Applied Data Science: Midterm Project

Authors: Hanao Li, Gengyu Zhang, Yiyang Fu

March 12, 2019

Introduction

In this midterm project, we will use image data from the MNIST Fashin Database to classify different types of apparels by constructing ten machine learning models. In order to have faster and more tractable computations, the original pixel of 28x28 was condensed to 7x7. We will choose three sample sizes from the training dataset and then we will generate three model development sets from those three sample sizes and compare the performance for each machine learning model using the sample sets we generated from the training dataset. The performance is based on the sample sizes, the computation time and the accuracy using the testing dataset with a weight of 0.25, 0.25 and 0.5, respectively. So our goal is to have the highest accuracy while maintaining the lowest computational time and smallest datasets. We will use the difference in “proc.time()” to calculate the time elapsed for all models and we will use caret package (train function) to train the model and tune the parameters. But we choose to tune off the parameter tuning step for all our models that have parameters and it will be discussed below in the Discussion section.

Load Libraries

```
library(dplyr)
library(randomForest)
library(caret)
library(MASS)
library(gbm)
library(e1071)
library(glmnet)
library(class)
library(xgboost)
library(nnet)
library(rpart)
library(pracma)
library(DT)
library(data.table)
library(glmnet)
```

Read in data

```
#Read Data
setwd("~/Desktop/R")
training <- (read.csv("Train.csv", header=T))
testing <- (read.csv("Test.csv", header=T))

#Check Missing
unlist(lapply(training, function(x){sum(is.na(x))}))
```

```
##   label  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8
##       0       0       0       0       0       0       0       0       0
##  pixel9 pixel10 pixel11 pixel12 pixel13 pixel14 pixel15 pixel16 pixel17
##       0       0       0       0       0       0       0       0       0
## pixel18 pixel19 pixel20 pixel21 pixel22 pixel23 pixel24 pixel25 pixel26
##       0       0       0       0       0       0       0       0       0
## pixel27 pixel28 pixel29 pixel30 pixel31 pixel32 pixel33 pixel34 pixel35
##       0       0       0       0       0       0       0       0       0
## pixel36 pixel37 pixel38 pixel39 pixel40 pixel41 pixel42 pixel43 pixel44
##       0       0       0       0       0       0       0       0       0
## pixel45 pixel46 pixel47 pixel48 pixel49
##       0       0       0       0       0
```

```
unlist(lapply(testing, function(x){sum(is.na(x))}))
```

```
##   label  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8
##       0       0       0       0       0       0       0       0       0
##  pixel9 pixel10 pixel11 pixel12 pixel13 pixel14 pixel15 pixel16 pixel17
##       0       0       0       0       0       0       0       0       0
## pixel18 pixel19 pixel20 pixel21 pixel22 pixel23 pixel24 pixel25 pixel26
##       0       0       0       0       0       0       0       0       0
## pixel27 pixel28 pixel29 pixel30 pixel31 pixel32 pixel33 pixel34 pixel35
##       0       0       0       0       0       0       0       0       0
## pixel36 pixel37 pixel38 pixel39 pixel40 pixel41 pixel42 pixel43 pixel44
##       0       0       0       0       0       0       0       0       0
## pixel45 pixel46 pixel47 pixel48 pixel49
##       0       0       0       0       0
```

Sampling Function (Sampling)

```
#Sampling for n = 1%, 5%, 10%
Sampling <- function(n){
  for (i in 1:3){
    assign(paste0("samp", i), setNames(lapply(1:3, function(x){
    x <- training[sample(1:nrow(training), n[i] * nrow(training), replace = FALSE), ];
      x}), paste0("train_n", i, "_", 1:3)))
  }
  Samples <- list(samp1, samp2, samp3)
}

n <- c(0.01, 0.05, 0.10)
traindata <- Sampling(n)
```

Points Function (PTS)

```
#Create function to calculate the points
PTS <- function(x){
  Points <- 0.25 * x[1] + 0.25 * x[2] + 0.5 * x[3]
}
```

Results Function (RFN)

```
#Store the results into a list
RFN <- function(A, B, C, Pmatrix){
  ABC <- c(A, min(1, B[3] / 60), 1 - C, NA)
  names(ABC) <- c("A", "B", "C", "Points")
  ABC[4] <- PTS(ABC)
  ABC <- round(ABC, 4)
  PredMatrix <- as.numeric(Pmatrix)

  Results <- list("ABC" = ABC, "PredMatrix" = PredMatrix)
}
```

Iteration Function (ITF)

```
ITF <- function(model, n, t = NA, m1 = NA, m2 = NA, m3 = NA, m4 = NA, m5 = NA, m6 = NA, m7 = NA,
m8 = NA, m9 = NA){
  name <- substitute(model)
  if (name == "EM"){
    n <- rep(n, each = 3)
    k <- rep(1:3, 3)
    j <- rep(1:3, each = 3)
    for(i in 1:9){
      saveRDS(do.call(model, list(m1[[i]], m2[[i]], m3[[i]], m4[[i]], m5[[i]], m6[[i]], m7[[i]],
m8[[i]], m9[[i]], testing, n[i])), file = paste0(name, "_n", j[i], "_", k[i]))
    }
    return(NA)
  }
  for (i in 1:3){
    for (j in 1:3){
      saveRDS(do.call(model, list(t[[i]][[j]], testing, n[i])), file = paste0(name, "_n", i, "_"
, j))
    }
  }
}
```

