ASEN 6519 Project Final Project
Using Variational Gaussian Mixture Models for
Autonomous Task Segmentation

Carl Mueller

May 16, 2019

Contents

Application and Context **2**
Overview 2
GMM's as Segments 2

Probabilistic Graphical Model **2**
Variational Approximation 3
Variational Iterative (EM) Algorithm 5

Project Level Results and Discussion **5**
Level 1 6
Level 2 6
Level 3 7

Conclusion **7**

Technical Contributors **7**

References **7**

Appendix A: Code **8**

Application and Context

Overview

This project will implement and extend prior work on robot trajectory segmentation. A robot trajectory consists of a sequence of robot state data, captured over a given time period, that encodes the progression of movements / actions taken by the robot. State data usually consists of low-level features such as robot joint configuration, end effector position, object position, and object-object relative data etc,. This project will also incorporate high-level features such as behavioral constraints that capture abstract concepts, such as keeping a certain object upright for a determined segment of the task. These trajectories are often used as sample data in robotic learning techniques such as Learning from Demonstration (LfD).

A robot trajectory might constitute a number of sub-tasks that have goal and stylistic behaviors distinct from other sub-task in other parts of the trajectory. Unless explicitly segmented and outline by a human operator, there is no clear obvious indicators that differentiate one sub-task sequence from another sub-task sequence withing the same trajectory. Thus, the goal of autonomous trajectory segmentation is to generate a candidate segmentation sequence for which each segment ideally captures the dynamics (style) and goals (pertinent objects, target locations, constraints etc,.) of some sub-task. For example, the standard pick-and-place task ubiquitously used throughout robotic research might consist of three sub-tasks:

1. **Pick**: The goal of this sub-task is to pick up a target object. Most sub-sequences of a trajectory will consist of data that signifies the end-effector moving towards the target object.
2. **Carry**: This sub-task can be vague, but often consists of the aspect of the pick-and-place task where the robot carries an object over to a the target area for placement.
3. **Place**: The goal of this sub-task is to place. The state data sequence will generally consist of moving towards the target location and releasing the object.

GMM Components as Segments

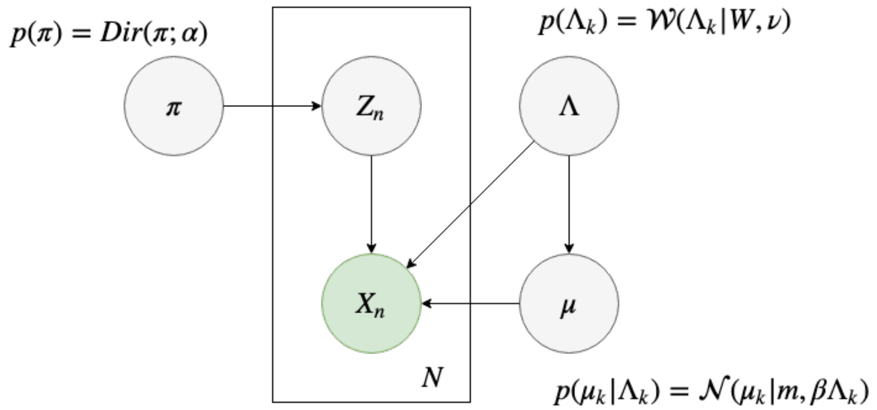
Many autonomous segmentation techniques make use of Gaussian Mixture Models. Ghahramani et. al. [1] showed how Gaussian Mixture models are a form of linear regression that linearize around the mixture means. In the spirit of this idea, Lee et. al. [2] fit GMM directly to the robot trajectory data where each data point $x_t \in X$, where X is a a sequence of state vectors x_i, is assigned to the best fitting component. Thus each $x_i \in X$ is assigned to a cluster of points representing a segment. Each component is, in a sense, a regression around the data for that specific segment. To capture more of the model dynamics, [3] utilizes the stacked vector of $\begin{bmatrix} x_t \\ x_{t+1} \end{bmatrix}$ and fits a GMM on this newly generated sequence of data. The GMM in this case can be thought of as a switched linear dynamical system in which each component's μ and Σ represent the dynamics for all data most associated with that given component. In either case, these works utilized the clustering of points to their corresponding mixture component to generate a segmentation of the entire trajectory.

Probabilistic Graphical Model

The model will be the Variational GMM model a la Bishop [4], built upon an observation set of state vectors but augmented with additional "observations" that consists of a vector of boolean, or one-hot, constraints assignments. The motivation is that during any given robot trajectory demonstration, a user might assign a set of boolean evaluated hard constraints. The constraints might include such high-level concepts such as, "keeping a cup upright" or "the object must be over the target before pouring". These abstract constraints are evaluated on each $x_i \in X$ of the robot state vector i.e. the set of observations.

