**Daniel Nussbaum and Sonia Nigam**

**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. A kernel is also used to transform the data to be more linearly separable. Which kernel works best is an empirical question that depends on the specifics of the data set that is being used.

**Problem 3**

**A.** Implemented in classifier\_1.py. It can be run using the following command:

*python classifier\_1.py*

**B.**

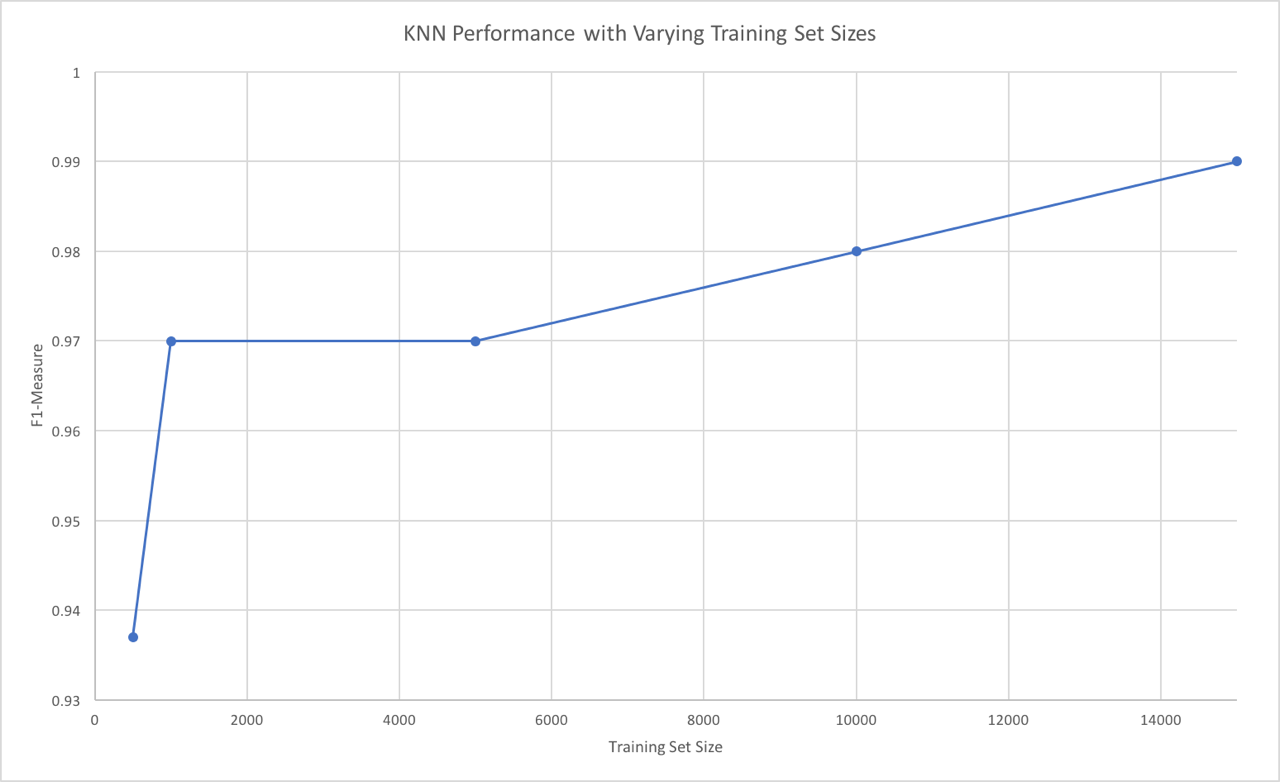
In order to determine the best hyperparamters for KNN, we conducted two experiments. These can both be found in the classifier\_1.py file under the two methods experiment\_one() and experiment\_two(). First, we looked to determine which training set size produced the best performance on the testing data. 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. 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. With these classifiers, we found the F1-measure, and printed out the corresponding confusion matrix. Examining the F1-measures, there is a clear trend of performance increasing with training data size.

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.

We chose F1-measure as a means to determine performance of a classifier because it provides a holistic measure of accuracy. F1-measure takes both precision and recall into account, and in turn is often used to gauge classification performance.