Load Function (LF)

```
LF <- function(model){
  name <- substitute(model)
  temp <- list.files(pattern = paste0(name, "_"))
  files <- lapply(temp, readRDS)
}
```

Average Results Function (ARFN)

```
ARFN <- function(model, result){
  name <- substitute(model)
  for (i in 1:3){
    assign(paste0("MT", i), round(apply(sapply(result, function(x){x[[1]]})[, (3*i-2):(3*i)], 1,
mean), 4))
  }
  AverageResults <- list(MT1, MT2, MT3)
}
```

Sample Table

```
#Datasets table
SampleSize <- c(as.character(n * nrow(training)))
RandomSample <- matrix(paste0("train_n", rep(1:3, each = 3), "_", 1:3), nrow = 3)

TABLE1 <- data.table(SampleSize, RandomSample)
colnames(TABLE1) <- c("Sample Size", "First Random Sample", "Second Random Sample", "Third Rando
m Sample")
datatable(TABLE1)
```

Show 10 **entries** **Search:**

	Sample Size	First Random Sample	Second Random Sample	Third Random Sample
1	600	train_n1_1	train_n2_1	train_n3_1
2	3000	train_n1_2	train_n2_2	train_n3_2
3	6000	train_n1_3	train_n2_3	train_n3_3

Showing 1 to 3 of 3 entries Previous 1 Next

Model 1 Random Forest

Random Forest Overview

As the saying goes, 'Random Forest is always the second best choice'. So we made it the "first" model to build this time. It is used through randomForest package and randomForest function. It is based on the concpet of decision trees. It starts with several features, split the nodes using best split, repeats this process to create multiple trees and using voting to get the results. The variable $CV = 5$ stands for the 5-fold cross validation, but it has been disabled in our project and it will be discussed with parameter tuning in the Discussion Section. We use the randomForet package (randomForest function) to build the model. The parameters we choose are $mtry = 8$ (number of variables available for splitting at each tree node) and $ntree = 500$ (number of trees to grow) as they are the most common parameters used in the random forest model. It has many advantages. It is one of the most accurate learning algorithms and it runs efficiently and we do expect the random forest to have a lower running time and higher accuracy compared to other models we will use. It could also estimates of what variables are important in the classfiication. Meanwhile, it also has some disadvantages. It has been observed to overfit for some datasets with noisy classification taksks and it is difficult for us to interpret.

Random Forest Function

```
RF <- function(x, y, size, CV = 5, mtry = 8, ntree = 500, Tune = FALSE){
  startTime <- proc.time()
#Parameter Tuning
  if (Tune == TRUE){
    RFtrControl <-trainControl(method = "cv", number = CV)
    RFtuneGrid <- expand.grid(mtry = mtry)
    cvRF <- train(label~., data = x, method = "rf", ntree = ntree, trControl = RFtrControl, tune
Grid = RFtuneGrid)
    mtry <- cvRF$bestTune[1,1]
  }
#Build Model
  RFmodel <- randomForest(label~., data = x, ntree = ntree, mtry = mtry)
#Prediction and Record Time
  RFpred <- predict(RFmodel, y)
  RFtime <- proc.time() - startTime
#Accuracy
  RFacc <- sum(RFpred == y$label) / nrow(y)
#Store Results
  RFpred <- as.numeric(RFpred)
  Results <- RFN(size, RFtime, RFacc, RFpred)
}
```

Random Forest Model

```
ITF(RF, n, t = traindata)
```

Load and Average Random Forest Results

```
#Load Results
RFresult <- LF(RF)
#Average Results
ARF <- ARFN(RF, RFresult)
```

Model 2

Linear Discriminant Analysis Overview

We happened to learn this model in our Statistical Machine Learning class recently so this becomes our second model to build. It is used through MASS package and lda function. This model is fast and simple. It does not have any parameters to tune in our case. It is frequently used as a tool to reduce the dimension or pattern classification. It computes the directions or linear discriminants that will represent the axes that maximize the seperation between multiple classes. One of the disadvantage is that it assumes the data are normally distributed and mean is the discriminating factor. It may also overfit the data. Although it is an old algorithm, it could still beat some of the machine learning models like logistic regression when its assumptions are met.