A demonstration is defined as a single episode of capturing a sequence of robot state trajectory and constraint assignments. Thus a demonstration represents X. For any set of demonstrations associated with a specific skill or task, this model will be run separately. There will be K parameters associated with S components of the mixture model. K will be determined automatically from the way the variational method up-scales / down-scales the mixand component weights. However, since the model is only using a Dirichlet prior and not a Dirichlet process, K must be appropriately high for the model.

The random variables μ and Λ represent means and precisions of the components as they best fit to the data $x_1 : N$. Precisions are used as they are more computationally tractable during the algorithm. What I am still attempting to design is the emission distribution for C_i, the set of applied boolean constraints for any given component. Perhaps each constraint will receive some probability of being true, and the output for a generated sample will the component-wise threshold indication of the estimated distribution. Thus if $p(c_i|Z_i = m) = [.1, .5, .7, .2]$ the resulting output will be $[0, 1, 1, 0]$ assuming a threshold of .5.


$p(\pi) = Dir(\pi; \alpha)$
$p(\Lambda_k) = \mathcal{W}(\Lambda_k | W, \nu)$
π
Z_n
Λ
X_n
μ
N
$p(\mu_k | \Lambda_k) = \mathcal{N}(\mu_k | m, \beta\Lambda_k)$

GMM.png

Figure 1: Graphical model of the variational GMM that emits both robot state vectors as well as a one-hot boolean encoding of constraints.

Variational Approximation

Often the main goal of a probabilistic model is to infer the posterior distribution $P(z|x)$ where z are latent random variables. Ultimately the inference problem can be stated with the following [5]:

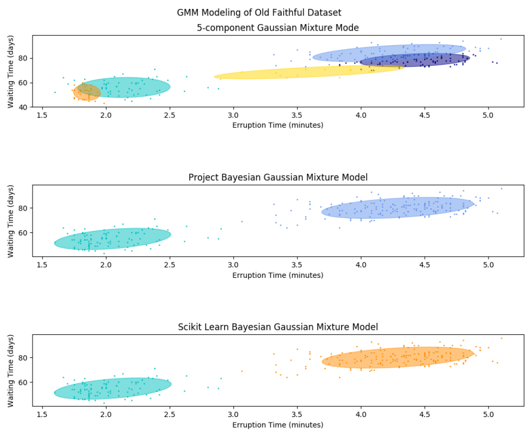
$$p(z|x, \theta) = \frac{p(z, x|\theta)}{\int_z p(x, z|\theta)}$$

where, again, z are the latent variables, x is the known data, and θ represents the parameters on the latent variables. Depending on the graphical model, such as when the latent variables z are themselves representative of unknown prior distributions on the model, the above inference equation becomes intractable. Specifically, the integral for the evidence in the denominator that marginalizes over the latent variables is often unavailable in closed form or requires exponential time to compute [5].

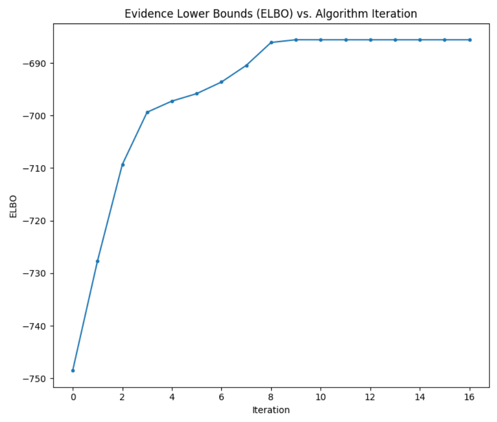
To over come this intractability, an approximate distribution $q(Z)$ is chosen from a family of approximate densities Ω as a representative proxy for the actual posterior distribution, with Z representing the set of latent variables. This distribution is called the variational distribution. The closer $q(z)$ is to the true posterior, the more accurate our approximate inference becomes. Thus we transform the intractable inference problem to a tractable optimization problem where we solve for the optimal $q^*(Z)$ via:

$$q^*(z) = \underset{q(z)\in\Omega}{argmin}\ KL(q(Z)||p(z|x))$$

where KL is the Kullback-Leibler divergence, a pseudo-metric that serves as a measure of distances between two distributions. The true optimal solution is when $q^*(z) = p(z|x)$, but in practice we only achieve parity within a chosen tolerance. The KL divergence can be computed as follows, where all expectations are taken with respect to q(z) [5]:



(a) Top: Non-Bayesian GMM 5 component fit. Middle: Project Variational GMM fit where three of five components have been nullified. Bottom: Scikit-Learn Equivalent Variational GMM fit.



(b) Evidence Lower Bounds increases at every iteration, plateauing until it the successive difference is below tolerance.

Figure 2: Old Faithful eruption data set with 3 models and the ELBO increase for the project model.

$$KL(q(Z)||p(z|x)) = E[log(q(z))] - E[log(p(z,x))] + log(p(x))$$
$$\mathcal{L} = E[log(q(z))] - E[log(p(z,x))]$$

where the term $\mathcal{L}$ is known as the evidence lower bound (ELBO). By maximizing the ELBO (since $p(x)$ is positive), we minimize the KL divergence.

