STA 135: Final Project

Haomiao Meng
Jack Norman
Won Lee

March 21, 2014

We certify that we have acknowledged the source of all work in this assignment that is not our own original work, and that we have not copied or modified the work of others without attribution.

Introduction

The handwriting recognition is used in many fields because of its great applicability and convenience in data organization and processing. It reads handwritten input and interprets what that input says automatically. With this machine learning techniques, we can get a lot of works done much more efficiently since computer machines can recognize and report outputs much faster than human once the program is designed in machines. This technique is used in fields such as bank check processing, postal-address interpretation, signature verification, etc. However, the major challenge in the handwriting recognition is that people do not have the same handwriting. Hence, people are required to construct the best machine learning algorithm to minimize making errors. In this experiment, we are only going to focus on recognizing single digits from 0 to 9, and try to produce an algorithm to maximize the accuracy of the successful digit recognition for different kinds of machine learning techniques, such as K-th nearest neighbor, support vector machine, random forest, neutral network, etc. Then, we will compare them, and determine which one is the best digit recognizer in overall and also for each digit.

Data

Our data comes from the famous MNIST (Modified National Institute of Standards and Technology) dataset, which is a large database of handwritten digits. Each handwriting is actually a picture fitted into a 28×28 pixel bounding box. So for each observation, we have 784 variables to describe this picture.
The original dataset contains 60000 training images and 10000 testing images. Since we acquired the data from Kaggle competition, we only have 42000 training data with labels. So we will treat the 42000 data as our whole dataset and sample 28000 of them as our training data, the rest of which are testing data.
In our 28000 training data, each digit distributed as in table 1:

Table 1: Number of 10 digits in training data

0	1	2	3	4	5	6	7	8	9
2759	3096	2792	2902	2724	2561	2762	2898	2705	2801

In our 14000 testing data, each distributed as following:

Table 2: Number of 10 digits in testing data

0	1	2	3	4	5	6	7	8	9
1373	1588	1385	1449	1348	1234	1375	1503	1358	1387

From table 1 and table 2, we can see that these digits are well balanced. So we can say that our sample is good representative of original data. But in the following section, we only focus on training data.

Figure 1 demonstrate the first 100 images of the digits in training data.

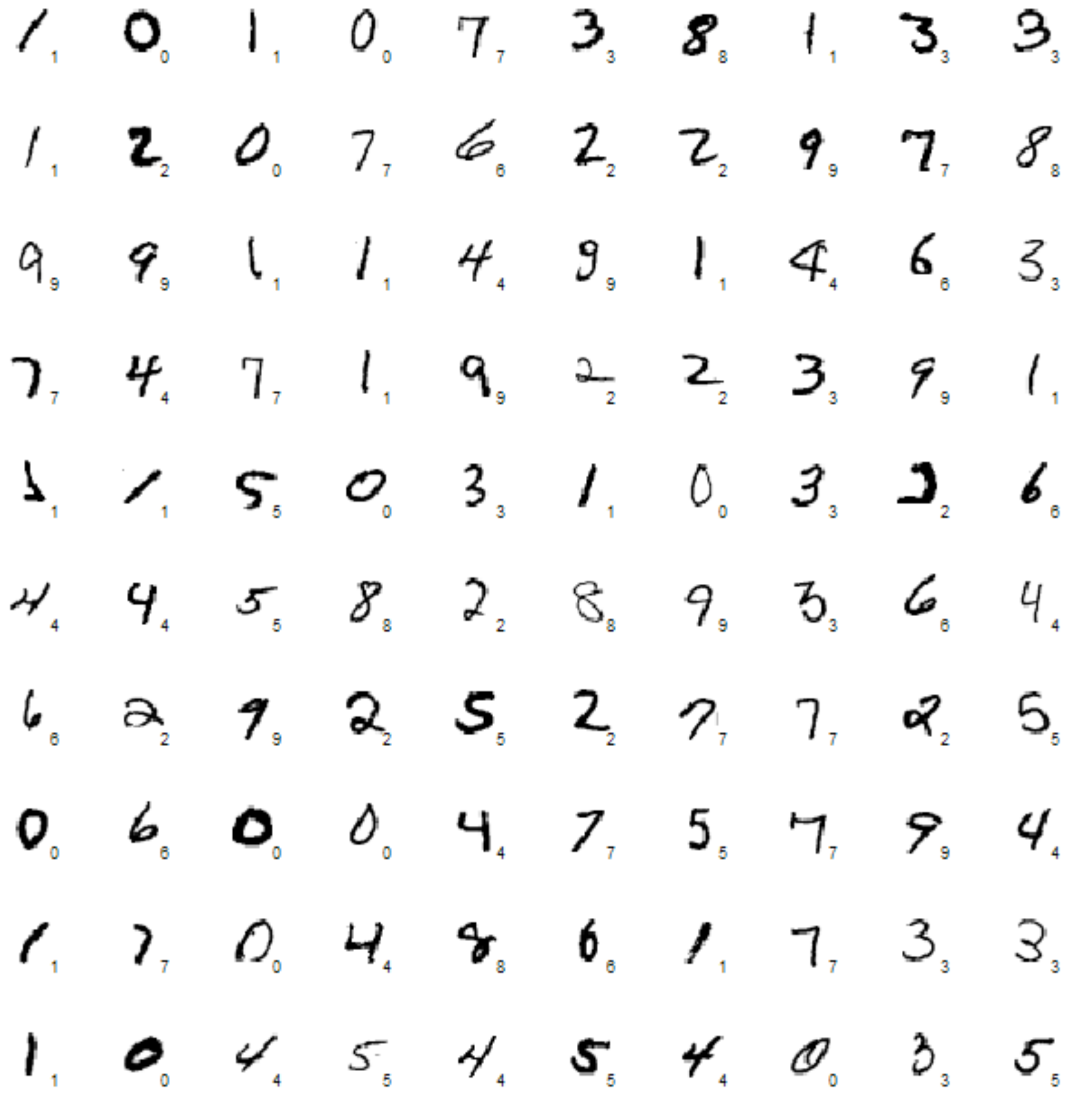


Figure 1: First 100 digits in training data

Preprocessing the Data

Of the 784 pixels, many have little to no variance. Furthermore, there are some pixels that rarely, if ever, have ink. Because of these situations where we have many variables with little information, we realized reducing the dimensions would be an appropriate place to start. Throughout our analysis, we try two dimension reduction techniques: First, we try the well-known principal component analysis. Second, we come up with our own method, which we will refer to as "blocks" or "blocking". The blocking algorithm is described later.

Principal Component Analysis

The idea of principal components is to reduce the number of dimensions while retaining most of the information stored that's in the raw data. To perform this reduction, our goal is to take a linear combination of the original 784 variables,

$$Y = \alpha_1 X_1 + \alpha_2 X_2 + \ldots + \alpha_{784} X_{784}, \; s.t. \sum_{p=1}^{784} \alpha_p = 1$$

such that the variance of *Y* is maximized. To calculate the *nth* principal component, we optimize *Y* like before with an additional constraint: the vector of alphas is orthogonal to all previous vectors of previous components. It turns out that the *nth* principal component is the *Y* that has its vector of alphas being the *nth* eigenvectors of *X*.

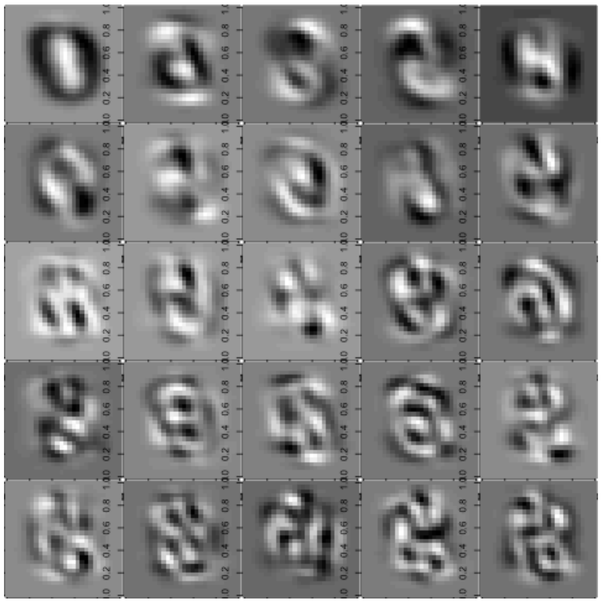


Figure 2: First 25 principal components

In our handwriting data, we found all 784 principal components of *X*. To get an idea of the principal components, below are images of the first 25 principal components. Each principal component image gives a visualization of the loadings (the values of a1, a2, …, a784) where black indicates positive loadings, white indicates negative loadings, and grey indicates near-zero loadings.

