PRML Assignment 2

ME21B165
Rudra Panch

April 14, 2024

1 EM Algorithm

1.1 Bernoulli Mixture Modelling

As the points are in $0, 1_{50}$, it is feasible to assume that the model was generated from a Bernoulli distribution. Let us assume this is generated from a mixture of 4 Bernoulli distributions. A parameter π defines the probability of a point being from a certain mixture. Every Bernoulli mixture should have a set of probabilities p_j which define the probability of the jth feature in a data point being 1. We have a dataset of 400 points and we need to find the maximum likelihood using these parameters

$$\sum_{k=1}^{4} \pi_k = 1$$

$$0 \leq p_{kj} \leq 1 \qquad \forall k, j$$

A latent variable z classifies these points into the 4 clusters depending on the values on π but we do not have access to these values. If we assume point i comes from the cluster z_i, the function value of point i would be

$$f(x_i) = \prod_{j=1}^{50} p_{z_i j}^{x_{ij}} (1 - p_{z_i j}^{(1-x_{ij})})$$

As we do not know the value of z_i, we assume a probabilistic model where we take the π values of the 4 clusters and find an approximate value considering the possibility of the point belonging to any one of the 4 clusters

$$f(x_i) = \sum_{k=1}^{4} \left[\pi_k \prod_{j=1}^{50} p_{z_i j}^{x_{ij}} (1 - p_{z_i j}^{(1-x_{ij})}) \right]$$

1.1.1 Likelihood Function

From Fischer's Maximum Likelihood function, we know that the formula of Likelihood is given by. We can substitute the above derived formula into the Likelihood function and obtain

$$L(\theta|x) = \prod_{x} f(x|\theta)$$

$$L(\pi, p|x) = \prod_{i=1}^{400} \sum_{k=1}^{4} \left[\pi_k \prod_{j=1}^{50} p_{z_i j}^{x_{ij}} (1 - p_{z_i j}^{(1-x_{ij})}) \right]$$

$$\log(L(\pi, p|x)) = \sum_{i=1}^{400} \log(\sum_{k=1}^{4} \left[\pi_k \prod_{j=1}^{50} p_{z_i j}^{x_{ij}} (1 - p_{z_i j}^{(1-x_{ij})}) \right])$$

But this log function contains a $\sum$ inside and hence, it is analytically very difficult to solve this by differentiating and setting to 0. Hence, we take a small deviation and use the Jensen's Inequality

to our advantage. The Jensen's formula allows us to transform a $\log(\sum_)$ into a $\sum_(\log)$ by adding an inequality sign. However, for this, we would have to add extra parameters, which I would like to denote as λ_k^i. This introduces almost 1600 new parameters which makes our problem even more tedious, but we have at least brought it to a solvable form.

$$\log(\sum_i^n \lambda_i x_i) \geq \sum_i^n \lambda_i \log(x_i)$$

$$\log(L(\pi, p|x)) \geq \sum_{i=1}^{400} \sum_{k=1}^{4} \left[\lambda_k^i \log(\frac{\pi_k}{\lambda_k^i} \prod_{j=1}^{50} p_{z_ij}^{x_{ij}} (1 - p_{z_ij}^{(1-x_{ij})})) \right]$$

This is the modified log likelihood function which gives a lower bound for the maximum log likelihood function. It can be proved that maximizing the MML leads to maximizing the original likelihood function. Since the MML has only a product inside the log, it would be easier to take it's derivative and set it to zero to find the parameters.

1.1.2 EM Algorithm

To maximize π λ and p all at once would be inefficient and tedious and hence we shall do it 2 steps. We can see that, if we fix the values of π and p, it would be easy to solve for λ and the vice-versa would also be true. We can fix the values of λ and maximize π and p. We iterate until convergence of π and p. The algorithm can be summarized as

- Step 1: Initialization. p^0 and π^0 where the superscript 0 refers to iteration number 0

$$p^0 = \left\{p_{kj}^0\right\} \qquad \forall k, j$$
$$\pi^0 = \left\{\pi_k^0\right\} \qquad \forall k$$

