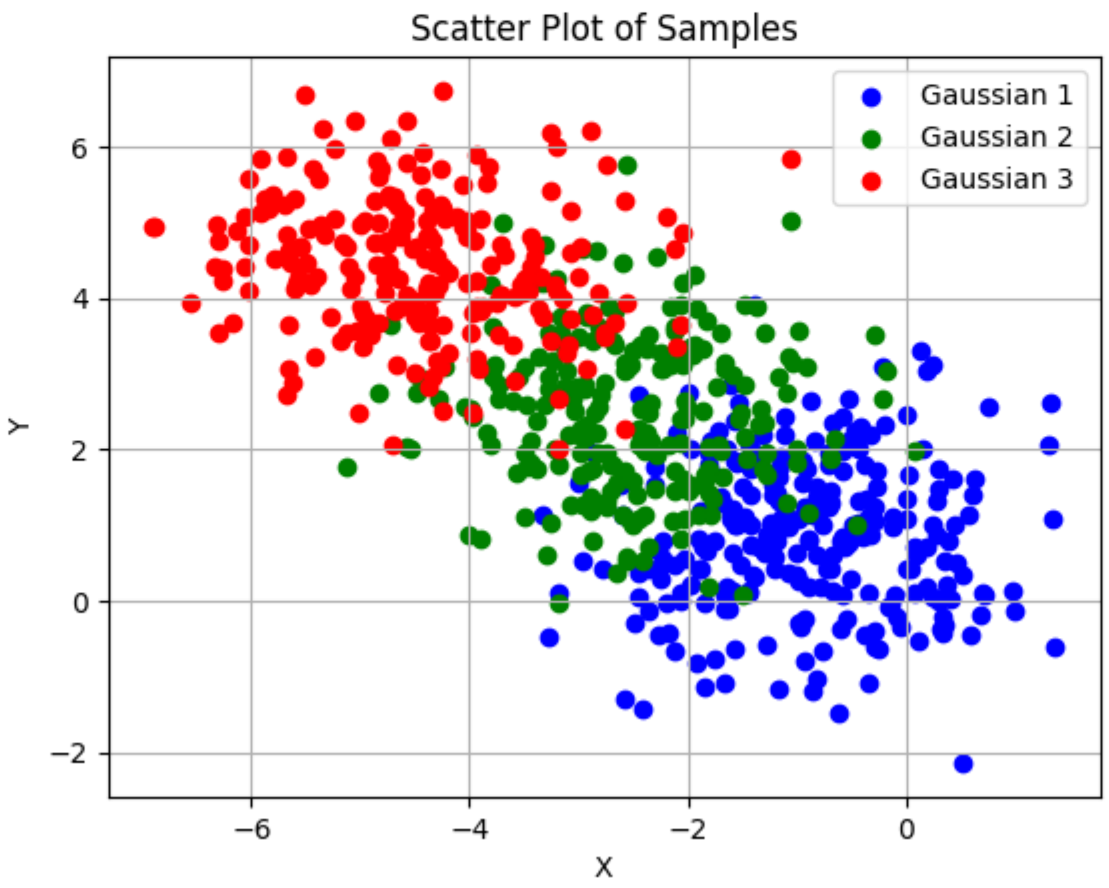
Submitted By: Omer Lugasi 322836180 & Yuval Angel 212536924