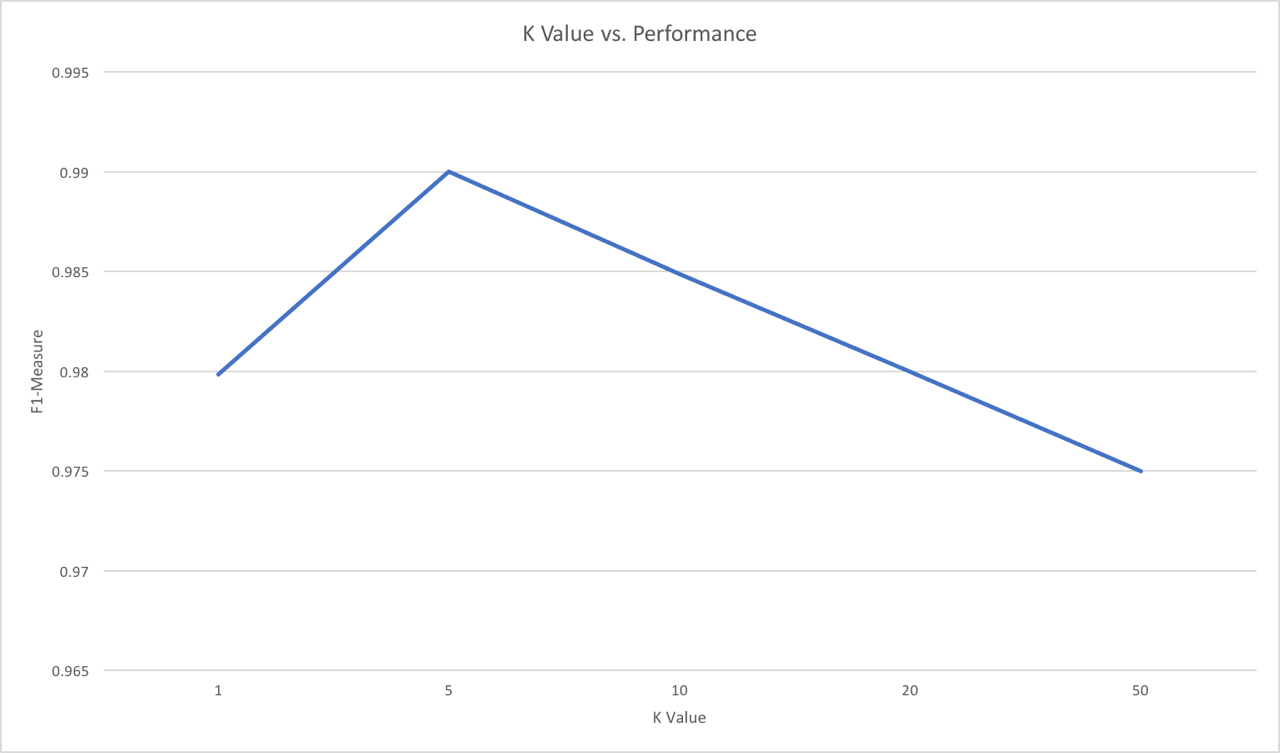
**1) KNN Training Set Size vs. Performance on Test Data**

|  |  |
| --- | --- |
| **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)** Below is a confusion matrix for the best performing KNN classifier, with an F1-measure of 0.989993746091.

**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.**

**A.** Implemented in classifier\_2.py. It can be run using the following command:

*python classifier\_2.py*

**B.** In order to determine the best hyperparameters for SVM, we conducted two experiments. These can both be found in the classifier\_2.py file under the two methods experiment\_one() and experiment\_two(). The first experiment that looked to determine training set size is the same as the first experiment for KNN. The same description can be found below.