- Step 2: Expectation Step. Maximize λ keeping π and p constant

$$\lambda^{t+1} = \arg\max_{\lambda} MML(\pi^t, p^t, \lambda^t)$$

- Step 3: Maximization Step. Maximize π and p keeping λ constant

$$\pi^{t+1}, p^{t+1} = \arg\max_{\pi,p} MML(\pi^t, p^t, \lambda^{t+1})$$

- Step 4: Repeat Step 2 and 3 until convergence of π and p values

$$\|\pi^{t+1} - \pi^t\| \leq \epsilon$$

$$\|p^{t+1} - p^t\| \leq \epsilon$$

1.1.3 Expectation Step

We need to differentiate the MML with respect to λ_k^i assuming the other parameters to be constant. When we do this, we end up with the solution for λ_k^i

$$\lambda_k^i = \frac{\pi_k \prod_{j=1}^{50} p_{z_ij}^{x_{ij}} (1 - p_{z_ij}^{(1-x_{ij})}))}{\sum_{k=1}^{4} \pi_k \prod_{j=1}^{50} p_{z_ij}^{x_{ij}} (1 - p_{z_ij}^{(1-x_{ij})}))} \qquad \forall i, k$$

1.1.4 Maximization Step

We need to differentiate the MML with respect to p_k^i and π_k assuming the other parameters to be constant. When we do this, we end up with the solution for p_k^i and π_k

$$\pi_k = \frac{\sum_{i=1}^{400} \lambda_k^i}{400}$$

$$p_{kj} = \frac{\sum_{i=1}^{400} \lambda_k^i x_{ij}}{\sum_{i=1}^{400} \lambda_k^i}$$

1.1.5 Code and Plots

Shown below is the code written for the Expectation step, Maximization step, EM algorithm main code and the plot of the log likelihood function averaged over 100 random intializations against number of iterations taken

```
[4]  def f(x, p):  # Calculate function value of Bernoulli Distribution
       ans=1       # Initialize a variable
       for j in range(x.shape[0]):  # Iterate over all features
         ans = ans * (p[j]**x[j]) * ((1-p[j])**(1-x[j]))
       return ans

[5]  def expectation(X, p, pi):    # Expectation Step of the EM Algorithm
       lamda = np.zeros((X.shape[0],4))  # Initialize a matrix
       for i in range(X.shape[0]):
         sum=0
         for k in range(4):
           g = f(X[i],p[k])*pi[k]  # Find the function value
           sum = sum + g           # Sum them up in a separate value
           lamda[i][k] = g
         if sum != 0:
           lamda[i] = lamda[i]/sum # Divide by the total sum
       return lamda
```

Figure 1: Expectation Step

```
[6]  def maximization(X, lamda):   # Maximization Step of the EM ALgorithm
       pi = np.zeros(4)            # Initialize
       p = np.zeros((4,X.shape[1]))

       for k in range(4):          # Calculate the new values of pi
         for i in range(X.shape[0]):
           pi[k] = pi[k] + lamda[i][k]

       for k in range(4):          # Calculate the new values of p
         for j in range(50):
           for i in range(400):
             p[k][j] = p[k][j] + lamda[i][k]*X[i][j]

       for k in range(4):
         if pi[k] != 0:
           p[k] = p[k]/pi[k]

       pi = pi/400

       return pi, p
```

Figure 2: Maximization step

```
[12] def EM(X):  # EM Algorithm main code
       pi = np.random.rand(4)  # Randomly initialize
       pi = pi/np.sum(pi)
       p = np.random.rand(4,X.shape[1])
       lamda = np.zeros((X.shape[0],4))

       it=0                    # Iterations variable
       logl=np.array([])       # Log Likelihood to be maintained
       err=1                   # Variable to calculate error

       while it<50:
         lamda_new = expectation(X,p,pi)      # Expectation step
         pi_new, p_new = maximization(X,lamda_new)  # Maximization step
         logl = np.append(logl,log_likelihood(X,pi_new,p_new))
         err = error(pi_new,p_new,pi,p)
         p=p_new                  # Update the new parameters
         pi=pi_new
         lamda = lamda_new
         it = it+1

       return logl   # Return the results

[18] def EM_iter(X):  # EM ALgorithm over 100 random inititalizations
       log_likelihood = np.zeros(50)  # Array to plot log likelihood function
       for i in range(100):
         logl = EM(X)
         log_likelihood = log_likelihood + logl
       return log_likelihood
```

