Table of Contents.

IN. Introduction.

NT. Gaussian Mixture Model (GMM) Intuition.

MF. GMM Modelling and Training.

  MF.1. Prerequisite Applied Math.

  MF.2. Modelling GMM.

  MF.3. Training GMM.

EX. GMM In Practice.

  EX.1. Utilities Functions.

  EX.2. GMM Used Cases.

    EX.2.1. GMM as Generative Model: Create Non-Existent Person Image.

    EX.2.2 GMM for predicting Time Series (e.g Stocks, Signals etc).

SM. Summary.

IN. Introduction.

In this Notebook, we will explore Gaussian Mixture Model from various perspectives.

We will first give a short intuitive explanation for Gaussian Mixture Model (GMM) and why it makes sense. Then we will go deeper into the actual derivation of GMM using the EM techniques. Once we have understood the theory and concept, we will dive deeper into the use cases and examples. We will consider following scenarios with examples.

1. GMM as Generative Model: Create Non-Existent Person.
2. GMM for predicting Time Series (e.g Stocks, Signals.

In the end, we will summarize our discussion with various pointers resources.

NT. Gaussian Mixture Model (GMM) Intuition.

NT.1 Why GMM ?

We have seen in the applied math section that how joint probability distribution can be used to model the complex distribution.

However, it is mathematically and practically convenient to have the distribution which can be expressed analytically in closed form and also can be represented by parameter vectors. Gaussian distribution is one such distribution. In fact any distribution given enough samples will converge to Gaussian Distribution due to central limit theorem.

We know that given enough sample, every distribution of data will converge to Gaussian. However, in real life, there may not be generated from one Gaussian. For example, for analysis the time series for stock, we might have many factors like sentiments, fundamentals, stock technicals and earning reports. Each of those factor contribute to one of the Multivariate Gaussain.

We could argue that, we can use Multivariate Gaussian with high dimensions. However, it is a known fact that for higher dimensional distribution (included Gaussian Distribution), most of the mass of the probability density is centered around mean and hence we won't be able to accurately predict outcome.

NT.2 What is GMM ?

Usually, it is the case that dataset is mixture of different proportion of different Multivariate Gaussian. This allows us to avoid curse of dimensionality and each of the Gaussian can model one aspect of the complex problem.

We will see in the example section, how mixture of Gaussian Distribution can robustly model various complex use cases in spite of noise and other factors which cause issues with other models.

This property allows us to think that if the underlying distribution is not Gaussian, we can use the combination or superposition of multiple Gaussian with different mean and covariance matrix.

Instead of just adding various Gaussian distribution with equal weight, we can add them in various proportion which itself can be represented by the another distribution.

NT.3 How is GMM used ?

Gaussian Mixture Model is widely used in Data Science and Mining. We will cover various techniques from theory to examples.

MF. GMM Modelling and Training.

To understand, how GMM works and deep understanding of it, we must go through it from intuition and mathematical formulation. Though, this section can be skipped in first reading. It is recommended that, reader goes through it as it allows him to extend the examples given in this tutorial and also handle more complex use cases.

We will first model the GMM and then using the EM algorithm we will find expression for each steps.

MF.1. Prerequisite Applied Math.

We recommend to go over Applied Math (AppliedMath.ipynb) notebook first as it formalizes all the background that will be needed in this notebook.

Here is quick summary of the topics covered in companion notebook.

IN. Introduction.

LA. Linear Algebra.

ST. Statistics.

PT. Probability Theory.

```
        PT.1. Random Variables.

        PT.2. Properties of Probability.

        PT.3. Parametric Distributions.

                PT.4. Distributions: Multinomial Distribution.

                PT.5. MultiVariate Gaussian Distribution.

        PT. Bayes Theorem.
```

NT. Numerical Techniques.

```
        NT.1. Maximizing a function via Matrix Calculus.
        NT.2. MLE Techniques.

        NT.3. EM Techniques.
```

MF.2. Modelling GMM.

Now we are ready to derive expression for GMM.

As we talked about in the previous sections that data is generated from fixed number of sources and we know that in nature and real life if we have enough data points than their distribution will converge to Gaussian with some mean μ with standard deviation σ. In our case, since there are multiple sources and the final data would be the mixture of K Gaussian distributions. However at any instance of time, data is generated from one of the Gaussian distributions. Since GMM is the unsupervised techniques, there won't be any levels i.e. no y_i. So our data generation process is as follows.

Generative Model for **Gaussian Mixture Model (GMM)**

1. For each i'th sample out of the N i.i.d. samples,

   a. **select one of the K components using Multivariate PMF** i.e. $p(\mathbf{z}_i) = \mathcal{M}ulti(K, \pi)$

   b. From the selected distribution, **draw a sample i with Gaussian PDF with mean μ_k and covariance $\mathbf{\Sigma}_k$** i.e. $p(\mathbf{x}_i|\mathbf{z}_i) = \mathcal{N}(\mu_k, \mathbf{\Sigma}_k)$.

   c. Find the **joint Distribution of GMM for one sample** i.e. $p(\mathbf{x}_i, \mathbf{z}_i) = p(\mathbf{x}_i|\mathbf{z}_i)p(\mathbf{z}_i)$

2. Joint PDF for N samples is GMM $p(\mathbf{X}, \mathbf{z})$

Let's explore these two steps in details.

MF.2.1 Modelling: Select a Distribution

Selection Random Variable.

Since each of the input sample (denoted by $\mathbf{x}_i$) is generated first selecting one of the K distributions. Let's say it is j'th distribution which can be model as a random variable $z_{ij} \in \{0, 1\}$. Since there are K components, j ranges from 1 to K .

However, we are given datasets represented by $\mathbf{x}_i$ and there is no z_{ij} given to us, and hence each these variables doesn't exist in practice and is hidden. These random variables are called hidden state variables.

Vector of Selection Random Variable.

We must note that, since we only select one of K distributions, all the z_{ij} are zeros except one at jth location. The whole setup will be conveniently modelled as vector if we represent each of z_{ij} a K dimensional vector with only j'th entry 1, instead of belonging to $\{0, 1\}$. For example if we select third distribution then $j == 3$ and $z_{ij} = (0, 0, 1, 0, 0, \cdots 0)^T$, a K dimensional vector. This scheme is called 1-of-K encoding and here is 1-of encoding of z_{ij} :

$$z_{ij} \in (0_1, 0_2, \cdots, 0_{j-1}, 1_j, 0_{j+1}, \cdots, 0_K) \qquad \text{;Only } j\text{'th location is 1}$$

Once we represent each selection random variable as 1-of-K scheme, we can represent the selection for i'th sample, as $\mathbf{z}_i$ which can take any values from the set $\{z_{i1}, z_{i2}, \ldots \cdots, z_{iK}\}$.

Note:

- Even though, we use bold case convention for vector, We will represent z_{ik} as non bold since it is not really a vector but a convenient notation.
-

Probability of Selection Random Variable.

What will be probability of each of those z_{ij}? One common way is to compute empirical frequency count. Let's assume that for the i'th input sample, probability that we select j'th sample is just average number of times $\mathbf{z}_i$ was z_{ij} as follows:

$$p(\mathbf{z}_i = z_{ij}) = p(z_{ij}) = \pi_j = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i == z_{ij}); \text{ We used indicator } \mathbb{I} \text{ for simplifying summation}$$

Note

- Since π_j is a expected value and hence it is always positive i.e. $\pi_j \geq 0; \forall j \in (1, \cdots, K)$.
- $\forall i \in (1 \cdots N)$, probability $p(\mathbf{z}_i = j) = \pi_j$ is independent on i.
- we will represent the all the selection probability using vector i.e. $\pi = (\pi_1, \pi_2, \cdots, \pi_K)$

In simple terms, we can summarize the above discussion as:

$$\text{probability that the i'th samples belongs to the class } j \text{ is } \pi_j \text{ i.e.} \\ p(\mathbf{z}_i = z_{ij}) = p(z_{ij}) = \pi_j; \forall j \in (1, \cdots, K)$$

Note This is case of multinomial distribution (https://en.wikipedia.org/wiki/Multinomial_distribution).

MF.2.2 Modelling: Draw a Sample from one Gaussian Distribution

As we have seen in the prerequisite notebook on Applied Math (AppliedMath.ipynb), once we have the probability distribution, we can draw samples out of it. Here is the pdf of multivariate Gaussian Distribution.

$$\begin{aligned} &\mathbf{x} \sim \mathcal{N}(\mu, \mathbf{\Sigma}) && \text{; a vector of rv X is drawn from Normal distribution v} \\ &p(\mathbf{x}) = \frac{1}{\sqrt{2\pi}\sqrt{|\mathbf{\Sigma}|}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)} && \text{; vector } \mathbf{x} \text{ is a sample of vector random variable and } p \end{aligned}$$

After we have selected j'th distribution, we can draw i'th sample $\mathbf{x}_i$ from Gaussian PDF $\mathcal{N}(\mu_j, \mathbf{\Sigma}_j)$ Note that we have K Gaussian Components, and j'th PDF is parametrized by $\mu_j, \mathbf{\Sigma}_j$.

Above is written using the conditional PDF as follows:

$$\begin{aligned} p(\mathbf{x}_i | \mathbf{z}_i = z_{ij}) &= \mathcal{N}(\mu_j, \mathbf{\Sigma}_j) \\ &= \frac{1}{\sqrt{2\pi}\sqrt{|\mathbf{\Sigma}_j|}} e^{-\frac{1}{2}(\mathbf{x}_i-\mu_j)^T \mathbf{\Sigma}_j^{-1}(\mathbf{x}_i-\mu_j)} \end{aligned}$$

MF.2.3 Modelling: Joint PDF of GMM

GMM is the generative process, in which we follow two steps to draw i'th sample:

1. Pick up a distribution
2. and draws a multivariate Gaussian distributed sample out of it.

MF.2.3.1 Joint PDF for one Sample.

Now to exactly determine the joint PDF of the i'th input sample, we will be using joint PDF representation in terms of conditional PDF (AppliedMath.ipynb#prob_prop) and independent PDF.

$$probability\_that\_we\_select\_j'th\_distribution\_and\_draw\_sample\_from\_j'th\_distribution =$$

=

=

we can use, defination of Multinomial and Gaussian distribution to get the complete expression. But since we will be dealing with higher level details, it is not necessary.