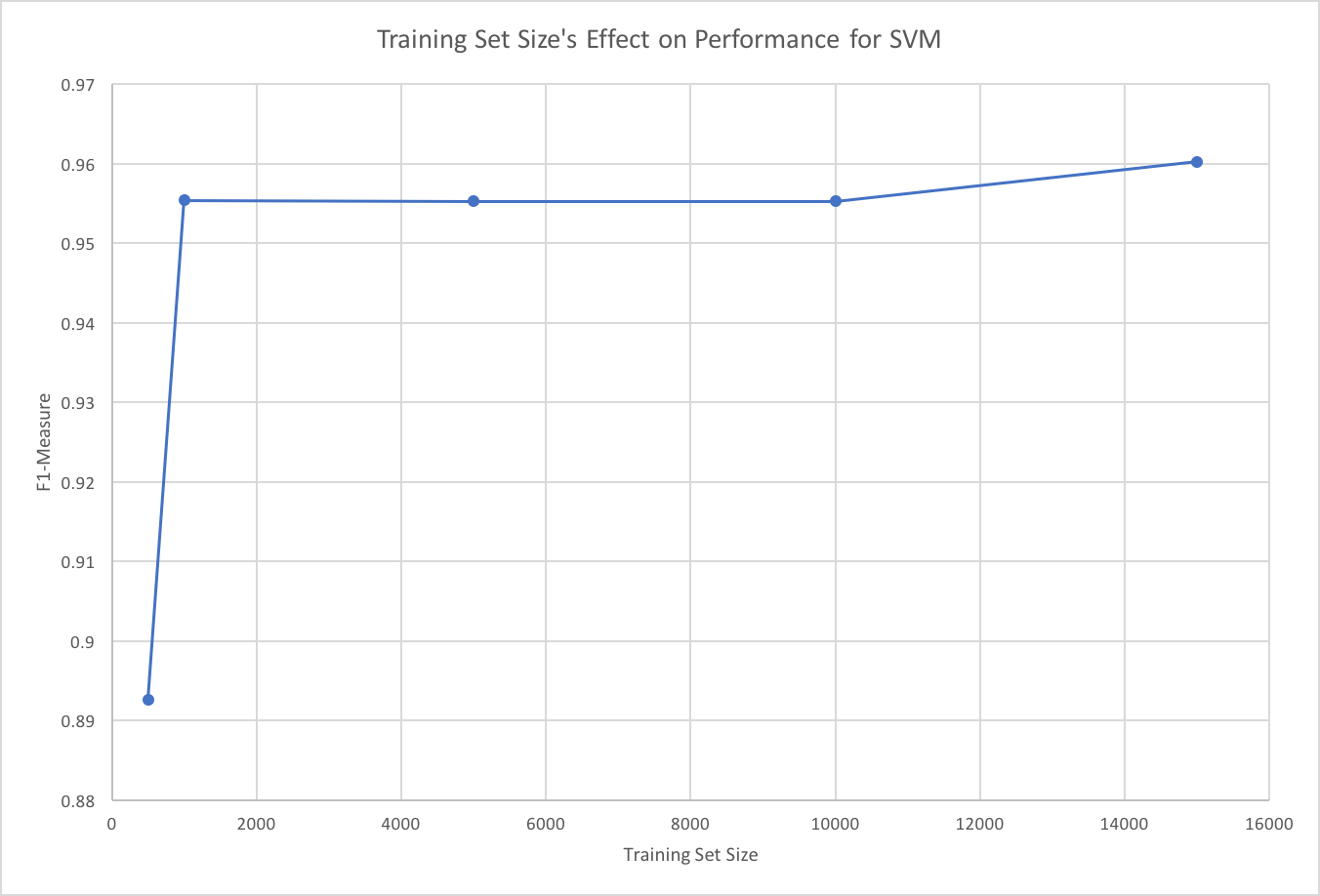
First, we looked to determine which training set size produced the best performance on the testing data. As such, we developed 5 different classifiers, using varying training set sizes: 500, 1000, 5000, 10000, and 15000. Each classifier was tested on the same testing data, so that the only variation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

Our second experiment, which looked to determine the optimal C value (the cost of misclassification) and the best kernel, 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 16 different classifiers, with a grid search over the parameter C of {1, 5, 10, 100} and the kernels of {“linear”, “poly”, “rbf”, and “sigmoid”}. The training data and testing data were kept constant for each classifier, such that the only difference was the combination of C and kernel. With these classifiers, we found the F1-measure. These results are shown below.

We chose F1-measure as a means to determine performance of a classifier because it provides a holistic measure of accuracy. F1-measure takes both precision and recall into account, and in turn is often used to gauge classification performance.

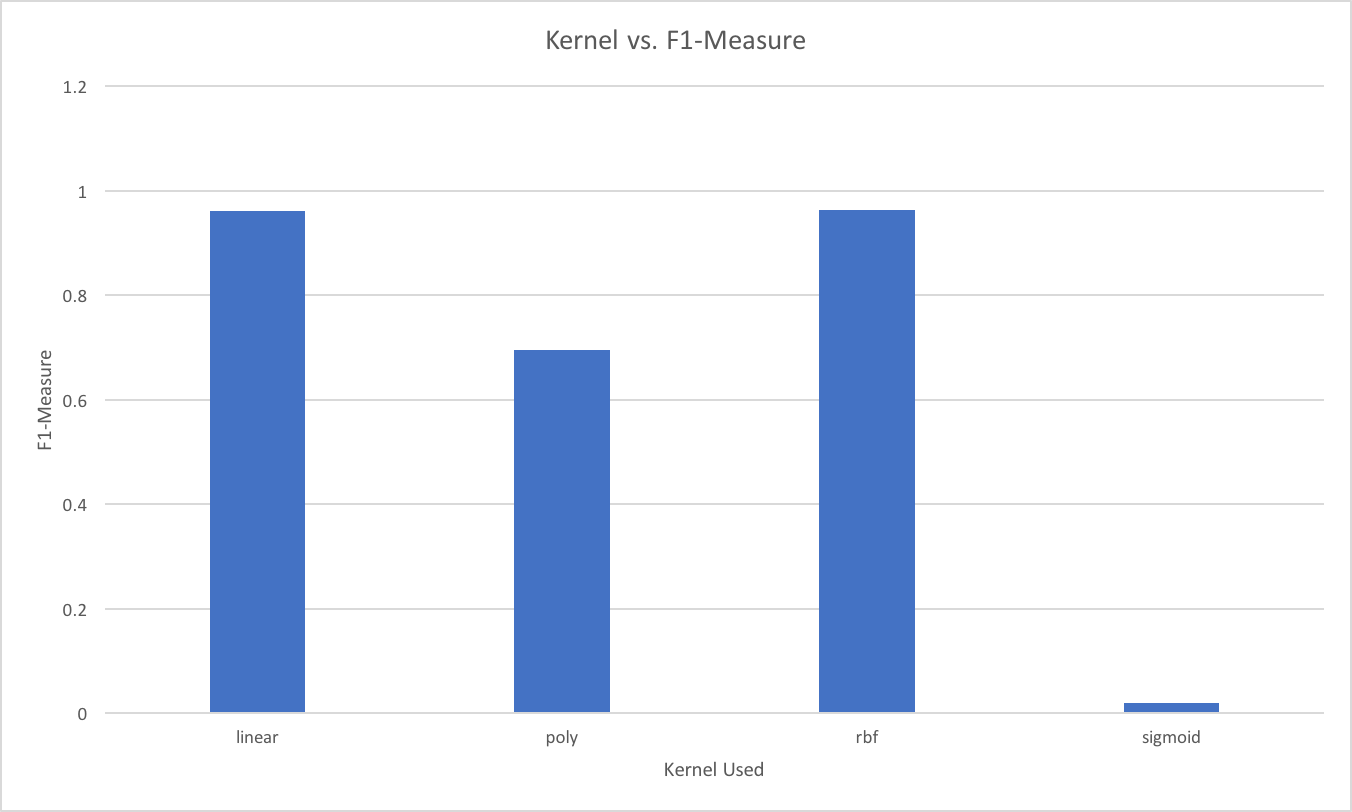
**1) SVM Training Set Size vs. Performance on Test Data**

