Computer Science 315
Assignment 4

2013

In this assignment, you will implement parameter estimation for Gaussian mixture models (GMMs) using the K-means and expectation-maximization (EM) algorithm. You will then model the wine data set as a GMM, and investigate the use of GMMs to classify diphthongs (vowel-like sounds) — an important task in speech recognition.

Implementation

1. Download and unzip the zipped resource file for this assignment on the course website. Import the unzipped folder into Eclipse, and set up a remote repository on `eniac` for storing a copy of your code for this assignment, and for making your final submission. (Remember to tag the version of your assignment you want to have marked.)

2. Treat the provided code and comments in the resource file as specifications, and implement all the omitted details (i.e. replace all the occurences of the `pass` keyword with appropriate code). You may not change any code in such a way as to limit the way in which various methods may be called, as this may cause automated testing scripts to break. Put another way, any call to a method or function which would be legal with the function and method declarations provided should remain legal in your implementation. For your convenience, HTML documentation generated from the docstrings in the provided code by `epydoc` is also provided in the resource file.

3. You are encouraged to re-use code from your previous assignments. To do so, do not link to previous assignment projects in Eclipse. Rather, copy and paste the relevant source files into the new project, and then alter them there if necessary. Also ensure that these extra source files are also stored in your Git repository.

4. Note that due to concerns of numerical overflow and underflow in calculations, it is advisable to perform as many of your calculations in the log-domain as possible. This includes working with (negative) log-likelihoods rather than likelihoods, as well as with the log-determinant of covariance matrices rather than the determinant, where possible.

5. When performing the later investigations, it will be useful to detect certain problematic cases your code does not have to handle, and throw exceptions

in these cases, so that the investigation code you write can be robust when fitting many GMMs sequentially. For your convenience, one such exception class is provided in `gmm.py`.

Investigation

Add a Python module called `wine.py` to the project. In this module, place code to perform and document the investigation topics below. Your module must be set up so that running the module outputs the results and any comments you have on the results.

1. Use `hard=False` throughout this investigation.

2. Note that your GMM fitting model is stochastic when initial weights are not provided, but deterministic when they are.[1] First, fit GMM models to the wine data (provided in `data/wine.data`) when the actual class label information is provided in the initial weights. Compare the negative log-likelihood of the fitted models for for `diagcov` set to `True` and `False`, and try to explain your observations.

3. For the GMM with `diagcov=False` above, plot the points of the wine data set on the two linear discriminant axes of the data set (calculated using `center=True` and `scale=True`). Colour each point using a colour with red, green, and blue (RGB) components determined by the responsibilities of each component of the GMM. In this way, we will be able to see how strongly each point is allocated to each component. If we assume the responsibilities are probabilities, we can calculate the log-odds of those probabilities, a value taking on arbitrary real values. Applying the logistic function to (a scaled version of) this value maps it to $(0, 1)$, where we can use it as an RGB component. This approach yields a much less abrupt colour change than simply plotting the probabilities as RGB components (try it). In other words, the component value associated with a responsibility r should be $\sigma(c \log \frac{r}{1-r})$, for some constant c. Find a value of c that gives a visible blending of colours between components.

4. Repeat the previous step, but without initializing the weights (i.e. so that the weights are initialized by `nubskmeans`). Instead, specify `K=3`. (Use the same value of c determined earlier.)

5. Print the negative log-likelihoods of the fitted models for 1 to 4 components, and for `diagcov` set to `True` and `False`, as a 4x2 array.

Add a Python module called `speech.py` to the project. In this module, place code to perform and document the investigation topics below. Your module must be set up so that running the module outputs the results and any comments you have on the results.

1. Use `hard=True` throughout this investigation.

[1] This is because the approach to finding new means in `nubskmeans` is stochastic.