Question 1

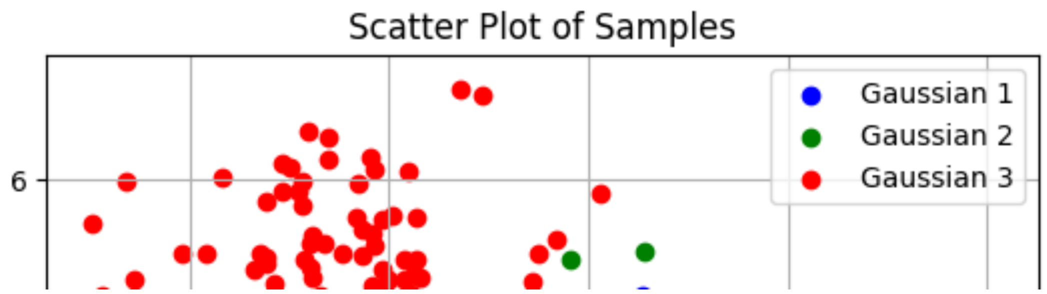
Sections 1+2

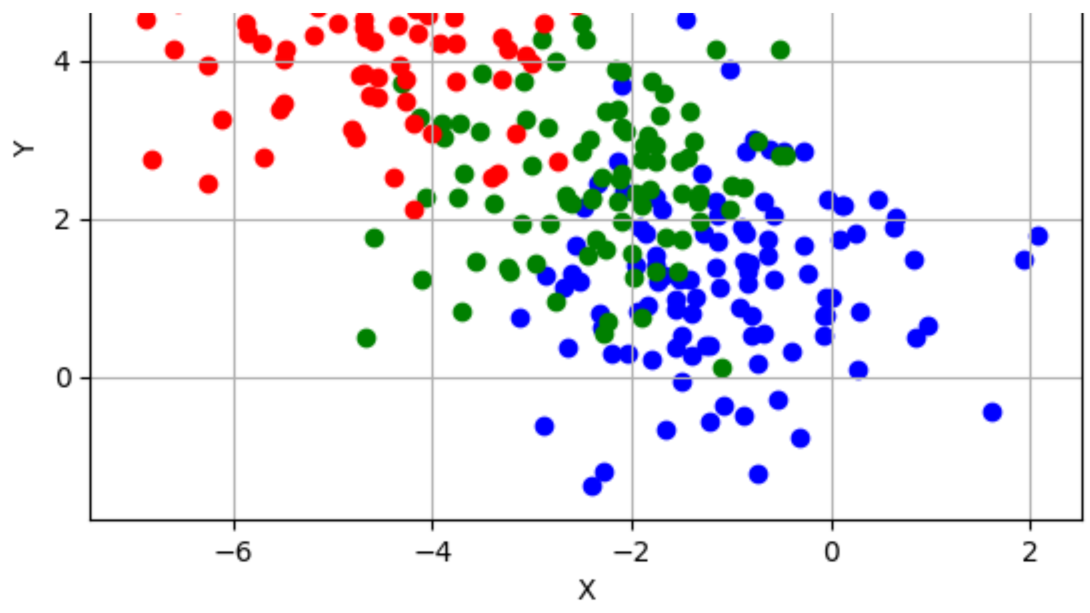
```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt


def generate_samples(mu1, mu2, mu3, n):
    # Generate n total samples of 3 given gaussian distributions
    mu_list = list([mu1, mu2, mu3])
    sigma = np.identity(mu1.shape[0])
    # Every index will contain matrix of samples from index's mu params
    samples = list()
    # Generate number of samples for each distribution:
    sizes = np.random.multinomial(n, [1/3, 1/3, 1/3])
    for i in range(3):
        size = sizes[i]
        vec = np.zeros((size, 2))
        for k in range(size):
            sample = np.random.multivariate_normal(mu_list[i], sigma)
            vec[k] = sample
        samples.append(vec)
    return samples


def plot_samples(samples):
    colors = ['b', 'g', 'r']

    for i in range(3):
        points = samples[i]
        color = colors[i]
        label = 'Gaussian {}'.format(i+1)
        plt.scatter(points[:, 0], points[:, 1], c=color, label=label)

    plt.legend()
    plt.title("Scatter Plot of Samples".format(n))
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.grid(True)
    plt.show()


mu1 = np.array([-1, 1]).T
mu2 = np.array([-2.5, 2.5]).T
mu3 = np.array([-4.5, 4.5]).T
train_size = 700
train_samples = generate_samples(mu1, mu2, mu3, train_size)
plot_samples(train_samples)
```



Section 3

```
test_size = 300
test_samples = generate_samples(mu1, mu2, mu3, test_size)
plot_samples(test_samples)
```





