CS5691
Pattern Recognition and Machine Learning
Assignment 2

Expectation-Maximization and Regression

Instructor: Prof. Arun Rajkumar

Student: Janmenjaya Panda

Release Date: 13/03/2024

Submission Deadline: 14/04/2024

Department of Computer Science & Engineering

Indian Institute of Technology Madras

Contents

List of Figures

List of Tables

List of Algorithms

1 Expectation- Maximization

You are given a data-set with 400 data points in $\{0,1\}$ 50 generated from a mixture of some distribution in the file A2Q1.csv. (Hint: Each datapoint is a flattened version of a $\{0,1\}$ 10×5 matrix.)

1. Determine which probabilisitic mixture could have generated this data (It is not a Gaussian mixture). Derive the EM algorithm for your choice of mixture and show your calculations. Write a piece of code to implement the algorithm you derived by setting the number of mixtures $K = 4$. Plot the log-likelihood (averaged over 100 random initializations) as a function of iterations.

2. Assume that the same data was in fact generated from a mixture of Gaussians with 4 mixtures. Implement the EM algorithm and plot the log-likelihood (averaged over 100 random initializations of the parameters) as a function of iterations. How does the plot compare with the plot from part (i)? Provide insights that you draw from this experiment.

3. Run the K-means algorithm with $K = 4$ on the same data. Plot the objective of K-means as a function of iterations.

4. Among the three different algorithms implemented above, which do you think you would choose to for this dataset and why?

This section discusses various mixture models and compares them with the K-means algorithm. For this problem we assume individual data points are in $\{0,1\}^{50}$ instead of $\{0,1\}^{10\times 5}$.

1.1 Bernoulli Mixture Model

Clearly, each data point lies in $\{0,1\}^{50}$. That is each of them is a 50-dimensional binary vector. Thus, we may assume that a **mixture of multivariate Bernoulli distribution corresponding to $\{0,1\}^{50}$** could have generated this data. Followingly, we discuss the Bernoulli Mixture Model. Please refer to the paper entitled "Bernoulli mixture models for binary images" by Alfons Juan Enrique Vidal [1] for a detailed overview.

1.1.1 The generative story

We have N data points, each of which lies in $\{0,1\}^d$. The generative story goes as follows:

1. We have K mixtures. Pick a mixture z_i from which the ith data point comes for $z_i \in \{1,2,3,\dots,K\}$

$$\mathbb{P}(z_i = l) = \pi_l \tag{1}$$

   where $\sum_{l=1}^{K} = 1$ and for each l, it holds $0 \leqslant \pi_l \leqslant 1$.

2. Generate the ith data point from that mixture.

$$X_i \sim \mathcal{B}(\boldsymbol{\mu}_{z_i}) \tag{2}$$

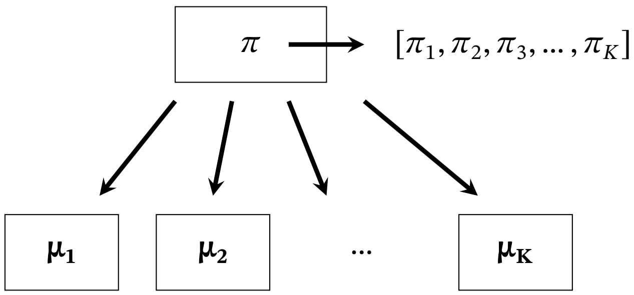


Figure 1: The data generative model in Bernoulli Mixuture

where $X_i \in \{0,1\}^d$, the term $\mathcal{B}(\boldsymbol{\mu}_{z_i})$ refers to the z_ith multivariate Bernoulli distribution in $\{0,1\}^d$ and $\boldsymbol{\mu}_{z_i} \in (0,1)^d$.

Note that here $\{X_1, X_2, X_3, \dots, X_N\}$ are the observed data points and $\{z_1, z_2, z_3, \dots, z_N\}$ are unobserved/ latent parameters. Thus, this model is a latent variable model. Clearly, We have $K-1$ parameters pertaining to π and d parameters pertaining to each of the K Bernoulli generators. Hence in total $dK + K - 1$ or $(d+1)K - 1$ numbers of parameters.

1.1.2 The likelihood and log-likelihood estimations

The likelihood estimation function goes as follows:

$$\begin{aligned} L(X; \boldsymbol{\pi}, \boldsymbol{\mu}) &= \prod_{i=1}^{N} \mathbb{P}(X_i | \boldsymbol{\pi}, \boldsymbol{\mu}) \\ &= \prod_{i=1}^{N} \left[\sum_{k=1}^{K} \pi_k \mathbb{P}(X_i | \boldsymbol{\mu}_\mathbf{k}) \right] \\ &= \prod_{i=1}^{N} \left[\sum_{k=1}^{K} \pi_k \left(\prod_{j=1}^{d} \mu_{kj}^{X_{ij}} (1 - \mu_{kj})^{1 - X_{ij}} \right) \right] \end{aligned} \tag{3}$$

where μ_{kj} refers to the jth entry of $\boldsymbol{\mu}_\mathbf{k}$ and likewise X_{ij} refers to the jth entry of X_i for some $1 \leqslant j \leqslant d$.

The log-likelihood estimation function goes as follows: The likelihood estimation function goes as follows:

$$\begin{aligned} \mathrm{LogL}(X; \boldsymbol{\pi}, \boldsymbol{\mu}) &= \log \prod_{i=1}^{N} \left[\sum_{k=1}^{K} \pi_k \left(\prod_{j=1}^{d} \mu_{kj}^{X_{ij}} (1 - \mu_{kj})^{1 - X_{ij}} \right) \right] \\ &= \sum_{i=1}^{N} \log \left[\sum_{k=1}^{K} \pi_k \left(\prod_{j=1}^{d} \mu_{kj}^{X_{ij}} (1 - \mu_{kj})^{1 - X_{ij}} \right) \right] \end{aligned} \tag{4}$$

1.1.3 Introducing latent variables λs

For ith data point introduce a set of K new latent variables, that is — $\{\lambda^i_1, \lambda^i_2, \lambda^i_3, \ldots, \lambda^i_K\}$ such that $\sum_{k=1}^{K} \lambda^i_k = 1$ and for each $1 \leqslant i \leqslant N$ and for each $1 \leqslant k \leqslant K$, it holds that $0 \leqslant \lambda^i_k \leqslant 1$.

Note that:

$$\mathrm{LogL}(X; \boldsymbol{\pi}, \boldsymbol{\mu}) = \sum_{i=1}^{N} \log \left[\sum_{k=1}^{K} \lambda^i_k \left(\frac{\pi_k \prod_{j=1}^{d} \mu_{kj}^{X_{ij}} (1 - \mu_{kj})^{1 - X_{ij}}}{\lambda^i_k} \right) \right]$$

By Jensen's inequality

$$\geqslant \sum_{i=1}^{N} \sum_{k=1}^{K} \lambda^i_k \log \left(\frac{\pi_k \prod_{j=1}^{d} \mu_{kj}^{X_{ij}} (1 - \mu_{kj})^{1 - X_{ij}}}{\lambda^i_k} \right)$$

Let $\text{Modified-LogL}(X; \boldsymbol{\lambda}, \boldsymbol{\pi}, \boldsymbol{\mu}) := \sum_{i=1}^{N} \sum_{k=1}^{K} \lambda^i_k \log \left(\frac{\pi_k \prod_{j=1}^{d} \mu_{kj}^{X_{ij}} (1 - \mu_{kj})^{1 - X_{ij}}}{\lambda^i_k} \right)$.

Note that $\text{Modified-LogL}(X; \boldsymbol{\lambda}, \boldsymbol{\pi}, \boldsymbol{\mu})$ provides a lower bound for $\mathrm{LogL}(X; \boldsymbol{\pi}, \boldsymbol{\mu})$ for any choice of valid $\boldsymbol{\lambda}$.

1.1.4 Maximizing Modified-LogL over $(\boldsymbol{\pi}, \boldsymbol{\mu})$ by fixing $\boldsymbol{\lambda}$

Firstly, the objective is to find $\hat{\boldsymbol{\pi}}$ that is the solution to

$$\begin{aligned} \max_{\boldsymbol{\pi}} \quad & \text{Modified-LogL}(X; \boldsymbol{\lambda}, \boldsymbol{\pi}, \boldsymbol{\mu}) \\ \text{such that} \quad & \sum_{k=1}^{K} \pi_k = 1 \\ & 0 \leqslant \pi_k \leqslant 1 \quad \text{for each} \quad 1 \leqslant k \leqslant K \end{aligned}$$

Let us introduce the Lagrangian $\mathcal{L}(\boldsymbol{\pi}, \eta)$ with parameter η:

$$\mathcal{L}(\boldsymbol{\pi}, \eta) := \sum_{i=1}^{N} \sum_{k=1}^{K} \lambda^i_k \log \left(\frac{\pi_k \prod_{j=1}^{d} \mu_{kj}^{X_{ij}} (1 - \mu_{kj})^{1 - X_{ij}}}{\lambda^i_k} \right) - \eta \left(1 - \sum_{k=1}^{K} \pi_k \right) \tag{5}$$

Now taking partial derivative of $\mathcal{L}(\boldsymbol{\pi}, \eta)$ wrt π_l, we obtain:

$$
\begin{aligned}
\frac{\partial}{\partial \pi_l}\mathcal{L}(\boldsymbol{\pi}, \eta) &= \frac{\partial}{\partial \pi_l}\left[\sum_{i=1}^{N}\sum_{k=1}^{K}\lambda_k^i \log\left(\frac{\pi_k \prod_{j=1}^{d}\mu_{kj}^{X_{ij}}(1-\mu_{kj})^{1-X_{ij}}}{\lambda_k^i}\right) - \eta\left(1-\sum_{k=1}^{K}\pi_k\right)\right] \\
&= \frac{\partial}{\partial \pi_l}\sum_{i=1}^{N}\sum_{k=1}^{K}\lambda_k^i \log\left(\frac{\pi_k \prod_{j=1}^{d}\mu_{kj}^{X_{ij}}(1-\mu_{kj})^{1-X_{ij}}}{\lambda_k^i}\right) + \eta \\
&= \frac{\partial}{\partial \pi_l}\sum_{i=1}^{N}\sum_{k=1}^{K}\lambda_k^i \left[\log\left(\pi_k \prod_{j=1}^{d}\mu_{kj}^{X_{ij}}(1-\mu_{kj})^{1-X_{ij}}\right) - \log\lambda_k^i\right] + \eta \\
&= \frac{\partial}{\partial \pi_l}\sum_{i=1}^{N}\sum_{k=1}^{K}\lambda_k^i \log\left(\pi_k \prod_{j=1}^{d}\mu_{kj}^{X_{ij}}(1-\mu_{kj})^{1-X_{ij}}\right) + \eta \\
&= \sum_{i=1}^{N}\frac{\partial}{\partial \pi_l}\sum_{k=1}^{K}\lambda_k^i \log\left(\pi_k \prod_{j=1}^{d}\mu_{kj}^{X_{ij}}(1-\mu_{kj})^{1-X_{ij}}\right) + \eta \\
&= \sum_{i=1}^{N}\lambda_l^i\frac{\partial}{\partial \pi_l} \log\left(\pi_l \prod_{j=1}^{d}\mu_{lj}^{X_{ij}}(1-\mu_{lj})^{1-X_{ij}}\right) + \eta \\
&= \sum_{i=1}^{N}\lambda_l^i\frac{\partial}{\partial \pi_l} \left[\log\pi_l + \sum_{j=1}^{d}\log\left(\mu_{lj}^{X_{ij}}(1-\mu_{lj})^{1-X_{ij}}\right)\right] + \eta \\
&= \left(\sum_{i=1}^{N}\lambda_l^i\right)\frac{1}{\pi_l} + \eta
\end{aligned}
$$

Setting this to zero we obtain:

$$\pi_l = -\frac{1}{\eta}\sum_{i=1}^{N}\lambda_l^i$$

Let's have a look at the constraint on $\boldsymbol{\pi}$.
That is —

$$
\begin{aligned}
&\sum_{m=1}^{K}\pi_m = 1 \\
\implies &\sum_{m=1}^{K}-\frac{1}{\eta}\sum_{i=1}^{N}\lambda_m^i = 1
\end{aligned}
$$

$$
\begin{aligned}
&\implies -\frac{1}{\eta}\sum_{i=1}^{N}\sum_{m=1}^{K}\lambda_m^i = 1\\
&\implies -\frac{1}{\eta}\sum_{i=1}^{N}1 = 1\\
&\implies \eta = -N
\end{aligned}
$$

Thus, using the Lagrangian method, we obtain:

$$
\hat{\pi}_k = \frac{\sum\limits_{i=1}^{N}\lambda_k^i}{N} \tag{6}
$$

Analogously, we want to find $\hat{\boldsymbol{\mu}}$ that is the solution to

$$
\begin{aligned}
\max_{\boldsymbol{\mu}} \quad & \text{Modified-LogL}(X;\boldsymbol{\lambda},\boldsymbol{\pi},\boldsymbol{\mu})\\
\text{such that} \quad & 0 \leqslant \mu_k \leqslant 1 \quad \text{for each} \quad 1 \leqslant k \leqslant K
\end{aligned}
$$

Taking partial derivative of Modified-LogL$(X;\boldsymbol{\lambda},\boldsymbol{\pi},\boldsymbol{\mu})$ wrt μ_{lm} (that is the mth entry of μ_l), we obtain:

$$
\begin{aligned}
\frac{\partial}{\partial\mu_{lm}}\sum_{i=1}^{N}\sum_{k=1}^{K}\lambda_k^i\log\left(\frac{\pi_k\prod\limits_{j=1}^{d}\mu_{kj}^{X_{ij}}(1-\mu_{kj})^{1-X_{ij}}}{\lambda_k^i}\right) &= \frac{\partial}{\partial\mu_{lm}}\sum_{i=1}^{N}\sum_{k=1}^{K}\lambda_k^i\left[\log\left(\pi_k\prod_{j=1}^{d}\mu_{kj}^{X_{ij}}(1-\mu_{kj})^{1-X_{ij}}\right)-\log\lambda_k^i\right]\\
&= \frac{\partial}{\partial\mu_{lm}}\sum_{i=1}^{N}\sum_{k=1}^{K}\lambda_k^i\left[\log\left(\pi_k\prod_{j=1}^{d}\mu_{kj}^{X_{ij}}(1-\mu_{kj})^{1-X_{ij}}\right)-\log\lambda_k^i\right]\\
&= \sum_{i=1}^{N}\frac{\partial}{\partial\mu_{lm}}\sum_{k=1}^{K}\lambda_k^i\log\left(\pi_k\prod_{j=1}^{d}\mu_{kj}^{X_{ij}}(1-\mu_{kj})^{1-X_{ij}}\right)\\
&= \sum_{i=1}^{N}\frac{\partial}{\partial\mu_{lm}}\lambda_l^i\log\left(\pi_l\prod_{j=1}^{d}\mu_{lj}^{X_{ij}}(1-\mu_{lj})^{1-X_{ij}}\right)\\
&= \sum_{i=1}^{N}\frac{\partial}{\partial\mu_{lm}}\lambda_l^i\left[\log\pi_l+\sum_{j=1}^{d}\big(X_{ij}\log\mu_{lj}+(1-X_{ij})\log(1-\mu_{lj})\big)\right]\\
&= \sum_{i=1}^{N}\lambda_l^i\left[\left(X_{im}\frac{1}{\mu_{lm}}-(1-X_{im})\frac{1}{1-\mu_{lm}}\right)\right]\\
&= \frac{1}{\mu_{lm}(1-\mu_{lm})}\sum_{i=1}^{N}\lambda_l^i(X_{im}-\mu_{lm})
\end{aligned}
$$

Setting this to zero we obtain:

$$\hat{\mu}_{lm} = \frac{\sum_{i=1}^{N} \lambda_l^i X_{lm}}{\sum_{i=1}^{N} \lambda_l^i}$$

In other words,

$$\boxed{\hat{\mu}_k = \frac{\sum_{i=1}^{N} \lambda_k^i X_i}{\sum_{i=1}^{N} \lambda_k^i} \qquad (7)}$$

1.1.5 Maximizing Modified-LogL over $\boldsymbol{\lambda}$ by fixing $(\boldsymbol{\pi}, \boldsymbol{\mu})$

The objective is to find $\hat{\boldsymbol{\lambda}}$ that is the solution to

$$\begin{aligned} \max_{\boldsymbol{\lambda}} \quad & \text{Modified-LogL}(X; \boldsymbol{\lambda}, \boldsymbol{\pi}, \boldsymbol{\mu}) \\ \text{such that} \quad & \sum_{k=1}^{K} \lambda_k^i = 1 \quad \text{for each} \quad 1 \leqslant i \leqslant N \\ & 0 \leqslant \lambda_k^i \leqslant 1 \quad \text{for each} \quad 1 \leqslant k \leqslant K \quad \text{and} \quad 1 \leqslant i \leqslant N \end{aligned}$$