Note: The above model of mixing K Gaussian Distribution in various proportion is called **Mixture of Gaussian models**

MF.2.3.1 Joint PDF for N Samples.

We observe that events generated by rolling a fair dice multiple times are independent of each other and also each outcome is derived out of same distribution. This is called **i.i.d.** assumptions and is very common in machine learning. Similarly, all the N GMM samples are **i.i.d.** and hence given the PDF of one sample, we can calculate joint PDF of N samples.

Let's derive the joint PDF for N samples.

$$\begin{aligned} probability\_that\_we\_get\_N\_datapoints\_given\_its\_parameters = & \\ & = p(\mathbf{X}, \mathbf{z}) \\ & = \prod_{i=1}^{N} p(\mathbf{x}_i, \mathbf{z}_i) \\ & = \prod_{i=1}^{N} \{ \sum_{j=1}^{K} p(\mathbf{x}_i | \mathbf{z}_i = z_{ij}) p(\mathbf{z}_i \end{aligned}$$

MF.2.3.1 Final Model:Joint PDF.

Now we can write the final joint PDF of GMM for N samples.

Parameters:

$$\pi = (\pi_1, \pi_2, \cdots, \pi_j, \cdots, \pi_K)$$
$$\mu = (\mu_1, \mu_2, \cdots, \mu_j, \cdots, \mu_K)$$
$$\boldsymbol{\Sigma} = (\Sigma_1, \Sigma_2, \cdots, \Sigma_j, \cdots, \Sigma_K)$$

Gaussian Mixture Model:

$$p(\mathbf{X}) = \prod_{i=1}^{N} \left\{ \sum_{j=1}^{K} p(\mathbf{z}_i = z_{ij}) p(\mathbf{x}_i | \mathbf{z}_i = z_{ij}) \right\}$$
$$= \prod_{i=1}^{N} \left\{ \sum_{j=1}^{K} p_{multinomial}(\mathbf{z}_i = z_{ij}; \pi) p_{normal}(\mathbf{x}_i | \mathbf{z}_i = z_{ij}; \mu_{\mathbf{k}}, \boldsymbol{\Sigma}\right.$$
$$= \prod_{i=1}^{N} \left\{ \sum_{j=1}^{K} \mathcal{M}ulti(\mathbf{z}_i = z_{ij}; 1, \pi) \mathcal{N}(\mathbf{x}_i | \mathbf{z}_i = z_{ij}; \mu_j, \boldsymbol{\Sigma}_j) \right\}$$
$$= \prod_{i=1}^{N} \left\{ \sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i | \mathbf{z}_i = z_{ij}; \mu_j, \boldsymbol{\Sigma}_j) \right\}$$

MF.2.3. Training GMM.

Now, we have GMM model from input data and we want to train the model given dataset. Note that from previous sections, we found out that GMM is characterized by three parameter vectors/matrices. Learning involves finding best estimates of those parameters. Let's find out, how will we do it.

We will use the various techniques (like probability to log likelihood, maximizing a function.) from the Applied Math (AppliedMath.ipynb) section.

Here are the flow of GMM training.

- Convert Model into Log Likelihood.
- Estimating parameters using MLE fails due to intractable Log Likelihood.
- Analysis and make simplifying assumption to do iteration.
- Introduce EM and iteratively compute Expectation and Maximization.

MF.2.3.1Training GMM: Joint PDF to Log Likelihood

MF.2.3.1.1 PDF to Likelihood.

Machine learning is about modelling i.e. you have seen something and you wonder whether you can predict a new sample in future? Model can be described by a set of parameters. Those parameters could be coefficient in linear model or it could be the parameters of the probability distribution that our data sets are generated from? With this line of thinking, we can see that given the dataset, if we are interested in likelihood of the model (and its parameters), which by definition is equal to probability that data is generated from that model (and its parameters). This is the core idea that is used to find the model which best describes dataset.

In notations, the above discussion can be summarised as follows:

$$\mathbf{L}(parameters, model|Dataset) = p(Dataset|parameters, model)$$

Since, we represent our dataset as a matrix $\mathbf{X}$ where each row corresponds to one data point or data vector. and our model as the combination of Multinomial and Gaussian distribution with parameters $(\pi, \mu, \mathbf{\Sigma})$, we can simplify above expression as:

$$\begin{aligned}
\mathbf{L}(\pi, \mu, \mathbf{\Sigma}|\mathbf{X}) &= p(\mathbf{x}_i|\mathbf{z}_i = z_{ij}; \pi, \mu, \mathbf{\Sigma}) && \text{;Definition of Likelihood} \\
&= \prod_{i=1}^{N} \left\{ \sum_{j=1}^{K} p(\mathbf{z}_i = z_{ij}) p(\mathbf{x}_i|\mathbf{z}_i = z_{ij}; \pi, \mu, \mathbf{\Sigma}) \right\} && \text{;From section MF.2.3.1} \\
&= \prod_{i=1}^{N} \left\{ \sum_{j=1}^{K} p(\mathbf{z}_i = z_{ij}) p(\mathbf{x}_i|\mathbf{z}_i = z_{ij}) \right\} && \text{;Simplifying notation}
\end{aligned}$$

MF.2.3.1.2 Why Log Likelihood.

Our goal is to find parameters, which maximizes likelihood. Our expression consists of many product terms and we know from vector calculus that maximizing involves taking first and second derivatives which can get hairy.

Logarithm is convenient function for simplifying math because:

- It is the monotonic increasing function, taking logarithm of both sides, won't change the x axis values, which maximizes it.
- Logarithm converts multiplication into summation and takings its derivative is much simpler.

And hence we will take logarithm of both sides

$$\begin{aligned}
\mathbf{LL}(\pi, \mu, \mathbf{\Sigma}|\mathbf{X})) &= log(\mathbf{L}(\pi, \mu, \mathbf{\Sigma}|\mathbf{X})) && \text{;Definition of log Likeliho} \\
&= log\left(\prod_{i=1}^{N} \left\{ \sum_{j=1}^{K} p(\mathbf{z}_i = z_{ij}) p(\mathbf{x}_i|\mathbf{z}_i = z_{ij}) \right\} \right) && \text{;Likelihood of GMM} \\
&= \sum_{i=1}^{N} log\left\{ \sum_{j=1}^{K} p(\mathbf{z}_i = z_{ij}) p(\mathbf{x}_i|\mathbf{z}_i = z_{ij}) \right\} && \text{;Log Trick} \\
&= \sum_{i=1}^{N} \mathbf{LL}(\pi, \mu, \mathbf{\Sigma}|\mathbf{x}_i)) && \text{; Expressing in terms of } LL
\end{aligned}$$

Above is summarized as follows:

$$\mathbf{LL}(\pi, \mu, \mathbf{\Sigma}|\mathbf{X})) = \sum_{i=1}^{N} \mathbf{LL}(\pi, \mu, \mathbf{\Sigma}|\mathbf{x}_i)) = \sum_{i=1}^{N} log\left\{ \sum_{j=1}^{K} p(\mathbf{z}_i = z_{ij})p(\mathbf{x}_i|\mathbf{z}_i$$

MF.2.3.2 Training GMM: Partial Solution.

Typically, one using MLE Techniques (AppliedMath.ipynb#num_mle) to estimate parameter vectors of probabilistic models.

Let's attempt to estimate parameter vectors.

MF.2.3.2.1 Problem in Estimating Parameters:Intractable Log Likelihood.

Let's looks at expression for log likelihood derived $\mathbf{LL}$ in previous section.

If we try to differentiate log likelihood w.r.t each of $(\pi, \mu, \mathbf{\Sigma})$ and set to $\mathbf{0}$, we will find that we can't find its solution in the closed form.

It is due to the fact the log doesn't expand summation in the curly brackets. We also notice that there are i and j terms inside second summation and thus there is interaction happening between its terms. And as we always do if we can't find the perfect solution, we try to map problem into a known problem and then solves its value. In numerical analysis techniques, if the problems are not solvable in closed form, we use iterations i.e. we initialize to some random values and iterate until convergence.

MF.2.3.2.2 Key Observation in Intractable Log Likelihood .

Let's look at the joint model for i'th sample i.e. from section $MF.2.3.1.2$, the terms inside curly brackets:

$$\mathbf{LL}(\pi, \mu, \mathbf{\Sigma}|\mathbf{x}_i)) = log\left\{ \sum_{j=1}^{K} p(\mathbf{z}_i = z_{ij})p(\mathbf{x}_i|\mathbf{z}_i = z_{ij}) \right\}$$

To simplyfy, let's make certain assumptions as follows:

Observation 1: Selection Assumption => We know the selected distribution k'th distribution i.e. $p(\mathbf{x}_i == z_i k)$
What if, we assume that somehow some oracle has done step one for us and told us that select k'th component. With this information in hand, we might be able to simplify joint likelihood expression! First step is to select k using multinomial distribution . Since oracle has already selected this for us i.e. $\mathbf{z}_i = z_{ik}$. Also since $\mathbf{z}_i$ has 1-of-K schema, all other multinomial probability must be zero. Formally it is expressed as:

$$\mathcal{M}ulti(\mathbf{z}_i = z_{ij}; \pi) = p(\mathbf{z}_i = z_{ij}; \pi) = p(z_{ij}) = \begin{cases} \pi_k & \forall j == k \\ 0 & \forall j \neq k \end{cases}$$

Observation 2: Complement of Selection assumption:

For sample $\mathbf{x}_i$, we have selected a k'th distribution with probability $p(\mathbf{x}_i = z_i k) = \pi_k$, this means that for $\mathbf{x}_i$ probability that we don't select k'th distribution is complement of π_k. Formally, we express that as:

$$p(\mathbf{x}_i != z_{ik}) = 1 - p(\mathbf{x}_i = z_{ik}) = 1 - \pi_k$$

MF.2.3.2.3 Partial Solution: Simplifying Log Likelihood Using Generative Process.

