A3 - Final

November 7, 2023

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

0.0.1 Exercise 1: Non-Parametric Estimation

[2]:
```
import torch
import torchvision
import torchvision.datasets as datasets

print("PyTorch version:", torch.__version__)
```

```
PyTorch version: 2.1.0+cpu
```

[104]:
```
# initializing training set
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=None)
mnist_train_loader = torch.utils.data.DataLoader(mnist_trainset, batch_size=64,␣
 ↪shuffle=True)

# initializing test set
mnist_testset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=None)
mnist_test_loader = torch.utils.data.DataLoader(mnist_testset, batch_size=64,␣
 ↪shuffle=True)
```

[107]:
```
# flatten the training set
flattened_train_images = mnist_trainset.data.view(mnist_trainset.data.size(0),␣
 ↪-1)
flattened_train_labels = mnist_trainset.targets

# flatten the test set
flattened_test_images = mnist_testset.data.view(mnist_testset.data.size(0), -1)
flattened_test_labels = mnist_testset.targets
```

[109]:
```
from sklearn.decomposition import PCA

# create a PCA object with 1 component
pca = PCA(n_components=1)
```

```
# fit the PCA model to the flattened training images
pca.fit(flattened_train_images)

# transform the training set and test set images into 1x1 vectors using PCA
transformed_train_images = pca.transform(flattened_train_images)
transformed_test_images = pca.transform(flattened_test_images)
```

[111]:
```
# filter the training set to include only classes 0 and 1
indices_train = (flattened_train_labels == 0) | (flattened_train_labels == 1)
filtered_train_images = transformed_train_images[indices_train]
filtered_train_labels = flattened_train_labels[indices_train]

# filter the test set to include only classes 0 and 1
indices_test = (flattened_test_labels == 0) | (flattened_test_labels == 1)
filtered_test_images = transformed_test_images[indices_test]
filtered_test_labels = flattened_test_labels[indices_test]
```