As one can see from these images, the positive and negative emphases of the loadings on particular pixels can be seen in patterns. In the first principal component, the blackest area resembles the shape of the number zero (and almost resembles an eight as well because of some pinching in the middle), while the whitest area resembles the inside of the number zero. Alternatively, in the third principal component, the blackest area resembles the inside of the number eight while the lighter pixels resemble the outline of the number eight. It is interesting to see how each principal component has such a unique distribution of positive and negative loadings accounting for different recognizable patterns of numbers. Also, notice how the patterns of the loadings become more complex for the later principal components.

To determine how many principal components we wanted to use, we created a scree plot. The scree plot, which shows how much variance is accounted for by each component, cumulatively, can be seen below.

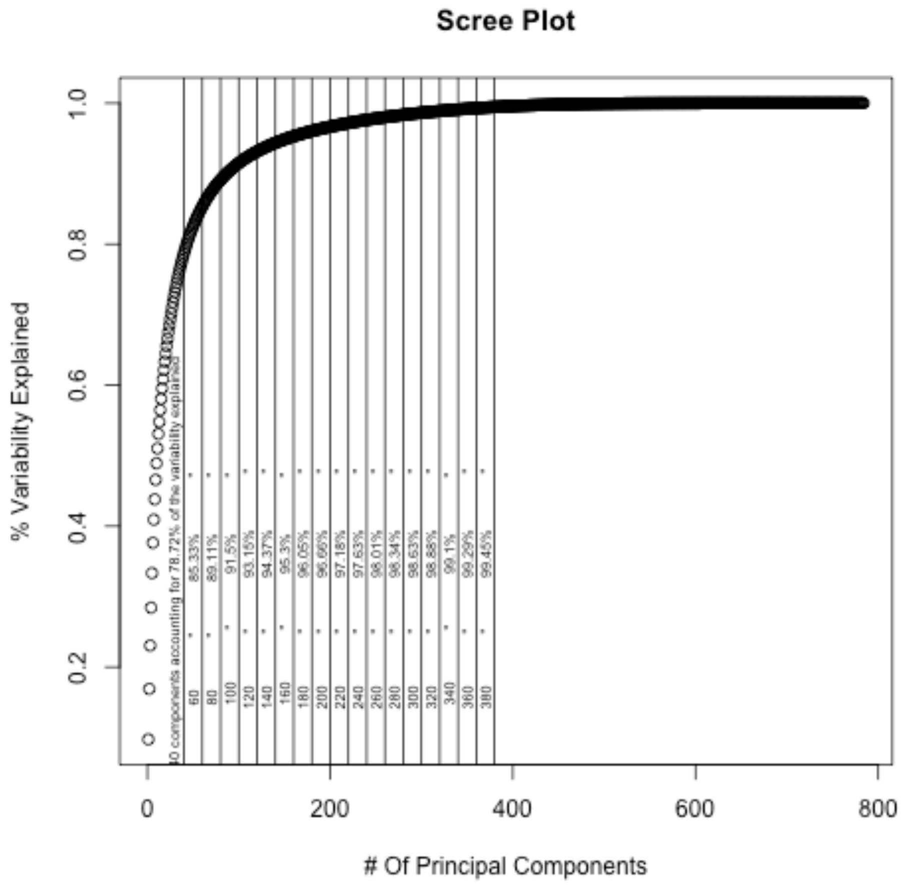


Figure 3: Scree plot for selecting principal components

From the scree plot, the curve of the points is fairly gradual so there isn't a trivial cutoff point.

When the curve almost completely flattens out, there are 380 components accounting for over 99% of the variability. We believe that using less than 380 components is appropriate since the amount of additional variability we can explain by adding the 380th component (or the 300th, 250th, 200th, and so on) is so little and that this miniscule improvement of explanation of variability is not worth the noise that we will be adding by having many more components. In our analyses to follow, we typically will be using either the first 59 principal components (85% of the variability), the first 87 principal components (90% of the variability), or the first 154 principal components (95% of the variability).

Blocking (Shrinking or Chunking)

The second dimension reducing technique we use is blocking. The idea of blocking is to group the pixels, by summing them into larger pixels. For instance, we begin with 784 variables, which correspond to a 28 by 28 matrix of pixels. We can group these pixels so that the matrix is now 14 by 14 by summing the 1st, 2nd, 29th, and 30th pixels in to one. Now, in the 14 by 14 matrix, the first pixel contains information about the 1st, 2nd, 29th, and 30th pixels. We can apply more blocking so that we have a 4 by 4 matrix. A visualization of the pixels can be seen below.

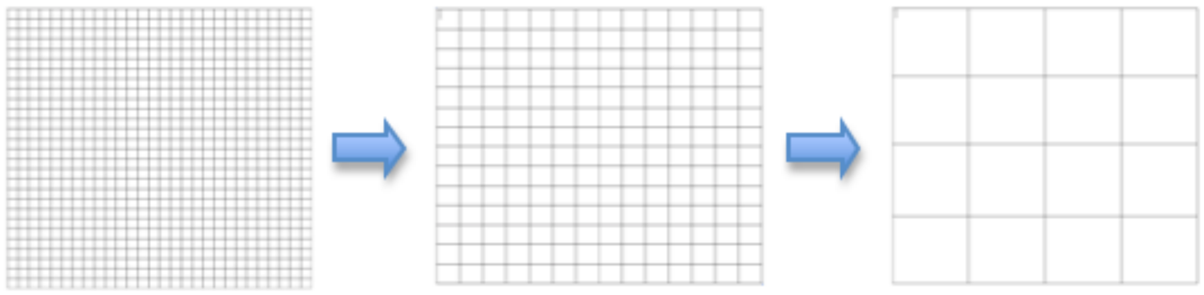


Figure 4: Chunking process

Not only are we reducing the dimension with this technique, but also the data becomes more normal due to the central limit theorem since we are adding many random variables together. For example, in the 4 by 4 grid, each pixel is the sum of 49 random variables. The goal of blocking is to reduce the dimension to make computation simpler and reduce noise, to retain as much information as possible, and to better normalize the data.

Methods and Results

LDA and QDA

Here, we are using linear discriminant analysis and quadratic discriminant analysis to build up two classifiers and determine the misclassification rates using both methods individually. For LDA, we assume that the data has multivariate normal distribution, and each class has the same covariance matrix. On the other hand, we do assume that the data has multivariate normal distribution, but we do not assume each class has the same covariance matrix. Note that we are using 59 principal components here because they preserve 85% variability, and we do not want to use more principal components because the more principal components we use, the more variability and noise we get.

The Linear Discriminant Function:

$$\delta_k(x) = x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \log(\pi_k)$$

The Quadratic Discriminant Function:

$$\delta_k(x) = -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) + \log\pi_k$$

Table 1: Error Rate of LDA

0	1	2	3	4	5	6	7	8	9	Average
6.92%	3.97%	19.71%	16.36%	12.39%	17.83%	8.00%	14.90%	19.07%	13.53%	13.11%

From Table 1, we see that the overall misclassification rate is 13.11%; hence, LDA is not a good classification model. We see that the misclassification rate for digit 1 is the lowest with 3.97% and the misclassification rate for digit 3 is the highest with 19.07%.

Table 2: Error Rate of QDA

0	1	2	3	4	5	6	7	8	9	Average
1.38%	4.47%	3.54%	5.45%	2.23%	4.30%	3.64%	6.39%	3.83%	6.13%	4.17%

From Table 2, we see that the overall misclassification rate is 4.17%, so QDA is a good classification model. We see that the misclassification rate for digit 0 is the lowest with 1.38%, and the misclassification rate for digit 6 is the highest with 6.39%.

