Jeremy Midvidy, jam658
EECS 349, Fall 2017
HW9, Extra Credit

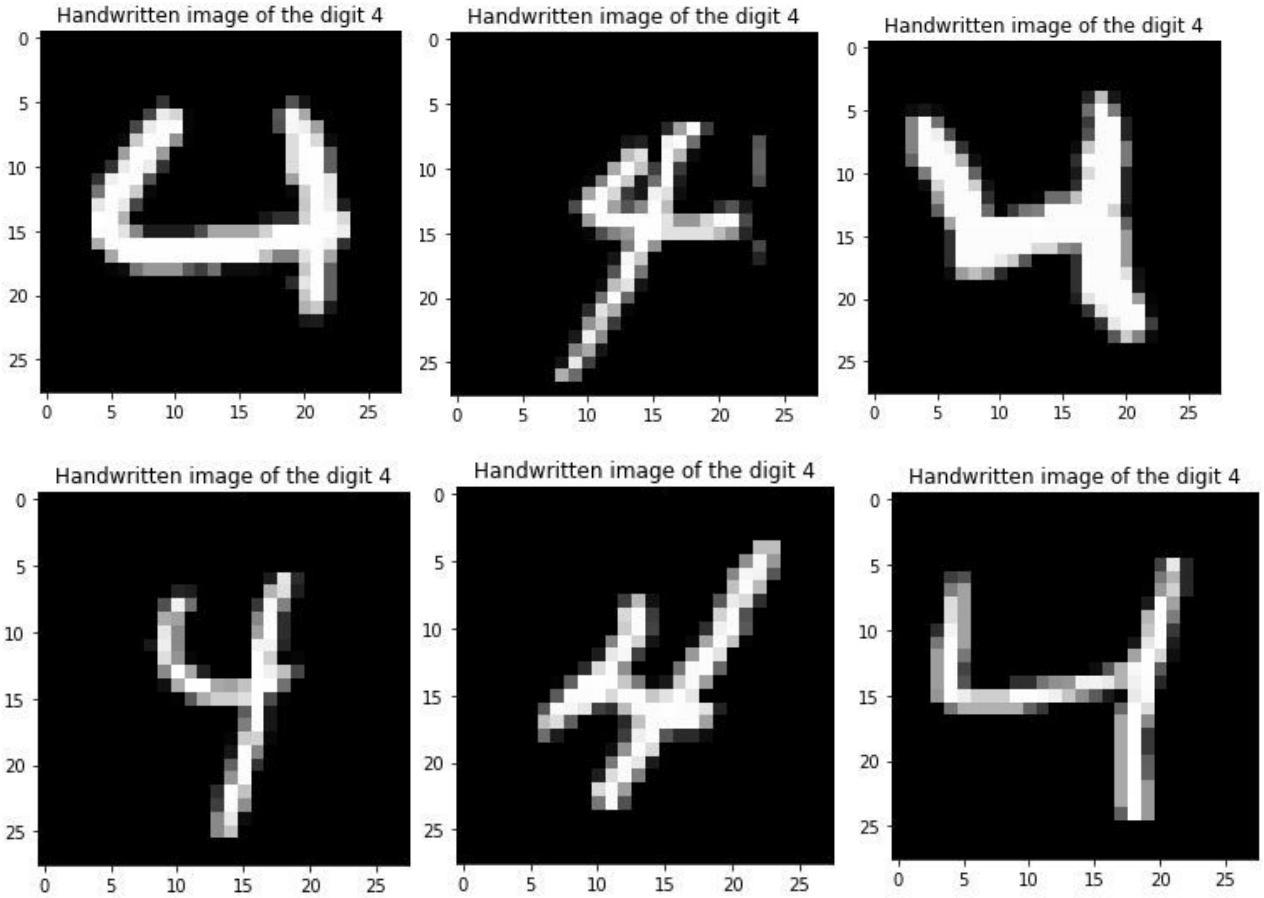
Problem One:

A.

There are 60,000 images in total. There are 6,000 examples per digit, with digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

B.