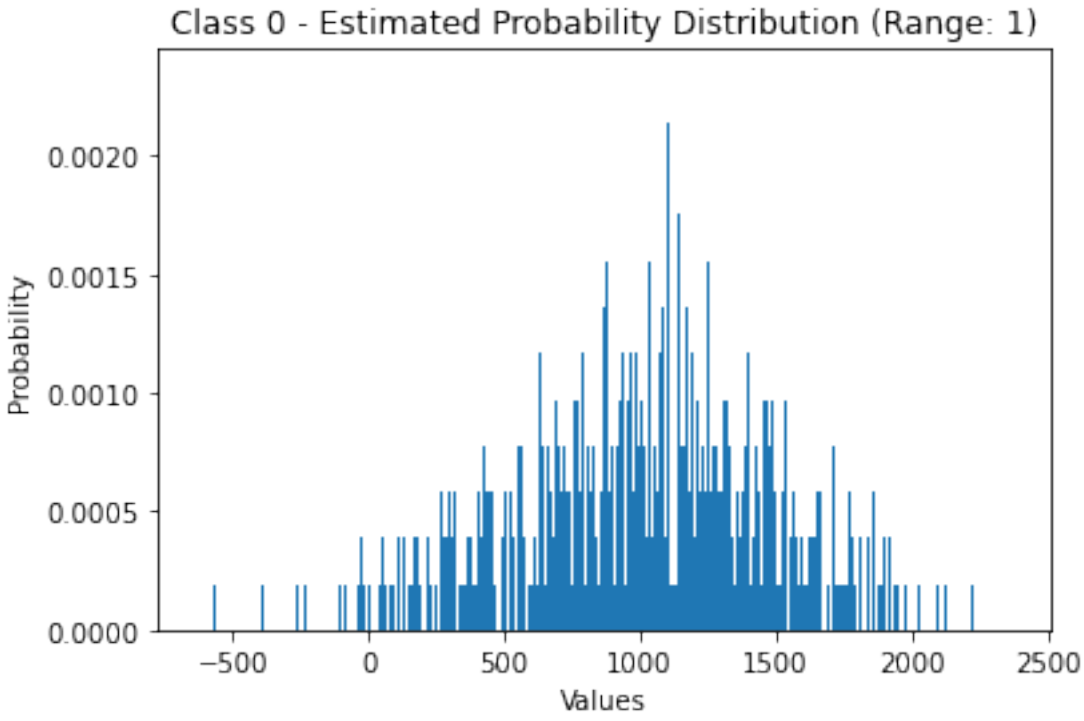
Question 1: Histogram-based estimation

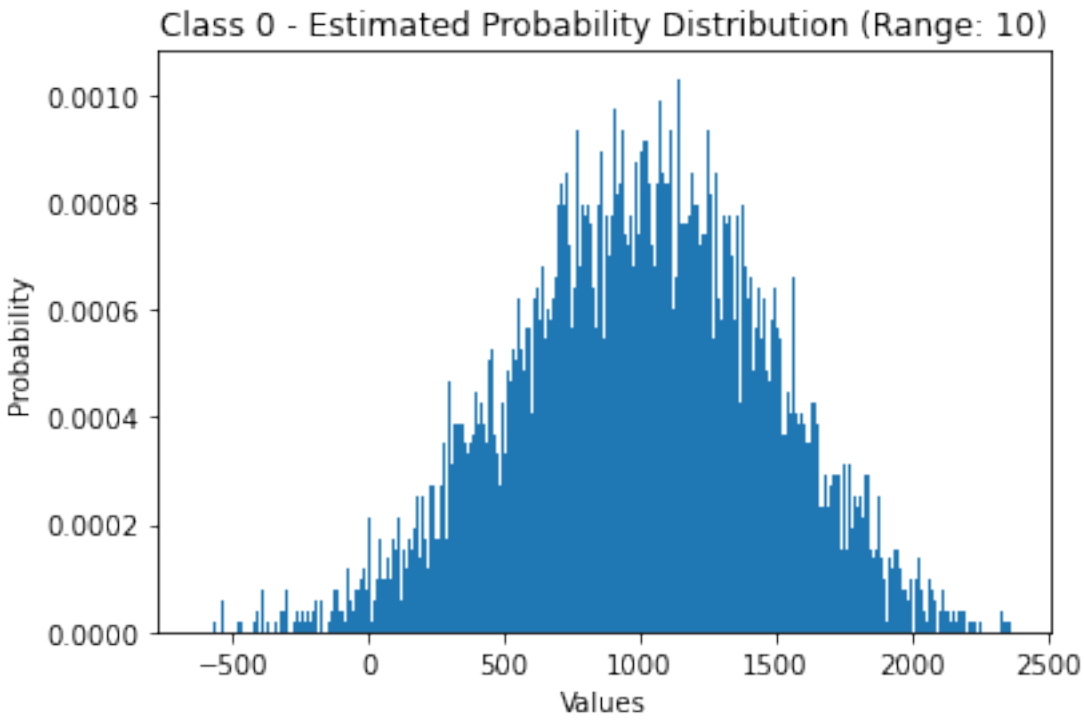
[112]:
```
# find the minimum and maximum values in the dataset
min_value = np.min(filtered_train_images)
max_value = np.max(filtered_train_images)

print("Minimum value in the dataset:", min_value)
print("Maximum value in the dataset:", max_value)
```

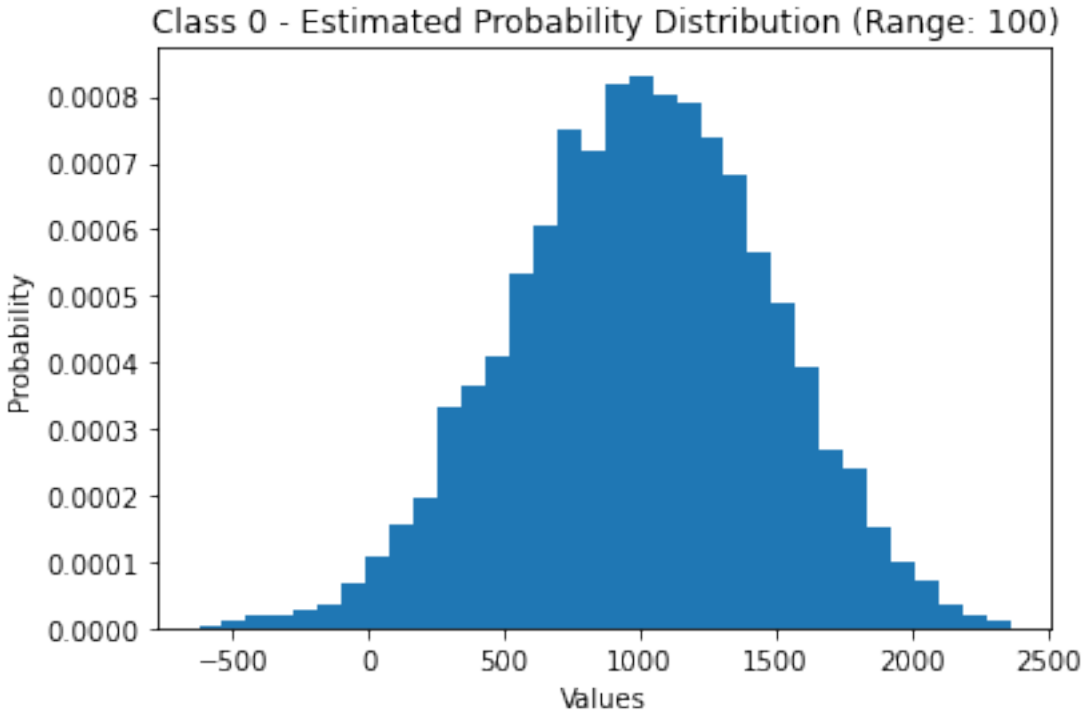
```
Minimum value in the dataset: -1081.6595180607333
Maximum value in the dataset: 2363.32360511183
```

