Prediction Assignment Writeup

Weilun Chiu

1. Getting data and data pre-processing

First, we need to set the working directory and read the training data into our R.

```
setwd("C:/Users/Waylan/Documents/ML_FP")
original.train.data<-read.csv("pml-training.csv")
```

After reading the training data, we find there are several predictors with NA value, and some character varaibel we'd prefer it to be factor variable, so we use below code to pre-process it.

```
isna_logic<-sapply(original.train.data, anyNA)
na.features<-names(original.train.data)[isna_logic]
features<-setdiff(names(original.train.data), na.features)
original.train.data_2<-original.train.data[, features]

original.train.data_2$user_name<-as.factor(original.train.data_2$user_name)
original.train.data_2$new_window<-as.factor(original.train.data_2$new_window)
original.train.data_2$classe<-as.factor(original.train.data_2$classe)

cha_logic<-sapply(original.train.data_2, is.character)
cha_features<-names(original.train.data_2)[cha_logic]
features2<-setdiff(names(original.train.data_2), cha_features)
train.data<-original.train.data_2[, features2]
train.data<-train.data[, c(-1)]
```

And then we do the same thing with our testing data.

```
original.test.data<-read.csv("pml-testing.csv")
features3<-features2[1:(length(features2)-1)]
original.test.data_2<-original.test.data[, features3]
original.test.data_2$user_name<-as.factor(original.test.data_2$user_name)
original.test.data_2$new_window<-as.factor(original.test.data_2$new_window)
test.data<-original.test.data_2
```

2. Split our train data to training and validation purpose.

Before we start applying prediction models with our data, we'd like to have a seperate/independent validation data set to verify how accurate our models are, so we split our data with package caret

```
library(caret)
set.seed(123)
inTrain<-createDataPartition(train.data$classe, p=0.8, list=FALSE)
training<-train.data[inTrain, ]
testing<-train.data[-inTrain, ]
```

3. Construct prediction model

1. Naive Bayes

First, we'll try to use Naive Bayes to see is it can accurately predicting the classes. And then use the model to make prediction with our testing data set and see how accurate the model is.

```
library("klaR")
```

```
set.seed(123)
system.time(
model_nb_1<-NaiveBayes(classe~., data=training)
)
```

```
##    user  system elapsed
##    0.27    0.00    0.26
```

```
pred_nb_1<-predict(model_nb_1, testing)
round(confusionMatrix(testing$classe, pred_nb_1$class)$overall, 3)
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##          0.488          0.370          0.472          0.504          0.363
## AccuracyPValue  McnemarPValue
##          0.000          0.000
```

From the result above, the model constructed by Naive Bayes only gets us accuracy rate around 49%. The model is worse than just randomly guessing the result.

2. Decision Tree

Second, we'll use the training data to construct a single decision tree and use it to do the prediction. And we also load package "rattle" in order to plot the result tree.

```
library(rpart)
```

```
set.seed(123)
system.time(
model_dt_2<-rpart(classe~., data=training)
)
```

```
##    user  system elapsed
##    6.62    0.01    6.78
```

```
pred_dt_2<-predict(model_dt_2, testing, type="class")
round(confusionMatrix(testing$classe, pred_dt_2)$overall, 3)
```