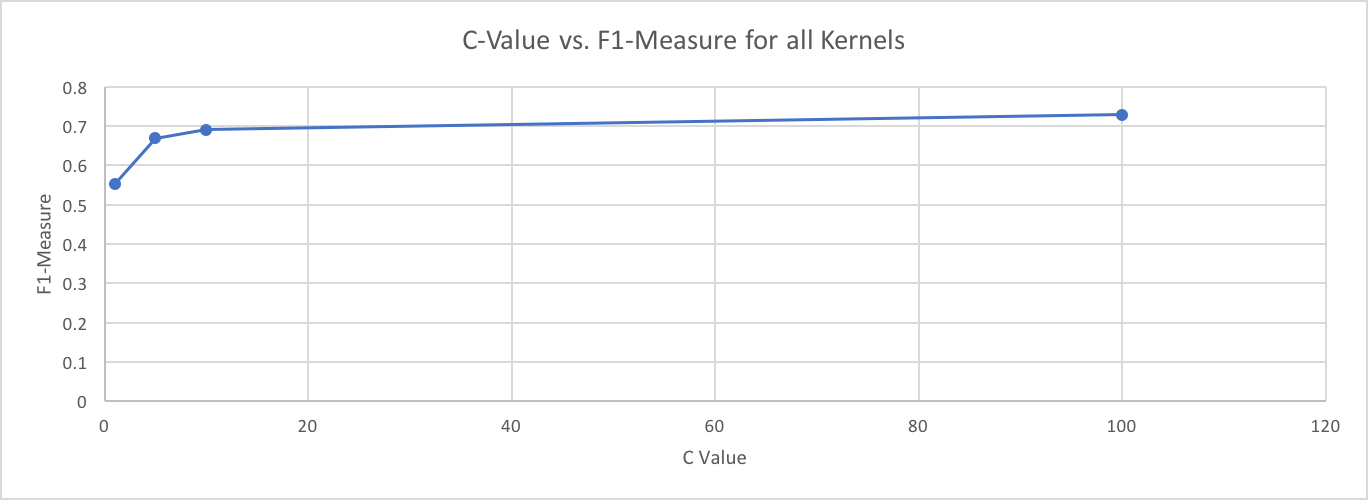
|  |  |
| --- | --- |
| **Testing Size** | **F1-measure** |
| 500 | 0.892654623674 |
| 1000 | 0.955399647074 |
| 5000 | 0.955280822808 |
| 10000 | |  | | --- | |  | | 0.955280822808 | |  | |
| 15000 | 0.960280822808 |