LDA model does not have good performance because our assumption of multivariate normality has been violated, and the variances of different digits are different.

K-nearest neighbor

In this part, we are using KNN to build up a classifier and determine the misclassification rate. KNN is a nonparametric way to classify observations in the data set. From the training set, the KNN will learn where each observations locate. Then, based on the information that is obtained from the training set, it will classify the class of observations from test/validation set by observing k-closest observations from the training set. It will take K-closest observations from the training data to the observations from the test data, and the most frequent class from the K observations from the training set becomes the class of the corresponding observation from the test data. Note that we only tried Euclidean distance metric because running KNN with

Manhattan distance metric takes very long time where we cannot even get the cross validation done for more than 2 hours.

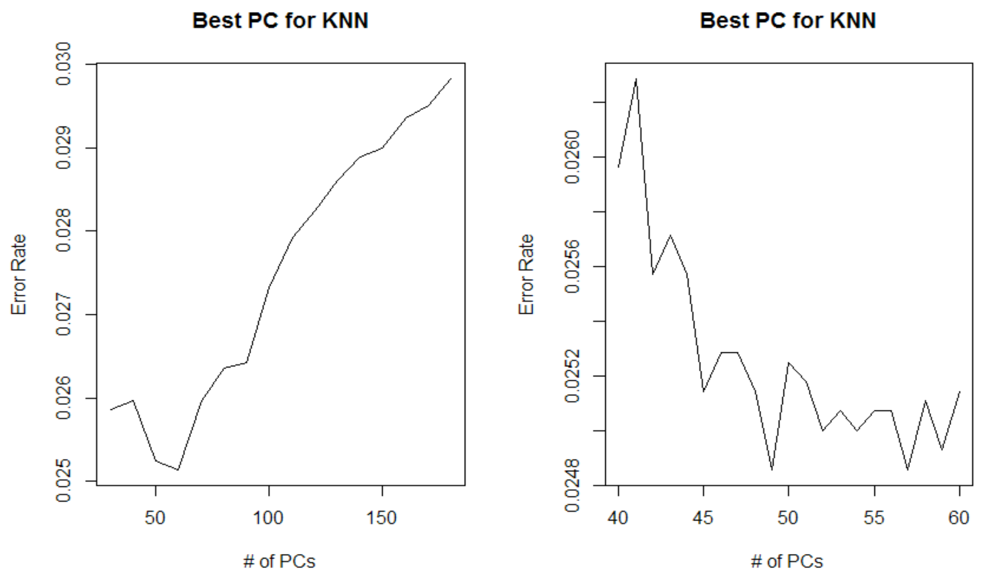


Figure 5: Best principal components for KNN

From Figure 5, we see that having 40~60 principal components give us the lowest misclassification rate when we cross validate the KNN. Taking a deep look into it, we see that having 57 principal component is the optimal option. We also have tried the Chunk method to reduce the dimension of the data, but the result was worse than the PCA. Using PCA has approximately 0.1% lower misclassification rate.

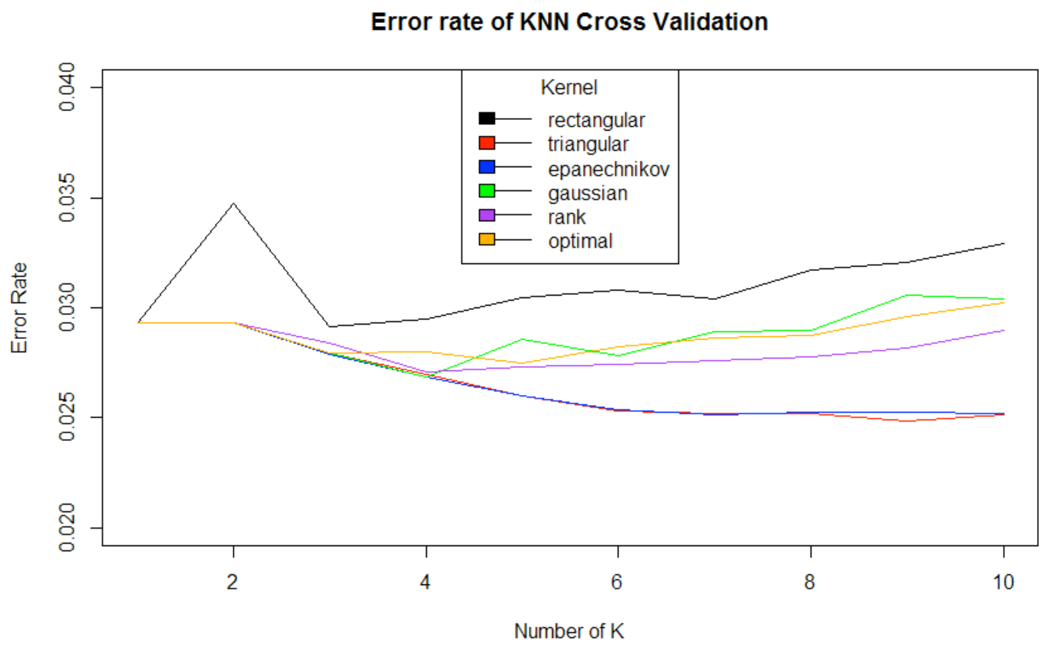


Figure 6: Error rate of KNN cross validation

From Figure 6, we see that when we conduct leave-one-out cross validation with 57 principal components, the best KNN model is when K=9 with triangular kernel with the misclassification rate = 2.485714%. Hence, we will see how well this model classifies the test data set now.

Table 3: Error Rate of KNN

0	1	2	3	4	5	6	7	8	9	Average
1.02%	0.63%	7.37%	4.90%	8.53%	10.37%	1.96%	6.59%	6.48%	2.88%	5.96%

From Table 3, we observe that the overall misclassification rate of KNN is 5.96%. The misclassification rate for digit 1 is the lowest with 0.63%, and the misclassification rate for digit 5 is the highest with 10.37%.

Random Forest

Random forest is an ensemble algorithm, which is built on classification tree. The basic idea behind this algorithm is to generate many different classification trees and let them vote. However, unlike classification tree, each tree in random forest is different. The “randomness” comes from two ways: 1. each tree is based on a dataset bootstrapped from the original dataset; 2. when searching the best cut on each node, the algorithm randomly select m variables and choose the best cut among those m variables. So for every single tree, the prediction may not be as good as a classification tree, but when they form a forest and vote, the prediction might enhance.

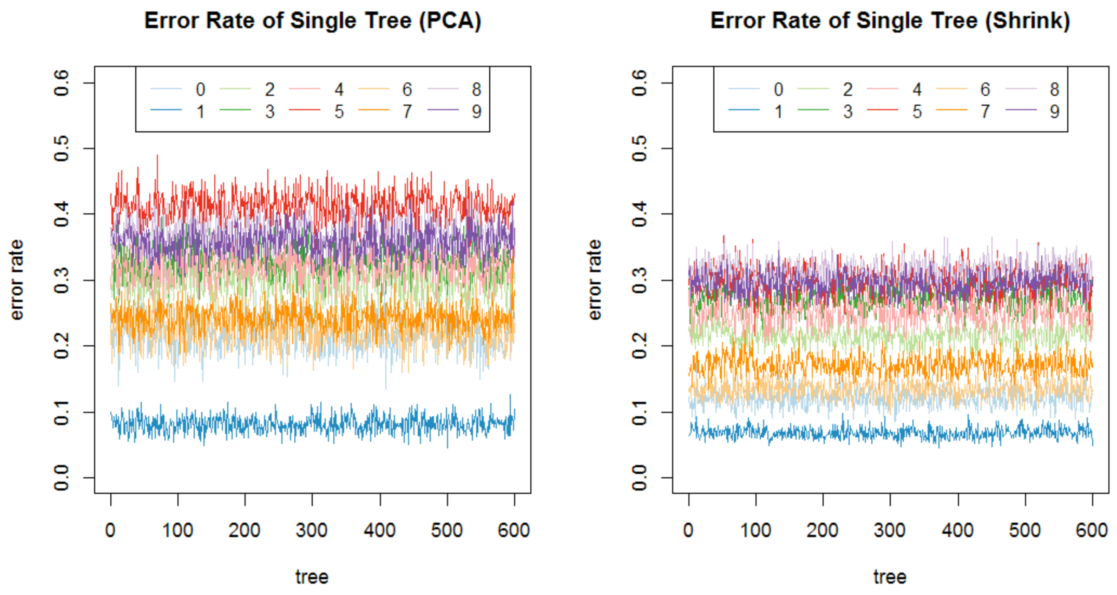


Figure 7: Error rate of PCA method against chunking method

As other methods, we also use PCA and shrinking image to preprocess the data. In this case, we fit 600 trees and choose m=8 as the number of variables randomly sampled as candidates for each split. Figure 7 shows the difference between the error rates by using two different preprocessing techniques.