Let us introduce the Lagrangian $\mathcal{L}(\boldsymbol{\lambda}, \eta)$ with parameters η_i for $1 \leqslant i \leqslant N$:

$$\mathcal{L}(\boldsymbol{\lambda}, \eta) := \sum_{i=1}^{N} \sum_{k=1}^{K} \lambda_k^i \log \left(\frac{\pi_k \prod_{j=1}^{d} \mu_{kj}^{X_{ij}} (1 - \mu_{kj})^{1 - X_{ij}}}{\lambda_k^i} \right) - \sum_{i=1}^{N} \eta_i \left\{ \left(1 - \sum_{k=1}^{K} \lambda_k^i \right) \right\} \tag{8}$$

Now taking partial derivative of $\mathcal{L}(\boldsymbol{\pi}, \eta)$ wrt λ_l^m, we obtain:

$$\begin{aligned} \frac{\partial}{\partial \lambda_l^m} \mathcal{L}(\boldsymbol{\lambda}, \eta) &= \frac{\partial}{\partial \lambda_l^m} \left[\sum_{i=1}^{N} \sum_{k=1}^{K} \lambda_k^i \log \left(\frac{\pi_k \prod_{j=1}^{d} \mu_{kj}^{X_{ij}} (1 - \mu_{kj})^{1 - X_{ij}}}{\lambda_k^i} \right) - \sum_{i=1}^{N} \left\{ \eta_i \left(1 - \sum_{k=1}^{K} \lambda_k^i \right) \right\} \right] \\ &= \frac{\partial}{\partial \lambda_l^m} \left[\lambda_l^m \log \left(\frac{\pi_l \prod_{j=1}^{d} \mu_{lj}^{X_{mj}} (1 - \mu_{lj})^{1 - X_{mj}}}{\lambda_l^m} \right) \right] + \eta_m \end{aligned}$$

$$= \frac{\partial}{\partial \lambda_l^m}\left[\lambda_l^m\left(\log\left(\pi_l \prod_{j=1}^{d} \mu_{lj}^{X_{mj}}(1-\mu_{lj})^{1-X_{mj}}\right) - \log \lambda_l^m\right)\right] + \eta_m$$
$$= \log\left(\pi_l \prod_{j=1}^{d} \mu_{lj}^{X_{mj}}(1-\mu_{lj})^{1-X_{mj}}\right) - \log \lambda_l^m - 1 + \eta_m$$

Setting this to zero, we obtain:

$$\lambda_l^m = \left(e^{-1+\eta_m}\right) \pi_l \prod_{j=1}^{d} \mu_{lj}^{X_{mj}}(1-\mu_{lj})^{1-X_{mj}}$$

Also,

$$\sum_{p=1}^{K} \lambda_l^p = 1$$
$$\implies \sum_{p=1}^{K}\left[\left(e^{-1+\eta_l}\right) \pi_p \prod_{j=1}^{d} \mu_{pj}^{X_{mj}}(1-\mu_{pj})^{1-X_{mj}}\right] = 1$$
$$\implies e^{-1+\eta_l} = \frac{1}{\sum\limits_{p=1}^{K} \pi_p \prod\limits_{j=1}^{d} \mu_{pj}^{X_{mj}}(1-\mu_{pj})^{1-X_{mj}}}$$

Substituting the value of $e^{-1+\eta_l}$ in the above expression of λ_l^m, we obtain:

$$\begin{aligned}\hat{\lambda}_k^i &= \frac{\pi_k \mathbb{P}(X_i|\boldsymbol{\mu_k})}{\sum\limits_{j=1}^{K} \pi_j \mathbb{P}(X_i|\boldsymbol{\mu_j})} \\ &= \frac{\pi_k \left(\prod\limits_{l=1}^{d} \mu_{kl}^{X_{il}}(1-\mu_{kl})^{(1-X_{il})}\right)}{\sum\limits_{j=1}^{K} \pi_j \left(\prod\limits_{l=1}^{d} \mu_{jl}^{X_{il}}(1-\mu_{jl})^{(1-X_{il})}\right)}\end{aligned} \tag{9}$$

1.1.6 The EM algorithm

The EM algorithm goes as follows:

Algorithm 1 : Expectation Maximization (Bernoulli Mixture Model)

```
Input: Dataset: $\{X_i\}_{i=1}^N$ where $X_i \in \{0,1\}^d$; # of Mixtures: $K$; Tolerance: $\epsilon$
Initialization: $\boldsymbol{\mu^o} = \{\boldsymbol{\mu_1^o}, \boldsymbol{\mu_2^o}, \boldsymbol{\mu_3^o}, \dots, \boldsymbol{\mu_k^o}\}$ and $\boldsymbol{\pi^o} = \{\boldsymbol{\pi_1^o}, \boldsymbol{\pi_2^o}, \boldsymbol{\pi_3^o}, \dots, \boldsymbol{\pi_k^o}\}$
while max $\left(\left|\pi^{t+1} - \pi^t\right|, \left|\mu^{t+1} - \mu^t\right|\right) \geqslant \epsilon$ do
    $\boldsymbol{\lambda}^{t+1} \leftarrow \arg\max_{\boldsymbol{\lambda}} \text{Modified-LogL}(X; \boldsymbol{\pi^t}, \boldsymbol{\mu^t}, \boldsymbol{\lambda})$        ▷ Expectation Step
    $\boldsymbol{\pi}^{t+1} \leftarrow \arg\max_{\boldsymbol{\pi}} \text{Modified-LogL}(X; \boldsymbol{\pi}, \boldsymbol{\mu^t}, \boldsymbol{\lambda^t})$        ▷ Maximization Step
    $\boldsymbol{\mu}^{t+1} \leftarrow \arg\max_{\boldsymbol{\mu}} \text{Modified-LogL}(X; \boldsymbol{\pi^t}, \boldsymbol{\mu}, \boldsymbol{\lambda^t})$        ▷ Maximization Step
end while
```

1.1.7 Implementing EM algorithm

The following code implements the EM algorithm for the Bernoulli Mixture Model:

```python
class BernoulliMixtureEM:
    def __init__(self, num_components=4, max_iteration=50, tolerance=1e-6,
     ↪  num_random_inits=10):
        self.num_components = num_components
        self.max_iteration = max_iteration
        self.tolerance = tolerance
        self.num_random_inits = num_random_inits
        self.means = None
        self.weights = None
        self.log_likelihoods = None
        self.responsibilities = None

    def _e_step(self, data):
        num_points, num_features = data.shape
        self.responsibilities = np.zeros((num_points, self.num_components))
        for i in range(num_points):
            for k in range(self.num_components):
                self.responsibilities[i, k] = self.weights[k] * np.prod(self.means[:, k] **
                 ↪  data.T[:, i] * (1 - self.means[:, k]) ** (1 - data.T[:, i]))

        sum_responsibilities_k = np.sum(self.responsibilities, axis=1, keepdims=True)
        for i in range(num_points):
            for k in range(self.num_components):
                if sum_responsibilities_k[i, 0] !=0:
                    self.responsibilities[i, k] /= sum_responsibilities_k[i, 0]
                else:
                    self.responsibilities[i, k] = 0

        return self.responsibilities

    def _m_step(self, data):
        num_points, num_features = data.shape
        prev_means = copy.deepcopy(self.means)

        self.means = data.T @ self.responsibilities
        sum_responsibilities_N = np.sum(self.responsibilities, axis=0, keepdims=True)

        for i in range(num_features):
            for k in range(self.num_components):
                if sum_responsibilities_N[0, k] != 0:
                    self.means[i, k] /= sum_responsibilities_N[0, k]
                else:
                    self.means[i, k] = 0

        self.means = np.clip(self.means, 0, 1)
        self.weights = np.mean(self.responsibilities, axis=0)
        return prev_means

    def cluster_assignment(self, data):
        return self.responsibilities

    def _log_likelihood(self, data):
        num_points, num_features = data.shape
        log_probs = np.zeros((num_points, self.num_components))
        for i in range(num_points):
            for k in range(self.num_components):
                log_probs[i, k] = self.weights[k] * np.prod(self.means[:, k] ** data.T[:, i] *
                 ↪  (1 - self.means[:, k]) ** (1 - data.T[:, i]))

        log_probs = np.maximum(log_probs, epsilon)
        log_likelihood = np.sum(np.log(np.sum(log_probs, axis=1)))
        return log_likelihood

    def fit(self, data):
        num_points, num_features = data.shape

        log_likelihoods = np.zeros((self.num_random_inits, self.max_iteration))

        for random_initialization in tqdm(range(self.num_random_inits), desc="Random
         ↪  Initialization", leave=False):
            np.random.seed(random_initialization)
            self.means = np.random.rand(num_features, self.num_components)
            self.weights = np.ones(self.num_components)
            self.weights /= np.sum(self.weights)
            log_likelihoods_per_init = []

            for itr in range(self.max_iteration):
                # E-step
                self.responsibilities = self._e_step(data)

                # M-step
                prev_means = self._m_step(data)

                # Log-likelihood
                log_likelihood_val = self._log_likelihood(data)
                log_likelihoods_per_init.append(log_likelihood_val)

                # Check for convergence
                if itr > 0 and np.max(np.abs(self.means - prev_means)) < self.tolerance:
                    break

            log_likelihoods_per_init = np.array(log_likelihoods_per_init)  # Convert to numpy
             ↪  array
            # Fill the remaining entries with the last computed log-likelihood value
            if len(log_likelihoods_per_init) < self.max_iteration:
                last_log_likelihood = log_likelihoods_per_init[-1]
                log_likelihoods_per_init = np.pad(log_likelihoods_per_init, (0,
                 ↪   self.max_iteration - len(log_likelihoods_per_init)),
                 ↪   constant_values=last_log_likelihood)

            log_likelihoods[random_initialization] = log_likelihoods_per_init

        self.log_likelihoods = np.mean(log_likelihoods, axis=0)

    def plot_log_likelihood(self, filename=None):
        plt.plot(np.arange(1, len(self.log_likelihoods) + 1), self.log_likelihoods,
         ↪   color='red')
        plt.xlabel('Iterations')
        plt.ylabel('Log-Likelihood')
        plt.title('Log-Likelihood vs. Iterations (BMM)')
        if filename:
            plt.savefig(filename, format='pdf', bbox_inches='tight')
        else:
            plt.show()
```

1.1.8 Parameter Values

Followingly we mention the values of the parameters used.

Parameters	Values
# of Components	4
maximum # of iterations	50
Tolerance	1e-10
epsilon	1e-20
# of random initialization	100

Table 1: Parameters used for Bernoulli Mixture Model

1.1.9 Results

Note that at the end of the training, the **value of log-likelihood obtained is -6634.358121844662. The weights π obtained for each of the clusters for a sample random initialization is:**

```
Weights:  [0.0025     0.2510236  0.72882715 0.01764925]
```

The plot of log-likelihood (averaged over 100 random initialization) against iterations goes as follows:

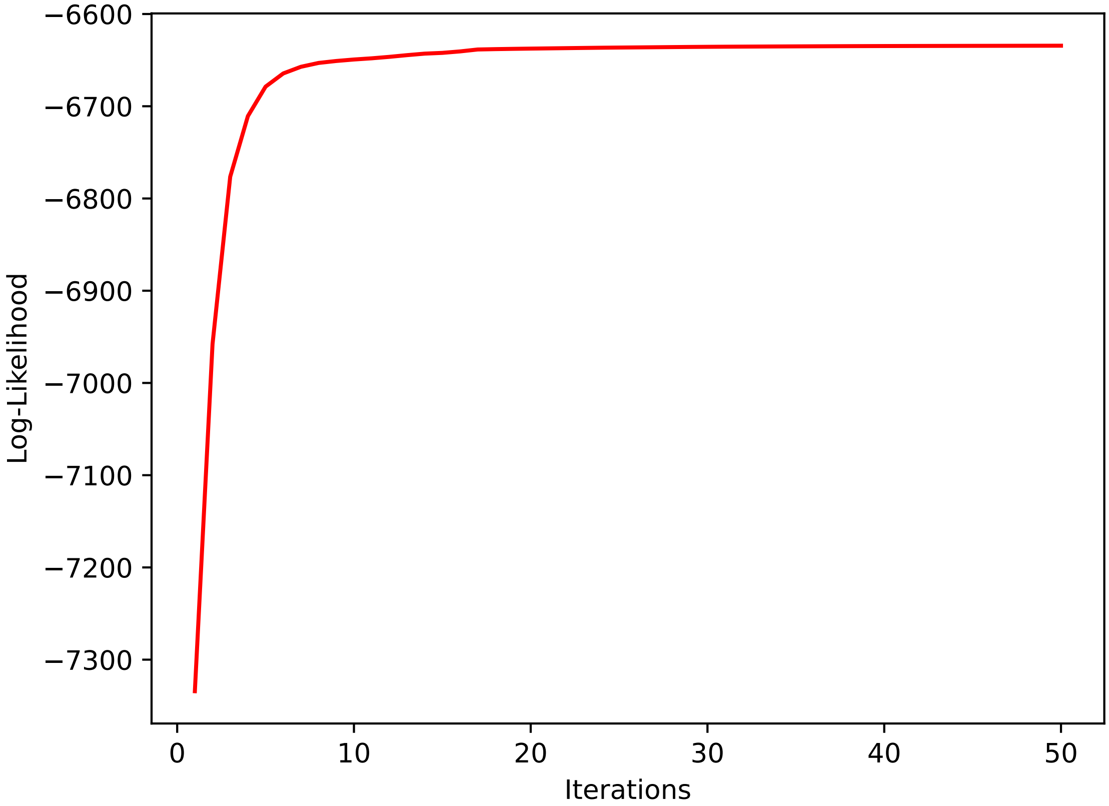


Figure 2: Log Likelihood vs Iterations in Bernoulli Mixture Model

1.2 Gaussian Mixture Model

Here we assume that a mixture of multivariate Gaussian distribution corresponding to $\mathbb{R}^{50}$ has generated this data. Please refer to the paper entitled "The EM algorithm for multi-dimensional Gaussian mixture model" by Qian Wang, Jian Wang [2] for a detailed overview.

1.2.1 The generative story

We have N data points, each of which lies in $\mathbb{R}^d$. The generative story goes as follows:

1. We have K mixtures. Pick a mixture z_i from which the ith data point comes for $z_i \in \{1, 2, 3, \dots, K\}$

$$\mathbb{P}(z_i = l) = \pi_l \tag{10}$$

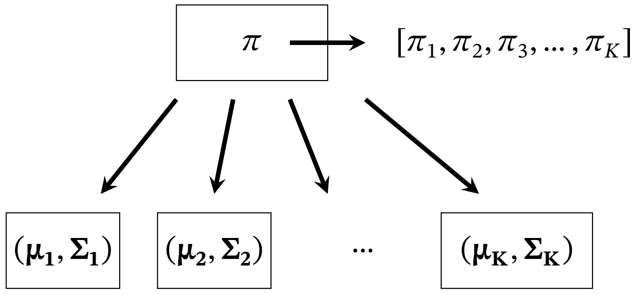
   where $\sum_{l=1}^{K} = 1$ and for each l, it holds $0 \leqslant \pi_l \leqslant 1$.

2. Generate the ith data point from that mixture.

$$X_i \sim \mathcal{N}(\boldsymbol{\mu}_{z_i}, \boldsymbol{\Sigma}_{z_i}) \tag{11}$$

   where $X_i \in \mathbb{R}^d$, the term $\mathcal{N}(\boldsymbol{\mu}_{z_i}, \boldsymbol{\Sigma}_{z_i})$ refers to the z_ith multivariate Gaussian distribution in $\mathbb{R}^d$, $\boldsymbol{\mu}_{z_i} \in \mathbb{R}^d$ is the mean and $\boldsymbol{\Sigma}_{z_i} \in \mathbb{R}^{d \times d}$ is the covariance matrix.

Note that here $\{X_1, X_2, X_3, \dots, X_N\}$ are the observed data points and $\{z_1, z_2, z_3, \dots, z_N\}$ are unobserved/ latent parameters. Thus, this model is a latent variable model.



π
$[\pi_1, \pi_2, \pi_3, \dots, \pi_K]$
$(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$
$(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$
$\dots$
$(\boldsymbol{\mu}_K, \boldsymbol{\Sigma}_K)$

Figure 3: The data generative model in Gaussian Mixuture

Clearly, We have $K - 1$ parameters pertaining to π and d parameters pertaining to each of the K means of the Gaussian generators and d^2 for the variances. Hence in total $(d + d^2)K + K - 1$ or $(d^2 + d + 1)K - 1$ numbers of parameters.

1.2.2 The likelihood and log-likelihood estimations

The likelihood estimation function for a Gaussian Mixture Model (GMM) goes as follows:

$$\begin{aligned} L(X; \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \prod_{i=1}^{N} \mathbb{P}(X_i | \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) \\ &= \prod_{i=1}^{N} \left[\sum_{k=1}^{K} \pi_k \mathcal{N}(X_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right] \\ &= \prod_{i=1}^{N} \left[\sum_{k=1}^{K} \pi_k \frac{1}{(2\pi)^{\frac{d}{2}} |\boldsymbol{\Sigma}_k|^{\frac{1}{2}}} \exp\left(-\frac{1}{2} (X_i - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (X_i - \boldsymbol{\mu}_k) \right) \right] \end{aligned} \tag{12}$$

where $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ refer to the mean vector and covariance matrix of the k-th Gaussian component, respectively.