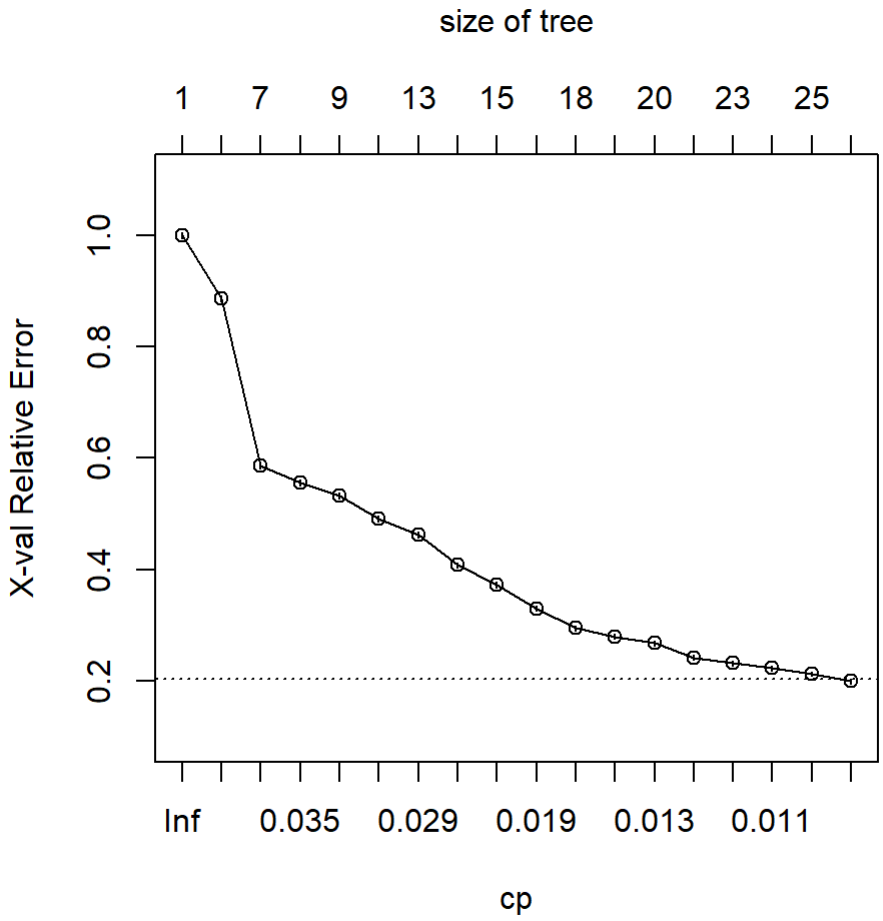
```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##          0.850          0.811          0.839          0.861          0.255
## AccuracyPValue  McnemarPValue
##          0.000          0.000
```

From above result, the single decision tree gives us around 85% accuracy. Behind the scenes, function rpart is applying a range of cost complexity values to prune the decision tree. To compare the error for each value, rpart performs a 10-fold cross validation so that the error associated with a given value is computed on the hold-out validation data.

```
model_dt_2$cptable
```

```
##            CP nsplit rel error    xerror        xstd
## 1  0.11473075      0 1.0000000 1.0000000 0.005030829
## 2  0.04263462      1 0.8852692 0.8858923 0.005372182
## 3  0.04094348      6 0.5963507 0.5862038 0.005503418
## 4  0.03070761      7 0.5554072 0.5562083 0.005458987
## 5  0.03017356      8 0.5246996 0.5328883 0.005416888
## 6  0.02910547      9 0.4945260 0.4913218 0.005324919
## 7  0.02839341     12 0.4072096 0.4631064 0.005249643
## 8  0.02661326     13 0.3788162 0.4078327 0.005070050
## 9  0.02429907     14 0.3522029 0.3727637 0.004932315
## 10 0.01428571     15 0.3279039 0.3298620 0.004735955
## 11 0.01424121     17 0.2993324 0.2951491 0.004552093
## 12 0.01344014     18 0.2850912 0.2796618 0.004462076
## 13 0.01281709     19 0.2716511 0.2679128 0.004390236
## 14 0.01112595     20 0.2588340 0.2418336 0.004218971
## 15 0.01103694     22 0.2365821 0.2319537 0.004149519
## 16 0.01085892     23 0.2255452 0.2227859 0.004082653
## 17 0.01068091     24 0.2146862 0.2124611 0.004004412
## 18 0.01000000     26 0.1933244 0.1994660 0.003901235
```

```
plotcp(model_dt_2)
```



From above plot, we can see when cp gets close to 0.01, the decision tree return a result with lowest cross validation error. But the problem is when we have a very small cp value, we might over fit our training data which will have low bias and higher variance when we use it to predict future data. Therefore, in the next step, we'll try to use bagging to reduct the varaince.

3. Decision Tree with Bagging

Let's use bagging method with decision tree. And use it to do the prediction with our testing data and see how accurate our model is.

