COMP0088
Introduction to Machine Learning
Lab Assignment 8

Matthew Caldwell

November 27, 2021

Task Summary

Week 8 Notes

In this final lab assignment, you will generate and fit data for a **Gaussian mixture model**, and generate and 'decode' data from a **hidden Markov model**. For the latter, the script will also attempt to fit using an external HMM library. You do not need to do any coding for this, but you will need to install the python `hmmlearn` package. This can be done with `pip`:

```
$ pip install hmmlearn
```

If for some reason you are unable or prefer not to install this, the rest of the script should work fine without it.

The assignment script is `week_8.py`, generating output file `week_8.pdf`. There are some potentially-useful command line options, which you can list using:

```
$ python week_8.py -h
```

Example output from a completed assignment is shown in Fig. 1. Plotting functionality is provided, so your output should look similar.

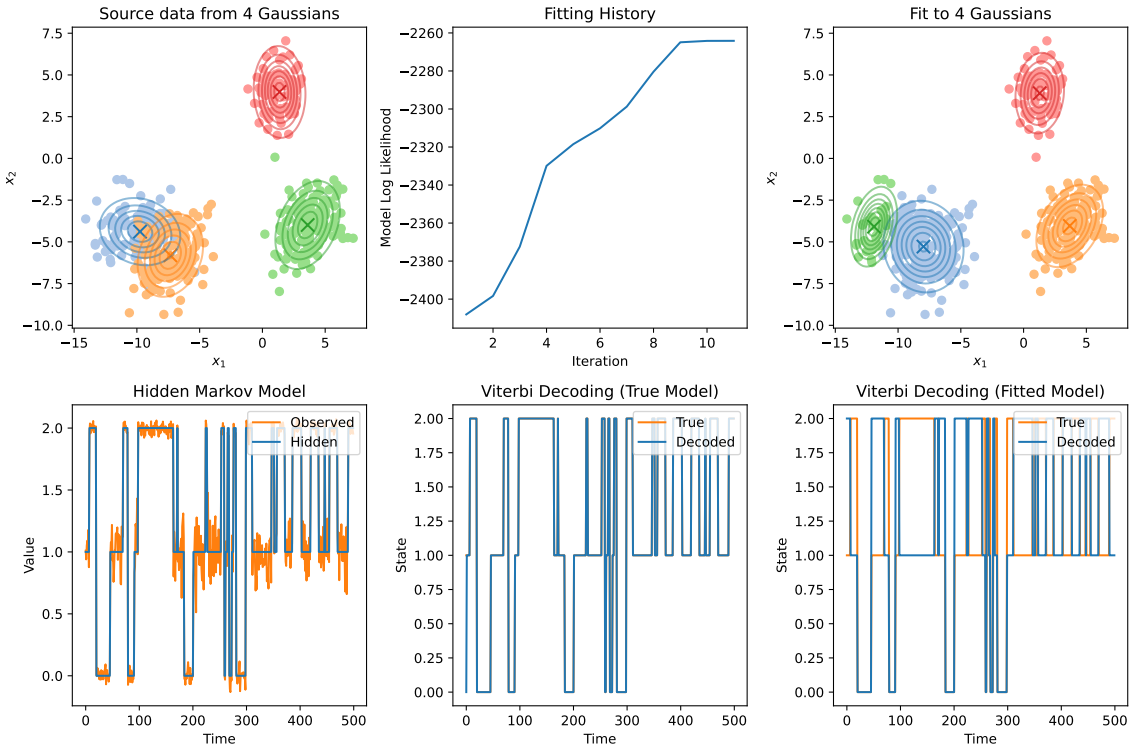

Figure 1: Example output from the `week_8.py` script

1 Generate data from a Gaussian mixture

In a Gaussian mixture model, the data distribution is a weighted combination of several independent multivariate Gaussian distributions, each with its own mean vector and covariance matrix:

$$P(\mathbf{x}) = \sum_{i}^{k} \alpha_i \phi(\mathbf{x}; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \tag{1}$$

Here ϕ represents the standard multivariate Gaussian density for $\mathbf{x} \in \mathbb{R}^d$:

$$\phi(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^d |\boldsymbol{\Sigma}|}} \tag{2}$$

The weights α_i constitute a categorical distribution, representing the probability of drawing from each of the component Gaussians. Hence $\sum_i \alpha_i = 1$ and $\alpha_i \geq 0 \; \forall i$.