Figure 3: EM Algorithm

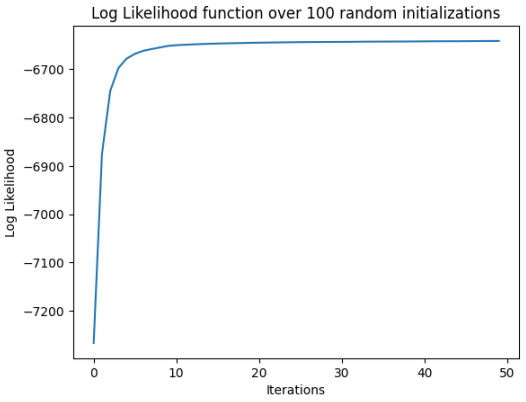

Figure 4: Plot of Log Likelihood function

1.2 Gaussian Mixture Modelling

We assume 4 Gaussian mixtures to be generating this data. The 4 Gaussian have means which are 50*1 dimensional vectors. We shall assume a covariance matrix of size 50*50. This is because there is no implicit indication of the independence of the features on each other. The Gaussian Probability Density Function is given by

$$f(x_i, \mu_k, \sigma^{@}) = \frac{1}{\sqrt{2\pi d|\sigma^2|}} \exp(\frac{-1}{2}(x_i - \mu_k)^T \left\{\sigma^2\right\}^{-1} (x_i - \mu_k)$$

1.2.1 EM Algorithm

The formulas in the case of GMM are given as

- Step 1: Initialize the parameters
  We initialize μ and π with random values but doing this with covariance matrix might be risky as we need to ensure the matrix is invertible and positive semi-definite. To do this, we randomly initialize a matrix G and set $cov[k] = GG^T$. We do this 4 times for all 4 cluster and this ensures both conditions are met.

- Step 2: Expectation step
  We assume μ π and σ^2 to be constant and maximize λ using the formula

$$\lambda_k^i = \frac{\pi_k f(x_i, \mu_k, \sigma_k^2)}{\sum_{i=1}^n \pi_k f(x_i, \mu_k, \sigma_k^2)}$$

- Step 3: Maximization step: We assume λ to be constant and maximize μ π and σ^2 using the formulas

$$\pi_k = \frac{\sum_{i=1}^n \lambda_k^i}{n}$$

$$\mu_{kj} = \frac{\sum_{i=1}^n \lambda_k^i x_{ij}}{\sum_{i=1}^n \lambda_k^i}$$

$$\sigma_k^2 = \frac{\sum_{i=1}^n \lambda_k^i (x_i - \mu_k)^T (x_i - \mu_k)}{\sum_{i=1}^n \lambda_k^i}$$

- Step 4: Repeat Steps 2 and 3 until convergence. Convergence criterion given by

$$\| \left\{\pi, \mu, \sigma^2\right\}^{t+1} - \left\{\pi, \mu, \sigma^2\right\}^{t} \| \leq \epsilon$$

1.2.2 Code and Plots

Shown below is the code written for the Expectation step, Maximization step, EM algorithm main code and the plot of the log likelihood function averaged over 100 random intializations against number of iterations taken

```
[21] def gaussian(x,mu,cov):  # Calculate Gaussian PDF value
       if np.linalg.det(cov)==0:
         cov += 0.1*np.eye(50)
       c = -0.5 * ((x.T - mu.T).T @ np.linalg.inv(cov)) @ (x.T - mu.T)
       c = np.exp(c)
       c /= np.sqrt((2 * np.pi) ** 50 * np.linalg.det(cov))
       return c

[7]  def expectation(X, mu, cov, pi):  # Expectation Step
       lamda = np.zeros((X.shape[0],4)) # Initialize with zeros
       for i in range(X.shape[0]):
         sum = 0
         for k in range(4):
           g = gaussian(X[i],mu[k],cov[k])*pi[k]  # Calculate PDF value
           lamda[i][k] = g
           sum = sum + g
         if sum!=0:
           lamda[i] = lamda[i]/sum     # Final results
       return lamda
```

Figure 5: Expectation Step

```
[16] def maximization(X, lamda):   # Maximization step
       pi = np.zeros(4)            # Initialization
       mu = np.zeros((4,X.shape[1]))
       cov  = np.zeros((4,X.shape[1],X.shape[1]))

       for k in range(4):                # Calculate pi
         for i in range(X.shape[0]):
           pi[k] = pi[k] + lamda[i][k]

       for k in range(4):                # calculate mu
         for j in range(50):
           for i in range(400):
             mu[k][j] = mu[k][j] + lamda[i][k]*X[i][j]

       for k in range(4):                # Calculate cov
         for j in range(400):
           cov[k] = cov[k] + ((X[j]-mu[k]).T  @ (X[j]-mu[k]))*lamda[j][k]

       for k in range(4):                # Final results
         if pi[k] != 0:
           mu[k] = mu[k]/pi[k]
           cov[k] = cov[k]/pi[k]

       pi = pi/400

       return pi, mu, cov
```

