Matrix completion using Gaussian mixtures

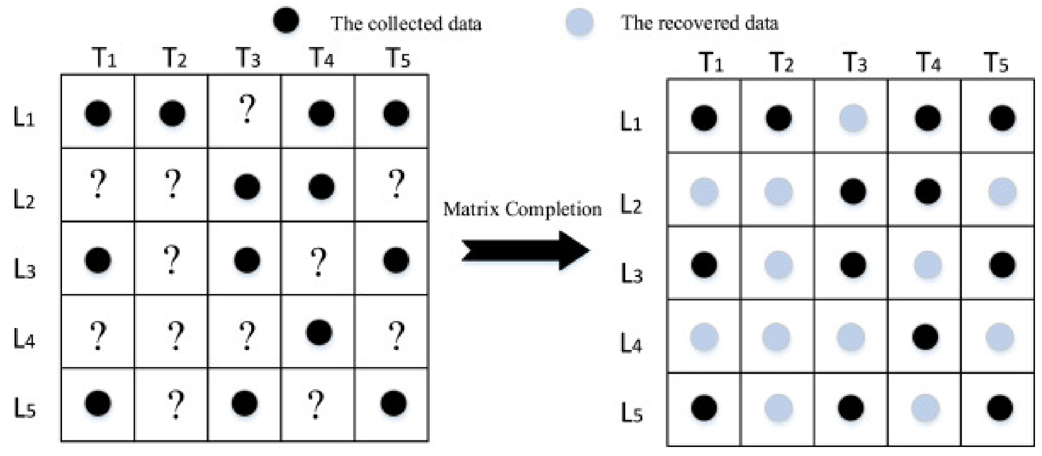
June 15, 2020

Rakesh Pavan (NIT Karnataka, Surathkal)

Advisor: Dr. V. Sreenath

1 Description of the problem

A data set can be represented as a matrix, with rows indicating different data instances, and the columns representing different attributes of a particular instance. However, most of the real-world data sets are incomplete, i.e due to various reasons, they have missing entries. The presence of missing values and incompleteness in data causes many issues in real-world data analysis. In this project we will implement a technique that makes use of a Gaussian mixture model to fill in incomplete entries of a matrix, hence completing the data matrix. For demonstration purposes, we will use the Movie Ratings data set by Netflix. The rows of this data set are different Netflix users, and the columns are rating given to different movies. Initially, this matrix has many missing entries since not all users would have watched all the movies. We will also use 2 experimental (synthetic) data matrix.



2 Approach to solving the problem:

This method assumes that the data instances (rows of the matrix) are independently sampled from a mixture of Gaussian distributions, and hence proceeds to estimate the parameters of those unknown Gaussian (using iterative EM algorithm which converges to a fixed solution after many iterations). Once the underlying distribution is estimated, the missing values of the data matrix can be filled, thereby approximately "recovering" the missing data. In the subsequent section we have briefly explained this statistical model and the equations used to estimate its parameters. References used to gain this information are:

1. Murphy, Kevin P. Machine Learning: A Probabilistic Perspective (2012) MIT Press, Cambridge, Mass
2. Bishop, Christopher M. Pattern Recognition and Machine Learning (2006) Springer-Verlag Berlin, Heidelberg.
3. Wikipedia article on "Mixture Model" - https://en.wikipedia.org/wiki/Mixture_model
4. Research paper on collaborative filtering using Gaussian mixtures - LINK

Before we can understand a Gaussian mixture model, we need to understand what are mixture models.

2.1 Mixture Models

We are given a data set $D = \{\underline{x}_1, \dots, \underline{x}_N\}$ where $\underline{x}_i$ is a d-dimensional vector measurement. Assume that the points are generated in an IID fashion from an underlying density $p(\underline{x})$. We further assume that $p(\underline{x})$ is defined as a finite mixture model with K components:

$$p(\underline{x}|\Theta) = \sum_{k=1}^{K} \alpha_k p_k(\underline{x}|z_k, \theta_k)$$

where:

- The $p_k(\underline{x}|z_k, \theta_k)$ are *mixture components*, $1 \leq k \leq K$. Each is a density or distribution defined over $p(\underline{x})$, with parameters θ_k.
- $z = (z_1, \dots, z_K)$ is a vector of K binary indicator variables that are mutually exclusive and exhaustive (i.e., one and only one of the z_k's is equal to 1, and the others are 0). z is a K-ary random variable representing the identity of the mixture component that generated $\underline{x}$. It is convenient for mixture models to represent z as a vector of K indicator variables.
- The $\alpha_k = p(z_k)$ are the mixture weights, representing the probability that a randomly selected $\underline{x}$ was generated by component k, where $\sum_{k=1}^{K} \alpha_k = 1$.