I looked through 50 examples of the digit '4.' Here are some examples of the same digit, 4, written differently.



It is clear that building a classifier for any digit will be hard. The digit 4 can be written a number of different ways, as illustrated above.

In some examples, 4 is closed on the top (R1:C2), in some examples it is open on the top (all others). In some examples, 4 has a tail through the vertical section (R2:C1, R2: C2) and in some examples 4 does not have a tail through the vertical section (R1:C3, R2:C3).

Even though I can instantly tell that each example is a 4, each example has so many different variations that building a program to decide which digit each example is will be understandably hard. There are so many possible variations that the tricky nature of building a classifier is clear.

C.

I am going to use a training set of 1,000 images, and a testing set of 100 images. I think this is a good subset of the data because the entire training set is 60,000 images and the entire testing set is 10,000 images. So a good downscale from 60,000-1,000 is 1,000-100. This is still enough information to train on and enough information to test well, but not too large a set to make my programs redundant, or too slow to perform a lot of testing with different parameters.

D.

The advantage of making the data set consistent along these lines is to normalize all the points so that their “center of mass” is uniform in the center of every image. This can help tremendously in classification because all images are now known to be centered, the same overall size (scaled to how much writing each image contains). This will make classification much easier because the classifier doesn't have to worry about this (because it’s already done) and also because the classifier doesn’t have to spend extra time worry about the formatting of the image.

<u>Problem Two:</u>

My two algorithms were SVM and kNN.

A.

Support Vector Machines (SVM) work by taking a set of training examples, each marked as belonging to a known label, and then builds a model that assigns new examples to one category or the other. Overtime, it uses the training data to learn the best way to assign classes to new inputs, so presto, you then have a useful classifier. SVMs would be useful in classifying images of handwritten digits because, in this example, we have known labels for known examples (ergo supervised learning). With this known data, the SVM algorithm will associate different features to classification labels in feature space, and then, in order to classify new

inputs (testing), by mapping the inputted features into the known label space, and then predict a label. With this kind of data set, of known labels with associated inputs, SVM will work pretty well.

k-Nearest Neighbor (kNN) works by taking a set of training examples, each marked as belonging to a known label, and then builds a model that assigns new examples to a category based on the classification of the k-nearest neighbors, of classification of the number (k) most similar known training inputs. kNN would be useful in classifying images of handwritten digits because, in this example, there are a lot of examples and each example has a known label. This means that there is a good likelihood that the kNN wil correctly classify inputs because with more known data, there is a greater change that the k-nearest neighbors are more similar to the testing input than are not similar. Since our data set has over 60,000 examples with known labels, kNN seems to be a logical choice of a classifier for this assignment.

B.

In SVM, a significant hyperparameter is 'slack'. Slack refers to the amount of "wiggle room" the model has when certain data points are just beyond a decision boundary, but still closer to the former label rather than the label on the other side of the boundary. With some slack, SVM classifiers become less rigid because points, even though they are beyond a decision boundary, are more classified as the appropriate class because those points are just beyond the decision boundary (which is just an estimated boundary, so it's not a perfect divider between classes).

In kNN, a significant hyperparameter is n_neighbors. This refers to the number of most-similar known inputs (called neighbors because to all be similar to the input, they must be close together themselves). As the number of neighbors that the classifier is used to then classify inputs changes, the performance of the classifier in predicting the correct label can then also change. The optimal n_neighbors value is very dependent on each data set, so I will try and find the optimal n_neighbors for this dataset later in the assignment.

<u>Problem Three:</u>

A.

SVM in program.

B.

