Pattern Recognition Laboratories

Project 3

EE 516 - 01

Pattern Recognition

Spring 2023

Group 4

Students: Tre Carmichael, Nickolas Ogilvie, Nathan Jaggers

Project Due: 05/03/23

Instructor: Dr. Zhang

Linear classification for Handwritten Digits

The goal of the project is to perform linear classification on handwritten digits. Features will be extracted from handwritten digit training samples. Based on these training samples, various procedures will be implemented to determine the weighting vector for a linear discriminant function.

For Zeros and Ones:

The first section of the project entails 2-class classification for handwritten zeros and ones. From Project 1, the features determined to be optimal for classifying zeros and ones were eccentricity and minor axis length. The first step is to create the augmented training sample matrix, **Y**. A MATLAB script was written to iterate over each handwritten sample in the MNIST dataset. Each sample is binarized with a threshold determined in Project 1 (T = 100). From each digit sample, the eccentricity and minor axis length are extracted with **regionprops()**. This is performed with both the dataset for zeros and ones. The **Y** matrix is then built. Each row of the matrix is a feature vector for one sample. The feature vector for zero digits is:

[1, eccentricity, minor axis length]

The feature vector for one digits is:

[-1, -1*eccentricity, -1*minor axis length]

With the **Y** matrix, MATLAB scripts can be written to determine the weighting vector of a linear discriminant function to classify the samples. The first procedure implemented is the Batch Perceptron Procedure. In this procedure, some initial weighting vector is selected. This weighting vector is applied to all feature vectors in the **Y** matrix. Any output that is less than 0 indicates a misclassification. The feature vectors for all misclassifications are summed together. The summation is multiplied by the learning rate. Finally, the weight vector is updated by adding the product (from the previous sentence) to the current weight vector. This process is repeated for a fixed number of iterations or until the product used to update the weight vector is smaller than some threshold. At this point, the final weight vector is returned. See the Appendix for the MATLAB code implementing the Batch Perceptron Procedure.

Another script is written to implement the Fixed-Increment Single-Sample Perceptron Procedure. This procedure is similar to the batch procedure, except the weight vector is updated immediately when a misclassification occurs. The code implementing this is also in the Appendix. A trick that greatly improved the performance of this procedure was to shuffle the rows of the **Y** matrix. When created, the **Y** matrix has all positive classifications as consecutive rows and then all negative classifications as consecutive rows. This procedure iterates over the rows one at a time. With a decreasing learning rate, the learning rate may be too small to significantly affect the weight vector by the time the procedure reaches the negative classifications. By shuffling the rows, the procedure produces much better results.

Note: In all single-sample perceptron procedure tests, the number of iterations was set to 500000. In all batch perceptron procedure tests, the number of iterations was set to 10000. In the Ho-Kashyap tests, the number of iterations was set to 10000.

The two procedures are applied to the training samples. The learning rates and initial weight vectors are varied. The results for varying learning rate with the single-sample perceptron procedure are in Figure 1.1. The plot shows the scatter plot of the feature space. Each line is a decision boundary created from the weighting vector output by the MATLAB script with varying initializations.

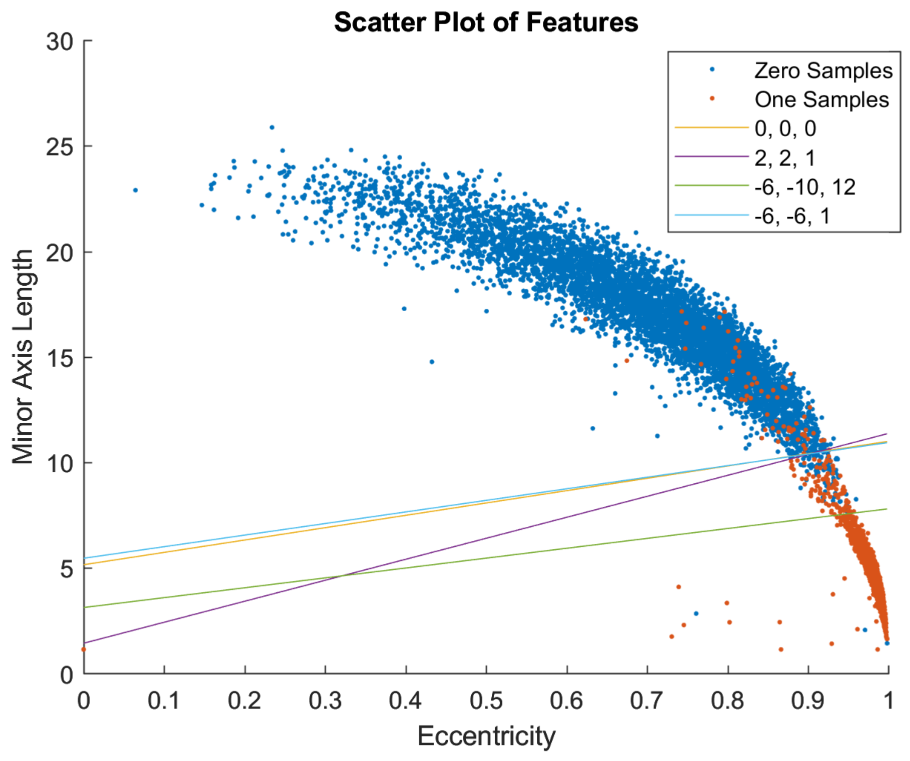

Figure 1.1: Single-Sample Perceptron with varying initial weight vectors

The performance of the determined weight vectors can easily be determined by applying the weight vector to the **Y** matrix. Multiply the weight vector with **Y**. Every output that is less than zero is a misclassification. The number of values less than zero in the product are the number of misclassifications. The number of values greater than zero in the product are the number of correct classifications.

With the [0, 0, 0] initial weight vector, there were 164 misclassifications and 12,501 correct classifications. With the [2, 2, 1] weight vector, there were 162 misclassifications and 12,503 correct classifications. With the [-6, -10, 12] weight vector, there were 189 misclassifications and 12,476 correct classifications. With the [-6, -6, 1] weight vector, there were 164 misclassifications and 12,501 correct classifications. Some observations can be made based on the results. The first observation is that each initialization will lead to a different solution. However, the solutions will be similar as long as the initial weight vector has relatively small magnitudes. The [0, 0, 0], [2, 2, 1], and [-6, -6, 1] initializations show similar solutions. In fact, all three decision boundaries appear to intersect at the same spot, which is right where the blue and red clusters meet. All three of these initializations show similar classification results. The [2, 2, 1] weight vector shows the best results by a small margin. The [-6, -10, 12] initialization shows the worst results both by inspection of its decision boundary and the number of misclassifications. The decision boundary clearly does not cross through where the blue and red clusters meet which leads to a higher number of misclassifications. Since the magnitude of the components of the initial weight vector here are higher, it takes longer for the procedure to change the weight vector. With a decreasing learning rate, there may not be enough time to correct a bad initialization before the procedure "converges". Based on these results, if the general format of a good weight vector solution is not known, it is better to use small initialization components. If a good estimate of the solution was already known, using higher magnitude components in the initialization is more reasonable. Due to the random shuffling of the rows of **Y**, the procedure will produce different results each run, even with the same weight initialization. Some runs the results may be slightly better and some runs may be slightly worse. Overall, the shuffling leads to significantly better results.

The effects of a various learning rate were also explored. The learning rates explored were: $\frac{1}{k}$, $\frac{1}{k^2}$, $\frac{1}{\sqrt{k}}$, $\frac{1}{\sqrt[3]{k}}$. The results are shown in Figure 1.2. The initial weight vector was set to [0, 0, 0] for all results in Figure 1.2.

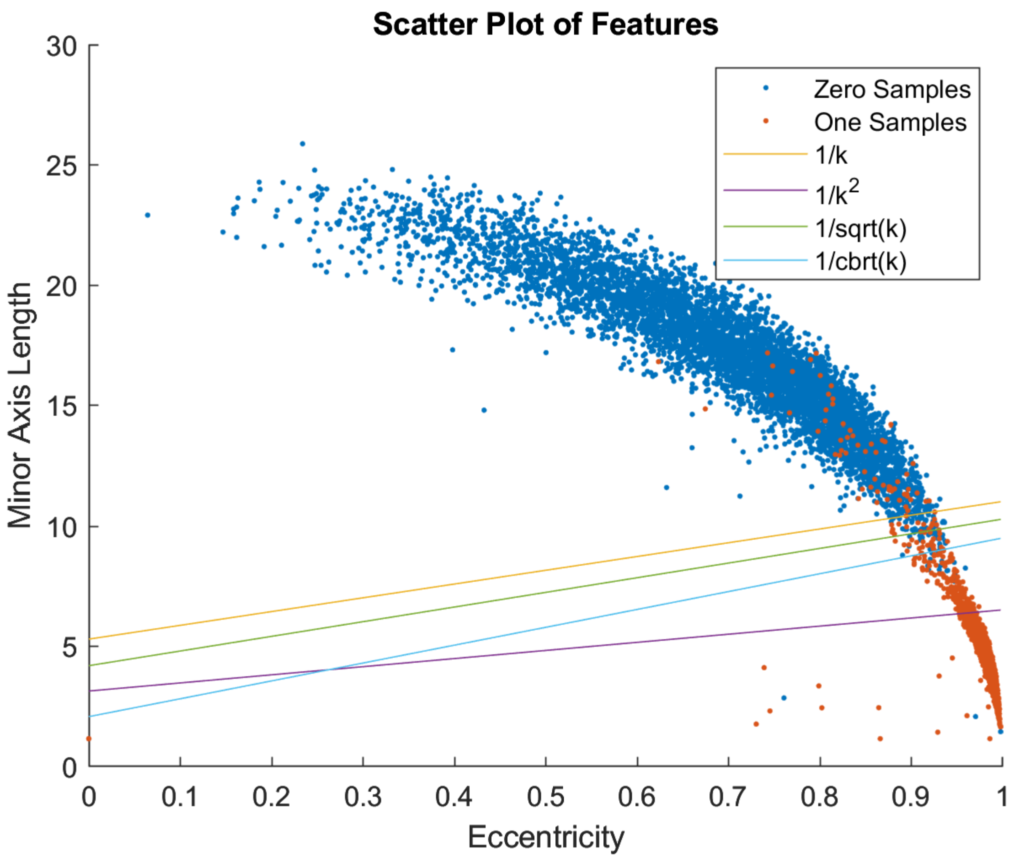

Figure 1.2: Single-Sample Perceptron with varying learning rates.

The results for a learning rate of $\frac{1}{k}$ were 165 misclassifications and 12,500 correct classifications. The results for a learning rate of $\frac{1}{k^2}$ were 320 misclassifications and 12,345 correct classifications. The results for a learning rate of $\frac{1}{\sqrt{k}}$ were 144 misclassifications and 12,521 correct classifications. The results for a learning rate of $\frac{1}{\sqrt[3]{k}}$ were 145 misclassifications and 12,520 correct classifications. Due to the shuffling of rows, the results would be different over each row. However, on average the learning rates $\frac{1}{\sqrt{k}}$ and $\frac{1}{\sqrt[3]{k}}$performed the best. The learning rate $\frac{1}{k^2}$ performed the worst. In certain cases the $\frac{1}{k^2}$ learning rate would lead to a solution with up to 50% misclassifications. These situations can be caused by a learning rate that drops off too quickly; the learning rate can cause the procedure to converge too early at a bad solution. The learning rates that drop off more slowly tend to be more consistent and achieve better results but come with the cost of increased simulation time.

The steps taken to produce Figure 1.1 and 1.2 are repeated with the Batch Perceptron Procedure. The results for the varying initial weight vectors are shown in Figure 1.3. In the batch procedure, every sample is considered at every iteration. This means the results will be the same every time the procedure is used, as long as the initialization is the same. In the batch procedure, the updates are based on all misclassified samples. This means the updates are generally larger in magnitude. To account for the larger updates, the initial weight vectors are increased by a scale factor. The final weight vectors were in the thousands to ten-thousands range. The batch procedure tests used the same initial weight vectors as the single-sample procedure but with a scale factor of ~7000 (this factor led to reasonable variation in the final results).

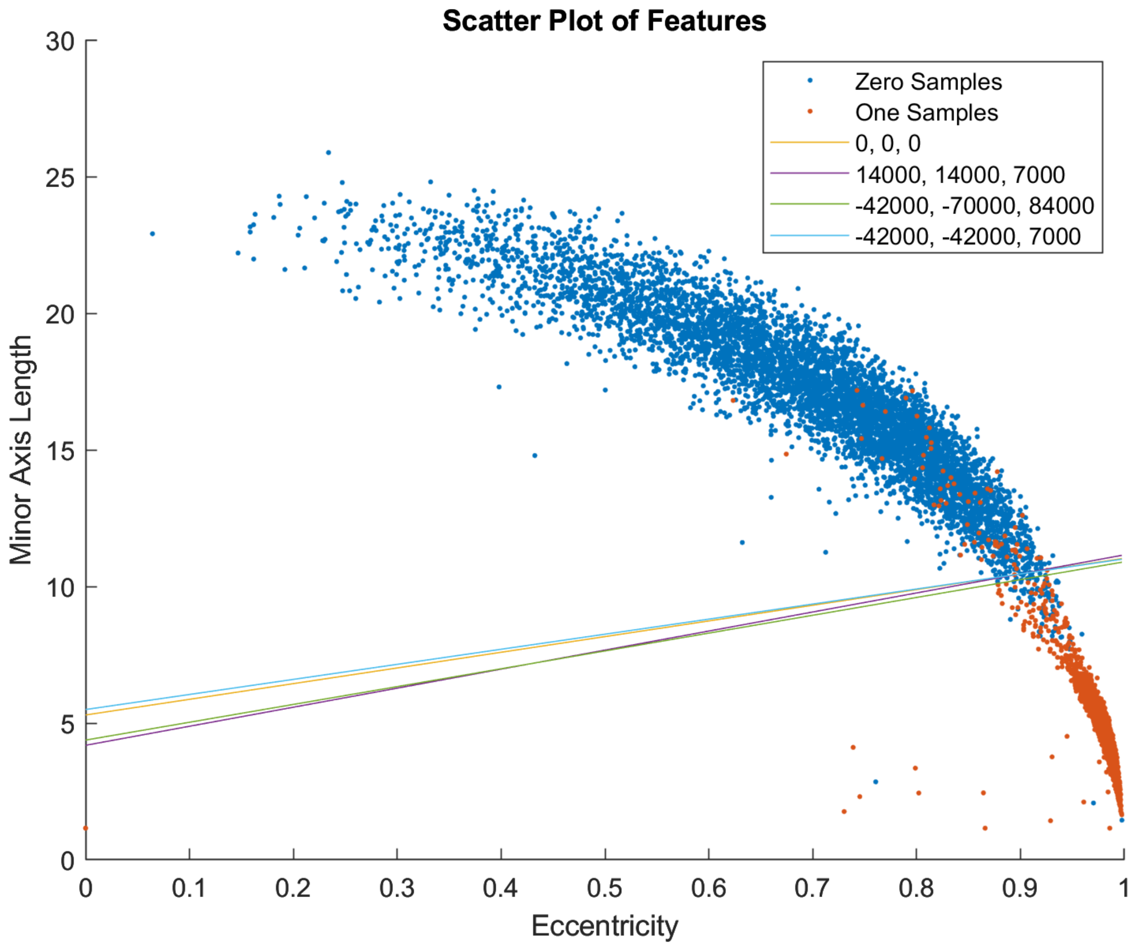

Figure 1.3: Batch Perceptron with varying initial weight vectors.