From figure 7, the shrink method is better than PCA on every aspect. First, each digit’s error rate of shrink method is lower than PCA. Besides, shrink method also has lower variance of error rate, which means its result is more stable. Actually the overall error rate of PCA is 5.21%, while the shrink method is 4.96%.

Next we explore the relationship between total error rate and chunks. We sample 80% training data and test the rest of 20% data. We do two times and average them.

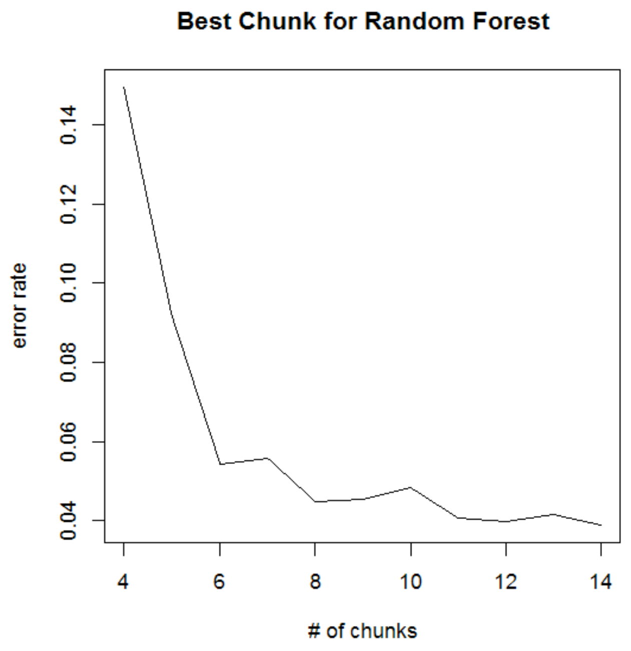


Figure 8: Best chunks for random forest

Figure 8 shows us the best chunk is 14, not 7. However, when chunks larger than 11, the error rate is about the same. We also try different PCs to see if there is a best k to select PCs.

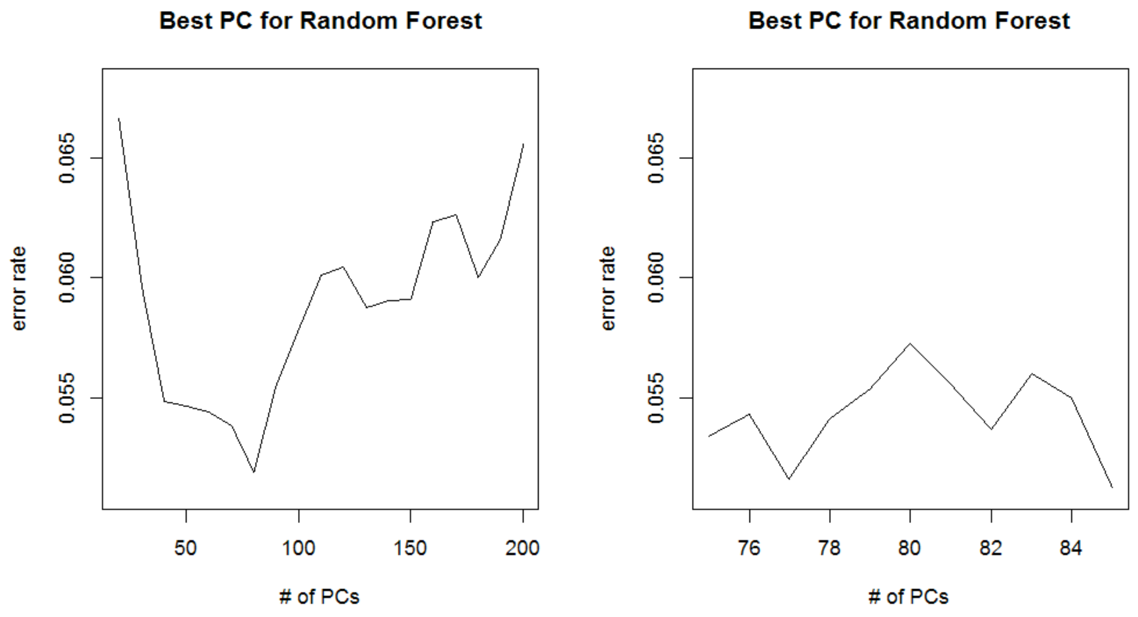


Figure 9: Best PCs for random forest

From figure 9, it seems that when k is around 70 to 80, the average error rate achieves its minimum. However, when we try k from 75 to 85, it seems that the error rate is not stable, though smaller than when k is below 40 or larger than 100. But even k is in this range, its error rate is still larger than shrink method with chunk larger than 8. So for random forest method, we use shrink method to predict test data. Table 4 is our final random forest model by fitting 900 trees.

Table 4: Error Rate of Random Forest

0	1	2	3	4	5	6	7	8	9	Average
1.95%	1.39%	3.90%	4.84%	3.39%	4.53%	2.39%	3.23%	5.18%	6.43%	3.69%

Support Vector Machines

Another classification technique that we used was support vector machines (SVM). The idea of SVM is to fit a hyperplane between two groups of points such that the margin between the groups' closest points is maximized. The logic behind this is that if the margin is maximized, then there will be a good chance (we hope) that the points on one side of the hyperplane belong to one group while the points on the other side of the hyperplane will belong to the other group. In our handwriting data, our response variable that we are trying to classify can take on ten different values. For SVM to work with ten groups, it finds a hyperplane that separates the 0's and 1's, 0's and 2's, …, 0's and 9's, …, 8's and 9's. The observation will then get voted on by these hyperplanes and which ever digit gets the most votes is the one in which we predict our observation as.

There are many parameters that one can change when computing the hyperplanes of SVM. For example, there are different types of kernels we can use such as a radial kernel, linear kernel, polynomial kernel, or sigmoid kernel. These kernels determine the shapes and behavior of the hyperplanes that separate our points. Of course, each kernel will excel in particular situations depending on the characteristics of the data. To get a sense of which kernel works best with out handwriting data, we used cross validation on the training data and computed the average success rate of each kernel (while keeping all other parameters as default).

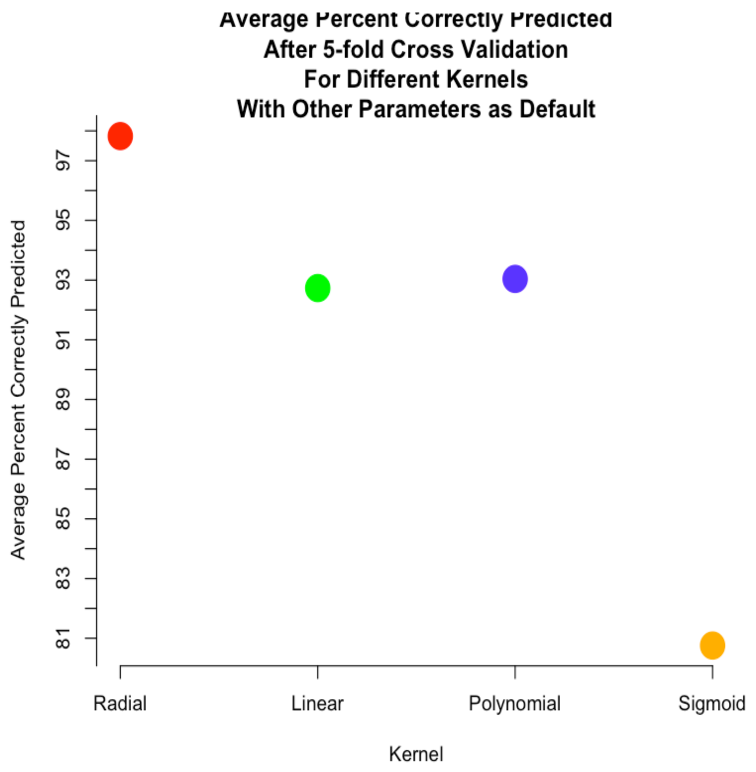


Figure 10: Correct rate for different kernels

As one can see from this kernel plot, it appears that the radial kernel will give us our best chance at capturing the most digits accurately. To be sure, we also computed the percent correctly predicted for linear, polynomial, and sigmoid while changing other parameters in order to help better the percentage. While we were able to improve the percentages of these kernels, they still didn't come close to the capability of the radial kernel.