Linear Discriminant Analysis Function

```
LDA <- function(x, y, size){
  startTime <- proc.time()
#Parameter Tuning
  #Nah, there is no parameter for LDA model
#Build Model
  LDAmodel <- lda(label~., data = x)
#Prediction and Record Time
  LDApred <- predict(LDAmodel, y)
  LDAtime <- proc.time() - startTime
#Accuracy
  LDAacc <- sum(as.character(unlist(LDApred[1])) == y$label) / nrow(y)
#Store Results
  LDApred <- as.numeric(unlist(LDApred[1]))
  Results <- RFN(size, LDAtime, LDAacc, LDApred)
}
```

Linear Discriminant Analysis Model

```
ITF(LDA, n, t = traindata)
```

Load and Average Linear Discriminant Analysis Model

```
#Load Results
LDAresult <- LF(LDA)
#Average Results
ALDA <- ARFN(LDA, LDAresult)
```

Model 3

Gradient Boosting Overview

The third algorithm we will use is the Gradient Boosting model. It is used through gbm package and gbm function. We would like to introduce XGboost as our third model, but we believe it makes more sense to use the Gradient boosting model first since XGboost is a variant of GBM algorithm. Gradient Boosting is a method of converting weak learners into strong learners. It builds a tree, optimizes the loss function using the gradient descent procedure, puts new weights and generates new tree one at a time and thus increase the accuracy of weights and minimizes the error. For the parameters, we chose $n.trees = 300$ (number of trees to grow). The default is usually 500 and we choose to reduce the time so we selected a smaller tree. $interaction.depth = 6$ (number of splits it performs on a tree) and $shrinkage = 0.01$ (defines the steps taken in the gradient descent of boosting) are the most common values to pick in a GBM model. $n.minobsinnode = 5$ (minimal observation in the terminal node to stop the tree) is also a common value for small training sample. The advantage of GBM model is it could have very high accuracy, it does not require data pre-processing and it has lots of flexibility. It also has some disadvantages. It may cause overfitting, it is computationally expensive when tuning parameter and training the model and it is not easily interpretable.

Gradient Boosting Function

```
GBM <- function(x, y, size, CV = 5, n.trees = 300, interaction.depth = 6, shrinkage = 0.01, n.mi
nobsinnode = 5, Tune = FALSE){
  startTime <- proc.time()
#Parameter Tuning
  if (Tune == TRUE){
    GBMtrControl <- trainControl(method = "cv", number = CV)
    GBMtuneGrid <- expand.grid(n.trees = n.trees,
                               interaction.depth = interaction.depth,
                               shrinkage = shrinkage,
                               n.minobsinnode = n.minobsinnode)
    cvGBM <- train(label~., data = x, method = "gbm", trControl = GBMtrControl, tuneGrid = GBMtu
neGrid)
    n.treees <- cvGBM$bestTune[1,1]
    interaction.depth <- cvGBM$bestTune[1,2]
    shrinkage <- cvGBM$bestTune[1,3]
    n.minobsinnode <- cvGBM$bestTune[1,4]
  }
#Build Model
  GBMmodel <- gbm(label~., data = x, distribution = "multinomial", n.trees = n.trees, interactio
n.depth = interaction.depth, shrinkage = shrinkage, n.minobsinnode = n.minobsinnode)
#Prediction and Record Time
  GBMpred <- predict.gbm(GBMmodel, y, n.trees, type = "response")
  GBMpred <- apply(GBMpred, 1, which.max)
  GBMtime <- proc.time() - startTime
#Accuracy
  GBMacc <-sum(GBMpred == as.numeric(y$label)) / nrow(y)
#Store Results
  Results <-RFN(size, GBMtime, GBMacc, GBMpred)
}
```

Gradient Boosting Model

```
ITF(GBM, n, t = traindata)
```

Load and Average Gradient Boosting Results

```
#Load Results
GBMresult <- LF(GBM)
#Average Results
AGBM <- ARFN(GBM, GBMresult)
```

Model 4

XGBoost Overview

Here we have our fourth model Extreme Gradient Boosting Machine aka XGboost. It is applied through the XGboost package and xgb.train function. It is a scalable tree boosting system widely used in different machine learning competitions such as Kaggle. It basically works the same as the GBM model we built in the previous section. Both of them follow the principle of gradient boosting. But XGboost uses a more regularized model formalization to control overfitting. XGboost has also improved data structures and supported parallelization and multicore processing to reduce the training time and we shall see it has a faster computational time than the GBM

model in the scoreboard section. Since our model for 5% of sample size converges at round 30, we choose our paramter $nrounds = 100$. $eta = 0.5$ (the learning rate), $gamma = 0$ (minimun loss reduction required to make a further partition), $max\_depth = 5$ (max depth of a tree), $min\_child\_weight = 2$, $subsample = 1$ and $colsample\_bytree = 1$ are the most common used values for these parameters. $showsd = TRUE$, $early\_stopping\_rounds = 10$, $print\_every\_n = 10$ are the parameters to check the convergence of the model. As we have mentioned, XGboost has a lot of advantages. It is very fast, efficient and it does not require normalization of data or missing value imputation. But, it could only work with numberic features and will lead to overfitting if parameters are not tuned properly. We have to convert it into design matrix while it could only count from 0 in the model building and prediction process, so we need to do the $+1$ and -1 to make sure we have the correction prediction.