The batch perceptron results appear to show smaller variation with initialization. However, similar levels of variation could likely be achieved by increasing the scale factor. By inspection, all four initializations lead to solutions that appear to provide a good separation of the clusters. The [0, 0, 0] initialization leads to 166 misclassifications and 12,499 correct classifications. The [14000, 14000, 7000] initialization leads to 166 misclassifications and 12,499 correct classifications. The [-42000, -70000, 84000] initialization leads to 155 misclassifications and 12,510 correct classifications. The [-42000, -42000, 70000] initialization leads to 164 misclassifications and 12,501 correct classifications. All initializations lead to good classifiers, but the [-42000, -70000, 84000] initialization leads to the best classifier. Each solution had a weight vector of format [negative number with smaller magnitude, negative number with larger magnitude, positive number]. The [-42000, -70000, 84000] initialization matches this format most closely which could explain why it achieves the best results.

The effects of varying learning rate are now explored for the batch perceptron procedure. These tests used the same initialization and learning factors as the tests for the single-sample perceptron. The results are shown in Figure 1.4.

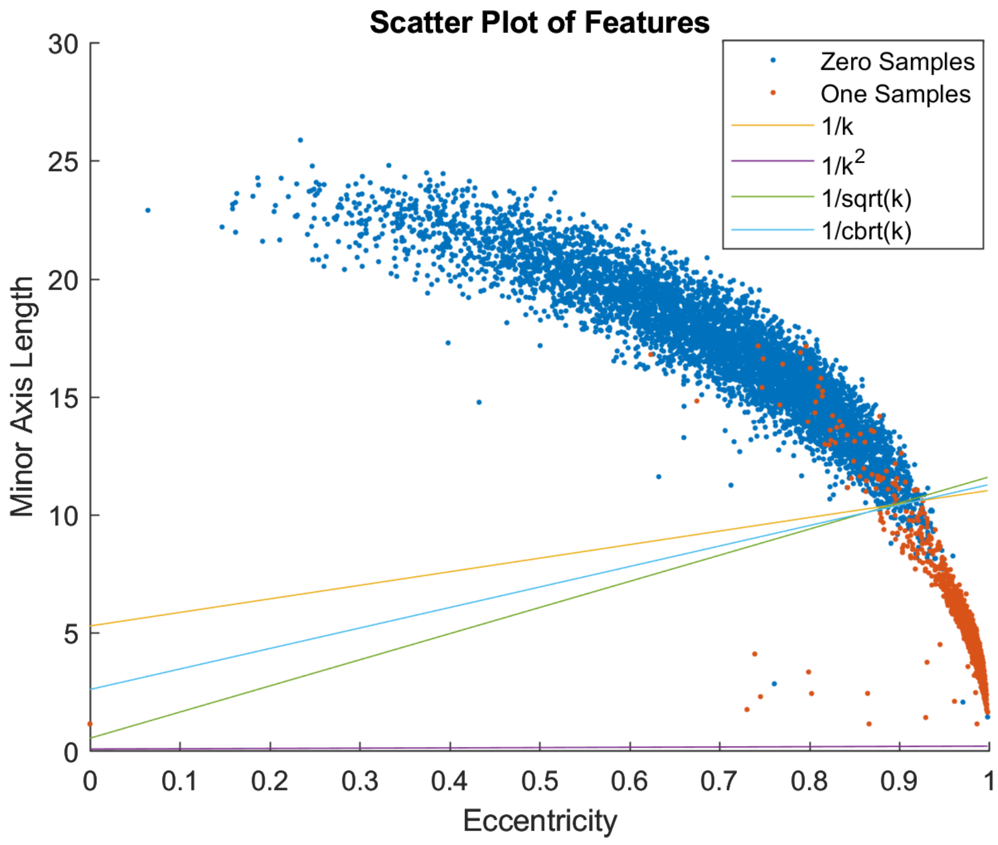

Figure 1.4: Batch Perceptron with varying learning rates.

The results for a learning rate of $\frac{1}{k}$ were 166 misclassifications and 12,499 correct classifications. The results for a learning rate of $\frac{1}{k^2}$ were 6,742 misclassifications and 5,923 correct classifications. The results for a learning rate of $\frac{1}{\sqrt{k}}$ were 180 misclassifications and 12,485 correct classifications. The results for a learning rate of $\frac{1}{\sqrt[3]{k}}$ were 163 misclassifications and 12,502 correct classifications. The $\frac{1}{k}$, $\frac{1}{\sqrt{k}}$, $\frac{1}{\sqrt[3]{k}}$ performed similarly. The learning rate $\frac{1}{k^2}$ performed the worst. The decision boundary created with the $\frac{1}{k^2}$ had over a 50% misclassification rate. The procedure returns a solution when the updates are under some threshold. The steep drop-off of this learning rate could cause the procedure to return too soon, when the procedure is at a bad solution. This is likely the reason for the bad results with this learning rate. Overall, the learning rates with a gentler drop-off seem to perform better.

After testing with various initializations and learning rates, optimal classifiers were determined for both procedures. For the single-sample perceptron, this optimal classifier was achieved with a learning rate of $\frac{1}{\sqrt{k}}$ and an initial weight vector of [0, 0, 0]. The solution weight vector was [-1.7594, -2.7426, 0.4606]. The results for this solution on the training data were 141 misclassifications and 12,524 correct classifications. This is a correct classification rate of 98.89%. The results on the test data were 18 misclassifications and 2,097 correct classifications. This is a correct classification rate of 99.15%. The scatter plot for the test data with the decision boundary is shown in Figure 1.5. Figure 1.5 and the correct classification rate both indicate that the classifier does an excellent job for the zero and one digits.

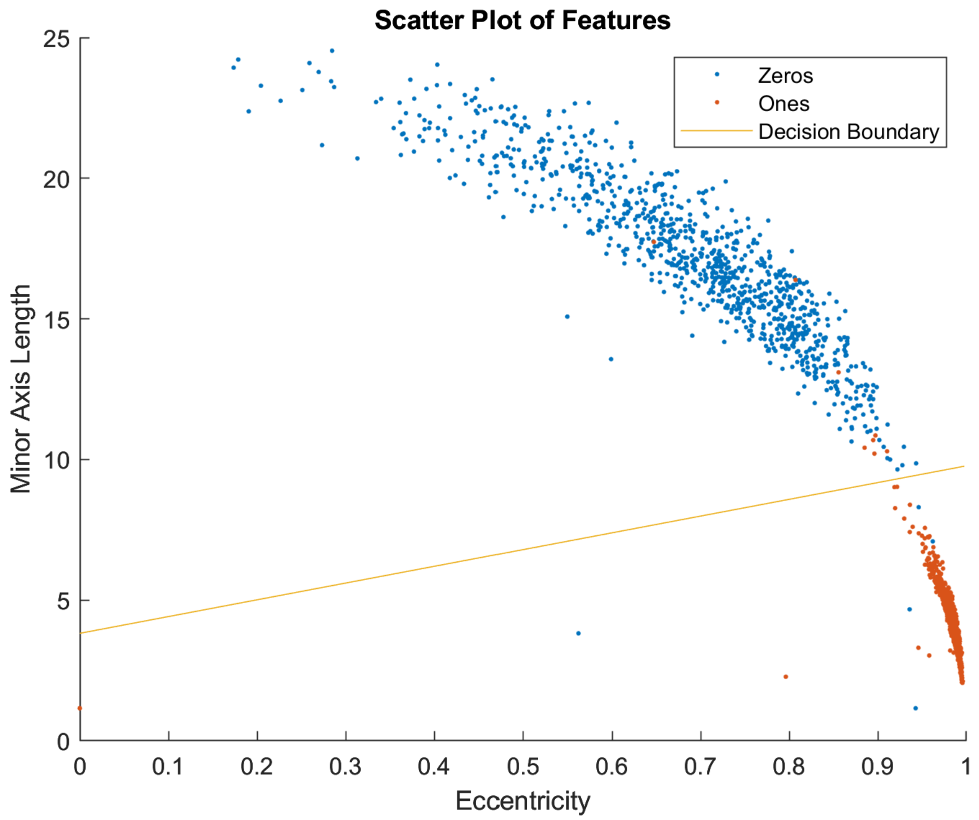

Figure 1.5: Scatter plot of test data and best decision boundary for single-sample perceptron.

For the batch perceptron, the best classifier was achieved with a learning rate of $\frac{1}{k}$ and an initial weight vector of [-42000, -70000, 84000]. The solution weight vector was [-56578, -84290, 12920]. The results for this solution on the training data were 155 misclassifications and 12,510 correct classifications. This is a correct classification rate of 98.78%. The results on the test data were 21 misclassifications and 2,094 correct classifications. This is a correct classification rate of 99.01%. The scatter plot with the decision boundary for are shown in Figure 1.6. Both by inspection of Figure 1.6 and the calculated correct classification rate, the batch perceptron creates an excellent classifier for the zeros and one digits.

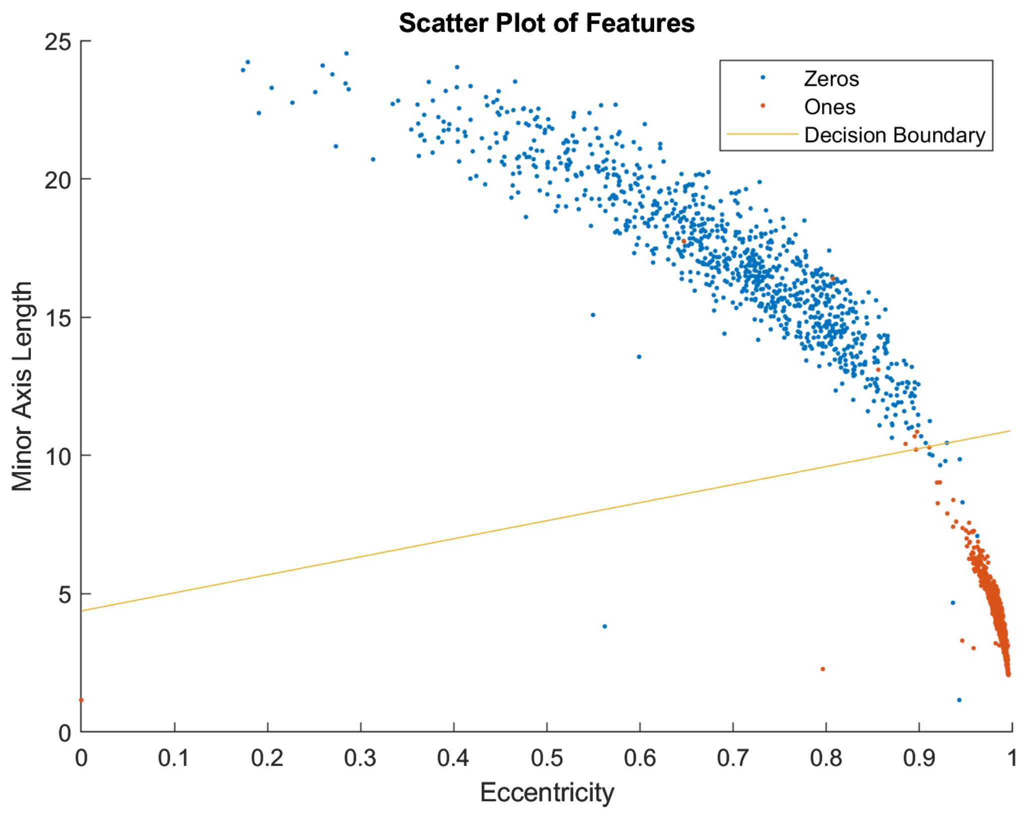

Figure 1.6: Scatter plot of test data and best decision boundary for batch perceptron.

Figure 1.7 shows the best decision boundaries generated by both procedures. By inspection, both procedures generated very similar boundaries.

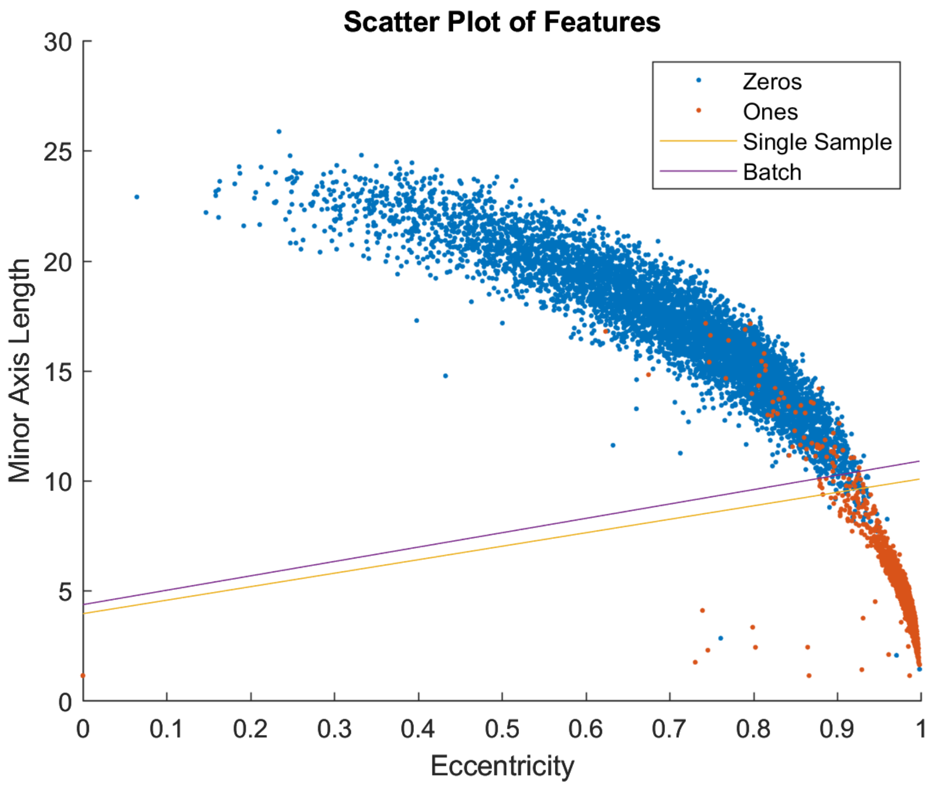

Figure 1.7: Best decision boundaries for single-sample and batch perceptron.