2. Load the data in the file `data/speech.dat`. This is a pickled[2] dictionary, containing a training and a test set for a diphthong classification task. Each set is a dictionary, with a key for each diphthong. The value corresponding to each key is a list of encoded speech fragments represented as nx16 arrays.[3]

3. Normalize the training and test data by subtracting the mean of all the 16-dimensional training points, and scaling each component so that the standard deviation of the component over the training set is 1.

4. We shall assume that the rows of the nx16 arrays are i.i.d. samples from an underlying distribution, which is the same for all samples from a given diphthong. Thus, we can use all the rows from all the speech fragments for a diphthong in the training set to construct a model of the 16-dimensional points appearing in a speech signal representing that diphthong. Do this using a GMM to model each diphthong. Since the GMM's initialization can lead to local optima, fit each diphthong's model a number of times, and keep the one with the best negative log-likelihood.

5. Given these models of each diphthong, and information on the prior probabilities of each diphthong appearing, one can calculate the most likely class for a given observation (i.e. collection of 16-dimensional points). This is done by appropriately adjusting the negative log-likelihood of each model for the given observation to take the prior proportion information into account. (Hint: you can use the `calcresps` method to obtain the negative log-likelihood for each model.) Implement such a classifier — you may find the `classifiers.optcriterion` method of assignment 2 helpful here.

6. Apply the classification approach described here to both the training and the test set, using a non-uniform prior on the proportions. (Cheat a little by using a prior which matches the label distribution on the test set.)

7. Perform the classification approach described above for K from 1 to 6, using both `diagcov=False` and `diagcov=True`. (This should take a while to run, and may sometimes give anomalous results, and thus should not be executed in your submission when `speech.py` is run. However, the code to do this should be in the submission, and your submission should print instructions on how to enable it.)

8. Plot typical training and test error rates calculated in the last step against K on a line plot, and try to explain what you see. In particular, discuss why you think the performance on the test set can get worse for larger values of K, even though more parameters are being used to model the data. Which combination `(K,diagcov)` gives you the best error rate?[4]

[2]Python uses a module called `pickle` to save Python objects to a file. Such saved objects are said to be pickled. If you are not familiar with reading and writing Python objects to file, it is recommended you consult the documentation of Python's `pickle` module.

[3]The data points are a small selection of *cepstra* of speech samples from a standard database for speech recognition problems, the Texas Instruments/Massachusetts Institute of Technology (TIMIT) database. To find out more about this representation, see the Wikipedia page for "cepstrum".

[4]In practice, we can not use our test set to select our model parameters as we do in this assignment. Instead, the training data is segmented into a training and a validation set, where

More challenging tasks

1. Investigate and implement a good way to automatically detect a good value for K to use in the K-means and GMM EM algorithm.

2. Compare the performance of GMM to K-means for classification, and try to explain your results.[5]

Advice

1. See the advice from earlier assignments.

2. Assigning the value `float("inf")` to a variable is a convenient way to ensure it will be larger than any future floats it gets compared to.

3. Your code for nearest neighbours classification from assignment 1 might be useful for the allocation of points to means in the K-means and hard-allocation EM algorithms.

4. The following NumPy methods may come in useful: `logaddexp` (see Section 8.3.3 of the class notes), `random.sample`, `slogdet`, `seterr`.

5. For analysing numerical issues arising in your program, using `seterr` to cause Python to raise exceptions rather than print warnings can be useful. This allows you to get extra information about the data causing the problems. It is recommended you find out about exception handling in Python if you are not yet familiar with it.

6. The `matplotlib.pyplot` method `scatter` can accept RGB specifications for the colour of each point.

Evaluation

1. Implementation: 65 marks (`gmm` — 17; `calcresps` — 17, `updatecovs` — 12, `calcweights` — 8, `updatemeans` — 7, `updatenums` — 4)

2. Investigation: 20 marks (wine investigation — 8; speech investigation — 12)

3. Further investigation: 15 marks

the validation set gets used for model selection. (Other approaches, such as cross-validation are also used.)

[5] For K-means, one can use the distortion measure for classification, since it has a maximum likelihood interpretation under certain assumptions. Find out about, and explain, this.