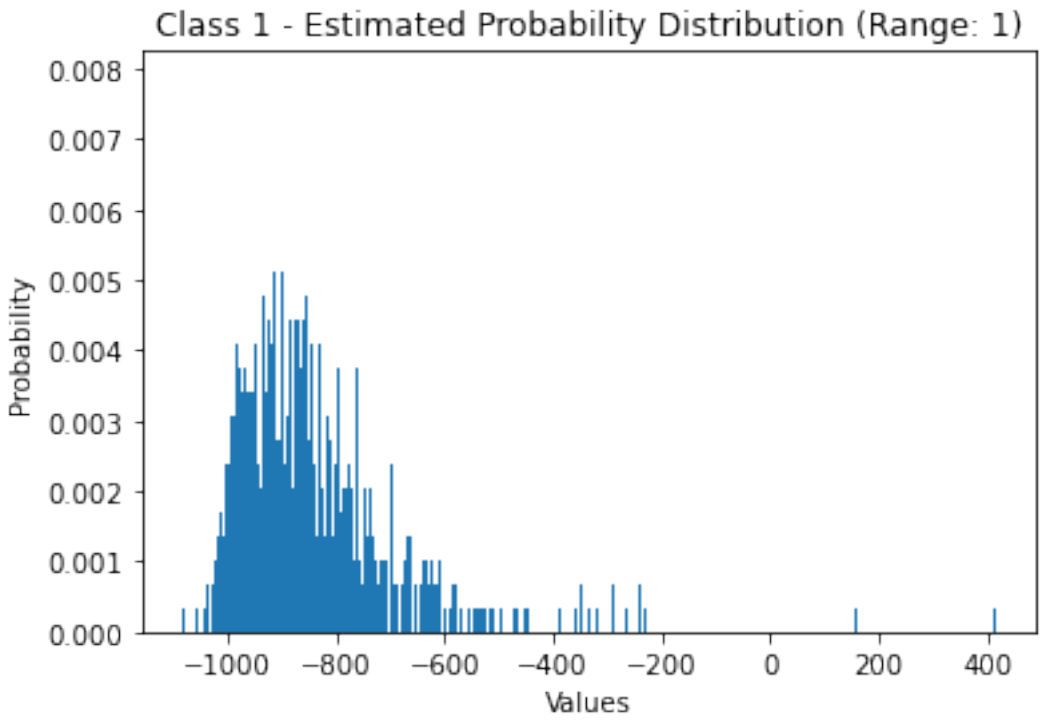
[113]:
```
ranges = [1, 10, 100]
labels = [0, 1]

for label in labels:
    # filter data for the current class
    class_data = filtered_train_images[filtered_train_labels == label]

    for r in ranges:
        num_bins = round((max_value - min_value)/r)

        # calculate histogram for the current class and range
        hist, bin_edges = np.histogram(class_data, bins=num_bins, density=True)
        bin_widths = bin_edges[1:] - bin_edges[:-1]

        plt.bar(bin_edges[:-1], hist, width=bin_widths, align='edge')
        plt.xlabel("Values")
        plt.ylabel("Probability")
```

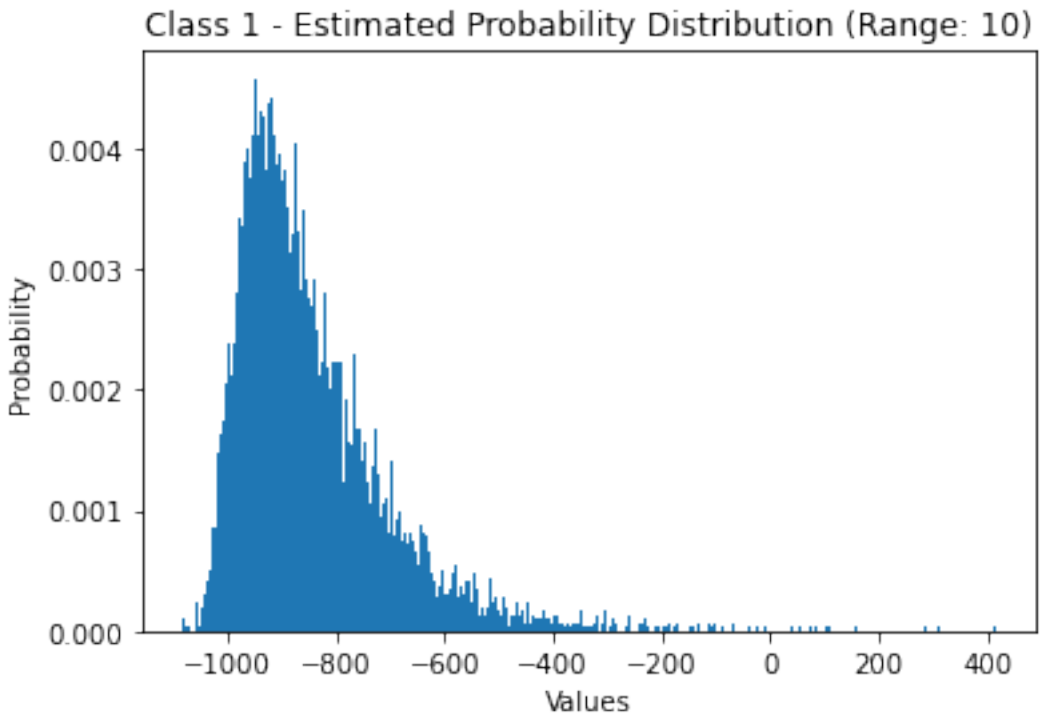
```
        plt.title(f"Class {label} - Estimated Probability Distribution (Range:␣
↪{r})")
        plt.show()
```