****

**2) SVM Parameters**

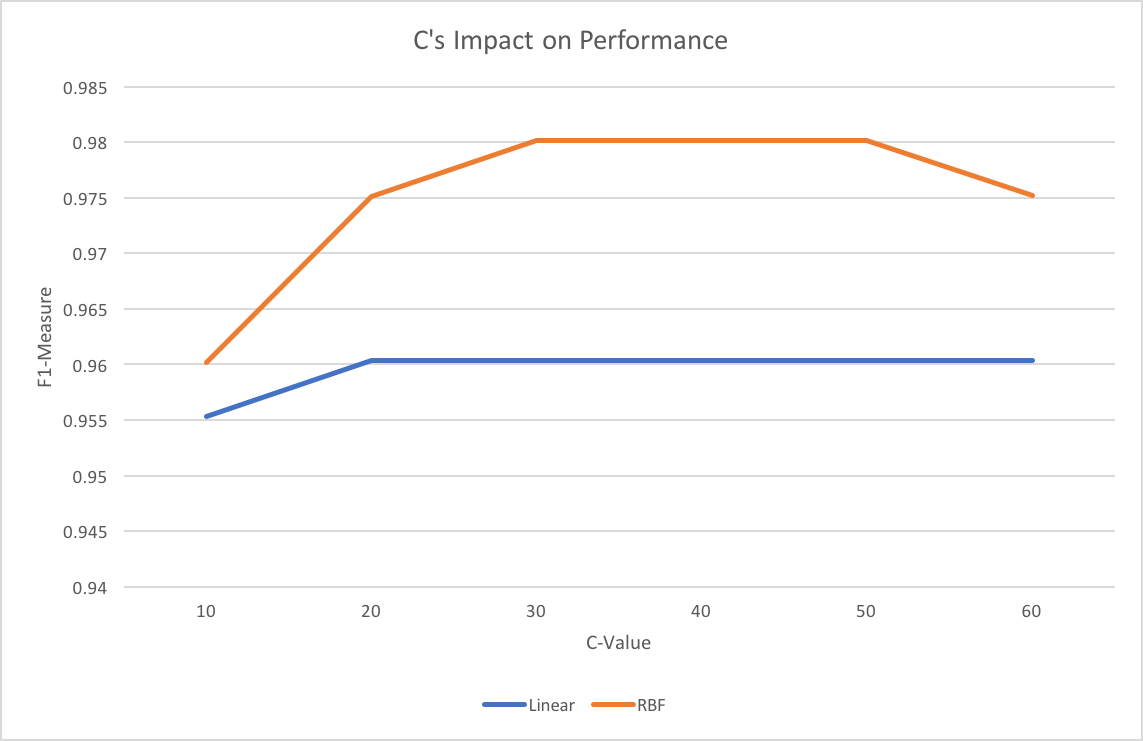
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Kernel** | **C Value** | | | |  |
| **1** | **5** | **10** | **100** | **Averages** |
| **Linear** | 0.965216653 | 0.964984365 | 0.955352578 | 0.960311693 | 0.961466322 |
| **Polynomial** | 0.263257587 | 0.731320374 | 0.826653224 | 0.960405901 | 0.695409271 |
| **RBF** | 0.960280823 | 0.960167806 | 0.960167806 | 0.975225587 | 0.963960505 |
| **Sigmoid** | 0.018181818 | 0.018181818 | 0.018181818 | 0.018181818 | 0.018181818 |
| **Averages** | 0.55173422 | 0.668663591 | 0.690088856 | 0.72853125 |  |

****

****

For kernels, linear and rbf performed comparably, with rdf performing slightly better. Sigmoid performed horribly. Our results also showed that a higher cost to misclassification yields a better classifier across all kernels tested. Combined, the best classifier used rbf with a cost of 100.

However, because C=100 did only marginally better than C=10, and because there was little performance improvement between linear and rbf kernels, we chose to run a second experiment between comparing linear and rbf kernels across C-values of {10, 20, 30, 40, 50, 60}. The intuition to further test the space between 10 and 100 turned out to be correct—a C-value of 30-50 is the correct optimum (with identical performance). We select 40 as the average of the 3.



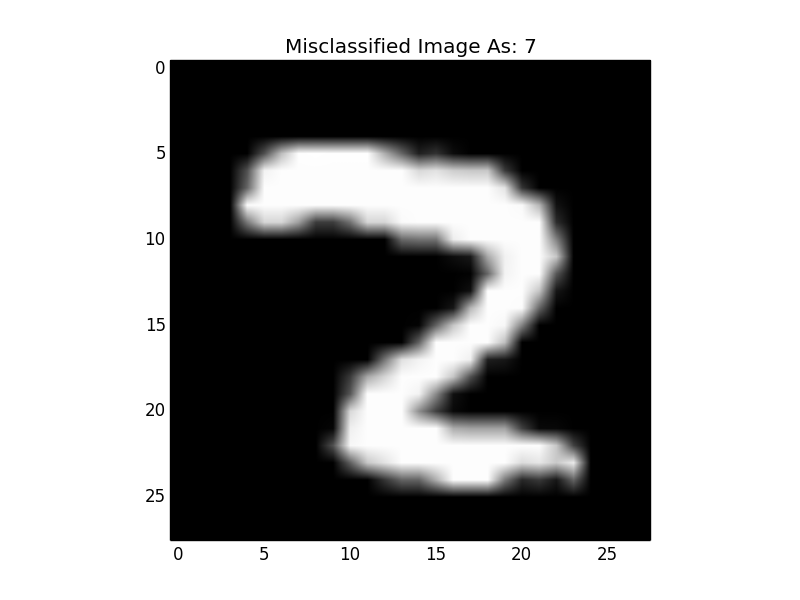
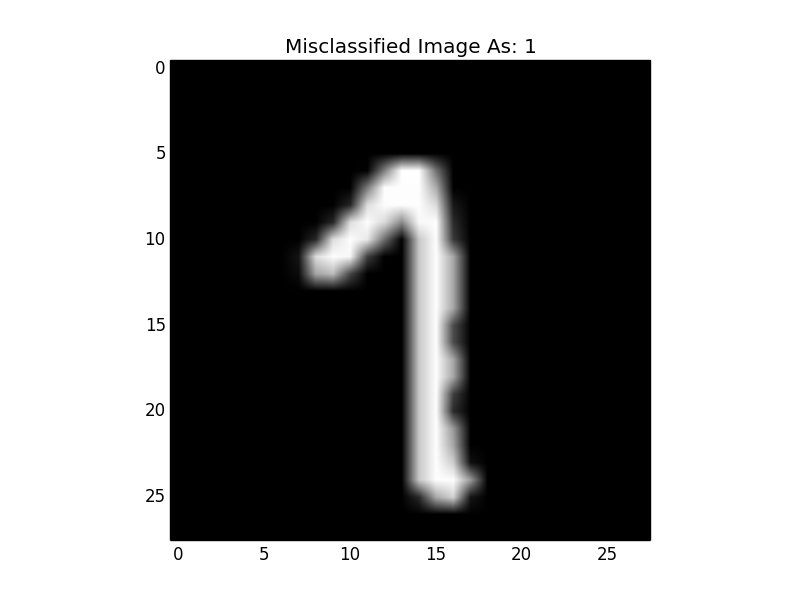
**3)** Below is a confusion matrix for the best performing SVM classifier, with an F1-measure of 0.980115697311.

