**Problem One**

The code used for this problem can be found in understanding\_data.py. It can be run using the following command:

*python understanding\_data.py*

**A.** I examined 50 images of the digit 8. I chose this digit because it is a relatively intricate number with two stacked loops, and can be misinterpreted as the digit 3, if written incompletely, or the digit 0 if written too thickly. With this intuition, I examined the set of images, and identified the following pictures to be potential difficult cases.



The above three figures demonstrate a key challenge with classifying the digit 8. When written messily, often times either one or two of the loops is not fully connected. Because the fundamental characteristic of the letter 8 is two closed loops, the digit becomes a suddenly very complex image of curling lines. One could easily misinterpret an unclosed digit 8 with a messy digit 3, which likely has a complementary issue of accidentally closing loops.



The above three figures demonstrate another key challenge when classifying the digit 8. When written extremely thickly, the loops become hard to identify. In turn, the classifier is likely to struggle when discerning a thick blob, and may assume the bottom loop is a thick line (far left), or perhaps a messy 0. With that, it is also challenging when handling finely written 8s (far right). If one loop is larger and significantly clearer, the classifier may only capture it and the thin stroke would make it increasingly difficult for the classifier to even recognize the smaller, messier loop, leading to misclassifications.

**B.**

The total number of images for digit 0: 5923

The total number of images for digit 1: 6742

The total number of images for digit 2: 5958

The total number of images for digit 3: 6131

The total number of images for digit 4: 5842

The total number of images for digit 5: 5421

The total number of images for digit 6: 5918

The total number of images for digit 7: 6265

The total number of images for digit 8: 5851

The total number of images for digit 9: 5949

The total number of images: 60000

When creating the training and testing sets during our experiments, we ensured that 1/10th of each set each came from each digit. This ensures that each digit is properly represented in both the training and testing set. Through this methodology, the classifier is trained for all digits within the testing set, so that our evaluations test how it classifies unseen data within the same realm. We split data like this such that a case never arises where the classifier has a training set of all 3’s, and is tested with the number 9, or something similar.

Moreover, we chose this distribution such that the model is exposed to a substantial number of images for each digit during training. As shown above, there is a large variation in handwriting for each given digit. As such, it is important the classifier sees a breadth of data during classification, so it can further refine its interpretation of each digit and understand the disparities.

**Problem Two**

**A.** K Nearest Neighbors is a classifier that depends on pattern recognition within a given vector space. It takes an input, and then finds k data points with feature vectors that are “nearest” to the input vector. In order to determine which data points are nearest, the system uses a distance measure to determine how closely related two vectors are, such as Euclidean distance. With this, the classifier looks at the k nearest neighbors’ labels, and ascertains the input’s label based on the majority of these neighbors’ labels. In turn, the input label is assigned basically on what the most common label is amongst its k nearest neighbors, utilizing clustering at the core of the classifier.

Support Vector Machines (SVMs) are a supervised classifier that works by mapping data to an N (or more)-dimensional space (where N is the number of features). The Support Vector Machine algorithm selects a division of the data that best divides the classes. Specifically, it finds the hyperplane that maximizes the margin to the nearest data points in each class, dubbed the “support vectors.” If the data is not linearly separable in a N-dimensional space, a kernel trick may be used to map the data to a higher dimension that it is linearly separable in. SVM can either be done with a hard-margin algorithm which requires the data to be fully linearly separable, or a soft margin algorithm that is more robust to classification error and outliers. In the soft-margin variety, misclassifications are allowed at a cost proportional to the value of the slack parameter.

To classify images, both the training set and testing set of gray scale images must be encoded such that they are a 2D array of pixel intensity values. These 2D arrays of pixel intensity values serve as each image’s feature vector, where each pixel is representative of a feature with a corresponding value.

In turn, KNN can utilize its pattern recognition algorithm to determine which sets of pixel values are most similar to the inputted set of pixel values. By leveraging Euclidean distance, KNN will determine k images that are nearest to the inputted image’s pixel intensity values. As such, taking the majority of the nearest neighbors’ labels will classify the inputted image, and essentially cast it as that majority label.

SVM, on the other hand, can find a hyperplane that linearly separates the observations possibly using a kernel trick to make the data linearly separable.

**B.** In K Nearest Neighbors, changing the hyperparamter, k, changes how many neighbors are considered during classification. Meaning, when a new data point is inputted, it finds k feature vectors that are the most similar. As such, increasing k takes more data points into account when labeling the input. Having a lower k makes the classification algorithm more susceptible to noise because fewer data points influence the label. With that, increasing k is computationally expensive, and may begin to defeat the purpose of clustering, as the number gets too high.