Figure 6: Maximization step

1.2.3 Comparison between GMM and BMM

As we can, the log Likelihood graph of BMM is very smooth and converges within 10-20 iterations to an average value of -6.7k. The BMM works well since the data is actually Bernoulli distributed unlike the case of GMM where it assumes the whole real space. But even then, GMM does not perform poorly, rather far from. The GMM also converges very fast, within 10-20 iterations and the log likelihood settles down at an average value of -1.7k. This indicates both models are suitable for this. But, the GMM is computationally expensive because of the calculation of inverse of a 50*50 matrix in every iteration. Therefore, the BMM might be a more suited algorithm in this case.

```
[34] def EM(X):   # EM algorithm main code
       pi = np.random.rand(4)  # Randomize pi and mu
       pi = pi/np.sum(pi)
       cov = np.empty((4,50,50))
       for i in range(4):      # Initialize cov to be invertible
         G = np.random.random((50,50))
         G = G @ G.T
         cov[i] = G
       mu = np.random.rand(4,50)
       it=0
       err = 1
       logl=np.array([])       # Array to maintain log likelihood values

       while it<50:            # Set constant number of iterations
         lamda = expectation(X,mu,cov,pi) # Expectation step
         pi_new, mu_new, cov_new = maximization(X,lamda)  # Maximization step
         logl = np.append(logl,log_likelihood(X,pi_new,mu_new,cov_new))
         err = error(mu_new,cov_new,pi_new,mu,cov,pi)
         mu = mu_new                    # Update parameters
         pi = pi_new
         cov = cov_new
         it = it+1                      # Update iterations

       return mu, cov, pi, lamda, logl
```

Figure 7: EM Algorithm

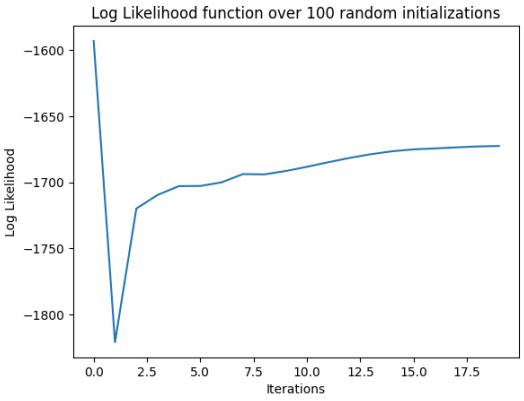

Figure 8: Plot of Log Likelihood function

1.3 K-Means Algorithm

In the Kmeans Algorithm, we classify the points into 4 clusters whose means are as far away from each other as possible. This algorithm is also called Lloyds' Algorithm. We begin by randomly initializing 4 of the data points as the cluster means. The, we do an assignment step where we assign all the points to one of the 4 clusters based on the distance between the point and the cluster mean. After the assignment, the means are recalculated using the new cluster classification. Then a reassignment is done based on the new cluster means and this process is repeated until convergence. Convergence here means, when none of the points move to a different cluster after reassignment and the cluster means effectively remain the same.

1.3.1 Reassignment step

Calculate the distance of a particular point from all 4 cluster means. Assign it to the cluster whose mean is closest to it. This ensures that the cluster means go as far away from each other in every iteration

$$z_i = \arg\min_k \|x_i - \mu_k\|^2$$

1.3.2 Recalculation of Means

After the reassignment, calculate the number points assigned to a particular cluster. Add up their data values and divide by the count. This gives u the new mean of the clusters. After this, proceed to repeat the reassignment step again

$$\mu_k = \frac{\sum_{i=1}^{n} 1(z_i = k)x_i}{\sum_{i=1}^{n} 1(z_i = k)}$$

1.3.3 Objective Function

The objective function is the sum of distance between all points to their respective cluster means. Shown below is the code for reassignment step, recalculation step and find the objective function. the value comes out to be 818.58