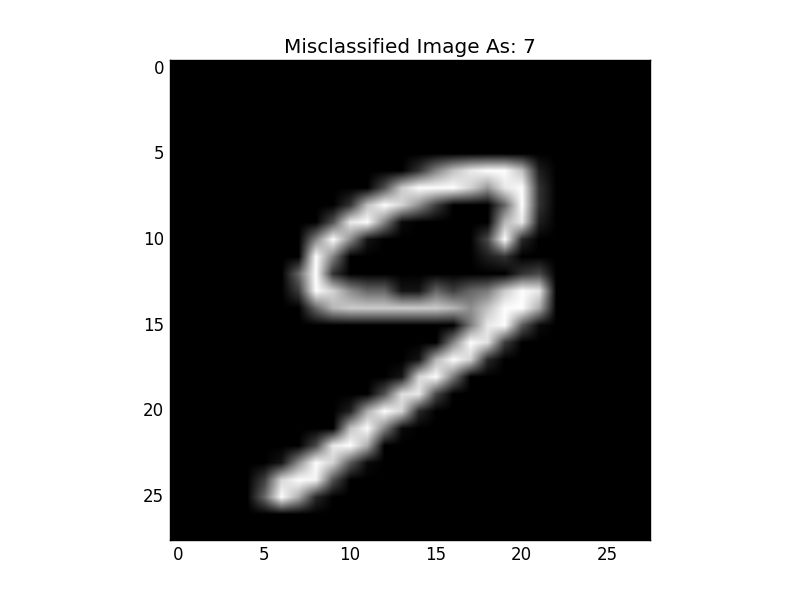
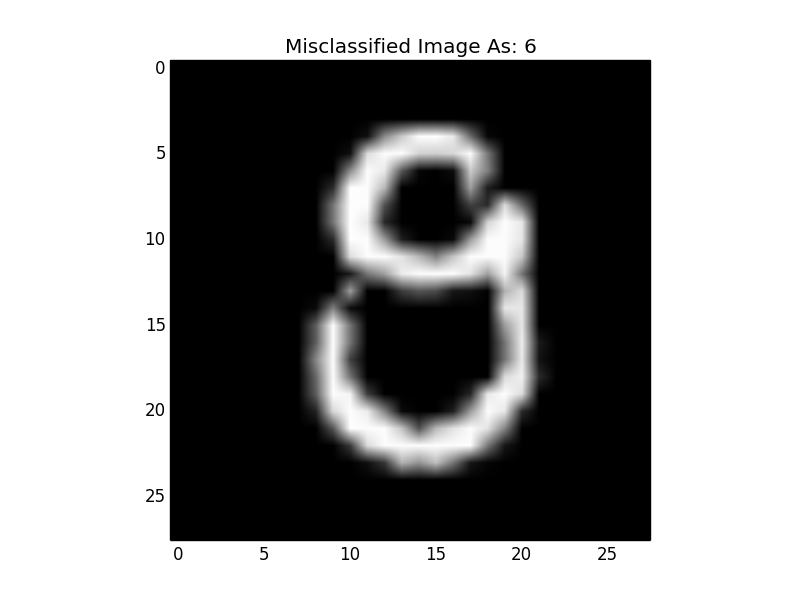
**SVM Confusion Matrix (C=40, training set = 15000, kernel=rbf)**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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 | 19 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 19 | 0 | 0 | 1 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 19 | 1 | 0 | 0 | 0 | 0 |
| **5** | 0 | 0 | 0 | 0 | 1 | 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 5.**