The log-likelihood estimation function for a GMM is given by:

$$\begin{aligned} \text{LogL}(X; \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \log \prod_{i=1}^{N} \left[\sum_{k=1}^{K} \pi_k \mathcal{N}(X_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right] \\ &= \sum_{i=1}^{N} \log \left[\sum_{k=1}^{K} \pi_k \mathcal{N}(X_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right] \\ &= \sum_{i=1}^{N} \log \left[\sum_{k=1}^{K} \pi_k \frac{1}{(2\pi)^{\frac{d}{2}} |\boldsymbol{\Sigma}_k|^{\frac{1}{2}}} \exp\left(-\frac{1}{2} (X_i - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (X_i - \boldsymbol{\mu}_k) \right) \right] \end{aligned} \tag{13}$$

1.2.3 Introducing latent variables λs

For ith data point introduce a set of K new latent variables, that is — $\{\lambda_1^i, \lambda_2^i, \lambda_3^i, \dots, \lambda_K^i\}$ such that $\sum_{k=1}^{K} \lambda_k^i = 1$ and for each $1 \leqslant i \leqslant N$ and for each $1 \leqslant k \leqslant K$, it holds that $0 \leqslant \lambda_k^i \leqslant 1$.

Note that:

$$\text{LogL}(X; \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{i=1}^{N} \log \left[\sum_{k=1}^{K} \lambda_k^i \left(\frac{\pi_k \prod_{j=1}^{d} \frac{1}{(2\pi)^{\frac{d}{2}} |\boldsymbol{\Sigma}_k|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(X_i - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (X_i - \boldsymbol{\mu}_k)\right)}{\lambda_k^i} \right) \right]$$

By Jensen's inequality

$$\geqslant \sum_{i=1}^{N} \sum_{k=1}^{K} \lambda_k^i \log \left(\frac{\pi_k \prod_{j=1}^{d} \frac{1}{(2\pi)^{\frac{d}{2}} |\boldsymbol{\Sigma}_k|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(X_i - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (X_i - \boldsymbol{\mu}_k)\right)}{\lambda_k^i} \right)$$

Let $\text{Modified-LogL}(X; \boldsymbol{\lambda}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) := \sum_{i=1}^{N} \sum_{k=1}^{K} \lambda_k^i \log \left(\frac{\pi_k \prod_{j=1}^{d} \frac{1}{(2\pi)^{\frac{d}{2}} |\boldsymbol{\Sigma}_{\mathbf{k}}|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(X_i - \boldsymbol{\mu}_{\mathbf{k}})^T {\boldsymbol{\Sigma}_{\mathbf{k}}}^{-1} (X_i - \boldsymbol{\mu}_{\mathbf{k}})\right)}{\lambda_k^i} \right).$

Note that $\text{Modified-LogL}(X; \boldsymbol{\lambda}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ provides a lower bound for $\text{LogL}(X; \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ for any choice of valid $\boldsymbol{\lambda}$.

1.2.4 Maximizing Modified-LogL over $(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ by fixing $\boldsymbol{\lambda}$

Firstly, the objective is to find $\hat{\boldsymbol{\pi}}$ that is the solution to

$$\begin{aligned} \max_{\boldsymbol{\pi}} \quad & \text{Modified-LogL}(X; \boldsymbol{\lambda}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) \\ \text{such that} \quad & \sum_{k=1}^{K} \pi_k = 1 \\ & 0 \leqslant \pi_k \leqslant 1 \quad \text{for each} \quad 1 \leqslant k \leqslant K \end{aligned}$$

Using the Lagrangian method, we obtain:

$$\hat{\pi}_k = \frac{\sum_{i=1}^{N} \lambda_k^i}{N} \tag{14}$$

Analogously, we want to find $\hat{\boldsymbol{\mu}}$ that is the solution to

$$\begin{aligned} \max_{\boldsymbol{\mu}} \quad & \text{Modified-LogL}(X; \boldsymbol{\lambda}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) \\ \text{such that} \quad & 0 \leqslant \mu_k \leqslant 1 \quad \text{for each} \quad 1 \leqslant k \leqslant K \end{aligned}$$

Taking the partial derivative wrt $\boldsymbol{\mu}$, we obtain:

$$\boxed{\hat{\mu}_k = \frac{\sum_{i=1}^{N} \lambda_k^i X_i}{\sum_{i=1}^{N} \lambda_k^i}} \tag{15}$$

Likewise, we want to find $\hat{\boldsymbol{\Sigma}}$ that is the solution to

$$\begin{aligned} \max_{\boldsymbol{\Sigma}} \quad & \text{Modified-LogL}(X; \boldsymbol{\lambda}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) \\ \text{such that} \quad & 0 \leqslant \mu_k \leqslant 1 \quad \text{for each} \quad 1 \leqslant k \leqslant K \end{aligned}$$

Taking the partial derivative wrt $\boldsymbol{\Sigma}$, we obtain:

$$\boxed{\hat{\Sigma}_k = \frac{\sum_{i=1}^{N} \lambda_k^i (X_i - \hat{\mu}_k)(X_i - \hat{\mu}_k)^T}{\sum_{i=1}^{N} \lambda_k^i}} \tag{16}$$

1.2.5 Maximizing Modified-LogL over $\boldsymbol{\lambda}$ by fixing $(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$

The objective is to find $\hat{\boldsymbol{\lambda}}$ that is the solution to

$$\begin{aligned} \max_{\boldsymbol{\lambda}} \quad & \text{Modified-LogL}(X; \boldsymbol{\lambda}, \boldsymbol{\pi}, \boldsymbol{\mu}) \\ \text{such that} \quad & \sum_{k=1}^{K} \lambda_k^i = 1 \quad \text{for each} \quad 1 \leqslant i \leqslant N \\ & 0 \leqslant \lambda_k^i \leqslant 1 \quad \text{for each} \quad 1 \leqslant k \leqslant K \quad \text{and} \quad 1 \leqslant i \leqslant N \end{aligned}$$

the solution to which goes as follows:

$$\hat{\lambda}_k^i = \frac{\pi_k \mathbb{P}(X_i|\boldsymbol{\mu}_\mathbf{k})}{\sum\limits_{j=1}^{K} \pi_j \mathbb{P}(X_i|\boldsymbol{\mu}_\mathbf{j})} = \frac{\pi_k \frac{1}{(2\pi)^{\frac{d}{2}}|\boldsymbol{\Sigma}_k|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(X_i - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1}(X_i - \boldsymbol{\mu}_k)\right)}{\sum\limits_{j=1}^{K} \pi_j \frac{1}{(2\pi)^{\frac{d}{2}}|\boldsymbol{\Sigma}_j|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(X_i - \boldsymbol{\mu}_j)^T \boldsymbol{\Sigma}_j^{-1}(X_i - \boldsymbol{\mu}_j)\right)} \tag{17}$$

1.2.6 The EM algorithm

The Expectation Maximization (EM) algorithm for a Gaussian Mixture Model (GMM) is as follows:

Algorithm 2 : Expectation Maximization (Gaussian Mixture Model)

```
Input: Dataset: $\{X_i\}_{i=1}^N$, Number of Components: $K$, Tolerance: $\epsilon$
Initialization: $\boldsymbol{\mu}^0 = \{\boldsymbol{\mu}_1^0, \boldsymbol{\mu}_2^0, \dots, \boldsymbol{\mu}_K^0\}$ and $\boldsymbol{\Sigma}^0 = \{\boldsymbol{\Sigma}_1^0, \boldsymbol{\Sigma}_2^0, \dots, \boldsymbol{\Sigma}_K^0\}$
while $\max(\|\boldsymbol{\mu}^{t+1} - \boldsymbol{\mu}^t\|, \|\boldsymbol{\Sigma}^{t+1} - \boldsymbol{\Sigma}^t\|) \geqslant \epsilon$ do
    $\boldsymbol{\lambda}^{t+1} \leftarrow \arg\max_{\lambda} \text{Modified-LogL}(X; \boldsymbol{\pi}^t, \boldsymbol{\mu}^t, \boldsymbol{\Sigma}^t, \boldsymbol{\lambda})$        ▷ Expectation Step
    $\boldsymbol{\pi}^{t+1} \leftarrow \arg\max_{\pi} \text{Modified-LogL}(X; \boldsymbol{\pi}, \boldsymbol{\mu}^t, \boldsymbol{\Sigma}^t, \boldsymbol{\lambda}^t)$        ▷ Maximization Step
    $\boldsymbol{\mu}^{t+1} \leftarrow \arg\max_{\mu} \text{Modified-LogL}(X; \boldsymbol{\pi}^t, \boldsymbol{\mu}, \boldsymbol{\Sigma}^t, \boldsymbol{\lambda}^t)$        ▷ Maximization Step
    $\boldsymbol{\Sigma}^{t+1} \leftarrow \arg\max_{\Sigma} \text{Modified-LogL}(X; \boldsymbol{\pi}^t, \boldsymbol{\mu}^{t+1}, \boldsymbol{\Sigma}, \boldsymbol{\lambda}^t)$        ▷ Maximization Step
end while
```

1.2.7 Implementing the EM algorithm

The following code implements the EM algorithm for the Gaussian Mixture Model.

```python
class GaussianMixtureEM:
    def __init__(self, num_components=4, max_iteration=50, tolerance=1e-10, num_random_inits=100):
        self.num_components = num_components
        self.max_iteration = max_iteration
        self.tolerance = tolerance
        self.num_random_inits = num_random_inits
        self.means = None
        self.covariances = None
        self.weights = None
        self.log_likelihoods = None
        self.responsibilities = None

    def _e_step(self, data):
        num_points, num_features = data.shape
        log_responsibilities = np.zeros((num_points, self.num_components))
        epsilon = np.finfo(float).eps

        for k in range(self.num_components):
            for i in range(num_points):
                diff = data[i] - self.means[:, k]
                exponent = np.dot(diff.T, np.dot(np.linalg.inv(self.covariances[k]), diff))
                log_responsibilities[i, k] = np.log(self.weights[k]) - 0.5 * exponent

        self.responsibilities = np.maximum(np.exp(log_responsibilities), epsilon)

        sum_responsibilities_k = np.sum(self.responsibilities, axis=1, keepdims=True)
        for i in range(num_points):
            for k in range(self.num_components):
                self.responsibilities[i, k] /= sum_responsibilities_k[i, 0]

        return self.responsibilities

    def _m_step(self, data):
        num_points, num_features = data.shape
        prev_means = copy.deepcopy(self.means)

        self.means = data.T @ self.responsibilities
        sum_responsibilities_N = np.maximum(np.sum(self.responsibilities, axis=0,
         ↪  keepdims=True), epsilon)

        for k in range(self.num_components):
            self.covariances[k] *=0
            for i in range(num_points):
                diff = data[i] - self.means[:, k]
                self.covariances[k] += self.responsibilities[i, k]* np.outer(diff, diff.T)

        for k in range(self.num_components):
            if sum_responsibilities_N[0, k] != 0:
                self.covariances[k] /= sum_responsibilities_N[0, k]
            for i in range(num_features):
                if sum_responsibilities_N[0, k] != 0:
                    self.means[i, k] /= sum_responsibilities_N[0, k]
                    self.covariances[k] /= sum_responsibilities_N[0, k]

        self.means = np.clip(self.means, 0, 1)

        self.weights = np.mean(self.responsibilities, axis=0)
        return prev_means

    def cluster_assignment(self):
        return self.responsibilities

    def _log_likelihood(self, data):
        log_likelihood = 0
        num_points, num_features = data.shape
        log_probs = np.zeros((num_points, self.num_components))

        for k in range(self.num_components):
            for i in range(num_points):
                diff = data[i] - self.means[:, k]
                exponent = np.dot(diff, np.dot(np.linalg.inv(self.covariances[k]), diff.T))
                log_probs[i, k] = np.log(self.weights[k]) - 0.5 * exponent

        log_probs = np.clip(np.exp(log_probs), epsilon, 1)
        log_likelihood = np.sum(np.log(np.sum(log_probs, axis=1)))

        return log_likelihood

    def fit(self, data):
        num_points, num_features = data.shape
        log_likelihoods = np.zeros((self.num_random_inits, self.max_iteration))

        for random_initialization in tqdm(range(self.num_random_inits), desc="Random
         ↪  Initialization", leave=False):
            np.random.seed(random_initialization)
            self.means = np.random.rand(num_features, self.num_components)
            self.covariances = [np.eye(num_features) for _ in range(self.num_components)]

            self.weights = np.ones(self.num_components)
            self.weights /= np.sum(self.weights)
            log_likelihoods_per_init = []

            for itr in range(self.max_iteration):
                # E-step
                self.responsibilities = self._e_step(data)

                # M-step
                prev_means = self._m_step(data)

                # Log-likelihood
                log_likelihood_val = self._log_likelihood(data)
                log_likelihoods_per_init.append(log_likelihood_val)

                # Check for convergence
                if itr > 0 and np.max(np.abs(self.means - prev_means)) < self.tolerance:
                    break

            log_likelihoods_per_init = np.array(log_likelihoods_per_init)

            if len(log_likelihoods_per_init) < self.max_iteration:
                last_log_likelihood = log_likelihoods_per_init[-1]
                log_likelihoods_per_init = np.pad(log_likelihoods_per_init, (0,
                    ↪  self.max_iteration - len(log_likelihoods_per_init)),
                    ↪  constant_values=last_log_likelihood)

            log_likelihoods[random_initialization] = log_likelihoods_per_init

        self.log_likelihoods = np.mean(log_likelihoods, axis=0)

    def plot_log_likelihood(self, filename=None):
        plt.plot(np.arange(1, len(self.log_likelihoods) + 1), self.log_likelihoods,
         ↪  color='blue')
        plt.xlabel('Iterations')
        plt.ylabel('Log-Likelihood')
        plt.title('Log-Likelihood vs. Iterations (GMM)')
        if filename:
            plt.savefig(filename, format='pdf', bbox_inches='tight')
        else:
            plt.show()
```

1.2.8 Parameter Values

The values of the parameters used are as follows.

Parameters	Values
# of Components	4
maximum # of iterations	50
Tolerance	1e-10
epsilon	1e-20
# of random initialization	100

Table 2: Parameters used for Gaussian Mixture Model

1.2.9 Results

Note the following details regarding the dataset:

```
Mean vector shape:  (50,)
Covariance matrix shape:  (50, 50)
Covariance matrix determinant:  2.7543550134960894e-53
```

Thus, it turns out the Covariance matrix has a significantly low determinant, thus it is essentially singular. Thus, while learning the GMM, after the initial iteration, the responsibilities value λ becomes negligible small before normalization. Thus, we assume that at any state $|\mathbf{\Sigma_k^t}|$ is constant for each k. Also, whenever, the numerator of the responsibilities value λ becomes significantly small, we prune it by considering its value to be at least epsilon.

Consequently, GMM didn't learn the value of the weights, means and covariances, and it assigned equal probabilities to each of the mixtures, that is the weights after fitting the dataset are as follows:

```
Weights:  [0.25 0.25 0.25 0.25]
```

Note that at the end of the training, the **value of log-likelihood obtained is -17866.162999504428** The plot of the log-likelihood (averaged over 100 random initialization) against the iterations goes as follows (This is dependent on the value of the epsilon that was chosen to be 1e-20):

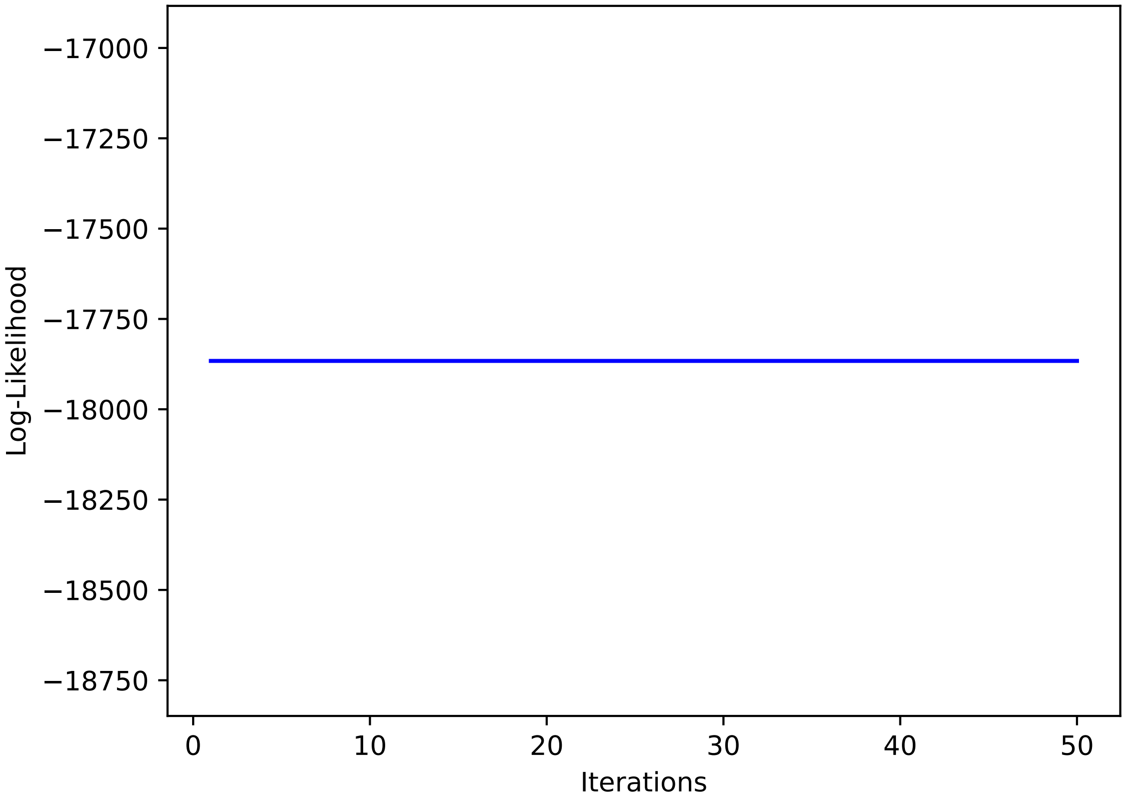


Figure 4: Log Likelihood vs Iterations in Gaussian Mixture Model

1.2.10 Observations and Inference

Clearly, the Bernoulli Mixture Model performed significantly better as compared to the Gaussian Mixture Model. Here are some potential reasons why the GMM might perform poorly compared to a Bernoulli Mixture Model:

1. **Assumption Violation:** GMM assumes that data is generated from a mixture of Gaussian distributions, which might not be appropriate for binary data. Binary data inherently have different characteristics from continuous data, violating the underlying assumptions of the Gaussian distribution.