XGBoost Function

```
XGB <- function(x, y, size, CV = 5, nrounds = 100, eta = 0.5, gamma = 0, max_depth = 5, min_chil
d_weight = 2, subsample = 1, colsample_bytree = 1, showsd = TRUE, early_stopping_rounds = 10, pr
int_every_n = 10, Tune = FALSE){
 startTime <- proc.time()
#Parameter Tuning
  if (Tune == TRUE){
  XGBtrControl <- trainControl(method = "cv", number = CV)
  XGBtuneGrid <- expand.grid(nrounds = nrounds,
                             eta = eta,
                             max_depth = max_depth,
                             gamma = gamma,
                             colsample_bytree = colsample_bytree,
                             min_child_weight = min_child_weight,
                             subsample = subsample)
  cvXGB <- train(x[, -1], x[, 1], method = "xgbTree", trControl = XGBtrControl, tuneGrid = XGBtu
neGrid)
  nrounds <- cvXGB$bestTune[1,1]
  max_depth <- cvXGB$bestTune[1,2]
  eta <- cvXGB$bestTune[1,3]
  gamma <- cvXGB$bestTune[1,4]
  colsample_bytree <- cvXGB$bestTune[1,5]
  min_child_weight <- cvXGB$bestTune[1,6]
  subsample <- cvXGB$bestTune[1,7]
  }
#Best Tune
  parameters <- list(
    objective = "multi:softmax",
    booster = "gbtree",
    eval_metric = "merror",
    num_class = length(unique(x$label)),
    eta = eta,
    gamma = gamma,
    max_depth = max_depth,
    min_child_weight = min_child_weight,
    subsample = subsample,
    colsample_bytree = colsample_bytree
  )
#Convert into Design Matrix
  XGBlabel <- as.numeric(x$label) - 1
  x <- sparse.model.matrix(label~., data = x)
  XGBTRAIN <- xgb.DMatrix(data = x, label = XGBlabel)
#Build Model
  XGBmodel <- xgb.train(params = parameters, data = XGBTRAIN, nrounds = nrounds, showsd = shows
d, early_stopping_rounds = early_stopping_rounds, print_every_n = print_every_n, watchlist = lis
t(x = XGBTRAIN), verbose = 0)
#Prediction and Record Time
  XGBtest <- y
  XGBtest$label <- 0
  XGBtest <- sparse.model.matrix(label~., data = XGBtest)
  XGBpred <- predict(XGBmodel, XGBtest)
  XGBtime <- proc.time() - startTime
#Accuracy
  XGBacc <- sum((XGBpred + 1) == as.numeric(y$label)) / nrow(y)
```

```
#Store Results
  XGBpred <- XGBpred + 1
  Results <- RFN(size, XGBtime, XGBacc, XGBpred)
}
```

XGBoost Model

```
ITF(XGB, n, t = traindata)
```

Load and Average XGBoost Results

```
#Load Results
XGBresult <- LF(XGB)
#Average Results
AXGB <- ARFN(XGB, XGBresult)
```

Model 5

K-Nearest Neighbors Overview

The fifth model is K-nearest Neighbors algorithm. It is used through class package and KNN function. In the classification setting, the K-Nearest Neighbors algorithm calculate the Euclidean distance between the data points and observations and then estimates the conditional probability for each class given the class label. When k is small, it will provide the most flexible fit with low bias but high variance. With a high value of k, the decision boundaries will be smoother and more resilient to outliers but the bias will increase. In this method, we decided to choose the parameter as $k = 8$ because it is at the mean of the most common picked values from $k = 5$ to $k = 10$. The main advantages are: the model is very simple, easy to interpret output and we don't need any assumptions for our data. But at the same time, it is also computationally expensive for large datasets since it calculates all the distance between observations and features and the prediction will be slow.

K-Nearest Neighbors Function

```
KNN <- function(x, y, size, CV = 5, k = 8, Tune = FALSE){
  startTime <- proc.time()
#Parameter Tuning
  if (Tune == TRUE){
    KNNtrControl <- trainControl(method = "cv", number = CV)
    KNNtuneGrid <- expand.grid(k = k)
    cvKNN <- train(label~., data = x, method = "knn", trControl = KNNtrControl, tuneGrid = KNNtu
neGrid)
    k <- cvKNN$bestTune[1,1]
  }
#Build Model, Predict and Record Time
  KNNmodel <- knn(x[, -1], y[, -1], x[, 1], k = k)
  KNNtime <- proc.time() - startTime
#Accuracy
  KNNacc <- sum(as.numeric(KNNmodel) == as.numeric(y$label)) / nrow(y)
#Store Results
  KNNpred <- as.numeric(KNNmodel)
  Results <- RFN(size, KNNtime, KNNacc, KNNpred)
}
```

K-Nearest Neighbors Model

```
ITF(KNN, n, t = traindata)
```

Load and Average K-Nearest Neighbors Results

```
#Load Results
KNNresult <- LF(KNN)
#Average Results
AKNN <- ARFN(KNN, KNNresult)
```

Model 6

Support Vector Machine Overview

We choose Support Vector Machine as our sixth model. It is used through e1071 package and svm function. The idea of Support Vector Machine is very simple: It creates a line or a hyperplane to separate the data into different classes. It only focus on the points that are most difficult to tell apart while tring to make a decision boundary as wide as possible. $cost = 10$ is the regularization term and weight for penalizing the soft marigin. Since a small cost will give us high bias while a large cost will have a high variance. We picked 10 for cost which is in the middle position among the most used values. Support Vector Machine is very accurate when working with small and clean datasets. It is guaranteed to find the global mininun. But, it is not suited to larger datasets because the tranining time would be large. It is not effective when training noiser datasets with overlapping classes.