Simplifying Log Likelihood for One Sample

With that assumption, let's look at likelihood for one sample:

$$\begin{aligned}
\mathbf{LL}(\pi, \mu, \mathbf{\Sigma}|\mathbf{x}_i)) &= log\left\{ \sum_{j=1}^{K} p(\mathbf{z}_i = z_{ij}) p(\mathbf{x}_i|\mathbf{z}_i = z_{ij}) \right\} \\
&= log\Big\{ p(\mathbf{x}_i|\mathbf{z}_i = z_{i1})p(\mathbf{z}_i = z_{i1}) + p(\mathbf{x_i}|\mathbf{z}_i = z_{i2})p(\mathbf{z}_i = z_{i2}) \cdots + p(\mathbf{x_i}|\mathbf{z}_i = z_{ik})p(\mathbf{z}_i = z_{ik}) + \cdot\cdot \\
&= log\Big\{ p(\mathbf{x}_i|\mathbf{z}_i = z_{i1}) \times 0 + p(\mathbf{x_i}|\mathbf{z}_i = z_{i2}) \times 0 \cdots + p(\mathbf{x_i}|\mathbf{z}_i = z_{ik}) \times \pi_k + \cdots + p(\mathbf{x_i}|\mathbf{z}_i = z_{iK}) \\
&= log\Big\{ p(\mathbf{x}_i|\mathbf{z}_i = z_i k) \times \pi_k \Big\} \\
&= log\Big\{ p(\mathbf{x}_i|\mathbf{z}_i = z_{ik}) p(\mathbf{z}_i = z_{ik}) \Big\}
\end{aligned}$$

And we notice that interaction terms are gone and PDF looks more tractable for MLE.

Above discussion can be summarized as log likelihood for GMM for i'th sample and pre-selecting k'th distribution:

$$\text{Log likelihood for one sample after pre-selecting mixture component:}$$
$$\mathbf{LL}(\pi, \mu, \mathbf{\Sigma}|\mathbf{x}_i)) = log\Big\{ p(\mathbf{x}_i; \pi, \mu, \mathbf{\Sigma}) \Big\} = log\Big\{ p(\mathbf{x}_i|\mathbf{z}_i = z_{ik}) p(\mathbf{z}_i = z_i$$

Simplifying Log Likelihood for N Sample

Since all N samples are i.i.d., we can extend log likelihood expression for one sample to N samples, by just adding all the log likelihood of each sample.

$$\begin{aligned}
\mathbf{LL}(\pi, \mu, \mathbf{\Sigma}|\mathbf{X})) &= \sum_{i=1}^{N} \mathbf{LL}(\pi, \mu, \mathbf{\Sigma}|\mathbf{x}_i)) = \sum_{i=1}^{N} log\Big\{ p(\mathbf{x}_i|\mathbf{z}_i = z_{ik}) p(\mathbf{z}_i = z_{ik}) \Big\} && \text{;From MF.2.3.1} \\
&= \sum_{i=1}^{N} \Big\{ log\big(p(\mathbf{x}_i|\mathbf{z}_i = z_{ik})\big) + log\big(p(\mathbf{z}_i = z_{ik})\big) \Big\} && \text{; Log trick} \\
&= \sum_{i=1}^{N} \Big\{ log\big(p(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma})\big) + log\big(p(\mathbf{z}_i = z_{ik}; \pi)\big) \Big\} && \text{; Showing para} \\
&= \sum_{i=1}^{N} \Big\{ log\big(p(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu_k, \mathbf{\Sigma}_k)\big) + log\big(p(\mathbf{z}_i = z_{ik}; \pi_k)\big) \Big\} && \text{; From MF.2.3.}
\end{aligned}$$

Above discussion can be summarized as:

Log likelihood for N sample after pre-selecting k'th Gaussian mixture

$$\mathbf{LL}(\pi_k, \mu_k, \mathbf{\Sigma}_k|\mathbf{X})) = p(\mathbf{X}; \pi_k, \mu_k, \mathbf{\Sigma}_k)$$

$$= \sum_{i=1}^{N} \left\{ log\left[p(\mathbf{x}_i|\mathbf{z}_i = z_{ij}; \mu_k, \mathbf{\Sigma}_k)\right] + log\left[p(\mathbf{z}_i =\right.\right.$$

$$= \sum_{i=1}^{N} \left\{ log\left[\mathcal{N}(\mathbf{x}_i|\mathbf{z}_i = z_{ij}; \mu_k, \mathbf{\Sigma}_k)\right] + log\left[Multi\right.\right.$$

MF.2.3.2.4 Partial Solution: Maximizing Simplified Log Likelihood.

We will now maximize it w.r.t $\pi_k, \mu_k, \mathbf{\Sigma}_k$.

MF.2.3.2.4 Partial Solution: Maximizing Simplified Log Likelihood.

Maximize w.r.t. to π_k

Since we have selected k'th component, let's maximize it wrt corresponding mixture component probability π_k We will now first differentiate log likelihood function wrt to π_k i.e. the empirical probability of the selecting k'th distribution.

$$\frac{\partial^1(\mathbf{LL}(\pi_k, \mu_k, \mathbf{\Sigma}_k|\mathbf{X}))}{\partial^1 \pi_k} = \nabla_{\pi_k}\{\mathbf{LL}(\pi_k, \mu_k, \mathbf{\Sigma}_k|\mathbf{X}))\} \qquad ;ML$$

$$= \nabla_{\pi_k}\left(\sum_{i=1}^{N} \left\{ log\left[\mathcal{N}(\mathbf{x}_i|\mathbf{z}_i = z_{ij}; \mu_k, \mathbf{\Sigma}_k)\right] + log\left[Multi(\mathbf{z}_i = z_{ij}; \pi_k)\right] \right\} \right) \qquad ;Fro$$

$$= \nabla_{\pi_k}\left(\sum_{i=1}^{N} log\left[\mathcal{N}(\mathbf{x}_i|\mathbf{z}_i = z_{ij}; \mu_k, \mathbf{\Sigma}_k)\right] \right) + \nabla_{\pi_k}\left(\sum_{i=1}^{N} log\left[Multi(\mathbf{z}_i = z_{ij}; \pi_k)\right] \right) \qquad ;Lin$$

$$= \mathbf{0} + \nabla_{\pi_k}\left(\sum_{i=1}^{N} log\left[Multi(\mathbf{z}_i = z_{ij}; \pi_k)\right] \right) \qquad ;Firs$$

$$= \nabla_{\pi_k}\left(\sum_{i=1,j==k}^{N} \log\left[Multi(\mathbf{z}_i = z_{ik}; \pi_k)\right] \right) + \nabla_{\pi_k}\left(\sum_{i=1,j!=k}^{N} log\left[Multi(\mathbf{z}_i != z_{ik}; \pi_k)\right] \right) \qquad ;Spli$$

$$= \nabla_{\pi_k}\left(\sum_{i=1,j==k}^{N} \log\left[\pi_k\right] \right) + \nabla_{\pi_k}\left(\sum_{i=1,j!=k}^{N} log\left[1 - \pi_k\right] \right) \qquad ;Fro$$

$$= \nabla_{\pi_k}\left(\sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i = z_{ik}) log(\pi_k) \right) + \nabla_{\pi_k}\left(\sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i != z_{ik})] log(1 - \pi_k) \right) \qquad ;Usi$$

$$= \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i = z_{ik}) \nabla_{\pi_k}\left\{log(\pi_k)\right\} + \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i != z_{ik}) \nabla_{\pi_k}\left\{log(1 - \pi_k)\right\} \qquad ;Lin$$

$$= \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i = z_{ik}) \left\{ \frac{1}{\pi_k} \nabla_{\pi_k}(\pi_k) \right\} + \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i != z_{ik}) \left\{ \frac{1}{(1 - \pi_k)} \nabla_{\pi_k}(1 - \pi_k) \right\} \qquad ;com$$

$$= \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i = z_{ik}) \left\{ \frac{1}{\pi_k} \times 1 \right\} + \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i != z_{ik}) \left\{ \frac{1}{(1 - \pi_k)} \times (0 - 1) \right\} \qquad ;\nabla\ o$$

$$= \left\{ \frac{1}{\pi_k} \times 1 \right\} \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i = z_{ik}) + \left\{ \frac{1}{(1 - \pi_k)} \times (0 - 1) \right\} \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i != z_{ik}) \qquad ;taki$$

$$= \frac{\left\{(1 - \pi_k)\right\}\left\{ \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i = z_{ik}) \right\} + \left\{(-\pi_k)\right\}\left\{ \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i != z_{ik}) \right\}}{\pi_k(1 - \pi_k)} \qquad ;cros$$

Above derivation is summarized as:

Differentiation of Log Likelihood:

$$\nabla_{\pi_k}\{\mathbf{LL}(\pi_k, \mu_k, \mathbf{\Sigma}_k|\mathbf{X}))\} = \frac{(1 - \pi_k)\left\{ \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i = z_{ik}) \right\} - \pi_k \left\{ \sum_{i=1}^{N} \mathbb{I}(\mathbf{z}_i != z_{ik}) \right.}{\pi_k(1 - \pi_k)}$$

Now for log likelihood to be maximum, the first derivative expression derived in previous step must be zero.

$$\nabla_{\pi_k}\left\{\mathbf{LL}(\pi_k,\mu_k,\mathbf{\Sigma}_k|\mathbf{X}))\right\} = \mathbf{0}$$

$$=> \frac{(1-\pi_k)\left\{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik})\right\} - \pi_k\left\{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i != z_{ik})\right\}}{\pi_k(1-\pi_k)} = \mathbf{0}$$

$$=> (1-\pi_k)\left\{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik})\right\} - \pi_k\left\{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i != z_{ik})\right\} = \mathbf{0}$$

$$=> 1\times\left\{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik})\right\} - \pi_k\left\{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik}) + \sum_{i=1}^N \mathbb{I}(\mathbf{z}_i \neq z_{ik})\right\} = \mathbf{0}$$

$$=> \pi_k = \frac{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik})}{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik}) + \sum_{i=1}^N \mathbb{I}(\mathbf{z}_i \neq z_{ik})}$$

Since $\mathbf{x}_i$ either belongs to cluster k or not belongs to cluster k, sum of indicators must be equal

$$=> \pi_k = \frac{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik})}{N}$$

Summary

We derived MLE estimate for π_k using our assumption of pre-selecting a particular mixture component. Similarly, we can derive optimum value of μ_k and Σ_k and they are as follows:

$$\text{Partial MLE Estimates for parameters for } k\text{'th cluster } \forall k \in (1 \cdots K).$$

$$\pi_k = \frac{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik})}{N}$$

$$\mu_k = \frac{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik})\mathbf{x}_i}{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik})}$$

$$\Sigma_k = \frac{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik})(\mathbf{x}_i - \mu_k)^T(\mathbf{x}_i - \mu_k)}{\sum_{i=1}^N \mathbb{I}(\mathbf{z}_i = z_{ik})}$$

MF.2.3.2.5 Partial Solution: Refined Estimate of mixing probability.