2. **Covariance Matrix:** The determinant of the covariance matrix being significantly low indicates that the features might be highly correlated or linearly dependent. In a GMM, the covariance matrix plays a crucial role in defining the shape and orientation of each Gaussian component. Low determinant suggests that the covariance matrix might be close to singular, leading to challenges in accurately modelling the data distribution.

3. **Model Complexity:** GMM is a more complex model compared to a Bernoulli Mixture Model, especially when applied to binary data. The additional parameters in GMM might lead to overfitting, especially if the data is not well-represented by Gaussian distributions.

4. **Initialization Sensitivity:** GMM initialization is sensitive, and poor initialization can lead to convergence to suboptimal solutions or even failure to converge. Initialization methods such as K-means or random initialization might not work well for binary data, leading to poor performance.

5. **Scalability:** GMM can become computationally expensive, especially with a large number of Gaussian components or high-dimensional data. If the dataset is large or high-dimensional, GMM might struggle to converge or generalize well.

1.3 K-means Clustering

The objective function of K-means clustering is to minimize the sum of squared distances between data points and their respective cluster centroids. Mathematically, the objective function J of K-means clustering can be represented as:

$$J = \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \tag{18}$$

where

- J is the objective function,
- K is the number of clusters,
- C_i represents the ith cluster,
- μ_i is the centroid (mean) of cluster C_i, and
- $|| \cdot ||$ denotes the Euclidean distance.

The objective is to find the cluster centroids $\mu_1, \mu_2, \dots, \mu_k$ that minimize this total within-cluster variance.

During the optimization process of K-means, the algorithm iteratively assigns data points to the nearest cluster centroid and updates the centroids to minimize this objective function. The algorithm continues iterating until convergence, typically when the cluster assignments or centroids no longer change significantly between iterations.

Therefore, the objective value of K-means clustering is the final value of the objective function after convergence, representing the total within-cluster variance achieved by the clustering. The reader may refer to Assignment 1 for the theory concerning K-means Clustering. Here we shall directly mention the implementation, results and observations.

1.3.1 Implementation

The following code implements the K-means clustering algorithm:

```
class KMeans:
    def __init__(self, k=4, max_iteration=50, tolerance=1e-10, k_means_plusplus=False):
        self.k = k
        self.max_iteration = max_iteration
        self.tolerance = tolerance
        self.k_means_plusplus = k_means_plusplus
        self.centroids = None
        self.labels = None
        self.n_samples = None
        self.n_features = None
        self.objective_value = []

    def fit(self, X):
        self.n_samples, self.n_features = X.shape

        if self.k_means_plusplus:
            self.centroids = self.k_means_plusplus_init(X)
        else:
            self.centroids = X[np.random.choice(self.n_samples, self.k, replace=False)]

        for iteration in tqdm(range(self.max_iteration), desc='Iteration', leave=False):
            distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
            self.labels = np.argmin(distances, axis=1)

            new_centroids = np.array([X[self.labels == j].mean(axis=0) for j in range(self.k)])

            if np.linalg.norm(new_centroids - self.centroids) < self.tolerance:
                break

            self.centroids = new_centroids

            # Calculate objective value and record
            objective_value = self.calculate_objective_value(X)
            self.objective_value.append(objective_value)

        self.labels = np.argmin(distances, axis=1)
        return self.labels, self.centroids

    def plot_objective(self, filename=None):
        plt.plot(np.arange(1, len(self.objective_value) + 1), self.objective_value,
         ↪  color='blue')
        plt.xlabel('Iterations')
        plt.ylabel('Objective Value')
        if self.k_means_plusplus:
            plt.title('Objective Value vs. Iterations (K-means++)')
        else:
            plt.title('Objective Value vs. Iterations (K-means++)')
        if filename:
            plt.savefig(filename, format='pdf', bbox_inches='tight')
        else:
            plt.show()

    def k_means_plusplus_init(self, X):
        self.centroids = np.zeros((self.k, self.n_features), dtype=X.dtype)

        self.centroids[0] = X[np.random.choice(self.n_samples)]

        for i in range(1, self.k):
            min_distances = np.full((self.n_samples,), np.inf)

            for j in range(i):
                distances = np.linalg.norm(X - self.centroids[j], axis=1)
                min_distances = np.minimum(min_distances, distances)

            if np.all(min_distances == 0):
                self.centroids[i] = X[np.random.choice(self.n_samples)]
            else:
                min_distances[np.isnan(min_distances)] = 0

                if np.any(np.isnan(min_distances)) or np.any(min_distances == 0):
                    self.centroids[i] = X[np.random.choice(self.n_samples)]
                else:
                    probabilities = min_distances**2 / np.sum(min_distances**2)
                    probabilities[np.isnan(probabilities)] = 0
                    self.centroids[i] = X[np.random.choice(self.n_samples, p=probabilities)]

        return self.centroids

    def calculate_objective_value(self, X):
        distances = np.linalg.norm(X - self.centroids[self.labels], axis=1)
        return np.sum(distances**2)

    def log_likelihoods(self, X):
        n_samples = X.shape[0]
        distances = np.linalg.norm(X - self.centroids[self.labels], axis=1)
        log_likelihood = -np.sum(distances**2)
        return log_likelihood

    def cluster_assignment(self, X):
        distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
        min_distances = np.min(distances, axis=1, keepdims=True)
        responsibilities = np.exp(-distances**2 / min_distances**2)
        responsibilities /= np.sum(responsibilities, axis=1, keepdims=True)
        return responsibilities
```

1.3.2 Parameter Values

Followingly we mention the values of the parameters used:

Parameters	Values
Algorithm	K-means++
# of Components	4
maximum # of iterations	50
Tolerance	1e-10
epsilon	1e-20
# of random initialization	100

Table 3: Parameters used for K means Clustering

1.3.3 Results

Note that at the end of the iteration, the **value of the objective function of K-means clustering obtained is 1746.187816915332.** The plot of the objective function vs iterations goes as follows:

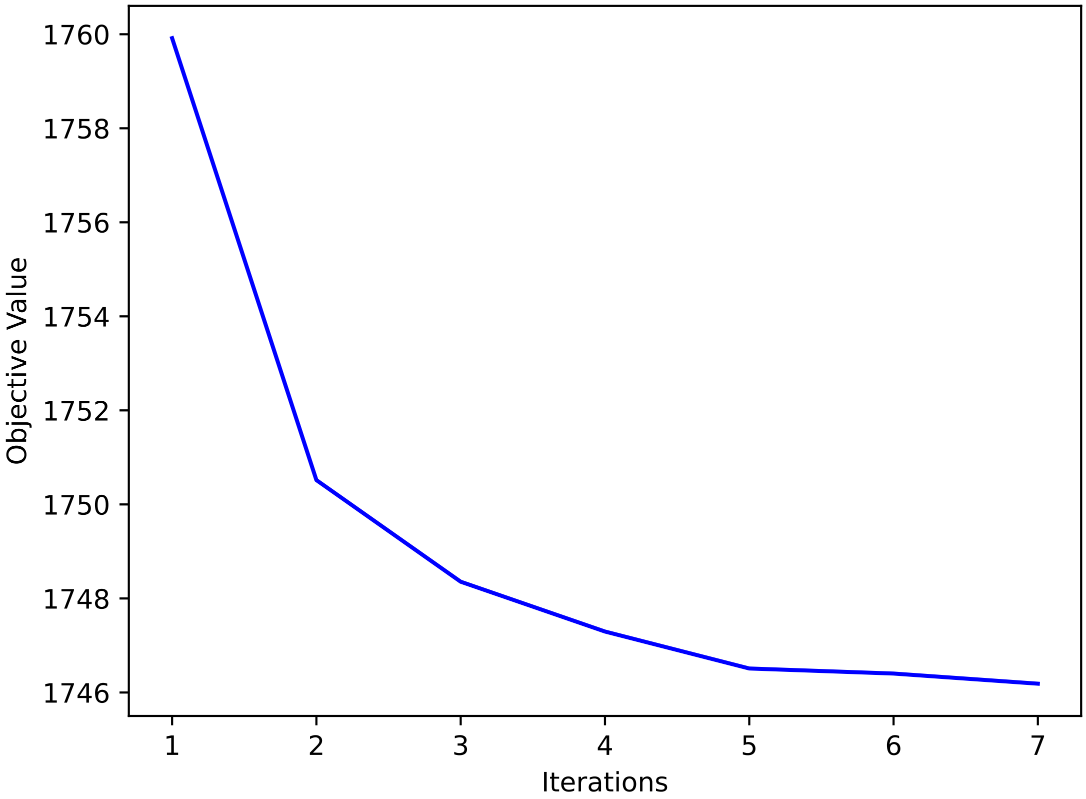


Figure 5: Objective Value vs Iterations in K-means++

1.4 Comparison amongst different models

Recall that earlier we established why the Bernoulli Mixture Model performed better as compared to the Gaussian Mixture Model. Please refer to Section 1.1.9, 1.2.9 and 1.2.10 for the details.

Here we shall compare the Bernoulli Mixture Model with the K-means clustering:

1. **Model Assumptions:** BMM explicitly models binary data by assuming that each feature follows a Bernoulli distribution. This makes it well-suited for binary data as it directly captures the probability of each feature being 1 or 0 within each cluster. On other other hand, K-means is computationally efficient and easy to implement. It assigns each data point to the nearest centroid, making it suitable for large datasets.

2. **Cluster Shape:** BMM allows for flexible cluster shapes since it models the probability of each feature independently. This means clusters can have arbitrary shapes in the binary feature space. On contrast, K-means clusters data points based on their distances to cluster centroids, which might not capture the underlying probability distribution of binary features as explicitly as BMM.

3. **Cluster Interpretability:** BMM provides probabilities for each feature being 1 or 0 within each cluster, allowing for easy interpretation of cluster characteristics, whereas K-means assumes isotropic clusters (spherical clusters with equal variance in all dimensions), which might not be suitable for binary data with complex cluster shapes.

4. **Parameter Estimation:** BMM involves estimating parameters such as cluster probabilities and feature probabilities, which might require more computational resources compared to

k-means. As far as K-means is concerned, it is sensitive to initialization, and different initializations can lead to different clustering results.

In the given context, the interpretability and capturing the underlying probability distribution of binary features are essential, and we have computational resources, so **BMM is preferable.**

2 Regression

You are given a data-set in the file A2Q2Data train.csv with 10000 points in $(\mathbb{R}^{100}, \mathbb{R})$ (Each row corresponds to a datapoint where the first 100 components are features and the last component is the associated y value).

1. Obtain the least squares solution $\mathbf{w}_{ML}$ to the regression problem using the analytical solution.
2. Code the gradient descent algorithm with suitable step size to solve the least squares algorithms and plot $||\mathbf{w}^t - \mathbf{w}_{ML}||_2$ as a function of t. What do you observe?
3. Code the stochastic gradient descent algorithm using a batch size of 100 and plot $||\mathbf{w}^t - \mathbf{w}_{ML}||_2$ as a function of t. What are your observations?
4. Code the gradient descent algorithm for ridge regression. Cross-validate for various choices of λ and plot the error in the validation set as a function of λ. For the best λ chosen, obtain $\mathbf{w}_R$. Compare the test error (for the test data in the file A2Q2Data test.csv) of $\mathbf{w}_R$ with $\mathbf{w}_{ML}$. Which is better and why?

This section discusses several kinds of regression.

2.1 Analytical Solution to the Least Square

Let's discuss how to obtain the analytical solution for the fit of a given dataset for minimizing the squared error term.

2.1.1 Introduction

To obtain the maximum likelihood solution $\mathbf{w}_{\text{ML}}$ to the regression problem using the analytical solution without the bias term, we first define our linear regression model. Let $\mathbf{X}$ be the design matrix of size $N \times d$, where N is the number of data points and d is the number of features. Each column of $\mathbf{X}$ represents a data point. The target vector $\mathbf{y}$ has size $N \times 1$, containing the observed target values.

The linear regression model is given by:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \epsilon \tag{19}$$

where $\mathbf{w}$ is the parameter vector of size $d \times 1$ to be estimated, and ϵ represents the error term.

The maximum likelihood solution $\mathbf{w}_{\text{ML}}$ is obtained by maximizing the likelihood function of the observed data under the assumption of Gaussian noise. This is equivalent to minimizing the negative log-likelihood. Mathematically, we seek:

$$\mathbf{w}_{\text{ML}} = \arg\min_{\mathbf{w}} ||\mathbf{y} - \mathbf{X}\mathbf{w}||_2^2 \tag{20}$$

2.1.2 Analytical Solution using Maximum Likelihood estimation

To find the analytical solution, we differentiate the negative log-likelihood with respect to $\mathbf{w}$, set the derivative equal to zero, and solve for $\mathbf{w}$. The analytical solution is given by:

$$\mathbf{w}_{\text{ML}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

where $(\mathbf{X}^T\mathbf{X})^{-1}$ is the inverse of the matrix $\mathbf{X}^T\mathbf{X}$.

This analytical solution provides the parameter vector $\mathbf{w}_{\text{ML}}$ that maximizes the likelihood of observing the given data, under the assumption of Gaussian noise. It represents the best-fitting linear model for the given data.

For the given training dataset:

1. Shape of Feature matrix: (10000, 100)
2. Shape of Value vector: (10000,)
3. The $\mathbf{w_{ML}}$ (Analytical) goes as follows:

```
[-7.84961009e-03 -1.36715320e-02 -3.61656438e-03  2.64909160e-03
  1.88551446e-01  2.65314657e-03  9.46531786e-03  1.79809481e-01
  3.73757317e-03  4.99608944e-01  8.35836265e-03  4.29108775e-03
  1.42141179e-02  3.94232414e-03  9.36795890e-03 -1.12038274e-03
  3.35727500e-03  1.16152212e-03 -9.40884707e-03 -2.45575476e-03
 -1.17409629e-02 -1.01960612e-02  7.95771321e-03 -1.00574854e-02
  6.04882939e-03 -4.67345192e-03 -3.09091547e-03  8.14909193e-03
  1.20264599e-02 -6.82458163e-03 -8.65405539e-03  9.86273479e-04
  4.92968011e-03  5.99772461e-03 -1.34667860e-02  1.07075729e-03
  1.32745992e-02 -1.14148742e-02 -2.01056697e-02  5.85096240e-01
  4.94483247e-04 -7.86666920e-04 -2.71926574e-03 -9.54021938e-03
 -5.44161058e-03  9.80679209e-03 -6.72540624e-03 -4.45414276e-04
  6.98516508e-03  3.16138907e-02  4.51763485e-01 -8.75221380e-03
  2.55167390e-03  4.24921150e-03  2.89847927e-01  7.03723255e-03
 -1.95796946e-03  1.41523883e-02 -1.06508170e-02  7.72743903e-01
 -5.67126044e-03 -6.30026188e-04  6.50943015e-03 -4.84019165e-03
  4.63832329e-03  4.54887177e-03 -2.99475114e-03  8.38781696e-03
 -2.47558716e-03  9.00947922e-04  1.14713514e-03 -1.87641345e-03
 -1.05175760e-02 -9.31304110e-03 -1.23550002e-03  5.97797559e-01
 -4.78625013e-03 -1.13727852e-02  2.88477060e-03  8.48999776e-01
 -1.08924235e-02  2.26346489e-03 -1.38099800e-03 -6.35934691e-03
  5.83784109e-03  5.69286755e-03  5.35566859e-03 -8.20616315e-03
  1.29884015e-02 -2.30575631e-03 -1.22263765e-04  8.66629171e-03
 -4.29446300e-03  5.69510898e-03  7.55483353e-03 -9.43540843e-03
  1.82905446e-02 -1.16998887e-03 -2.61599136e-03 -8.58616114e-03]
```

2.2 Gradient Descent

You know! Matrix multiplications are expensive. Let's try to approximate the Least square solution using the gradient descent optimization method.

