SVMs, kNN, and Random Forest for Handwriting Recognition

Sudhanshu Kulkarni

# Introduction

The data set comes from the Kaggle Digit Recognizer competition. The goal is to recognize digits 0 to 9 in handwriting images by constructing the prediction models using kNN, SVM and Random Forest Model and compare them with Naïve Bayes and decision tree algorithms.

## Importing Libraries

First, we begin with installing and importing libraries. To help us with sampling the data, We will use the “dplyr” library. Also, to filter data, we will use “tidyverse” library. To convert the dataset from numeric to nominal we will use the RWeka interface for which we have imported “RWeka” library. To build and evaluate kNN Model we will use the “class” and “gmodels” libraries. Similarly, we will use “randomForest” to build Random Forest model. For building SVM model we will use “kernlab”.

library(dplyr)

##   
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':  
##   
## filter, lag

## The following objects are masked from 'package:base':  
##   
## intersect, setdiff, setequal, union

library(class)  
library(gmodels)  
library(RWeka)  
library(tidyverse)

## -- Attaching packages ----------------------------------------------------------------------------------------------------------------- tidyverse 1.2.1 --

## v ggplot2 3.1.0 v readr 1.3.1  
## v tibble 2.1.1 v purrr 0.3.2  
## v tidyr 0.8.3 v stringr 1.4.0  
## v ggplot2 3.1.0 v forcats 0.4.0

## -- Conflicts -------------------------------------------------------------------------------------------------------------------- tidyverse\_conflicts() --  
## x dplyr::filter() masks stats::filter()  
## x dplyr::lag() masks stats::lag()

library(kernlab)

##   
## Attaching package: 'kernlab'

## The following object is masked from 'package:purrr':  
##   
## cross

## The following object is masked from 'package:ggplot2':  
##   
## alpha

library(randomForest)

## randomForest 4.6-14

## Type rfNews() to see new features/changes/bug fixes.

##   
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':  
##   
## margin

## The following object is masked from 'package:dplyr':  
##   
## combine

## Loading and Sampling Data

Then, we load the csv data into two data sets i.e. trainData and testData to train and predict respectively. Since the original data set is too large to be loaded in RStudio, we will randomly sample 1400 observations of trianData, called sampleTrainData, and 840 (35% of total dataset) observations of the testData, called sampleTestData. We then check the structures to know if any data preprocessing is required.

setwd("C:\\Sudhanshu\\SU\\Semester 2\\707\\Assignment 7\\")  
trainData <- read.csv("Kaggle-digit-train.csv")  
sampleTrainData <- sample\_n(trainData,1400)  
str(sampleTrainData)

testData <- read.csv("Kaggle-digit-test.csv")  
sampleTestData <- sample\_n(testData,840)  
str(sampleTestData)

## Data Preprocessing

Now, we will convert numeric data type to nominal data type using the RWeka filter interface so the models can be built on nominal values. Followed by this, we will separate train and test data sets into the pixel information (sampleTrainDataNormalizedP, sampleTestDataNormalizedP) and label information (trainLabels, testLabels). Then we will again check the structure of the normalized data sets to verify.

NN <- make\_Weka\_filter("weka/filters/unsupervised/attribute/NumericToNominal")   
sampleTrainDataNormalized <- NN(data=sampleTrainData, control= Weka\_control(R="1-3"), na.action = NULL)  
trainLabels <- sampleTrainDataNormalized %>% select(starts\_with("label"))  
sampleTrainDataNormalizedP <- sampleTrainDataNormalized %>% select(starts\_with("pixel"))  
str(sampleTrainDataNormalized)

sampleTestDataNormalized <- NN(data=sampleTestData, control= Weka\_control(R="1,3"), na.action = NULL)  
testLabels <- sampleTestDataNormalized %>% select(starts\_with("label"))  
sampleTestDataNormalizedP <- sampleTestDataNormalized %>% select(starts\_with("pixel"))  
str(sampleTestDataNormalized)

# kNN Model

kNN or  k-Nearest Neighbor model is a supervised learning algorithm. This algorithm is also easy to learn and apply. To build the kNN model we will use the knn function from the class package.