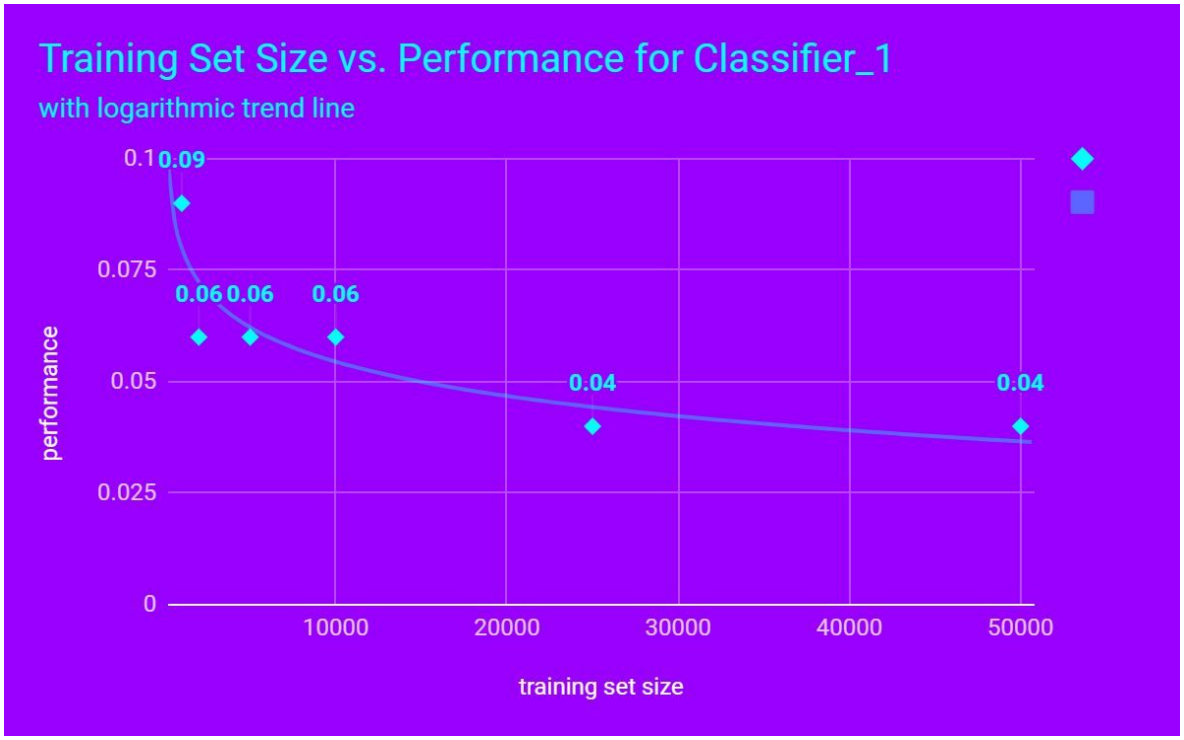
1.

I experimented with training set sizes [1000,2000,5000,1000,25000,50000] holding the testing set size to a constant 100.

Here are the results in a table:

Training Set Size	Performance Error
1000	.09
2000	.06
5000	.06
10000	.06
25000	.04
50000	.04

And here are the results graphically:



2.

I then experimented with different parameters for the SVM.svc() C parameter. As aforementioned, this determines the amount of “slack” given to each decision boundary. So with testing training 1000 and testing size 100, I evaluated the performance of these C values : [1, 2, 4, 8, 16, 32, 64]

Here are the results in a table:

C=	**Performance Error**
1	.09
2	.07
4	.06
8	.05
16	.03
32	.00
64	.00

And here are the results graphically:

As illustrated by the graph, the performance error of the classifier decreases with the increasing “slack” allowed for the program. When the program gets to a certain C value, C=32 (or around), the performance error becomes zero, and stays zero for any increased value of C (40, 50, 100, 5000, etc…). This is where the program overfits, so any data beyond that point isn’t indicative of the program's ability to classify because it can only classify individual inputs

rather than grouped classifications. According to my data, the optimal C-value parameter is 16. This means that, with this data set, a slack buffer of 16, when using a SVM classifier, will best classify input data points.

3.