With respect the variational Gaussian Mixture Model outlined in Figure 1, the chosen variational distribution is a factorized distribution built upon the chosen prior distributions of the model itself [4]:

$$q(Z, \pi, \mu, \Lambda) = q(Z)q(\pi, \mu, \Lambda)$$

and the variational lower bounds is represented by [4]:

$$\mathcal{L} = E[log(q(X, Z, \pi, \mu, \Lambda))] - E[log(q(Z, \pi, \mu, \Lambda))]$$

The optimization algorithm, akin the the Expectation Maximization algorithm, for the Variational Gaussian mixture model that optimizes the above is outlined in Algorithm 1.

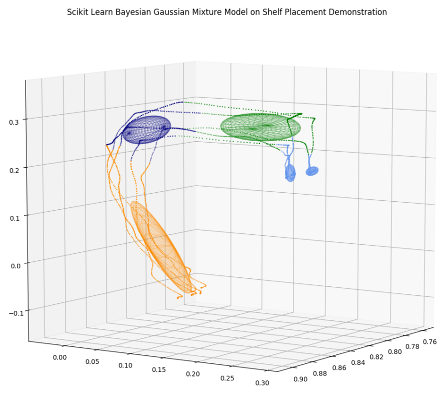
Variational Iterative (EM) Algorithm

```
/* Initial or prior parameters of prior distributions.                          */
/* W: Prior Wishart scaling matrix                                              */
/* $\nu$:  Degrees of freedom on Wishart distribution                           */
/* $\alpha$:  Prior Dirichlet parameter                                         */
/* $\beta$:  Prior scale on precisions for normal distribution                  */
/* tolerance:  Lower bound change tolerance to indicate convergence.            */
/* maxIter:  Maximum number of iterations allowed.                              */
Input: W, $\nu$, $\alpha$, $\beta$, tolerance, maxIter
/* $\pi$:  Expected weights                                                     */
/* means:  Expected means                                                       */
/* $\Lambda$:  Expected precisions.                                             */
/* resp:  Responsibilities.                                                     */
Output: $\pi$, means, $\Lambda$, resp
/* Initialize all parameters.  Often uses KMeans clustering to generate means and
   covariances for prior distributions.                                         */
initialization();
$lowerBoundPrior \leftarrow -\inf$;
for $i \leftarrow 2$ to $n$ do
    /* Variational E-Step:                                                      */
    $logPi \leftarrow$ updateExpectedLogPi(); // Log weights
    $logLambda \leftarrow$ updateExpectedLogLambda(); // Expected log determinant of precisions
    $resp \leftarrow$ estimateResponsibilities(); // Calculate component responsbilities.

    /* Variational M-Step:                                                      */
    $N_k, x_bar_k, S_k \leftarrow$ estimateGaussianStatistics($resp$); // Calculates current component
        association, means, and covariances
    $\alpha \leftarrow$ updateDirichletPriorParameter($N_k$); // Updates current dirichlet parameter
    $\nu \leftarrow$ updateWishartPriorParameters($N_k$); // Updates component degrees of freedom.
    $\beta \leftarrow$ updateGaussianPriorParameters($N_k$); // Updates beta precision scaling for each
        component.
    $\pi \leftarrow$ estimateWeights($N_k$); // Updates current weights.
    means $\leftarrow$ estimateMeans($N_k, x_bar_k$); // Updates current means.
    $\Lambda \leftarrow$ calculatePrecisionsCholesky(estimateWishartMatrix($N_k, x_bar_k, S_k$)); // Updates
        precisions from estimated Wishart matrix

    /* Check for convergence:                                                   */
    $lowerBoundCurr \leftarrow$ calculateELBO($logPi$, $logLambda$, $resp$, $\alpha$, $\beta$, means, $\Lambda$); if
     $abs(lowerBoundPrior - lowerBoundCurr) <= tolerance$ then
        break;
    end
end
return $\pi$, means, $\Lambda$, resp;
```

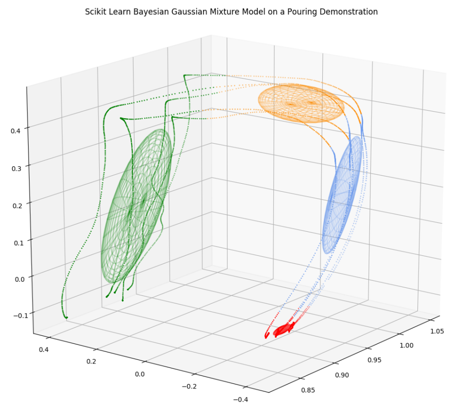
Algorithm 1: Variational EM algorithm.

Project Level Results and Discussion

This project originally sought to emulate the the work of [3] but incorporate the use of conceptual constraints from my own prior work [6]. What the project was able to accomplish is a generation of a a sequence of broad segments using the variational Gaussian Mixture Model which automatically select the number of components. The following outlines the progress made towards each project level:



(a) Mixand ellipsoids overlayed on 5 trajectories of demonstrations of placing an object onto a shelf.