In the soft-margin variety of SVM, misclassifications are allowed at a cost proportional to the value of the slack parameter. It allows the optimization problem to allow misclassifications so long as the cost is below some threshold. The larger this slack parameter the more misclassification will be permitted. The lower the threshold the closer to a “hard” margin SVM it acts.

**Problem 3**

**A.** The code used for this problem can be found in classifier\_1.py. It can be run using the following command:

*python classifier\_1.py*

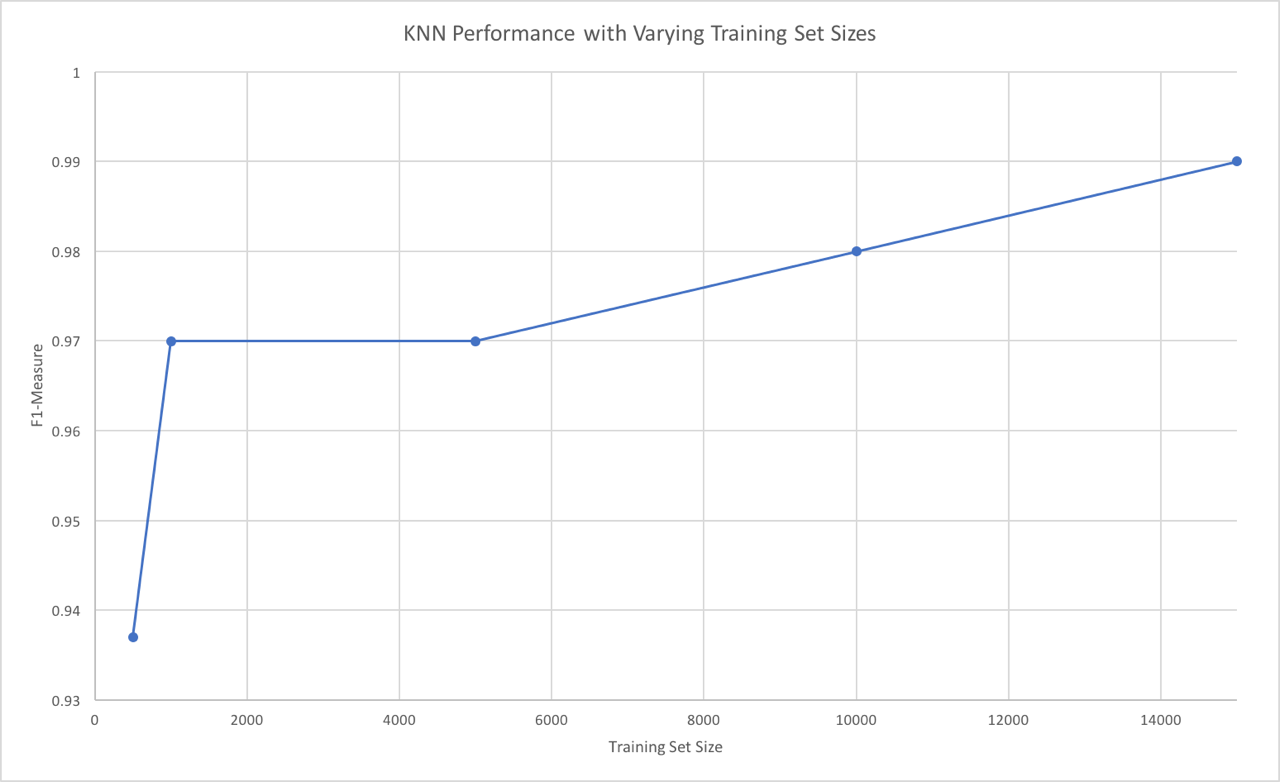
**B.**

In order to determine the best hyperparamters for KNN, we conducted two experiments. First, we looked to determine which training set size produced the best performance on the testing data. As such, we developed 5 different classifiers, using varying training set sizes: 500, 1000, 5000, 10000, and 15000. Each classifier was tested on the same testing data, so that the only variation was the training set size. With these classifiers, we found the F1-measure, and printed out the corresponding confusion matrix. Examining the F1-measures, there is a clear trend where performance increases with training data size.

In turn, our second experiment, which looked to determine the optimal number of neighbors, used a training set size of 15,000 and a testing set size of 200. In both the testing and training data sets, there is an average distribution of digits, where each digit is represented by 1/10th of the data. We trained 5 different classifiers, each utilizing a different number of neighbors within the KNN classifier: 1, 5, 10, 20, and 50. The training data and testing data were kept constant for each classifier, such that the only difference was the number of neighbors. With these classifiers, we found the F1-measure, and printed out the corresponding confusion matrix. These results are shown below.