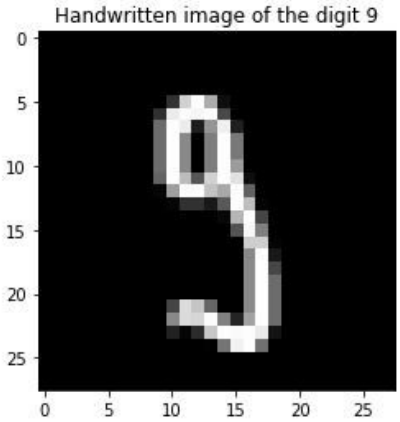
This confusion matrix shows how each digit was classified. I used optimal parameters C=16 and training set size 100. The vertical rows represent known labels, and the horizontal columns represent how each input was classified [digits 0-9]. Where row = 0 and col = 0, all 13 inputs were correctly classified as '0'. Where row = 1 and col = 1, 14 were correctly classified'. Where row = 4 and col = 4, 10 inputs were correctly classified and one was misclassified as digit '1'. And so on and so forth. This shows how the testing set performed over all of the examples.

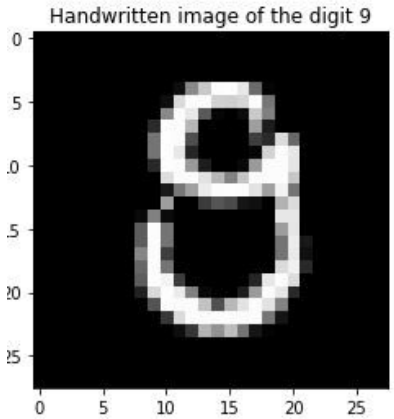
```
[13  0  0  0  0  0  0  0  0  0]
[ 0 14  0  0  0  0  0  0  0  0]
[ 0  0  6  0  0  0  0  0  0  0]
[ 0  0  0 11  0  0  0  0  0  0]
[ 0  1  0  0 10  0  0  0  0  0]
[ 0  0  0  0  0  5  0  0  0  0]
[ 0  0  0  0  0  0 11  0  0  0]
[ 0  0  0  0  0  0  0 10  0  0]
[ 0  0  0  0  0  0  0  0  8  0]
[ 0  1  0  0  0  1  0  0  0  9]
```

C.

As seen in the above matrix, my classifier had the most difficulty classifying the digit 9. 2 examples were misclassified as digit '1' and digit '5', respectively.

Here are the two images which were misclassified:





Here, it is clear why the program had trouble with these examples.

On the left, this 9 has an extremely circular top as well as a sharply curved tail. The classifier labeled this example as ‘1’. It does kind of look like a ‘1’. The bottom could be mistaken as a straight line and the top could be interpreted as the triangle on top of the one.

On the right, this 9 has an extremely wide shape; it has a circle at the top which connects strangely at the right-top most point. Additionally, its tail is very curved and almost touches the top portion of the digit. It could easily be mistaken for a 5; the top of the circle could be a line, and then it drops from top-right to the middle, then to the left where it connects to the bottom which then hooks into the bottom of the 5.

Although my classifier performed pretty well, these two examples show how hard it is to make a classifier work perfectly. You want allowance for edge case examples (setting C=16 greatly improved the dexterity of my classifier), but this can lead to misclassification of very borderline examples. My naked eye can easily tell that these two are 9s, despite me being awake that they are funkily drawn. Each of these examples fits into the “slack” provided by the program. They kind of look like a ‘1’ and a ‘5’, so it makes sense that my classifier would label them as such. Again, these two misclassifications really illustrate the difficulty of creating a perfect classifier.

Problem Five:

A.

kNN in program.

B.

1.

Again, I experimented with training set sizes [1000,2000,5000,1000,25000,50000] holding the testing set size to a constant 100.

Here are the results in a table:

Training Set Size	**Performance Error**
1000	.06
2000	.03
5000	.03
10000	.02
25000	.01
50000	.01

And here are the results in a graph

2.

Again, I then experimented with different values for the kNN number of nearest neighbors n_neighbors parameter. This is the most consequential parameter for kNN, so understanding how it changed was important to understanding the usefulness of kNN in this application. So with testing training 1000 and testing size 100, I evaluated the performance of these n_neightbors values: [1,2,3,4,8,16,24,32,64].