Section 4

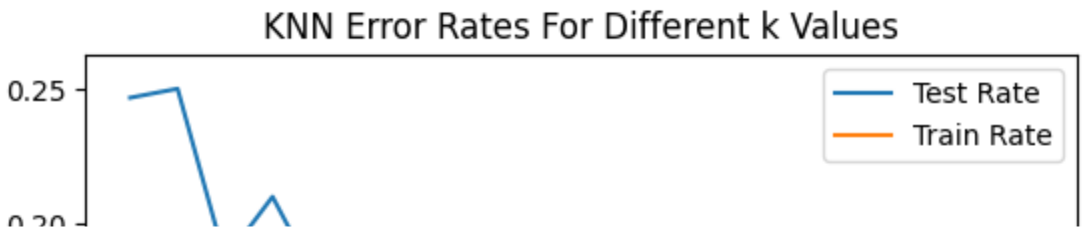
```
def setup_X_y(samples_list):
  X = []
  y = []
  for distribution, samples_matrix in enumerate(samples_list):
    for sample in samples_matrix:
      X.append(sample)
      y.append(distribution)
  return X, y


def run_model_with_k(train_samples, test_samples, k):
  knn = KNeighborsClassifier(k)

  X_train, y_train = setup_X_y(train_samples)
  X_test, y_test = setup_X_y(test_samples)

  knn.fit(X_train, y_train)

  y_pred = knn.predict(X_test)

  return y_test, y_pred


def calc_error(true, predicted):
  n = len(true)
  return 1 / n * np.sum(true != predicted)


def get_rates(train_samples, test_samples, k):
  y_test, y_pred = run_model_with_k(train_samples, test_samples, k)
  test_rate = calc_error(y_test, y_pred)
  y_train, y_train_pred = run_model_with_k(train_samples, train_samples, k)
  train_rate = calc_error(y_train, y_train_pred)

  return test_rate, train_rate


test_rate, train_rate = get_rates(train_samples, test_samples, 1)
print(test_rate, train_rate)
```

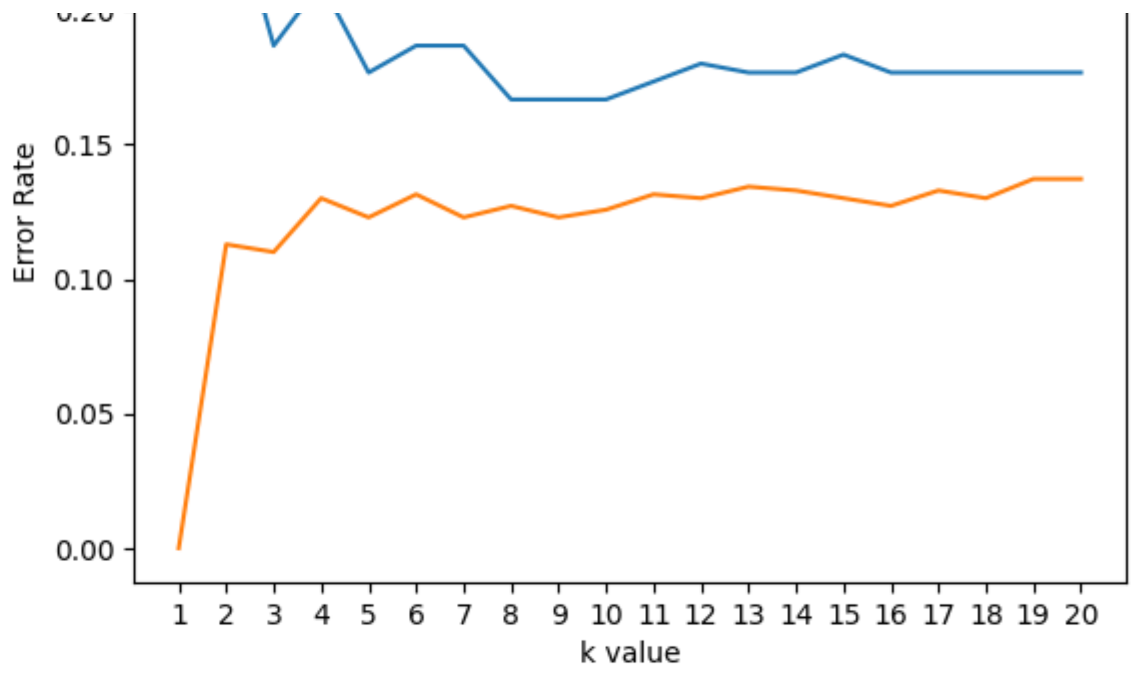
```
     0.24666666666666667 0.0
```

The train error rate is 0%, while the test error rate is ~20%. As we've learned in class, setting k=1 makes the model overfit to the train data and underperform on the test set, so these results are expected.

