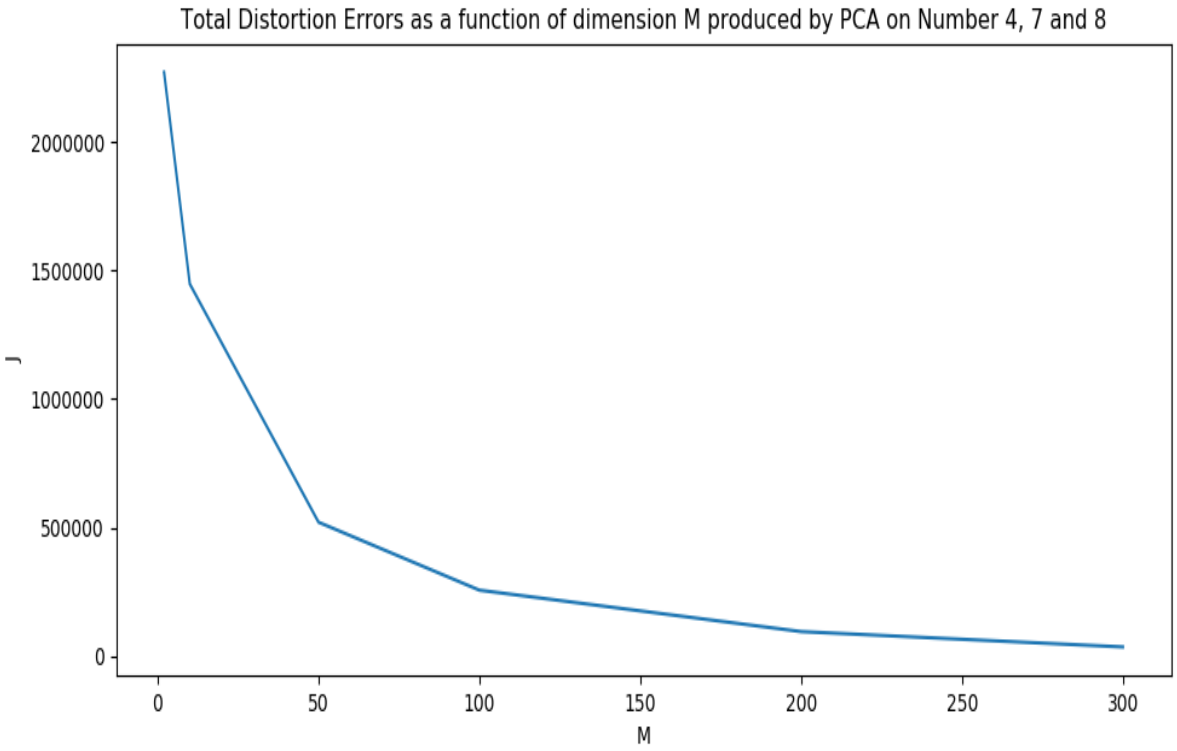
GS-EECS6327A Project 1 Report

Yunge Hao

214361760

1.

a.

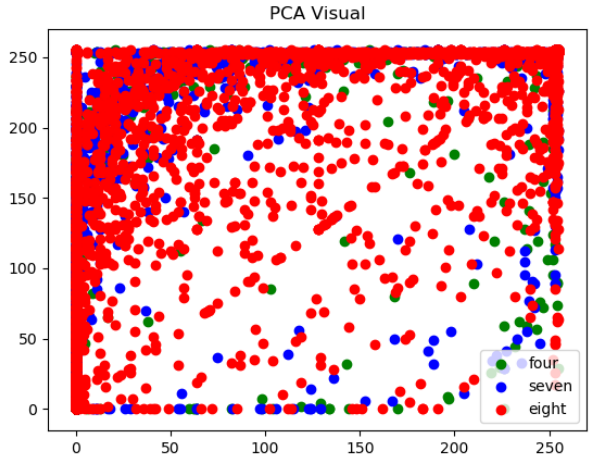


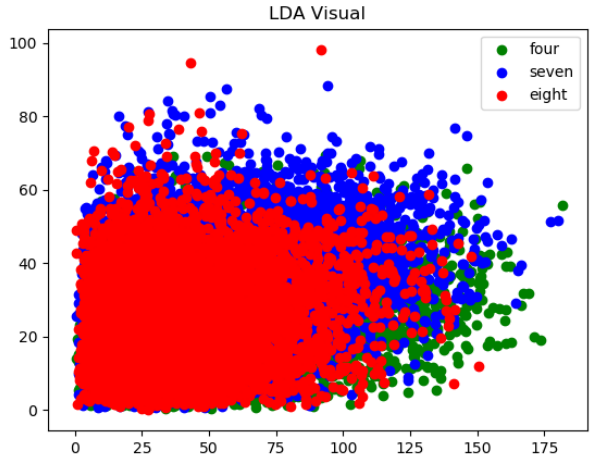
When M is larger than 50 the dropping slows. At M = 300 it is nearly 0 and no more significant change.