After running these tests with the batch perceptron and single-sample perceptron, several observations can be made about the two procedures. First, the run-time in MATLAB to generate the classifier for the single-sample perceptron is nearly instant. The run-time for the batch perceptron is about 45 seconds. Each step of the iteration in batch perceptron accounts for all samples which causes this procedure to take significantly longer. The single sample perceptron appears to be more sensitive to the learning rate and initialization. Since each iteration in the single-sample perceptron makes smaller changes, a learning rate with too steep a drop-off can cause the procedure to completely fail. Another observation is that the batch perceptron is more consistent than the single-sample perceptron due to single-sample depending on the order of the **Y** matrix. Some runs the single-sample perceptron would output better solutions than the batch perceptron, and some runs the solution would be worse. Both procedures produced similar results in that they both produced excellent classifiers. Overall, the single-sample perceptron performed slightly better in the zero and one classification. It ran much faster and was able to reach solutions that performed slightly better than solutions from the batch perceptron.

The next step in the project is to implement the Ho-Kashyap procedure to find the solution vector. The Ho-Kashyap procedure attempts to minimize the mean square error between the output of the **Y** matrix multiplied by the solution vector, **a**, with some arbitrary, positive margin vector, **b**. A MATLAB script was written to implement this procedure (see Appendix for code). The procedure was run with various initializations for the margin vector. The resulting decision boundaries are shown in Figure 1.8. The initializations were all ones, 0.1 to 10 linear spaced, 1 to the number of training samples, and the absolute value of Gaussian noise. The learning rate is a constant $\frac{1}{k}$ for these tests.

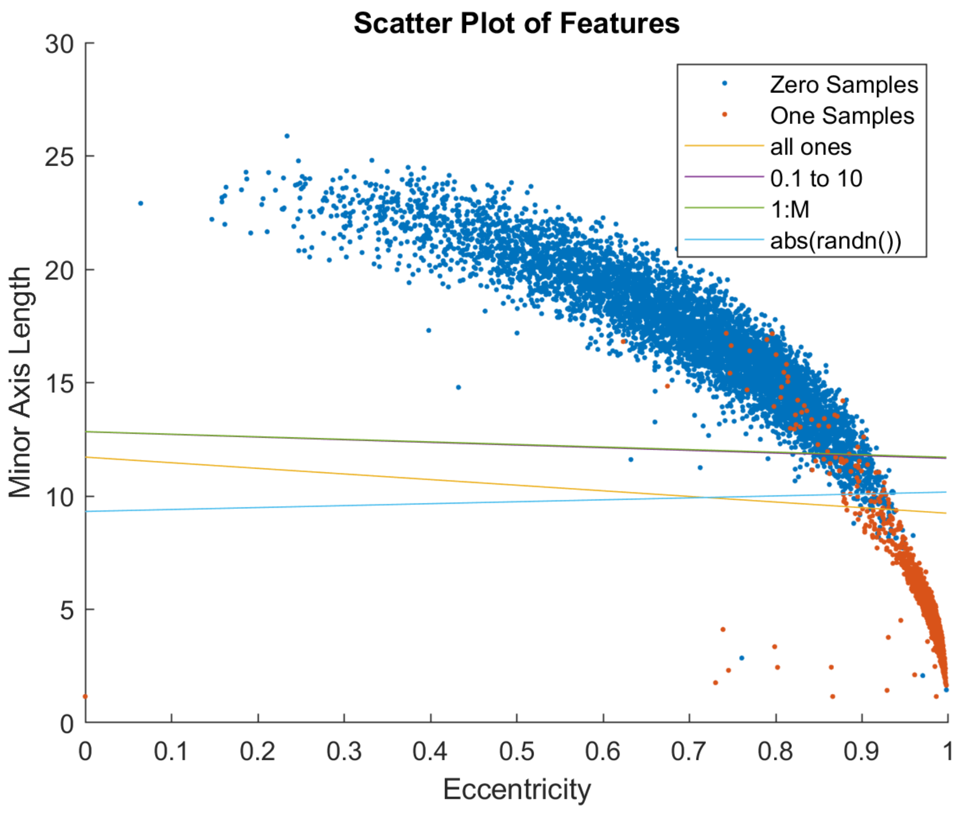

Figure 1.8: Ho-Kashyap with various initializations.

The margin vector is essentially representing the desired distance of each training point from the decision boundary. With no prior knowledge about a reasonable distance about each point from the boundary, it is difficult to set up the initial margin vector. The all ones initial vector is useful as it is just saying to try to find a decision boundary that gives equal spacing to all training points. With no prior knowledge, this is a reasonable initialization. This initialization produced the best results with 136 misclassifications and 12,529 correct classifications. The second best initialization was the randn() initialization. This decision boundary 148 misclassifications and 12,517 correct classifications. The random initialization, similar to the all ones initialization, doesn't assume to know the locations of the training points. This could be why it achieves decent results. The linearly increasing initializations both achieved bad results with ~350 misclassifications. The location of all the training points are known. Theoretically, one could choose a margin vector to get the best results. However, determining the optimal distance of each training sample from the optimal boundary would be extremely time-consuming. Overall, choosing a simple all ones initialization appears to be the best option.

The effects of varying the learning rate with the Ho-Kashyap procedure are also explored. The same learning rates used in the perceptron procedures are used here. The results are shown in Figure 1.9.

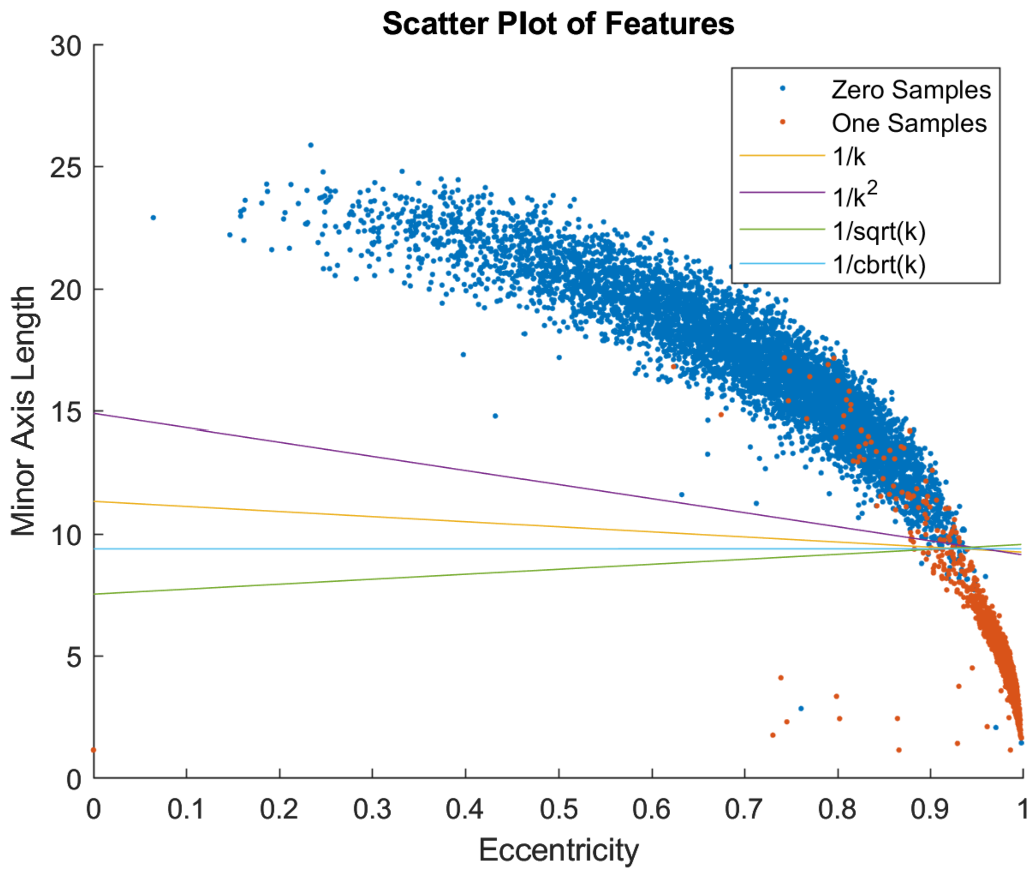

Figure 1.9: Ho-Kashyap with varying learning rates.

The results for a learning rate of $\frac{1}{k}$ were 136 misclassifications and 12,259 correct classifications. The results for a learning rate of $\frac{1}{k^2}$ were 137 misclassifications and 12,528 correct classifications. The results for a learning rate of $\frac{1}{\sqrt{k}}$ were 137 misclassifications and 12,528 correct classifications. The results for a learning rate of $\frac{1}{\sqrt[3]{k}}$ were 136 misclassifications and 12,529 correct classifications. All learning rates achieved noticeably different boundaries. However, the classification results for all learning rates were extremely similar. The impact of learning rate does not appear to be as significant in the Ho-Kashyap procedure as it is in the perceptron procedures.

With the learning rate and initialization tests complete, the best results are achieved with a learning rate of 1/k and an initialization of all ones. The solution weight vector was **a** = [-2.5364, 0.4654, 0.2238]. The training data results for this solution was 136 misclassifications and 12,529 correct classifications. This is a correct classification rate of 98.93%. The results for this solution weight vector on the test data was 18 misclassifications and 2,097 correct classifications. This is a correct classification rate of 99.15%. The decision boundary is shown with the feature space scatter plot for both the training and test images below.

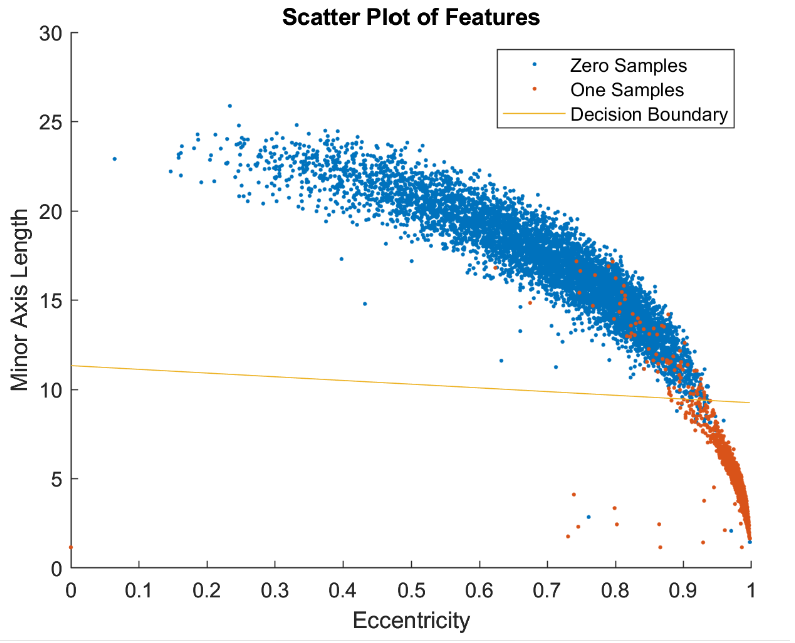

Figure 1.10: Best decision boundary with training samples.

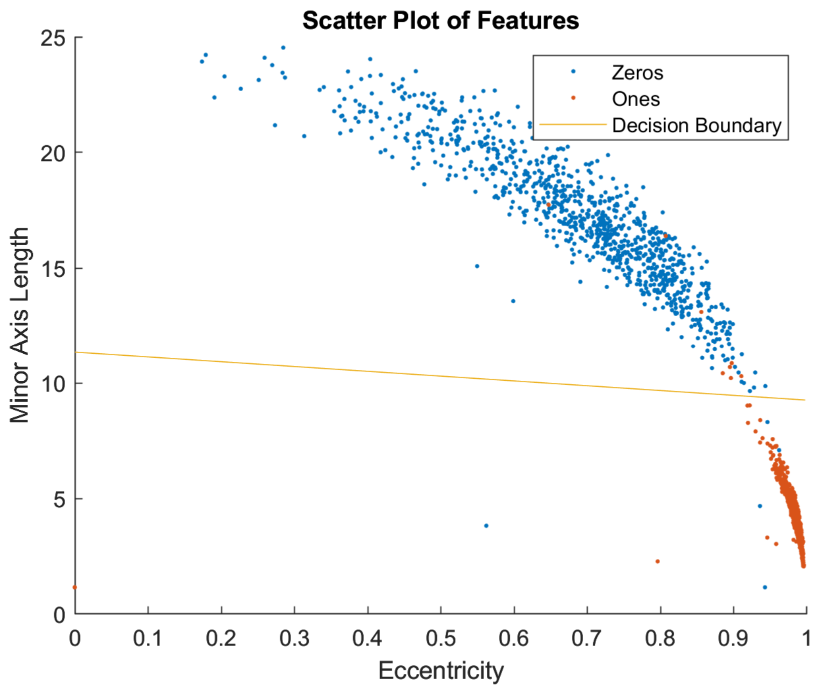

Figure 1.11: Best decision boundary with test samples.

The classification results with the Ho-Kashyap procedure are also excellent. It maintains a correct classification rate of ~99% for both the training and test samples.