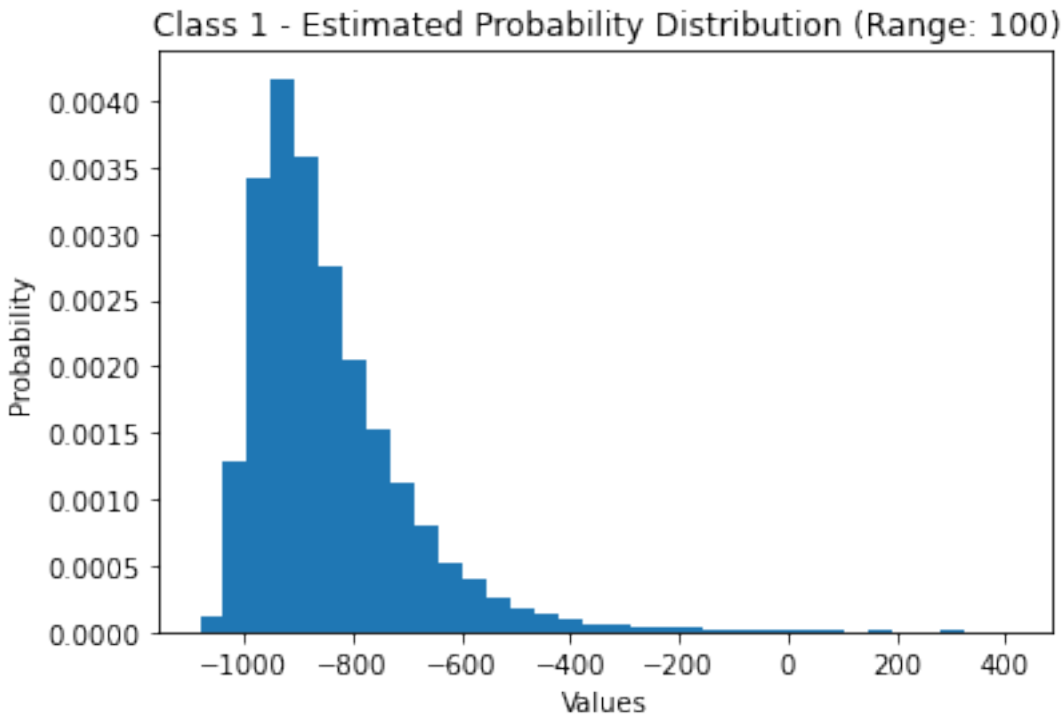



Class 0 - Estimated Probability Distribution (Range: 10)
Probability
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
−500
0
500
1000
1500
2000
2500
Values


Class 0 - Estimated Probability Distribution (Range: 100)
Probability
0.0008
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
−500
0
500
1000
1500
2000
2500
Values


Class 1 - Estimated Probability Distribution (Range: 1)
Probability
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0.000
−1000
−800
−600
−400
−200
0
200
400
Values


Class 1 - Estimated Probability Distribution (Range: 10)
Probability
0.004
0.003
0.002
0.001
0.000
−1000
−800
−600
−400
−200
0
200
400
Values



Question 2: ML Classifier

[116]:
```python
def ML_classify(width, x):
    num_bins = round((max_value - min_value)/width)

    # define the bin edges explicitly to cover the full data range
    bin_edges = np.linspace(min_value, max_value, num=num_bins+1, endpoint=True)

    # filtering for classes
    c1 = filtered_train_images[filtered_train_labels == 0]
    c2 = filtered_train_images[filtered_train_labels == 1]

    # calculate histogram for c1
    hist1, _ = np.histogram(c1, bins=bin_edges, density=True)

    # calculate histogram for c2
    hist2, _ = np.histogram(c2, bins=bin_edges, density=True)

    # determine which bin x belongs to in both histograms
    if x >= max_value:
        bin_idx_c1 = len(hist1)-1
        bin_idx_c2 = len(hist2)-1
    else:
```

```
        bin_idx_c1 = np.digitize(x, bin_edges) - 1  # subtract 1 to convert to␣
↪0-based indexing
        bin_idx_c2 = np.digitize(x, bin_edges) - 1

    # access the heights of the bins for x in both histograms
    height_c1 = hist1[bin_idx_c1]
    height_c2 = hist2[bin_idx_c2]

    # compare the heights to classify
    if height_c1 > height_c2:
        return 0
    else:
        return 1
```