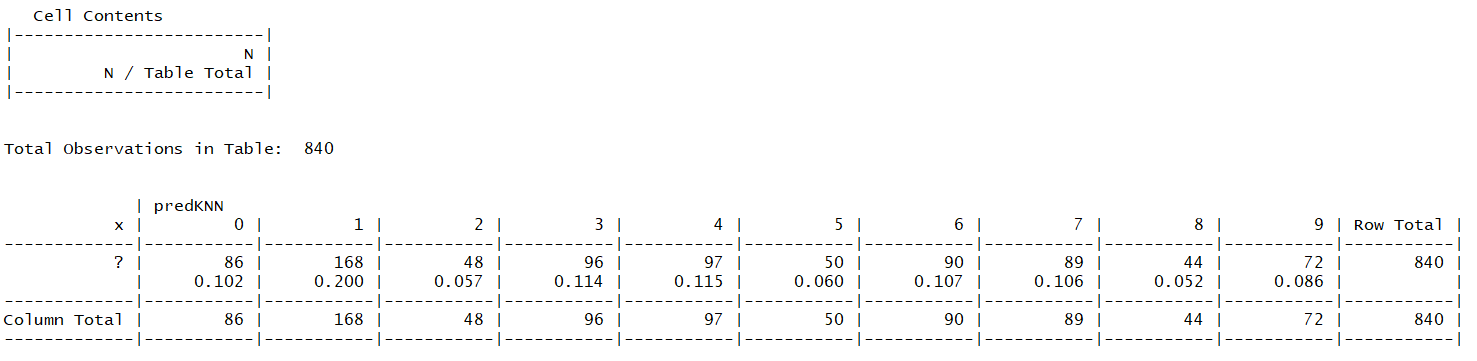
cl = trainLabels[,1]  
predKNN <- knn(train = sampleTrainDataNormalizedP,test = sampleTestDataNormalizedP, cl, k = 10)  
predKNN

## [1] 2 0 6 6 4 7 7 7 3 6 1 8 8 8 6 8 8 9 9 7 1 0 8 2 8 8 4 5 1 5 4 8 8 0 5  
## [36] 6 1 5 5 4 4 2 4 8 1 1 8 6 6 4 7 2 5 4 5 3 8 1 7 8 1 5 7 4 5 3 8 4 4 5  
## [71] 2 4 0 4 6 1 2 4 3 7 3 0 3 5 1 7 3 4 9 4 0 9 5 9 5 3 6 9 6 6 1 0 2 5 0  
## [106] 9 6 6 6 9 1 3 3 6 4 4 0 4 0 6 2 0 8 9 5 1 7 7 0 4 7 2 2 9 9 2 5 7 8 7  
## [141] 3 4 8 8 8 4 8 8 1 5 3 5 4 0 7 4 6 4 6 8 7 3 3 0 9 5 0 4 2 0 5 7 9 3 3  
## [176] 1 2 6 6 7 8 5 6 1 7 6 5 9 2 6 1 8 0 7 7 1 5 7 4 6 0 3 2 1 8 5 8 0 1 4  
## [211] 2 7 7 6 2 8 1 0 8 2 7 3 7 6 0 9 8 9 2 9 3 8 2 0 7 1 9 7 0 3 0 7 8 2 6  
## [246] 6 5 9 4 8 5 0 1 4 0 1 8 4 7 2 7 3 1 0 1 5 1 6 6 2 2 4 0 5 2 9 7 1 7 0  
## [281] 1 7 0 3 7 7 1 2 1 6 7 3 2 9 1 7 0 4 4 7 6 5 7 4 1 9 5 3 0 0 6 8 0 4 1  
## [316] 7 8 1 6 6 3 1 6 3 8 6 7 8 8 6 3 0 1 8 7 7 2 1 6 9 4 5 2 8 3 2 0 7 1 1  
## [351] 2 0 1 0 7 5 7 0 6 0 9 4 0 1 8 3 0 2 2 5 3 9 9 3 3 5 6 1 7 0 1 7 9 8 8  
## [386] 6 8 9 5 3 3 6 3 7 6 2 7 2 3 6 8 1 0 3 2 2 4 1 7 0 4 8 4 0 3 1 1 3 7 7  
## [421] 5 9 1 9 4 6 9 1 2 6 7 1 8 6 0 9 1 3 8 7 7 9 4 1 5 2 1 3 5 9 6 0 6 1 4  
## [456] 2 2 8 7 0 3 6 0 1 1 3 2 1 9 6 2 3 6 1 0 0 3 4 7 3 0 0 1 8 8 9 1 8 4 5  
## [491] 9 9 7 8 7 1 0 3 9 1 7 6 4 1 1 1 7 0 6 1 9 8 4 1 9 4 5 8 5 1 8 4 4 6 7  
## [526] 0 5 4 1 1 1 7 0 1 4 5 2 6 0 5 4 0 3 3 2 8 6 6 8 1 7 0 5 1 6 8 4 4 8 0  
## [561] 2 5 3 4 1 9 7 1 2 9 4 0 7 9 6 4 1 1 6 4 9 8 9 7 2 2 6 0 8 3 9 1 8 9 1  
## [596] 7 3 4 9 7 1 7 1 5 7 9 1 2 7 7 1 8 6 1 7 5 6 7 9 4 2 4 6 5 1 3 1 5 3 4  
## [631] 8 2 4 2 1 6 7 0 8 8 7 1 1 6 2 5 4 8 2 8 0 0 9 1 2 8 1 7 9 1 4 2 1 4 9  
## [666] 1 7 6 6 5 8 5 5 4 2 3 5 1 5 2 6 8 1 9 5 6 4 9 7 4 7 5 5 1 7 3 8 4 1 3  
## [701] 0 9 8 4 6 0 8 8 1 6 2 6 9 7 2 4 7 1 9 9 9 3 0 4 2 0 4 8 3 7 3 0 9 9 3  
## [736] 4 2 6 2 2 0 1 5 1 8 3 7 7 4 7 2 2 8 0 8 0 5 2 1 3 5 6 8 0 9 3 4 7 8 3  
## [771] 4 7 6 7 5 3 7 5 6 5 0 8 1 0 5 0 3 9 0 9 1 3 3 7 9 6 1 9 7 5 1 6 9 8 4  
## [806] 2 6 1 4 5 8 2 6 3 3 9 5 7 0 0 1 6 2 8 6 5 6 2 6 6 4 0 6 0 0 3 7 1 5 9  
## Levels: 0 1 2 3 4 5 6 7 8 9