Here are the results in a table:

n_neighbors	**Error Performance**
1	.00

2	.05
3	.03
4	.04
8	.05
16	.08
32	.14
64	.21

And in a graph:

As illustrated in the graph, the optimal n_neighbors value for this data set is n_neighbors = 3. In my data, my classifier performed best with those parameters. The optimal number of neighbors the classifier should use is very data specific, so the optimal value 3 is only refers to this data set, and not all kNN classifiers in general.

3.

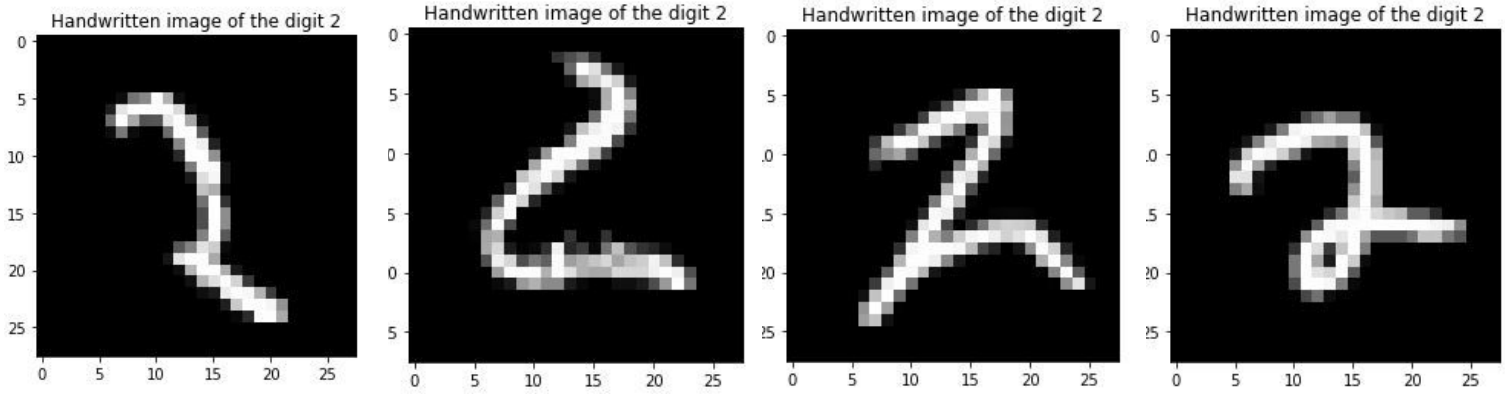
This confusion matrix shows how each digit was classified. I used optimal parameters n_neigbors=3 and testing set/training set size of 5000/500. The vertical rows represent known labels, and the horizontal columns represent how each input was classified [digits 0-9]. Where row = 0 and col = 0, all 50 inputs were correctly classified as '0'. Where row = 2 and col = 2, 48 were correctly classified as digit '2' while three were misclassified as digit '1' and one was misclassified as digit '7'. Where row = 6 and col = 6, 44 inputs were correctly classified as digit '6' and one was misclassified as digit '0'. And so on and so forth. This shows how the testing set performed over all of the examples.

```
[50  0  0  0  0  0  0  0  0  0]
[ 0 65  0  1  0  0  0  0  0  0]
[ 0  3 48  0  0  0  0  1  0  0]
[ 0  0  0 50  0  0  0  0  0  0]
[ 0  2  0  0 50  0  0  0  0  0]
[ 0  2  0  0  0 35  1  0  0  1]
[ 1  0  0  0  0  0 44  0  0  0]
[ 0  2  0  0  0  0  0 49  0  1]
[ 0  0  0  0  0  1  0  0 38  0]
[ 0  0  0  0  0  0  1  0  0 54]
```

D.

As seen in the above matrix, my classifier had difficulty classifying the digit 2. 3 examples were misclassified as digit '1,' one example was misclassified as digit '7'.

Here are the four images which were misclassified:



Here, it is clear why the program had trouble with these examples.