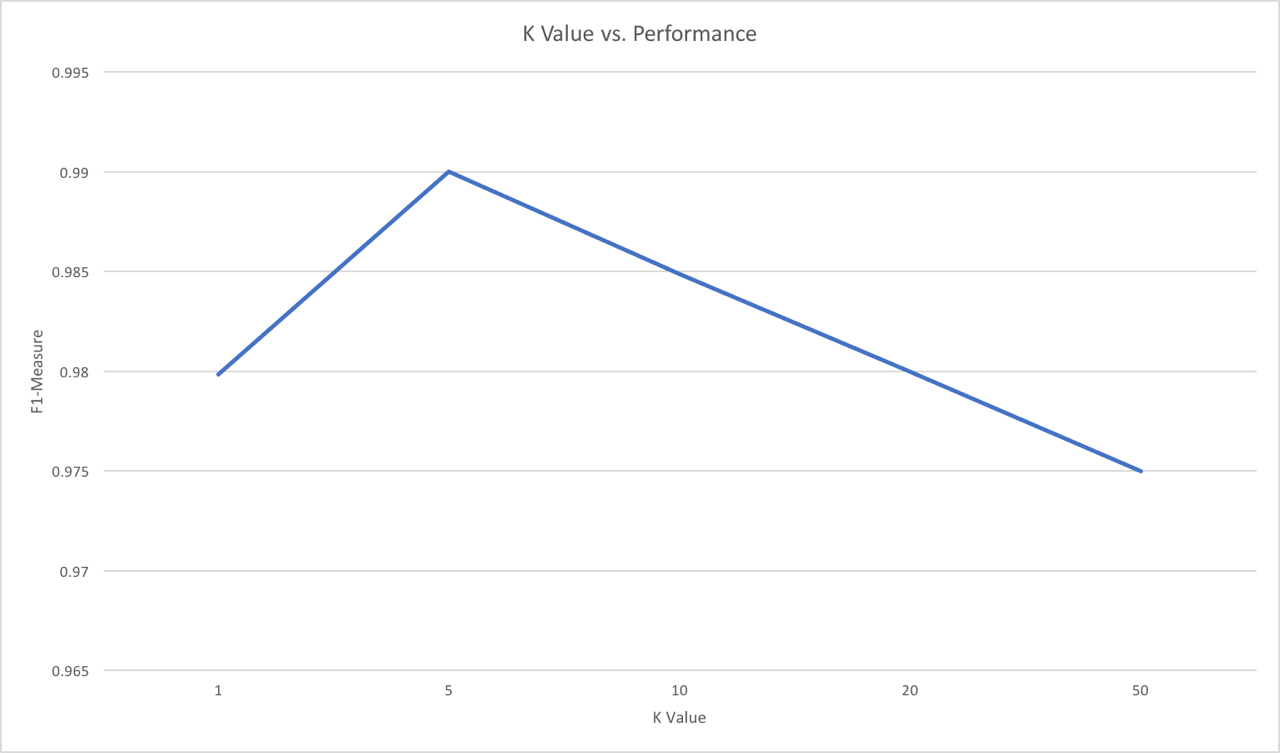
**1) KNN Training Set Size vs. Performance on Test Data**

|  |  |
| --- | --- |
| **Testing Size** | **F1-measure** |
| 500 | 0.937035376904 |
| 1000 | 0.969971810389 |
| 5000 | 0.969968683435 |
| 10000 | |  | | --- | |  | | 0.979968683435 | |  | |
| 15000 | 0.989993746091 |

****

**2) KNN Parameters**

|  |  |
| --- | --- |
| **Number of Neighbors (K)** | **F1-measure** |
| 1 | 0.97985566637 |
| 5 | 0.989993746091 |
| 10 | 0.98485566637 |
| 20 | 0.979968683435 |
| 50 | 0.974968683435 |



As shown in both the results table and graph above, k=5 produced the best performing classifier. Analyzing the numbers, it appears that increasing k improves performance to a point, but then performance dramatically dips once it reaches a certain threshold. We believe this is due to the inherent nature of clustering. Having too small of a k is suscpetible to noise, resulting in lower performance. That being said, once k is too large, the algorithm takes data into account that is beyond the immediate cluster when classifiying a given inputted data point. In turn, classification performance goes down significantly because data points beyond scope have an influence during classification. This behavior is clearly demonstrated in the results above.