Refined value of π_k using computing expectation using Bayes' Theorem.

We must note that MLE value in previous section is a partial solution. The reason for that is since we assumed that we knew membership of i'th sample i.e. $\mathbf{z}_i$, but we really didn't knew it and made a guess for it and hence we need to find the way to find the complete solution using this as a starting point.

Since, now missing piece of puzzle is assignment of each sample into a particular component i.e. finding $\mathbf{z}_{ik}\forall k \in (1 \cdots K)$. We notice that in partial solution, we have indicator function of $\mathbf{z}_i = z_{ik}$. What will be the best estimate of indicator function? As usual, we will use the expectation to estimate. As explained in PT.5 of Probability Properties (AppliedMath.ipynb#prob_prop) expectation of indicator function of a set is probability of that set.

What is $p_(\mathbf{z}_i = z_{ik})$? Well we know its prior from oracle from MF.2.3.2.2 i.e. π_k. But note that pi_k was our first guess and we can refine that guess using posterior of $p(\mathbf{z}_i = z_{ik})$ as per Bayes's Theorem (AppliedMath.ipynb#prob_bayes).

Let's consider similarity in Bayes' Theorem process and GMM process.

In Bayes theorem,

- Use Belief as prior probability on parameters.
- Using prior probability, we compute the probability of likelihood of data.

In GMM generative process

- we select k'th distribution with probability π_k
- Once a Gaussian distribution is selected, draw sample out of it .

Comparing two process, we see that act of drawing from k'th Gaussian distribution is same as likelihood of data. Using this insight we can compute posterior of selection probability using Bayes' Theorem.

$$p_{posterior}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i) \propto p_{likelihood\_given\_prior}(\mathbf{x}_i|\mathbf{z}_i = z_{ik}).\, p_{prior}(\mathbf{z}_i = z_{ik}$$
$$p^{t+1}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i) \propto p^t_{likelihood\_given\_prior}(\mathbf{x}_i|\mathbf{z}_i = z_{ik}).\, p^t(\mathbf{z}_i = z_{ik})$$
$$p^{t+1}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i) \propto p^t_{\mathcal{N}}(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma}).\, p^t(\mathbf{z}_i = z_{ik})$$
$$p^{t+1}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i; \pi, \mu, \mathbf{\Sigma}) \propto p^t_{\mathcal{N}}(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma}).\, p^t(\mathbf{z}_i = z_{ik}; \pi)$$
$$p^{t+1}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i; \pi, \mu, \mathbf{\Sigma}) = \frac{1}{Z} p^t_{\mathcal{N}}(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma}).\, p^t(\mathbf{z}_i = z_{ik}; \pi)$$
$$p^{t+1}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i; \pi, \mu, \mathbf{\Sigma}) = \frac{1}{Z} p^t_{\mathcal{N}}(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma}).\, p^t(\mathbf{z}_i = z_{ik}; \pi, \mu, \mathbf{\Sigma})$$
$$p^{t+1}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i; \pi, \mu, \mathbf{\Sigma}) = \frac{p^t_{\mathcal{N}}(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma}).\, p^t(\mathbf{z}_i = z_{ik}; \pi, \mu, \mathbf{\Sigma})}{\sum_{k=1}^{K} p^t_{\mathcal{N}}(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma}).\, p^t(\mathbf{z}_i = z_{ik}; \pi, \mu,}$$

Notes:

- Since this probability also intuitively means responsibility of k'th cluster for sample $\mathbf{x}_i$, we have:
$$\text{responsibility } = r_k^{t+1}(\pi, \mu, \mathbf{\Sigma}) = p^{t+1}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i; \pi, \mu, \mathbf{\Sigma})$$
- Since this probability can also be viewed as weight for sample $\mathbf{x}_i$, we have:
$$\text{weight } = \omega_k^{t+1}(\pi, \mu, \mathbf{\Sigma}) = p^{t+1}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i; \pi, \mu, \mathbf{\Sigma})$$
- Since weight or responsibility is nothing but best estimate of indicator $\mathbb{I}(\mathbf{z}_i = z_{ik})$ for k'th cluster, we have:
$$\mathbb{I}(\mathbf{z}_i = z_{ik}) = p^{t+1}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i; \pi, \mu, \mathbf{\Sigma})$$

Since, we took best guess of indicator function as an expectation of the posterior of dataset based on k'th Gaussian likelihood and prior, this is called **Expectation computation**.

Summary

We refined our estimate of mixing components (i.e. π vector) using the partial MLE estimates of parameters. Since we are computing expectation. We call this act **expectation** computation.

Note: Alternate names for $p^{t+1}(\mathbf{z}_i = z_{ik}|\mathbf{x}_i; \pi, \mu, \mathbf{\Sigma})$

- Since this probability also intuitively means responsibility of k'th cluster for sample $\mathbf{x}_i$, we have:
$$\text{responsibility of } k\text{'th cluster for } i\text{'th sample } = \gamma_k^{t+1}(i; \pi, \mu, \mathbf{\Sigma}) =$$
- Since this probability can also be viewed as weight for sample $\mathbf{x}_i$, we have:

$$\text{weight of } i\text{'th sample in cluster } k = \omega_k^{t+1}(i; \pi, \mu, \mathbf{\Sigma}) = p^{t+1}(\mathbf{z}_i = z_{i}$$

Expected weights or responsibility or mixing probability for k'th

$$\mathbb{I}(\mathbf{z}_i = z_{ik}) = \gamma_k^{t+1}(i; \pi, \mu, \mathbf{\Sigma}) = \omega_k^{t+1}(i; \pi, \mu, \mathbf{\Sigma}) = p$$
$$= \frac{p^t_{\mathcal{N}}(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma}).\, p^t(\mathbf{z}_i = z_{ik}; \pi, \mu, \mathbf{\Sigma})}{\sum_{k=1}^{K} p^t_{\mathcal{N}}(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma}).\, p^t(\mathbf{z}_i = z_{ik}; \pi, \mu, \mathbf{\Sigma})}$$

MF.2.3.2 Training GMM: Using EM Algorithm for Estimating Paran

EM algorithm (AppliedMath.ipynb#num_em) is commonly used iterative techniques for estimating the parameters of intractable distribution using iterations of expectation and maximization.

Here is outline of EM algorithm

```
**Conventions :**
    Let's use subscript to denote component count i.e. $\mu_k$ is $k$'th component of $\mu$
    Let's use superscript to denote iteration count i.e. $\mu^t$ is $\mu$ at iteration $t$
**Initializations :**
    $\pi^{\mathbf{0}} = (\pi_1^0, \pi_2^0, \cdots, \pi_K^0) = \text{random\_init()}$
    $\mu^{\mathbf{0}} = ({\mu_{\mathbf{1}}}^0, {\mu_{\mathbf{2}}}^0, \cdots, {\mu_{\mathbf{K}}}^0) = \text{random\_init()}$
    $\mathbf{\Sigma}^{\mathbf{0}} = ({\mathbf{\Sigma_1}}^0, {\mathbf{\Sigma_2}}^0, \cdots, {\mathbf{\Sigma_K}}^0) = \text{random\_init()}$
    $t = 0$
**EM Iterations :**
    **LoopUntilConvergence :**
        **E – Step :**
            $\gamma^{\mathbf{t+1}} = expectation(\mathbf{X}, \pi^{\mathbf{t}}, \mu^{\mathbf{t}}, \mathbf{\Sigma}^{\mathbf{t}})$
        **M – Step :**
            $(\pi^{\mathbf{t+1}}, \mu^{\mathbf{t+1}}, \mathbf{\Sigma}^{\mathbf{t+1}}) = maximization(\mathbf{X}, \gamma^{\mathbf{t+1}})$
        $t = t + 1$
```

With this and result of expectation and maximization (see MF.2.3.2.4 and MF.2.3.2.5) at each iteration, we can now write the complete EM algorithm for GMM as:

```
**Conventions :**
    ;Let's use subscript to denote component count i.e. $\mu_k$ is $k$'th comp
    ;Let's use superscript to denote iteration count i.e. $\mu^t$ is $\mu$ at iteratio
**Variables :**
    ;At $t$'th iteration, let parameters be denoted:
        $\pi^{\mathbf{t}} = (\pi_1^t, \pi_2^t, \cdots, \pi_K^t)$
        $\mu^{\mathbf{t}} = ({\mu_{\mathbf{1}}}^t, {\mu_{\mathbf{2}}}^t, \cdots, {\mu_{\mathbf{K}}}^t)$
        $\mathbf{\Sigma}^{\mathbf{t}} = ({\mathbf{\Sigma_1}}^t, {\mathbf{\Sigma_2}}^t, \cdots, {\mathbf{\Sigma_K}}^t)$
    ;At $t$'th iteration, let responsibility each of $K$ cluster be denoted:
        $\gamma_k^t = (\gamma_{1k}^t, \gamma_{1k}^t, \cdots, \gamma_{Nk}^t) \; \forall k \in (1 \cdots K)$
        $\gamma^t = (\gamma_1^t, \gamma_2^t, \cdots, \gamma_K^t); \gamma^t$ is $K \times N$ matrix
**Initialize :**
    $\forall k \in (1 \cdots K)$ :
        $\gamma_k^0 = (\gamma_{1k}^0, \gamma_{1k}^0, \cdots, \gamma_{Nk}^0) = \text{random\_init()}$
    $\pi^{\mathbf{0}} = (\pi_1^0, \pi_2^0, \cdots, \pi_K^0) = \text{random\_init()}$
    $\mu^{\mathbf{0}} = ({\mu_{\mathbf{1}}}^0, {\mu_{\mathbf{2}}}^0, \cdots, {\mu_{\mathbf{K}}}^0) = \text{random\_init()}$
    $\mathbf{\Sigma}^{\mathbf{0}} = ({\mathbf{\Sigma_1}}^0, {\mathbf{\Sigma_2}}^0, \cdots, {\mathbf{\Sigma_K}}^0) = \text{random\_init()}$
    $t = 0$
```

```
EM Iterations :
    LoopUntilConvergence :
        E – Step :
            $\gamma^{\mathbf{t+1}} = expectation(\mathbf{X}, \pi^{\mathbf{t}}, \mu^{\mathbf{t}}, \mathbf{\Sigma^t})$
        M – Step :
            $(\pi^{\mathbf{t+1}}, \mu^{\mathbf{t+1}}, \mathbf{\Sigma^{t+1}}) = maximization(\mathbf{X}, \gamma^{\mathbf{t+1}})$
        $t = t + 1$

Function $expectation(\mathbf{X}, \pi^t, \mu^t, \mathbf{\Sigma}^t)$ :
    for k in $(1 \cdots K)$:
        for i in $(1 \cdots N)$:
            $\gamma_{ik}^{t+1} = \frac{p_{\mathcal{N}}^t(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma}).\, p^t(\mathbf{z}_i = z_{ik}; \pi, \mu, \mathbf{\Sigma})}{\sum_{k=1}^K p_{\mathcal{N}}^t(\mathbf{x}_i|\mathbf{z}_i = z_{ik}; \mu, \mathbf{\Sigma}).\, p^t(\mathbf{z}_i = z_{ik}; \pi, \mu, \mathbf{\Sigma})}$; comp
    return $\gamma^{t+1}$

Function $maximization(\mathbf{X}, \gamma^{t+1}, \pi^t, \mu^t, \mathbf{\Sigma}^t)$ :
    for k in $(1 \cdots K)$:
        $\pi_k^{t+1} = \frac{\sum_{i=1}^N \gamma_{ik}^{t+1}}{N}$
        $\mu_k^{t+1} = \frac{\sum_{i=1}^N \gamma_{ik}^{t+1}\mathbf{x}_i}{\sum_{i=1}^N \gamma_{ik}^{t+1}}$
        $\Sigma_k^{t+1} = \frac{\sum_{i=1}^N \gamma_{ik}^{t+1}(\mathbf{x}_i - \mu_k^t)^T(\mathbf{x}_i - \mu_k^t)}{\sum_{i=1}^N \gamma_{ik}^{t+1}}$
```