The complete set of parameters for a mixture model with K components is

$$\Theta = \{\alpha_1, \dots, \alpha_K, \theta_1, \dots, \theta_K\}$$

Membership Weights

We can compute the "membership weight" of data point $\underline{x}_i$ in cluster k, given parameters Θ as

$$w_{ik} \;=\; p(z_{ik} = 1|\underline{x}_i, \Theta) \;=\; \frac{p_k(\underline{x}_i|z_k, \theta_k) \cdot \alpha_k}{\sum_{m=1}^{K} p_m(\underline{x}_i|z_m, \theta_m) \cdot \alpha_m}, \quad 1 \leq k \leq K, \quad 1 \leq i \leq N.$$

This follows from a direct application of Bayes rule.

The membership weights above reflect our uncertainty, given $\underline{x}_i$ and Θ, about which of the K components generated vector $\underline{x}_i$. Note that we are assuming in our generative mixture model that each $\underline{x}_i$ was generated by a single component—so these probabilities reflect our uncertainty about which component $\underline{x}_i$ came from, not any "mixing" in the generative process.

2.2 Gaussian Mixture Models

For $\underline{x} \in \mathcal{R}^d$ we can define a Gaussian mixture model by making each of the K components a Gaussian density with parameters $\underline{\mu}_k$ and Σ_k. Each component is a multivariate Gaussian density

$$p_k(\underline{x}|\theta_k) \;=\; \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} e^{-\frac{1}{2}(\underline{x}-\underline{\mu}_k)^t \Sigma_k^{-1}(\underline{x}-\underline{\mu}_k)}$$

with its own parameters $\theta_k = \{\underline{\mu}_k, \Sigma_k\}$.

2.3 The EM Algorithm for estimating the model

The EM (Expectation-Maximization) algorithm for Gaussian mixtures as follows. The algorithm is an iterative algorithm that starts from some initial estimate of Θ (e.g., random), and then proceeds to iteratively update Θ until convergence is detected. Each iteration consists of an E-step and an M-step.

E-Step: Denote the current parameter values as Θ. Compute w_{ik} (using the equation above for membership weights) for all data points $\underline{x}_i, 1 \le i \le N$ and all mixture components $1 \le k \le K$. Note that for each data point $\underline{x}_i$ the membership weights are defined such that $\sum_{k=1}^{K} w_{ik} = 1$. This yields an $N \times K$ matrix of membership weights, where each of the rows sum to 1.

M-Step: Now use the membership weights and the data to calculate new parameter values. Let $N_k = \sum_{i=1}^{N} w_{ik}$, i.e., the sum of the membership weights for the kth component—this is the effective number of data points assigned to component k.

Specifically,

$$\alpha_k^{new} = \frac{N_k}{N}, \quad 1 \le k \le K.$$

These are the new mixture weights.

$$\underline{\mu}_k^{new} = \left(\frac{1}{N_k}\right) \sum_{i=1}^{N} w_{ik} \cdot \underline{x}_i \quad 1 \le k \le K.$$

The updated mean is calculated in a manner similar to how we could compute a standard empirical average, except that the ith data vector $\underline{x}_i$ has a fractional weight w_{ik}. Note that this is a vector equation since $\underline{\mu}_k^{new}$ and $\underline{x}_i$ are both d-dimensional vectors.

$$\Sigma_k^{new} = \left(\frac{1}{N_k}\right) \sum_{i=1}^{N} w_{ik} \cdot (\underline{x}_i - \underline{\mu}_k^{new})(\underline{x}_i - \underline{\mu}_k^{new})^t \quad 1 \le k \le K.$$

Again we get an equation that is similar in form to how we would normally compute an empirical covariance matrix, except that the contribution of each data point is weighted by w_{ik}. Note that this is a matrix equation of dimensionality $d \times d$ on each side.

The equations in the M-step need to be computed in this order, i.e., first compute the K new α's, then the K new $\underline{\mu}_k$'s, and finally the K new Σ_k's.

After we have computed all of the new parameters, the M-step is complete and we can now go back and recompute the membership weights in the E-step, then recompute the parameters again in the E-step, and continue updating the parameters in this manner. Each pair of E and M steps is considered to be one iteration.