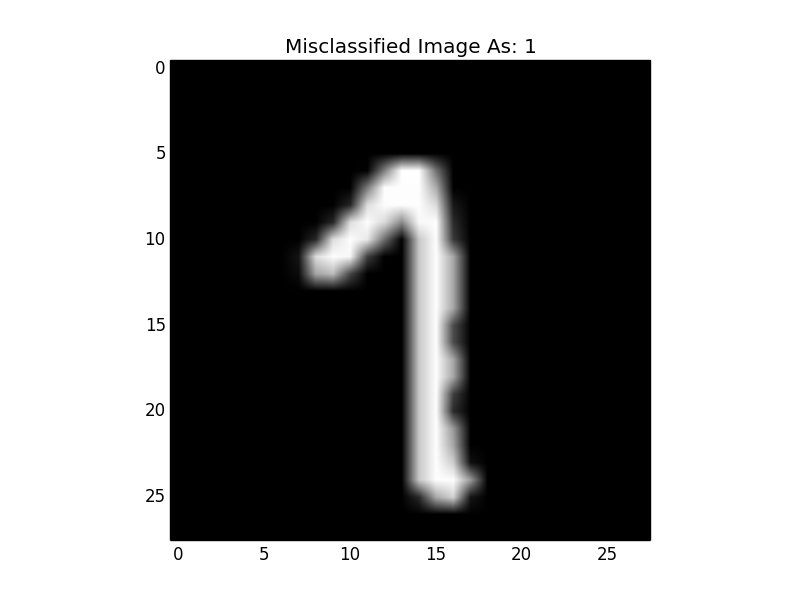
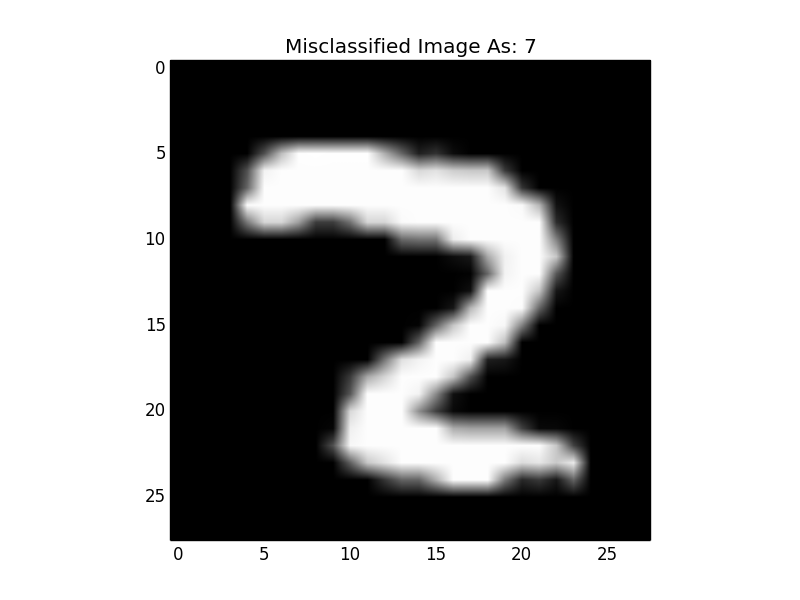
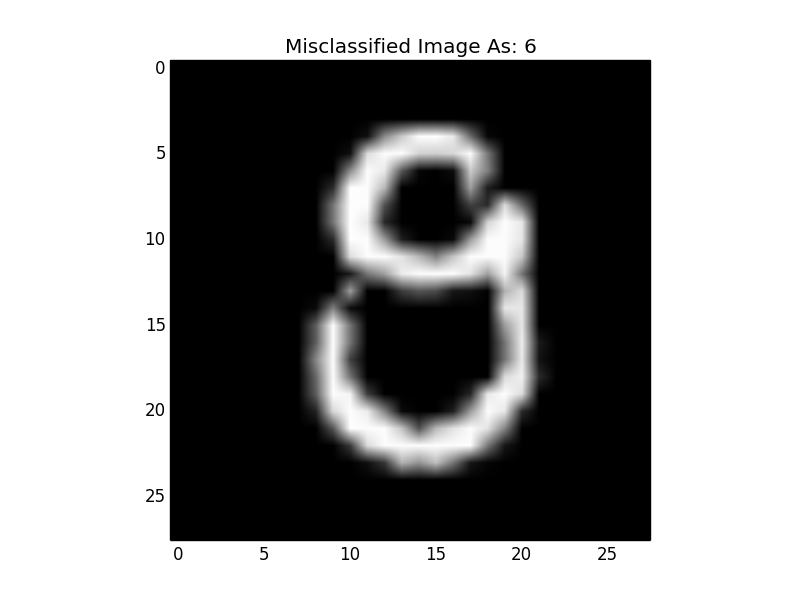
**3)** **KNN Confusion Matrix (k=5, training set = 15000)**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Predicted** | | | | | | | | | | | |
| **Actuals** |  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **0** | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 19 | 0 | 0 | 1 | 0 | 0 |
| **5** | 0 | 0 | 0 | 1 | 0 | 19 | 0 | 0 | 0 | 0 |
| **6** | 0 | 0 | 0 | 0 | 0 | 0 | 20 | 0 | 0 | 0 |
| **7** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 20 | 0 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 20 | 0 |
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 20 |

**Problem 4.**

**Problem 5.**

**KNN**

****

**Problem 6.**

**Problem 7.**