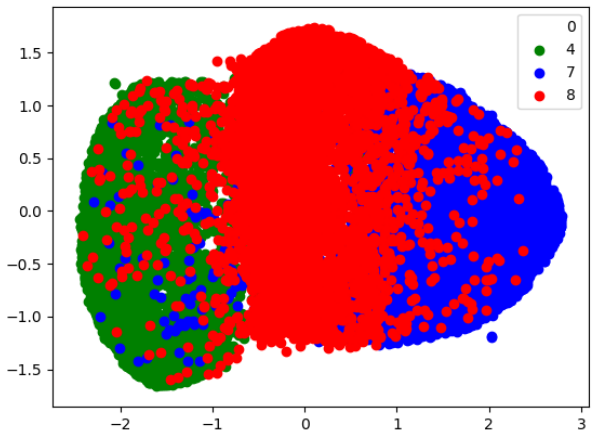
b.

The maximum dimension that LDA can achieve of classification is 2, since this is the number of classes in total subtracted by 1. This is the rank of the projection matrix of LDA. (See it in the code and the output when running it.)

c.


PCA Visual
four
seven
eight


LDA Visual
four
seven
eight


0
4
7
8

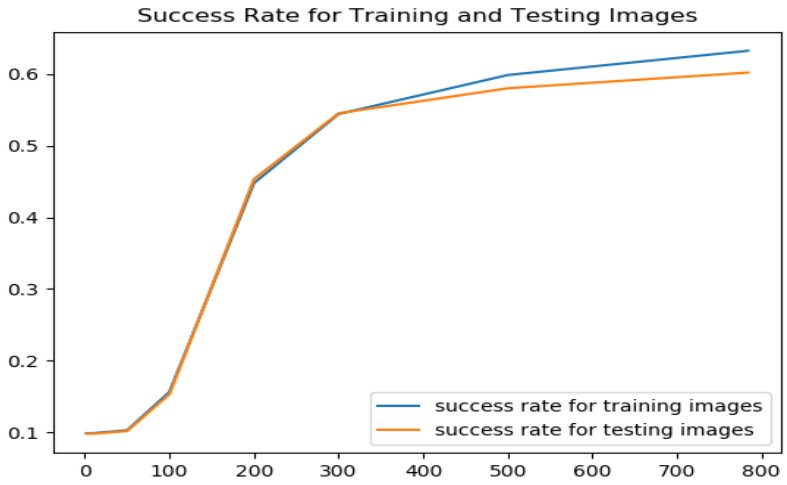
t-SNE visual

The plots are along the two dimensions of the projected images, it looks that LDA performs better in separating three classes than PCA in a low dimension. While PCA aims at maximizing the variance of the projected data, it scatters more widely. But t-SNE is way better than LDA, it uses student-t distribution which is very flat so it covers more data within one class. All three plots show overlaps in 2D of the images.

2.

Linear Regression

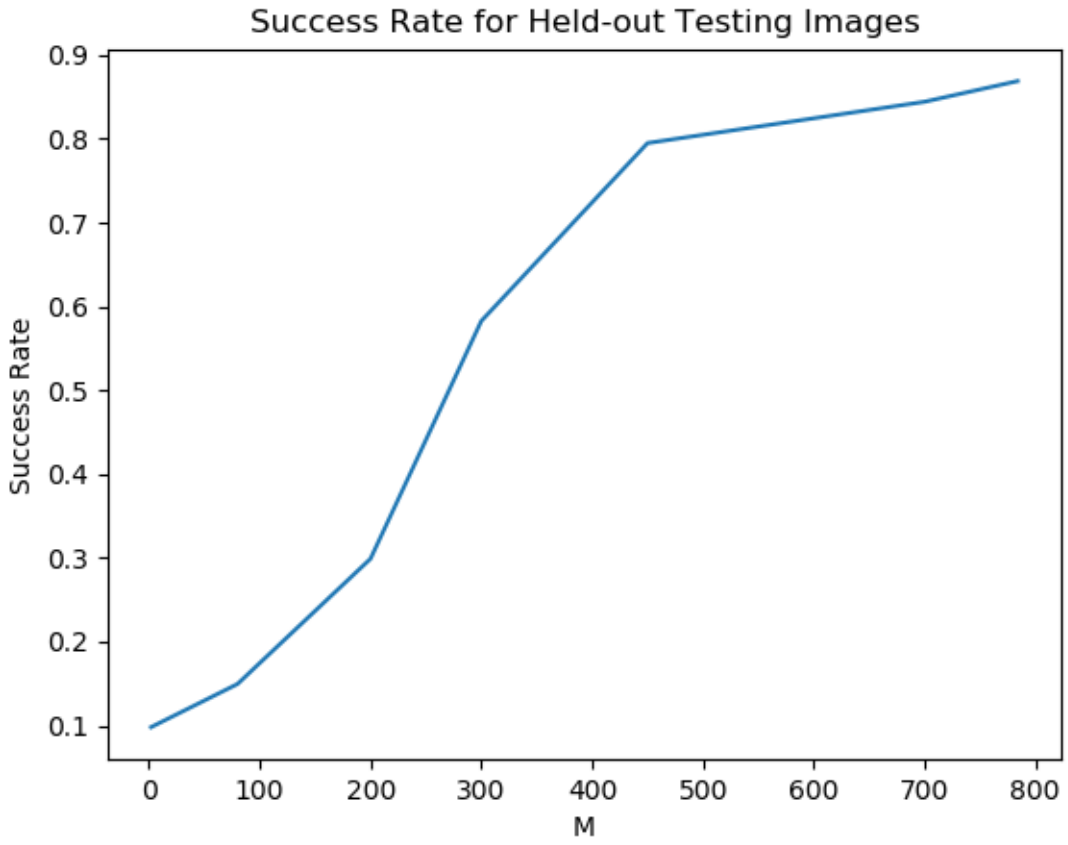
I used least square linear regression. I plotted the success rate as an average of success rates of all 10 digits.