(b) Mixand ellipsoids overlayed on 5 trajectories of demonstrations of a pouring task.

Figure 3: Example of project model fitted onto actual Sawyer trajectory demonstration data two tasks.

Level 1

While implementation woes prevented much progress towards the stated goal of expanding upon the work of [3], the project does successfully reach most of the stated Level 1 goals:

- Implement variational expectation-maximization-like algorithm for the variational model.
- Compare output of implementation to expected results on Old Faithful data set.

The project employs a successful implementation of a Variational Gaussian Mixture Model, mirrors established clustering on the Old Faithful data set (Figure 2a), produces adequate lower bounds increase (Figure 2b), and, lastly, provides reasonable autonomous segmentation of existing demonstration trajectory data from prior work done in [6]. For the lower bound, two approaches were attempted for the lower bound. One used the lower bound equations in chapter 10 of [4], verbatim, but could not successfully integrate with the chosen implementation of using Cholesky decomposed precision matrices in lieu of the standard Wishart scaling matrix. As such, the successful implementation mirrors Scikit-Learns BayesianGaussianMixture class implementation, and uses the same simplified lower bound / ELBO functions.

Level 2

For level two, the project only managed to reach one goal:

- Use model on robot trajectory demonstration data used in the experiments conducted by Mueller et. al [6].

The variational Gaussian Mixture model provided quite intuitive segmentation based on the mixands (See Figures 3a & 3b). For this project, the model only uses end-effector (X, Y, Z) position and thus the segmentation is inherently confined to variations in trajectories based on the scope of the state data. However, the model successfully segments broad motor movements that outline changes the dynamics of the end-effector movement, as well as segments where the end effector shows little variation in its position. This is usually when the end-effector is doing high-precision movement (pouring or placing the object) but is localized in a tight X, Y, Z space. This type of dynamics-based segmentation seems to validate the Transition State Clustering approach of [3].

Unfortunately, the planned MIME data set modeling to not come to fruition. The data set only uses robot joint position for the Baxter robot. While the model could be run on this data, running forward kinematics to create easily visualized end-effector data was not possible in the remaining time since updating the project's goals. However, software development kits do exist that could allow this same approach on the Baxter data. This would provide a larger data set to evaluate the segmentation approach.

Level 3

Unfortunately, I did not make any progress towards my level 3 goals. One continued approach to advance this project towards a more substantial research endeavour would be to develop a mechanisms to easily distinguish high variation broad movements from low variation (with respect to euclidean end-effector space) that indicate high precision movements. This would enable a system to autonomously deconstruct a task from a low-level perspective in order to build a more abstracted representation of the task. Additionally, the models do not directly incorporate conceptual constraints. Encoding constraints and generating meaningful segmentation is a next avenue of research. Determining how to encode constraints into the Transition States outlined in [3] will be a challenging extension to explore next.

Conclusion

I learned a tremendous amount completing this project. I learned how to represent a problem from my research domain as a graphical model. While the model I used was well-established and studied, learning how to apply an existing problem to fit such a model is a tremendous skill to learn. Likewise, I've developed an appreciation for Bayesian approaches to inference and the awesome expressive power they provide over standard expectation maximization approaches.

My advice to those following in my footsteps with regards to Probabilistic Modeling is to not trust the implementations of examples throughout the internet. Only by really deconstructing the algorithms with a deep understanding can one be certain that their own implementation is theoretically correct. This is something I failed to fully accomplish with respect to the evidence lower bound calculation. While I understand and implemented the equations outlined in Bishop's book, my own implementation was different than that used by Scikit-Learn. However, I feel confidence that I am armed with enough knowledge to fully deconstruct the ELBO calculation, and any other probabilistic model.

Thank you for offering such a great course. It was one of my favorites thus far in my graduate career.

Technical Contributors

All technical contributions will be completed by yours truly, Carl Mueller, as I am doing this project by myself.

References

[1] Zoubin Ghahramani and Michael I Jordan. Supervised learning from incomplete data via an em approach. In *Advances in neural information processing systems*, pages 120–127, 1994.

[2] Sang Hyoung Lee, Il Hong Suh, Sylvain Calinon, and Rolf Johansson. Autonomous framework for segmenting robot trajectories of manipulation task. *Autonomous robots*, 38(2):107–141, 2015.

[3] Sanjay Krishnan, Animesh Garg, Sachin Patil, Colin Lea, Gregory Hager, Pieter Abbeel, and Ken Goldberg. Transition state clustering: Unsupervised surgical trajectory segmentation for robot learning. *The International Journal of Robotics Research*, 36(13-14):1595–1618, 2017.

[4] Christopher M Bishop. *Pattern recognition and machine learning.* springer, 2006.

[5] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

[6] Carl Mueller, Jeff Venicx, and Bradley Hayes. Robust robot learning from demonstration and skill repair using conceptual constraints. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6029–6036. IEEE, 2018.