Support Vector Machine

The next part of this project solves this classification problem with another technique. Here we use the MNIST training data to train a Support Vector Machine (SVM) that can classify new data as either a 0 or 1. To train a SVM model we can use the built in MATLAB function **fitcsvm**(). Although this function cannot take our augmented matrix as an argument, we slightly modify it and make it usable for this function. One of the acceptable ways to pass data to this function is through a matrix of predictors and a column vector of labels. The predictors in this case are our features, where each feature is a column, so the matrix of predictors will be a nx2 matrix while the labels vector is a nx1. To create the predictor matrix we can just use the last two columns of our augmented matrix and multiply them by the first column in the augmented matrix. To create our label vector, we just use the first column of the augmented matrix. We can pass these parameters in the

fitcsvm() function and in turn, receive a trained SVM model. Figure 1.12 below shows the training data, boundary line, and margins of the SVM.

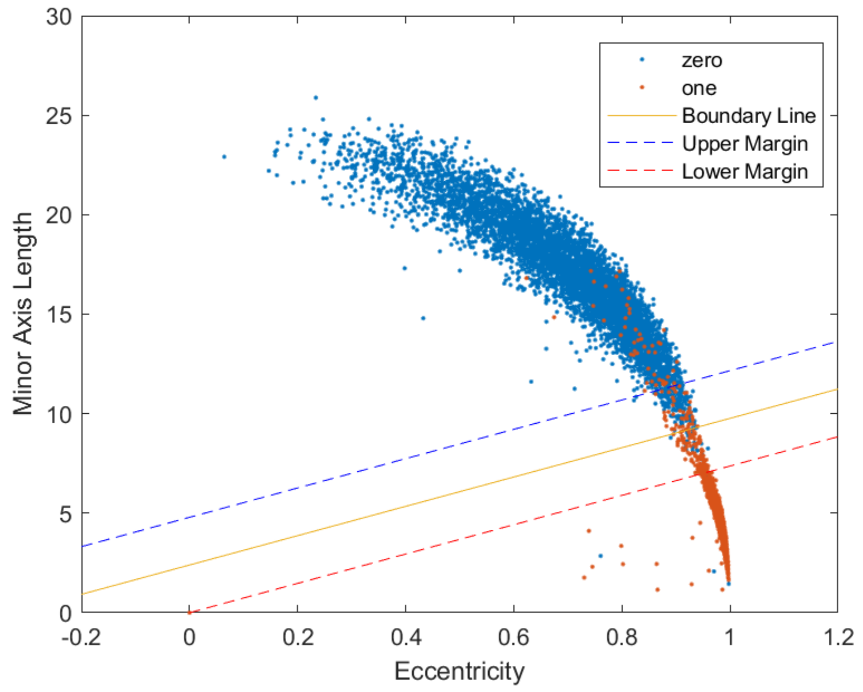

Figure 1.12: Creating 0 and 1 SVM Model with MNIST Training Data

To reduce clutter, we opted out of showing the support vectors in the plot above. In total there are 424 support vectors used in the model. Using this trained model we can now classify new data as either a 0 or 1 using the builtin MATLAB **predict**() function. The first argument we pass to it is the SVM model we created previously and the second is a predictor matrix with new data we want to classify. In this case we will use the MNIST test data for 0s and 1s and create it the same way we had with the training data. The **predict**() function returns a column vector which contains the labels assigned to the data fed to it where each row is a different sample. To find the amount of misclassified samples, we compare this label vector to our known classifications of the test data. The results of the classification of the SVM model can be seen in Figure 1.13 below.

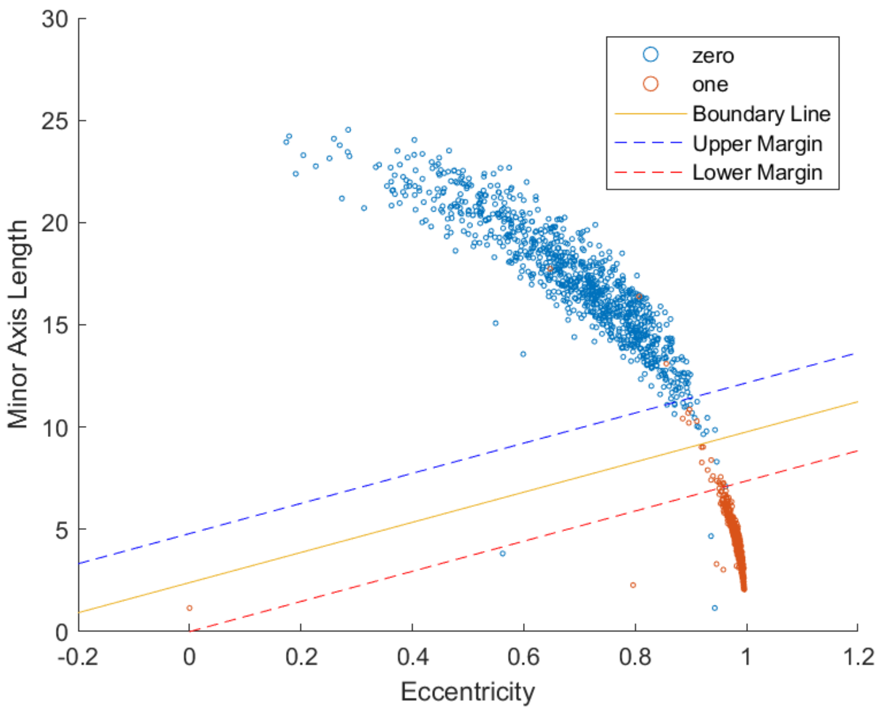

Figure 1.13: Classifying 0 and 1 Data with SVM Model.

Here we can see that the model performs well maximizing the margin between the boundary and in classifying the data. The amount of samples in the test set total up to 2,115. The amount of misclassifications from the test set total up to 18. Therefore the SVM has an error rate of 0.85%.

The previous results were using a linear kernel and C-parameter of 1 when training the SVM model. These are the defaults for a two-class classification when using **fitcsvm**(). The kernel and C-parameters used during training can be specified in this function, however in our case it did not appear to lead to different results. Changing the C-parameter from 1 to 10, 100, 1000, 1e6 didn't noticeably affect performance of the SVM, just the time required to complete the training of the model. All resulted in the same amount and rate of misclassification of 18 and 0.85%. When changing the kernel function, it is a similar story. The results stay the same with a misclassification count and rate of 18 and 0.85% respectively. These are believed to be the results because there is so much training data and support vectors that irrespective of what method, the final result of the SVM training reaches a similar point.

Another example of a feature we could use to classify the data as 0s or 1s is the data itself. Each sample in the MNIST set is a 28x28 pixel image of a digit. We can take the rows of a sample and concatenate them into a larger 1x784 row vector. Then going through each sample in the data set we can create a nx784 feature matrix. This will be our new predictor matrix used for training the SVM. Using the same functions as described before we can train our model and then classify new data with it. Using a nx784 feature matrix results in a misclassification count of 14 which is a 0.652% error rate. This reduces the error while also reducing complexity of computation. It does increase the amount of data we use for the SVM and make it so that there are too many dimensions and we can no longer visualize it.

For Zeros and Twos:

The same process for the zeros and ones digit samples is repeated for the zeros and twos digit samples. In project 1, the optimal features to separate zeros and twos were Circularity and Perimeter. The **Y** matrix is built with the new training samples using these two features.
Note: The second entries of the legends on the Figures of this section should all say 'Twos Samples'. Some of the plots take several minutes to create

First the single-sample perceptron procedure is used on the new **Y** matrix. The learning rate is set to a constant $\frac{1}{k}$ but the initial weight vector is varied. The various decision boundaries are shown in Figure 2.1. The first initialization (0, 0, 0) produced a boundary that gave 1,028 misclassifications and 10,838 correct classifications. The second, third, and fourth initializations produced boundaries that gave 792, 1,585, and 808 misclassification, respectively. The second initialization produced the best results in this case. However, different runs gave slightly different results due to the row shuffling. In certain cases the first and fourth initializations produced the best solutions. The fourth initialization never proved to be the optimal solution. Bad initializations can lead to decision boundaries with high misclassification rates. The all zeros initialization seems to be the safest initialization as it always produces a boundary with a reasonably low misclassification rate.

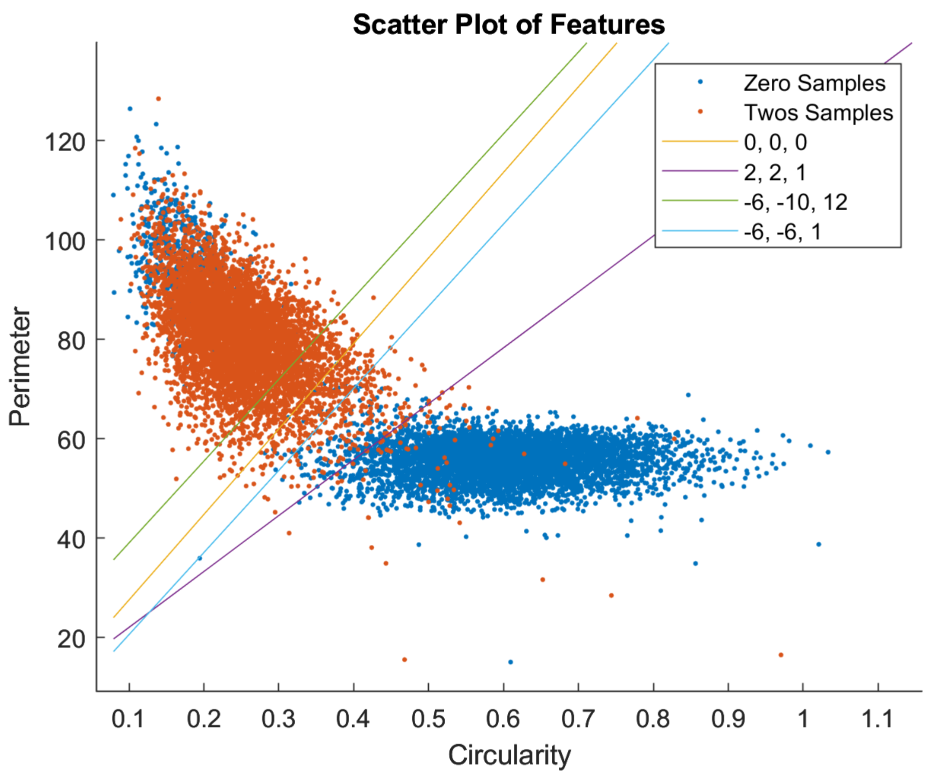

Figure 2.1: Single-sample perceptron with varying initial conditions.

The batch perceptron procedure is used with the **Y** matrix. The learning rate is set to a constant $\frac{1}{k}$ but the initial weight vector is varied. The various decision boundaries are shown in Figure 2.2. By inspection of Figure 2.2, some of the initializations lead to solutions that do not do a good job of separating the clusters. The initializations with larger magnitudes tend to perform worse. The initializations with smaller magnitudes tend to perform better. This is supported by observing the number of misclassifications for each decision boundary. The first weight vector and boundary leads to 1,338 misclassifications and 10,528 correct classifications. (Note: first refers to the boundary produced by the first initialization vector in the legend of Figure 2.2. Second refers to the second boundary in legend …) The second boundary leads to 1,342 misclassifications. The third and fourth boundaries lead to 8,869 and 4,564 misclassifications, respectively. The third and fourth initializations led to bad solutions. If no information is known about the shape of a good decision boundary, it is better in most situations to initialize with a weight vector close to zero. If there was prior information about a good decision boundary, this good boundary could be used to set the initialization. That could lead to better results than an all-zero initialization.

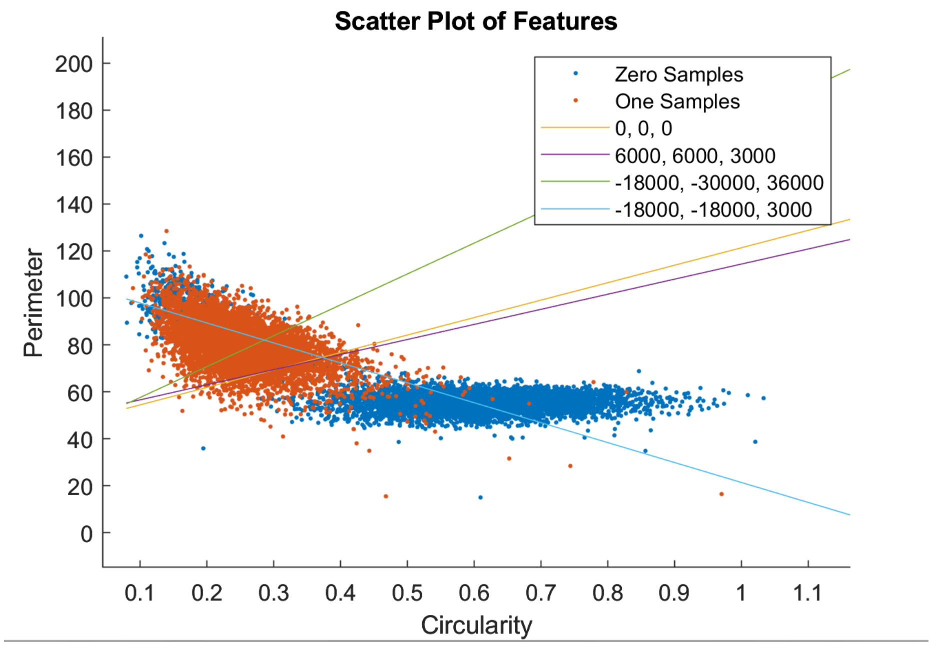

Figure 2.2: Batch perceptron with various initial conditions (for zeros and twos). Note: There is a typo in the legend. 'One Samples' should say 'Two Samples'.

The effects of varying learning rate are explored in the zeros and twos classifications. The initial weight vector is held constant as all zeros. The same learning rates are explored (same as for zeros and ones classifications). The results for single-sample perceptron with the various learning rates are shown in Figure 2.3. The first learning rate (the order follows that of the legend in Figure 2.3) produced a boundary that gave 804 misclassifications and 11,062 correct classifications. The second, third, and fourth learning rates produced boundaries that gave 5,910, 848, and 812 misclassifications, respectively. The $1/k^2$ learning rate again proved to be the worst. The drop-off is too quick and lets the procedure converge to bad solutions. The other three learning rates all produced noticeably different boundaries but had very similar classification results. These learning rates have slow enough drop-offs to allow the procedure to converge at a reasonable solution.