$$f(\mu, z, X) = \sum_{i=1}^{n} (x_i - \mu_{z_i})^2$$

1.4 Comparison between the 3 methods

The objective function for GMM is 708.9, for BMM is 2.8k, for Kmeans is 818. Clearly kmeans and GMM are the better ones if we consider Objective function. But we also need to consider the fact

```
[9]  def obj(X,z,mu):    # Objective function of the kmeans algorithm
       err=0
       # Error here is the distance between point and its assigned cluster mean

       for i in range(X.shape[0]):
         z_curr = int(z[i])
         err += np.linalg.norm(X[i] - mu[z_curr])
       return err;

[15] error = obj(X,z,mu_final)   # Calculate the objective function value
     error

     818.5819281357393
```

Figure 9: Objective Function

```
[7]  def kmeans(X, mu):          # Kmeans algorithm
       z = np.zeros(X.shape[0])    # Initialize with zeros
       mu_temp = np.zeros((4,50))
       it = 0
       err=1

       # Set a threshold for convergence
       # Error here is the difference between means of successive iterations
       while err>1e-6:
         z = reassignment(X, mu)          # Reassignment Step
         mu_temp = calculate_mean(X, z)   # Recalculation of means
         it=it+1
         err=0
         for i in range(4):
           err += np.linalg.norm(mu[i]-mu_temp[i])  # Calculate error
         mu = mu_temp                     # Update means
       return it, z, mu
```

Figure 10: Recalculation Step

```
[5]  def reassignment(X, mu):  # Reassignment step
       z = np.zeros(X.shape[0])  # Initialize with zeros
       for i in range(X.shape[0]):
         min = 1e9
         for k in range(mu.shape[0]):
           dif = X[i] - mu[k]    # Find distance between point and means
           dif = np.linalg.norm(dif)
           if dif < min:         # Find the mean which is the closest
             min=dif
             z[i] = k            # Assign the point to that cluster
       return z

[6]  def calculate_mean(X, z):   # Recalculation of means
       mu = np.zeros((4,50))     # Initialize with zeros
       n = np.zeros(4)
       for i in range(X.shape[0]):
         z_curr = int(z[i])
         mu[z_curr] = mu[z_curr] + X[i]  # Add point values to their clusters
         n[z_curr] += 1                  # Increase counter of that cluster
       for i in range(4):
         mu[i] = mu[i]/n[i]              # Find the mean
       return mu
```

Figure 11: Reassignment Step

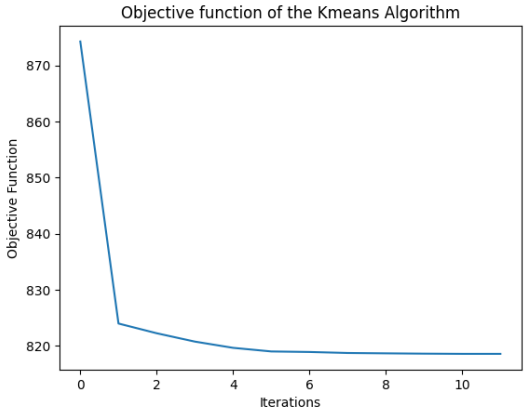

Figure 12: Objective function vs Iterations

that GMM and BMM are performing soft clustering whereas Kmeans is doing hard clustering. If all three are tested on hard clustering, the results could turn out much differently. GMM and BMM are estimation techniques which do not really have an objective function which they want to minimize. These techniques are not optimization techniques and do not have a goal of optimality in their core. Therefore, it is not ideal to compare them based on the objective function.

2 Gradient Descent and Linear Regression

We are given a dataset consisting of 10000 data points, each one having 100 feature variables and 1 label. We load this dataset onto our database and store it in a numpy array. We can then further split this into 2 matrices, a 100*10000 matrix X which represents our data points and a 1*10000 matrix which represents the labels. Now that we have set up our data as we need it, we can start with finding the analytical solution of the dataset.