2.4 Initialization and Convergence condition

The EM algorithm can be started by either initializing the algorithm with a set of initial parameters and then conducting an E-step, or by starting with a set of initial weights and then doing a first M-step. The initial parameters or weights can be chosen randomly (e.g. select K random data points as initial means and select the covariance matrix of the whole data set for each of the initial K covariance matrices) or could be chosen via some heuristic method (such as by using the k-means algorithm to cluster the data first and then defining weights based on k-means memberships).

Convergence is generally detected by computing the value of the log-likelihood after each iteration and halting when it appears not to be changing in a significant manner from one iteration to the next. Note that the log-likelihood (under the IID assumption) is defined as follows:

$$\log l(\Theta) = \sum_{i=1}^{N} \log p(\underline{x}_i|\Theta) = \sum_{i=1}^{N} \left(\log \sum_{k=1}^{K} \alpha_k p_k(\underline{x}_i|z_k, \theta_k) \right)$$

where $p_k(\underline{x}_i|z_k, \theta_k)$ is the Gaussian density for the kth mixture component.

This iterative algorithm eventually converges to a stationary solution, hence while coding we can measure the relative change in log-likelihood and keep a threshold of, say 10^{-6}, as the stopping criteria for the loop.

3 Implementation of the algorithm and equations

In this section we provide for the algorithm and formulas mentioned in the above section (section 2.1-2.4). All the equations are directly hard-coded in the cells. The functions "estep()" and "mstep()" are the code for E-step and M-step of the algorithm (described in section 2.3). The "run()" function is the iterative algorithm which alternatively calls E,M steps until convergence.

```
[2]: class GaussianMixture(NamedTuple):
         """Tuple holding a gaussian mixture"""
         mu: np.ndarray  # (K, d) array - each row corresponds to a␣
      ↪gaussian component mean
         var: np.ndarray  # (K, ) array - each row corresponds to the␣
      ↪variance of a component
         p: np.ndarray  # (K, ) array = each row corresponds to the weight␣
      ↪of a component
```

```python
def init(X: np.ndarray, K: int,
         seed: int = 0) -> Tuple[GaussianMixture, np.ndarray]:
    """Initializes the mixture model with random points as initial
    means and uniform assingments

    Args:
        X: (n, d) array holding the data
        K: number of components
        seed: random seed

    Returns:
        mixture: the initialized gaussian mixture
        post: (n, K) array holding the soft counts
            for all components for all examples

    """
    np.random.seed(seed)
    n, _ = X.shape
    p = np.ones(K) / K

    # select K random points as initial means
    mu = X[np.random.choice(n, K, replace=False)]
    var = np.zeros(K)
    # Compute variance
    for j in range(K):
        var[j] = ((X - mu[j])**2).mean()

    mixture = GaussianMixture(mu, var, p)
    post = np.ones((n, K)) / K

    return mixture, post

def logsumexp(ns):
    mx = np.max(ns)
    ds = ns - mx
    sumOfExp = np.exp(ds).sum()
    return mx + np.log(sumOfExp)
```

```python
[3]: def estep(X: np.ndarray, mixture: GaussianMixture) -> Tuple[np.
 ↪ndarray, float]:
    """E-step: Softly assigns each datapoint to each of the Gaussian␣
 ↪distributions

    Args:
        X: (n, d) array holding the data, with incomplete entries␣
 ↪(set to 0)
```

```
        mixture: the current gaussian mixture

    Returns:
        np.ndarray: (n, K) array holding the soft counts
            for all components for all examples
        float: log-likelihood of the assignment

    """
    n, _ = X.shape
    K, _ = mixture.mu.shape
    post = np.zeros((n, K))

    ll = 0
    for i in range(n):
        mask = (X[i, :] != 0)
        for j in range(K):
            log_likelihood = log_gaussian(X[i, mask], mixture.mu[j,␣
↪mask],
                                          mixture.var[j])
            post[i, j] = np.log(mixture.p[j] + 1e-16) + log_likelihood
        total = logsumexp(post[i, :])
        post[i, :] = post[i, :] - total
        ll += total

    return np.exp(post), ll


def log_gaussian(x: np.ndarray, mean: np.ndarray, var: float) ->␣
↪float:
    """Computes the log probablity of vector x under a normal␣
↪distribution

    Args:
        x: (d, ) array holding the vector's coordinates
        mean: (d, ) mean of the gaussian
        var: variance of the gaussian

    Returns:
        float: the log probability
    """
    d = len(x)
    log_prob = -d / 2.0 * np.log(2 * np.pi * var)
    log_prob -= 0.5 * ((x - mean)**2).sum() / var
    return log_prob
```

```python
def mstep(X: np.ndarray, post: np.ndarray, mixture: GaussianMixture,
          min_variance: float = .25) -> GaussianMixture:
    """M-step: Updates the Gaussian statistics (p,mu,var) for better␣
 ↪fit
    of the weighted dataset

    Args:
        X: (n, d) array holding the data, with incomplete entries␣
 ↪(set to 0)
        post: (n, K) array holding the soft counts
            for all components for all examples
        mixture: the current gaussian mixture
        min_variance: the minimum variance for each gaussian

    Returns:
        GaussianMixture: the new gaussian mixture
    """
    n, d = X.shape
    _, K = post.shape

    n_hat = post.sum(axis=0)
    p = n_hat / n

    mu = mixture.mu.copy()
    var = np.zeros(K)

    for j in range(K):
        sse, weight = 0, 0
        for l in range(d):
            mask = (X[:, l] != 0)
            n_sum = post[mask, j].sum()
            if (n_sum >= 1):
                # Updating mean
                mu[j, l] = (X[mask, l] @ post[mask, j]) / n_sum
            # Computing variance
            sse += ((mu[j, l] - X[mask, l])**2) @ post[mask, j]
            weight += n_sum
        var[j] = sse / weight
        if var[j] < min_variance:
            var[j] = min_variance

    return GaussianMixture(mu, var, p)
```

