Assignment 1:

RESULT:

```
Error Rates: {1: 0.09599999999999997, 3: 0.08699999999999997, 5: 0.08399999999999996, 7:
0.08599999999999997}
Best k: 5
Confusion Matrix:
 [[ 83   0   0   0   0   0   2   0   0   0]
 [  0 126   0   0   0   0   0   0   0   0]
 [  2   4  98   1   1   0   2   6   2   0]
 [  0   1   0  98   0   2   2   2   0   2]
 [  0   2   0   0  99   0   1   1   0   7]
 [  1   1   0   0   1  81   1   0   1   1]
 [  2   0   0   0   1   0  84   0   0   0]
 [  0   6   0   0   1   1   0  88   0   2]
 [  3   1   1   4   1   3   2   0  71   3]
 [  0   0   0   0   4   0   0   1   2  88]]
```

DISCUSSION:

I've computed the kNN algorithm's performance for k=1,3,5, and 7. Here are the findings:
Error rates for different k values are as follows:
For k=1, the error rate is 9.5%.
For k=3, the error rate is 8.6%.
For k=5, the error rate is 8.3%.
For k=7, the error rate is 8.5%.
The best performance was observed for k=5 with the lowest error rate of 8.3%.

Based on the confusion matrix, we can identify which digits the kNN algorithm is most commonly confused with. Here's a brief analysis:

- 2 and 7: There are 6 instances where digit 2 was misclassified as 7. It might be because they look similar in handwritten forms.
- 4 and 9: The algorithm confused digit 4 for 9 seven times. Both digits share a similar structure in their upper halves, which can lead to confusion.
- 8 and others (3, 5, 9): The digit 8 was misclassified as 3, 5, and 9 a few times each (4, 3, and 3 times, respectively). This is likely because 8 contains elements that can resemble these digits

- 2 and 8: 2 were confused with 8 twice, and similarly, 8 was confused with 2 once. The top part of 2 and the lower loop of 8 might be visually similar in some handwritten forms, leading to these misclassifications.

One change to improve performance of the algorithm: Advanced Distance Metrics

Improvement Area: Algorithm Optimization. To make the digit recognizing algorithm work better, we can think about how we're comparing the differences or "distances" between the pictures of the digits. Right now, we use a method called Euclidean distance, which is like measuring a straight line between two points. However, there are other ways to measure how similar or different two pictures are.

One idea is to use Manhattan distance, which is more like measuring the distance between two points in a city, where you can only move along the streets in a grid, instead of cutting straight across blocks. This way of measuring can be better for some types of data because it looks at differences in a way that's more straightforward and less diagonal.

Another idea is to use Cosine similarity, which is a bit like finding out if two arrows point in the same direction, regardless of how long the arrows are. This method is great when you're dealing with a lot of data points, and you want to see if the overall direction or trend of the data is similar, rather than focusing on the exact numbers.

By trying out these different methods of measuring distances or similarities, we might find a better way to tell the digits apart, especially when the pictures of them are a bit unclear or written in different styles. This is like finding the best way to compare things to get the clearest picture of how they're related or how they're different.

Reference: I took a little bit of help from 'MachineLearningMastery.com' to understand how they implemented it. But I tried to modify the code as the pseudocode in slides.

```
#I did the code in Google Colab.

import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix, accuracy_score

train_file = '/content/mnist_train.csv'
test_file = '/content/mnist_test.csv'

def process_file(fname):
    '''assistant function for process_data that opens a CSV, converts it to numpy,
    df = pd.read_csv(fname)
    data = df.to_numpy()
    feats = data[:, 1:]  # all columns but the first
    labs = data[:, 0]  # first column only
    return feats, labs

# For atleast 6000 training points and at least 1000 test points
def process_data(train_file, test_file, train_size=6000, test_size=1000):
    '''takes in data CSVs, segments them into features and labels, and converts the
    optionally also shortens the training set and test set, if the train_size and t

    if train_size > 60000:
        train_size = 60000
    if test_size > 10000:
        test_size = 10000

    train_feats, train_labs = process_file(train_file)
    test_feats, test_labs = process_file(test_file)

    return train_feats[:train_size], train_labs[:train_size], test_feats[:test_size

def euclidean_distance(v1, v2):
    '''calculates the Euclidean distance between vectors v1 and v2'''
    return np.linalg.norm(v1 - v2)

# I took a little bit of help from 'MachineLearningMastery.com' to understand how t
def knn_predict(train_feats, train_labs, test_feats, k=3):
    predictions = []
    for test_vector in test_feats:
        distances = np.array([euclidean_distance(test_vector, train_vector) for tra
        distance_label_pairs = list(zip(train_labs, distances))
        sorted_pairs = sorted(distance_label_pairs, key=lambda pair: pair[1])
        nearest = [pair[0] for pair in sorted_pairs[:k]]
        nearest = [pair[0] for pair in sorted_pairs[:k]]
        var = np.bincount(nearest).argmax()
        predictions.append(var)
    return predictions

# Testing for different k values
k_values = [1, 3, 5, 7]
error_rates = {}
train_feats, train_labs, test_feats, test_labs = process_data(train_file, test_file

for k in k_values:
    predictions = knn_predict(train_feats, train_labs, test_feats, k)
    accuracy = accuracy_score(test_labs, predictions)
    error_rates[k] = 1 - accuracy

# Best k and its confusion matrix
best_k = min(error_rates, key=error_rates.get)
best_predictions = knn_predict(train_feats, train_labs, test_feats, best_k)
conf_matrix = confusion_matrix(test_labs, best_predictions)

print("Error Rates:", error_rates)
print("Best k:", best_k)
print("Confusion Matrix:\n", conf_matrix)
```

```
Error Rates: {1: 0.09599999999999997, 3: 0.08699999999999997, 5: 0.0839999999
Best k: 5
Confusion Matrix:
 [[ 83   0   0   0   0   0   2   0   0   0]
 [  0 126   0   0   0   0   0   0   0   0]
 [  2   4  98   1   1   0   2   6   2   0]
 [  0   1   0  98   0   2   2   2   0   2]
 [  0   2   0   0  99   0   1   1   0   7]
 [  1   1   0   0   1  81   1   0   1   1]
 [  2   0   0   0   1   0  84   0   0   0]
 [  0   6   0   0   1   1   0  88   0   2]
 [  3   1   1   4   1   3   2   0  71   3]
 [  0   0   0   0   4   0   0   1   2  88]]
```