for each time point t, find the state i with the maximum a posteriori probability $P(state_t = i|observedseq)$. Compare this state sequence with the maximum a posteriori state sequence. Are they different?

15. Give an example of a hidden Markov model, with two hidden states and two output symbols, and a length two output sequence, such that the MAP state sequence differs from the sequence of MAP states. Be sure to specify all the parameters of the HMM (these need not be the maximum likelihood parameters for this specific length two output sequence).

In the above example, we assumed no knowledge about the languages. We would like to improve the segmentation by incorporating some additional information about the languages such as single-character statistics.

16. What parameters of the HMM still need to be estimated ? Modify the routine `esthmm.m` accordingly. (Turn in the modifications)

17. *(Optional)* Use the single-character statistics you calculated from `cnn.spa` and `cnn.ger` to segment the text. What is the maximum likelihood estimate of the remaining parameters ? Compare the resulting most likely state sequence to the one you found without this prior information. Are they different ?

18. Hidden Markov models can also be useful for classification. Suggest how you would use HMMs in place of the Markov models described above to identify the language of any specific (single-language) document. Specifically, list the routines we have provided that you could make use of and describe which additional routines you would need.

19. *(Optional)* Use hidden Markov models, with a varying number of states, for the language classification task investigated in the first part of this problem. Using the same training set and test set as before, create a graph of the probability of correct classification as a function of the length of the text. Discuss how the results compare to using naive classification and first-order Markov models, and how changing the number of hidden states affects the results.