We have just derived the maximization expression for all terms and also the expectation terms in previous sections. This allows us write the complete EM algorithm for GMM as follows:

EX. GMM in Practice.

We have seen, intuition and mathematical formulation of Gaussian Mixture Model. Now its time to play around with real dataset to get more insights and practical application. We will be using Scikit-learn (http://scikit-learn.org/stable/) and matplotlib (https://matplotlib.org/) to dive deep into examples

EX.1 Utilities Functions.

Let's load relevant library.

In [1]:
```
# to make sure that notebooks are plotting
%matplotlib notebook
```

In [2]:
```



# numpy and scipy
import numpy as np
import scipy as sp
from scipy.stats import norm

# plotting utilities
import matplotlib.pyplot as plt

# set random seed for reproducibility
np.random.seed(42)
import random
random.seed(42)
```

EX.1.1 Utilities for Image Processing.

Let's develop multiple image processing utilities. This will be used multiple times in subsequence sections

Normalize Image

Many scikit learn algorithms works on the image which are each pixel is in the range of 0-255. This utility allow one to normalize any image into the range.

Scale Down Image

Since this document is the tutorial, we would like our code to execute in a reasonable time and this utility allow one to scale down image so that subsequent image processing runs faster.

Plot multiple images

Since plotting multiple images is a repetitive task, this utils take care of it.

In [3]:
```
class ImageUtils(object):

    @staticmethod
    def normalize(image):
        # make sure that image pixel is in the range 0-255 ( a normal level
        """
        Convert to floats instead of the default 8 bits integer coding. Divi
        255 is important so that plt.imshow behaves works well on float data
        be in the range [0-1])
        """
        norm_image= np.array(image, dtype=np.float64) / 255
        return norm_image

    @staticmethod
    def scaledown(image, size_pct = 0.2, sigma = 2):
        """
        Resize it to size_pct of the original size to speed up the processin
        Applying a Gaussian filter for smoothing prior to down-scaling
        reduces aliasing artifacts.
        """
        smoothed_image = gaussian_filter(image, sigma)
        rescaled_smooth_image = rescale(smoothed_image, size_pct, mode="refl
        return rescaled_smooth_image



    def dimensions(im):
        """
        In Scikit-learn images can be a tuple of 2d array i.e. black and whi
        3d array i.e. rgb image.This utils returns dimensions of image
        """
        w,h,d = (None, None, None)
        if len(im.shape) == 2:
            w,h = im.shape
        else:
            w,h,d = im.shape
        return (w,h,d)

    def image_to_npimage(im, dims):
        """
        Convert, raw image into numpy 2d array to be used in processing
        """
        im_2d = None
        w,h,d = dims
        #print("w,h,d=",w,h,d)
        if d:
            im_2d = im.reshape((w*h, d))
        else:
            im_2d = im.reshape((w*h, 1))
        return im_2d


    def npimage_to_image(arr, dims):
        "Convert 2d numpy image to standard image"
        im = None
        w,h,d = dims
        #print("w,h,d=",w,h,d)
        if d:
            im = arr.reshape(w, h, d)
        else:
            im = arr.reshape(w, h)
        return im



    @staticmethod
    def plot(image_infos, ncols=1, nrows=None, width=None, height=None, cmap
        """
        Plot multiple images in nrows X ncols grid.
        @image_infos: an array of image_info. size of array is nrows X ncols
                  image_info: a dictionary of following
                      image: 2D or 3D numpy array  representing images
                      title: title of image
        It returns nrows X ncols numpy array of axes corresponding to each i
        """
        import math

        image_infos = image_infos if isinstance(image_infos, (list,)) else 
        num_images = len(image_infos)
        nrows = math.ceil(num_images/ncols) if not nrows else nrows
        figsize = (width, height) if width and height else None
        fig, axndarr = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsiz

        #fig, axarr = plt.subplots(nrows=nrows, ncols=ncols, figsize=(width,
        axarru = axndarr.ravel()
        #print("nrows,ncols=", nrows, ncols)
        for y in range(nrows):
            for x in range(ncols):

                idx = x+y*ncols
                #print("x, y, idx, len ", x, y, idx, len(image_infos))
                ax = axarru[idx]
                #print(axarru)
                ax.set_xticks([])
                ax.set_yticks([])
                ax.grid(False)
                if idx >= len(image_infos):
                    continue
                ax = axarru[idx]
                image_info = image_infos[idx]

                title = image_info['title'] if 'title' in image_info else "'
                image = image_info['image'] if 'image' in image_info else No
                if cmap:
                    ax.imshow(image, cmap)
                else:
                    ax.imshow(image)
                ax.set_title(title)


        plt.show()
        return axndarr
```

```
class ImageUtils_Test(object):
    @staticmethod
    def test_all():
        ImageUtils_Test.test_plot()

    @staticmethod
    def test_plot():

        test_plot_images = False # change to test for testing
        if test_plot_images:
            im=load_sample_image("china.jpg")
            ImageUtils.plot( ncols=2, width=10, height=10,
                        image_infos=[
                                    {'image':im, 'title':'China Building'},
                                    {'image':im},
                                    {'image':im},
                                    {'image':im},
                                    {'image':im},
                                    {'image':np.empty(shape=(400,500))}
                              ])

            ImageUtils.plot(nrows=1, ncols=1, width=5, height=5,
                        image_infos={'image':im, 'title':'China Building'},
                        )

            ImageUtils.plot({'image':im, 'title':'China Building'}, )



#ImageUtils_Test.test_all()
```

EX.1.2 Utilities for Plotting 3D.

We create convenient utility for plotting 3d data.

Surface data creation.

Since most of the 3d plot in matplotlib is conveniently done using the meshgrid. Given the function for z, this utility provides a way to create the 3d numpy array for meshgrid.

Surface

Convenient utility for plotting 3d meshgrid data.

In [4]:
```
class PlotUtils(object):

    @staticmethod
    def create_surface_data(xy_box, z_calc):


        x_min = xy_box['x_min']
        x_max = xy_box['x_max']
        y_min = xy_box['y_min']
        y_max = xy_box['y_max']

        x_cnt = (x_max-x_min)*10
        y_cnt = (y_max-y_min)*10

        #x, y = np.mgrid[x_min:x_max:30j, y_min:y_max:30j]

        xv = np.linspace(x_min, x_max, x_cnt)
        yv = np.linspace(y_min, y_max, y_cnt)
        x,y = np.meshgrid(xv,yv)

        xy = np.column_stack([x.flat, y.flat])
        z = z_calc(xy)
        z = z.reshape(x.shape)
        return (x, y, z)

    @staticmethod
    def surface(axis, x, y, z, **kwargs):
        print(type(kwargs))
        assert(x.shape[0] == y.shape[0])
        assert(x.shape[1] == y.shape[1])
        assert(x.shape[0] == z.shape[0])
        assert(x.shape[1] == z.shape[1])

        print(kwargs)
        return axis.plot_surface(x,y,z, **kwargs)





class PlotUtils_Test(object):
    @staticmethod
    def test_all():
        PlotUtils_Test.test_surface()

    @staticmethod
    def test_surface():

        do_test = True # change to test for testing
        if not do_test:
            return

        mu = np.array([0.0, 0.0])

        sigma = np.array([.5, 1.0])
        covariance = np.diag(sigma**2)
        def mvn_pdf(xy):
            return mvn.pdf(xy, mean=mu, cov=covariance)

        #x,y,z = PlotUtils.create_surface_data(
        #            {'x_min': -3, 'x_max': 3, 'y_min': -3, 'y_max': 3},
        #            lambda xy: mvn.pdf(xy, mean=mu, cov=covariance)
        #        )

        x,y,z = PlotUtils.create_surface_data(
            {'x_min': -3, 'x_max': 3, 'y_min': -3, 'y_max': 3},mvn_pdf
        )

        (width, height) = (7,5)
        figsize = (width, height) if width and height else None
        fig = plt.figure(3, figsize=figsize)
        ax = fig.add_subplot(1,1,1, projection='3d')
        PlotUtils.surface(ax,x,y,z,rstride=1, cstride=1,
                          cmap="hot",
                          edgecolor='none',alpha=0.99)
        #ax.plot_surface(x,y,z, rstride=1, cstride=1,
        #                     edgecolor='none',alpha=0.99, cmap="hot")


        plt.show()


#PlotUtils_Test.test_all()
```

EX.2 GMM Used Cases.

We will consider following use cases.