[4]:
```python
''' The main algorithm to estimate the parameters/statistics of the␣
 ↪Gaussians'''
def run(X: np.ndarray, mixture: GaussianMixture,
        post: np.ndarray) -> Tuple[GaussianMixture, np.ndarray,␣
↪float]:
    """Runs the main algorithm. Iteratively perform E-step
                                    and M-step until convergence

    Args:
        X: (n, d) array holding the data
        post: (n, K) array holding the soft counts
            for all components for all examples

    Returns:
        GaussianMixture: the new gaussian mixture
        np.ndarray: (n, K) array holding the soft counts
            for all components for all examples
        float: log-likelihood of the current assignment
    """

    prev_ll = None
    ll = None

    while (prev_ll is None or ll - prev_ll > 1e-6 * np.abs(ll)):
        ''' This algorithm will iteratively run until convergence
            The convergence is measured by relative improvement in␣
↪the
            log-likelihood measure of the Gaussians
        '''
        prev_ll = ll
        post, ll = estep(X, mixture) # Perform E-step
        mixture = mstep(X, post, mixture)  # Perform M-step

    return mixture, post, ll
```

Predicting the incomplete entries from the calculated Statistics: Let G_i be the $\mathrm{i}^{\mathrm{th}} -$ Gaussian distribution, for $1 \leq i \leq K$. Then prediction for the n^{th} missing element is:

$$pred_n = \sum_{i=1}^{K} \mathbb{P}(X_n \in G_i) \times Mean(G_i)$$

where $\mathbb{P}(X_n \in G_i)$ is probability that X_n was sampled from G_i, and $Mean(G_i)$ is the mean of the $\mathrm{i}^{\mathrm{th}} -$ Gaussian.

Below code implements this formula to fill each of the missing matrix entries:

```
[5]: def fill_matrix(X: np.ndarray, mixture: GaussianMixture) -> np.
↪ndarray:
    """Fills an incomplete matrix according to
    previously estimated Gaussian mixture statistics

    Args:
        X: (n, d) array of incomplete data (incomplete entries =0)
        mixture: a mixture of gaussians

    Returns
        np.ndarray: a (n, d) array with completed data
    """
    n, d = X.shape
    X_pred = X.copy()
    K, _ = mixture.mu.shape

    for i in range(n):
        mask = X[i, :] != 0
        mask0 = X[i, :] == 0
        post = np.zeros(K)
        for j in range(K):
            log_likelihood = log_gaussian(X[i, mask], mixture.mu[j,␣
↪mask],
                                          mixture.var[j])
            post[j] = np.log(mixture.p[j]) + log_likelihood
        post = np.exp(post - logsumexp(post))
        X_pred[i, mask0] = np.dot(post, mixture.mu[:, mask0])
    return X_pred
```

4 Experiments and Results

We conducted 3 experiments on 3 data matrix. They are summarised below:

1. **Experiment 1:** We first construct a synthetic data set by randomly sampling points on a 2D plane. Can be found in the file "exp_data5.txt" file. This data set is a set of 2D points (1000 such points) on the Cartesian plane, hence can be represented by a 1000 x 2 matrix. It is also to visualize on the 2D Cartesian and hence we will use this to demonstrate how the algorithm converges trying to estimate the statistics for the Gaussian. The final result of this experiment is the graph/plot that can be seen in below section along with the code. **We also have made an animation (in form of a GIF file named "converge.gif") which shows how after each iteration the algorithm updates the mean and variances of the Gaussian.**

2. **Experiment 2:** In this experiment we try to obtain a plot between the percentage of missing data, and the error obtained after recovery (in terms of Mean Absolute Error). We use the same data created in above experiment for this experiment also.