Support Vector Machine Function

```
SVM <- function(x, y, size, cost = 10, Tune = FALSE){
  startTime <- proc.time()
#Parameter Tuning
  if (Tune == TRUE){
    SVMparameters_n2_1 <- tune.svm(label~., data = x, cost = cost)
    cost <- unlist(summary(SVMparameters)[1])
  }
#Build Model
  SVMmodel <- svm(label~., data = x, kernal = "radial", cost = cost)
#Prediction and Record Time
  SVMpred <- predict(SVMmodel, y)
  SVMtime <- proc.time() - startTime
#Accuracy
  SVMacc <- sum(SVMpred == y$label) / nrow(y)
#Store Results
  SVMpred <- as.numeric(SVMpred)
  Results <- RFN(size, SVMtime, SVMacc, SVMpred)
}
```

Support Vector Machine Model

```
ITF(SVM, n, t = traindata)
```

Load and Average Support Vector Machine Results

```
#Load Results
SVMresult <- LF(SVM)
#Average Results
ASVM <- ARFN(SVM, SVMresult)
```

Model 7

Multinomial Logistic Regression Overview

The seventh model we introduce is the Multinomial Logistic Regression Model. It is used through nnet package and multinom function. It is also called Softmax regression and it is a generalization of logistic regression for multiple classes. The technique will be the same like the logistic regression for binary outcome. It assumes the data are case specific and the dependent variable can not be perfectly predicted from independent variables for any case. $maxit = 500$ is the number of iterations it will perfrom until the model converges and 500 is the default value. $decay = 0.05$ is a parameter for weight decay which is a L2 regularization for avoiding overfitting. We choose 0.05 as it is a common used value for this parameter. This model is easy to interpret and performs well when features are expected to be roughly linear. A major shortcoming of the model is its property of the independence of the relative probability of chocie of two alternatives irrespective of the presence or characteristics of other alternatives and this model also suffers multicollinearity.

Multinomial Logistic Regression Function

```
ML <- function(x, y, size, CV = 5, decay = 0.05, maxit = 500, Tune = FALSE){
  startTime <- proc.time()
#Parameter Tuning
  if (Tune == TRUE){
    MLtrControl <- trainControl(method = "cv", number = CV)
    MLtuneGrid <- expand.grid(decay = decay)
    cvML <- train(label~., data = x, method = "multinom", trControl = MLtrControl, tuneGrid = ML
tuneGrid)
    decay <- cvML$bestTune[1,1]
  }
#Build Model
  MLmodel <- multinom(label~., data = x, decay = decay, maxit = maxit, trace = FALSE)
#Prediction and Record Time
  MLpred <- predict(MLmodel, y)
  MLtime <- proc.time() - startTime
#Accuracy
  MLacc <- sum(MLpred == y$label) / nrow(y)
#Store Results
  MLpred <- as.numeric(MLpred)
  Results <- RFN(size, MLtime, MLacc, MLpred)
}
```

Multinomial Logistic Regression Model

```
ITF(ML, n, t = traindata)
```

Load and Average Multinomial Logistic Regression Results

```
#Load Results
MLresult <- LF(ML)
#Average Results
AML <- ARFN(ML, MLresult)
```

Model 8

Neural Network Overview

Our eighth model is the Neural Network model. It is used through nnet package and nnet function. The idea of Neural Network is inspired by and partially modeled on biological neural networks. They are capable of modeling and processing nonlinear relationships between inputs and outputs in parallel. It is made up of three components: input layer, hidden layer and ouput layer. The learning happens in two steps: Forward-Propagation (make guess about the answer) and Back-Propagation (minimize the error between the actual answer and guessed answer). We will not go into too much details here as it will take the whole report explaining the concepts and theories. $n = 20$ is the number of units in hidden layer and $decay = 0.25$ is the regularization parameter to avoid overfitting. We choose to use a small n and $decay$ to lower the computational time. $maxit = 1000$ is the number iterations it will perform. We choose 1000 to make sure the model converges. $MaxNWts = 1500$ will be the max number of weights. Since the mininum required for this model and $n = 20$ is around 1300, we choose to use a value close to 1300. Neural networks have a lot of advantages. It has the ability to learn and model non-linear and complex relationships and it does not have any restrictions on the input variables. However, we need to normalize the data and the process of neural networks are black boxes. We don't know what happened during the training process and it takes time to tune the parameters and to train the model.