2.1 Analytical Solution

The analytical solution w_{ML} is the argument which minimizes the value of the objective function over all possible choices of w. Taking the derivative of f(w) and setting it to zero, we can find the optimal value of w which is denoted as w_{ML}

$$w_{ML} = \arg\min_{w} f(w)$$
$$f(w, X, Y) = \sum_{i=1}^{n} (w^T x_i - y_i)^2$$
$$f(w, X, Y) = \|X^T w - Y\|^2$$
$$f^{'}(w, X, Y) = \ 0 \ = 2XX^T w_{ML} - 2XY$$
$$w_{ML} = (XX^T)^{-1} XY$$

2.2 Gradient Descent

Calculating the analytical solution can be computationally expensive due to the large size of the matrix and the problem of calculating inverse of such large matrices. Hence, we use gradient descent to solve this.

$$w^{t+1} = w^t - \eta^t \nabla f(w)$$
$$\nabla f(w, X, Y) = 2XX^T w - 2XY$$

where η^t is the step size we take in each step. In this problem, I have chosen the step size to be a constant of 10^{-6}. This was decided upon after thorough checking of different step sizes and the number of iterations it took for the algorithm to converge. Shown below are the code written for Gradient Descent and the plot of $\|w^t - w_{ML}\|$ against the number of iterations taken. We can observe that the difference between the gradient descent solution and the analytical solution starts converging towards 0.5, which is a very healthy sign. The gradient descent does not always converge towards the optimal solution, it just settles down at a local minima. The optimality of the solution depends on the random initialization. For practical purposes, the solution obtained from the algorithm is as good as the analytical solution.

```
[29] def grad(w, X, Y):
       T = 2*((X @ X.T) @ w) - 2*(X @ Y) # Gradient of the objective function
       return T

[30] def gradient_descent(X, Y):

       w=np.random.random((100,1)) # Initialize w with a random matrix
       error = np.array([])        # Make an array to store error values
       err_prev=0                  # Variable to maintain previous error value
       err=0                       # Variable to calculate current error value
       it=0                        # Variable to keep track of iterations

       # Error here refers to the distance between w and analytical solution
       # Set a Threshold of 1e-3 between error values as break condition
       while np.abs(err_prev-err) > 1e-3 or it<2:
         err_prev=err
         step_size = (1e-6)                  # Constant Learning rate
         w = w - step_size*grad(w,X,Y)       # Apply gradient descent
         err = np.linalg.norm(w-w_ml)        # Calculate error
         error = np.append(error,err)        # Sppend to the array
         it = it+1                           # Update iterations

       return w, error, it         # Return values of w, it and error

[31] w_gd, error_gd, it_gd = gradient_descent(X,Y) # Call the GD function
     print(it_gd)                                  # Check no of iterations
```

Figure 13: Gradient Descent Algorithm

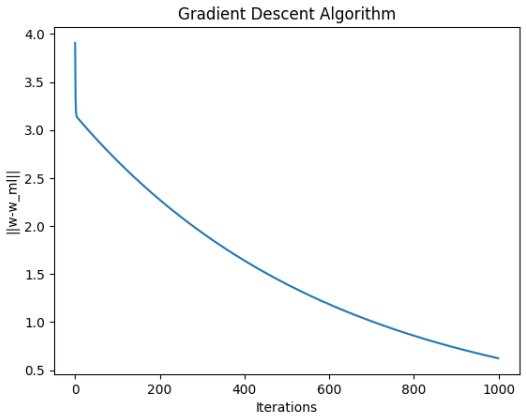

Figure 14: Error vs Iterations

2.3 Stochastic Gradient Descent

Stochastic gradient descent is an algorithm that attempts to converge to a solution much faster as compared to the gradient descent algorithm and also to ease the computational stress on the system. It takes a batch consisting of a fraction of the data points, assumes this to be the dataset and runs one iteration of the gradient descent algorithm on this. As the size of the input data is lowered, calculation is much faster and the solution converges much more rapidly. Here we take batch size k=100.