Proceeding using the radial kernel, we then altered the two parameters of the radial kernel: 'C' and '□'. The gamma parameter affects the way the kernel is computed in the following equation:
$e^{-\gamma|u-v|^2}$. The 'C' is cost of constraints violation. According to Joachims of "Learning to Classify Text Using Support Vector Machines: Methods, Theory and Algorithms", the 'C' parameter "allows one to trade of training error vs. model complexity. A small value for C will increase the number of training errors, while a large C will lead to a behavior similar to that of a hard-margin SVM." In addition, a very large C will tend to overfit the training data, which we would like to avoid. The authors of the 'e1071' package in R who wrote the svm() function recommend to try a large range of values of C and to then adjust gamma accordingly. After trying different values of C, the best value of C was 100 with an average error rate of 1.9%. Costs of larger value produced the same error rate, but for the sake of parsimony and risk of overfitting, we choose C to be 100. With C to be 100, we alter gamma seen in the plot below.

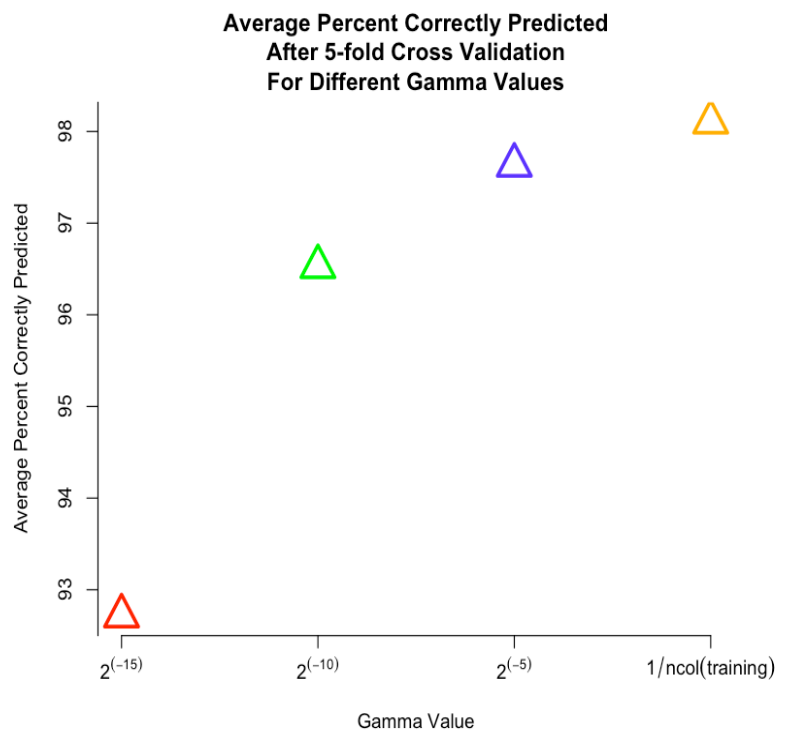


Figure 11: Correct rate for different gamma

As one can see from the plot above, it appears that using 1 divided by the number of columns of the training dataset, we yield the best prediction rate in a 5-fold cross validation. Because of the above plots, the parameters we decide to use on the testing dataset are 1) the radial kernel with 2)$C = 100$ and 3) $\gamma = \frac{1}{\text{ncol(training)}} = \frac{1}{59} = .01694915$.

Using this model on the 10,000 testing data points, we get the following table:

Table 5: Error Rate of SVM

0	1	2	3	4	5	6	7	8	9	Average
0.40%	0.76%	2.35%	2.82%	2.30%	2.04%	1.13%	1.58%	4.21%	1.74%	1.91%

As one can see, SVM performs very well especially when the handwritten digit is a 0 or 1. The only digit that SVM "struggles" with is when the digit is an 8 and the error rate is 4.21%.

Naive Bayes

Naive Bayes is another classification technique we used in our quest to classify the handwritten digits. The idea of Naive Bayes is to apply Bayes rule. Recall that Bayes Rule is:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

To put this rule in the perspective of the handwriting digits and to give it intuition, it is best to think of Bayes Rule as in the following equation:

$$P(digit = 1 \mid values\ of\ pixels) = \frac{P(values\ of\ pixels \mid digit = 1)\ P(digit = 1)}{P(values\ of\ pixels)}$$

We can then calculate the probability of the digit being a 1, 2, …, 10, and the digit with the highest probability given the values of the pixels we observed will be the digit we predict. The reason this method is called naive is because it assumes that each variable (pixel) is independent of all other variables. In the digits data, the pixels are clearly not independent since the value of one pixel can give you an idea about the values of other pixels (particularly the pixels near by). This assumption that we would be violating would hurt our successful prediction rate.

In an attempt to ameliorate this problem, we first used principal component analysis to reduce the dimension and also to make each variable orthogonal to all other variables. In doing so, we become "less naive" in a sense that our assumption of independence between variables is satisfied.

After using cross validation to find the optimal number of principal components to use, we found that 59 components (accounting for 85% of the variability) yielded the best (lowest) misclassification rate. Applying Naive Bayes (using 59 principal components) on the 10,000 testing observations, we get the following table.

Table 6: Error Rate of Naive Bayes

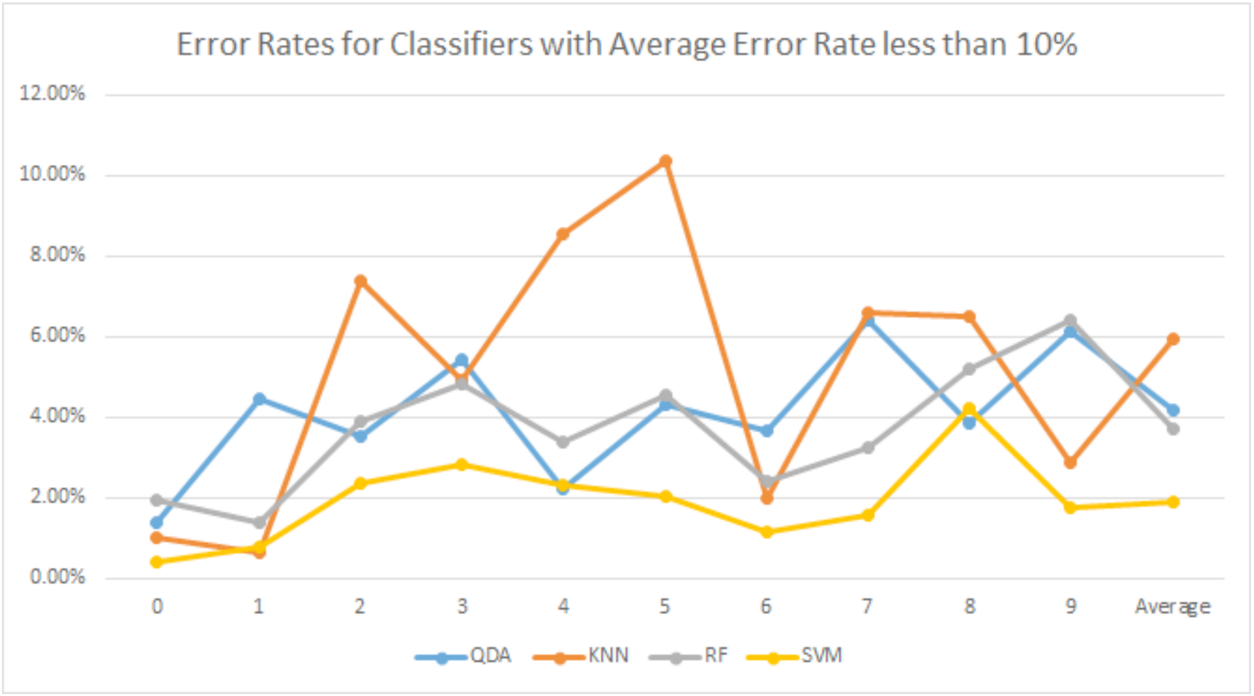
0	1	2	3	4	5	6	7	8	9	Average
3.47%	75.17%	17.5%	18.93%	47.31%	65.15%	18.54%	33.04%	9.53%	33.00%	32.61%