Neural Network Function

```
NNE <- function(x, y, size, CV = 5, n = 20, decay = 0.25, maxit = 1000, MaxNWts = 1500, Tune = F
ALSE){
  startTime <- proc.time()
#Noramalize pixel values
  NNlabel <- class.ind(as.numeric(x$label))
  NNtest <- y[, -1] / 255
  x <- x[, - 1] / 255
#Parameter Tuning
  if (Tune == TRUE){
  NNtrControl <- trainControl(method = "cv", number = 5)
  NNtuneGrid <- expand.grid(size = n, decay = decay)
  cvNN <- train(label~., data = x, method = "nnet", trControl = NNtrControl, tuneGrid = NNtuneGr
id, MaxNWts = MaxNWts)
  n <- cvNN$bestTune[1,1]
  decay <- cvNN$bestTune[1,2]
  }
#Build Model
  NNmodel <- nnet(x, NNlabel, size = n, softmax = TRUE, maxit = maxit, decay = decay, MaxNWts =
 MaxNWts, trace = FALSE)
#Prediction and Record Time
  NNpred <- predict(NNmodel, NNtest, type = "class")
  NNtime <- proc.time() - startTime
#Accuracy
  NNacc <- sum(as.numeric(NNpred) == as.numeric(y$label)) / nrow(y)
#Store Results
  NNpred <- as.numeric(NNpred)
  Results <- RFN(size, NNtime, NNacc, NNpred)
}
```

Neural Network Model

```
ITF(NNE, n, t = traindata)
```

Load and Average Neural Network Results

```
#Load Results
NNEresult <- LF(NNE)
#Average Results
ANNE <- ARFN(NNE, NNEresult)
```

Model 9

Ridge Regression Overview

The ridge regresion model will be our ninth model to build. It is used through glmnet package and glmnet function. It is a technique for analyzing multiple regression data that suffer from multicollinearity and optimized for prediction. It allows you to regularize coefficients by imposing a penalty equivalent to the square of the magnitude of coefficients. We will use $alpha = 0$ as our parameter for this model using the instruction from the description of midterm project. Ridge regression can reduce the variance and improve predictive performance. It also has simple calculations. But it is not able to shrink coefficients to exactly zero like Lasso Regression which means that it could not perform a variable selection.

Ridge Regression Function

```
RR <- function(x, y, size){
  startTime <- proc.time()
#Build Model
  RRmodel <- glmnet(as.matrix(x[, -1]), as.factor(x[, 1]), family = "multinomial", alpha = 0)
#Prediction and Record Time
  RRpred <- predict(RRmodel, newx = as.matrix(y[, -1]), type = "class")
  RRtime <- proc.time() - startTime
#Accuracy
  RRacc <- sum(as.numeric(as.factor(RRpred[,100])) == as.numeric(y$label)) / nrow(y)
#Store Results
  RRpred <- as.numeric(as.factor(RRpred[, 100]))
  Results <- RFN(size, RRtime, RRacc, RRpred)
}
```

Ridge Regression Model

```
ITF(RR, n, t = traindata)
```

Load and Average Ridge Regression Results

```
#Load Results
RRresult <- LF(RR)
#Average Results
ARR <- ARFN(RR, RRresult)
```

Model 10

Ensembled Overview

Finally, we have reached our last model. The Ensembled Model is based on the predictions from previous results we have obtained. We choose the level 1 ensembling method because this will take the least amount of time to build the model. We first choose to use all nine different models to build our Ensembled Model. But we found out that there might be some noise that will affect the prediction performance. Then, we revised our algorithm a little bit. In our function, we will choose the top three models and combine their results to obtain the Ensembled Model. This way, we could reduce the computational time and increase the accuracy of our model. The benefits of Ensembled Model is obvious. It is simple and fast to train using voting method and it is not likely to overfit and they will average out biases and variances. The weakness of this model is that it depends on the previous models we have built. If the previous models performed poorly, it would also have a poor performance most of the times.

Ensembled Function

```
EM <- function(m1, m2, m3, m4, m5, m6, m7, m8, m9, y, size){
  startTime <- proc.time()
#Find three models with highest accuracy
  Order <- order(cbind(m1[[1]][3], m2[[1]][3], m3[[1]][3], m4[[1]][3], m5[[1]][3], m6[[1]][3], m
7[[1]][3], m8[[1]][3], m9[[1]][3]))[1:3]
#Build Model
  EMmodel <- cbind(m1[[2]], m2[[2]], m3[[2]], m4[[2]], m5[[2]], m6[[2]], m7[[2]], m8[[2]], m9[[2
]])[, Order]
#Prediction and Record Time
  EMpred <- apply(EMmodel, 1, Mode)
  EMtime <- proc.time() - startTime
#Accuracy
  EMacc <- sum(EMpred == as.numeric(y$label)) / nrow(y)
#Store Results
  EMpred <- as.numeric(EMpred)
  Results <- RFN(size, EMtime, EMacc, EMpred)
}
```

Ensembled Model

```
ITF(EM, n, m1 = RFresult, m2 = LDAresult, m3 = GBMresult, m4 = XGBresult, m5 = KNNresult, m6 = S
VMresult, m7 = MLresult, m8 = NNEresult, m9 = RRresult)
```