$$X_{sgd} = X[i_1, i_2, ..., i_k]$$

$$1 \leq i_1, i_2, ..., i_k \leq n$$

Shown below are the code for stochastic gradient descent algorithm and the plot of error w.r.t. number of iterations. We can see that this converges almost 3 times faster than the normal gradient descent, but converges to a solution which is much farther away from optimal as compared to the latter. The reason for this being that we are training only on part of the dataset and not on all of them.

```
[34] def stochastic_gradient_descent(X,Y,k):   # k here is the batch size

       w=np.random.random((100,1))        # Randomly initialize w
       error = np.array([])               # Declare an array to store error
       err_prev=0                         # Variable to store previous error
       err=0                              # Variable to store current error
       it=0                               # Variable to store iterations

       # Error here is the distance between w and analytical solution
       # Set a Threshold of difference in error values less than 1e-3
       while np.abs(err_prev-err) > 1e-3 or it<2:
         err_prev=err
         step_size = (1e-6)                       # Constant learning rate
         ind = np.random.randint(0,10000,size=k)  # Take a batch of 100 data
         X_sgd = X[:, ind]                        # Create the batch of X
         Y_sgd = Y[ind, :]                        # Create the batch of Y
         w = w - step_size*grad(w,X_sgd,Y_sgd)    # Perform gradient descent
         err = np.linalg.norm(w-w_ml)             # Find error
         error = np.append(error,err)             # Append to the error array
         it = it+1                                # Update iterations

       return w, error, it              # Return w, it and the error array

[37] w_sgd, error_sgd, it_sgd = stochastic_gradient_descent(X,Y,100)
     print(it_sgd)             # Run the algorithm and print the results
```

Figure 15: Gradient Descent Algorithm

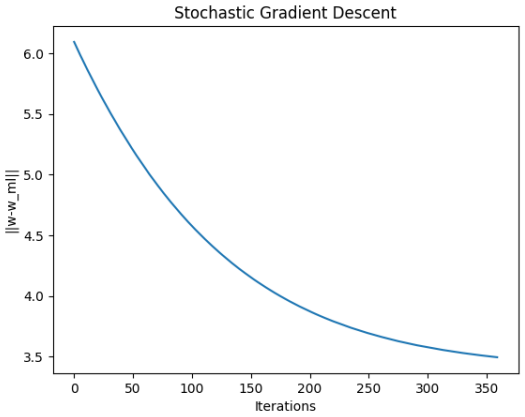

Figure 16: Error vs Iterations

2.4 Ridge Regression

First we need to normalize the dataset. It is necessary to bring the mean and standard deviation of the dataset to 0 and 1 respectively before running Ridge Regression. After normalization, we choose specific values of λ to run the algorithm on and cross-validate the error generated by each of these parameters in order to choose the optimal value. For each value of λ, we split the dataset into 2 parts, training and cross-validation sets. 80pc of the data goes into the training set and 20pc into the CV set. We run ridge regression on the training set and find the optimal w and then find error of prediction on the CV set using the objective function. The objective function and the gradient for Ridge Regression is given by

$$\begin{aligned}
X_{norm} &= \frac{X - mean(X)}{std(X)} \\
f_{ridge}(w) &= \sum_{i=1}^{n} (w^T x_i - y_i)^2 + \lambda \|w\|^2 \\
Error(w, X, Y) &= \sum_{i=1}^{n} (w^T x_i - y_i)^2 \\
\nabla f_{ridge}(w, X, Y, \lambda) &= 2XX^T w - 2XY + 2\lambda w \\
w_{ridge}^{t+1} &= w_{ridge}^{t} - \eta^t \nabla f_{ridge}(w, X_{test}, Y_{test}, \lambda) \\
Error_{ridge} &= Error(w_{ridge}, X_{CV}, Y_{CV})
\end{aligned}$$

Shown below are the codes for normalization of data, the ridge regression algorithm and the cross validation of λ values. Also shown is a plot of the Cross Validation error for each parameter involved.

The optimal λ is found to be 50. We now find the optimal w_{ridge}. The variation of error with iterations is shown in the plot below. The same code mentioned above for Ridge Regression was used.

2.5 Test data

We can now test our observations on the test data. The provided test data is loaded and run using $w = w_{ML}$ and $w = w_{ridge}$ and the results are shown below. The objective function values are pretty

```
[39] def normalize_X(X):
       mean = np.mean(X,axis=1)          # Caluculate mean of the data points
       mean = mean.reshape(100,1)        # Reshape to a column vector
       X = X - mean                      # Center the dataset

       std = 0                           # Calculate Standard Deviation
       for i in range(X.shape[1]):
         std += np.linalg.norm(X[:,i])
       std = std/X.shape[1]
       std = np.sqrt(std)
       X = X/std                         # Bring Devaition of the matrix to 1

       return mean, std, X               # Values of mean and std will be
                                         # required to normalize test data
```