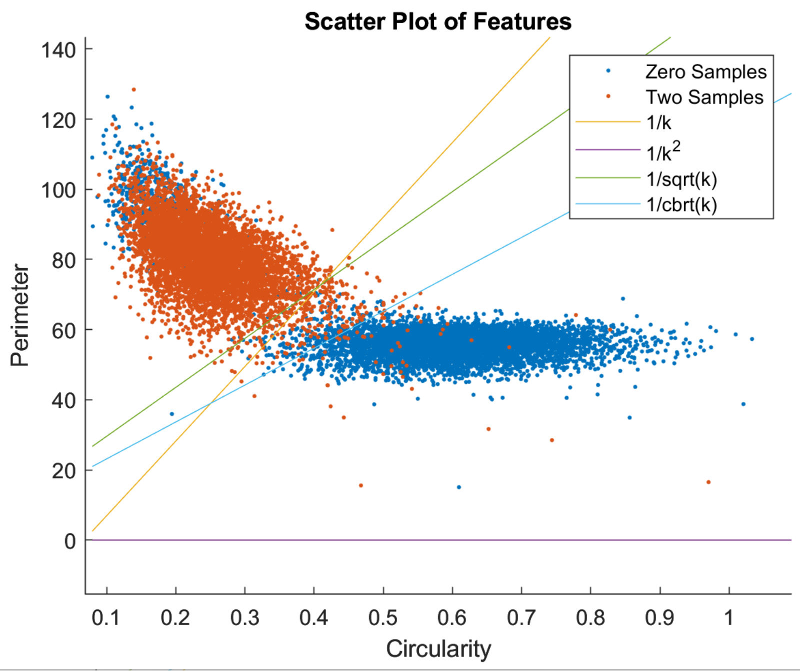

Figure 2.3: Single-sample perceptron with varying learning rates.

The results for batch perceptron with the various learning rates are shown in Figure 2.4. The first learning rate (the order follows that of the legend in Figure 2.4) produced a boundary that gave 1340 misclassifications and 10526 correct classifications. The second, third, and fourth learning rates produced boundaries that gave 1340, 819, and 728 misclassifications, respectively. The results show mostly the same trends. However, in the batch perceptron, the $\frac{1}{k^2}$ learning rate shows significantly better results than in the single-sample perceptron. This is likely due to the batch perceptron doing larger updates at each iteration. The $\frac{1}{k}$ and $\frac{1}{k^2}$ show similar results. The learning rate with the slowest drop-off, 1/cbrt(k), has the best results in this case.

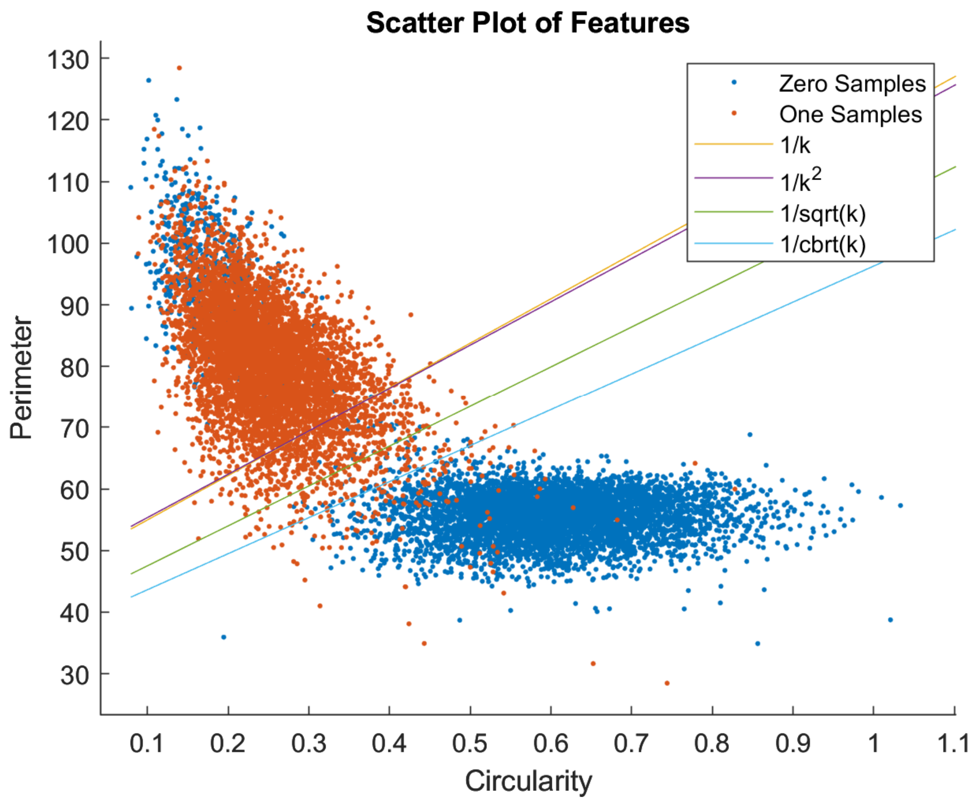

Figure 2.4: Batch Perceptron with varying learning rates. Note: There is a typo in the legend. ‘One Samples’ should say ‘Two Samples’.

After testing with varying initializations, the best classifiers were taken. The optimal classifiers are applied to both the training data and the test data. Figure 2.5 shows the optimal classifiers for the batch and single-sample perceptron applied to the training data. The optimal classifier for the batch perceptron was achieved with an initial weight vector of all zeros and a learning rate of 1/cbrt(k). The optimal classifier for the batch perceptron was with a solution weight vector of **a** = [322086, 498390, -8521.48]. This classifier achieved 728 misclassifications and 11,138 correct classifications. This is a correct classification rate of 93.86%. The optimal classifier for the single-sample perceptron was achieved with an initial weight vector of [2, 2, 1] and a learning rate of 1/sqrt(k). The optimal classifier for the single-sample perceptron was with a weight vector of **a** = [1.2069, 12.6065, -0.1072]. This classifier achieved 741 misclassifications and 11,125 correct classifications. This is a correct classification rate of 93.76%. Both classifiers achieve good classification results. The ~94% correct classification rate is good, considering the noticeable overlap between the clusters of the scatter plot.

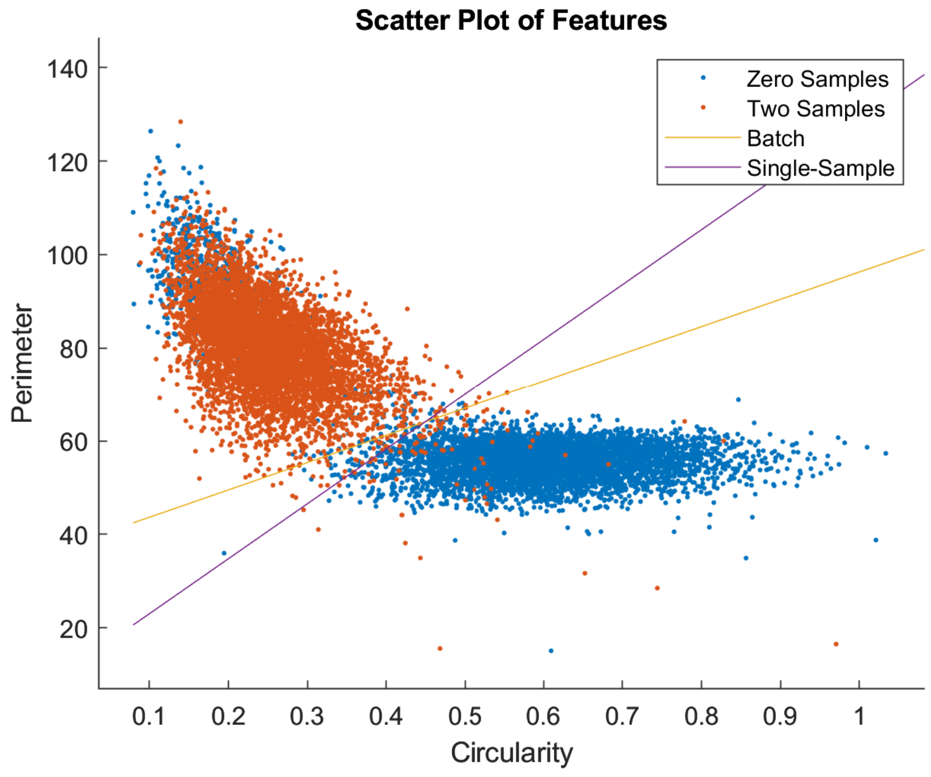

Figure 2.5: Optimal classifiers for batch and single-sample procedure with training data.

These optimal classifiers are now applied to the test data. The results are shown in Figure 2.6. The batch perceptron boundary produced 141 misclassifications and 1,871 correct classifications. This is a correct classification rate of 92.99%. The single-sample perceptron boundary produced 153 misclassifications and 1,859 correct classifications. This is a correct classification rate of 92.40%. Both boundaries again produced good solutions. The ~93% correct rate is good considering the two clusters in the feature space have considerable overlap.

Many of the same trends for the batch and single-sample perceptron noted in the zeros and ones section also hold true in the zeros and twos section. The single-sample perceptron tends to be more sensitive to changes in initialization and learning rate. The single-sample perceptron can also tend to be inconsistent. One run it may produce great results; another run it may produce results with a 50%

misclassification rate. The batch perceptron takes significantly longer to run and produce a result, but the results of this procedure are consistently good. In the zeros and twos classifications, the batch perceptron proved to perform better overall. Based on this, it seems that the batch perceptron is a better procedure when the clusters have stronger overlaps.

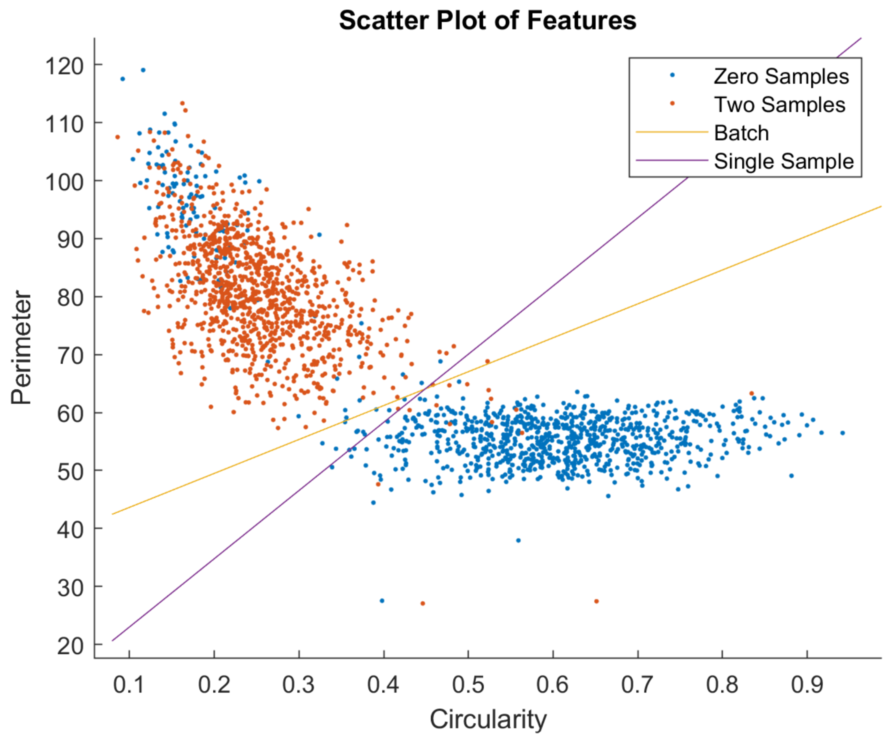

Figure 2.6: Optimal classifiers for batch and single-sample perceptron with test data.

The next step in the project is to use the Ho-Kashyap procedure for the zeros and twos classification. Various initial margin vectors were used. Similar to the zeros and ones section, the all ones initialization showed the best results. The Ho-Kashyap procedure tends to show large variations in the results with different initial margin vectors. The Ho-Kahsyap was also tested with the various learning rates used previously in this project. Similar to the zeros and ones section, the Ho-Kashyap procedure proves to not be very sensitive to learning rate. All four learning rates tested produced nearly the exact same classification results. The determined optimal classifier for the Ho-Kashyap was determined to be achieved using an initial margin vector of all ones and a learning rate of 1/k. This solution had a weight vector of **a** = [0.5654, 2.2374, -0.0211]. Figure 2.7 shows the decision boundary created with this classifier. The optimal classifier achieved 807 misclassifications and 11,059 correct classifications. This is a correct classification rate of 93.20%.

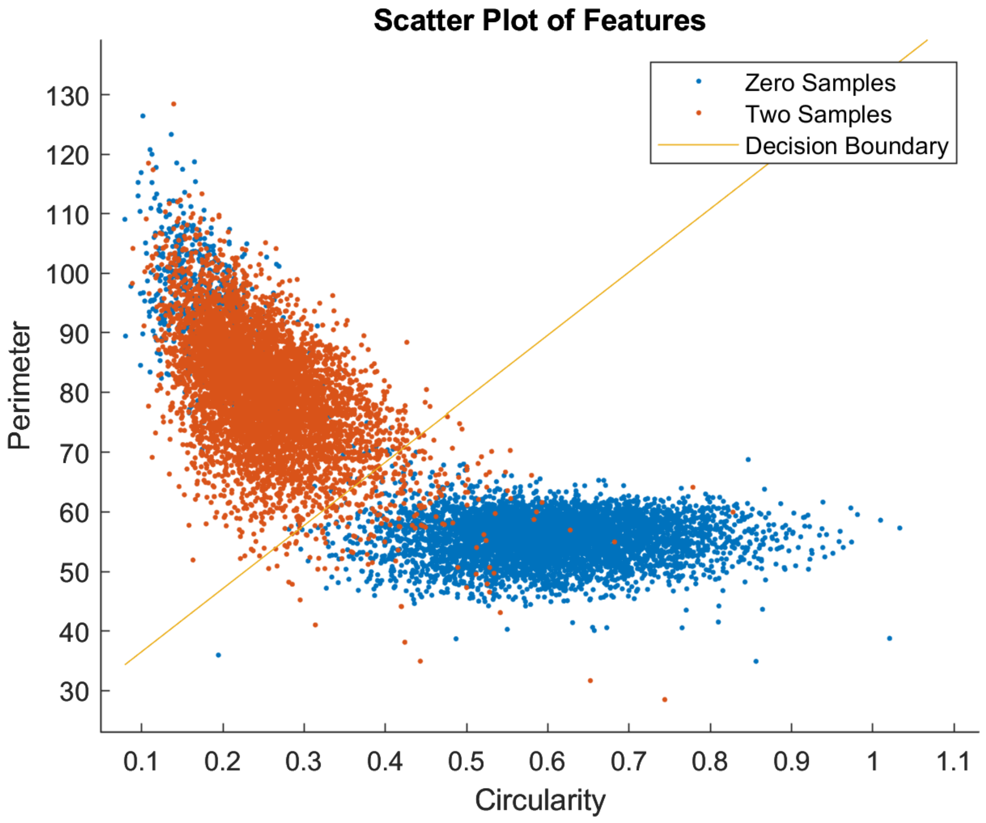

Figure 2.7: Optimal classifier for training data with Ho-Kashyap procedure.