The first 3 examples were labeled as 1. Each kind of looks like the digit '1' with triangular tops, flat bases, and relatively straight vertical sections. While they each look like they could be a 1, they also each look different from a 2 in their own unique way. The first is almost rotated 45 degrees, the second has very flat top and bottoms, and the third doesn't have a loop connecting the vertical to the bottom, but is also indented at the end of the tail. So each looks like a one, but each also doesn't look like a two in its own unique way. This shows how hard it is for the classifier to get each one correct; they all share features of a 1, but since the features of the two are different, how could it form an optimal way to classify each as two if they are all "different" 2s? These kind of problems make these types of classifiers very tricky.

Likewise, the rightmost 2 looks like it could be a 7. It has a horizontal(ish) top that then cuts inward like the stem of the 7. And it's horizontal bottom looks it overlaps with part of the vertical, so the program really only sees the small partition to the right of the vertical. It is logical that kNN would classify this as a 7 given those features, and how it doesn't look a lot like a regular 2.

These misclassified examples show how a kNN could misclassify; examples where the instances share similar features of *another* label (the first three have features of a '1') and have different differents of the target label (they each don't look like a normal 2 for different reasons). Since a kNN evaluates on the closest, or most similar, instances in the training set, it makes sense that it would classify more based on strength of similarities (in this case more similar features to 1 or 7) rather than commonality of differences (all four have different reasons for *not* being a 2.)

Problem 5:

SVM was a better model because it performed better with optimal parameters.

With training/testing sets of size 5000/500, kNN (optimal n_neightbor=3) performed worse than SVM (optimal C=16) did, with errors .034 and .024 respectively.

With larger and larger training/testing sets, this difference would only increase, so it would be wiser to go with SVM in this dataset over kNN.

This makes logical sense as well. SVM builds a decision boundary and then assigns labels to inputs based on known training labels. kNN only assigns labels to inputs based on the labels of the n-most similar known training instances.

In this example, many examples (which are the most commonly misclassified) share more similarities with the digits they are misclassified as than share less commonalities with the digits they are supposed to be classified as. Therefore, since kNN evalues solely based on shared similarities, it is more likely that digits that look more like another number will be classified as that number instead of their correct classification.

SVM, on the other hand, builds a decision boundary and allows for slack. So if an instance looks a good amount like its actual digit, but also shares a lot of features with another, wrong digit, then it should make the correct classification more often than kNN because it has

‘slack’ to allow for differences with its target. So, with this type of data, SVM is the better classifier.

Problem 6:

A.

Confusion matrix:

```
[12  0  0  0  0  0  1  0  0  0]
 [ 0  6  0  6  0  0  0  2  0  0]
 [ 2  1  1  0  0  1  0  1  0  0]
 [ 0  0  0 10  0  0  0  1  0  0]
 [ 6  0  0  0  2  0  0  2  0  1]
 [ 0  0  1  1  0  0  0  3  0  0]
 [ 5  0  1  0  0  0  3  0  1  1]
 [ 0  0  0  0  0  0  1  8  1  0]
 [ 1  2  0  0  0  0  1  0  4  0]
 [ 0  0  0  2  1  0  2  4  1  1]
```

Error Rate:

.53

No, this is not better than the classifier built in q2.

B.

Confusion matrix:

```
[13  0  0  0  0  0  0  0  0  0]
 [ 0 14  0  0  0  0  0  0  0  0]
 [ 0  2  2  1  0  0  0  1  0  0]
 [ 0  7  0  4  0  0  0  0  0  0]
 [ 0  2  0  0  9  0  0  0  0  0]
 [ 0  5  0  0  0  0  0  0  0  0]
 [ 0  6  0  0  0  0  5  0  0  0]
 [ 0  1  0  0  0  0  0  9  0  0]
 [ 0  7  0  0  0  0  0  1  0  0]
 [ 0  2  0  0  0  0  0  9  0  0]
```

Error rate:

.44

No, this is not better than the classifier I built in q2.

C.

B was better but A had a much shorter run time.

B had an error rate of .44, while A had an error rate of .53 (on training/testing sets of 1000/100). This is a pretty big difference in classification performance.

Yet, B had a much slower run time than A. So B is not that great either. In a data set of 60,000, running B would take a lot longer than it would take to run A. So while B performs better, its runtime is significantly worse, and any programmer would need to take this into consideration when choosing which AdaBoost algorithm to implement.