- **Generate a face of non existent person**

  GMM being generative model is often used to generate samples which we haven't observed. We will use olivetti face dataset to generate a new person's face, who doesn't exists.

- **Time Series Analysis**

  Time Series occurs very commonly in Machine learning and we use GMM to predict a toy time series. The example can be extended to other things like stocks etc.

EX.2.1 GMM as Generative Model: Create Non-Existent

GMM is a generative model and hence it doesn't need any labelled data set and it can be used for **generating new data set that has not been seen** and it generative property makes it very useful for many practical applications.

We will illustrate this idea with Olivetti Faces dataset which has faces of 400 individual with different expressions.

In [5]:
```
# Load the relevant library for this sub-section

#olivetti faces dataset
from sklearn.datasets import fetch_olivetti_faces

# for reproducibility
from numpy.random import RandomState

# GMM model
from sklearn.mixture import GaussianMixture

# for performance testing we need time
from time import time
```

EX.2.1.1 Exploring Input Datasets

To get the feel of for datasets, let's see first 100 images which we will use for training.

In [6]:
```
#n_row, n_col = 2, 3
n_row, n_col = 40, 10
n_train_samples = n_row * n_col
image_shape = (64, 64)
rng = RandomState(0)
dataset = fetch_olivetti_faces(shuffle=True, random_state=rng)
faces = dataset.data
n_samples, n_features = faces.shape

def centering(data):
    """
    Since most of the ML algorithms perform good for standardized dataset,
    This utility allows to center the dataset globally and locally
    """
    #global centering i.e. along columns i.e. we substract mean of each
    # column with elements along the axis
    # in other words, we are getting mean of each samples
    data_centered = data - data.mean(axis=0)

    # local centering i.e. along row i.e. for each image, we shift it
    # with mean along the row.
    data_centered -= data_centered.mean(axis=1).reshape(n_samples, -1)

    return data_centered

###############################################################################

def plot_gallery(title, images, wtitle="", n_col=n_col, n_row=n_row, scale=3
    #plt.figure(figsize=(2. * n_col, 2.26 * n_row))
    #plt.figure(figsize=(1. * n_col, 1.26 * n_row))
    fig = plt.figure(figsize=(scale * n_col, scale * n_row))
    fig.canvas.set_window_title(wtitle)
    plt.suptitle(title, size=16)
    for i, comp in enumerate(images):
        plt.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
        plt.imshow(comp.reshape(image_shape), cmap=plt.cm.gray,
                   interpolation='nearest',
                   vmin=-vmax, vmax=vmax)
        plt.xticks(())
        plt.yticks(())
    plt.subplots_adjust(0.01, 0.05, 0.99, 0.93, 0.04, 0.)


faces_centered = centering(faces)

###############################################################################
# Plot the input data
n_display_samples = int(n_train_samples*0.75)
plot_gallery(title="Dataset: Centered Olivetti faces ", images=faces_centere
###############################################################################
plt.show()
```

EX.2.1.2 Model Training

Let's train Gaussian Mixture Models with 8 components and full covariance type. Using full covariance gives us more better model although it is somewhat slower.

In [7]:

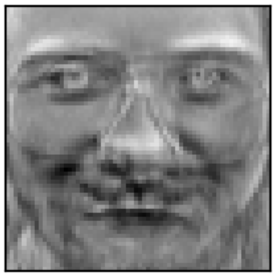
```
# use full covariance for better model. Other good options is tied
covariance_type = "full" # full, tied, diag, spherical

# number of gaussian components
n_gaussians = 24

# instantiate GMM model
gmm = GaussianMixture(
        n_components = n_gaussians,
        covariance_type=covariance_type,
        tol=1e-2,
        max_iter=300,
      )


t0 = time()
sampled_faces_centered = faces_centered[:n_train_samples]
gmm.fit(sampled_faces_centered)
t1 = time()

print("Model Training Took ", t1-t0, " seconds")
```

```
Model Training Took  212.94789695739746  seconds
```

EX.2.1.3 Generating New Data i.e. New Person Image

Let's train Gaussian Mixture Models with 8 components and full covariance type. Using full covariance gives us more better model although it is somewhat slower.

In [8]:
```
t0 = time()

new_faces= gmm.sample(1)

t1 = time()

plot_gallery(wtitle="Generated Olivetti faces", title="", images=new_faces[(

plt.show()

print("Model New Sample Creation Took ", t1-t0, " seconds")
```



```
Model New Sample Creation Took  568.6313271522522  seconds
```

We can see the new generated face doesn't exists in the olivetti dataset.

EX.2.2 GMM for predicting Time Series (e.g Stocks, Sig

Time Series modelling and prediction has many applications. Typically, one uses Markov Chain (https://en.wikipedia.org/wiki/Markov_chain) for time series. However, Since GMM is so diverse, we can use it to model time series.

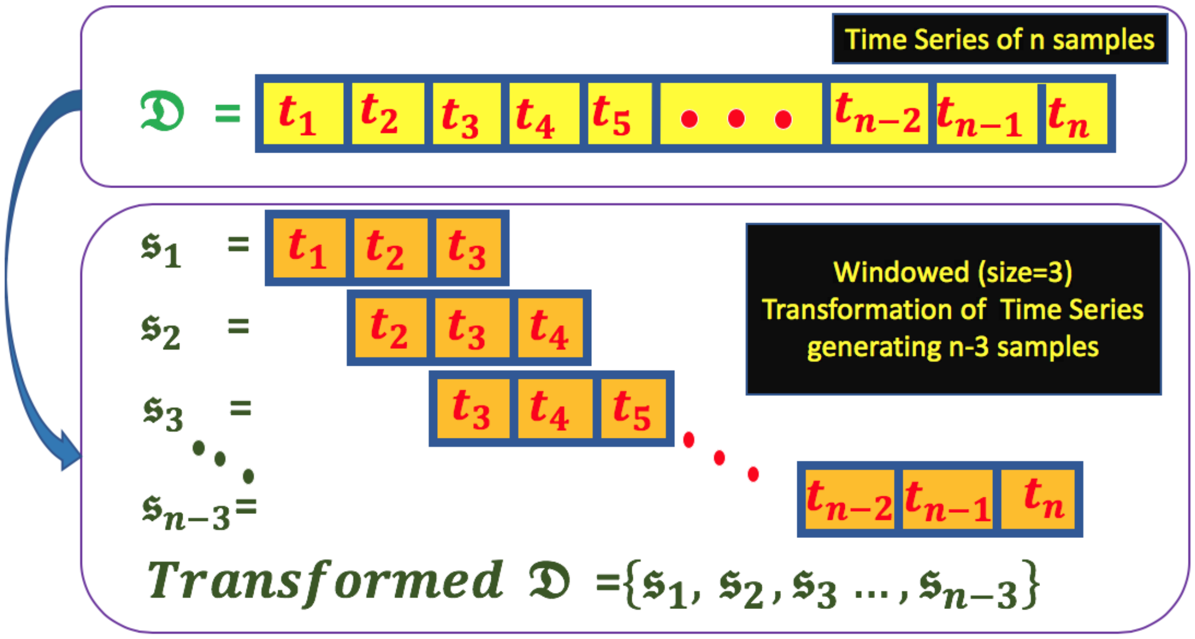
We will be following the idea from this paper on GMM for Time Series Modelling, Forecasting, and Interpolation (https://pdfs.semanticscholar.org/5a46/a53e2255d271b337f7a8271350a9cf6c7a02.pdf) by Eirola and Lendasse.

We will illustrate the concept and then deep dive into code. Note that we will use the toy dataset but the idea can be applied to other time series datasets like signals from equipments (radar, ecg), stock markets etc.

EX.2.2.1 Mixture Model for Time Series: Transforming 1d Time Seri

The core idea for modelling Time Series is similar to Markov Chain (which has memory of last M states). We can do that, by preprocessing the data set in the sliding window manner of width M.

The above idea is illustrated in the following figure.


Time Series of n samples
$\mathfrak{D} = t_1\ t_2\ t_3\ t_4\ t_5 \ \bullet\bullet\bullet\ t_{n-2}\ t_{n-1}\ t_n$
Windowed (size=3) Transformation of Time Series generating n-3 samples
$\mathfrak{s}_1 = t_1\ t_2\ t_3$
$\mathfrak{s}_2 = t_2\ t_3\ t_4$
$\mathfrak{s}_3 = t_3\ t_4\ t_5$
$\mathfrak{s}_{n-3} = t_{n-2}\ t_{n-1}\ t_n$
Transformed $\mathfrak{D} = \{\mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3 \ldots, \mathfrak{s}_{n-3}\}$

Above can be achieved by following code:

In [9]:
```


def timeSeries_2_SlidingWindow_DataSet(np_y, dims):
    """
    Convert 1d time series dataset into 2d time series using sliding window
    @args =>
        np_y: 1d numpy array of time series
        dims: width of the sliding window
    @return =>
        2d numpy array containing n-w samples and each row corresponds to di
    """
    y = np_y.tolist()
    r = []
    for i in range(len(y)-dims):
        cy = y[i:i+dims]
        r.append(cy.tolist())
    return np.array(r)
```

EX.2.2.2 Training Time Series Model

Once we have re-formatted the data as explained in previous section, we can just instantiate GMM from scikit learn and run the fit method to train our GMM model.

These can be achieved by following code.

In [10]:

```


def train_GMM_TS_Model(slidingWindow_ds, n_gaussians=16, covariance_type="fu
    """
    @args =>
        slidingWindow_ds: transformed time series dataset as explained in pr
    @return =>
        fitted GMM model
    """
    # instantiate GMM model with full 16 component with full variance
    gmm = GaussianMixture(
            n_components = n_gaussians,
            covariance_type=covariance_type,
            max_iter=max_iter,
            tol=tol
          )

    gmm.fit(slidingWindow_ds)
    return gmm
```

EX.2.2.3 Predicting Time Series.

Once we have trained GMM model. We can use it to predict short-to-medium term time series. To understand the details, we recommend use to see Section (2.2) of GMM of Time Series Paper (https://pdfs.semanticscholar.org/5a46/a53e2255d271b337f7a8271350a9cf6c7a02.pdf).