2.2.1 Theory

To solve the least squares problem using the gradient descent algorithm with a suitable step size, we first define the objective function as the mean squared error (MSE) between the observed target values $\mathbf{y}$ and the predictions made by the linear model:

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y}_i - \mathbf{X}_i\mathbf{w})^2$$

where $\mathbf{X}_i$ is the feature vector for the i-th data point, $\mathbf{y}_i$ is the observed target value for the i-th data point, $\mathbf{w}$ is the parameter vector, and N is the number of data points.

The gradient of the objective function $J(\mathbf{w})$ with respect to the parameter vector $\mathbf{w}$ is given by:

$$\nabla J(\mathbf{w}) = -\frac{2}{N}\sum_{i=1}^{N}\mathbf{X}_i^T(\mathbf{y}_i - \mathbf{X_i}\mathbf{w}) \tag{21}$$

The gradient descent algorithm iteratively updates the parameter vector $\mathbf{w}$ using the following update rule:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha\nabla J(\mathbf{w}^{(t)}) \tag{22}$$

where α is the step size or learning rate.

The choice of step size α is crucial for the convergence of the algorithm. If α is too small, the convergence may be slow, while if α is too large, the algorithm may overshoot the minimum and fail to converge. A suitable step size can be determined using techniques such as line search or fixed step size selection based on heuristics or theoretical analysis.

The algorithm terminates when a stopping criterion is met, such as reaching a maximum number of iterations or achieving a small change in the objective function between iterations.

Algorithm 3 : Gradient Descent for Least Squares

```
Input: Training data: $\mathbf{X}$, target values: $\mathbf{y}$, initial parameter vector: $\mathbf{w}^{(0)}$, learning rate: $\alpha$, maximum number of iterations: $T$, tolerance: $\epsilon$
Output: Parameter vector: $\mathbf{w}$
for $t = 1, 2, 3, \ldots, T$ do
    Compute the gradient: $\nabla J(\mathbf{w}^{(t)}) = -\frac{1}{N}\sum_{i=1}^{N}\mathbf{X}_i^T(\mathbf{y}_i - \mathbf{X}_i\mathbf{w}^{(t)})$
    Update the parameter vector: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha\nabla J(\mathbf{w}^{(t)})$
    if $||\nabla J(\mathbf{w}^{(t)})|| < \epsilon$ then
        break
    end if
end for
```

2.2.2 Implementation of Gradient Descent

Followingly, we present the code for the gradient descent written for the implementation.

```
def analytical_solution(self):
    if self.regression == 'Least Square':
        return np.linalg.inv(self.X.T @ self.X) @ self.X.T @ self.y
    elif self.regression == 'Ridge':
        I = np.eye(self.X.shape[1])
        return np.linalg.inv(self.X.T @ self.X + self.lambda_ridge * I) @ self.X.T @ self.y

def compute_gradient(self, w, batch_X, batch_y):
    if self.regression == 'Least Square':
        return -2 * batch_X.T @ (batch_y - batch_X @ w)
    elif self.regression == 'Ridge':
        I = np.eye(self.X.shape[1])
        return -2 * batch_X.T @ (batch_y - batch_X @ w) + 2 * self.lambda_ridge * I @ w

def compute_loss(self, w):
    return np.linalg.norm(self.X @ w - self.y, ord=2) ** 2 / self.X.shape[0]

def update_weights(self, w, gradient):
    return w - self.learning_rate * gradient

def train(self, w_init, verbose=True):
    w = w_init
    w_analytical = self.analytical_solution()
    norm_diff = []
    loss = []
    indices = np.arange(self.X.shape[0])
    start_time = time.process_time()
    for epoch in range(self.max_iterations):
        np.random.shuffle(indices)
        shuf_X = self.X[indices]
        shuf_y = self.y[indices]
        batch_X = shuf_X[:self.batch_size]
        batch_y = shuf_y[:self.batch_size]
        gradient = self.compute_gradient(w, batch_X, batch_y)
        w = self.update_weights(w, gradient)
        norm_diff.append(np.linalg.norm(w - w_analytical, ord=2))

        epoch_loss = self.compute_loss(w)
        loss.append(epoch_loss)
        if verbose and (epoch<10 or (epoch+1)%100 == 0):
            print('Epoch {}, Learning Rate: {}, Loss: {}, Norm Difference: {}'.format(epoch+1,
             ↪  self.learning_rate, epoch_loss, norm_diff[-1]))
        if epoch_loss < self.threshold_relative_error:
            break
```

2.2.3 Tuning the Learning Rate

For the given dataset the Learning Rate was tuned on several values; the following table summarizes the MSE loss and the $||\mathbf{w}-\mathbf{w}_{ML}||$ obtained on each of the final $\mathbf{w}$ value obtained. Note the following parameter values that are set:

- The maximum # of iteration: 2000,
- $\mathbf{w}$ is initialized as a null vector in $\mathbb{R}^d$;
- the threshold relative error: $\mathbf{10}^{-10}$, and

- Clearly the batch size is the entire batch.

Followingly, we mention the method of hyperparameter tuning that involves systematically searching through a predefined set of hyperparameters to find the combination that yields the best performance for the given dataset.

2.2.3.1 Initialization

The `HyperparameterTuning` class is initialized with the following parameters:

- `model`: The machine learning model for which hyperparameters need to be tuned.
- `parameter_grid`: A dictionary specifying the hyperparameters to be tuned and their respective values or ranges.
- `cross_validation`: The number of folds to be used in cross-validation for evaluating model performance.
- `verbose`: A flag indicating whether to print progress during the tuning process.

2.2.3.2 Grid Search

The hyperparameter tuning process typically employs a grid search approach, where all possible combinations of hyperparameters are explored to find the best combination. The `_generate_parameter_combinations` method generates all possible combinations of hyperparameters from the specified grid.

2.2.3.3 Cross-Validation

Cross-validation is employed to estimate the performance of each combination of hyperparameters. The dataset is divided into multiple folds (usually k folds), with one fold reserved for validation and the rest for training. For each combination of hyperparameters:

- The model is instantiated with the current set of hyperparameters.
- The model is trained on the training data.
- The trained model is evaluated on the validation data to compute a performance metric, such as the mean squared error (MSE).
- The average performance metric across all folds is calculated to provide a robust estimate of the model's performance with the given hyperparameters.

2.2.3.4 Optimization

The combination of hyperparameters that yields the lowest average performance metric (e.g., lowest MSE) across all cross-validation folds is considered optimal. These optimal hyperparameters are stored in the `optimum_parameters_` attribute along with the corresponding performance metric (e.g., loss) in `optimum_loss_`.

2.2.3.5 Progress Reporting

If verbosity is enabled, the tuning process may print progress information, such as the performance metric for each combination of hyperparameters.

Overall, by systematically exploring different combinations of hyperparameters and evaluating their performance using cross-validation, the hyperparameter tuning process aims to identify the set of hyperparameters that optimizes the model's performance on unseen data, thereby improving its generalization ability.

Note that the Learning rate was tuned based on the minimisation of the final value obtained for the MSE.

Table 4: Tuning Learning rate for batch mode of gradient descent in maximum likelihood estimation

Learning Rate	$\frac{1}{N}\|\|\mathbf{y}-\mathbf{X}\mathbf{w}\|\|_2^2$	$\|\|\mathbf{w}-\mathbf{w}_{ML}\|\|_2^2$
5e-08	0.1939506314703723	1.4627443028608227
1e-07	0.15848134390210558	1.3722980065796149
5e-07	0.055304508678857056	0.8706861317075958
1e-06	0.041757381889203936	0.5556253780390124
3e-06	0.04065692827650064	0.20097954639772542
3.5e-06	0.0406575563846478	0.17224777718459325
3.75e-06	0.04065769614442924	0.16073405073240016
4e-06	0.0406577775250922	0.15065635649158504
5e-06	593275867149937.2	158874.51394365443
1e-05	nan	nan
5e-05	nan	nan

Thus, the optimal Learning Rate obtained is **3e-06** with the MSE and Norm difference being 0.0406575563846478 and 0.17224777718459325 respectively. Followingly we plot the MSE Loss obtained as a function of Learning rate.

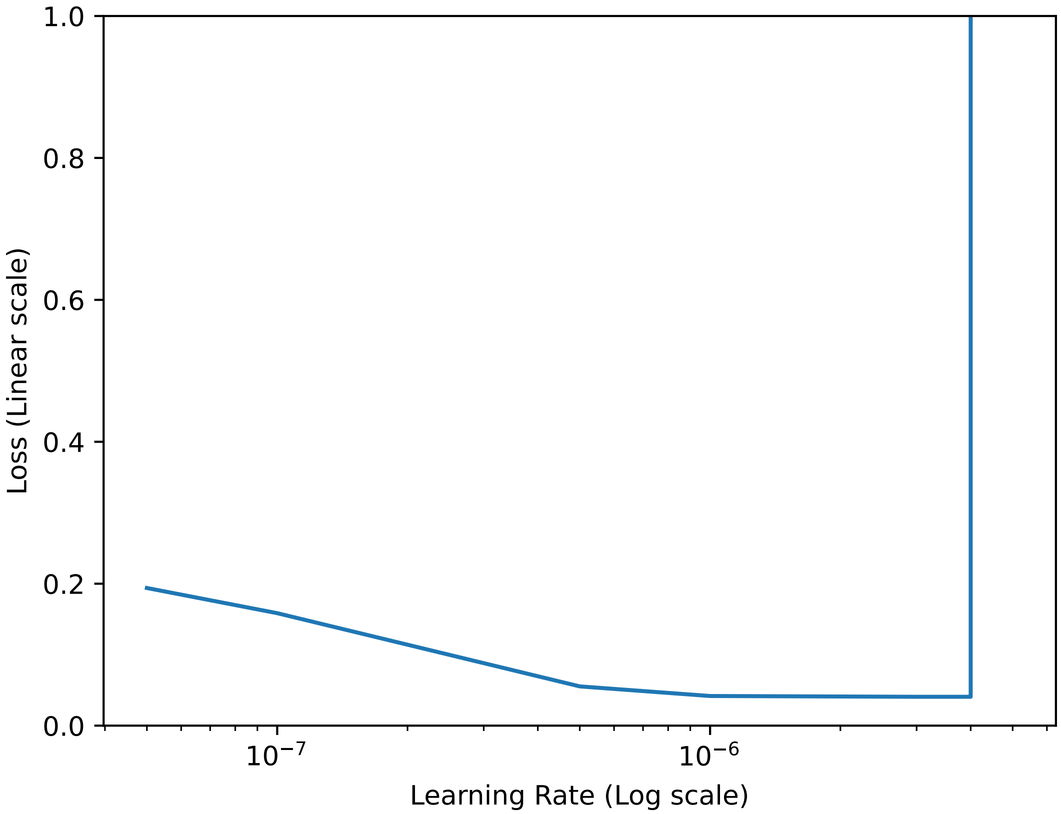


Figure 6: Loss vs. Learning Rate (Least Square Regression: Batch mode)

2.2.4 Parameter Values

The following table represents the final set of values chosen/ tuned for the batch mode of gradient descent for the maximum likelihood estimation.

Parameter	**Value**
regression	Least Square
Learning Rate	3e-6
Max # of iterations	2000
$\mathbf{w}$ initialization	$\mathbf{0} \in \mathbb{R}^d$
Threshold relative MSE	1e-10
Batch size	N

Table 5: Parameters chosen/ set/ tuned for the batch gradient descent for maximum likelihood extimation

2.2.5 Results

We obtain the following result through the batch gradient descent for the maximum likelihood estimation.

Table 6: Epoch vs MSE vs Norm Difference in Batch gradient Descent for maximum likelihood estimation

Epoch (t)	$\frac{1}{N}\|\|\mathbf{y} - \mathbf{X}\mathbf{w}^t\|\|_2^2$	$\|\|\mathbf{w}^t - \mathbf{w}_{ML}\|\|_2^2$
1	1.4942837892727647	1.5682563851542437
2	0.5569146758082525	1.5486730638626833
3	0.3161167557223516	1.537968968040264
4	0.2531954583346647	1.5295855679670678
5	0.23570748336929273	1.521826950361447
6	0.22983469399006903	1.5142569293167543
7	0.2269418434534549	1.5067633749530183
8	0.22482387016605399	1.4993173664296093
9	0.2229175935178053	1.4919113392241399
10	0.22107907122465376	1.4845432099779747
100	0.11375611803194272	0.952539108057926
200	0.06730294885358544	0.5842401006803338
300	0.05007454798564643	0.35989689416946796
400	0.04362778135945991	0.22262801408311056
500	0.04119431950419657	0.13826838772190617
600	0.04026795237694386	0.08620372746862207
700	0.03991241531881283	0.05393899501753523
800	0.039774891232515965	0.033866025961208704
900	0.03972129912167598	0.021331458528820547
1000	0.039700267189162114	0.013476691875044753
1100	0.0396919583920103	0.008538203503914174
1200	0.03968865543861861	0.005423600482543228
1300	0.03968733475136266	0.003453572498684692
1400	0.03968680378823259	0.0022041210931276066
1500	0.03968658923429581	0.0014096761563592596
1600	0.039686502124762685	0.0009033500890822857
1700	0.039686466601706744	0.0005799440726023618
1800	0.0396864520558939	0.00037295318240667134
1900	0.03968644607693594	0.00024022034819682898
2000	0.03968644361056439	0.000154954981636489

Also, followingly, the CPU time taken for training and the final **w** obtained is written down.

1. CPU time taken for training: 206.30499813000006 seconds
2. wML (Batch Mode of Gradient Descent):

```
[-7.84539365e-03 -1.36653828e-02 -3.63298110e-03  2.64365096e-03
  1.88524882e-01  2.65731677e-03  9.45359284e-03  1.79784909e-01
  3.75535448e-03  4.99583072e-01  8.35795835e-03  4.29792461e-03
  1.42193402e-02  3.94222067e-03  9.37473153e-03 -1.12267377e-03
  3.37735342e-03  1.17917801e-03 -9.39706620e-03 -2.43601722e-03
 -1.17391261e-02 -1.01873569e-02  7.95484727e-03 -1.00581940e-02
  6.05164368e-03 -4.66294950e-03 -3.09169489e-03  8.14556026e-03
  1.20210933e-02 -6.81929831e-03 -8.65427504e-03  9.74490205e-04
  4.95048449e-03  5.99456300e-03 -1.34575868e-02  1.06592184e-03
  1.32835638e-02 -1.14008260e-02 -2.00998058e-02  5.85066256e-01
  4.85826764e-04 -7.79518354e-04 -2.71475018e-03 -9.54095595e-03
 -5.43061744e-03  9.80393202e-03 -6.74232445e-03 -4.56127487e-04
  6.98228584e-03  3.16126382e-02  4.51745833e-01 -8.75962773e-03
  2.55888711e-03  4.24288916e-03  2.89829752e-01  7.02384480e-03
 -1.95118148e-03  1.41497186e-02 -1.06379062e-02  7.72681171e-01
 -5.66629496e-03 -6.07290727e-04  6.53024292e-03 -4.82656350e-03
  4.64013426e-03  4.54344467e-03 -2.99114626e-03  8.39450265e-03
 -2.46081254e-03  8.90955730e-04  1.15962949e-03 -1.87324018e-03
 -1.05100582e-02 -9.30826494e-03 -1.20817256e-03  5.97769067e-01
 -4.76785550e-03 -1.13499768e-02  2.89944327e-03  8.48927303e-01
 -1.08913594e-02  2.27590557e-03 -1.37140208e-03 -6.37571973e-03
  5.84057271e-03  5.67938275e-03  5.35091238e-03 -8.20744013e-03
  1.29918968e-02 -2.29924207e-03 -1.34644651e-04  8.65576275e-03
 -4.30407621e-03  5.68131266e-03  7.55447763e-03 -9.41323055e-03
  1.82989715e-02 -1.15681823e-03 -2.61377470e-03 -8.57725574e-03]
```

Followingly we plot $\|\mathbf{w}^t - \mathbf{w}_{ML}\|_2$ as a function of t.