3. **Experiment 3:** Testing on real-life NetFlix data set. The dataset provides movie rating for 1200 different movies by 1200 different Netflix users. Hence it is a 1200 x 1200 matrix. (This is a subset of the original Netflix Prize challenge 2009 which has rating information for millions of users on thousands of movies). The "netflix_incomplete.txt" dataset is the incomplete version of the this dataset (with around 23%missing entries). And "netflix_complete.txt" is

the complete version to be used for measuring the error. More information on the data set can be found here and here. Moreover, for demonstration purposes we also select a small (20 x 5) subset of this dataset and display the input (incomplete matrix) and output (filled matrix).

4.1 Implementation of the experiments and plots

[6]:
```
def plot(X: np.ndarray, mixture: GaussianMixture, post: np.ndarray,
         title: str):
    """Plots the Gaussians and the data points for 2D data"""
    _, K = post.shape

    percent = post / post.sum(axis=1).reshape(-1, 1)
    fig, ax = plt.subplots()
    ax.title.set_text(title)
    ax.set_xlim((-20, 20))
    ax.set_ylim((-20, 20))
    r = 0.75
    color = ["r", "b", "k", "y", "m", "c"]
    for i, point in enumerate(X):
        theta = 0
        for j in range(K):
            offset = percent[i, j] * 360
            arc = Arc(point,
                      r,
                      r,
                      0,
                      theta,
                      theta + offset,
                      edgecolor=color[j])
            ax.add_patch(arc)
            theta += offset
    for j in range(K):
        mu = mixture.mu[j]
        sigma = np.sqrt(mixture.var[j])
        circle = Circle(mu, sigma, color=color[j], fill=False)
        ax.add_patch(circle)
        legend = "mu = ({:0.2f}, {:0.2f})\n stdv = {:0.2f}".format(
            mu[0], mu[1], sigma)
        ax.text(mu[0], mu[1], legend)
    plt.axis('equal')
    plt.xlabel('X axis')
    plt.ylabel('Y axis')
    plt.show()
```