Section 5

```
def plot_rates(changing_values, test_rates, train_rates, param):
  plt.plot(changing_values, test_rates, label='Test Rate')
  plt.plot(changing_values, train_rates, label='Train Rate')

  plt.xlabel('{} value'.format(param))
  plt.xticks(changing_values)
  plt.ylabel('Error Rate')

  plt.title('KNN Error Rates For Different {} Values'.format(param))
  plt.legend()
  plt.show()


test_rates = []
train_rates = []
k_values = range(1, 21)

for k in k_values:
  test_rate, train_rate = get_rates(train_samples, test_samples, k)
  test_rates.append(test_rate)
  train_rates.append(train_rate)

plot_rates(k_values, test_rates, train_rates, 'k')
```



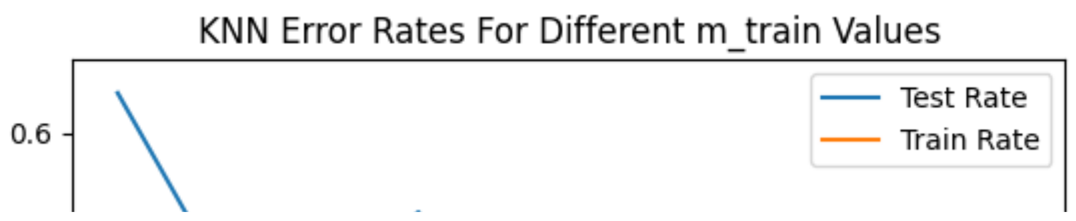


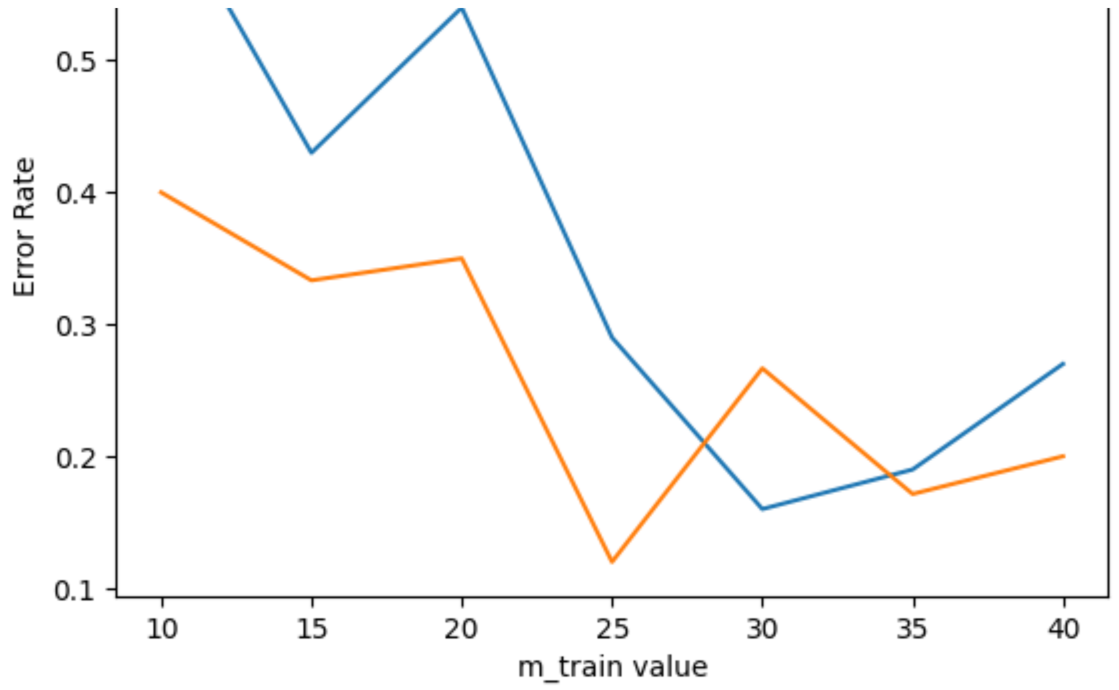
The test error does decrease with k initially, but then it starts increasing again after a certain threshold. This is expected, as if we take a very extreme example of setting k to be the entire train set, then the model just selects a majority label, which can be very inaccurate.