Figure 17: Normalization of data

```
[13] def ridge_grad(w, X, Y, lamda):
       # Calculate gradient of new objective function
       # Lamda being the regularization factor
       T = 2*((X @ X.T) @ w) - 2*(X @ Y) + 2*lamda*w
       return T

[14] def ridge_regression(X, Y, lamda):

       w=np.random.random((100,1))     # Randomize w
       err_prev=1                      # Variable to store previous error
       err=0                           # Variable to calculate current error
       error = np.array([])

       # Set a Threshold of 1e-3 between successive error values
       # Error here is difference between current w and analytical solution
       while np.abs(err_prev-err) > 1e-3:
         err_prev=err
         step_size = (1e-6)                          # Constant step size
         w = w - step_size*ridge_grad(w,X,Y,lamda)  # Run gradient descent
         err = np.linalg.norm(w-w_ml)                # Calculate error
         error = np.append(error,err)

       return w, error                 # Return the optimal value found
```

Figure 18: Ridge Regression

```
[43] def cross_validation(X, Y, lamda):

       # Split the datset into training and cross validation set
       # Put 80% of the data into the training set
       # Rest 20% go into the cross validation set
       X_train = X[:, :8000]
       X_cv = X[:, 8000:]
       Y_train = Y[:8000, :]
       Y_cv = Y[8000:, :]
       error = np.array([])

       for i in range(lamda.shape[0]):
         w = ridge_regression(X_train,Y_train,lamda[i])  # Run on training data
         w_array = np.array(w)                           # Convert to np array
         e = X_cv.T @ w_array - Y_cv                     # Find error on CV set
         er = np.linalg.norm(e)                          # Find the norm
         error = np.append(error,er)                     # Append to array

       return error
```

Figure 19: Cross-Validation

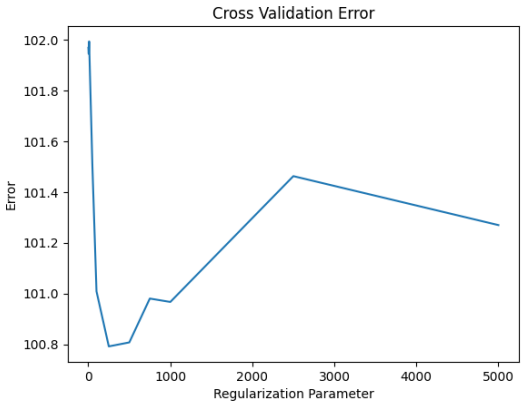

Figure 20: CV error for each λ

close with 51.01 for analytical solution and 50.77 for ridge regression. We can see that the ridge parameter performs a little better on the test data but this is heavily dependent on the randomization used in these algorithms. As the values are pretty close, it would be better to use ridge regression in this case. Ridge provides a solution which is not too large (constrained to some θ which depends on λ) and it is not computationally very expensive. The analytical solution needs to find the inverse which will be very difficult as the training set goes into the millions, whereas ridge regression can handle this.

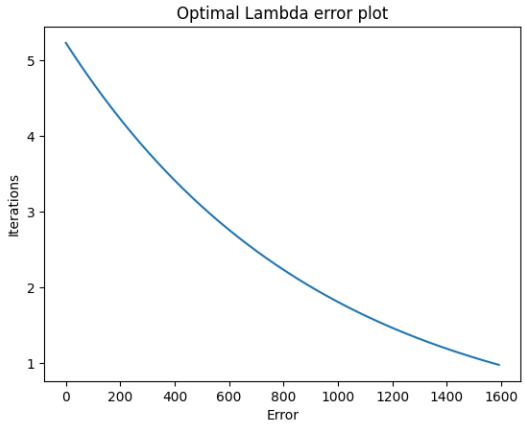

Figure 21: Ridge Regression Error

```
[35] Data_test=np.array(df_test)   # Load the Test data into an array
     Y_test = Data_test[:, 100:]   # Separate the labels from the set
     X_test = Data_test[:, :100]   # Separate the feature labels
     X_test = X_test.T

[36] X_test = X_test - mean_x      # Normalize the test data
     X_test = X_test/std_x

[37] def mse(X, Y, w):             # function to find objective function
       err = X.T @ w - Y
       return np.linalg.norm(err)

[38] err_w_ml = mse(X_test, Y_test, w_ml)   # Objective function with w=w_ml
     err_w_r = mse(X_test, Y_test, w_ridge) # Objective function with w=w_ridge
     print(err_w_ml)                        # Print the 2 values
     print(err_w_r)

     51.018747109394674
     50.7702112368051
```

Figure 22: Test Data