[8]:
```
def missing(data):
    ''' Measure the percentage of elements
```

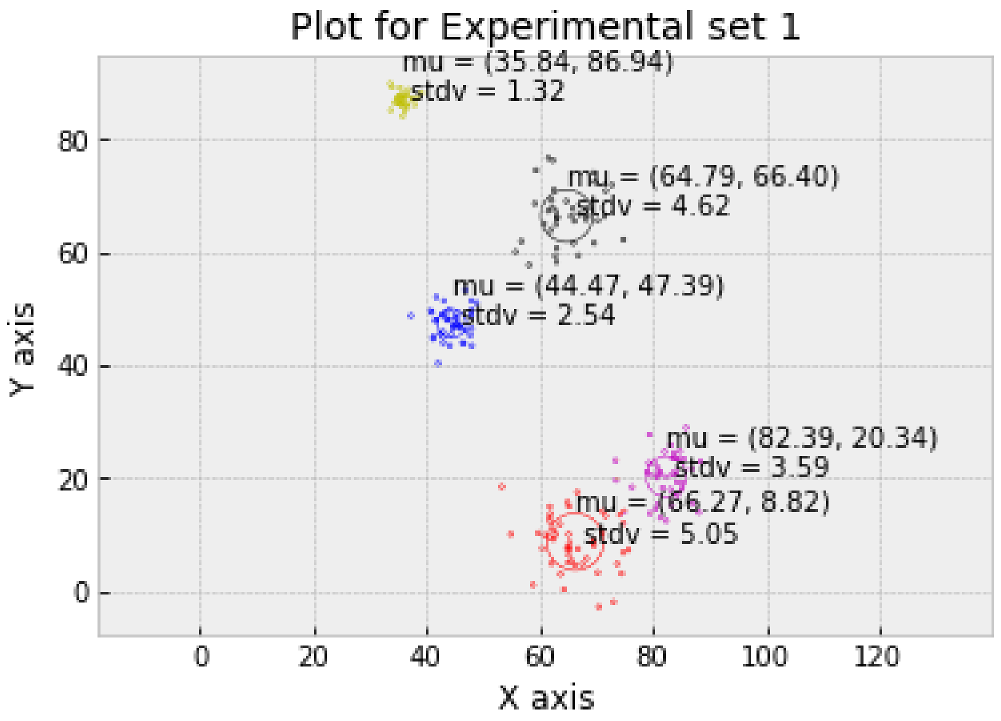
```
        missing in the data matrix
    '''
    return 100*(data==0).mean()

def run_test():
    '''
    Estimate the statistics for:
        Experimental dataset 1
    '''
    K = 5 # Number of clusters / classes
    seed = 0 # Setting numpy random seed=0
    mixture, post = init(X, K, seed)
    mixture, post, ll = run(X, mixture, post)
    title = "Plot for Experimental set 1"
    plot(X,mixture,post,title) #PLotting the Gaussians and the data

def run_matrix_completion():
    '''Evaluating the technique on the Netflix data'''
    K = 12
    seed = 1
    mixture, post = init(X, K, seed)
    mixture, post, ll = run(X, mixture, post)
    X_pred = fill_matrix(X, mixture)
    X_gold = np.loadtxt('netflix_complete.txt')
    print("RMSE Error:", rmse(X_gold, X_pred))

def Test():
    ''' Evaluating the technique on
        Experimental dataset 2 '''
    X = np.loadtxt("test_incomplete.txt")
    X_gold = np.loadtxt("test_complete.txt")
    print("X_incomplete:\n" + str(X)) # Printing the incomplete data␣
↪matrix
                                        # 0's indicate missing entries
    K = 5 # Number of Gaussians
    n, d = X.shape
    seed = 0 # Numpy random seed
    print('Missing %:',np.round(missing(X),6))
    mixture, _ = init(X, K, seed)
    print()
    post, ll = estep(X, mixture)

    mu, var, p = mstep(X, post, mixture)
    (mu, var, p), post, ll = run(X, mixture, post)

    X_pred = fill_matrix(X, GaussianMixture(mu, var, p))
            # Filling the matrix uing the above computed statistics
    error = rmse(X_gold, X_pred) # Evaluating RMSE error
    print("X_complete:\n" + str(X_gold)) # Printing original complete
                                         #  data matrix
    print("X_pred:\n" + str(X_pred))
    print("RMSE Error: " + str(error))
```

4.1.1 Loading and testing on Experimental Dataset 1

Experimental dataset 1 can be found in "exp_data5.txt" file. This data set is a set of 2D points (1000 such points) on the cartesian plane, hence can be represented by a 1000 x 2 matrix. It is also to visualize on the 2D cartesian and hence we will use this to demonstrate how the algorithm converges tying to estimate the statistics for the Gaussians.

This dataset was originally contructed by randomly sampling points from 5 different Gaussian distributions.

```
[9]: X = np.loadtxt("exp_data5.txt")
     run_test()
```



The circles are the different Gaussian distributions. The centre of the circle is the Mean (mu) and the radius of the circle is equal to the Standard Deviation (stdv) of the corresponding Gaussian distribution.

All the Gaussians are initialized randomly.

```
[10]: '''
         Displaying a GIF to demonstrate how the Gaussian statistics
         iteratively converge to the data points.
         Obtained by saving the above plot for iterations from 1 to 30
         of the algorithm.
      '''
      from IPython.display import HTML
      HTML('<img src="result.gif">')
```

```
[10]: <IPython.core.display.HTML object>
```

The GIF file ("**results/result.gif**") can be viewed locally or in the Jupyter-notebook, since PDF report doesn't support GIF. In the GIF we can observe that after each iteration of the algorithm, the Gaussian statistics (the circles) adjust more and more to the data points. After some iterations (30 in the above case) the algorithm converges.