Load and Average Ensembled Model Results

```
#Load Results
EMresult <- LF(EM)
#Average Results
AEM <- ARFN(EM, EMresult)
```

Reporting Function and Scoreboard

```
Reporting <- function(model, n, results, m, data = T){
  Model <- cbind(rep(model, each = 3 * m))
  Sample_Size <- cbind(rep(rep(n * nrow(training), each = m), 10))
  if (data == F){
    Values <- rbind(t(matrix(unlist(results), nrow = 4)))
    Table <- data.table(Model, Sample_Size, Values)
    colnames(Table)[1:6] <- c("Model", "Sample Size", "A", "B", "C", "Points")
    setorderv(Table, cols = "Points", order = 1)
    datatable(Table)
  }
  else{
   Data <- cbind(rep(paste0("train_n", rep(1:3, each = 3), "_", 1:3), 10))
   Values <- rbind(t(sapply(results, function(x){x[[1]]})))
   Table <- data.table(Model, Sample_Size, Data, Values)
   colnames(Table)[1:3] <- c("Model", "Sample Size", "Data")
   datatable(Table)
  }
}

models <- c("Random Forest", "Linear Discriminant Analysis", "Gradient Boosting Machine", "XGBoo
st", "K-Nearest Neighbors", "Support Vector Machine", "Multinomial Logistic Regression", "Neural
Network", "Ridge Regression", "Ensembled Model")
results <- c(RFresult, LDAresult, GBMresult, XGBresult, KNNresult, SVMresult, MLresult, NNEresul
t, RRresult, EMresult)

Reporting(models, n, m = 3, results)
```

Show 10 ▼ **entries** **Search:**

	Model	Sample Size	Data	A	B	C	Points
1	Random Forest	600	train_n1_1	0.01	0.0213	0.2259	0.1208
2	Random Forest	600	train_n1_2	0.01	0.0212	0.2353	0.1254
3	Random Forest	600	train_n1_3	0.01	0.0197	0.2372	0.126
4	Random Forest	3000	train_n2_1	0.05	0.0954	0.1819	0.1273
5	Random Forest	3000	train_n2_2	0.05	0.0955	0.1828	0.1278
6	Random Forest	3000	train_n2_3	0.05	0.0921	0.1781	0.1246
7	Random Forest	6000	train_n3_1	0.1	0.2107	0.1652	0.1603
8	Random Forest	6000	train_n3_2	0.1	0.2042	0.1673	0.1597
9	Random Forest	6000	train_n3_3	0.1	0.2087	0.1656	0.16
10	Linear Discriminant Analysis	600	train_n1_1	0.01	0.0009	0.2426	0.124

Showing 1 to 10 of 90 entries

Previous 1 2 3 4 5 … 9 Next

Final Scoreboard

```
results_F <- c(ARF, ALDA, AGBM, AXGB, AKNN, ASVM, AML, ANNE, ARR, AEM)

Reporting(models, n, results_F, m = 1, data = F)
```

Show 10 entries Search:

	Model	Sample Size	A	B	C	Points
1	Ensembled Model	3000	0.05	0.0194	0.1602	0.0974
2	Ensembled Model	6000	0.1	0.0185	0.1486	0.1039
3	Support Vector Machine	3000	0.05	0.0388	0.1656	0.105
4	Ensembled Model	600	0.01	0.0188	0.2129	0.1136
5	Support Vector Machine	600	0.01	0.0077	0.2191	0.114
6	Random Forest	600	0.01	0.0207	0.2328	0.1241
7	Support Vector Machine	6000	0.1	0.0975	0.1515	0.1251
8	Linear Discriminant Analysis	600	0.01	0.0009	0.2451	0.1252
9	Neural Network	600	0.01	0.0491	0.2216	0.1256
10	K-Nearest Neighbors	3000	0.05	0.0231	0.2156	0.1261

Showing 1 to 10 of 30 entries

Previous 1 2 3 Next

Discussion

In this project, our sample sizes are 1%, 5% and 10%. We choose the range from 1 to 10% because from our test runs, we found out any sample sizes greater than 10% will increase the time while not providing a much higher accuracy and any samle sizes smaller than 1 will have very poor performance for the label prediction. 5% will just be the mean of those two values that will balance the running time and prediction accuracy.

Since our goal is to minimize the value of points instead of obtaining the higest accuracy and the time takes up to 25% of the points, we decide to turn off the parameter tuning function for our models. That is because tuning the parameters will drastically increase the computational time for some models and they will lose all the points for the time they used such as Gradient Boosting and Neural Network models. But this is also not fair for some models that do require tuning to achienve a better result such as XGboost model. So, in the end, we decide to choose some most common parameter values for our models. This is fair to the models that require tuning and reduce the

computational time at the same time. If the weights of time spent is much lower than 25% or the goal is to build a model that is highly accurate, then we would have turn on the parameter tuning and spend time obtanining the best parameters.