This decision boundary was also applied to the training data. Figure 2.8 shows the test data with the decision boundary. The classifier achieved 152 misclassifications and 1,860 correct classifications. This is a correct classification rate of 92.45%. The single-sample perceptron, batch perceptron, and Ho-Kashyap procedures all produced very similar classification results (~92 to 93% correct rate). Each procedure produced good results in the zeros and twos classification. Due to the similar results, it is difficult to say which performed the best. However, due to the consistency of the batch perceptron and Ho-Kashyap, these procedures seem to be the best in the zeros and twos classification.

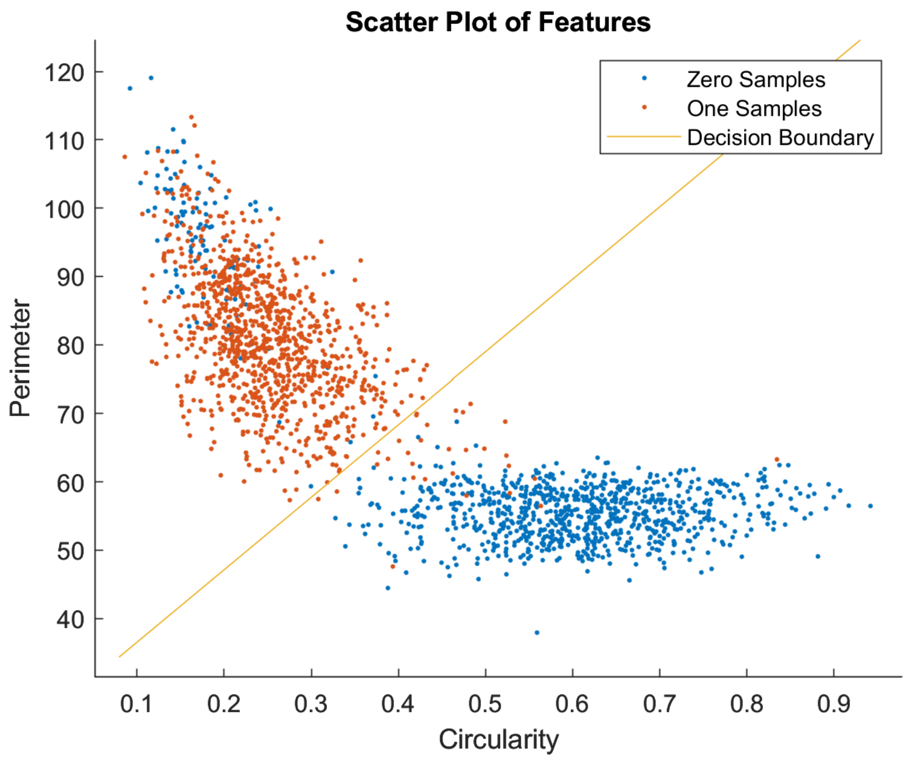

Figure 2.8: Optimal classifier with test data for Ho-Kashyap procedure.

Support Vector Machine

We once again used the same procedures as in the 0s and 1s SVM analysis but with 0s and 2s. When feeding the circularity and perimeter data into the SVM generator, we get the boundary line and margins shown below in Figure 2.9.

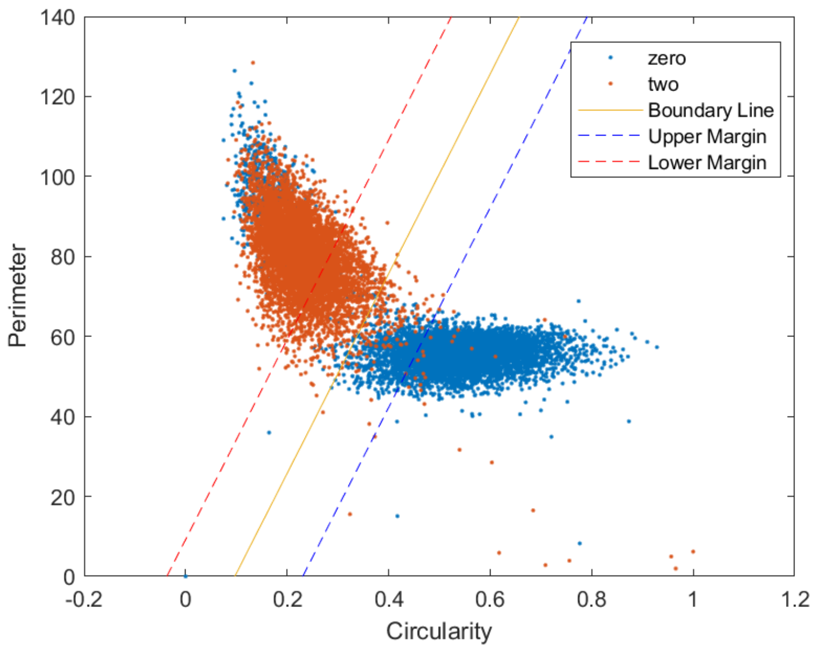

Figure 2.9: Creating 0 and 2 SVM model with MNIST Training Data.

This model was used to classify the MNIST test data for 0s and 2s using the same methods described to classify 0s and 1s previously. The results are shown below in Figure 2.10.

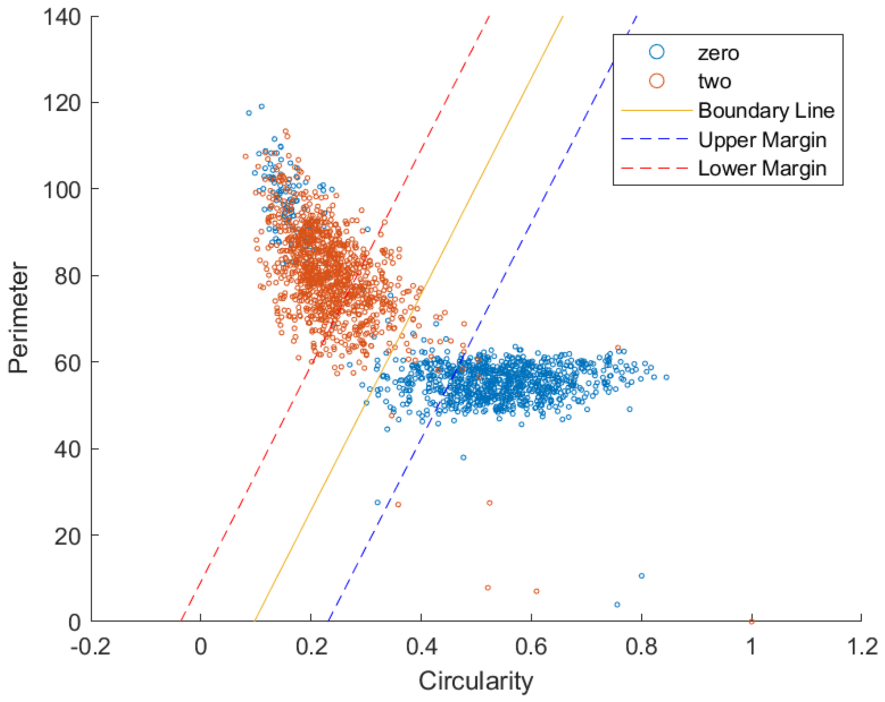

Figure 2.10: Classifying 0 and 2 test data with SVM Model.

The two characteristic SVM model contains 157 misclassifications out of 2,048 total samples, resulting in a 7.66% error rate. This result is actually worse than some of the previous methods, which is unexpected. The outlier zeroes may have more of an effect in SVM analysis than they did in the other methods.

The raw data feature matrix was created, used to train a SVM model and produced promising results when classifying data. When used to classify the test data, the model misclassified 41 samples out of a total of 2,048. Therefore the model had an error rate of 2.002%. This is promising, considering 0s and 2s had features that are similar and make them harder to distinguish from each other than 0s and 1s.

Reflection

Tre Carmichael

Choosing the correct characteristics is critical when creating a classification algorithm. We used a variety of algorithms ranging from very simple to very computing intensive and the most important factor was how well the characteristics served their purpose of differentiating the classes. For zeroes and ones, the single-sample Perceptron algorithm had a correct classification rate of 98.78%. For zeroes and twos, the two-characteristic SVM model had a correct classification rate of 92.34%. The simplest zeroes and ones algorithm beat the most complex zeroes and twos algorithm because the zeroes and ones characteristics were nearly ideal while the zeroes and twos characteristics had significant overlap. With that being said, the raw data SVM model performed excellently in both scenarios since it was effectively using over 700 “characteristics” to classify data.

Nickolas Ogilvie

With well-picked features, excellent classifiers can be created. With just the two features, the zeros and ones classifier achieved a correct classification rate of ~99%. In the zeros and twos case, the classifier was able to achieve a correct classification rate of ~93% even though the feature space clusters seemed to show significant overlap. Another takeaway is that the perceptron and Ho-Kashyap procedures are very sensitive to the initialization. With well-picked initializations, all three of these procedures produced excellent decision boundaries. However, with bad initializations, the procedures can produce decision boundaries with high misclassification rate. It can be difficult to set good initializations. In this project, using basic initializations proved to be consistent. Basic initializations were all-zero or all near-zero components in the perceptron procedure. In the Ho-Kashyap the basic initialization was the all ones vector.

Nathan Jaggers

There are many tools at our disposal for accomplishing the objective of pattern recognition. I think what this project shows is how those tools perform, their benefits and pitfalls. In single sample training, we had to randomize the samples because if the training was blocked out in chunks for one digit, then the other, by the time you get to the end of the training data the learning rate is so small you are making significant enough changes for class 2. This is a problem that could be avoided in batch training, however batch training is computationally intensive and takes a while. When conducting the SVM training and classification, although using the two features we had used previously was familiar and intuitive, it did take a while to run and was not as effective as using the raw pixel data. That being said we could still visualize what was happening with just two features but that is impossible for a 784 dimension feature vector.

Teamwork

Tre Carmichael

Worked on batch perceptron, single-sample perceptron, 2-characteristic SVM, and report.

Nickolas Ogilvie

Worked on batch perceptron, single-sample perceptron, and Ho-Kashyap code and report.

Nathan Jaggers

Worked on single sample and SVM code concurrently with Tre. Wrote SVM 0s and 1 section of the report.