[117]:
```
ranges = [1, 10, 100]

for r in ranges:
    print(f"Range = {r}")
    predicted_labels = []
    for sample in filtered_test_images:
        predicted_labels.append(ML_classify(r, sample))

    # calculate classification accuracy
    accuracy = np.mean(filtered_test_labels.numpy() == predicted_labels)
    print(f"Accuracy for range = {r}: {accuracy * 100:.2f}%\n")
```

```
Range = 1
Accuracy for range = 1: 95.70%

Range = 10
Accuracy for range = 10: 99.34%

Range = 100
Accuracy for range = 100: 99.48%
```

Question 3: Kernel-based density estimation

[118]:
```
def gaussian_kernel(x, mean, sigma):
    return (1 / (sigma * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x - mean) /␣
↪sigma) ** 2)

def kd_classify(x, mean1, mean2, sigma):
    # calculate probability densities for both classes
    c1_density = gaussian_kernel(x, mean1, sigma)
    c2_density = gaussian_kernel(x, mean2, sigma)

    # compare densities to classify
```

```
    if c1_density > c2_density:
        return 0
    else:
        return 1
```

[119]:
```
c1 = filtered_train_images[filtered_train_labels == 0]
c2 = filtered_train_images[filtered_train_labels == 1]

c1_mean = np.mean(c1)
c2_mean = np.mean(c2)
sigma = 20

predicted_labels = []
for sample in filtered_test_images:
    predicted_labels.append(kd_classify(sample, c1_mean, c2_mean, sigma))

# calculate classification accuracy
accuracy = np.mean(filtered_test_labels.numpy() == predicted_labels)
print(f"Accuracy = {accuracy * 100:.2f}%")
```

```
Accuracy = 95.46%
```

Question 4 Histogram-based estimation appears to perform the best, yielding extremely high accuracies for bin ranges of 10 and 100 (>99%). Even the bin range of 1 is greater than the kernel-based density estimation of 95.46%. However, this is likely a sign that those models are overfit. The accuracy of all histogram-based estimation is highly sensitive to bin width, which makes it less stable and less generalizable. On the other hand, kernel-based density estimation does not depend on specifying bin width, is thus more consistent, and is more capable of capturing complex data distributions. Therefore, although the histogram-based estimatin yielded higher accuracies, kernel-based density estimation is likely the more appropriate choice given its relatively high accuracy, robustness, and capacity for generalization.

Question 5 Parametric estimation methods make assumptions about the underlying data distribution, unlike non-parametric meethods. Non-parametric methods offer flexibility in modelling complex and potentially unknown data distributions. They are more suitable when you have limited prior knowledge about the data, or when the true data distribution is not represented well by common parametric models, such as Gaussian. Parametric methods are suitable when you do have prior knowledge about the data, or are quite confident the data follows a specific distribution, such as Gaussian. As such, I would expect that parametric methods would not perform as well on the MNIST dataset as non-parametric methods, because the dataset is high-dimensional and the distribution of pixels for the images is likely quite complex.

0.0.2 Exercise 2: K-means Clustering

[5]:
```
# initializing training set
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=None)
mnist_train_loader = torch.utils.data.DataLoader(mnist_trainset, batch_size=64,␣
 ↪shuffle=True)

# initializing test set
mnist_testset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=None)
mnist_test_loader = torch.utils.data.DataLoader(mnist_testset, batch_size=64,␣
 ↪shuffle=True)

# flatten the training set
flattened_train_images = mnist_trainset.data.view(mnist_trainset.data.size(0),␣
 ↪-1)
flattened_train_labels = mnist_trainset.targets

# flatten the test set
flattened_test_images = mnist_testset.data.view(mnist_testset.data.size(0), -1)
flattened_test_labels = mnist_testset.targets

# rename for clarity
train_data = flattened_train_images.numpy()
train_labels = flattened_train_labels.numpy()
test_data = flattened_test_images.numpy()
test_labels = flattened_test_labels.numpy()

# verify the shape of flattened images for the training set
print("Shape of flattened training images:", train_data.shape)

# verify the shape of flattened images for the test set
print("Shape of flattened test images:", test_data.shape)
```

```
Shape of flattened training images: (60000, 784)
Shape of flattened test images: (10000, 784)
```