From the Final Scoreboard, the rank of each model using their lowest points is: Ensembled model, Support Vector Machine, Random Forest, Linear Discriminant Analysis, Neural Network, K-Nearest Neighbors, XGboost, Ridge Regression, Multinomial Logistic Regression and Gradient Boosting Machine. We could see that all the models achieved their lowest points for a sample size at $n = 600$ and $n = 3000$. It satisfies with our hypothesis that 5% of the sample size will balance the time spent and the accuracy of the prediction. Any sample size that is lower than 1% will have an obvious impact to the prediction performance and Sample size higher than 10% will increase the points for time component while the prediction accuracy will not increase a lot.

We could see that the Ensembled Model has the lowest points. This is expected since the ensembled model does not take too long to build the model and it gathers the prediction results from the best three models. It should have the lowest points. Support Vector Machine, Neural Network, Random Forest and XGboost also did a good job in terms of accuracy for larger datasets. All of th models achieved their highest accuracy at the sample size of 3000 or 6000. For the smallest dataset, all of the models did poorly and their accuracy went down to less than 80%. Multinomial Logistic was absolutely terrible with the worst acuuracy of 66% in the final scoreboard. But it increased accuracy greatly when the sample sie went up to 3000. In terms of time consuming, Linear Discriminet Analysis is the fastest but it is also the model that basically has the same accuracy for all three different sample sizes. K-Nearest Neighbors and Support Vector Machine are als very fast for small datasets. Surprisingly, Ensembled Model with largest sample sizes is the fastest compared to other sample sizes. However, they are very close so this might be some variance of the computer processing units. Gradient Boosting and Neural Network are the slowest as expected. Even without tuning parameters, Gradient Boosting with sample size equal to 6000 spent almost one minute completing the model.

The weight of point components played a significant role when determining the tuning for parameter and the sample size. If there is less weight to the size component A and time component B (probably 10% or less), we would turn on the parameter tuning function and choose a larger sample size since obtaining the best parameters will have an obvious impact to some models such as XGboost and Neural Network. If there is more weight to the size component A and time component B, we would probably stay the same or even lower our sample size depends on the opportunity cost of the change in error rate. If we had the computing resources and time, we would definitely increase the sample size and find the best parameter for each model. We will be able to try some other machine learning models and deep learning techiniques such as Convolutional Neural Network and Keras package.

References

The knowledge of caret package from the following website:

https://topepo.github.io/caret/available-models.html (https://topepo.github.io/caret/available-models.html)

The knowledge of Random Forest from the following websites:

http://rstudio-pubs-static.s3.amazonaws.com/4239_fcb292ade17648b097a9806fbe026e74.html (http://rstudio-pubs-static.s3.amazonaws.com/4239_fcb292ade17648b097a9806fbe026e74.html)

http://dataaspirant.com/2017/05/22/random-forest-algorithm-machine-learing/ (http://dataaspirant.com/2017/05/22/random-forest-algorithm-machine-learing/)

The knowledge of Linear Discriminant Analysis from the course GR5241 Statistical Machine Learning and the following website:

https://sebastianraschka.com/Articles/2014_python_lda.html (https://sebastianraschka.com/Articles/2014_python_lda.html)

The knowledge of Gradient Boosting from the following websites:

https://towardsdatascience.com/understanding-gradient-boosting-machines-9be756fe76ab (https://towardsdatascience.com/understanding-gradient-boosting-machines-9be756fe76ab)

https://explained.ai/gradient-boosting/ (https://explained.ai/gradient-boosting/)

https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/ (https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/)

The knowledge of XGboost from the following websites:

https://xgboost.readthedocs.io/en/latest/index.html (https://xgboost.readthedocs.io/en/latest/index.html)

https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost-HowItWorks.html (https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost-HowItWorks.html)

The knowledge of KNN from the following website:

https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/#how-does-knn-work (https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/#how-does-knn-work)

The knowledge of SVM from the following website:

https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989 (https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989)

The knowledge of Multinomial Logistic from the following website:

http://deeplearning.stanford.edu/tutorial/supervised/SoftmaxRegression/ (http://deeplearning.stanford.edu/tutorial/supervised/SoftmaxRegression/)

The knowledge of Neural Network from the following websites:

http://neuralnetworksanddeeplearning.com/ (http://neuralnetworksanddeeplearning.com/)

https://skymind.ai/wiki/neural-network (https://skymind.ai/wiki/neural-network)

http://www.cs.toronto.edu/~hinton/coursera/lecture1/lec1.pdf (http://www.cs.toronto.edu/~hinton/coursera/lecture1/lec1.pdf)

The knowledge of Ridge Regression from the following website:

https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Ridge_Regression.pdf (https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Ridge_Regression.pdf)

The knowledge of Ensembled Regression from the following website:

https://www.sas.com/content/dam/SAS/en_ca/User%20Group%20Presentations/Toronto-Data-Mining-Forum/ngo_ensemble_modeling.pdf (https://www.sas.com/content/dam/SAS/en_ca/User%20Group%20Presentations/Toronto-Data-Mining-Forum/ngo_ensemble_modeling.pdf)