We use train function from package "caret", and use 10-fold cross validation with function trainControl.

```
ctrl<-trainControl("cv", number=10)
set.seed(123)
system.time(
model_bagging_3<-train(
  classe~.,
  data=training,
  method="treebag",
  trControl=ctrl,
  importance=TRUE,
  metric="Accuracy"
)
)
```

```
##    user  system elapsed
##  378.03    2.33  382.92
```

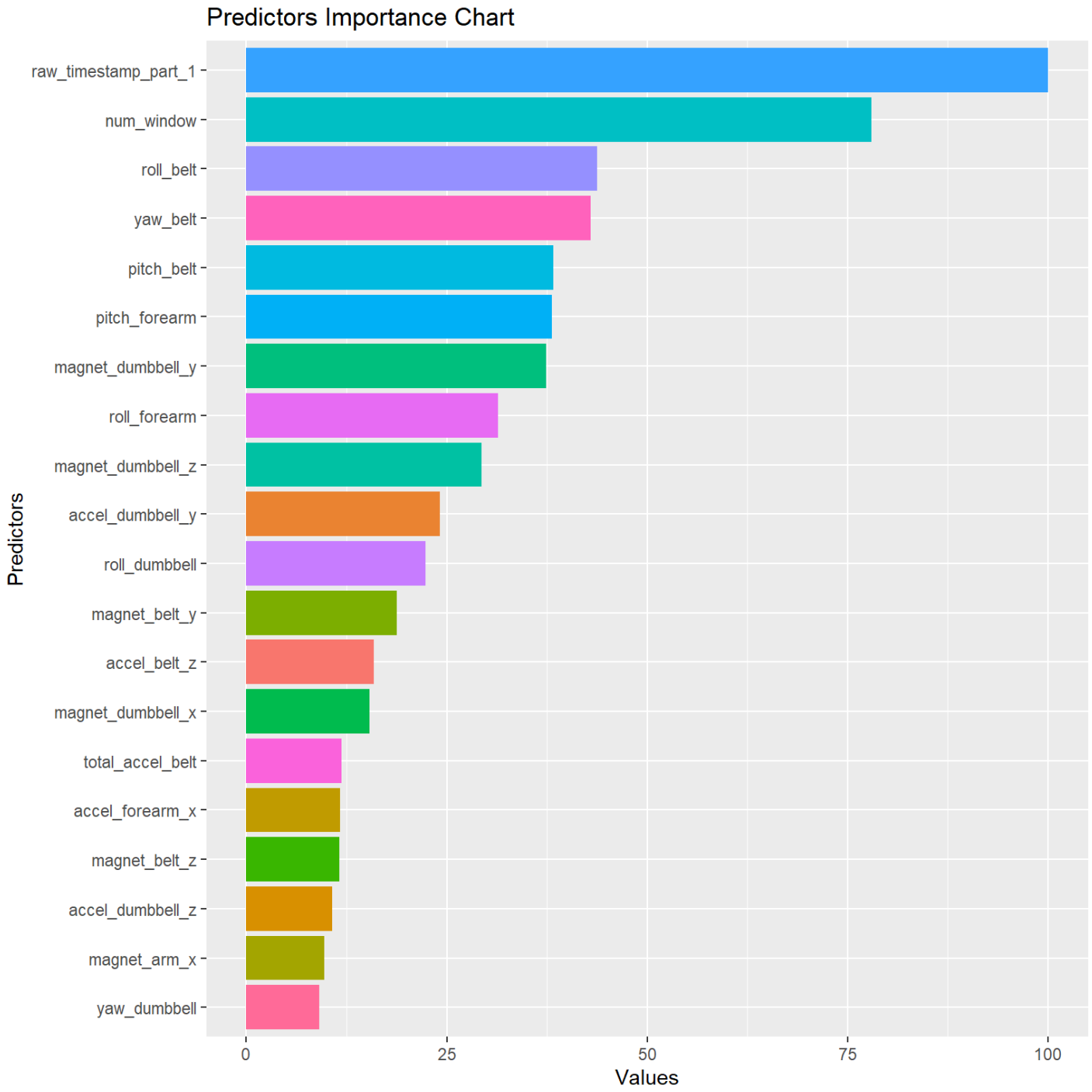
```
pred_bagging_3<-predict(model_bagging_3, testing)
round(confusionMatrix(testing$classe, pred_bagging_3)$overall, 3)
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##          0.999          0.998          0.997          1.000          0.285
## AccuracyPValue  McnemarPValue
##          0.000            NaN
```

So the bagging decision trees return us an accuracy rate around 99% which is a significant improvement comparing to previous single decision tree. But it does take a while to complete our model. What if we only use top 20 important predictors to construct our model? Will it be faster to complete our model, and will the accuracy rate decrease since we use less predictors?

```
library(dplyr)
library(ggplot2)
```

```
varImp<-varImp(model_bagging_3)$importance %>% arrange(desc(Overall))
top_20_predictors<-rownames(varImp)[1:20]
training_trim<-training[, c(top_20_predictors, "classe")]
```



4. Decision Tree with Bagging (top 20 variables)

```
set.seed(123)
system.time(
model_bagging_4<-train(
  classe~.,
  data=training_trim,
  method="treebag",
  trControl=ctrl,
  importance=TRUE,
  metric="Accuracy"
)
)
```

```
##    user  system elapsed
##  136.88    0.80  137.71
```

```
model_bagging_4$resample
```

```
##     Accuracy     Kappa Resample
## 1  0.9987261 0.9983886   Fold01
## 2  0.9993635 0.9991949   Fold02
## 3  0.9993631 0.9991944   Fold03
## 4  0.9974506 0.9967756   Fold04
## 5  0.9987261 0.9983888   Fold05
## 6  0.9987261 0.9983889   Fold06
## 7  0.9980880 0.9975817   Fold07
## 8  0.9980892 0.9975830   Fold08
## 9  0.9974539 0.9967801   Fold09
## 10 1.0000000 1.0000000   Fold10
```

```
model_bagging_4$results
```

```
##   parameter  Accuracy     Kappa  AccuracySD     KappaSD
## 1      none 0.9985986 0.9982276 0.000838698 0.001060711
```

From above, we can see the cross validation accuracy is around 99% which indicating only 1% **out of bag error rate**. Let's try it with our testing data set.

```
pred_bagging_4<-predict(model_bagging_4, testing)
confusionMatrix(testing$classe, pred_bagging_4)$overall
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.9994902      0.9993551      0.9981596      0.9999383      0.2849860
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```

We still manage to maintain around 99% accuracy rate, and improve the processing time. In other words, only using top 20 predictors doesn't sacrifice the model's accuracy rate.

5. Random Forest with ranger

Last, we'll try the random forest by using package ranger to see how the model performs.

```
library(ranger)
```

```
system.time(
model_rf_5<-ranger(classe~., data=training_trim)
)
```

```
##    user  system elapsed
##   46.11    0.25    5.22
```

```
model_rf_5
```

```
## Ranger result
##
## Call:
##  ranger(classe ~ ., data = training_trim)
##
## Type:                             Classification
## Number of trees:                  500
## Sample size:                      15699
## Number of independent variables:  20
## Mtry:                             4
## Target node size:                 1
## Variable importance mode:         none
## Splitrule:                        gini
## OOB prediction error:             0.05 %
```

Based on above result, we have only 0.05% **out of bag error rate** which is extremely small

```
pred_rf_5<-predict(model_rf_5, testing)
round(confusionMatrix(testing$classe, pred_rf_5$predictions)$overall, 3)
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##          0.998          0.998          0.997          0.999          0.285
## AccuracyPValue  McnemarPValue
##          0.000            NaN
```

So the random forest still provides a really good accuracy rate around 99% and the processing time is even shorter than using decision tree with bagging.

4. Apply selected models with test data set

Here, we'll use our best model "model_rf_5" to make predictions with our test data set.

```
pred_final<-predict(model_rf_5, test.data)
pred_final$predictions
```

```
##  [1] B A B A A E D B A A B C B A E E A B B B
## Levels: A B C D E
```