Misclassified images for KNN

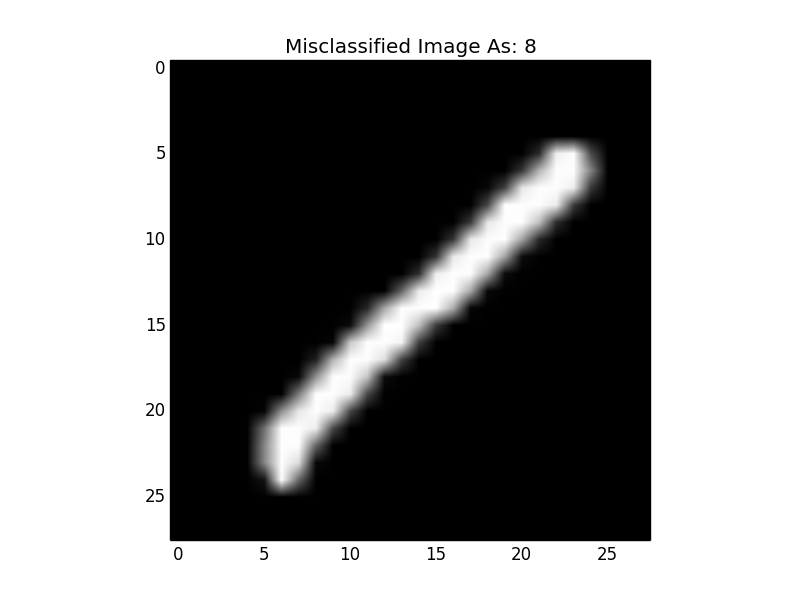
****



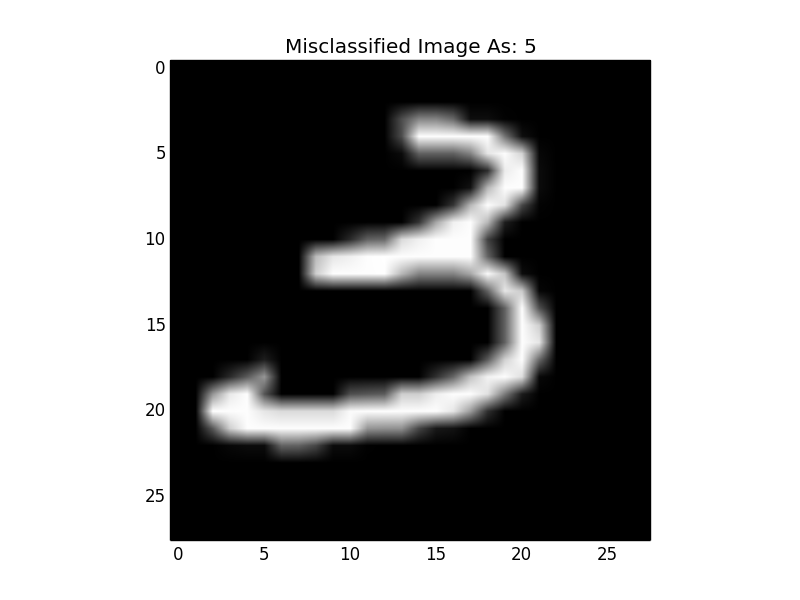
KNN finds k feature vectors that are most similar to the inputted feature vector, and classifies it as the majority label of these neighbors. In turn, if a number is messily written so that a dominant component of it appears to be another number, then it is often misclassified. For example, the “9” in the bottom right is written at a slant, such that the bottom of the curve and straight line up well with a “7” feature vector. Moreover, the top curve of the “9” is not in a standard position, so it’s likely that it did not match up with the pixels of “9” images in the training set. Similarly, the top left “7” was misclassified as a “1” due to its proportions and angles. A standard “7” image likely has a longer top line that is more flat. In turn, this slight tweak made it such that it matched up better with the standard “1” feature vector. The other two images are similarly disproportionate, resulting in their misclassifications.

When inspecting the data in problem 1, we focused on images that were so messy that it was hard to discern if it is even intended to be a digit. While we did touch upon confusion between digits, the examples primarily looked to expose how handwriting can result in digits that are confusing to distinguish beyond squiggly lines and blobs. That being said, these misclassified images shed light on our initial misconception. Although the examples in problem 1 were confusing to the human eye, they were predominantly well classified under KNN because their actual feature vectors still match up best to their given digit. Meaning, even if defining elements were missing, faint, or too thick, such that a human eye is confused, the feature vectors did not emulate any other digit more. In turn, these examples were not as challenging as anticipated.

