gmm_diag and gmm_full: C++ classes for multi-threaded Gaussian mixture models and Expectation-Maximisation

Conrad Sanderson [†◇*] and Ryan Curtin [‡*]

[†] Data61, CSIRO, Australia
[‡] Symantec Corporation, USA
[◇] University of Queensland, Australia
[*] Arroyo Consortium

Statistical modelling of multivariate data through a convex mixture of Gaussians, also known as a Gaussian mixture model (GMM), has many applications in fields such as signal processing, econometrics, and pattern recognition [2]. Each component (Gaussian) in a GMM is parameterised with a weight, mean vector (centroid), and covariance matrix.

gmm_diag and *gmm_full* are C++ classes which provide multi-threaded (parallelised) implementations of GMMs and the associated Expectation-Maximisation (EM) algorithm for learning the underlying parameters from training data [5, 6]. The *gmm_diag* class is specifically tailored for diagonal covariance matrices (all entries outside the main diagonal in each covariance matrix are assumed to be zero), while the *gmm_full* class is tailored for full covariance matrices. The *gmm_diag* class is typically much faster to train and use than the *gmm_full* class, at the potential cost of some reduction in modelling accuracy.

The interface for the *gmm_diag* and *gmm_full* classes allows the user easy and flexible access to the trained model, as well as control over the underlying parameters. Specifically, the two classes contain functions for likelihood evaluation, vector quantisation, histogram generation, data synthesis, and parameter modification, in addition to training (learning) the GMM parameters via the EM algorithm. The classes use several techniques to improve numerical stability and promote convergence of EM based training, such as keeping as much as possible of the internal computations in the log domain, and ensuring the covariance matrices stay positive-definite.

To achieve multi-threading, the EM training algorithm has been reformulated into a MapReduce-like framework [4] and implemented with the aid of OpenMP pragma directives [3]. As such, the EM algorithm runs much quicker on multi-core machines when OpenMP is enabled during compilation (for example, using the `-fopenmp` option in GCC and clang compilers).

The *gmm_diag* and *gmm_full* classes are included in version 7.960 of the Armadillo C++ library [7], with the underlying mathematical details described in [8]. The documentation for the classes is available online: `http://arma.sourceforge.net/docs.html#gmm_diag`

For an instance of the *gmm_diag* or *gmm_full* class named as **`M`**, an overview of its member functions and variables is given below; all vectors, matrices and cubes refer to corresponding objects from the Armadillo library; the word "heft" is explicitly used in the classes as a shorter version of "weight", while keeping the same meaning with the context of GMMs. Figure 1 contains a complete C++ program which demonstrates usage of the *gmm_diag* class.

Published as:
Conrad Sanderson and Ryan Curtin.
gmm_diag and gmm_full: C++ classes for multi-threaded Gaussian mixture models and Expectation-Maximisation.
Journal of Open Source Software, Vol. 2, 2017.
DOI: 10.21105/joss.00365

- **`M.log_p(V)`**
  return a scalar (of type *double*) representing the log-likelihood of column vector **`V`**

- **`M.log_p(V, g)`**
  return a scalar (of type *double*) representing the log-likelihood of column vector **`V`**, according to Gaussian with index **`g`** (specified as an unsigned integer of type *uword*)

- **`M.log_p(X)`**
  return a row vector (of type *rowvec*) containing log-likelihoods of each column vector in matrix **`X`**

- **`M.log_p(X, g)`**
  return a row vector (of type *rowvec*) containing log-likelihoods of each column vector in matrix **`X`**, according to Gaussian with index **`g`** (specified as an unsigned integer of type *uword*)

- **`M.sum_log_p(X)`**
  return a scalar (of type *double*) representing the sum of log-likelihoods of all column vectors in matrix **`X`**

- **`M.sum_log_p(X, g)`**
  return a scalar (of type *double*) representing the sum of log-likelihoods of all column vectors in matrix **`X`**, according to Gaussian with index **`g`** (specified as an unsigned integer of type *uword*)

- **`M.avg_log_p(X)`**
  return a scalar (of type *double*) representing the average log-likelihood of all column vectors in matrix **`X`**

- **`M.avg_log_p(X, g)`**
  return a scalar (of type *double*) representing the average log-likelihood of all column vectors in matrix **`X`**, according to Gaussian with index **`g`** (specified as an unsigned integer of type *uword*)