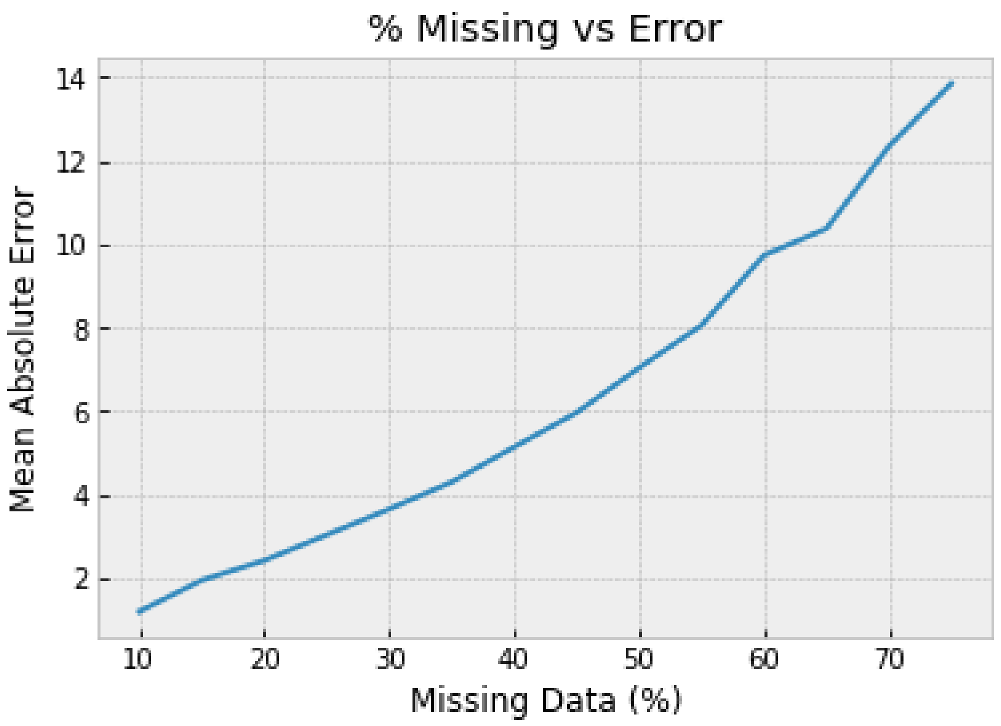
4.1.2 Plotting Missing(%) vs Mean Absolute error for Experimental dataset 1

In this experiment we try to obtain a plot between the percentage of missing data, and the error obtained after recovery (in terms of Mean Absolute Error) for this method. We use the Experimental dataset 1 for this purpose.

$$\text{MeanAbsoluteError} = \frac{\sum_i^N \mid X_i - Y_i \mid}{N}$$

```
[11]: np.random.seed(0)  # Setting random seed
      sp = []
      mae = []
      for spar in [0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65,0.
       ↪7,0.75]:
          data = np.loadtxt("exp_data5.txt") # Loading the data
          X_gold = np.array(data)  # The original complete data

          num = 2*len(data)*spar # Number of missing elements

          while num>0: # Randomly selecting 'num' elements and setting them␣
       ↪0
              i = np.random.randint(0,data.shape[0])
              j = np.random.randint(0,data.shape[1])
              if data[i][j] > 0:
                  data[i][j] = 0
                  num-=1

          X= np.array(data) # The incomplete data

          K = 5 # Number of Gaussians
          seed = 0
```

```
    mixture, post = init(X, K, seed) #Initializing the statistics
    mixture, post, ll = run(X, mixture, post) # Estimating the␣
↪statistics
    X_pred = fill_matrix(X, mixture) # Filling the incomplete matrix
                                     # using the above calculated␣
↪stats
    sp.append(100*spar)
    mae.append(np.mean(np.abs(X_pred-X_gold)))

plt.plot(sp,mae)
plt.title('% Missing vs Error')
plt.xlabel('Missing Data (%)')
plt.ylabel('Mean Absolute Error')
plt.show()
```



As expected, more the missing data, greater the recovery error.

4.1.3 For Experimental data set 2

This data set is a small subset of the Netflix version, to demonstrate the input, output of the algorithm. It is a matrix of 20 x 5 dimension.

X_incomplete: Is the incomplete data with missing entries (0 indicated missing value)

X_complete: This is the original complete data

X_pred: The prediction matrix after running our algorithm and filling the matrix.