Misclassified images for SVM

****

****



Similar to KNN, SVM struggled with images where changes in angles and proportions resulted in the feature vector being better aligned with another digit’s. For example, both images in the left column were “6” digits where the angle and the minimized circular component made it so the primary element of the image was what resembled the slant in a “2”, followed by its bottom line. On the same note, the image in the top right corner was classified as an 8, as the “1” was angled such that the dominant component resembled the cross in an “8”. The bottom left right image is a little less clearly another digit, but because this is based off of feature vectors, the proportions seemed to lead the algorithm astray. In general, SVM and KNN both struggled with images that are not necessarily difficult for the human eye, but instead have feature vectors that better emulate another digit’s from the training data.

**Problem 6**

Our SVM classifier with the optimal hyper-parameters had an F1-measure of 0.9801. The KNN classifier with the optimal hyper-parameters had an F1-measure of 0.9810. The KNN classifier performed better than the SVM by 0.0918%. This is hardly a practical improvement—the classifiers were extremely similar in performance. However, what little performance improvement the KNN classifier had was likely due to the fact that there is a very high probability of any given handwritten digit being similar to several other handwritten digits of the same type. For example, there is a somewhat low chance of an “8” from the testing set being similar to a “1” from the training set. This type of problem is ideal for a KNN classifier, as the clusters are well defined, for there is not much room for haziness or overlap. Meaning, for the most part, the 10 digits are quite distinctive. By limiting the number of neighbors to be well within a given cluster, the KNN algorithm excelled in classification. While some handwritten digits are extremely messy, and are hard to distinguish even with a human eye, for the most part they fall well within their defined cluster. This is further bolstered by the fact that this image set is preprocessed such that the feature vectors are well aligned.

**Problem 7**

**A.**

**Confusion Matrix for AdaBoosting with a Weak Classifier**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Predicted** | | | | | | | | | | | |
| **Actuals** |  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **0** | 18 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| **1** | 0 | 17 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **2** | 6 | 1 | 4 | 0 | 0 | 0 | 9 | 0 | 0 | 0 |
| **3** | 0 | 2 | 0 | 15 | 0 | 1 | 2 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 17 | 1 | 0 | 1 | 1 | 0 |
| **5** | 0 | 1 | 0 | 2 | 0 | 14 | 1 | 0 | 2 | 0 |
| **6** | 1 | 0 | 0 | 0 | 0 | 0 | 19 | 0 | 0 | 0 |
| **7** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 15 | 1 | 3 |
| **8** | 0 | 1 | 3 | 0 | 0 | 5 | 0 | 0 | 11 | 0 |
| **0** | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 2 | 2 | 12 |

AdaBoosting with a weak classifier produced an F1-measure of 0.696320382937. In problem 3, our best KNN classifier had an F1-measure of 0.989993746091. Our best SVM classifier had an F1-measure of 0.980115697311. In turn, this classifier did not outperform our classifiers in problem 3.

**B.**

**Confusion Matrix for AdaBoosting with SVM**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Predicted** | | | | | | | | | | | |
| **Actuals** |  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **0** | 19 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 18 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **2** | 0 | 0 | 19 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 19 | 0 | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 19 | 1 | 0 | 1 | 0 | 0 |
| **5** | 0 | 0 | 1 | 0 | 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 | 1 | 1 | 0 | 0 | 0 | 0 | 18 | 0 |
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 18 |

The AdaBoosting with an SVM classifier had an F1-measure of 0.945612622791. In problem 3, our best KNN classifier had an F1-measure of 0.989993746091. Our best SVM classifier had an F1-measure of 0.980115697311. In turn, this classifier did not outperform our classifiers in problem 3.

One would expect the AdaBoosting with an SVM classifier to outperform the standalone SVM classifier built in problem 3. However, when we implemented AdaBoosting with an SVM classifier like used in problem 3, the algorithm ran for a few hours without deriving a solution. With that, we thought it was important to retain the amount of data being processed, and instead adapt the SVM classifier to be more efficient. As such, we used a LinearSVC with AdaBoosting instead of passing a linear kernel parameter to a standard SVC. According to the documentation, the two should produce extremely similar results, as the difference is that LinearSVC utilizes liblinear rather than libsvm. This change in implementation essentially enables LinearSVC to be more flexible with penalties and loss functions such that it is more scalable. In turn, leveraging a LinearSVC in the AdaBoosting SVM made for an increasingly efficient algorithm, but likely was the cause for a decrease in F1-measure.

**C.** AdaBoosting with SVM produced better results than AdaBoosting with a weak classifier. Based on the confusion matrices above, one can see that AdaBoosting with SVM produced more accurate results. Moreover, we calculated the F1-measure for both classifiers utilizing the same training and testing data. As such, the only variation between the two algorithms was the base classifier. AdaBoosting with a weak classifier had an F1-measure of 0.696320382937 and AdaBoosting with SVM had an F1-measure of 0.945612622791. Normally you are advised to use a weaker classifier than SVM with AdaBoosting because SVM oftentimes overfits to the training data. In this case, due to the increase in F1-measure, we conclude that the training and testing data are so well aligned that the fear of overfitting isn’t as applicable. That being said, with less processed images, this may be a viable concern to consider.