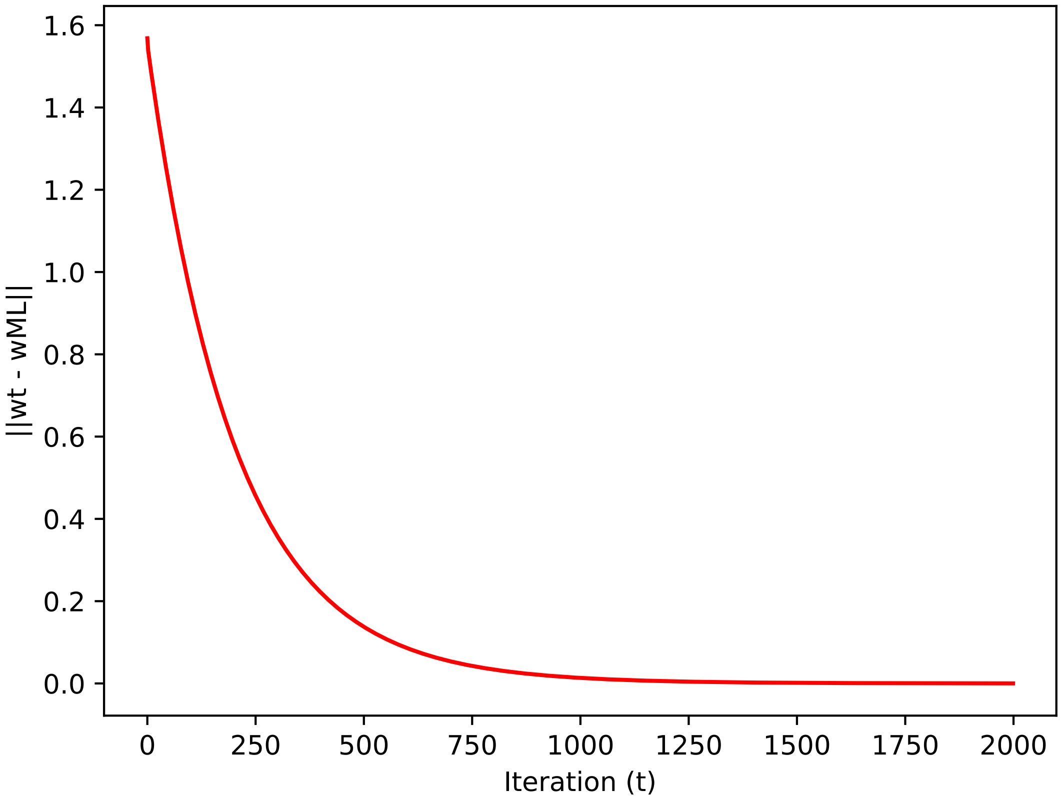


Figure 7: Convergence in Norm Difference (Least Square)

Analogously, we plot the MSE Loss as a function of epoch.

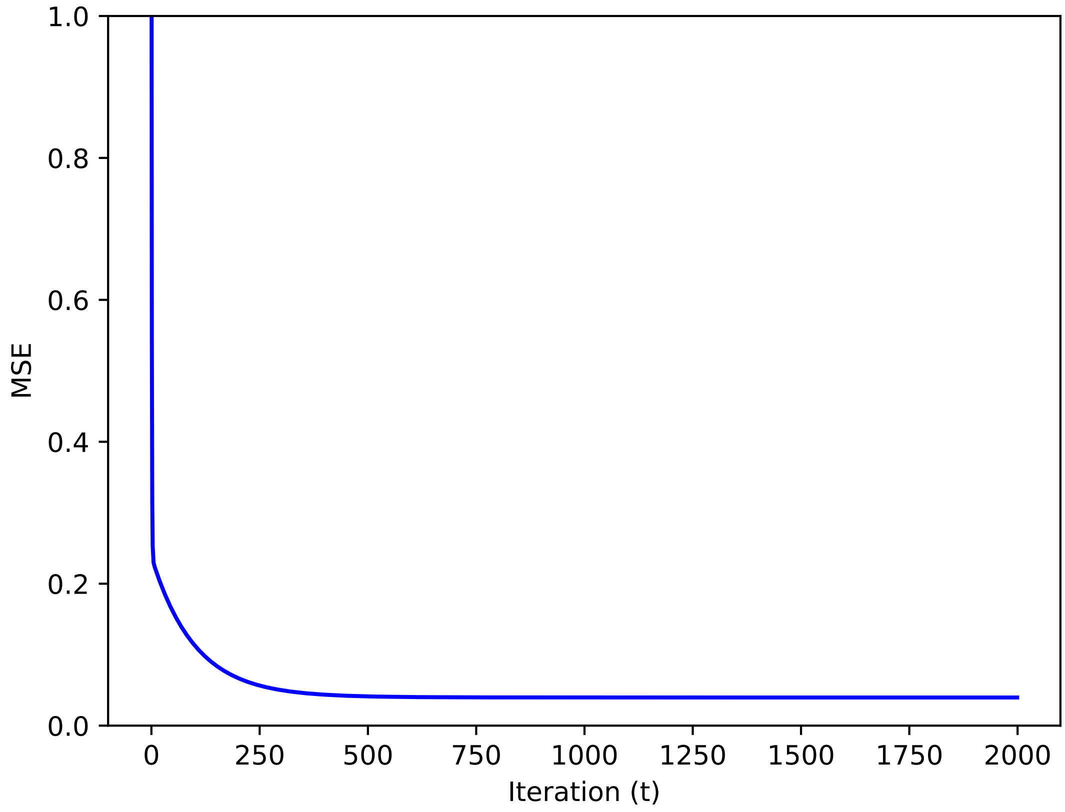


Figure 8: Convergence in Loss (Least Square)

2.2.6 Observations

Followingly, we mention the observations made on the data/ plots obtained.

1. **Tuning the Learning rate:** Here are some observations/ inference on the tuning of learning rate:

   (a) **Decreasing Loss with Increasing Learning Rate**: Initially, as the learning rate increases from 5×10^{-8} to 3×10^{-6}, the loss decreases significantly from 0.1939 to 0.0406. This suggests that higher learning rates initially lead to faster convergence and lower loss.

   (b) **Minimal Loss**: There's a plateau in loss around the learning rates of 3×10^{-6} to 4×10^{-6}, where the loss remains approximately constant (0.0406 to 0.0407). This indicates that this range of learning rates is likely optimal for this problem.

   (c) **Divergence**: Beyond a learning rate of 4×10^{-6}, the loss starts to increase drastically, indicating that the learning rate is too high, causing divergence rather than convergence.

   (d) **NaN Loss**: At learning rates of 1×10^{-5} and 5×10^{-5}, the loss becomes NaN (not a number), which suggests that the optimization algorithm encountered numerical instability or overflow issues at these learning rates.

   (e) **Norm Difference**: The norm difference seems to decrease with increasing learning rate up to a certain point, indicating that the parameter updates become smaller as the learning rate increases. However, beyond a certain threshold, the norm difference starts to increase, indicating instability in the optimization process.

2. **The MSE Loss and Norm DIfference:** From the plots for batch gradient descent with the optimal learning rate of 3×10^{-6}, the following observations can be made:

   (a) **Decreasing Loss over Epochs**: As the number of epochs increases, the loss decreases gradually, indicating that the optimization process is converging towards a minimum. The loss decreases from 1.4943 in the first epoch to approximately 0.0397 after 2000 epochs.

   (b) **Convergence**: The norm difference also decreases over epochs, indicating that the parameter updates become smaller as the optimization progresses. This suggests that the algorithm is converging towards a stable solution.

   (c) **Stability**: The learning rate remains constant throughout all epochs, which ensures stability in the optimization process.

   (d) **Time Taken**: The total CPU time taken for training is approximately 206.305 seconds, indicating the computational cost of training the model.

   (e) **Optimization Progress**: The rate of decrease in both loss and norm difference slows down as the number of epochs increases, indicating that the optimization is nearing convergence.

Overall, the results indicate that the batch gradient descent with the optimal learning rate effectively minimizes the loss and converges towards a stable solution within a reasonable time frame.

2.3 Stochastic Gradient Descent

To solve the least squares problem using gradient descent with stochastic updates, we aim to minimize the mean squared error (MSE) between the predicted outputs of our model and the actual target values in the training data.

Let $\mathbf{X}$ be the design matrix of size $N \times D$, where N is the number of data points and D is the number of features. Each row of $\mathbf{X}$ represents a data point. The target vector $\mathbf{y}$ has size $N \times 1$, containing the observed target values.

We define our linear regression model as:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$$

where $\mathbf{w}$ is the parameter vector of size $D \times 1$ to be estimated, and $\hat{\mathbf{y}}$ represents the predicted outputs.

The objective function $J(\mathbf{w})$ to be minimized is the mean squared error (MSE), given by:

$$J(\mathbf{w}) = \frac{1}{N}||\mathbf{y} - \mathbf{X}\mathbf{w}||_2^2$$

To minimize $J(\mathbf{w})$ using gradient descent with stochastic updates, we initialize $\mathbf{w}$ with random values and iteratively update it using the gradient of $J(\mathbf{w})$ with respect to $\mathbf{w}$. The update rule for $\mathbf{w}$ is given by:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla J(\mathbf{w})$$

where α is the learning rate, and $\nabla J(\mathbf{w})$ is the gradient of the objective function.

For stochastic gradient descent, at each iteration, we randomly select a stochastic of data points and compute the gradient using only those data points. The gradient with respect to $\mathbf{w}$ is given by:

$$\nabla J(\mathbf{w}) = \frac{2}{|B|}\mathbf{X}_B^T(\mathbf{X}_B\mathbf{w} - \mathbf{y}_B) \tag{23}$$

where $|B|$ is the size of the stochastic B, $\mathbf{X}_B$ is the design matrix corresponding to the stochastic, and $\mathbf{y}_B$ is the target vector corresponding to the stochastic.

The step size α is a crucial hyperparameter in gradient descent. Choosing a suitable step size is essential for convergence and stability of the algorithm. A common approach is to use a fixed step size or to adaptively adjust the step size based on the magnitude of the gradient or the past history of gradients.

By iteratively updating $\mathbf{w}$ using stochastic gradient descent with a suitable step size, we can find the parameter vector $\mathbf{w}$ that minimizes the mean squared error and provides the best-fitting linear model for the given data.

Algorithm 4 : Gradient Descent with stochastic Updates for Least Squares

```
Input: Design matrix $\mathbf{X}$, target vector $\mathbf{y}$, learning rate $\alpha$, number of iterations $T$, stochastic size $|B|$
Initialize: Parameter vector $\mathbf{w}$ randomly
for $t = 1$ to $T$ do
    Randomly select a stochastic $B$ from $\mathbf{X}$ and $\mathbf{y}$
    Compute stochastic design matrix $\mathbf{X}_B$ and stochastic target vector $\mathbf{y}_B$
    Compute gradient: $\nabla J(\mathbf{w}) = \frac{1}{|B|}\mathbf{X}_B^T(\mathbf{X}_B\mathbf{w} - \mathbf{y}_B)$
    Update parameter vector: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla J(\mathbf{w})$
end for
Output: Optimized parameter vector $\mathbf{w}$
```

2.3.1 Implementation of Stochastic Gradient Descent

Note that the program written in Subsubsection 2.2.2 is written generally and is used for the implementation of the stochastic gradient descent as well (by just changing the batch size).

2.3.2 Tuning the learning rate

For the given dataset the Learning Rate was tuned on several values; the following table summarizes the MSE loss and the $||\mathbf{w}-\mathbf{w}_{ML}||$ obtained on each of the final $\mathbf{w}$ value obtained. Note the following parameter values that are set:

- The maximum # of iteration: 2000,
- $\mathbf{w}$ is initialized as a null vector in $\mathbb{R}^d$;

- the threshold relative error: $\mathbf{10^{-10}}$, and
- the batch size: 100.

We follow the analogous method explained in Subsubsection 2.2.3 to tune the learning rate. Note that the Learning rate was tuned based on the minimisation of the final value obtained for the MSE.

Table 7: Tuning Learning rate for stochastic mode (batch size: 100) of gradient descent in maximum likelihood estimation

Learning Rate	$\frac{1}{N}\|\|\mathbf{y} - \mathbf{X}\mathbf{w}\|\|_2^2$	$\|\|\mathbf{w} - \mathbf{w}_{ML}\|\|_2^2$
1e-07	0.8999073750337343	1.584642871344514
5e-07	0.23389336898836638	1.5538436772556898
1e-06	0.2274840253370624	1.5380028888395951
5e-06	0.18430050810478962	1.4395429544927338
1e-05	0.14417565371967536	1.3301954735326766
5e-05	0.04835784367178149	0.7715159812646876
0.0001	0.041079427920576186	0.46456175308417935
0.0002	0.04093152590827652	0.2503472533825705
0.00025	0.04140205214726559	0.20769582311748505
0.0005	nan	nan
0.001	nan	nan
0.005	nan	nan
0.01	nan	nan

Thus, the optimal learning rate obtained is **0.0002** with the MSE and the norm difference being 0.04093152590827652 and 0.2503472533825705 respectively. Followingly, we plot the MSE Loss obtained as a function of Learning rate.

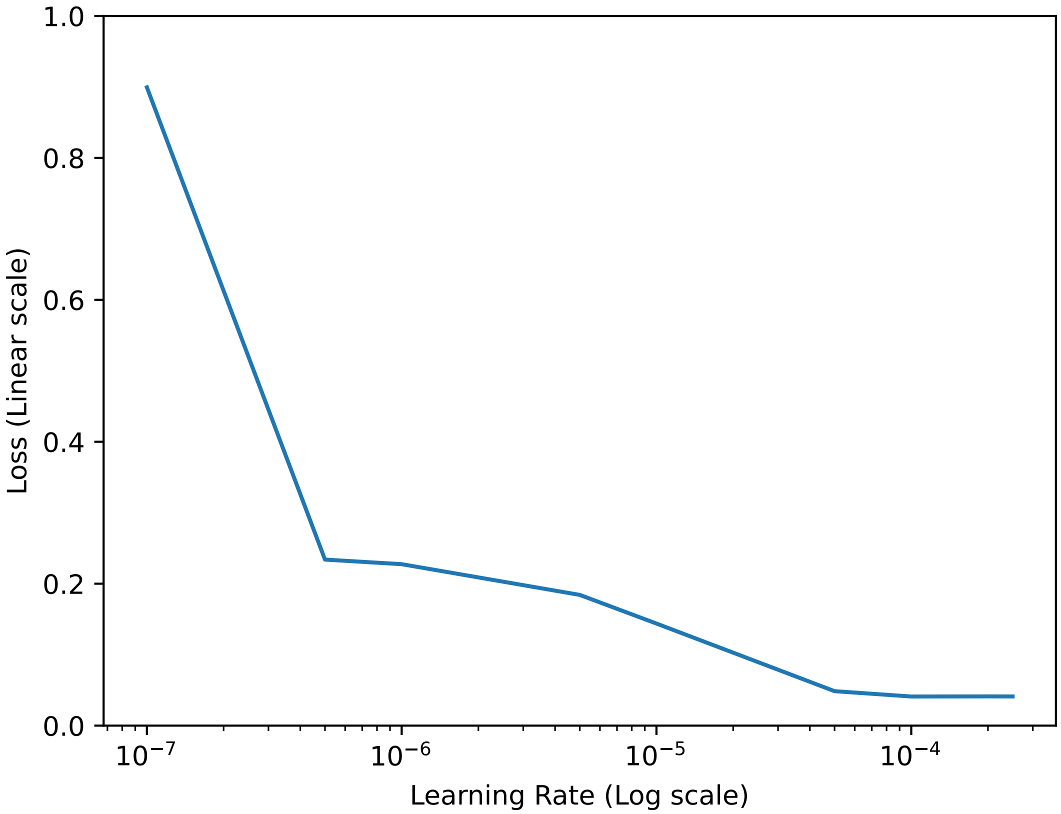


Figure 9: Loss vs. Learning Rate (Least Square Regression: Stochastic mode)
Batch Size: 100

2.3.3 Parameter Values

The following table represents the final set of values chosen/ tuned for the stochastic mode of gradient descent for the maximum likelihood estimation.

Parameter	**Value**
regression	Least Square
Learning Rate	0.0002
Max # of iterations	2000
$\mathbf{w}$ initialization	$\mathbf{0} \in \mathbb{R}^d$
Threshold relative MSE	1e-10
Batch size	10

Table 8: Parameters chosen/ set/ tuned for the stochastic gradient descent for maximum likelihood extimation

2.3.4 Results

We obtain the following result through the stochastic gradient descent for the maximum likelihood estimation.

Epoch (t)	$\frac{1}{N}\|\|\mathbf{y}-\mathbf{X}\mathbf{w}^t\|\|_2^2$	$\|\|\mathbf{w}^t-\mathbf{w}_{ML}\|\|_2^2$
1	0.23836301359141826	1.5526297623688807
2	0.23787548158378877	1.5487115567070382
3	0.23719734551956673	1.5418895331382811
4	0.23859701049114018	1.5374791321713634
5	0.2339805820896285	1.5328018034081399
6	0.23890316064473055	1.5285007398663206
7	0.2350918749001917	1.5238055436071833
8	0.2304841447643837	1.5198081724645338
9	0.2291725054785824	1.5150670969169266
10	0.22774124964284093	1.5110315356420114
100	0.14361840108084262	1.1256057323530688
200	0.09441359415215793	0.8160418619237121
300	0.06815094239475816	0.59252093545935
400	0.056675449158868574	0.4279072382076132
500	0.047499478466454574	0.30904666867369224
600	0.04399988985045376	0.2240345199632608
700	0.042165544370606864	0.16412196633128368
800	0.04118811712386882	0.12262833177575873
900	0.04136200230971499	0.09025767662371033
1000	0.04042576183425512	0.06940476205423117
1100	0.04012770169923226	0.05505123186774433
1200	0.04061903834437669	0.04513737410178456
1300	0.04002168940707822	0.03815137684407447
1400	0.039964561908378764	0.03513493461466228
1500	0.04055402074929779	0.03488580202405366
1600	0.03977490197379677	0.031369466955543386
1700	0.039888581843712134	0.02787342491019798
1800	0.03977658985200903	0.028545937441517722
1900	0.039744142367523626	0.026272731615442114
2000	0.04012771355816948	0.027103775178684716

Also, followingly, the CPU time taken for training and the final **w** obtained is written down.