Question 1: K-means algorithm implementation

[6]:
```
class Kmeans:
    def __init__(self, x, k):
        self.x = x
        self.k = k
        self.prototypes = None

    def fit(self, tolerance=1e-4, max_iterations=100):
        # random initialization of prototypes
```

```
        np.random.seed(42)
        indices = np.random.choice(self.x.shape[0], self.k, replace=False)
        self.prototypes = self.x[indices]

        # repeat until convergence
        for _ in range(max_iterations):
            labels = []
            for sample in self.x:
                # calculate euclidean distances between sample and all prototypes
                distances = []
                for prototype in self.prototypes:
                    distances.append(np.linalg.norm(prototype - sample))

                # cluster label is index of minimum sample-prototype distance
                cluster = distances.index(min(distances))
                labels.append(cluster)

            # initializing empty list of cluster means
            cluster_means = []

            # iterating through clusters
            for cluster in range(self.k):
                # find all indices where label == cluster
                indices = [idx for idx, label in enumerate(labels) if label ==␣
↪cluster]

                # calculate mean of cluster
                cluster_mean = np.mean(self.x[indices], axis=0, keepdims=True)
                cluster_means.append(cluster_mean)

            # check for convergence
            new_prototypes = np.vstack(cluster_means)
            if np.all(np.isclose(new_prototypes, self.prototypes,␣
↪rtol=tolerance)):
                break # converged

            # set cluster means as new prototypes
            self.prototypes = cluster_means

    def predict(self, test_data):
        labels = []
        for sample in test_data:
            # calculate euclidean distances between sample and all prototypes
            distances = []
            for prototype in self.prototypes:
                distances.append(np.linalg.norm(prototype - sample))
```

```
            # cluster label is index of minimum sample-prototype distance
            cluster = distances.index(min(distances))
            labels.append(cluster)

        return labels
```

Question 2: Applying k-means to MNIST dataset

[21]:
```
import time

k_values = [5, 10, 20, 40]
cluster_labels = {'5': [], '10': [], '20': [], '40': []}

for k in k_values:
    print(f"k = {k}")
    start = time.time()

    kmeans = Kmeans(train_data, k)
    kmeans.fit()
    cluster_labels[str(k)] = kmeans.predict(test_data)

    end = time.time()
    print(f"Elapsed time: {(end - start):.3f}s\n")
```

```
k = 5
Elapsed time: 232.776s

k = 10
Elapsed time: 452.435s

k = 20
Elapsed time: 1085.146s

k = 40
Elapsed time: 1752.498s
```

Question 3: Calculating cluster consistency for each k

[24]:
```
consistency_scores = []

for k in k_values:
    print(f"k = {k}")
    Q_values = []

    # iterating through each cluster
    for i in range(k):
        cluster_indices = np.where(np.array(cluster_labels[str(k)]) == i)[0]
        cluster_data = test_labels[cluster_indices]
```

```
        class_counts = np.bincount(cluster_data)
        max_count = class_counts.max()
        Q = max_count / len(cluster_data)
        Q_values.append(Q)

    Q_total = np.mean(Q_values)
    print(f"Q_total = {Q_total}\n")
    consistency_scores.append(Q_total)
```

```
k = 5
Q_total = 0.38997367741130895

k = 10
Q_total = 0.6135704055555864

k = 20
Q_total = 0.7373298685377399

k = 40
Q_total = 0.7957417116388881
```

Question 4 From the results above, it's clear that cluster consistency gets better as we increase k. There is a significant improvement from k = 5 to k = 10 (Q increases by ~0.2), and continuing to double k results in slower improvement of Q. This suggests that one should maximize k to maximize Q, but that would be misleading. After a certain point, increasing the compactness of separation of clusters will start to capture variation that is not useful and can result in non-meaningful clusters. Intuitively, I would expect k = 10 to be the most meaningful, as the MNIST dataset comprises 10 digits that we would naturally cluster as 1 digit per cluster.

0.0.3 Exercise 3: Gaussian Mixture Model (GMM)

Question 1: Derivation and implementation of GMM algorithm

[49]:
```
import numpy as np
import torch
import torchvision
import torchvision.datasets as datasets
```