- **`M.assign(V, dist_mode)`**
  return an unsigned integer (of type *uword*) representing the index of the closest mean (or Gaussian) to vector **`V`**; the parameter **`dist_mode`** is one of:

`eucl_dist`	Euclidean distance (takes only means into account)
`prob_dist`	probabilistic "distance", defined as the inverse likelihood (takes into account means, covariances, hefts)

- **`M.assign(X, dist_mode)`**
  return a row vector of unsigned integers (of type *urowvec*) containing the indices of the closest means (or Gaussians) to each column vector in matrix **`X`**; parameter **`dist_mode`** is **`eucl_dist`** or **`prob_dist`**, as per the **`.assign()`** function above

- **`M.raw_hist(X, dist_mode)`**
  return a row vector of unsigned integers (of type *urowvec*) representing the raw histogram of counts; each entry is the number of counts corresponding to a Gaussian; each count is the number times the corresponding Gaussian was the closest to each column vector in matrix **`X`**; parameter **`dist_mode`** is **`eucl_dist`** or **`prob_dist`**, as per the **`.assign()`** function above

- **`M.norm_hist(X, dist_mode)`**
  similar to the **`.raw_hist()`** function above; return a row vector (of type *rowvec*) containing normalised counts; the vector sums to one; parameter **`dist_mode`** is either **`eucl_dist`** or **`prob_dist`**, as per the **`.assign()`** function above

- **`M.generate()`**
  return a column vector (of type *vec*) representing a random sample generated according to the model's parameters

- **`M.generate(N)`**
  return a matrix (of type *mat*) containing **`N`** column vectors, with each vector representing a random sample generated according to the model's parameters

- **`M.n_gaus()`**
  return an unsigned integer (of type *uword*) containing the number of means/Gaussians in the model

- **`M.n_dims()`**
  return an unsigned integer (of type *uword*) containing the dimensionality of the means/Gaussians in the model

- **`M.reset(n_dims, n_gaus)`**
  set the model to have dimensionality **`n_dims`**, with **`n_gaus`** number of Gaussians, specified as unsigned integers of type *uword*; all the means are set to zero, all diagonal covariances are set to one, and all the hefts (weights) are set to be uniform

- **`M.save(filename)`**
  save the model to a file and return a *bool* indicating either success (*true*) or failure (*false*)

- **`M.load(filename)`**
  load the model from a file and return a *bool* indicating either success (*true*) or failure (*false*)

- **`M.means`**
  read-only matrix (of type *mat*) containing the means (centroids), stored as column vectors

- **`M.dcovs`** [only in *gmm_diag*]
  read-only matrix (of type *mat*) containing the diagonal covariances, with the set of diagonal covariances for each Gaussian stored as a column vector; applicable only to the *gmm_diag* class

- **`M.fcovs`** [only in *gmm_full*]
  read-only *cube* containing the full covariance matrices, with each covariance matrix stored as a slice within the cube; applicable only to the *gmm_full* class

- **`M.hefts`**
  read-only row vector (of type *rowvec*) containing the hefts (weights)

- **`M.set_means(X)`**
  set the means (centroids) to be as specified in matrix **`X`** (of type *mat*), with each mean (centroid) stored as a column vector; the number of means and their dimensionality must match the existing model

- **`M.set_dcovs(X)`** [only in *gmm_diag*]
  set the diagonal covariances to be as specified in matrix **`X`** (of type *mat*), with the set of diagonal covariances for each Gaussian stored as a column vector; the number of diagonal covariance vectors and their dimensionality must match the existing model; applicable only to the *gmm_diag* class

- **`M.set_fcovs(X)`** [only in *gmm_full*]
  set the full covariances to be as specified in *cube* **`X`**, with each covariance matrix stored as a slice within the cube; the number of covariance matrices and their dimensionality must match the existing model; applicable only to the *gmm_full* class

- **`M.set_hefts(V)`**
  set the hefts (weights) of the model to be as specified in row vector **`V`** (of type *rowvec*); the number of hefts must match the existing model

- **`M.set_params(means, covs, hefts)`**
  set all the parameters at the same time; the type and layout of the parameters is as per the **`.set_hefts()`**, **`.set_means()`**, **`.set_dcovs()`** and **`.set_fcovs()`** functions above; the number of Gaussians and dimensionality can be different from the existing model