1. CPU time taken for training: 68.87136408700098 seconds

2. wML (Stochastic Mode of Gradient Descent w/- batch size: 100):

```
[-7.77872928e-03 -1.14325510e-02 -6.59082755e-03  4.58148816e-03
  1.88390897e-01  3.65336322e-03  6.40492535e-03  1.78573588e-01
  5.08013480e-03  5.00852099e-01  1.36475805e-02  9.04730226e-03
  1.38103905e-02  4.14178944e-03  1.06366796e-02 -3.14795943e-03
  1.48411928e-03 -1.06932126e-04 -1.47254545e-02 -1.42080464e-03
 -1.32958696e-02 -1.19174523e-02  1.33621154e-02 -1.70217708e-02
  4.13909139e-03 -1.11291823e-02 -4.83352635e-03  5.99077068e-03
```

```
                                  4.66640518e-03 -9.39601177e-03 -8.43315483e-03  7.79953582e-04
                                  3.74841972e-03  2.86638098e-03 -1.26251476e-02  2.77244247e-03
                                  1.42645214e-02 -1.08657937e-02 -1.94183144e-02  5.87369175e-01
                                 -4.27967283e-03 -1.96532347e-04 -2.46679798e-03 -1.22720563e-02
                                 -6.64317146e-03  9.94414984e-03 -6.87604909e-03 -1.61394844e-03
                                  9.54419101e-03  2.73863120e-02  4.51837670e-01 -8.60441982e-03
                                  6.55197776e-03  9.30600223e-03  2.92878624e-01  4.47851044e-03
                                 -3.17761615e-05  1.66384471e-02 -1.40898179e-02  7.68956749e-01
                                 -5.42851175e-03  2.78091513e-04  7.75858844e-03 -6.49709824e-03
                                  6.60227747e-03  2.34193128e-03  2.47228851e-03  9.08338741e-03
                                 -3.34095762e-03 -1.00049113e-03 -3.82322788e-04 -3.43449865e-04
                                 -9.65392672e-03 -1.14107718e-02 -2.40475678e-03  5.99436079e-01
                                 -6.20110062e-03 -7.85329589e-03  1.62040699e-03  8.45297700e-01
                                 -1.24169033e-02  5.38098072e-03 -2.21214279e-03 -6.05462764e-03
                                  2.03952608e-03  9.47403720e-03  5.56066282e-03 -1.33341495e-02
                                  1.07443487e-02 -4.10081386e-03 -2.97207797e-03  1.20937168e-02
                                 -4.82971226e-03  8.04607794e-03  5.39220015e-03 -6.56674426e-03
                                  1.91628057e-02 -5.13036445e-03 -3.88332521e-03 -1.25499749e-02]
```

Followingly we plot $\|\mathbf{w}^t - \mathbf{w}_{ML}\|_2$ as a function of t.

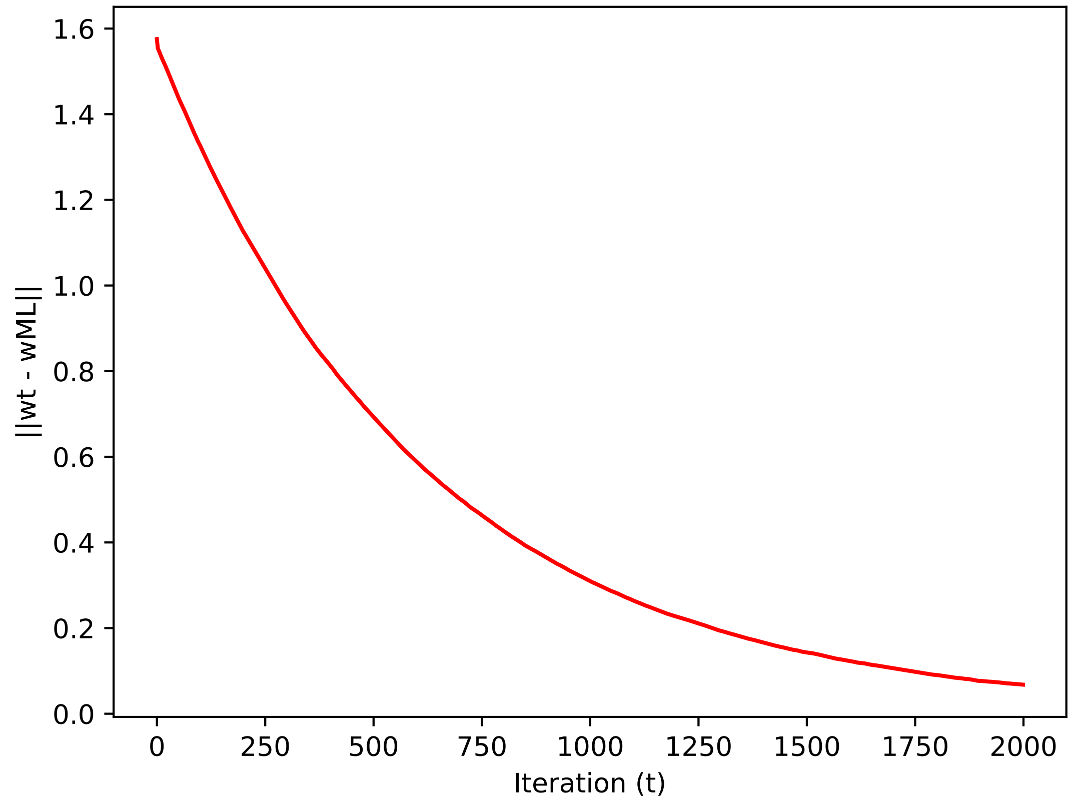


Figure 10: Convergence in Norm Difference (Least Square)
Batch size: 100

Analogously, we plot the MSE Loss as a function of epoch.

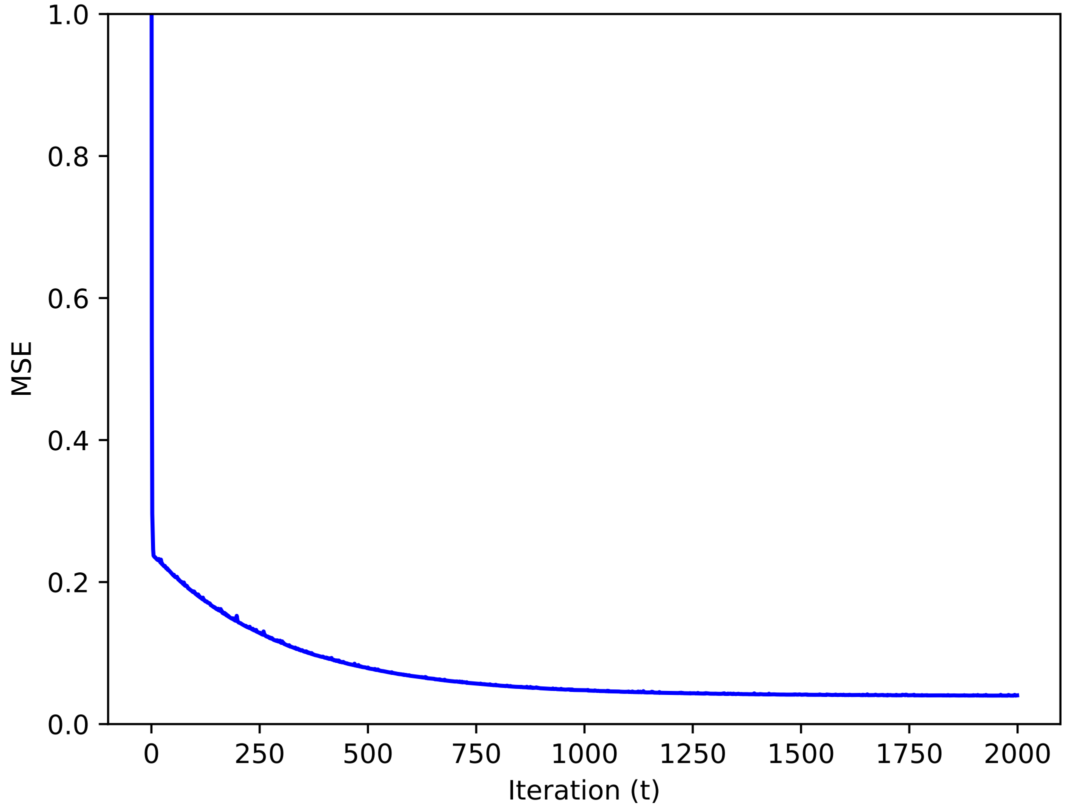


Figure 11: Convergence in Loss (Least Square)
Batch size: 100

2.3.5 Observations

Here are some observations based on the results obtained. The stochastic gradient descent (SGD) with a batch size of 100 converged faster than the batch gradient descent (BGD) because of its inherent nature of updating the model parameters more frequently. Here's why SGD with a batch size of 100 have performed better in terms of convergence speed:

1. **Frequent Updates:** In SGD, the model parameters are updated after processing each mini-batch of data. With a batch size of 100, these updates are more frequent compared to BGD, where updates are made after processing the entire dataset. This frequent updating allows SGD to quickly adjust the model parameters in a direction that reduces the loss.

2. **Efficient Use of Data:** With a batch size of 100, SGD processes smaller chunks of data at a time, allowing it to efficiently utilize the entire dataset in each epoch. This can lead to faster convergence compared to BGD, which processes the entire dataset in each epoch and may require more iterations to converge.

3. **Avoidance of Local Minima:** The frequent updates in SGD with smaller batches can help the optimization process to escape local minima more easily. It allows the algorithm to explore the parameter space more dynamically, potentially leading to finding better solutions.

However, despite converging faster, SGD with a batch size of 100 resulted in a slightly higher final loss compared to BGD. This discrepancy can be attributed to the stochastic nature of SGD.

The updates in SGD are noisy due to the random selection of mini-batches, which can lead to fluctuations in the loss function during training. As a result, while SGD may converge faster, it might settle in a suboptimal solution with a slightly higher final loss compared to the more deterministic BGD.

2.4 Ridge Regression

Ridge regression is a linear regression technique that adds a regularization term to the least squares objective function in order to prevent overfitting. The regularization term penalizes large values of the coefficients, encouraging them to be small.

In ridge regression, the objective function to be minimized is given by:

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mathbf{X}_i\mathbf{w})^2 + \lambda\sum_{j=1}^{d} w_j^2 \tag{24}$$

where:

- N is the number of data points,
- $\mathbf{x}_i$ is the feature vector for the ith data point,
- y_i is the target value for the ith data point,
- $\mathbf{w}$ is the vector of coefficients,
- d is the number of features,
- λ is the regularization parameter, and
- w_j represents the jth coefficient.

The first term in the objective function measures the sum of squared errors between the predicted values and the actual target values, while the second term penalizes large values of the coefficients. The regularization parameter λ controls the strength of the penalty: larger values of λ result in smaller coefficients, reducing the model's complexity and making it less prone to overfitting.

The ridge regression objective function can be minimized using various optimization techniques such as gradient descent or closed-form solutions.

Ridge regression is particularly useful when dealing with multicollinearity (high correlation among predictor variables) or when the number of features is close to or exceeds the number of data points, as it helps stabilize the coefficient estimates and improves the generalization performance of the model.

2.4.1 Implementation of Ridge Regression

Note that the program written in Subsubsection 2.2.2 is written generally and is used for the implementation of the gradient descent in ridge regression. Please refer to it to find the relevant changes.

2.4.2 Tuning the hyperparameters

In this section, we discuss about the tuning of two hyperparameters, that are — Lambda and Learning rate.

2.4.2.1 Tuning Lambda

For the given dataset the Lambda was tuned on several values from 1e-5 to 1e5 using the analytical solution of (Please refer to the results folder output.txt for the exact data). Note the following parameter values that are set:

- The maximum # of iteration: 2000,
- $\mathbf{w}$ is initialized as a null vector in $\mathbb{R}^d$;
- the threshold relative error: $\mathbf{10^{-10}}$, and
- the batch size: $\boldsymbol{N}$.

The optimal Lambda obtained is 1.7886495290574351 with the MSE being 0.04065599997176069. Followingly we plot the MSE Loss obtained against various values of Lambda.

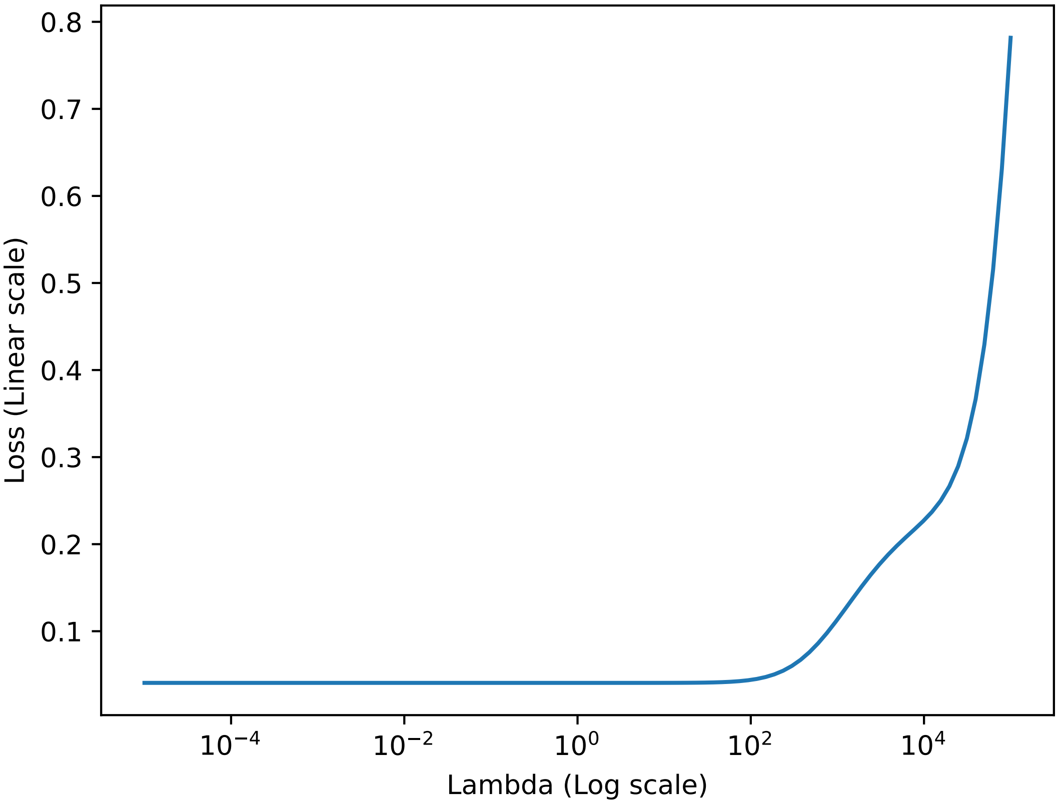


Figure 12: Loss vs. Lambda (Ridge Regression: Batch Mode)

2.4.2.2 Tuning the Learning rate

Analogous to the previous section, we tune the learning rate with the value of the parameter mentioned above. **The optimal Learning rate obtained is precisely the same as that obtained in the batch gradient descent for maximum likelihood estimation; that is — 3e-6 but with the loss and the norm difference being 0.04065565296032657 and 0.19985011408069767.** Followingly, we plot the MSE Loss obtained as a function of Learning rate.