As we can see, even without setting k to be so big, there's an increase in errors between k=3 and k=4.

Section 6

```
m_train_values = [10 + i * 5 for i in range(7)]
m_test = 100
k = 10
new_train_rates = []
new_test_rates = []
for m_train in m_train_values:
  new_train_samples = generate_samples(mu1, mu2, mu3, m_train)
  new_test_samples = generate_samples(mu1, mu2, mu3, m_test)
  new_test_rate, new_train_rate = get_rates(new_train_samples, new_test_samples, k)
  new_test_rates.append(new_test_rate)
  new_train_rates.append(new_train_rate)

plot_rates(m_train_values, new_test_rates, new_train_rates, 'm_train')
```





as the size of the training set increases, we see both rates become much better. We would expect that as the training data's size increases, the model would become better. This graph shows something unexpected - there are jumps in the error rates, meaning that in some instances, the model becomes worse by increasing the training set's size.

Section 7

After running the code in section 6 multiple times, the graph changes a lot between each trial run. Sometimes, it shows what we would expect - a descending graph that improves as m increases. Sometimes, it's unexpected - the spikes in error rates still show up. The reason for this large variance between trials might be that even at m_train = 40, the training set is still too small and prone to changes.

Section 8

We can implement a weighted k-NN model, that scales with distance between nodes. We'll use the following Formula seen in the tutorial:

$$h_s(x) = \sum_{i=1}^{k} \frac{\omega\left(x, x'_{\pi_i(x)}\right)}{\sum_{j=1}^{k} \omega\left(x, x'_{\pi_j(x)}\right)} y_{\pi_i(x)}$$

$h_s(x)$ would be the metric to select label instead of euclidean distance, where $\omega = \frac{1}{\rho}$ as required, and we can choose ρ to be euclidean distance for example. In case of a tie we can choose randomly.

Question 3

Section 1

Let $w_1, w_2 \in \mathbb{R}^d$.

$$P_{w_1}(Y_i = 0|x_i) = \frac{e^{w_1^T x_i}}{e^{w_1^T x_i} + e^{w_2^T x_i}} = p_1$$

$$P_{w_2}(Y_i = 1|x_i) = \frac{e^{w_2^T x_i}}{e^{w_1^T x_i} + e^{w_2^T x_i}} = p_2$$

In the binary case, every point is either of class 0 or class 1. Thus: $p_1 + p_2 = 1$.

Choose $w = w_1 - w_2$. We get:

$$p_2 = \frac{e^{w_2^T x_i}}{e^{w_1^T x_i} + e^{w_2^T x_i}} = \frac{e^{w_2^T x_i}}{e^{w_2^T x_i}(1 + e^{w_1^T x_i - w_2^T x_i})} = \frac{1}{1 + e^{w^T x_i}} = 1 - \frac{e^{w^T x_i}}{1 + e^{w^T x_i}}$$

Denote $p = \frac{e^{w^T x_i}}{1 + e^{w^T x_i}}$. We get the required $p_1 = p, p_2 = 1 - p$.