It is evident from the confusion matrix of the Naive Bayes classifier that this is a very poor classification technique for our handwriting digits data. The error rate for when the digit is a 1 is a whopping 75%! One can see from the confusion matrix that this classifier overpredicts the digit '8' by a lot. This seems to make sense because the 8 typically covers the most pixels. Therefore, it will usually have more pixels that have nonzero probability when computing the Bayes formula. This is consistent with the fact that 3's, 2's, and 0's cover a lot of pixels where 1's cover the least. Our Naive Bayes model predicts 3's, 2's, and 0's a lot, but the 1's are rarely ever predicted. From this finding, it appears that Naive Bayes doesn't do well when the variances between groups are different.

Conclusion

From what we have discussed, we choose four methods to classify digits. Table 7 shows the average error rate of these methods.

Table 12: Error rates for classifiers that have error rate less than 10%



1. SVM has the best performance compare to other classifiers. However, for each digit, SVM is not necessarily always better than other classifiers for predicting every digit. Some of methods, like QDA and KNN, have the two highest error rates in the table, but QDA appears to be the best model to predict 4 and 8, and KNN appears to be the best model to predict 1.

2. Different preprocessing techniques of dimension reduction should be applied to different methods. For example, PCA works better for QDA and KNN, but the chunk method works better for random forest.

Appendix

A1. LDA and QDA R code

```
library(MASS)
library(car)
library(caret)
library(class)
library(kknn)
library(parallel)
load("digitsTrain.rda")
load("digitsTest.rda")
train=sampleTrain[,-1]
test=sampleTest[,-1]
PC=princomp(train)
B=(PC$sdev)^2
prop=cumsum(B)/sum(B)
k=which(prop >= 0.85)[1]
sum(B[1:k])/sum(B) #k=59 because it gives 0.85
P=PC$loadings[,1:k]
s.dat=PC$scores[,1:k]
t.dat=as.matrix(test)%*%P
PLDA=lda(s.dat,as.factor(sampleTrain$label))
PQDA=qda(s.dat,as.factor(sampleTrain$label))
PLR=predict(PLDA,t.dat)
QLR=predict(PQDA,t.dat)
ConfMat1=confusionMatrix(PLR$class,as.factor(sampleTest$label))
ConfMat2=confusionMatrix(QLR$class,as.factor(sampleTest$label))
#End of LDA and QDA R Code
```

A2. KNN R code

```
#Chunk Method
V=NULL
for (x in 2:14) {
  sample=chunk(data=sampleTrain,quadrants=x)
KKNN=train.kknn(as.factor(sampleTrain$label)~as.matrix(sample),data=sample,km
ax=10,distance=2,
c("rectangular","triangular","epanechnikov","gaussian","rank","optimal"))
  V1=min(KKNN$MISCLASS)
  V=c(V,V1)
}
min(V) #0.02596429
#Finding the optimal number of PC
a=lapply(seq(30,180,10),function (x) {
sample.dat=PC$scores[,1:x]
result=train.kknn(as.factor(sampleTrain$label)~as.matrix(sample.dat),data=as.
data.frame(sample.dat),kmax=10,distance=2,
c("rectangular","triangular","epanechnikov","gaussian","rank","optimal"))
V=min(result$MISCLASS)
return(V)
})
#PC between 40 and 60 is the best
#Find the best PC
b=lapply(40:60,function (x) {
  sample.dat=PC$scores[,1:x]
result=train.kknn(as.factor(sampleTrain$label)~as.matrix(sample.dat),data=as.
data.frame(sample.dat),kmax=10,distance=2,
c("rectangular","triangular","epanechnikov","gaussian","rank","optimal"))
```

```
  V=min(result$MISCLASS)
  return(V)
})
par(mfrow=c(1,2))
plot(x=seq(30,180,10),y=unlist(a),type='l',xlab="# of PCs",ylab="Error
Rate",main="Best PC for KNN")
plot(x=40:60,y=unlist(b),type='l',xlab="# of PCs",ylab="Error
Rate",main="Best PC for KNN")
which(b==min(unlist(b))) #reports that 18th obs is the minimum -> 57 PC's
PTRAIN=as.data.frame(PC$scores[,1:57])
KNN.CV=train.kknn(as.factor(sampleTrain$label)~as.matrix(PTRAN),
                            data=PTRAIN,kmax=10,distance=2,
c("rectangular","triangular","epanechnikov","gaussian","rank","optimal"))
#The result claims the best k=9 with misclassification=0.02485714 with
triangular kernel
KK=KNN.CV$MISCLASS
plot(x=1:10,y=KK[,1],type='l',lty=1,main="Error rate of KNN Cross
Validation",ylab="Error Rate",xlab="Number of K",ylim=c(0.02,0.04))
points(KK[,2],lty=1,type='l',col='red')
points(KK[,3],lty=1,type='l',col='blue')
points(KK[,4],lty=1,type='l',col='green')
points(KK[,5],lty=1,type='l',col='purple')
points(KK[,6],lty=1,type='l',col='orange')
legend("top",c("rectangular","triangular","epanechnikov","gaussian","rank","o
ptimal"),title="Kernel",fill=c("black","red","blue","green","purple","orange"
),lty=c(1,1,1,1,1,1),)
#KNN on the Test data
t.dat=as.matrix(test)%*%PC$loadings[,1:57]
PTEST=as.data.frame(t.dat)
KNN=kknn(as.factor(sampleTrain$label)~.,PTRAIN,PTEST,k=10,distance=2,kernel="
triangular")
ConfMat3=confusionMatrix(KNN$fitted.values,sampleTest$label)
#End of KNN R code
```