1.1 Generate the data

Implement this function in `week_8.py`:

```
def generate_gaussian_mix ( num_samples, means, covs,
                            class_probs, rng ):
    """
    Draw labelled samples from a mixture of multivariate
    gaussians.

    # Arguments
        num_samples: number of samples to generate
            (ie, the number of rows in the returned X
            and the length of the returned y)
        means: a list of vectors specifying mean of each gaussian
            (all the same length == the number of features)
        covs: a list of covariance matrices
            (same length as means, with each matrix being
            num features x num features, symmetric and
            positive semidefinite)
        class_probs: a vector of class probabilities,
            (same length as means, all non-negative and
            summing to 1)
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        X: a matrix of sample inputs, where
            the samples are the rows and the
            columns are features, ie size is:
              num_samples x num_features
        y: a vector of num_samples labels matching
            the samples to the gaussian from which
            they were drawn
    """
    assert(len(means)==len(covs)==len(class_probs))
```

```
    # TODO: implement this
    return None, None
```

The `numpy.random.Generator` class includes the methods `multinomial` for drawing from a multinomial distribution and `multivariate_normal` for drawing from a (single) multivariate Gaussian.

Note that you are asked to return the ground truth labels specifying which samples belong to which Gaussian. Fitting a Gaussian mixture is an unsupervised learning method, but in generating the data you know which distribution is responsible for each sample. The return labels will be used in the first plot of the output figure, to show which samples belong to which of the component Gaussians.

2 Fit a Gaussian mixture model

Gaussian mixture models are fitted by a version of the expectation-maximisation (EM) algorithm, starting from an initial guess and iteratively maximising the likelihood of the model parameters in alternating steps.

In the **E-step**, soft assignments or **responsibilities**, $\gamma_{i,j}$ are estimated for each sample i with respect to each Gaussian component j:

$$\gamma_{i,j} = \frac{\phi(\mathbf{x}_i; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)\alpha_j}{\sum_t \phi(\mathbf{x}_i; \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)\alpha_t} \tag{3}$$

The normalising factor in the denominator ensures that for each sample the responsibilities sum to 1.

In the **M-step**, the model parameters are updated from the data according to the responsibilities:

$$\boldsymbol{\mu}_j = \frac{\sum_i \gamma_{i,j}\mathbf{x}_i}{\sum_i \gamma_{i,j}} \tag{4}$$

$$\boldsymbol{\Sigma}_j = \frac{\sum_i \gamma_{i,j}(\mathbf{x}_i - \boldsymbol{\mu}_j)(\mathbf{x}_i - \boldsymbol{\mu}_j)^\mathsf{T}}{\sum_i \gamma_{i,j}} \tag{5}$$

$$\alpha_j = \frac{\sum_i \gamma_{i,j}}{n} \tag{6}$$

Note that for these purposes the vectors should be considered as column vectors, so the product in the numerator of Eq. (5) is an **outer product** whose result is a $d \times d$ matrix.

The overall model log likelihood can be estimated as the sum of the log probabilities of all the observations given the Gaussian parameters, weighted by the class probabilities:

$$l(\boldsymbol{\theta}) = \sum_i^n \log \sum_j^k \alpha_j \phi(\mathbf{x}_i; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \tag{7}$$

2.1 Calculate the log likelihood of a GMM

Implement the following function in `week_8.py`:

```
def gaussian_mix_loglik ( X, means, covs, class_probs ):
    """
    Estimate the log likelihood of the given mixture model.

    # Arguments
        X: a matrix of sample inputs, where
            the samples are the rows and the
            columns are features, ie size is:
              num_samples x num_features
        means: a list of vectors specifying mean of each gaussian
            (all the same length == the number of features)
        covs: a list of covariance matrices
            (same length as means, with each matrix being
            num features x num features, symmetric and
            positive semidefinite)
        class_probs: a vector of class probabilities,
```

```
            (same length as means, all non-negative and
            summing to 1)

    # Returns
        loglik: the (scalar) log likelihood of the model
    """

    # TODO: implement this
    return None
```

The formula for this calculation is given in Eq. (7). Note that SciPy provides a method for calculating the multivariate Gaussian density as `scipy.stats.multivariate_normal.pdf`.