The dimensions in the linear model are 2, 10, 50, 100, 200, 300, 500 and 784. The way I did it is training ten classifiers for ten digits. When testing, all ten classifiers participates, but only the one with the minimum error is chosen as the classification with this testing image. It is shown in the figure that both of the success rate increases as model size increases, until around 300, where the overfitting overcomes its success on training set. Both the success rates keep slowly increasing and then become stable, but it is not good, only 60%. I also tried it differently, to calculate the closed form solution for all of the images at once, instead of calculating the closed form one by one like I did here. But the result is worse, a closed form solution for all digits only has 30% success rate, even though a closed form solution for each pair gives 80 to 90% success rate.

Logistic Regression

Using softmax as the activation function is not a good choice for Python since the exponential part always causes an overflow. I switched to sigmoid function instead, then overflow can be avoided by adjusting the sigma value σ. Its absolute value has to be small enough to make the exponent part not too large. I found out that $\sigma = -0.00001$ is the largest magnitude factoring of 10 to avoid a RuntimeOverflow warning in a most general case. The optimization is done by stochastic gradient descent. (Gradient descent takes too long.) I tested it for various dimensions. A learning rate of 0.5 and number of iterations of 100, 000 suffices to produce a fairly good prediction. The plot for success rate is still based on discrete model sizes as in linear regression.



Logistic regression can do better than 85% success rate.

I used one-vs.-the-rest strategy for linear regression and logistic regression.

3.

Linear Soft-SVM

SGD is still used here. Loss is calculated by Hinge Loss function, and gradient is by sub-gradient. The parameter set is {C, K, T, L, M} where C stands for the penalty for misclassification, K stands for total number of classes involved in the training and testing, T stands for number of iterations, L stands for learning rate, M stands for the number of dimensions of features.

One-vs.-one is suggested, so I trained $K(K-1)/2 = 45$ one-vs.-one pairs of classes. The predicting schema is "majority votes" among the 10 classes for each testing iteration. This approach has a problem with ambiguous prediction, in my implementation you can see that I classify the test results into 3 classes: success, failure and ambiguity, each represent by their raw counts. I found out that for small value of K, it is very easy to get 90% accuracy prediction, but as the value of K increases, failure and ambiguity increase as well. But ambiguity is also a form of misclassification. I have to dramatically adjust for C, the penalty, and L, learning rate. Eventually, a C of exp(700) and L of 0.00006, T of 100, 000. For large K, I want it to punish wrong classification really hard, and step very slow, provided a large enough number of iterations. The most crucial parameters are the learning rate and the penalty, combining them together would tremendously reduce failure and

ambiguity, and give a success rate stabled at almost 90%, that is the percentage of success out of the sum of success, failure and ambiguity.

Each iteration of testing, a Votes vector is printed, showing each class's votes.

Non-Linear RBF Kernel SVM

My laptop could not allocate a 60, 000 by 60, 000 matrix nor one half of it so I reduced the size by one-third. But I could not construct the kernel, it takes too much time. I was unable to finish this one. This part of code is commented out.

4.

Neural Networks

I began with one hidden layer of 15 nodes. The input layer has 784 nodes. The output layer has ten nodes, each represents a number from 0 to 9. The prediction is based on the index of the maximum of the outputs. All activation functions are sigmoid function, including the output activations.

The error of each output node is its output, a value between 0 and 1, subtracted by 1, if the data being trained is at the corresponding index, otherwise the error of an output node is just its output. So eventually the all the error terms will convert to 0. I did not calculate squared error since the mere subtraction is already the gradient for labels in terms of outputs. Sigmoid function has simple gradient as $s(1 - s)$. Gradients for weights between hidden layer and output layer are calculated first, then use these to calculate gradients for weights between input layer and hidden layer, by chain rule. It is important to make use of dynamic programming technique, storing partial gradients after calculating weights between hidden layer and output layer in an array which will be looked up later for weights between input layer and hidden layer. Avoiding nested loop, the runtime is much less.

In order to have error converged, data needs to be normalized. Dividing by 2550 suffices.

Gradient descent is applied. For each epoch, success rate is printed. I have the two learning rates, one for the first weights, one for the second weight, initialized to be 3.5 and 2.5, and incremented after each EPOCH. But the result is not so good, the highest success rate I ever got is 92.6%. I tried increasing number of nodes in the hidden layer. Eventually at 86 nodes of hidden layer the success rate is guaranteed at 98%.