[50]:
```
class GMM:
    def __init__(self, X, K):
        self.X = X
        self.K = K
        self.pi = None
        self.mu = None
        self.sigma = None
```

```
    def initialize_parameters(self):
        n, d = self.X.shape

        self.pi = np.ones(self.K) / self.K  # initialize mixture weights␣
↪uniformly
        self.mu = self.X[np.random.choice(n, self.K, replace=False)]  # randomly␣
↪initialize means
        self.sigma = [np.ones(d) for _ in range(self.K)]  # initialize diagonal␣
↪covariance vectors with ones

        print("parameters initialized.")


    def calculate_responsibility(self, x, k):
        """
        Calculate the responsibility for a data point x and a Gaussian component␣
↪k in a GMM

        Parameters:
        - x: Data point (a 1D NumPy array)
        - k: Index of the Gaussian component
        - pi: Mixture weights for all components (kx1)
        - mu: Means for all components (kxd)
        - sigma: Covariance matrices for all components (kx(dx1) --> list of 1D␣
↪vectors)
            - NOTE: sigma is a diagonal matrix and is being represented by a 1D␣
↪vector of diagonal elements

        Returns:
        - r_k: Responsibility for x and component k
        """
        d = len(x)  # dimension of data

        # calculate the likelihood of x under the Gaussian component k
        diff = x - self.mu[k]  # difference between x and kth mean
        sigma_inv = 1.0 / self.sigma[k]  # inverse of kth diagonal cov matrix
        exponent = -0.5 * (diff.T @ np.diag(sigma_inv) @ diff)
        likelihood = np.exp(exponent) / np.sqrt(np.prod(self.sigma[k]))  #␣
↪det(diag) = product of diagonal elements

        # calculate the numerator of the responsibility formula
        numerator = self.pi[k] * likelihood

        # calculate the denominator of the responsibility formula
        denominator = 0.0
```

```
        for j in range(len(self.pi)):
            diff_j = x - self.mu[j]  # difference between x and kth mean
            sigma_inv_j = 1.0 / self.sigma[j]  # inverse of kth diagonal cov␣
↪matrix
            exponent_j = -0.5 * (diff_j.T @ np.diag(sigma_inv_j) @ diff_j)
            likelihood_j = np.exp(exponent_j) / np.sqrt(np.prod(self.sigma[j])) ␣
↪# det(diag) = product of diagonal elements
            denominator += self.pi[j] * likelihood_j

        # Calculate the responsibility
        r_k = numerator / denominator

        return r_k


    def calculate_neg_log_likelihood(self):
        n, d = self.X.shape

        log_likelihood = 0.0
        for i in range(n): # summing over n data points

            likelihood = 0.0 # likelihood of x under the Gaussian component k
            for k in range(K): # summing over K

                diff = self.X[i] - self.mu[k]  # difference between x and kth␣
↪mean
                sigma_inv = 1.0 / self.sigma[k]  # inverse of kth diagonal cov␣
↪matrix
                exponent = -0.5 * (diff.T @ np.diag(sigma_inv) @ diff)
                likelihood += self.pi[k] * (2*np.pi**(-d/2.0)) * np.prod(self.
↪sigma[k])**(-0.5) * np.exp(exponent)

            log_likelihood += np.log(likelihood)

        neg_log_likelihood = -log_likelihood
        return neg_log_likelihood


    def train(self, max_iter=500, tol=10e-5):
        """
        Calculate the parameters (pi, mu, sigma) for k Gaussian components of a␣
↪GMM

        Parameters:
        - X: Data of ONE CLASS
        - K: Number of Gaussian components (clusters)
```

```
        - pi: Mixture weights for all components (kx1)
        - mu: Means for all components (kxd)
        - sigma: Covariance matrices for all components (kx(dx1) --> list of 1D␣
↪vectors)
            - NOTE: sigma is a diagonal matrix and is being represented by a 1D␣
↪vector of diagonal elements

        Returns estimated parameters for all components + log likelihood?

        """
        n, d = self.X.shape  # number of data points and dimensions

        # repeat for max iterations or until negative log-likelihood diff <␣
↪desired tolerance
        for iteration in range(max_iter):
            print(f"iteration = {iteration}")
            # E-step: Calculate responsibilities

            # empty matrix to store responsibility of X[i] for each k
            r_ik = np.zeros((n, self.K))
            for i in range(n):
                print(f"i = {i}")
                for k in range(self.K):
                    r_ik[i, k] = self.calculate_responsibility(self.X[i], k)

            # calculate negative log-likelihood
            # log_likelihood = np.sum(np.log(np.sum(r_ik, axis=1)))
            neg_log_likelihood = self.calculate_neg_log_likelihood()
            print(f"negative log likelihood = {neg_log_likelihood}")

            # does not compare to tolerance on first iteration (no previous␣
↪log-likelihood)
            if iteration > 0 and np.abs(neg_log_likelihood -␣
↪prev_log_likelihood) < tol * np.abs(log_likelihood):
                break

            # M-step: Update parameters
            r_ik_sum = np.sum(r_ik, axis=0)
            self.pi = r_ik_sum / n
            self.mu = (r_ik.T @ self.X) / r_ik_sum[:, np.newaxis]

            sum_change = 0.0
            for i in range(n):
                for k in range(self.K):
                    diff = self.X - self.mu[k] # this isn't gonna work
                    self.sigma[k] += np.sum((r_ik[:, k] * (diff ** 2)), axis=0)
```

```
            prev_log_likelihood = neg_log_likelihood
```

Question 2: Applying GMM to MNIST dataset

- for EACH CLASS (digit), can fit a GMM with k = 5 corresponding to 5 styles of writing this digit
- x is an image and y is the class (digit)
- `Pr(Y = c)` is the proportion of the cth class in the training set
- density `p(X = x | Y = c)` is **estimated using GMM** for each class c separately –> we need to get the estimation distribution of each class
- the estimate y(x) is the MAXIMUM LIKELIHOOD THAT the image belongs to class c, in other words we have to **calculate it for each class** and then the **highest probability becomes the class prediction**