RMSE Error = $\left(\frac{\sum_{i=1}^{N}(X_i - Y_i)^2}{N}\right)^{1/2}$

[12]:
```
Test()
```

```
X_incomplete:
[[2. 5. 3. 0. 0.]
 [3. 5. 0. 4. 3.]
 [2. 0. 3. 3. 1.]
 [4. 0. 4. 5. 2.]
 [3. 4. 0. 0. 4.]
 [1. 0. 4. 5. 5.]
 [2. 5. 0. 0. 1.]
 [3. 0. 5. 4. 3.]
 [0. 5. 3. 3. 3.]
 [2. 0. 0. 3. 3.]
 [3. 4. 3. 3. 3.]
 [1. 5. 3. 0. 1.]
 [4. 5. 3. 4. 3.]
 [1. 4. 0. 5. 2.]
 [1. 5. 3. 3. 5.]
 [3. 5. 3. 4. 3.]
 [3. 0. 0. 4. 2.]
 [3. 5. 3. 5. 1.]
 [2. 4. 5. 5. 0.]
 [2. 5. 4. 4. 2.]]
Missing %: 19.0

X_complete:
[[2. 5. 3. 4. 3.]
 [3. 5. 3. 4. 3.]
 [2. 4. 3. 3. 1.]
 [4. 4. 4. 5. 2.]
 [3. 4. 4. 4. 4.]
 [1. 5. 4. 5. 5.]
 [2. 5. 4. 5. 1.]
 [3. 4. 5. 4. 3.]
 [3. 5. 3. 3. 3.]
 [2. 5. 3. 3. 3.]
 [3. 4. 3. 3. 3.]
 [1. 5. 3. 5. 1.]
 [4. 5. 3. 4. 3.]
 [1. 4. 3. 5. 2.]
 [1. 5. 3. 3. 5.]
 [3. 5. 3. 4. 3.]
 [3. 5. 4. 4. 2.]
 [3. 5. 3. 5. 1.]
 [2. 4. 5. 5. 3.]
 [2. 5. 4. 4. 2.]]
X_pred:
[[2.         5.         3.         3.94430072 1.5291021 ]
 [3.         5.         3.11560886 4.         3.        ]
 [2.         4.99001402 3.         3.         1.        ]
 [4.         4.21254903 4.         5.         2.        ]
 [3.         4.         3.18503123 3.64288289 4.        ]
 [1.         4.99965652 4.         5.         5.        ]
 [2.         5.         3.16507367 4.01202234 1.        ]
 [3.         4.21314223 5.         4.         3.        ]
 [2.99360835 5.         3.         3.         3.        ]
 [2.         4.6344124  3.16745754 3.         3.        ]
 [3.         4.         3.         3.         3.        ]
 [1.         5.         3.         4.00011263 1.        ]
 [4.         5.         3.         4.         3.        ]
 [1.         4.         4.50283802 5.         2.        ]
 [1.         5.         3.         3.         5.        ]
 [3.         5.         3.         4.         3.        ]
 [3.         4.40676172 4.03609658 4.         2.        ]
 [3.         5.         3.         5.         1.        ]
 [2.         4.         5.         5.         2.30793523]
 [2.         5.         4.         4.         2.        ]]
RMSE Error: 0.3157193993270509
```

4.1.4 Testing on the NetFlix dataset

The dataset provides movie rating for 1200 different movies by 1200 different Netflix users. Hence it is a 1200 x 1200 matrix. (This is a subset of the orignal Netflix Prize challenge 2009 which has rating information for millions of users on thousands of movies)

The "netflix_incomplete.txt" dataset is the incomplete version of the this dataset (with around 23% missing entries). And "netflix_complete.txt" is the complete version to be used for measuring the error.

RMSE Error = $\left(\frac{\sum_{i=1}^{N}(X_i - Y_i)^2}{N}\right)^{1/2}$

[13]:
```
X = np.loadtxt("netflix_incomplete.txt")
print('Missing %:',np.round(missing(X),6))
run_matrix_completion()    # Takes about 1-2 mins to run
```

```
Missing %: 22.793889
RMSE Error: 0.4804908505400682
```

5 Conclusions

Matrix completion is the task of filling in the missing entries of a partially observed matrix. A wide range of datasets are naturally organized in matrix form. Hence this technique is useful in wide variety of data completion/recovery tasks (in many different application domains like Biological data, high-energy physics data, e-commerce data, etc) to recover useful information from incomplete data. Apart from data recovery, it is also useful in collaborative filtering, system identification and IOT related tasks.

In this project we study and implement a popular algorithm which makes use of Gaussian mixtures to fill in incomplete entires of the matrix. From our experiments and results we learn that this method indeed produces good quality recoveries from incomplete data. We obtained RMSE error of 0.48 and 0.31 on the Netflix data and its 20x5 subset respectively. We also obtained a plot of missing data (%) vs the Mean Absolute error, and the trend in the plot made it clear that more missing data implies that error after recovery will be higher. For demonstrating the convergence of the algorithm we also make a GIF animation plot ("**results/result.gif**") and we observe that indeed the algorithm converges after some iterations and estimates the Gaussian parameters fitting to the data.

6 Additional Information about the Code, Plots and GIF

The folder "results/" contain all the plots (PNG files) and the GIF animation ("results.gif" file). The "code.ipynb" file contains all the implementations. All the other additional ".txt" files are the required data files which get loaded in the code when the code is run.