2.2 Perform the E-step

Implement the following function in `week_8.py`:

```
def gaussian_mix_E_step ( X, means, covs, class_probs ):
    """
    Given a candidate set of gaussian mix parameters,
    estimate the responsiblilites of each component for
    each data sample.

    # Arguments
        X: a matrix of sample inputs, where
            the samples are the rows and the
            columns are features, ie size is:
              num_samples x num_features
        means: a list of vectors specifying mean of each gaussian
            (all the same length == the number of features)
        covs: a list of covariance matrices
            (same length as means, with each matrix being
            num features x num features, symmetric and
            positive semidefinite)
        class_probs: a vector of class probabilities,
            (same length as means, all non-negative and
            summing to 1)

    # Returns
        resps: a matrix of weights attributing
            samples to source gaussians, of size
              num_samples x num_gaussians
    """
    assert(len(means)==len(covs)==len(class_probs))

    # TODO: implement this
    return None
```

The calculation is given in Eq. (3). Again, you will probably want to use the SciPy method `scipy.stats.multivariate_normal.pdf`.

2.3 Perform the M-step

Implement this function in `week_8.py`:

```
def gaussian_mix_M_step ( X, resps ):
    """
    Given a candidate set of responsibilities,
    estimate new gaussian mixture model parameters.

    # Arguments
        X: a matrix of sample inputs, where
            the samples are the rows and the
            columns are features, ie size is:
              num_samples x num_features
        resps: a matrix of weights attributing
            samples to source gaussians, of size
              num_samples x num_gaussians

    # Returns
        means: a list of vectors specifying mean of each gaussian
            (all the same length == the number of features)
        covs: a list of covariance matrices
            (same length as means, with each matrix being
            num features x num features, symmetric and
            positive semidefinite)
        class_probs: a vector of class probabilities,
            (same length as means, all non-negative and
            summing to 1)
    """

    # TODO: implement this
    return None, None, None
```

The update calculations are given in Eqs. (4) to (6).

2.4 Iteratively fit the model

Implement the following function in `week_8.py`:

```
def fit_gaussian_mix ( X, num_gaussians, rng, max_iter=10,
                       loglik_stop=1e-1 ):
    """
    Fit a gaussian mixture model to some data.

    # Arguments
        X: a matrix of sample inputs, where
            the samples are the rows and the
            columns are features, ie size is:
              num_samples x num_features
        num_gaussians: the number of components
            to fit
        rng: an instance of numpy.random.Generator
```

```
            from which to draw random numbers
        max_iter: the maximum number of iterations
            to perform
        loglik_stop: stop iterating once the improvement
            in log likelihood drops below this

    # Returns
        resps: a matrix of weights attributing
            samples to source gaussians, of size
              num_samples x num_gaussians
        means: a list of num_gaussians vectors specifying means
        covs: a list of num_gaussians covariance matrices
        class_probs: a vector of num_gaussian class probabilities
        logliks: a list of the model log likelihood values after
            each fitting iteration
    """

    # TODO: implement this
    return None, None, None, None, None
```

Use the previously defined functions to perform the EM steps and to evaluate the likelihood. You will need to choose an appropriate set of `num_gaussians` initial values for the means, covariances and class probabilities.

3 Generate data from a Hidden Markov model

In a hidden Markov model, the visible output is conditioned on some unobserved latent state that varies over time, with the state at any discrete time step t, z_t, dependent only on the state at the previous time step, z_{t-1}.

For these exercises we will consider only scalar observations, x_t, whose values are normally distributed with mean and variance according to the latent state:

$$(x_t|z_t = j) \sim N(\mu_j, \sigma_j^2) \tag{8}$$

So the parameters $\mu_1, \mu_2, ..., \mu_k$ and $\sigma_1, \sigma_2, ..., \sigma_k$ define the **emission probabilities**.