The core idea for prediction is as follows:

- Since GMM is mixture of Gaussians. Each Gaussian is characterized by its mean vector and covariance vector. So we can split mean vector into given mean vector (i.e. past) and predicted mean vector (i.e. future).
- Similarly, we can split the rectangular covariance matrix into four component i.e. $\mathbb{cov}(\mathbb{past}, \mathbb{past}), \mathbb{cov}(\mathbb{past}, \mathbb{future}), \mathbb{cov}(\mathbb{future}, \mathbb{past}), \mathbb{cov}(\mathbb{future}, \mathbb{future})$
- We can compute the future probability using conditional PDF for Multivariate Gaussain as Conditional PDF from joint PDF (AppliedMath.ipynb#prob_dist_mvn)

The above logic can be implemented by following code

In [11]:
```
def partition_stats(gmm, ts_seg, X):
    """
    Partition the mean and covariance matrix into past and future components
    @args =>
        gmm: fitted Gaussian Mixture Model
        ts_seg: original time series data
        X: transformed 2d time series data on which gmm was trained
    @return =>
        an array corresponding to each of the components in the gmm.
        Each array contains
            "mu_k_p" = past component of mean vector for kth comp
            "mu_k_f" = future component of mean vector for kth comp
            "cov_k_pp" = past and past component of cov matrix for kth comp
            "cov_k_pf" = past and future component of cov matrix for kth co
            "cov_k_fp" = future and past component of cov matrix for kth co
            "cov_k_ff" = future and future component of cov matrix for kth c
    """
    ts_seg_dim = ts_seg.shape[0]
    fit_dim = self.X.shape[1]
    fut_dim = fit_dim - ts_seg_dim

    #DEBUG print(ts_seg_dim, fut_dim)
    mu = self.gmm.means_
    cov = self.gmm.covariances_
    part_stats = []
    for k in range(mu.shape[0]):
        mu_k = mu[k]
        cov_k = cov[k]
        mu_k_p = mu_k[:ts_seg_dim]
        mu_k_f = mu_k[ts_seg_dim:]
        cov_k_pp = cov_k[:ts_seg_dim,:ts_seg_dim]
        cov_k_pf = cov_k[:ts_seg_dim,ts_seg_dim:]
        cov_k_fp = cov_k[ts_seg_dim:,:ts_seg_dim]
        cov_k_ff = cov_k[ts_seg_dim:,ts_seg_dim:]
        stats = {}
        stats["mu_k_p"] = mu_k_p
        stats["mu_k_f"] = mu_k_f
        stats["cov_k_pp"] = cov_k_pp
        stats["cov_k_pf"] = cov_k_pf
        stats["cov_k_fp"] = cov_k_fp
        stats["cov_k_ff"] = cov_k_ff
        part_stats.append(stats)

    return part_stats

def predict(gmm, ts_seg):
    """
    Predict using the GMM techniques outlines in paper
    @args =>
        gmm: fitted Gaussian Mixture Model
        ts_seg: transformed 2d time series data on which gmm was trained
    @return =>
        predicted values of the time series
    """
    part_stats = self.partition_stats(ts_seg)

    # compute the final probability as per section 2.2 of paper by Eirola or
    # also see the section AppliedMath.ipynb#prob_dist_mvn in companion note
    ts_seg_f = []
    for k, stats in enumerate(part_stats):
        mu_k_p = stats["mu_k_p"]
        ts_seg_k_f = stats["mu_k_f"]
        cov_k_pp_inv = np.linalg.inv(stats["cov_k_pp"])
        cov_k_fp = stats["cov_k_fp"]
        ts_seg_shift = ts_seg - mu_k_p
        ts_seg_k_f += np.matmul(np.matmul(cov_k_fp, cov_k_pp_inv), ts_seg_sh
        ts_seg_f.append(ts_seg_k_f)

    np_ts_seg_f = np.array(ts_seg_f)

    #mu_k_f = part_stats[0]["mu_k_f"]
    mu_k_f = np.full(shape=part_stats[0]["mu_k_f"].shape, fill_value=0)
    ts_seg_ext = np.hstack((ts_seg,mu_k_f))
    np_probas = self.gmm.predict_proba([ts_seg_ext])
    ts_seg_f = np.dot(np_ts_seg_f.T, np_probas.T)
    return ts_seg_f
```

EX.2.2.4 Time Series Dataset.

Now we are ready to run our code but before we do it we would like to have some dataset. For this notebook, we will use toy dataset as generating by following function.

In [12]:
```
class TimeSeriesData(object):

    @classmethod
    def get_sine(cls, n=1000, pseudo_freq=2):
        """
        Generate the dataset for time series.
        This function also converts 1d ts to 2d ts and split it into train a
        @args =>
            n: number of samples to generate.
            pseudo_freq: number of waves since unit doesn't match we call it
        @return => (x, y)
            x: time series x co-ord
            y: time series y co-ord
        """

        x = np.linspace(0,pseudo_freq*2*np.pi, n)
        y = np.sin(x)

        return (x, y)

    @classmethod
    def get_cosine(cls, n=1000, pseudo_freq=2):
        """
        Generate the dataset for time series.
        This function also converts 1d ts to 2d ts and split it into train a
        @args =>
            n: number of samples to generate.
            pseudo_freq: number of waves since unit doesn't match we call it
        @return => (x, y)
            x: time series x co-ord
            y: time series y co-ord
        """
        x = np.linspace(0,pseudo_freq*2*np.pi, n)
        y = np.cos(x)

        return (x, y)

    @classmethod
    def get_sine_with_noise(cls, n=1000, pseudo_freq=2):
        """
        Generate the dataset for time series.
        This function also converts 1d ts to 2d ts and split it into train a
        @args =>
            n: number of samples to generate.
            pseudo_freq: number of waves since unit doesn't match we call it
        @return => (x, y)
            x: time series x co-ord
            y: time series y co-ord
        """
        x, y = cls.get_sine(n, pseudo_freq)

        n = np.zeros(shape=y.shape)
        # introduce a spike to have complicated data
        n[50] = 1
        y = y+n
```

```
            return (x, y)

        @classmethod
        def get_random_wave(cls, n=1000):
            """
            Generate the dataset for time series.
            This function also converts 1d ts to 2d ts and split it into train a
            @args =>
                n: number of samples to generate.
                pseudo_freq: number of waves since unit doesn't match we call it
            @return => (x, y)
                x: time series x co-ord
                y: time series y co-ord
            """
            # using the idea of decomposition similar to fourier transform add
            (x, y) = (np.zeros(n), np.zeros(n))
            for pseudo_freq in (1, 1.5, 2, 2.5, 3, 3.5):
                # note all the func call returns same x
                xi, yi = cls.get_sine(n, pseudo_freq)
                x = xi
                y = y + yi

                #xi, yi = cls.get_cosine(n, pseudo_freq)
                #x = xi
                #y = y + yi

            return (x, y)


        @classmethod
        def get_random_walk(cls, n=1000):
            # must create smooth version
            #TBD
            base = 100
            x = np.linspace(0,n, n)
            def random_step():
                step_size = 1.1
                rstep = None
                if np.random.uniform(0, 1, 1) < 0.5:
                    rstep = -1*step_size
                else:
                    rstep = 1*step_size
                return rstep

            random_walk = list()
            random_walk.append(base)
            for i in range(n-1):
                random_walk.append(random_walk[-1]+random_step())

            y = np.array(random_walk)
            return(x, y)

_test = False

if _test:
    #x,y = TimeSeriesData.get_cosine(n=1000, pseudo_freq=1)
    #x,y = TimeSeriesData.get_random_wave()
    x,y = TimeSeriesData.get_random_walk(n=100)

    print(x.shape, y.shape)
    fig, ax = plt.subplots()

    plt.plot(x,y, label="time series data")
    plt.legend(loc='best')
    plt.show()
```

EX.2.2.3 GMM Time Series Demo.

We have examined all the sub components of this code. Let's combine it together and we can get our code running