The Selection of k will determine how well the data can be utilized to generalize the results of the kNN algorithm. A large k value has benefits which include reducing the variance due to the noisy data but this develops a bias due to which the learner tends to ignore the smaller patterns which may have useful insights.

To evaluate the model accuracy we will utilize the CrossTable from gmodels package.

x=testLabels[,1]  
CrossTable(x, y=predKNN, prop.chisq = FALSE)



In the above table we can see the how many instances of each digit has been classified along with the error percentage. As we can see the error percentage lies between 5.7% tp 20%. The average error rate is 9.99%.

# Random Forest Model

Random Forest is one very powerful ensembling machine learning algorithm which works by creating multiple decision trees and then combining the output generated by each of the decision trees. Random Forest works on the same weak learners. It combines the output of multiple decision trees and then finally come up with its own output. Random Forest works on the same principle as Decision Tress; however, it does not select all the data points and variables in each of the trees. It randomly samples data points and variables in each of the tree that it creates and then combines the output at the end. It removes the bias that a decision tree model might introduce in the system. Also, it improves the predictive power significantly.

Now we will proceed with building the decision with default parameters.

rf.model1 <- randomForest(label ~ ., data = sampleTrainDataNormalized, importance = TRUE)  
rf.model1

##   
## Call:  
## randomForest(formula = label ~ ., data = sampleTrainDataNormalized, importance = TRUE)   
## Type of random forest: classification  
## Number of trees: 500  
## No. of variables tried at each split: 28  
##   
## OOB estimate of error rate: 8.64%  
## Confusion matrix:  
## 0 1 2 3 4 5 6 7 8 9 class.error  
## 0 141 0 0 0 1 0 2 1 1 0 0.03424658  
## 1 0 132 1 3 0 0 0 0 0 0 0.02941176  
## 2 0 2 132 2 2 1 1 4 2 0 0.09589041  
## 3 1 1 3 122 0 4 1 2 2 2 0.11594203  
## 4 0 0 0 0 159 0 1 0 0 2 0.01851852  
## 5 2 0 1 11 2 111 2 1 2 2 0.17164179  
## 6 1 0 0 0 1 0 141 0 0 0 0.01398601  
## 7 0 3 3 1 5 1 0 134 1 1 0.10067114  
## 8 0 4 1 9 3 3 2 0 109 1 0.17424242  
## 9 1 1 1 2 8 0 0 3 0 98 0.14035088

Now, we will tune the parameters of the random forest to get better results. We will get change the mtry value to 12.

rf.model2 <- randomForest(label ~ ., data = sampleTrainDataNormalized, ntree = 500, mtry = 12, importance = TRUE)  
rf.model2

##   
## Call:  
## randomForest(formula = label ~ ., data = sampleTrainDataNormalized, ntree = 500, mtry = 12, importance = TRUE)   
## Type of random forest: classification  
## Number of trees: 500  
## No. of variables tried at each split: 12  
##   
## OOB estimate of error rate: 8.14%  
## Confusion matrix:  
## 0 1 2 3 4 5 6 7 8 9 class.error  
## 0 142 0 0 0 1 1 0 1 1 0 0.02739726  
## 1 0 132 2 1 0 1 0 0 0 0 0.02941176  
## 2 0 2 133 2 2 1 0 6 0 0 0.08904110  
## 3 1 1 4 119 1 5 1 2 3 1 0.13768116  
## 4 0 0 0 0 156 1 1 2 0 2 0.03703704  
## 5 2 0 0 6 1 117 3 2 2 1 0.12686567  
## 6 1 0 1 0 1 0 140 0 0 0 0.02097902  
## 7 0 2 3 1 3 2 0 136 1 1 0.08724832  
## 8 0 3 1 9 1 2 1 0 113 2 0.14393939  
## 9 0 1 1 3 7 1 0 3 0 98 0.14035088

Now we will predict the results and check the accuracy on the train data set followed by test data set.