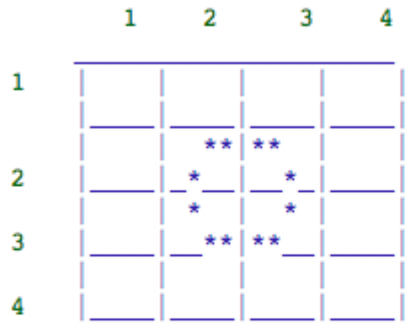
A3. SVM R code

```
setwd("~/Desktop/Winter_2014/STA_135/Project/SVM")
print(load('~/Desktop/Winter_2014/STA_135/Project/SVM/digitsTrain.rda'))
print(load('~/Desktop/Winter_2014/STA_135/Project/SVM/digitsTest.rda'))
training <- sampleTrain
testing <- newTest
library("e1071")
library("caret")

# Fit models with different kernels and parameters
fit1 <- svm(train, label, kernel = "radial", cost = 1, cross = 5)
fit2 <- svm(train, label, kernel = "radial", cost = 100, cross = 5)
fit3 <- svm(train, label, kernel = "radial", cost = 300, cross = 5)
fit4 <- svm(train, label, kernel = "radial", cost = 670, cross = 5)
fit5 <- svm(train, label, kernel = "radial", cost = 1000, cross = 5)
```

```
fit6 <- svm(train, label, kernel = "radial", cost = .25, cross = 5)
fit7 <- svm(train, label, kernel = "radial", cost = 100, cross = 5, gamma =
2^(-15))
fit8 <- svm(train, label, kernel = "radial", cost = 100, cross = 5, gamma =
2^(-10))
fit9 <- svm(train, label, kernel = "radial", cost = 100, cross = 5, gamma =
2^(-5))
fit12 <- svm(train, label, kernel = "linear", cost = 100, cross = 5)
fit13 <- svm(train, label, kernel = "polynomial", degree = 1, cost = 100,
cross = 5)
fit14 <- svm(train, label, kernel = "polynomial", degree = 2, cost = 100,
cross = 5)
fit15 <- svm(train, label, kernel = "polynomial", degree = 3, cost = 100,
cross = 5)
fit17 <- svm(train, label, kernel = "sigmoid", cost = 100, cross = 5)
save(fit1, fit2, fit3, fit4, fit5, fit6, file = "5fitstest3_15_14.Rda")
save(fit7, fit8, fit9, fit12, fit13, fit14, fit15, fit17, file =
"5fitstest3_17_14.Rda")

# Plot kernels against each other with default parameters
rad <- fit1$tot.accuracy
lin <- fit12$tot.accuracy
pol <- fit13$tot.accuracy
sig <- fit17$tot.accuracy
kerns <- rbind(rad, lin, pol, sig)
plot(kerns, col = c("red", "green", "blue", "orange"), pch = 16, cex = 3,
axes = FALSE, xlab = "Kernel", ylab = "Average Percent Correctly Predicted",
main = "Average Percent Correctly Predicted\nAfter 5-fold Cross
Validation\nFor Different Kernels\nWith Other Parameters as Default")
axis(1, at=1:4, labels = c("Radial", "Linear", "Polynomial", "Sigmoid"))
axis(2, at=75:100)

# Plot gammas against each other with C = 100
gdefault <- fit2$tot.accuracy
g215 <- fit7$tot.accuracy
g210 <- fit8$tot.accuracy
g205 <- fit9$tot.accuracy
gammas <- rbind(g215, g210, g205, gdefault)
plot(gammas, col = c("red", "green", "blue", "orange"), pch = 2, lwd = 3, cex
= 3, axes = FALSE, xlab = "Gamma Value", ylab = "Average Percent Correctly
Predicted", main = "Average Percent Correctly Predicted\nAfter 5-fold Cross
Validation\nFor Different Gamma Values")
axis(1, at=1:4, labels = c(expression(2^(-15)), expression(2^(-10)),
expression(2^(-5)), expression(1/ncol(training))))
axis(2, at=75:110)

# Use on testing data
test <- as.matrix(testing[,-1]) %*% pc$loadings[,1:59]
#label <- as.factor(training[,1])
```

```
labels <- as.factor(testing[,1])
bestfit <- svm(train, label, kernel = "radial", cost = 100)
pred <- predict(bestfit, test)
confusionMatrix(pred, labels)
```

A4. Blocking R Code



```
# Isolate pixels
chunking <- function(q, dataset = training) {
      data <- dataset[ , 2:785]
      # Pixels will be 784
      pixels <- ncol(data) # number of total pixels (784)
      # nc will be sqrt(784) = 28
      nc <- floor(sqrt(pixels))
      # Number of rows/cols per quadrants
      r = floor(nc / q) # number of rows per quadrant
      # How many rows/cols will be left over
      takeaway <- nc - (r * q)
      # initial matrix that is nc X nc
      initial.mat <- matrix(1:(nc^2), nrow = 28, byrow = TRUE)
      # If there are left over rows/cols, get rid of those variable nums
      nums <- initial.mat[1:(nrow(initial.mat) - takeaway),
1:(ncol(initial.mat) - takeaway)]
      nums <- as.numeric(nums)
      # Take those variables out of the data
      data <- data[ , nums]
      # Pixels will be 784
      pixels <- ncol(data)
      # nc
      nc <- floor(sqrt(pixels))
      # Number of rows/cols per quadrants
      r = floor(nc / q) # number of rows per quadrant
      # Initialize matrix that has a new column for each q X q variable
      mat <- matrix(0, nrow = nrow(data), ncol = q * q)
      #
      # Start looping
      counter <- 1
      for (qc in 0:(q-1)) {
            for (qr in 0:(q-1)) {
                  rows <- NULL
                  for (ri in 0:(r-1)) {
                         rows <- c(rows, ((qr * r + ri)*nc+1+qc*r):((qr * r +
ri)*nc+r+qc*r))
                  }
                  mat[ , counter] <- rowSums(data[ , rows])
                  counter <- counter + 1
            }
      }
      mattable <- as.data.frame(mat)
}

fourteen <- chunking(14, training) # 14 x 14 = 196
#.
#.
#.
seven <- chunking(7, training) # 2.45 minutes? 7 x 7 = 49
#.
#.
#.
four <- chunking(4, training) # 2.526 seconds? 4 x 4 = 16
```

A5. Naive Bayes R Code

```
# Load data, libraries
print(load('~/Desktop/Winter_2014/STA_135/Project/SVM/digitsTrain.rda'))
print(load('~/Desktop/Winter_2014/STA_135/Project/SVM/digitsTest.rda'))
training <- sampleTrain
testing <- newTest
rm(sampleTrain, newTest)
library("e1071")
library("MASS")
library("nortest")
library("moments")
library("MVN")
library("caret")

# Principal components
pc <- princomp(training[,-1])

# Code to produce /Naive_Bayes/cm.Rda
{
# train:    28000
# pcs:            59
data <- training[,-1]
labels <- as.factor(training[,1])
testlabels <- as.factor(testing[,1])
reduced <- pc$scores[,1:59]
test <- as.matrix(testing[,-1]) %*% pc$loadings[,1:59]
model <- naiveBayes(reduced, labels)
predtest <- predict(model, test) # 3 min
cm_bayes <- confusionMatrix(predtest, testlabels)
1 - cm_bayes$byClass[,"Sensitivity"]
}

# Naive Bayes with blocking with 5-fold cv
# End up not using blocking because it performs very badly on training data.

# Find split indices for doing CV
indices <- nrow(training)/5
########################################
# 14 x 14 # 122 * 5 = 610 seconds = 10 minutes ish
########################################
start <- proc.time()
# 122 seconds
model14_1 <- naiveBayes(fourteen, labels, subset = setdiff(1:nrow(training),
1*(1:indices)))
pred14_1 <- predict(model14_1, fourteen[1*(1:indices),])
cm14_1 <- confusionMatrix(pred14_1, labels[1*(1:indices)]) # .4987

model14_2 <- naiveBayes(fourteen, labels, subset = setdiff(1:nrow(training),
2*(1:indices)))
pred14_2 <- predict(model14_2, fourteen[2*(1:indices),])
cm14_2 <- confusionMatrix(pred14_2, labels[2*(1:indices)]) # .4964

model14_3 <- naiveBayes(fourteen, labels, subset = setdiff(1:nrow(training),
3*(1:indices)))
pred14_3 <- predict(model14_3, fourteen[3*(1:indices),])
cm14_3 <- confusionMatrix(pred14_3, labels[3*(1:indices)]) # .5005

model14_4 <- naiveBayes(fourteen, labels, subset = setdiff(1:nrow(training),
4*(1:indices)))
pred14_4 <- predict(model14_4, fourteen[4*(1:indices),])
cm14_4 <- confusionMatrix(pred14_4, labels[4*(1:indices)]) # .4918

model14_5 <- naiveBayes(fourteen, labels, subset = setdiff(1:nrow(training),
5*(1:indices)))
pred14_5 <- predict(model14_5, fourteen[5*(1:indices),])
cm14_5 <- confusionMatrix(pred14_5, labels[5*(1:indices)]) # .507

avgaccuracy14 <- mean(c(cm14_1$overall[1], cm14_2$overall[1],
cm14_3$overall[1], cm14_4$overall[1], cm14_5$overall[1]))
########################################
# 7 x 7 # 32 * 5 = 160 seconds = 2min40sec
########################################
# 32 seconds
model7_1 <- naiveBayes(seven, labels, subset = setdiff(1:nrow(training),
1*(1:indices)))
pred7_1 <- predict(model7_1, seven[1*(1:indices),])
```

```
cm7_1 <- confusionMatrix(pred7_1, labels[1*(1:indices)]) # .4987

model7_2 <- naiveBayes(seven, labels, subset = setdiff(1:nrow(training),
2*(1:indices)))
pred7_2 <- predict(model7_2, seven[2*(1:indices),])
cm7_2 <- confusionMatrix(pred7_2, labels[2*(1:indices)])

model7_3 <- naiveBayes(seven, labels, subset = setdiff(1:nrow(training),
3*(1:indices)))
pred7_3 <- predict(model7_3, seven[3*(1:indices),])
cm7_3 <- confusionMatrix(pred7_3, labels[3*(1:indices)])

model7_4 <- naiveBayes(seven, labels, subset = setdiff(1:nrow(training),
4*(1:indices)))
pred7_4 <- predict(model7_4, seven[4*(1:indices),])
cm7_4 <- confusionMatrix(pred7_4, labels[4*(1:indices)])

model7_5 <- naiveBayes(seven, labels, subset = setdiff(1:nrow(training),
5*(1:indices)))
pred7_5 <- predict(model7_5, seven[5*(1:indices),])
cm7_5 <- confusionMatrix(pred7_5, labels[5*(1:indices)])

avgaccuracy7 <- mean(c(cm7_1$overall[1], cm7_2$overall[1], cm7_3$overall[1],
cm7_4$overall[1], cm7_5$overall[1]))

########################################
# 4 x 4 # 12 * 5 = 60 seconds = 1 minute
########################################
# 12 seconds
model4_1 <- naiveBayes(four, labels, subset = setdiff(1:nrow(training),
1*(1:indices)))
pred4_1 <- predict(model4_1, four[1*(1:indices),])
cm4_1 <- confusionMatrix(pred4_1, labels[1*(1:indices)]) # .4987

model4_2 <- naiveBayes(four, labels, subset = setdiff(1:nrow(training),
2*(1:indices)))
pred4_2 <- predict(model4_2, four[2*(1:indices),])
cm4_2 <- confusionMatrix(pred4_2, labels[2*(1:indices)])

model4_3 <- naiveBayes(four, labels, subset = setdiff(1:nrow(training),
3*(1:indices)))
pred4_3 <- predict(model4_3, four[3*(1:indices),])
cm4_3 <- confusionMatrix(pred4_3, labels[3*(1:indices)])

model4_4 <- naiveBayes(four, labels, subset = setdiff(1:nrow(training),
4*(1:indices)))
pred4_4 <- predict(model4_4, four[4*(1:indices),])
cm4_4 <- confusionMatrix(pred4_4, labels[4*(1:indices)])
```

```
model4_5 <- naiveBayes(four, labels, subset = setdiff(1:nrow(training),
5*(1:indices)))
pred4_5 <- predict(model4_5, four[5*(1:indices),])
cm4_5 <- confusionMatrix(pred4_5, labels[5*(1:indices)])

avgaccuracy4 <- mean(c(cm4_1$overall[1], cm4_2$overall[1], cm4_3$overall[1],
cm4_4$overall[1], cm4_5$overall[1]))

# Compare average accuraies
avgaccuracy14; avgaccuracy7; avgaccuracy4;
```