In [13]:
```
import numpy as np
from sklearn.mixture import GaussianMixture

import time

class TimeSeriesGaussianMixture(object):
    """
        Implement Time Series Modelling and Prediction Using GMM.
        Reference Paper is https://pdfs.semanticscholar.org/5a46/a53e2255d27
    """
    def __init__(self, win_size,n_gaussians=16, *args, **kwargs):
        """
        @args =>
            self: instance of TimeSeriesGaussianMixture
            win_size: window size of the sliding window algorithm to convert
            ts_x: x co-ord of time series
            ts_y: y co-ord of time series

        @fields=>
            ts: time series internal representation
            ts_dims: window size of the sliding window algorithm to convert
            ts_x:x co-ord of time series
            ts_y:y co-ord of time series
            trained:boolean indicating if models were trained
            gmm:an instance of Gaussian Mixture Model scikit class
        """
        self.ts = np.array([])
        self.ts_dims = win_size
        self.ts_x = np.array([])
        self.ts_y = np.array([])
        self.trained = False
        self.X = np.array([[]]) # design matrix

        # number of gaussian component to use in mixture model is n_gaussia
        # covariance type to use: full is slow but gives better result
        covariance_type = "full"

        # Gaussian Mixture Model scikit class
        self.gmm = GaussianMixture(n_components = n_gaussians,
                                   covariance_type=covariance_type,
                                   max_iter=300
                                  )
    def ts2X(self, np_y, dims):
        """
        Convert 1d time series dataset into 2d time series using sliding wi
        @args =>
            self: instance of TimeSeriesGaussianMixture
            np_y: 1d numpy array of time series
            dims: width of the sliding window
        @return =>
            2d numpy array containing n-w samples and each row corresponds t
        """
        y = np_y.tolist()
        r = []
        for i in range(len(y)-dims):
            cy = y[i:i+dims]
            r.append(cy)
        return np.array(r)


    def fit(self, ts):
        """
        Fit Gaussian Mixture Model on 2d time series dataset.
        @args =>
            self: instance of TimeSeriesGaussianMixture
            ts:  time series data set
        """
        self.ts_y = ts
        X = self.ts2X(ts, self.ts_dims)
        self.X = X
        self.trained = True
        r = self.gmm.fit(X)
        print("Fitted Gaussian Mixture Model with convergence = ", self.gmm.
        return r

    def predict(self, ts_seg):
        """
        Predict using the GMM techniques outlines in paper
        @args =>
            self: instance of TimeSeriesGaussianMixture
            gmm: fitted Gaussian Mixture Model
            ts_seg: transformed 2d time series data on which gmm was trainec
        @return =>
            predicted values of the time series

        """
        if not self.trained:
            print("model is not been trained")
            return


        part_stats = self._partition_stats(ts_seg)

        # Compute Eirola Paper's eqn 14
        ts_seg_f = []
        for k, stats in enumerate(part_stats):
            mu_k_p = stats["mu_k_p"]
            ts_seg_k_f = stats["mu_k_f"]
            cov_k_pp_inv = np.linalg.inv(stats["cov_k_pp"])
            cov_k_fp = stats["cov_k_fp"]
            ts_seg_shift = ts_seg - mu_k_p
            ts_seg_k_f += np.matmul(np.matmul(cov_k_fp, cov_k_pp_inv), ts_se
            ts_seg_f.append(ts_seg_k_f)

        np_ts_seg_f = np.array(ts_seg_f)

        #mu_k_f = part_stats[0]["mu_k_f"]
        # Compute Eirola Paper's eqn 15
        mu_k_f = np.full(shape=part_stats[0]["mu_k_f"].shape, fill_value=0)
        ##Debug print("mu_k_f=",mu_k_f)
        ts_seg_ext = np.hstack((ts_seg,mu_k_f))
        ## Debug print("ts_seg_ext=", ts_seg_ext)
        np_probas = self.gmm.predict_proba([ts_seg_ext])
        ## Debug print("np_probas=", ts_seg_ext)
        ts_seg_f = np.dot(np_ts_seg_f.T, np_probas.T)
        ## Debug print("np_ts_seg_f=", np_ts_seg_f)
        ## Debug print("np_ts_seg_f.T=", np_ts_seg_f.T)
        ## Debug print("ts_seg_f=", ts_seg_f)
        return ts_seg_f

    def _partition_stats(self, ts_seg):
        """
        Partition the mean and covariance matrix into past and future compor
        @args =>
            self: instance of TimeSeriesGaussianMixture
            gmm: fitted Gaussian Mixture Model
            ts_seg: original time series data
        @return =>
            an array corresponding to each of the components in the gmm.
            Each array contains factorization of mean and covariance matrix
                "mu_k_p" = past component of mean vector for kth comp
                "mu_k_f" = future component of mean vector for kth comp
                "cov_k_pp" = past and past component of cov matrix for kth c
                "cov_k_pf" = past and future component of cov matrix for ktl
                "cov_k_fp" = future and past component of cov matrix for ktl
                "cov_k_ff" = future and future component of cov matrix for k
        """
        ts_seg_dim = ts_seg.shape[0]
        fit_dim = self.X.shape[1]
        fut_dim = fit_dim - ts_seg_dim

        #DEBUG print(ts_seg_dim, fut_dim)
        mu = self.gmm.means_
        cov = self.gmm.covariances_
        part_stats = []
        for k in range(mu.shape[0]):
            mu_k = mu[k]
            cov_k = cov[k]
            mu_k_p = mu_k[:ts_seg_dim]
            mu_k_f = mu_k[ts_seg_dim:]
            cov_k_pp = cov_k[:ts_seg_dim,:ts_seg_dim]
            cov_k_pf = cov_k[:ts_seg_dim,ts_seg_dim:]
            cov_k_fp = cov_k[ts_seg_dim:,:ts_seg_dim]
            cov_k_ff = cov_k[ts_seg_dim:,ts_seg_dim:]
            stats = {}
            stats["mu_k_p"] = mu_k_p
            stats["mu_k_f"] = mu_k_f
            stats["cov_k_pp"] = cov_k_pp
            stats["cov_k_pf"] = cov_k_pf
            stats["cov_k_fp"] = cov_k_fp
            stats["cov_k_ff"] = cov_k_ff
            part_stats.append(stats)

        return part_stats
```

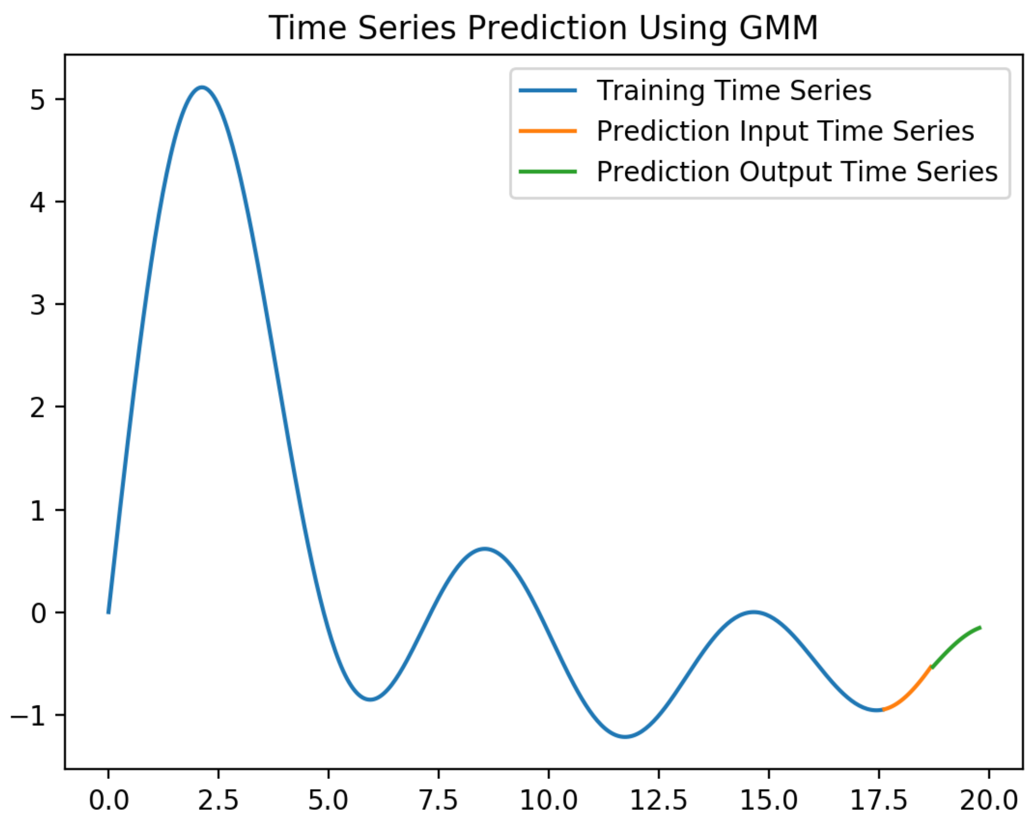
We have all the code ready and we are ready to run it. Here is commented code for predicting time series dataset

In [14]:
```
# As explained in EX.2.2.1, we transform 1d time series into 2d



# BEGIN dataset
num_samples = 500


# As explained in EX.2.2.1, we transform 1d time series into 2d
win_size = int(0.1*num_samples)

# get toy samples from sine waves
x_points, points  = TimeSeriesData.get_sine_with_noise(n=num_samples)
x_points, points  = TimeSeriesData.get_random_wave(n=num_samples)
#x_points, points  = TimeSeriesData.get_random_walk(n=num_samples)
# split dataset into train:test in the ratio 80:20
# train samples
train_pct = 0.80
train_num_samples = int(train_pct*num_samples)
train = points[:train_num_samples]
x_train = x_points[:train_num_samples]

# test samples
test_num_samples = int(num_samples-train_num_samples)
test = points[-test_num_samples:] # take sample in the end for test
x_test = x_points[-test_num_samples:]


# As explained in EX.2.2.3, since sample size is win_size and we use part o
# we choose to have input i.e. past dim to be 60% of window size

# number of samples in 2d dataset used as input at testing time
past_dims = int(win_size*0.5)
# number of dims or samples in 2d dataset that will be predicted by GMM mode
fut_dims = int(win_size - past_dims)

# END dataset

# BEGIN training

# instantiate GMM model
tsgmm_model = TimeSeriesGaussianMixture(win_size=win_size)

# train model
tsgmm_model.fit(train)

# END training


# BEGIN test

# transform test dataset into 2d representation as explained in EX.2.2.1
test_2d  = tsgmm_model.ts2X(test, tsgmm_model.ts_dims)
#print("test_2d = ", test_2d)
# Since as explained in EX.2.2.2, we give only past_dim as input to the mode
# and hence make a slice
test_query_points_2d = test_2d[:,:-fut_dims]

# We will have (num_test_samples-win_size) number of test points.
# We will run our prediction for 1 test points.
ts_past_points = test_query_points_2d[0]
ts_past_x_points = x_test[:past_dims]
#ps = ts_est.partition_stats(ts_seg)
#print(ps[0])
##print(ts_est.gmm.covariances_[0])

# make prediction , output will have fut_dims points
ts_future_points_tmp = tsgmm_model.predict(ts_past_points)
# post process to make output 1d
ts_future_points = ts_future_points_tmp.flatten()
ts_future_x_points =  x_test[past_dims:past_dims+fut_dims]

## DEBUG print("x_test", x_test.shape)
## DEBUG print("ts_past_points", ts_past_points.shape, ts_past_x_points.shap
##DEBUG print("ts_future", ts_future_points.shape, ts_future_x_points.shape,
# END test
```

```
Fitted Gaussian Mixture Model with convergence =  True
```

In [15]:

```
_debug = False

if _debug:
    print("ts_future_points=",ts_future_points)
    print("ts_past_points",ts_past_points)
    print("test_2d=", test_2d)
    print("test=", test)
    print("x_test=", x_test)
    print("x_test=", x_test)
    print("ts_past_points=",ts_past_points)

plt.title("Time Series Prediction Using GMM")
#plt.plot(x_points, points)
plt.plot(x_train, train, label="Training Time Series")
#plt.plot(x_test, test)
plt.plot(ts_past_x_points, ts_past_points, label="Prediction Input Time Seri
plt.plot(ts_future_x_points, ts_future_points, label="Prediction Output Time
plt.legend(loc='upper right')
plt.show()
```



Observations:

- We compute all the observation at once instead of other techniques which compute only partial
- it is robust to noise
- can handle arbitrary shape unlike normal regression and other techniques.

- There are other techniques for time series analysis but this techniques has advance as mentioned.

SM. Summary.

Here is what we did, we got the intuition of GMM and using our intuition we derived, the mathematical model of Gaussian Mixture Model and Numerical techniques for find its parameters using EM algorithms. In the end, to consider two application.

- Generating a new picture of a person, which we have not seen i.e. that person might not exists.
- GMM to predict time series by re-modelling input data set.