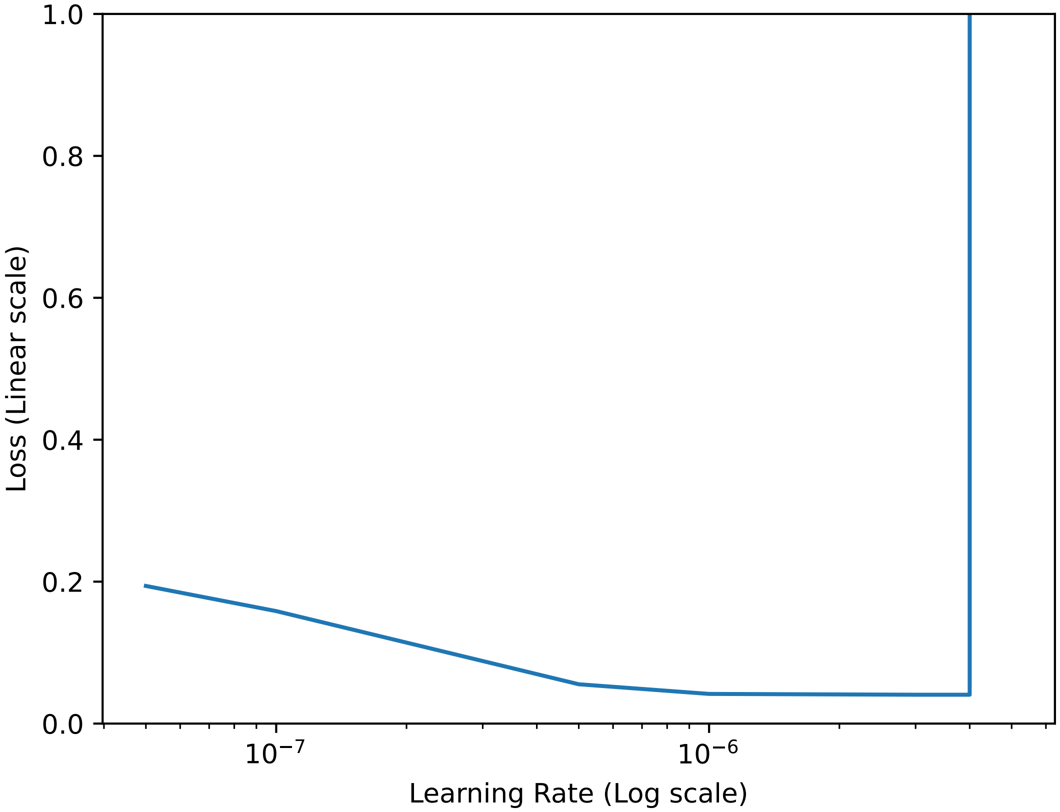


Figure 13: Loss vs. Learning Rate (Ridge Regression: Batch mode)

2.4.3 Parameter Values

The following table represents the final set of values chosen/ tuned for the gradient descent of the ridge regression.

Parameter	Value
regression	Ridge
Learning Rate	3e-6
Max # of iterations	2000
$\mathbf{w}$ initialization	$\mathbf{0} \in \mathbb{R}^d$
Threshold relative MSE	1e-10
Batch size	N
Lambda	1.7886495290574351

Table 10: Parameters chosen/ set/ tuned for the gradient descent of ridge regression

2.4.4 Results

We obtain the following result through the gradient descent for the ridge regression.

Epoch (t)	$\frac{1}{N}\lvert\lvert\mathbf{y}-\mathbf{X}\mathbf{w}^t\rvert\rvert_2^2$	$\lvert\lvert\mathbf{w}^t-\mathbf{w}_{ML}\rvert\rvert_2^2$
1	1.4942837892727636	1.5648810492272138
2	0.5569552836758438	1.545271649466341
3	0.3161170369921348	1.5345606305493549
4	0.25320341266751945	1.5261754958775569
5	0.23570643759864351	1.5184165727108505
6	0.2298364983500413	1.5108466807050966
7	0.22694177990592498	1.5033534278102048
8	0.22482476321117428	1.4959078237476144
9	0.22291817258427596	1.4885022851869625
10	0.22108004391781189	1.4811347232535463
100	0.11380270912122332	0.949413228730078
200	0.06738629741257426	0.5816838664454413
300	0.0501599587354277	0.35792922603124705
400	0.043698396876636764	0.22116813186655565
500	0.04124675462916065	0.13721121900143254
600	0.040304640971958	0.0854509902384705
700	0.03993724766181473	0.05340951100901734
800	0.0397914248243712	0.033496934410473314
900	0.039732265146167095	0.02107593415372092
1000	0.039707593626054305	0.013300728078599779
1100	0.039696943899884686	0.008417532860285654
1200	0.03969215054998423	0.005341122861528204
1300	0.03968988691321194	0.0033973504911706384
1400	0.039688761569019654	0.0021658802868692153
1500	0.03968817309759393	0.0013837125423801528
1600	0.03968785096010144	0.0008857485021768367
1700	0.03968766772019037	0.000568026409058018

Epoch (t)	$\frac{1}{N}\|\|\mathbf{y}-\mathbf{X}\mathbf{w}^t\|\|_2^2$	$\|\|\mathbf{w}^t-\mathbf{w}_{ML}\|\|_2^2$
1800	0.039687560292822215	0.00036489260296 22876
1900	0.03968749586666205	0.00023477351151319202
2000	0.039687456585143915	0.00015127725137632872

Also, followingly, the CPU time taken for the training of gradient descent, wR (analytical) and wR (gradient descent) are written down.

1. wR (Analytical):

```
[-7.73147278e-03 -1.35408330e-02 -3.55762109e-03  2.71291974e-03
  1.88188324e-01  2.74078975e-03  9.51110034e-03  1.79478486e-01
  3.86404400e-03  4.98651035e-01  8.42309314e-03  4.37808323e-03
  1.42858676e-02  4.01722389e-03  9.45372002e-03 -1.06929162e-03
  3.48553034e-03  1.30103784e-03 -9.28271861e-03 -2.30257892e-03
 -1.16388267e-02 -1.00846229e-02  8.00905390e-03 -9.95201112e-03
  6.11939130e-03 -4.55119103e-03 -3.01457699e-03  8.21931538e-03
  1.20773142e-02 -6.69882657e-03 -8.55724081e-03  1.04780474e-03
  5.09213467e-03  6.05308273e-03 -1.33322608e-02  1.14249342e-03
  1.33635816e-02 -1.12493271e-02 -1.99807820e-02  5.83932666e-01
  5.73533411e-04 -6.79482484e-04 -2.59982401e-03 -9.43929606e-03
 -5.31082561e-03  9.84394055e-03 -6.66669530e-03 -3.75102012e-04
  7.03625004e-03  3.16284476e-02  4.50919026e-01 -8.65818248e-03
  2.65334667e-03  4.29233593e-03  2.89298360e-01  7.07157511e-03
 -1.85193391e-03  1.41877517e-02 -1.04998910e-02  7.71118879e-01
 -5.54128540e-03 -4.91090746e-04  6.64730114e-03 -4.69757265e-03
  4.70757932e-03  4.60795821e-03 -2.88696729e-03  8.49222256e-03
 -2.33109929e-03  9.30145380e-04  1.28250746e-03 -1.76721065e-03
 -1.03789942e-02 -9.18659784e-03 -1.06462598e-03  5.96627719e-01
 -4.63758491e-03 -1.11979153e-02  3.02320421e-03  8.47207250e-01
 -1.07726740e-02  2.39649699e-03 -1.25569312e-03 -6.32514232e-03
  5.91630098e-03  5.74647374e-03  5.41582379e-03 -8.09938250e-03
  1.30466059e-02 -2.21655705e-03 -5.99797996e-05  8.70263511e-03
 -4.23458983e-03  5.73789965e-03  7.63360848e-03 -9.25867889e-03
  1.83705135e-02 -1.03379484e-03 -2.51812820e-03 -8.43483339e-03]
```

2. CPU time taken for training: 192.73462437400303 seconds

3. wR (Batch Mode of Gradient Descent):

```
[-7.72735544e-03 -1.35348295e-02 -3.57364412e-03  2.70760933e-03
  1.88162392e-01  2.74486049e-03  9.49965706e-03  1.79454497e-01
  3.88140152e-03  4.98625775e-01  8.42269858e-03  4.38475695e-03
  1.42909656e-02  4.01712354e-03  9.46033129e-03 -1.07153030e-03
  3.50512866e-03  1.31827323e-03 -9.27121947e-03 -2.28331210e-03
 -1.16370341e-02 -1.00761277e-02  8.00625596e-03 -9.95270174e-03
  6.12213726e-03 -4.54093797e-03 -3.01533776e-03  8.21586969e-03
  1.20720764e-02 -6.69366770e-03 -8.55745459e-03  1.03630529e-03
  5.11244583e-03  6.04999579e-03 -1.33232805e-02  1.13777407e-03
  1.33723333e-02 -1.12356119e-02 -1.99750589e-02  5.83903387e-01
  5.65085940e-04 -6.72504153e-04 -2.59541405e-03 -9.44001439e-03
 -5.30009346e-03  9.84114727e-03 -6.68320883e-03 -3.85558260e-04
  7.03343847e-03  3.16272242e-02  4.50901790e-01 -8.66541704e-03
  2.66038844e-03  4.28616434e-03  2.89280614e-01  7.05850751e-03
 -1.84530746e-03  1.41851450e-02 -1.04872876e-02  7.71057631e-01
 -5.53643619e-03 -4.68898383e-04  6.66761833e-03 -4.68426810e-03
  4.70934638e-03  4.60266065e-03 -2.88344739e-03  8.49875075e-03
 -2.31667540e-03  9.20389622e-04  1.29470568e-03 -1.76411114e-03
 -1.03716543e-02 -9.18193403e-03 -1.03794995e-03  5.96599899e-01
 -4.61962843e-03 -1.11756502e-02  3.03752890e-03  8.47136493e-01
 -1.07716337e-02  2.40864240e-03 -1.24632436e-03 -6.34112537e-03
  5.91896775e-03  5.73331340e-03  5.41118135e-03 -8.10062741e-03
  1.30500174e-02 -2.21019973e-03 -7.20632536e-05  8.69235806e-03
 -4.24397359e-03  5.72443400e-03  7.63326234e-03 -9.23702870e-03
  1.83787402e-02 -1.02093721e-03 -2.51596358e-03 -8.42613796e-03]
```

Followingly, we plot $||\mathbf{w}^t - \mathbf{w}_R||_2$ as a function of t.

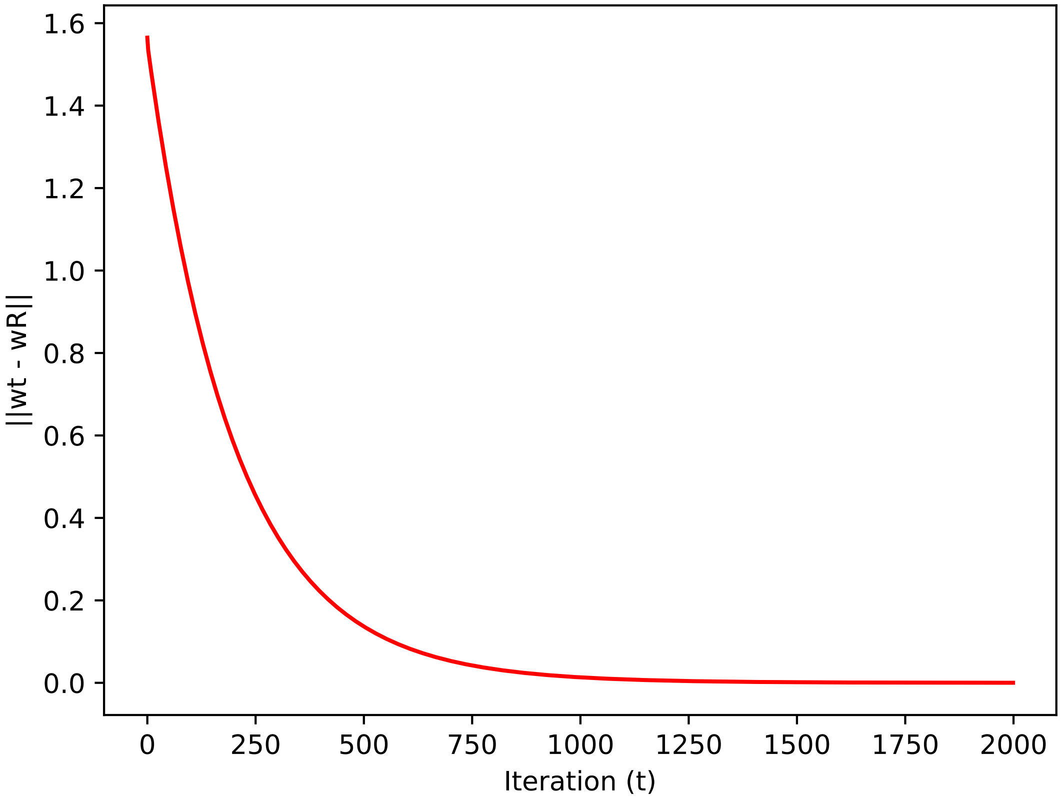


Figure 14: Convergence in Norm Difference (Ridge Regression)

Analogously, we plot the MSE Loss as a function of epoch.

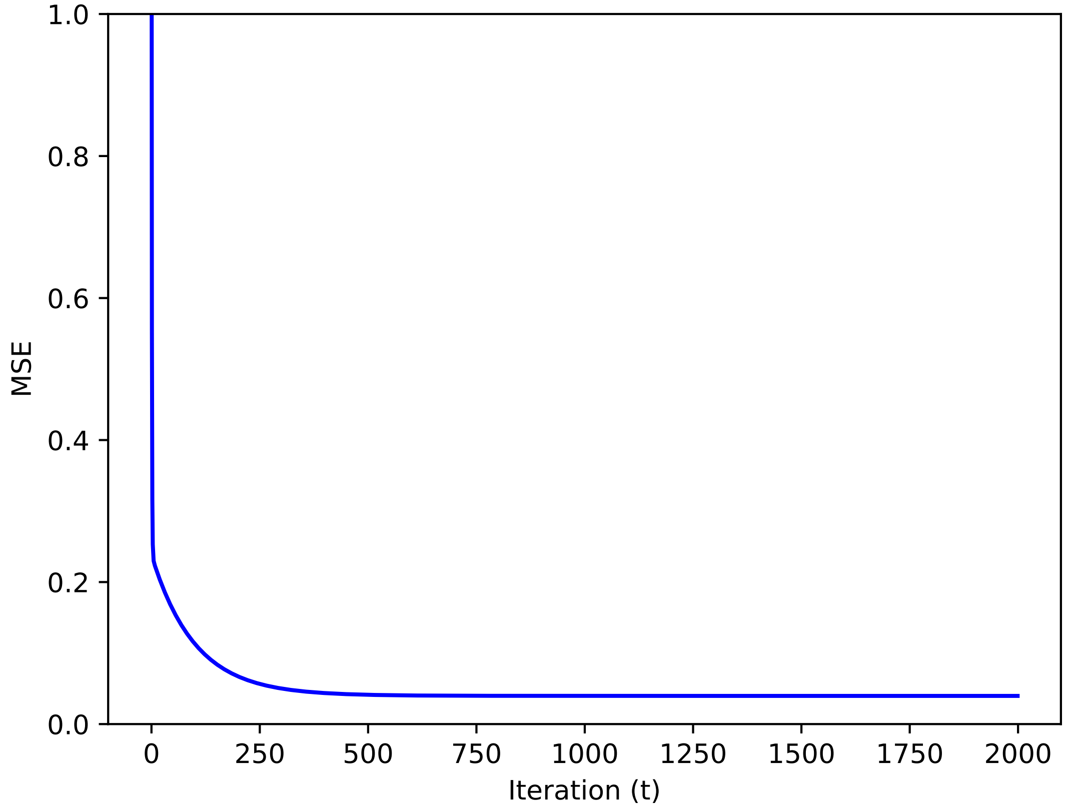


Figure 15: Convergence in Loss (Ridge Regression)

2.5 Test Errors and comparison between $\mathbf{w_R}$ and $\mathbf{w_{ML}}$:

The mean square error was computed on the test data and the following result is obtained.

Mean Square Error Loss computed on Test data ####################

1. Maximum Likelihood Estimartor (Analytical): 0.37072731116979096
2. Maximum Likelihood Estimartor (Batch Gradient Descent): 0.37070214860750766
3. Maximum Likelihood Estimartor (Stochastic Gradient Descent w/- batch size of 10): 0.3712169118961298
4. Ridge Regression (Analytical): 0.36997126278713083
5. Ridge Regression (Batch Gradient Descent): 0.36994676373274016

Clearly, **Ridge estimator generated lesser mean squared error as compared to the maximum likelihood estimator.** Here is the reason why:

1. **Multicollinearity**: When there is multicollinearity (high correlation) among the predictor variables, the maximum likelihood estimator can become unstable or have inflated variance. This leads to large fluctuations in the parameter estimates, resulting in higher MSE. Ridge regression introduces a bias into the parameter estimates to reduce variance, thus stabilizing

the estimates and often leading to lower MSE.

2. **High Dimensionality**: The maximum likelihood estimator may not even exist or may lead to overfitting due to its tendency to find a unique solution. Ridge regression, by introducing a penalty term to shrink the coefficients, can help in handling this high-dimensional data scenario, leading to lower MSE by reducing overfitting.

3. **Trade-off between Bias and Variance**: Ridge regression introduces a bias to the parameter estimates in exchange for reduced variance. This bias-variance trade-off often results in a lower overall MSE compared to the maximum likelihood estimator, especially when the reduction in variance outweighs the increase in bias.

3 References

[1] Alfons Juan and Enrique Vidal. Bernoulli mixture models for binary images. *Proceedings of the 17th International Conference on Pattern Recognition (ICPR)*, page 515, 2004.

[2] Qian Wang and Jian Wang. The EM algorithm for multi-dimensional Gaussian mixture model. *International Journal of Scientific and Research Publications*, 11(6):515, 2021.