Appendix A: Code

```python
import numpy as np
from scipy import linalg
from scipy.special import digamma, logsumexp, gammaln, multigammaln
from sklearn.cluster import KMeans


def compute_precisions_cholesky(covariances):
    # Pulled from sci-kit learn.
    n_components, n_features, _ = covariances.shape
    precisions_chol = np.empty((n_components, n_features, n_features))
    for k, covariance in enumerate(covariances):
        try:
            cov_chol = linalg.cholesky(covariance, lower=True)
        except linalg.LinAlgError:
            raise ValueError(
                "Error. COllapsed samples. Try decreasing number of components or ...
                    increasing regularization.")
        precisions_chol[k] = linalg.solve_triangular(cov_chol,
                                                     np.eye(n_features),
                                                     lower=True).T
    return precisions_chol

def _compute_log_det_cholesky(matrix_chol, n_features):
    """Compute the log-det of the cholesky decomposition of matrices.

    Parameters
    ----------
    matrix_chol : array-like
        Cholesky decompositions of the matrices.
        'full' : shape of (n_components, n_features, n_features)
        'tied' : shape of (n_features, n_features)
        'diag' : shape of (n_components, n_features)
        'spherical' : shape of (n_components,)

    covariance_type : {'full', 'tied', 'diag', 'spherical'}

    n_features : int
        Number of features.

    Returns
    -------
    log_det_precision_chol : array-like, shape (n_components,)
        The determinant of the precision matrix for each component.
    """

    n_components, _, _ = matrix_chol.shape
    log_det_chol = (np.sum(np.log(
        matrix_chol.reshape(
            n_components, -1)[:, ::n_features + 1]), 1))

    return log_det_chol

def _log_dirichlet_norm(dirichlet_concentration):
    """Compute the log of the Dirichlet distribution normalization term.

    Parameters
    ----------
    dirichlet_concentration : array-like, shape (n_samples,)
        The parameters values of the Dirichlet distribution.

    Returns
    -------
    log_dirichlet_norm : float
        The log normalization of the Dirichlet distribution.
    """
    return (gammaln(np.sum(dirichlet_concentration)) -
            np.sum(gammaln(dirichlet_concentration)))


def _log_wishart_norm(degrees_of_freedom, log_det_precisions_chol, n_features):
    """Compute the log of the Wishart distribution normalization term.

    Parameters
    ----------
    degrees_of_freedom : array-like, shape (n_components,)
        The number of degrees of freedom on the covariance Wishart
        distributions.

    log_det_precision_chol : array-like, shape (n_components,)
         The determinant of the precision matrix for each component.

    n_features : int
        The number of features.

    Return
    ------
    log_wishart_norm : array-like, shape (n_components,)
        The log normalization of the Wishart distribution.
    """
    # To simplify the computation we have removed the np.log(np.pi) term
    return -(degrees_of_freedom * log_det_precisions_chol +
             degrees_of_freedom * n_features * .5 * np.log(2.) +
             np.sum(gammaln(.5 * (degrees_of_freedom -
                                  np.arange(n_features)[:, np.newaxis])), 0))


class VariationalGMM():

    def __init__(self, n_components=3, max_iter=200, tolerance=1e-6, alpha_prior=None, ...
        beta_prior=1, dof=None,
                 wishart_matrix_prior=None, weights_prior=None, means_prior=None, ...
                     covariances_prior=None,
                 regularization=1e-6):
        self.n_components = n_components  # Number of mixture components (K)
        self.max_iter = max_iter  # number of iterations to run iterative update of ...
            varational inference
        self.tolerance = tolerance  # Log-likelihood tolerance for terminating EM
        self.alpha_prior = alpha_prior  # Dirichlet parameter for prior of weights. (1, K)
        self.beta_prior = beta_prior  # scaling on precisions matrix.
        self.wishart_matrix_prior = wishart_matrix_prior
        self.dof = dof  # degrees of freedom for Wishart distribution
        self.covariances_prior = covariances_prior  # Initial covariances of mixands.
        self.weights_prior = weights_prior  # Initial weights of mixands
        self.means_prior = means_prior  # Initial means of mixands
        self.regularization = regularization
        self.fitted = False

    def _initialize_parameters(self, X):
        self.n_samples_ = np.shape(X)[0]  # number of samples
        self.n_features_ = np.shape(X)[1]  # number of features
        self.alpha_prior = 1. / self.n_components
        self.alpha_k = np.full([self.n_components, ], self.alpha_prior)  # dirichlet ...
            parameters
        self.means_prior = np.mean(X, axis=0)