[51]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# initializing training set
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=None)
mnist_train_loader = torch.utils.data.DataLoader(mnist_trainset, batch_size=64,␣
 ↪shuffle=True)

# initializing test set
mnist_testset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=None)
mnist_test_loader = torch.utils.data.DataLoader(mnist_testset, batch_size=64,␣
 ↪shuffle=True)

# flatten the training set
flattened_train_images = mnist_trainset.data.view(mnist_trainset.data.size(0),␣
 ↪-1)
flattened_train_labels = mnist_trainset.targets

# flatten the test set
flattened_test_images = mnist_testset.data.view(mnist_testset.data.size(0), -1)
flattened_test_labels = mnist_testset.targets

# rename for clarity
train_data = flattened_train_images.numpy()
train_labels = flattened_train_labels.numpy()
test_data = flattened_test_images.numpy()
test_labels = flattened_test_labels.numpy()

# standardize data
scaler = StandardScaler()
train_data = scaler.fit_transform(train_data)
```

```
test_data = scaler.fit_transform(test_data)

# PCA
pca = PCA(n_components=200)
pca.fit(train_data)
train_data = pca.transform(train_data)
test_data = pca.transform(test_data)

# verify the shape of flattened images for the training set
print("Shape of flattened training images:", train_data.shape)

# verify the shape of flattened images for the test set
print("Shape of flattened test images:", test_data.shape)
```

```
Shape of flattened training images: (60000, 200)
Shape of flattened test images: (10000, 200)
```

[52]:
```
classes = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
class_data = {'0': [], '1': [], '2': [], '3': [], '4': [],
              '5': [], '6': [], '7': [], '8': [], '9': []}

# separating training data by class
for c in classes:
    class_indices = np.where(train_labels == c)[0]
    class_data[str(c)] = train_data[class_indices][:100,:] # first 100 points?

# dict of GMMs for each class
class_GMMs = {'0': [], '1': [], '2': [], '3': [], '4': [],
              '5': [], '6': [], '7': [], '8': [], '9': []}
```

[53]:
```
print(class_data['0'].shape)
```

```
(100, 200)
```

[54]:
```
import time

K = 5
start = time.time()

# for each class, train k = 5 GMMs
for c in classes:
    class_GMM = GMM(class_data[str(c)], K)
    class_GMM.initialize_parameters()
    class_GMM.train()

    # add to dict
    class_GMMs[str(c)] = class_GMM
```

```
    break

end = time.time()
print(f"Elapsed time: {(end - start):.3f}s\n")
```

```
parameters initialized.
iteration = 0
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
i = 11
i = 12
i = 13
i = 14
i = 15
i = 16
i = 17
i = 18
i = 19
i = 20
i = 21
i = 22
i = 23
i = 24
i = 25
i = 26
i = 27
i = 28
i = 29
i = 30
i = 31
i = 32
i = 33
i = 34
i = 35
i = 36
i = 37
i = 38
i = 39
i = 40
```

```
i = 41
i = 42
i = 43
i = 44
i = 45
i = 46
i = 47
i = 48
i = 49
i = 50
i = 51
i = 52
i = 53
i = 54
i = 55
i = 56
i = 57
i = 58
i = 59
i = 60
i = 61
i = 62
i = 63
i = 64
i = 65
i = 66
i = 67
i = 68
i = 69
i = 70
i = 71
i = 72
i = 73
i = 74
i = 75
i = 76
i = 77
i = 78
i = 79
i = 80
i = 81
i = 82
i = 83
i = 84
i = 85
i = 86
i = 87
i = 88
```

```
i = 89
i = 90
i = 91
i = 92
i = 93
i = 94
i = 95
i = 96
i = 97
i = 98
i = 99
negative log likelihood = inf
```

```
<ipython-input-50-5d5f66d7834b>:56: RuntimeWarning: invalid value encountered in
double_scalars
  r_k = numerator / denominator
<ipython-input-50-5d5f66d7834b>:75: RuntimeWarning: divide by zero encountered
in log
  log_likelihood += np.log(likelihood)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-54-da4525862a96> in <module>
      8     class_GMM = GMM(class_data[str(c)], K)
      9     class_GMM.initialize_parameters()
---> 10     class_GMM.train()
     11
     12     # add to dict

<ipython-input-50-5d5f66d7834b> in train(self, max_iter, tol)
    126                 for k in range(self.K):
    127                     diff = self.X - self.mu[k] # this isn't gonna work
--> 128                     self.sigma[k] += np.sum((r_ik[:, k] * (diff ** 2)),␣
 ↪axis=0)
    129
    130             prev_log_likelihood = neg_log_likelihood

ValueError: operands could not be broadcast together with shapes (100,) (100,200)
```

[ ]: