HW8

Cathy Su

2/11/2019

Q2 Classifying with trees

(a)

Split the data into training and test sets, as I did in gam-spam.Rmd. Using the ranger package, fit a random forest to the training data to classify the email. Show the confusion matrix for its predictions on the test set.

The normalized confusion matrix shows that the majority of predictions are correct (along teh diagonal) but there is some ~5% of cases are misclassified.

```r
# data
spam <- read.csv("../data/gam-spam.csv", header=FALSE)
spam$V58 <- as.factor(spam$V58)
training_rows <- sample.int(nrow(spam), round(nrow(spam) / 3))

spam.train <- spam[training_rows, ]
spam.test <- spam[-training_rows, ]

# apply log transform
cols <- names(spam.train)[1:(length(spam.train) - 1)]
for (col in cols) {
  spam.train[col] <- log(0.1 + spam.train[[col]])
  spam.test[col] <- log(0.1 + spam.test[[col]])
}
# Using the ranger pack- age,
# fit a random forest to the training data to classify the email.
fit <-ranger(dependent.variable.name = "V58", data=spam.train, write.forest = TRUE)
fit <- ranger(V58 ~ ., data = spam.train)
pred <- predict(fit, data = spam.test)
#fit$confusion.matrix
table(spam.test$V58, pred$predictions)/length(pred$predictions)
```

```
##
##              0          1
##   0 0.58069775 0.02054125
##   1 0.03456146 0.36419954
```

(b)

ranger automatically calculates the out-of-bag error for a forest and returns it as the prediction.error field of a forest object. In classification, this is the fraction of misclassified samples. How well does the out-of-bag error match the test set error?

Since 0.018+0.044=0.062 from the normalized confusion matrix, and the out of bag error is just under 0.06, the out-of-bag error matches the test set error well.

```r
fit$prediction.error
```

```
## [1] 0.05084746
```

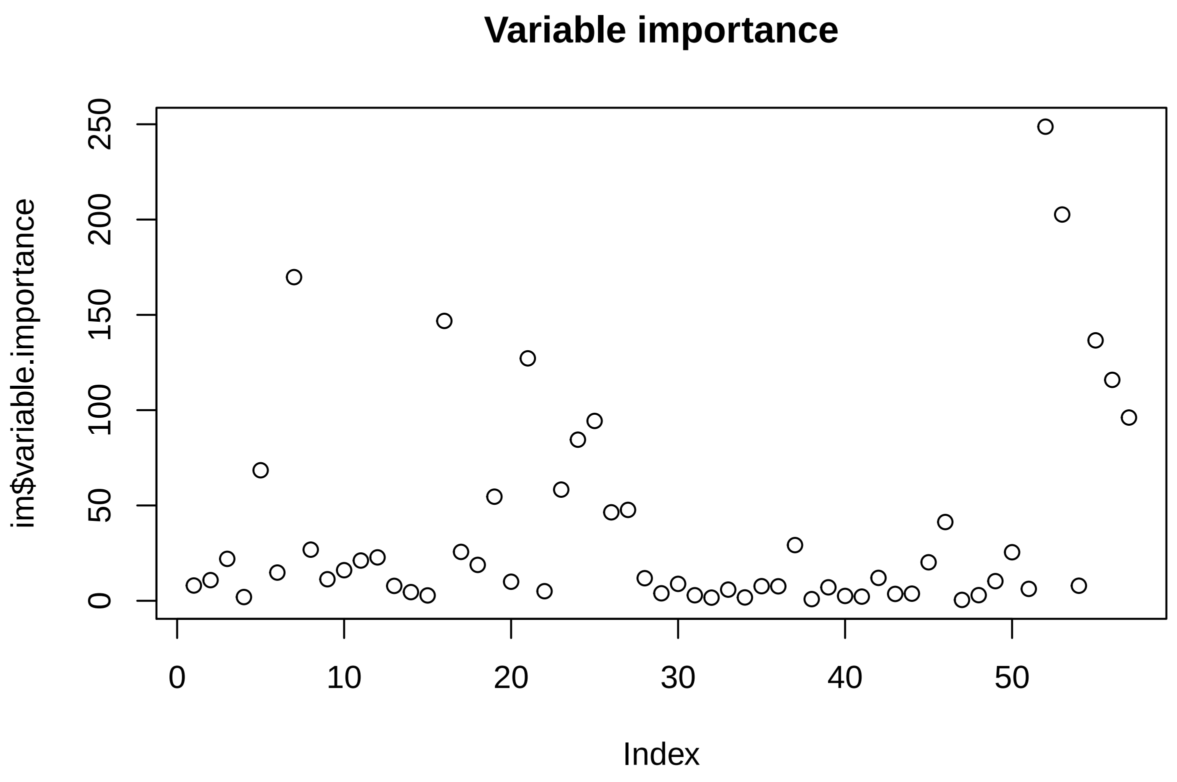
(c)

Extract the variable importance and plot them so you can see all the importances in one place. What variables appear most important? Does this make sense?

Here we used the impurity measure to assess importance, which calculates importance "as the sum over the number of splits (accross all tress) that include the feature, proportionaly to the number of samples it splits." From the gam-spam activity we know that: *the first 48 columns are the frequencies (percentages) of words in each email that are specific words, such as "business" or "credit" or "free". The next 6 columns cound the frequencies of specific characters. The next few columns look at capitalization in the email, and the last column indicates if the email was spam (1) or not (0).*

It seems from this graph that there are a few specific words among the 48 which are very important, the last few frequencies of specific characters are important (especially $ and !) and lastly the strings of capitalization within the email is really important. These observations seem to make sense since certain words, punctuation marks and use of capital letters are all distinctive characteristics of spam emails.

```r
im <- ranger(V58 ~ ., data = spam, importance = "impurity")
#im$variable.importance
plot(im$variable.importance, main = "Variable importance")
```



(d)

The ranger function has several tuning parameters. In particular, num.trees controls how many trees to include in the forest, while mtry is the number of variables to consider splitting at each node in the tree. (Recall that rather than considering splitting at every node, random forests pick the best split from a random subset at each node.) Build the random forests on the training sets with several different values of each parameter. (You might make a grid of possible values.) Time how long each forest takes to build (the system.time function may be useful). Compare the accuracy on the test set for all the combinations. What does the effect of each parameter seem to be? What are the tradeoffs between performance and accuracy? (I recommend starting with particularly extreme examples, such as mtry=1 or num.trees=5,

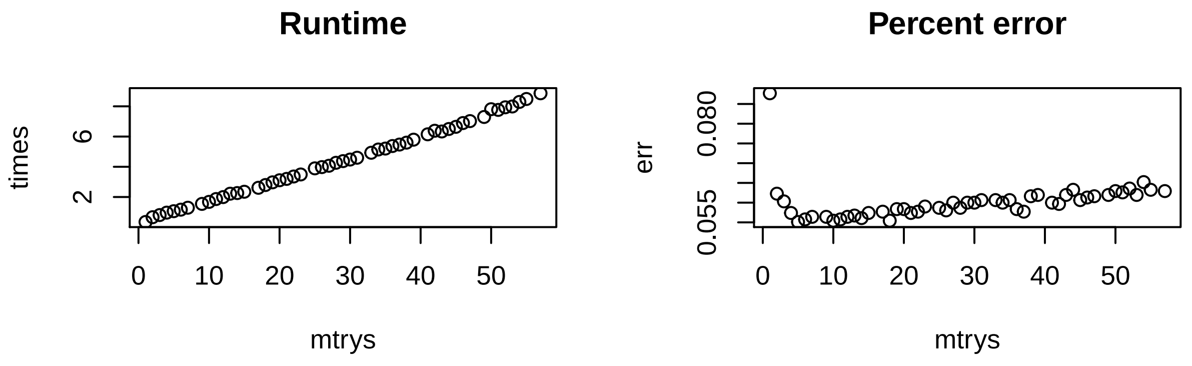
and don't raise the parameters much past their defaults, as making either too large will make the forests impractically slow to build.)

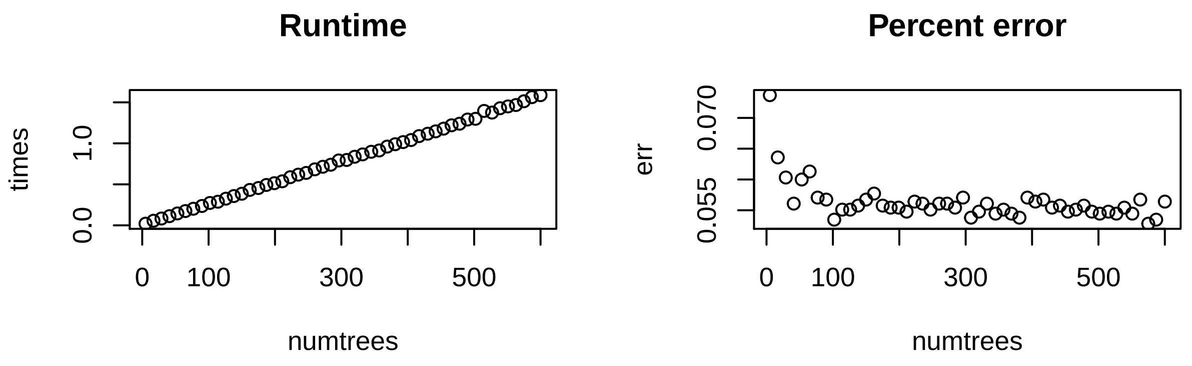
When increasing mtry, the time increases linearly and the accuracy increases, then starts to decrease past about mtry=10. This means that we obtain higher performance for each unit increase in the number of variables we can split at each node, but only up to about 6 variables.

When increasing num.trees, the time increases linearly and the accuracy decreases and plateaus past about num.trees=100. This means that we obtain higher performance for each unit increase in the number of trees, but only up to about 100 trees.

```r
# select ranges of interest
mtrys =floor(seq.int(1, 57, length.out=50))
numtrees=floor(seq.int(5, 600, length.out=50))

######## plot mtry vs Time how long each forest takes to build
# also  Compare the accuracy on the test set for all the combinations.
times <- numeric(length(mtrys))
err<-  numeric(length(mtrys))
for (m in 1:length(mtrys)){
  # Start the clock!
  ptm <- proc.time()
  fit <- ranger(V58 ~ .,
               mtry = mtrys[m],
               data = spam.train)
  # Stop the clock
  t <- proc.time() - ptm
  times[m] <- t[1]

  # find the accuracy
  pred <- predict(fit, data = spam.test, se.fit=TRUE)
  err[m] <- 1-length(spam.test$V58[spam.test$V58==pred$predictions])/length(spam.test$V58)
}

par(mfrow=c(2,2))
plot(mtrys, times, main = "Runtime")
plot(mtrys, err, main = "Percent error")

####### plot num.trees vs Time how long each forest takes to build
times <- numeric(length(mtrys))
err<-  numeric(length(mtrys))
for (m in 1:length(numtrees)){
  # Start the clock!
  ptm <- proc.time()
  fit <- ranger(V58 ~ .,
               num.trees=numtrees[m],
               data = spam.train)
  # Stop the clock
  t <- proc.time() - ptm
  times[m] <- t[1]

  # find the accuracy
  pred <- predict(fit, data = spam.test, se.fit=TRUE)
  err[m] <- 1-length(spam.test$V58[spam.test$V58==pred$predictions])/length(spam.test$V58)
}
```

```r
plot(numtrees, times, main = "Runtime")
plot(numtrees, err, main = "Percent error")
```





Q3

(a)

ranger has a max.depth parameter. Setting it to 0 means there is no depth restriction, and it builds trees with one observation per node; setting it to a natural number builds trees to that depth (so max.depth=1 corresponds to stumps). Build a forest with stumps and show that it does not perform well on the test set, as we'd expect.

We see from the normalized confusion matrix that the error rate is high, at about 50%.

```r
make_data <- function(N=150) {
  X <- matrix(runif(2 * N, min=0, max=10), nrow=N, ncol=2)
  Y <- ifelse((X[, 1] < 5 & X[, 2] < 5) | (X[, 1] > 5 & X[, 2] > 5), rbinom(N, 1, 0.1), rbinom(N, 1, 0.

  stopifnot(nrow(X) == length(Y)) # sanity check!
  new <- cbind(as.data.frame(X), as.data.frame(Y))
  new$Y <- as.factor(new$Y)
  #colnames(new)[-1] <- "Y"
  return(new)
}

spam <- make_data(N=800)
training_rows <- sample.int(nrow(spam), round(nrow(spam) / 3))

train <- spam[training_rows, ]
```

```r
test <- spam[-training_rows, ]

fit <- ranger(Y ~ ., data = train, max.depth = 1)
pred <- predict(fit, data = test)

table(test$Y, pred$predictions)/length(pred$predictions)
```

```
##
##              0          1
##   0 0.04315197 0.45028143
##   1 0.12195122 0.38461538
```

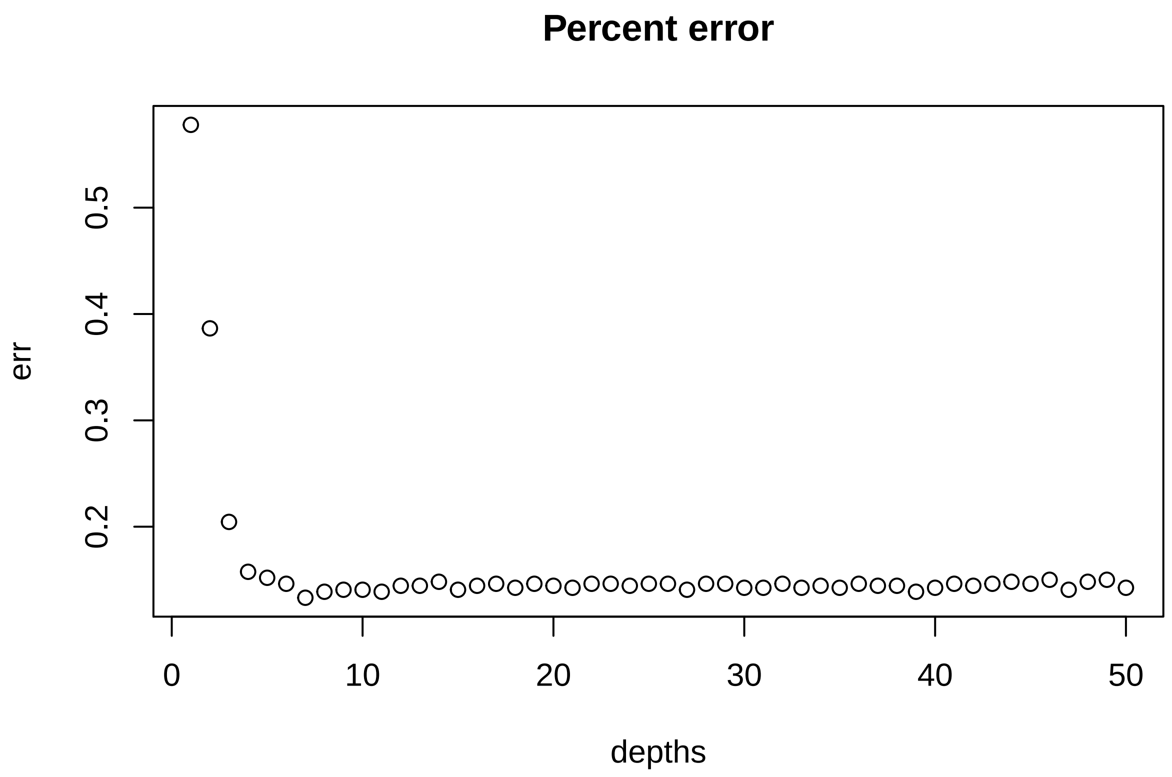
(b)

Gradually increase the max.depth, one step at a time, and evaluate the test set perfor- mance. Plot test error against depth. What depth seems to be the minimum depth needed for the forest to work? Does that match what you would have expected from our discussion of the activity? Is there harm from having a larger maximum depth than is strictly necessary?

We see that the error decreases to around 10% when depth > 4. This depth is a bit larger than we might have expected based on the in class activity (2), but it could be because this allows splitting along each of the two main dimensions multiple times.

Increased depth does not seem to lead to higher error rate in the test set. There might still be harm from allowing trees with such large depth that they could dramatically increase the runtime.

```r
# Plot test error against depth
depths =c(1:50)

######## Compare the accuracy on the test set for all the combinations.
err <-numeric(length(depths))
for (m in 1:length(depths)){
  fit <- ranger(Y ~ .,
                max.depth=depths[m],
                data = train)

  # find the accuracy
  pred <- predict(fit, data = test, se.fit=TRUE)
  err[m] <- 1-length(test$Y[test$Y==pred$predictions])/length(test$Y)
}

#par(mfrow=c(2,2))
plot(depths, err, main = "Percent error")
```



(c)

This question is open-ended. Random forests seem to have several tuning parameters, such as the number of trees, the depth, the number of parameters sampled, and so on. This example shows that some of those parameters do matter. In your own words, describe your recommendations for using random forests on real world data where you don't know the ideal tuning parameters. Where would you recommend you start? Describe the tradeoffs you make when increasing or decreasing each parameter, and describe how you could know if you have made good choices.

In our dataset we have 800 samples, with a 75:25 train:test split and we found that:

- increasing the number of trees does not improve test performance beyond 100 trees. But having too many trees does not hurt accuracy.
- increasing mtry (number of variables to consider) reaches minimum test error around 5 variables. Too many variables hurts accuracy.
- increasing depth improves test performance up to about depth 4. But having large depth does not hurt accuracy.

It seems based on this that for real world data of a similar format, that we may want to start with at least 100 trees and at least depth of 4. These parameters can be increased at teh cost of runtime and decreased at the cost of accuracy on the test set. Lastly we may want to start with mtry around 4 since increasing mtry past its optimum will both decrease performance and increase runtime. We can find an optimum by doing an analysis like the above where we measure performance on the test set for some simulated data.