        self.weights = self.weights_prior if self.weights_prior is not None else np.diag(
            np.random.uniform(0, 1, self.n_components))
        self.beta_k = np.full(self.n_components, self.beta_prior)  # scale of precision ...
            matrix.
        self.log_pi = digamma(self.alpha_k) - digamma(np.sum(self.alpha_k))
        self.log_lambda = np.zeros(self.n_components)
        self.dof = self.dof if self.dof is not None else self.n_features_
        self.nu_k = np.full([self.n_components, ], self.dof)
        self.W_k = np.zeros(
            [self.n_components, self.n_features_, self.n_features_])  # scaling matrix of ...
                wishart distribution
        self.W_lb = np.zeros(
            [self.n_components, self.n_features_,
             self.n_features_])  # Collects inverse Wishart scaling matricies for use in ...
                 lowerbound
        self.W_prior = self.wishart_matrix_prior if self.wishart_matrix_prior is not None ...
            else np.atleast_2d(
            np.cov(X.T))
        self.W_prior_inv = np.linalg.inv(self.W_prior)  # Inverse of initial wishart component
        self._kmeans_initialize(X)

    def _kmeans_initialize(self, X):
        kmeans = KMeans(n_clusters=self.n_components, n_init=1,
                                   random_state=None).fit(
            X)
        resp = np.zeros((self.n_samples_, self.n_components))
        resp[np.arange(self.n_samples_), kmeans.labels_] = 1
        self.means = kmeans.cluster_centers_.T
        self._initialize(X, resp)

    def _initialize(self, X, resp):
        """Initialization of the mixture parameters.

        Parameters
        ----------
        X : array-like, shape (n_samples, n_features)

        resp : array-like, shape (n_samples, n_components)
        """
        N_k, x_bar_k, S_k = self._estimate_gaussian_statistics(X, resp)
        self._update_dirichlet_prior_parameter(N_k)
        self._update_wishart_prior_parameters(N_k)
        self._update_gaussian_prior_parameters(N_k)
        self._estimate_weights(N_k)
        self._estimate_means(N_k, x_bar_k)
        self._estimate_wishart_matrix(N_k, x_bar_k, S_k)

    def e_step(self, X):
        self._update_expected_log_pi()
        self._update_expected_log_lambda()
        # In the variational E step, the ultiamte goal is to calculate the responsibilities.
        return self.estimate_responsibilities(X)

    def m_step(self, X, resp):
        N_k, x_bar_k, S_k = self._estimate_gaussian_statistics(X, resp)
        self._update_dirichlet_prior_parameter(N_k)
        self._update_wishart_prior_parameters(N_k)
        self._update_gaussian_prior_parameters(N_k)
        self._estimate_weights(N_k)
        self._estimate_means(N_k, x_bar_k)
        self._estimate_wishart_matrix(N_k, x_bar_k, S_k)
        return N_k, x_bar_k, S_k

    def fit(self, X):
        self._initialize_parameters(X)

        self.lower_bounds_ = []
        prior_lb = -np.inf
        for n in range(0, self.max_iter):
            # E-M Step
            self.log_resp, self.resp = self.e_step(X)
            N_k, x_bar_k, S_k = self.m_step(X, self.resp)
            new_lb = self._compute_lower_bound()
            # new_lb = self._calculate_lower_bound(N_k, x_bar_k, S_k)
            # new_lb = self.elbo()

            self.lower_bounds_.append(new_lb)
            if abs(new_lb - prior_lb) <= self.tolerance:
                print("Converged.")
                break
            prior_lb = new_lb
        if abs(new_lb - prior_lb) > self.tolerance:
            print("Algorithm maximum iterations inadequate to achieve convergence ...
                 according to given tolerance.")
        self.log_resp, self.resp = self.e_step(X)
        self.fitted = True
        return self

    def predict(self, X):
        if self.fitted is True:
            return self.get_weighted_log_probability(X).argmax(axis=1)
        else:
            raise RuntimeError("Model not fitted.")

    def estimate_responsibilities(self, X):
        log_rho_nk = self.get_weighted_log_probability(X)
        Z = logsumexp(log_rho_nk, axis=1)
        log_resp = log_rho_nk - Z[:, np.newaxis]
        resp = np.exp(log_resp)
        return log_resp, resp

    def log_gauss(self, X):
        # Using scikit learns implementation

        # Get the log determinant of the Cholesky decomposed precisions.
        log_det_chol = (np.sum(np.log(
            self.precisions_cholesky_.reshape(
                self.n_components, -1)[:, ::self.n_features_ + 1]), 1))

        # Get the log probability of the gaussian
        log_prob = np.empty((self.n_samples_, self.n_components))
        for k, (mu, prec_chol) in enumerate(zip(self.means_, self.precisions_cholesky_)):
            y = np.dot(X, prec_chol) - np.dot(mu, prec_chol)
            log_prob[:, k] = np.sum(np.square(y), axis=1)

        return - .5 * (self.n_features_ * np.log(2 * np.pi) + log_prob) + log_det_chol

    def estimate_log_prob(self, X):