Appendix

MATLAB live script to do single-sample, batch perceptron, and Ho-Kashyap tests for digit classification

```
% Perform various classifier tests.

% To swap between script working with 0s and 1s or 0s and 2s
% use find and replace to replace all 'mnist_train1.jpg' with
% 'mnist_train2.jpg'. Also 'mnist_train1.jpg' with 'mnist_test1.jpg'.
% Also change 'Eccentricity' to 'Circularity'. Change 'MinorAxisLength' to
% 'Perimeter'.
% Load training samples for zeros.
im = imread('mnist_train0.jpg');

% Binarize the image with threshold determined in Project 1.
T = 100;
im_bin = im > T;

% Extract feature vectors from samples
vec_0_ecc = extract_feature_vector(im_bin, 'Circularity');
vec_0_minor = extract_feature_vector(im_bin, 'Perimeter');

% Some samples had circularity of infinity for some reason. Remove these inf values.
inf_index = find(vec_0_ecc == inf);
vec_0_ecc(inf_index) = [];
vec_0_minor(inf_index) = [];

% Load one or twos samples image
im = imread('mnist_train2.jpg');

% Apply the threshold to the image and create the binarized image
T = 100;
im_bin = im > T;

% Extract feature vectors from the image.
vec_1_ecc = extract_feature_vector(im_bin, 'Circularity');
vec_1_minor = extract_feature_vector(im_bin, 'Perimeter');

% There were some circularities of infinity. Remove these training samples.
inf_index = find(vec_1_ecc == inf);
vec_1_ecc(inf_index) = [];
vec_1_minor(inf_index) = [];
% Generate the matrix of augmented training samples. Zeros will be
% considered the positive class. Ones will be considered the negative class
% such that the ones rows are multiplied by -1.
% The matrix has 3 columns. Column one is the class identifier for the
% sample (1 or -1). Column 2 is the first feature used. Column 3 is the
% second feature. Each row of Y is the augmented feature vector for a
% sample.
Y = [ones(length(vec_0_ecc), 1), vec_0_ecc', vec_0_minor'; -1*ones(length(vec_1_ecc), 1),
-1*vec_1_ecc', -1*vec_1_minor'];


% Find the weighting vector with batch perceptron.
```

```
a = single_sample_perceptron(Y, [0; 0; 0], 100000)

% Now want to plot decision boundary with scatter plot.
% g(x) = a(1) + a(2)*x1 + a(3)*x2; Boundary at g(x) = 0.
% Solve for x2 to find the boundary line equation.

% Define range of values for x1 (the range is the minimum value the x1
% feature ever takes to the maximum the x1 feature ever takes.
x1_min = min([vec_0_ecc, vec_1_ecc]);
x1_max = max([vec_0_ecc, vec_1_ecc]);

% Generate a vector with 100 linearly spaced values.
x1_vals = linspace(x1_min, x1_max, 100);

% Plug x1_vals with the weighting vector to find the equation of the line.
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);

% Create a scatter plot with the feature vectors
figure
hold on
scatter(vec_0_ecc, vec_0_minor, '.');
scatter(vec_1_ecc, vec_1_minor, '.');
plot(x1_vals, x2_vals)
hold off
xlabel('Circularity')
ylabel('Perimeter')
title('Scatter Plot of Features')
xlim([0, 2])



% Test with varying initial conditions.
% This section is used for both single sample and batch perceptron testing.
% Just subsitute the function calls with the correct procedure as needed.
%Run single sample perceptron stuff with various initializations

% Define initial weight vector and call perform procedure.
init_a0 = [0, 0, 0]';

a = single_sample_perceptron(Y, init_a0, 1000000)
scale = 3000;
% Determine the number of misclassifications and correct classifications.
% Mulitply the Y matrix with the weight vector output by the procedure. Any
% negative value is a misclassification. Count the number of negative
% values in the product to find the number of misclassifications. The
% number of positive values is the number of correct classifications.
num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)

% Create the equation of the line described by the weight vector a.
% This is done so that the decision boundary can be drawn on the scatter
% plot.
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);

% Plot scatter plot and first decision boundary.
figure
hold on
scatter(vec_0_ecc, vec_0_minor, '.');
scatter(vec_1_ecc, vec_1_minor, '.');
plot(x1_vals, x2_vals)
```

```
xlabel('Circularity')
ylabel('Perimeter')
title('Scatter Plot of Features')

% Call procedure with new initialization and plot resulting decision boundary.
init_a1 = [2, 2, 1]'%*scale
a = single_sample_perceptron(Y, init_a1, 1000000)
num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);
plot(x1_vals, x2_vals)

% Call procedure with new initialization and plot resulting decision boundary.
init_a2 = [-6, -10, 12]'%*scale % Uncomment the *scale for batch_perceptron
a = single_sample_perceptron(Y, init_a2, 1000000)
num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);
plot(x1_vals, x2_vals)

% Call procedure with new initialization and plot resulting decision boundary.
init_a3 = [-6, -6, 1]'%*scale
a = single_sample_perceptron(Y, init_a3, 1000000)
num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);
plot(x1_vals, x2_vals)
legend('Zero Samples', 'One Samples', ...
    strjoin(string(init_a0), ", "), strjoin(string(init_a1), ", "), ...
    strjoin(string(init_a2), ", "), strjoin(string(init_a3), ", "))
hold off
xlim([0, 2])



% Various Learning Rates
% Go into single_sample_perceptron (or batch_perceptron) and manually adjust learning rate.
% Run this section 4 times. Each time change the learning rate and
% change the name of x2_vals. On the fourth run, uncomment the plotting
% code at the end of this section.
init_a0 = [0, 0, 0]';
a = batch_perceptron(Y, init_a0, 10000)

num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals_3 = (-a(1) - a(2).*x1_vals)/a(3);

figure
hold on
scatter(vec_0_ecc, vec_0_minor, '.');
scatter(vec_1_ecc, vec_1_minor, '.');
plot(x1_vals, x2_vals_0)
plot(x1_vals, x2_vals_1)
plot(x1_vals, x2_vals_2)
plot(x1_vals, x2_vals_3)
xlabel('Circularity')
ylabel('Perimeter')
title('Scatter Plot of Features')
legend('Zero Samples', 'One Samples', '1/k', '1/k^2', '1/sqrt(k)', '1/cbrt(k)')
hold off
```

```
% Tests for Optimal Classifiers
% Change the function between batch_perceptron and single_sample_perceptron
% in this section to find the best classifier for each.
% Define initial weight vector and call perform procedure.
init_a0 = [0, 0, 0]';
a = batch_perceptron(Y, init_a0, 10000)
a1 = a;
num_misclass = sum((Y*a)  < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals_0 = (-a(1) - a(2).*x1_vals)/a(3);

init_a0 = [2, 2, 1]';
a = single_sample_perceptron(Y, init_a0, 1000000)
a2 = a;
num_misclass = sum((Y*a)  < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals_1 = (-a(1) - a(2).*x1_vals)/a(3);

% Plot scatter plot and decision boundaries.
figure
hold on
scatter(vec_0_ecc, vec_0_minor, '.');
scatter(vec_1_ecc, vec_1_minor, '.');
plot(x1_vals, x2_vals_0)
plot(x1_vals, x2_vals_1)
xlabel('Circularity')
ylabel('Perimeter')
title('Scatter Plot of Features')
hold off
legend('Zeros', 'Ones', 'Batch', 'Single-Sample')

% Run classifier with test images
% Build Y matrix for test image samples.
im = imread('mnist_test0.jpg');

% Binarize the image with threshold determined in Project 1.
T = 100;
im_bin = im > T;

vec_0_ecc = extract_feature_vector(im_bin, 'Circularity');
vec_0_minor = extract_feature_vector(im_bin, 'Perimeter');

im = imread('mnist_test2.jpg');

% Apply the threshold to the image and create the binarized image
T = 100;
im_bin = im > T;

% Extract feature vectors from the ones image.
vec_1_ecc = extract_feature_vector(im_bin, 'Circularity');
vec_1_minor = extract_feature_vector(im_bin, 'Perimeter');


Y_test = [ones(length(vec_0_ecc), 1), vec_0_ecc', vec_0_minor'; -1*ones(length(vec_1_ecc),
1), -1*vec_1_ecc', -1*vec_1_minor'];

num_misclass = sum((Y_test*a1) < 0)
num_correct_class = sum((Y_test*a1) > 0)
```

```
x2_vals_0 = (-a1(1) - a1(2).*x1_vals)/a1(3);

num_misclass = sum((Y_test*a2) < 0)
num_correct_class = sum((Y_test*a2) > 0)
x2_vals_1 = (-a2(1) - a2(2).*x1_vals)/a2(3);

% Plot scatter plot and decision boundary.
figure
hold on
scatter(vec_0_ecc, vec_0_minor, '.');
scatter(vec_1_ecc, vec_1_minor, '.');
plot(x1_vals, x2_vals_0)
plot(x1_vals, x2_vals_1)
xlabel('Circularity')
ylabel('Perimeter')
title('Scatter Plot of Features')
legend('Zeros', 'Ones', 'Batch', 'Single Sample')



% Ho-Kashyap w/ Various Inits
[M, N] = size(Y);

b0 = ones(M, 1);
[a, b] = ho_kashyap([0, 0, 0]', b0, Y, 10000);
num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);


% Plot scatter plot and first decision boundary.
figure
hold on
scatter(vec_0_ecc, vec_0_minor, '.');
scatter(vec_1_ecc, vec_1_minor, '.');
plot(x1_vals, x2_vals)
xlabel('Circularity')
ylabel('Perimeter')
title('Scatter Plot of Features')

% Call procedure with new initialization and plot resulting decision boundary.
b1 = linspace(0.1, 10, M)';
[a, b] = ho_kashyap([0, 0, 0]', b1, Y, 10000);
num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);
plot(x1_vals, x2_vals)

% Call procedure with new initialization and plot resulting decision boundary.
b2 = 1:M;
b2 = b2';
[a, b] = ho_kashyap([0, 0, 0]', b2, Y, 10000);
num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);
plot(x1_vals, x2_vals)

% Call procedure with new initialization and plot resulting decision boundary.
b3 = abs(randn(M, 1));
[a, b] = ho_kashyap([0, 0, 0]', b3, Y, 10000);
```

```
num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);
plot(x1_vals, x2_vals)
legend('Zero Samples', 'One Samples', 'all ones', '0.1 to 10', '1:M', 'abs(randn())')

hold off
legend('Zero Samples', 'Two Samples', 'Decision Boundary')
```

```
%% Ho-Kashyap Various Learning Rates

% Go into ho_kashyap and manually adjust learning rate. Run this section 4 times. Each time
% change the learning rate and change the name of x2_vals. On the fourth run, uncomment the
% plotting code at the end of this section.

[a, b] = ho_kashyap([0, 0, 0]', ones(M, 1), Y, 10000);

num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);

figure
hold on
scatter(vec_0_ecc, vec_0_minor, '.');
scatter(vec_1_ecc, vec_1_minor, '.');
plot(x1_vals, x2_vals_0)
plot(x1_vals, x2_vals_1)
plot(x1_vals, x2_vals_2)
plot(x1_vals, x2_vals_3)
xlabel('Circularity')
ylabel('Perimeter')
title('Scatter Plot of Features')
legend('Zero Samples', 'One Samples', '1/k', '1/k^2', '1/sqrt(k)', '1/cbrt(k)')
hold off
```

```
%% Ho-Kashyap Optimal Classifier Tests

% Run ho_kashyap with determined optimal init and learning rate.

[M, N] = size(Y);
[a, b] = ho_kashyap([0, 0, 0]', ones(M, 1), Y, 100000);

num_misclass = sum((Y*a) < 0)
num_correct_class = sum((Y*a) > 0)
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);

figure
hold on
scatter(vec_0_ecc, vec_0_minor, '.');
scatter(vec_1_ecc, vec_1_minor, '.');
plot(x1_vals, x2_vals)
xlabel('Circularity')
```

```
ylabel('Perimeter')
title('Scatter Plot of Features')
legend('Zero Samples', 'One Samples', 'Decision Boundary')
hold off


% Build Y matrix for test image samples.
im = imread('mnist_test0.jpg');

% Binarize the image with threshold determined in Project 1.
T = 100;
im_bin = im > T;

vec_0_ecc = extract_feature_vector(im_bin, 'Circularity');
vec_0_minor = extract_feature_vector(im_bin, 'Perimeter');

im = imread('mnist_test2.jpg');

% Apply the threshold to the image and create the binarized image
T = 100;
im_bin = im > T;

% Extract feature vectors from the ones image.
vec_1_ecc = extract_feature_vector(im_bin, 'Circularity');
vec_1_minor = extract_feature_vector(im_bin, 'Perimeter');


Y_test = [ones(length(vec_0_ecc), 1), vec_0_ecc', vec_0_minor'; -1*ones(length(vec_1_ecc),
1), -1*vec_1_ecc', -1*vec_1_minor'];

% Apply classifier to test data.
num_misclass = sum((Y_test*a) < 0)
num_correct_class = sum((Y_test*a) > 0)
x2_vals = (-a(1) - a(2).*x1_vals)/a(3);


% Plot scatter plot and decision boundary for test data.
figure
hold on
scatter(vec_0_ecc, vec_0_minor, '.');
scatter(vec_1_ecc, vec_1_minor, '.');
plot(x1_vals, x2_vals)
xlabel('Circularity')
ylabel('Perimeter')
title('Scatter Plot of Features')
legend('Zeros', 'Ones', 'Decision Boundary')
```

MATLAB script to iterate over digit samples and extract feature vectors.

```
function feature_vector = extract_feature_vector(im, field)
% Input the binarized image
% Input the field to extract for region props. This input should be a
% character array (like: 'Eccentricity').
feature_vector = [];
index = 0;
% Determine size of the image. To index over each possible segment, the
% final for loop iteration should be equal to the size divided by 28 (since
% the segments are 28x28) and minus one (since I am starting the for loop
% iterations at zero).
[M, N] = size(im);
a_last = M/28 - 1;
b_last = N/28 - 1;
% Iterate over each possible 28x28 segment in the input image.
for a = 0:a_last
   for b =0:b_last
       % Extract the correct 28x28 segment for the current a and b
       % indices.
       im_segment = im((a*28+1):(a*28+28), (b*28+1):(b*28+28));
       % Extract the feature specified by the field input.
       feature = regionprops(im_segment, field);
       % Want to check if the segment is all black (meaning there is
       % no sample at the current im_segment. To do this, sum up all
       % values in the segment. If the sum is not greater than some low
       % number, the segment should be ignored.
       im_segment = double(im_segment); % Need to convert to double to sum.
       segment_sum = sum(im_segment(:));
       if(segment_sum > 5)
           % use getfield to extract the actual number value from the
           % structure data type output by reigonprops()
           feature = getfield(feature, field);
           % Increment the position in the feature vector and place the
           % calculated feature value at the end of the feature vector.
           index = index + 1;
           feature_vector(index) = feature;
       end
   end
end
end
```

MATLAB script implementing single-sample perceptron

```
function a = single_sample_perceptron(Y, a1, kmax)
% Input the matrix of augmented training samples Y, an initial weighting
% vector a1, and a maximum number of iterations kmax.
% Output a weight vector
a = a1;
[m, n] = size(Y);
% Shuffle the rows of the training sample matrix.
Y = Y(randperm(size(Y, 1)), :);
% Set iteration to zero.
k = 0;
while(k <= kmax)
   % k_mod is the row for the Y. Take mod (base m as the number of rows in
   % Y) of k to get the correct row index for Y.
   k_mod = mod(k, m) + 1;
   % Increment the iteration count.
   k = k+1;
   % Extract the row from Y which is the vector for a training sample.
   yk = Y(k_mod, :);
   % Find the dot product of the training sample row and weight vector.
   val = yk*a;
   % If the dot product is less than zero, then a missclassification
   % occured.
   if(val <= 0)
       % Update the weight vector based on the misclass. The learning rate
       % is currently set to 1/k.
       a = a + (1/sqrt(k))*yk';
   end
end
% After running iterations to kmax, output the final weight vector a.
end
```

MATLAB script implementing batch perceptron

```
function a = batch_perceptron(Y, a1, kmax)
% Input augmented training sample matrix, an initial weight vector, and the
% maximum number of iterations.
% Output a weighting vector determined with batch-perceptron procedure.
a = a1;
% Determine number of rows in Y.
[m, n] = size(Y);
theta = [0.001, 0.001, 0.001]; %?
% Iterate up to kmax times.
for k = 1:kmax
   % Initialize the sum of all misclassifications to be zero.
   sum_misclass = zeros(1, 3);

   % Iterate over row in Y (each row corresponds to a training sample).
   % At each row compute the value produced by the row dotted with the
   % current weighting vector a. Any dot product less than zero indicates
   % a missclassification. For any row with a dot product less than zero
   % add it to sum_misclass.
   for i = 1:m
       val = Y(i, :)*a;
       if(val <= 0)
           sum_misclass = sum_misclass + Y(i, :);
       end
   end
   % After finding the sum of all misclassified training samples, multiply
   % the result by the learning rate. Here the learning rate is set to
   % 1/k. The adjustment is the learning rate multiplied by the sum of the
   % misclassifications.
   adjustment = (1/nthroot(k, 3))*sum_misclass;
   % Adjust the weighting vector, a, with the calculated adjustment.
   a = a + adjustment';
   % If the adjustment at each position in the vector is less than some
   % predetermined theta value, consider the algorithm to have converged.
   % Exit the function and return the final weight vector a.
   if(abs(adjustment) < theta)
       disp(k)
       return
   end
end
% If the function has iterated kmax times without converting, exit the
% function and return the weighting vector for the final iteration.
disp(kmax)
end
```