# Predicting on train set  
predTrain <- predict(rf.model2, sampleTrainDataNormalized, type = "class")  
# Checking classification accuracy  
table(predTrain, sampleTrainDataNormalized$label)

##   
## predTrain 0 1 2 3 4 5 6 7 8 9  
## 0 146 0 0 0 0 0 0 0 0 0  
## 1 0 136 0 0 0 0 0 0 0 0  
## 2 0 0 146 0 0 0 0 0 0 0  
## 3 0 0 0 138 0 0 0 0 0 0  
## 4 0 0 0 0 162 0 0 0 0 0  
## 5 0 0 0 0 0 134 0 0 0 0  
## 6 0 0 0 0 0 0 143 0 0 0  
## 7 0 0 0 0 0 0 0 149 0 0  
## 8 0 0 0 0 0 0 0 0 132 0  
## 9 0 0 0 0 0 0 0 0 0 114

# Predicting on Validation set  
predValid <- predict(rf.model2, sampleTestDataNormalized, type = "class")  
table(predValid,sampleTestDataNormalized$label)

##   
## predValid ?  
## 0 89  
## 1 91  
## 2 82  
## 3 72  
## 4 93  
## 5 68  
## 6 86  
## 7 97  
## 8 97  
## 9 65

# SVM Model

Support Vector Machines are an excellent tool for classification, novelty detection and regression. We will build the SVM model on train data set.

#Building the SVM model.  
svm.model <- ksvm(label~., data = sampleTrainData, kernel = "rbfdot", kpar="automatic", C=50, cross=50, prob.model=TRUE  
 , type = "C-svc")

## Warning in .local(x, ...): Variable(s) `' constant. Cannot scale data.

svm.model

## Support Vector Machine object of class "ksvm"   
##   
## SV type: C-svc (classification)   
## parameter : cost C = 50   
##   
## Gaussian Radial Basis kernel function.   
## Hyperparameter : sigma = 1.60817778747228e-07   
##   
## Number of Support Vectors : 961   
##   
## Objective Function Value : -11.8037 -30.0038 -28.1528 -24.1359 -41.272 -31.3718 -25.2153 -29.9629 -24.6442 -54.1006 -42.7603 -20.6857 -25.8412 -19.3104 -52.7064 -48.6617 -24.7294 -70.0809 -42.2649 -43.5429 -42.3136 -67.5471 -57.7022 -43.9478 -31.2693 -94.7577 -28.4324 -49.746 -82.5852 -50.5427 -43.9162 -34.3859 -64.3686 -43.2771 -102.2224 -46.9576 -43.869 -66.7156 -51.2195 -23.2776 -30.8152 -23.6422 -48.4982 -113.8295 -52.2612   
## Training error : 0   
## Cross validation error : 0.081429   
## Probability model included.

After building the model, we will predict and classify the test data set.

svm.Pred <- predict(svm.model,sampleTestData)  
  
table(svm.Pred,sampleTestData$label)

##   
## svm.Pred ?  
## 0 84  
## 1 87  
## 2 86  
## 3 75  
## 4 86  
## 5 71  
## 6 89  
## 7 95  
## 8 94  
## 9 73

# Conclusion

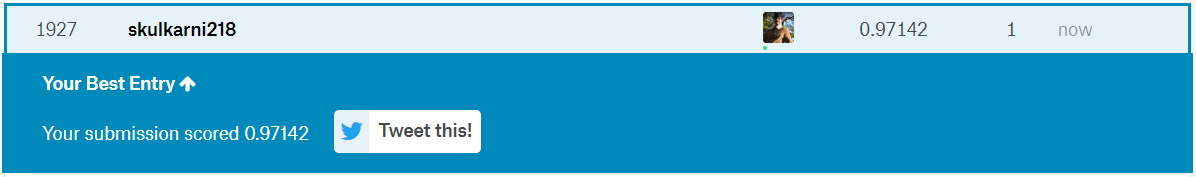
The inaccuracies of the kNN, SVM and Random Forest model can be seen in the table below.

|  |  |
| --- | --- |
| **Model** | **Inaccuracy** |
| kNN | 9.99% |
| SVM | 0% |
| Random Forest | 8.14% |

So, we can conclude that SVM model is the best model for handwriting recognition as it gives us the least inaccuracy. The conclusion which we have reached is correct as SVMs are good for text mining and image recognition. Also, SVMs are tolerant to noise and have been successful on various real-world data problems.

# Kaggle Submission

Below is the screenshot of Kaggle submission.



Here I achieved accuracy of 97.14%.