        # log rho, see Bishop 10.46
        log_gauss = self.log_gauss(X)
        return log_gauss + .5 * (self.log_lambda - self.n_features_ / self.beta_k)

    def get_weighted_log_probability(self, X):
        log_prob = self.estimate_log_prob(X)
        weighted_log_prob = log_prob + self.log_pi
        return weighted_log_prob

    def _estimate_gaussian_statistics(self, X, resp):
        S_k = np.zeros(
            [self.n_components, self.n_features_, self.n_features_])  # estimated ...
                 covariances of the components
        N_k = np.sum(resp,
                     axis=0) + 1e-10  # from Bishop 10.51, sum or responsibilities for ...
                         each component i.e. number of data samples in each component
        x_bar_k = np.dot(resp.T, X) / N_k[:, np.newaxis]  # Bishop 10.52
        for k in range(0, self.n_components):
            x_cen = X - x_bar_k[k]
            S_k[k] = np.dot(resp[:, k] * x_cen.T, x_cen) / N_k[k]  # Bishop equation 10.53
            S_k[k].flat[::self.n_features_ + 1] += self.regularization
        return N_k, x_bar_k, S_k

    def _update_dirichlet_prior_parameter(self, N_k):
        self.alpha_k = self.alpha_prior + N_k  # from Bishop 10.58

    def _estimate_weights(self, N_k):
        self.weights = (self.alpha_prior + N_k) / (
                self.n_components * self.alpha_prior + self.n_samples_)  # Bishop 10.69
        # self.weights = (self.alpha_prior + N_k)  # scikit learn doesn't divide by ...
            anything...why?

    def _update_wishart_prior_parameters(self, N_k):
        self.nu_k = self.dof + N_k  # from Bishop 10.63 and according to sci-kit learn, it ...
            shouldn't have the +1

    def _update_gaussian_prior_parameters(self, N_k):
        self.beta_k = self.beta_prior + N_k  # from Bishop 10.60

    def _estimate_means(self, N_k, x_bar_k):

        self.means = (self.beta_prior * self.means_prior + N_k[:, np.newaxis] * x_bar_k) / ...
            self.beta_k[:,
                                                                                np.newaxis] ...
                                                                                    ...
                                                                                   # ...
                                                                                   from ...
                                                                                   Bishop ...
                                                                                   10.61
        self.means_ = self.means

    def _estimate_wishart_matrix(self, N_k, x_bar_k, S_k):
        for k in range(0, self.n_components):
            mean_diff = x_bar_k[k] - self.means_prior
            self.W_k[k] = (self.W_prior + N_k[k] * S_k[k] + N_k[k] * self.beta_prior \
                           / self.beta_k[k] * np.outer(mean_diff,
                                                        mean_diff))  # from Bishop 10.62
            self.W_lb[k] = np.linalg.inv(self.W_k[k])
        self.W_k /= self.nu_k[:, np.newaxis, np.newaxis]
        self.covariances_ = self.W_k
        self.precisions_cholesky_ = compute_precisions_cholesky(self.covariances_)

    def _update_expected_log_pi(self):
        self.log_pi = digamma(self.alpha_k) - digamma(np.sum(self.alpha_k))  # from Bishop ...
            10.66

    def _update_expected_log_lambda(self):
        for k in range(0, self.n_components):
            digamma_sum = 0
            for i in range(1, self.n_features_ + 1):
                digamma_sum += digamma((self.nu_k[k] + 1 - i) / 2)
            self.log_lambda[k] = digamma_sum + self.n_features_ * np.log(2) + ...
                np.log(np.linalg.det(self.precisions_cholesky_[k])) # from Bishop 10.65


    def logB(self, W, nu):
        n_col = np.shape(W)[1]

        gamma_sum = 0
        for i in range(1, n_col + 1):
            gamma_sum += gammaln(0.5 * (nu + 1 - i))
        # Compute logB function via Bishop B.79
        return (-0.5 * nu * np.log(np.linalg.det(W)) - (0.5 * nu * n_col * np.log(2) + ...
            0.25 * n_col * (n_col - 1) *
                                                      np.log(np.pi) + gamma_sum))
```

```python

    def _calculate_lower_bound(self, N_k, x_bar_k, S_k):
        # DECREASES
        log_px = 0
        log_pml = 0
        log_pml2 = 0
        log_qml = 0
        for k in range(0, self.n_components):
            # Here we collect all terms that require summations index by the k-th component.
            diff = x_bar_k[k] - self.means[k]
            # see Bishop 10.71; we remove (- self.n_features_ * np.log(2 * np.pi)) since ...
                it is an additive constant.
            log_px = log_px + N_k[k] * (
                    self.log_lambda[k] - self.n_features_ / self.beta_k[k] - self.nu_k[k] ...
                        * np.trace(
                np.dot(S_k[k], self.W_lb[k])) - self.nu_k[k] * np.dot(