MATLAB script implementing Ho-Kashyap Procedure

```
function [a, b] = ho_kashyap(a1, b1, Y, kmax)
% Implement the Ho Kashyap Procedure.
% Input a1 is initial weight vector.
% Input b1 is initial margin vector.
% Input Y is matrix of augmented training samples.
% Input eta is the learning rate. It is assumed to be constant in this
% function.
% Input kmax is the maximum number of iterations before deciding that no
% solution can be found.
% Output a is the final weight vector determined by the procedure.
% Output b is the final margin vector that produced this weight vector.
% Set a and b to be equal to specified initial vectors.
a = a1;
b = b1;
% Set bmin as the minimum value in the initial b vector.
bmin = min(b1);
% Iterate upto kmax times.
for k = 1:kmax
   % For the iteration, update the weight vector for this iteration using
   % the updated margin vector.
   a = inv(Y'*Y)*Y'*b;
   % With the a and b vectors for this iteration, compute the error
   % vector.
   e = Y*a-b;
   % If all components of the error vector have a smaller magnitude than
   % bmin, then a solution has been found. Exit the function and output
   % the current a and b vectors.
   if(abs(e) < bmin)
       % Display the number of iterations and error vector at the time of
       % finding the soluton.
       disp(k)
       disp(e)
       return
   end
   % If we did not find a solution and exit, update the b vector using the
   % previous b vector, the learning rate, and the error vector. This is
   % essentially calculating b(k+1).
   b = b + (1/k)*(e + abs(e));
end

end
```

MATLAB script implementing SVM

```
%%
%read in image
im_train0 = imread("mnist_train0.jpg");
im_train1 = imread("mnist_train1.jpg");
im_train2 = imread("mnist_train2.jpg");
%%
%create binary image and show results
binary_imtrain0 = imbinarize(im_train0);
%%imshow(binary_im0);
binary_imtrain1 = imbinarize(im_train1);
%%imshow(binary_im1);
binary_imtrain2 = imbinarize(im_train2);
%%imshow(binary_im2);
%%
features_train0 = get_features(binary_imtrain0);
features_train1 = get_features(binary_imtrain1);
features_train2 = get_features(binary_imtrain2);
%%
featarry_train0 = make_feat_array(features_train0);
featarry_train1 = make_feat_array(features_train1);
featarry_train2 = make_feat_array(features_train2);
%%
%initialize variables for training
%Y is augmented matrix holding (+/-)1's and two values to describe digit
%in each row
%0 & 1
class_1 = [ones(length(featarry_train0(:,3)),1) featarry_train0(:,3)
featarry_train0(:,7)];
class_2 = [-ones(length(featarry_train1(:,3)),1) -featarry_train1(:,3)
-featarry_train1(:,7)];
Y = [class_1; class_2];
%%
% the predictors in the matrix X and the class labels in vector Y
features = Y(:,1).*Y(:,2:3); % matrix X
classification = Y(:,1); % matrix Y
%% Train a linear kernel SVM classifier
SVMModel = fitcsvm(features,classification);
%default for two class classifier is linear kernel, for one class
%classifier it is gaussian (rbf - radial basis function) kernel
%%
%using mathworks example to create plot
sv = SVMModel.SupportVectors; % Support vectors
beta = SVMModel.Beta; % Linear predictor coefficients
b = SVMModel.Bias; % Bias term
%%
hold on
color = [0.8500 0.3250 0.0980 ; 0 0.4470 0.7410 ];
gscatter(features(:,1),features(:,2),classification, color)
plot(sv(:,1),sv(:,2),'ko','MarkerSize',10)
%X1 = linspace(min(features(:,1)),max(features(:,1)),100);
X1 = linspace(-1,2,100); %jank to make margins extend whole plot in next section
```

```
X2 = -(beta(1)/beta(2)*X1)-b/beta(2);
plot(X1,X2,'-')
m = 1/sqrt(beta(1)^2 + beta(2)^2);  % Margin half-width
X1margin_low = X1+beta(1)*m^2;
X2margin_low = X2+beta(2)*m^2;
X1margin_high = X1-beta(1)*m^2;
X2margin_high = X2-beta(2)*m^2;
plot(X1margin_low,X2margin_low,'b--')
plot(X1margin_high,X2margin_high,'r--')
xlabel('Eccentricity')
ylabel('MinorAxisLength')
legend('Zeros','Ones','Support Vector', ...
   'Boundary Line','Upper Margin','Lower Margin')
hold off
```

Get features function

```
function features = get_features(bw_mnist_im)
%GET_FEATURES get all regionprop features from mnist training pictures
%   input photo must be a binary image
% go through 28x28 bit samples one by one
i = 1;
total_samples = (size(bw_mnist_im,1)/28)*(size(bw_mnist_im,2)/28);
features = cell(total_samples,1);
for m = 1:28:size(bw_mnist_im,1)
   for n = 1:28:size(bw_mnist_im,2)
       sample = bw_mnist_im(m:m+27, n:n+27);
       %row = m/28 + 1;
       %col = n/28 + 1;
       %imshow(sample);
       features{i} = regionprops(sample,"all");
       %features{i} = regionprops(sample,"Area", "Circularity",
"Eccentricity","Orientation");
       i = i + 1;
   end
end
```

Make feature array function

```
function featureArray = make_feat_array(features)
%MAKE_FEAT_ARRAY take features, select certain ones and put into an array
%   Detailed explanation goes here
count = 1;
featureArray = zeros(length(features),length(fieldnames(features{1})));
for index = 1 : length(features);
```

```
   if (length(features{index}) > 0)
       featureArray(count,1) = features{index}.Area;
       featureArray(count,2) = features{index}.Circularity;
       featureArray(count,3) = features{index}.Eccentricity;
       featureArray(count,4) = features{index}.EquivDiameter;
       featureArray(count,5) = features{index}.EulerNumber;
       featureArray(count,6) = features{index}.FilledArea;
       featureArray(count,7) = features{index}.MinorAxisLength;
       count = count + 1;
   end
end
end
% EquivDiameter 0 1
% EulerNumber !!! 0 1
% FilledArea 0 1
% MinorAxisLength 0 1
```

MATLAB script implementing SVM for raw data

```
%read in image
im_train0 = imread("mnist_train0.jpg");
im_train1 = imread("mnist_train1.jpg");
im_train2 = imread("mnist_train2.jpg");
%%
%create binary image and show results
binary_imtrain0 = imbinarize(im_train0);
%%imshow(binary_im0);
binary_imtrain1 = imbinarize(im_train1);
%%imshow(binary_im1);
binary_imtrain2 = imbinarize(im_train2);
%%imshow(binary_im2);
%%
features_train0 = get_features_28(binary_imtrain0);
features_train1 = get_features_28(binary_imtrain1);
features_train2 = get_features_28(binary_imtrain2);
%%
%initialize variables for training
%Y is augmented matrix holding (+/-)1's and two values to describe digit
%in each row
class_1 = [ones(length(features_train0),1) features_train0];
class_2 = [-ones(length(features_train2),1) features_train2];
Y = [class_1; class_2];
%%
% the predictors in the matrix X and the class labels in vector Y
features = Y(:,2:end); % matrix X
classification = Y(:,1); % matrix Y
%% Train a linear kernel SVM classifier
SVMModel = fitcsvm(features,classification);
%default for two class classifier is linear kernel, for one class
%classifier it is gaussian (rbf - radial basis function) kernel
%% classifying using SVM
% get the test data set features
im_test0 = imread("mnist_test0.jpg");
im_test1 = imread("mnist_test1.jpg");
im_test2 = imread("mnist_test2.jpg");
%create binary image and show results
binary_imtest0 = imbinarize(im_test0);
%%imshow(binary_im0);
binary_imtest1 = imbinarize(im_test1);
%%imshow(binary_im1);
binary_imtest2 = imbinarize(im_test2);
%%imshow(binary_im2);
%%
features_test0 = get_features_28(binary_imtest0);
features_test1 = get_features_28(binary_imtest1);
features_test2 = get_features_28(binary_imtest2);
%%
class_1 = [ones(length(features_test0),1) features_test0];
class_2 = [-ones(length(features_test2),1) features_test2];
Y = [class_1; class_2];
```

```
% the predictors in the matrix X and the class labels in vector Y
test_features = Y(:,2:end); % matrix X
test_classification = Y(:,1); % matrix Y
%%
% classify test data
[label,score] = predict(SVMModel,test_features);
%find how many samples are misclassified
miss_samples = find(test_classification ~= label);
miss_count = length(miss_samples);
classification_error = (miss_count/length(label))*100;
```

Get feature function

```
function features = get_features(bw_mnist_im)
%GET_FEATURES get all regionprop features from mnist training pictures
%   input photo must be a binary image
% go through 28x28 bit samples one by one
i = 1;
total_samples = (size(bw_mnist_im,1)/28)*(size(bw_mnist_im,2)/28);
features = zeros(total_samples,(28*28));
for m = 1:28:size(bw_mnist_im,1)
   for n = 1:28:size(bw_mnist_im,2)
       sample = bw_mnist_im(m:m+27, n:n+27);
       %row = m/28 + 1;
       %col = n/28 + 1;
       %imshow(sample);
       features(i,:) = sample(:)';
       %features{i} = regionprops(sample,"Area", "Circularity",
"Eccentricity","Orientation");
       i = i + 1;
   end
end
```

Alternate two characteristic SVM generation and display code

```
%% Generate 0-1 SVM
SVMX = [trn0Eccen' trn0MinAxis'
       trn1Eccen' trn1MinAxis']; % C1 = eccen C2 = MinAxis
SVMY = zeros(trnY01size(2),1);
for count = 1:trnY01size(2)
   if count <=5919
       SVMY(count) = 0;
   else
       SVMY(count) = 1;
   end
end
SVMModel01 = fitcsvm(SVMX,SVMY,KernelFunction="linear");
sv01 = SVMModel01.SupportVectors; % Support vectors
beta01 = SVMModel01.Beta; % Linear predictor coefficients
b01 = SVMModel01.Bias; % Bias term
X101 = -1:0.01:2;
X201 = -(beta01(1)/beta01(2)*X101)-b01/beta01(2);
m = 1/sqrt(beta01(1)^2 + beta01(2)^2);  % Margin half-width
X1margin_low = X101+beta01(1)*m^2;
X2margin_low = X201+beta01(2)*m^2;
X1margin_high = X101-beta01(1)*m^2;
X2margin_high = X201-beta01(2)*m^2;
%% Plot 0-1 SVM training data
figure(99);
gscatter(SVMX(:,1),SVMX(:,2),SVMY)
hold on
%plot(sv(:,1),sv(:,2),'ko','MarkerSize',10)
plot(X101,X201,'-')
plot(X1margin_high,X2margin_high,'b--')
plot(X1margin_low,X2margin_low,'r--')
xlabel('Eccentricity')
ylabel('Minor Axis Length')
legend('zero','one',...'Support Vector', ...
   'Boundary Line','Upper Margin','Lower Margin')
ylim([0 30]);
xlim([-0.2 1.2]);
hold off
%% Plot 0-1 SVM test data
figure(100);
scatter(tst0Eccen,tst0MinAxis,4);
hold on
scatter(tst1Eccen,tst1MinAxis,4);
plot(X101,X201,'-')
plot(X1margin_high,X2margin_high,'b--')
plot(X1margin_low,X2margin_low,'r--')
xlabel('Eccentricity')
ylabel('Minor Axis Length')
legend('zero','one',...'Support Vector', ...
   'Boundary Line','Upper Margin','Lower Margin')
ylim([0 30]);
xlim([-0.2 1.2]);
hold off
%% 0-1 SVM Stats
a_svm01 = -[SVMModel01.Bias SVMModel01.Beta'];
svm01errs = 0;
for count = 1:tstY01size(2)
   if a_svm01*tstY01(:,count) < 0
       svm01errs = svm01errs+1;
   end
end
svm01errrate = svm01errs / tstY01size(2) * 100;
%% Generate 0-2 SVM
SVMX = [trn0Circ' trn0Perim'
```

```
       trn2Circ' trn2Perim']; % C1 = circ C2 = perim
SVMY = zeros(trnY02size(2),1);
for count = 1:trnY02size(2)
   if count <=5919
       SVMY(count) = 0;
   else
       SVMY(count) = 2;
   end
end
SVMModel02 = fitcsvm(SVMX,SVMY,KernelFunction="linear");
sv02 = SVMModel02.SupportVectors; % Support vectors
beta02 = SVMModel02.Beta; % Linear predictor coefficients
b02 = SVMModel02.Bias; % Bias term
X102 = -1:0.01:2;
X202 = -(beta02(1)/beta02(2)*X102)-b02/beta02(2);
m = 1/sqrt(beta02(1)^2 + beta02(2)^2);  % Margin half-width
X1margin_low = X102+beta02(1)*m^2;
X2margin_low = X202+beta02(2)*m^2;
X1margin_high = X102-beta02(1)*m^2;
X2margin_high = X202-beta02(2)*m^2;
%% Plot 0-2 SVM training data
figure(101);
gscatter(SVMX(:,1),SVMX(:,2),SVMY)
hold on
%plot(sv(:,1),sv(:,2),'ko','MarkerSize',10)
plot(X102,X202,'-')
plot(X1margin_high,X2margin_high,'b--')
plot(X1margin_low,X2margin_low,'r--')
xlabel('Circularity')
ylabel('Perimeter')
legend('zero','two',...'Support Vector', ...
   'Boundary Line','Upper Margin','Lower Margin')
ylim([0 140]);
xlim([-0.2 1.2]);
hold off
%% Plot 0-2 SVM test data
figure(102);
scatter(tst0Circ,tst0Perim,4);
hold on
scatter(tst2Circ,tst2Perim,4);
plot(X102,X202,'-')
plot(X1margin_high,X2margin_high,'b--')
plot(X1margin_low,X2margin_low,'r--')
xlabel('Circularity')
ylabel('Perimeter')
legend('zero','two',...'Support Vector', ...
   'Boundary Line','Upper Margin','Lower Margin')
ylim([0 140]);
xlim([-0.2 1.2]);
hold off
%% 0-2 SVM Stats
a_svm02 = -[SVMModel02.Bias SVMModel02.Beta'];
svm02errs = 0;
for count = 1:tstY02size(2)
```

```
   if a_svm02*tstY02(:,count) < 0
       svm02errs = svm02errs+1;
   end
end
svm02errrate = svm02errs / tstY02size(2) * 100;
```