- **`M.learn(data, n_gaus, dist_mode, seed_mode, km_iter, em_iter, var_floor, print_mode)`**
  learn the model parameters via the *k*-means and/or EM algorithms, and return a boolean value, with *true* indicating success, and *false* indicating failure; the parameters have the following meanings:

  - **`data`**
    matrix (of type *mat*) containing training samples; each sample is stored as a column vector
  - **`n_gaus`**
    set the number of Gaussians to **`n_gaus`**; to help convergence, it is recommended that the given **`data`** matrix (above) contains at least 10 samples for each Gaussian
  - **`dist_mode`**
    specifies the distance used during the seeding of initial means and *k*-means clustering:

`eucl_dist`	Euclidean distance
`maha_dist`	Mahalanobis distance, which uses a global diagonal covariance matrix estimated from the given training samples

  - **`seed_mode`**
    specifies how the initial means are seeded prior to running *k*-means and/or EM algorithms:

`keep_existing`	keep the existing model (do not modify the means, covariances and hefts)
`static_subset`	a subset of the training samples (repeatable)
`random_subset`	a subset of the training samples (random)
`static_spread`	a maximally spread subset of training samples (repeatable)
`random_spread`	a maximally spread subset of training samples (random start)

    Note that seeding the initial means with **`static_spread`** and **`random_spread`** can be more time consuming than with **`static_subset`** and **`random_subset`**; these seed modes are inspired by the so-called *k-means++* approach [1], with the aim to improve clustering quality.

  - **`km_iter`**
    the maximum number of iterations of the *k*-means algorithm; this is data dependent, but typically 10 iterations are sufficient
  - **`em_iter`**
    the maximum number of iterations of the EM algorithm; this is data dependent, but typically 5 to 10 iterations are sufficient
  - **`var_floor`**
    the variance floor (smallest allowed value) for the diagonal covariances; setting this to a small non-zero value can help with convergence and/or better quality parameter estimates
  - **`print_mode`**
    boolean value (either *true* or *false*) which enables/disables the printing of progress during the *k*-means and EM algorithms

```
#include <armadillo>

using namespace arma;

int main()
  {
  // create synthetic data containing
  // 2 clusters with normal distribution

  uword d = 5;       // dimensionality
  uword N = 10000;   // number of samples (vectors)

  mat data(d, N, fill::zeros);

  vec mean1 = linspace<vec>(1,d,d);
  vec mean2 = mean1 + 2;

  uword i = 0;

  while(i < N)
    {
    if(i < N)  { data.col(i) = mean1 + randn<vec>(d); ++i; }
    if(i < N)  { data.col(i) = mean1 + randn<vec>(d); ++i; }
    if(i < N)  { data.col(i) = mean2 + randn<vec>(d); ++i; }
    }


  // model the data as a diagonal GMM with 2 Gaussians

  gmm_diag model;

  bool status = model.learn(data, 2, maha_dist, random_subset, 10, 5, 1e-10, true);

  if(status == false)  { cout << "learning failed" << endl; }

  model.means.print("means:");

  double overall_likelihood = model.avg_log_p(data);

  rowvec     set_likelihood = model.log_p( data.cols(0,9) );
  double  scalar_likelihood = model.log_p( data.col(0)    );

  uword   gaus_id  = model.assign( data.col(0),    eucl_dist );
  urowvec gaus_ids = model.assign( data.cols(0,9), prob_dist );

  urowvec histogram1 = model.raw_hist (data, prob_dist);
   rowvec histogram2 = model.norm_hist(data, eucl_dist);

  model.save("my_model.gmm");

  mat modified_dcovs = 2 * model.dcovs;

  model.set_dcovs(modified_dcovs);

  return 0;
  }
```

Figure 1: An example C++ program which demonstrates usage of a subset of functions available in the ***gmm_diag*** class.

References

[1] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. *ACM-SIAM Symposium on Discrete Algorithms*, pages 1027–1035, 2007.

[2] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] G. McLachlan and T. Krishnan. *The EM Algorithm and Extensions*. John Wiley & Sons, 2nd edition, 2008.

[6] T. K. Moon. Expectation-maximization algorithm. *IEEE Signal Processing Magazine*, 13(6):47–60, 1996.

[7] C. Sanderson and R. Curtin. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1:26, 2016.

[8] C. Sanderson and R. Curtin. An open source C++ implementation of multi-threaded Gaussian mixture models, k-means and expectation maximisation. *International Conference on Signal Processing and Communication Systems*, 2017.