A6. Random Forest R Code

```
#preprocess
library(parallel)
library('randomForest', lib.loc='/home/mhm/MyRlib')
print(load('digits.rda'))
pvar=cumsum(pc[[1]]^2)/sum(pc[[1]]^2)
k=length(which(pvar<0.85))+1
trainpc=pc[[6]][, 1:k]
testpc=predict(pc, test)[ ,1:k]
trainch=chunk(train, 7)
testch=chunk(test, 7)
#rf
digitrf=function(train, label, test, testlabel, multi=20, mtry=8,
replace=TRUE, every=TRUE) { #This is for parallel random forest
 seed=floor(10000*runif(30))
 cl=makeCluster(30, 'FORK')
 rf=clusterApply(cl, seed, function(i) {
  set.seed(i)
  randomForest(train, label, ntree=multi, mtry=mtry, replace=replace, norm=F)
 })
 stopCluster(cl)
 rf=do.call('combine', rf)
 pred=predict(rf, test, predict=T)
 error=length(which(pred[[1]]!=testlabel))/length(testlabel)
 confusion=table(testlabel, pred[[1]])
 if (every) {
  ntree=30*multi
  everytree=matrix(0, ntree, 10)
  for (i in 1:ntree) {
   t=table(testlabel, pred[[2]][, i])
   everytree[i, ]=1-diag(t)/colSums(t)
  }
 } else {everytree=NULL}
 return(list(predict=pred, error=error, confusion=confusion,
everytree=everytree))
}
rf1=digitrf(trainpc, label, testpc, testlabel)
```

```
rf2=digitrf(trainch, label, testch, testlabel)
#choose best PCs and chunks
digitrf.cv=function(data, label, p=0.8, multi=10, mtry=8, replace=TRUE) {
 n=nrow(data)
 ind=sample(1:n)
 data=data[ind, ]
 label=label[ind]
 tr=floor(p*n)
 d1=data[1:tr, ]
 l1=label[1:tr]
 d2=data[(tr+1):n, ]
 l2=label[(tr+1):n]
 rf=digitrf(d1, l1, d2, l2, multi=multi, mtry=mtry, replace=replace, every=F)
 return(list(error=rf[[2]], confusion=rf[[3]]))
}
errch=numeric(11)
for (i in 4:14) {
 data=chunk(train, i)
 rfcv=digitrf.cv(data, label)
 errch[i-3]=rfcv[[1]]
}
errpc1=numeric(19)
for (k in seq(130, 200, 10)) {
 data=pc[[6]][, 1:k]
 rfcv=digitrf.cv(data, label)
 errpc1[k/10-1]=rfcv[[1]]
}
errpc2=numeric(11)
for (k in seq(75, 85, 1)) {
 data=pc[[6]][, 1:k]
 rfcv=digitrf.cv(data, label)
 errpc2[k-74]=rfcv[[1]]
}
trainch=chunk(train, 12)
errmtry1=numeric(7)
for (m in seq(6, 18, 2)) {
 rfcv=digitrf.cv(trainch, label, mtry=m)
 errmtry1[m/2-2]=rfcv[[1]]
}
#plot random forest results
library(RColorBrewer)
print(load('error.rda'))
color=brewer.pal(10, 'Paired')
par(mfrow=c(1, 2))
matplot(m1, type='l', lty=1, col=color, ylim=c(0, 0.6), main='Error Rate of
Single Tree (PCA)', xlab='tree', ylab='error rate')
legend('top', legend=0:9, lty=1, col=color, ncol=5, cex=0.9)
matplot(m2, type='l', lty=1, col=color, ylim=c(0, 0.6), main='Error Rate of
Single Tree (Shrink)', xlab='tree', ylab='error rate')
```

```
legend('top', legend=0:9, lty=1, col=color, ncol=5, cex=0.9)
order(colMeans(m1))-1
order(colMeans(m2))-1
plot(4:14, errch, type='l', main='Best Chunk for Random Forest', xlab='# of
chunks', ylab='error rate')
par(mfrow=c(1, 2))
plot(seq(20, 200, 10), errpc1, type='l', ylim=c(0.051, 0.068), main='Best PC
for Random Forest', xlab='# of PCs', ylab='error rate')
plot(75:85, errpc2, type='l', ylim=c(0.051, 0.068), main='Best PC for Random
Forest', xlab='# of PCs', ylab='error rate')
```

A7. Digits Image R Code

```
digitimage=function(k, data=sampleTrain, color=grey(0:255/255)) {
 x=as.numeric(data[k, 2:785])
 j=0:27
 plot(c(0, 28), c(0, 28), type='n', xaxt='n', yaxt='n', bty='n', xlab='',
ylab='')
 sapply(0:27, function(i) rect(j, 27-i, j+1, 28-i, col=color[256-x[28*i+j+1]],
border=NA))
 text(26, 3, labels=print(data[k, 1]), cex=0.7)
}
par(mfrow=c(10, 10), mar=c(1, 1, 1, 1))
sapply(1:100, digitimage) #Draw first 100 digits.
```