Converse:

Let $w \in \mathbb{R}^d, w_1 = w_2 + w$ like before.

$$p = \frac{e^{w^T x_i}}{1 + e^{w^T x_i}} = \frac{e^{(w_1 - w_2)^T x_i}}{1 + e^{(w_1 - w_2)^T x_i}} = \frac{\frac{e^{w_1^T x_i}}{e^{w_2^T x_i}}}{\frac{e^{w_2^T x_i}}{e^{w_2^T x_i}} + \frac{e^{w_1^T x_i}}{e^{w_2^T x_i}}} = \frac{e^{w_1^T x_i}}{e^{w_1^T x_i} + e^{w_2^T x_i}} = p_1$$

$$1 - p = 1 - \frac{e^{w_1^T x_i}}{e^{w_1^T x_i} + e^{w_2^T x_i}} = \frac{e^{w_2^T x_i}}{e^{w_1^T x_i} + e^{w_2^T x_i}} = p_2$$

Section 2

We'll use the p_k formulation given:

$$log(P_w(Y_i = y_i|x_i)) = log(e^{w_k^T x_i}) - log(\sum_{j=1}^K e^{w_j^T x_i}) = w_k^T x_i - log(\sum_{j=1}^K e^{w_j^T x_i})$$

Thus, we can replace the expression inside the argmax with:

$$\sum_{i=1}^m log(P_w(Y_i = y_i|x_i)) = \sum_{i=1}^m \left(w_k^T x_i - log(\sum_{j=1}^K e^{w_j^T x_i})\right)$$

Denote the log-loss function $l_s(w_j) = -log(\sum_{j=1}^{K} e^{w_j^T x_i})$. We'll differentiate the main expression by each w_i in order to find the stationary point.

$$\frac{\partial}{\partial w_k} \sum_{i=1}^{m} (w_k^T x_i + l_s(w_j)) = \sum_{i=1}^{m} \left(x_i I\{y_i = k\} - x_i \cdot \frac{e^{w_k^T x_i}}{\sum_{j=1}^{K} e^{w_j^T x_i}}\right) = 0$$

The critial point is then found when:

$$\sum_{i=1}^{m} I\{y_i = k\} = \sum_{i=1}^{m} \frac{e^{w_k^T x_i}}{\sum_{j=1}^{K} e^{w_j^T x_i}}$$

If we differentiate twice:

$$\frac{\partial^2}{\partial^2 w_k} = -\sum_{i=1}^{m} \frac{x_i^2 e^{w_k^T x_i} \cdot \left(\sum_{j=1}^{K} e^{w_j^T x_i}\right) + x_i^2 e^{2 w_k^T x_i}}{\left(\sum_{j=1}^{K} e^{w_j^T x_i}\right)^2} < 0$$

The numerators and denominators are all positive, therefore with the minus sign before the sum we have a negative value.

Thus, the critical point we found was a maximum point, and solves the optimization problem given.

Section 3

a. There are 3 weight vectors because there are 3 distinct classes each point can belong to: 0, 1 or 2.

b. Like we've seen in the lecture, we can add a value (1) to the each of the given data points to account for bias. As we've seen in the lecture, we place this point in the beginning of the vectors x and w. This is the reason that x has one less dimension than w.

c. $i = 1$:

$e^{w_1^T x_i} = e^{8\cdot1-2.5\cdot1+2\cdot8} = e^{21.5}$,

$e^{w_2^T x_i} = e^{2\cdot1+0.5\cdot1-1.5\cdot8} = e^{-9.5}$,

$e^{w_3^T x_i} = e^{-10\cdot1+2\cdot1-0.5\cdot8} = e^{-12}$

Since the denominator is the across all weights, we'll classify by largest numerator, equation 1 - which corresponds to $y = 0$.

$i = 2$:

$e^{w_1^T x_i} = e^{8\cdot1-2.5\cdot6+2\cdot-2} = e^{-11}$,

$e^{w_2^T x_i} = e^{2\cdot1+0.5\cdot6-1.5\cdot-2} = e^{8}$,

$e^{w_3^T x_i} = e^{-10\cdot1+2\cdot6-0.5\cdot-2} = e^{3}$

[illegible]

Since the number in equation 2 is the largest, we'll classify this point $y = 1$.

$i = 3:$

$e^{w_1^T x_i} = e^{8\cdot1-2.5\cdot12+2\cdot4} = e^{-14},$

$e^{w_2^T x_i} = e^{2\cdot1+0.5\cdot12-1.5\cdot4} = e^{2},$

$e^{w_3^T x_i} = e^{-10\cdot1+2\cdot12-0.5\cdot4} = e^{12}$

Since the number in equation 3 is the largest, we'll classify this point $y = 2$.