                np.dot(diff, self.W_lb[k]), diff)) - self.n_features_ * np.log(2 * np.pi)

            # see Bishop 10.74
            log_pml = log_pml + self.n_features_ * np.log(self.beta_prior / (2 * np.pi)) + ...
                self.log_lambda[k] - \
                      (self.n_features_ * self.beta_prior) / self.beta_k[k] - ...
                          self.beta_prior * self.nu_k[k] * np.dot(
                np.dot(np.transpose(self.means[k] - self.means_prior),
                       self.W_lb[k]), self.means[k] - self.means_prior)

            # see Bishop 10.74
            log_pml2 = log_pml2 + self.nu_k[k] * np.trace(np.dot(self.W_prior_inv, ...
                self.W_lb[k]))

            # see Bishop 10.77
            log_qml = log_qml + 0.5 * self.log_lambda[k] + 0.5 * self.n_features_ * np.log(
                self.beta_k[k] / (2 * np.pi)) \
                      - 0.5 * self.n_features_ - (-self.logB(W=self.W_lb[k], ...
                          nu=self.nu_k[k]) \
                                                - 0.5 * (self.nu_k[k] - self.n_features_ ...
                                                    - 1) * self.log_lambda[
                                                    k] + 0.5 * self.nu_k[
                                                    k] * self.n_features_)

        log_px = 0.5 * log_px  # see Bishop 10.71
        log_pml = 0.5 * log_pml + self.n_components * self.logB(W=self.W_prior, ...
            nu=self.dof) + 0.5 * (
                self.dof - self.n_features_ - 1) * np.sum(self.log_lambda) - 0.5 * ...
                    log_pml2  # see Bishop 10.74
        log_pz = np.sum(np.dot(self.resp, self.log_pi))  # see Bishop 10.72
        log_qz = np.sum(self.resp * self.log_resp)  # 10.75
        log_pp = np.sum((self.alpha_prior - 1) * self.log_pi) + ...
            gammaln(np.sum(self.n_components * self.alpha_prior)) - \
                 self.n_components * np.sum(gammaln(self.alpha_prior))  # 10.73

        log_qp = np.sum((self.alpha_k - 1) * self.log_pi) + gammaln(np.sum(self.alpha_k)) ...
            - np.sum(
            gammaln(self.alpha_k))  # 10.76

        # Sum all parts to compute lower bound\
        print(log_px + log_pz + log_pp + log_pml - log_qz - log_qp - log_qml)
        return log_px + log_pz + log_pp + log_pml - log_qz - log_qp - log_qml


    def elbo(self):
        # ELBO: evidence lower bounds in order to test for convergence.
        # DECREASES
        lb = gammaln(np.sum(self.alpha_prior)) - np.sum(gammaln(self.alpha_prior)) \
             - gammaln(np.sum(self.log_lambda)) + np.sum(gammaln(self.log_pi))
        lb -= self.n_samples_ * self.n_features_ / 2. * np.log(2. * np.pi)
        for k in range(0, self.n_components):
            lb += (-(self.dof * self.n_features_ * np.log(2.)) / 2.) \
                   + ((self.nu_k[k] * self.n_features_ * np.log(2.)) / 2.)
            lb += - multigammaln(self.dof / 2., self.n_features_) \
                   + multigammaln(self.nu_k[k] / 2., self.n_features_)
            lb += (self.n_features_ / 2.) * np.log(np.absolute(self.beta_prior)) \
                   - (self.n_features_ / 2.) * np.log(np.absolute(self.beta_k[k]))
            lb += (self.dof / 2.) * np.log(np.linalg.det(self.W_prior)) \
                   - (self.nu_k[k] / 2.) * np.log(np.linalg.det(self.W_lb[k]))
            lb -= np.dot(np.log(self.alpha_k[k]).T, self.alpha_k[k])
        return lb

    def _compute_lower_bound(self):
        """Estimate the lower bound of the model.

        The lower bound on the likelihood (of the training data with respect to
        the model) is used to detect the convergence and has to decrease at
        each iteration.

        Parameters
        ----------
        X : array-like, shape (n_samples, n_features)

        log_resp : array, shape (n_samples, n_components)
            Logarithm of the posterior probabilities (or responsibilities) of
            the point of each sample in X.

        log_prob_norm : float
            Logarithm of the probability of each sample in X.

        Returns
        -------
        lower_bound : float
        """
        # Contrary to the original formula, we have done some simplification
        # and removed all the constant terms.
        n_features, = self.means_prior.shape

        # We removed `.5 * n_features * np.log(self.degrees_of_freedom_)`
        # because the precision matrix is normalized.
        log_det_precisions_chol = (_compute_log_det_cholesky(
            self.precisions_cholesky_, n_features) -
            .5 * n_features * np.log(self.nu_k))


        log_wishart = np.sum(_log_wishart_norm(
            self.nu_k, log_det_precisions_chol, n_features))

        log_norm_weight = _log_dirichlet_norm(self.alpha_k)

        return (-np.sum(np.exp(self.log_resp) * self.log_resp) -
                log_wishart - log_norm_weight -
                0.5 * n_features * np.sum(np.log(self.beta_k)))
```