The transition probabilities are defined by a $k \times k$ **transition matrix** in which the element at row i and column j specifies the probability of going from state i to state j, ie:

$$a_{i,j} = P(z_{t+1} = j|z_t = i) \tag{9}$$

The distribution of starting states of the model is determined by an **initial probability** vector $\boldsymbol{\pi} = [\pi_1, \pi_2, ..., \pi_k]$, where

$$\pi_j = P(z_1 = j) \tag{10}$$

3.1 Generate the data

Implement the following function in `week_8.py`:

```
def generate_hmm_sequence ( num_samples,
                            initial_probs, transitions,
                            emission_means, emission_sds,
                            rng ):
    """
    Generate a sequence of observations from a hidden Markov
    model with the given parameters. Emissions are univariate
    Gaussians.

    # Arguments
        num_samples: number of samples (ie timesteps) to generate
        initial_probs: vector of probabilities of being in each hidden
            state at time step 1; must sum to 1
        transitions: matrix of transition probabilities, from state
            indexed by row to state indexed by column; rows must sum to 1
        emission_means: mean of observations for each hidden state
        emission_sds: standard deviation of observations for each
            hidden state
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        x: a vector of observations for each time step
        z: a vector of hidden state (indices) for each time step
    """
    # TODO: implement this
    return None, None
```

The `numpy.random.Generator` method `choice` is useful for drawing single values from a categorical distribution. (The `multinomial` method models the outcomes of multiple trials, returning counts rather than single values.)

As in Task 1.1, you are asked to return the true hidden state sequence as well as the observations for plotting purposes. Obviously these would not be available for real data.

4 Decode the underlying state sequence using the Viterbi algorithm

The Viterbi algorithm iterates forwards through the sequence, at each timestep t estimating the highest probability of being in each state, given these factors:

- the present observation x_t
- the probability of transition to z_t from each previous state $z_{t-1} = j$
- the previously estimated probability of being in that previous state, $\alpha_{t-1}(j)$

The recurrence relation looks like this:

$$\alpha_t(j) = \max_i \; \alpha_{t-1}(i)\, a_{i,j}\, P(x_t; \mu_j, \sigma_j) \tag{11}$$

For each state at each step it also records which previous state produced that highest probability as a **backpointer**, $\zeta_t(j)$:

$$\zeta_t(j) = \operatorname*{argmax}_i \; \alpha_{t-1}(i)\, a_{i,j}\, P(x_t; \mu_j, \sigma_j) \tag{12}$$

At the end of the sequence, timestep T, the most likely final state is identified:

$$z_T = \operatorname*{argmax}_j \alpha_T(j) \tag{13}$$

and the ζ_t are traced backwards to reconstruct the complete most likely sequence:

$$z_{t-1} = \zeta_t(z_t) \tag{14}$$

4.1 Find the most likely hidden state sequence

Implement the following function in `week_8.py`:

```
def viterbi ( x, initial_probs, transitions, emission_means, emission_sds ):
    """
    Infer the most likely sequence of hidden states based on
    observations and HMM parameters.

    # Arguments
        x: a vector of observations for each time steps
        initial_probs: vector of probabilities of being in each hidden
            state at time step 1; must sum to 1
        transitions: matrix of transition probabilities, from state
            indexed by row to state indexed by column; rows must sum to 1
        emission_means: mean of observations for each hidden state
        emission_sds: standard deviation of observations for each
            hidden state

    # Returns
        z: a vector of predicted hidden state (indices) for each time step
    """
    # TODO: implement this
    return None
```

The SciPy method `scipy.stats.norm.pdf` evaluates the univariate Gaussian density.

Although the calculations as given above are not especially complicated, this can be a bit fiddly to implement. You will need to build a $k \times T$ 'trellis' matrix for both the α and ζ values. As with long chains of gradients in backpropagation, the probability products here can get vanishingly small and make things numerically unstable, so it's probably a good idea to normalise at each timestep.

If you're feeling adventurous, you could also try implementing the Baum-Welch algorithm to actually fit the model parameters—note that this is even more fiddly. If you just want to see the results, install hmmlearn (as described in the introduction) and compare the decoded trace in panel 6 of the output figure to your own reconstruction in panel 5. (Spoiler alert: yours should be better!)