Practical Machine Learning

Course Project Write-up

develHector

07/20/2015

The goal of this assignment is to select and train an algorithm to **predict the manner in which some subjects did a weigth lifting exercise** versus certain standard. Data source are sourced genuinely from http://groupware.les.inf.puc-rio.br/har.

For the case the reader wants to replicate the whole process, it's been assumed the data is already downloaded from where indicated on the instructions and the updated version of R and its libraries are installed on the current computer. On this Markdown you will find parts of the source code to get to the results but feel free to explore the repo which contains the R script created to perform this assignment.

PD: No graphs or plots needed for me, only data matrices, hoping those are not considered as figures.

I also found useful the recommendation (thanks to the Discussion Forums) around using the parallel processing to exploit the maximum capacity of the computer, basically in order to accelerate the lengthy learning processes, so, I added the following lines to the beginning of my R scripting:

```
Cores <- parallel::detectCores() ;
registerDoParallel( cores = Cores )
```

1. How the model was built

1.1 Basic Exploratory Data Analysis

Before identifying what kind of Machine Learning method was suited best, first of all, in order to get an idea of what we were supposed to do, my start was to perform a visual checking and navigating of what kind of data was there. For example with the **str** command:

```
Read the Data 'data.frame': 19622 obs. of  160 variables:
Read the Data  $ X                       : int  1 2 3 4 5 6 7 8 9 10 ...
Read the Data  $ user_name               : Factor w/ 6 levels "adelmo","carlitos",..: 2 2 2 2 2 2 2 2 2 2
Read the Data  $ raw_timestamp_part_1    : int  1323084231 1323084231 1323084231 1323084232 1323084232 1
Read the Data   [list output truncated]
```

That gave an idea of the data being extremely highly-dimensional (i.e. too many classes and value possibilities) including several factor variables, numerical ones, negative numbers and Na's, making the basic linear algorithms maybe not suited for this problem.

1.3 Data Cleaning and Completing

Visually reviewing the training data initially shows that there are a certain number of columns with pure NA values. So, I removed them under the assumption that those were irrelevant for our study:

```
Training set dimensions before cleaning [1] 19622   160
```

A zero-variance analysis was performed, but was found no classes to be candidate to be removed because of that technique. Also a removal of NZV's (near-zero-variance) variables was performed, but all accuracy estimations reduced visibly, even causing me an sad error on my evaluation submission, so I decided to stick with the simple NA removal which led me to a significant reduction of the irrelevant classes, see:

```
Remove NAs from training [1] 19622    93
```

As an extra NA column removal, I turned to numeric all those factor classes, and after that, extra columns were candidate to be removed, check the dimensions of the training data now and the final number of classes really reduced, significantly reducing complexity and learning time. The final data set with the training data already cleaned and completed ended up being called **training.complete**:

```
dim( training.complete )
```

```
Final training dataset after NA column removing process [1] 19622    57
```

Later during the development and after a first training process, I needed to get back here because it was found that variables like counters and Time stamps were adding Standard Error to training measurements. Those identified as such were removed from the training data set as well.

The **caret::varImp** function kept on showing a strange 100% importance level on those kind of variables, that helped my turn the eye upon them and understand their mistake at considering them.

```
training.complete$X = NULL ;
training.complete$raw_timestamp_part_1 = NULL ;
training.complete$raw_timestamp_part_2 = NULL ;
training.complete$cvtd_timestamp = NULL ;
training.complete$num_window = NULL ;
```

Notes on Cleaning and Completing the Data

1. Principal Component Analysis was evaluated, but I found the accuracy of a basic learning algorithm drastically reduced, you can see the details on how that was performed.

2. Highly Correlated Predictors analysis was also checked, but I found it unnecessary that step since it benefited not that greatly to the current number of classes and time was a resource I started to get short of.

1.4 Subset Training Data just for initial Method Selection

With a reduced training sub-set of data (20% sub-partition) would lead me to train quickly and identify what required for me to easily choose a method. Then, after that, with a method already chosen, then re-train a fit model again with the whole training set.

```
Partition0 <- caret::createDataPartition( y = training.complete$classe, p = 0.20, list = FALSE ) ;
training.Partition0 <- training.complete[ Partition0,  ] ;
```

1.2 Method Selection Process

Under the practical impossibility of either knowing in advance or checking each of the almost 200 methods available in caret, in order to choose who may fit best to this particular case, so, I took a small list of those seen at the class or quizzes or related ones, and evaluated its **Accuracy**, it's **Speed of Training** and it's **Standard Errror** versus each other.

You will see on the code that I systematically evaluated the following methods, one by one:

- As observed since the begging **glm** didn't work because of highly-dimension data.

- **rpart** and **rpart2** didn't work because of highly-dimension data as well.
- Linear Discriminant Analysis **lda** produced high accuracy, low p-value of rejection, and quite fast to train.
- Stochastic Gradient Boosting **gbm** produced great accuracy and low p-value of rejection, but really slow to train.
- The Basic Random Forest one **rf** also high accuracy but also quite slow.
- The one I liked the most was the **treebag** CART, which ended up being accurate, fast and low error prone.

The situation with **treebag** is that it demands cross validation as its nature calls for increased over-fitting possibilities and even bias extension or conservation instead of good accuracy.

Yes I removed the columns with pure NA values, but not those who contained both NA and values, and for that, I found this variable 'na.action = na.roughfix' of great help. Was found empirically just checking options until either it allowed me to continue or I showed speed or accuracy improving):

2. How cross validation was used

For that it was done through k-fold cross validation, so I created 10 subsets of training and testing data with something like the following:

```
training.TrainFolds <- createFolds( y=training.complete$classe, k=10, returnTrain=T )

for( i in 1:length( training.TrainFolds ) ) {

    # Create a K-th training and a testing fold
    training.kfold <- training.complete[ training.TrainFolds[[i]], ] ;
    testing.kfold <- training.complete[ -training.TrainFolds[[i]], ] ;

    # Train the K'th model with the Training K'th fold
    modFit <- train( classe ~ . , method="treebag", data=training.kfold, na.action=na.roughfix ) ;

    # Test the K'th model with the Testing K'th fold
    modPredict <- predict( modFit, newdata = testing.kfold, na.action = na.roughfix )

    # Get the K'th Accuracy statistics of the testing prediction moel
    cmtPredict <- confusionMatrix( testing.kfold$classe, modPredict  )

  }
```

A cross-validation matrix showed all folds accuracy and other data, turned clear this was on the correct path:

```
    Accuracy     Kappa AccuracyLower AccuracyUpper AccuracyNull AccuracyPValue
1  0.9979613 0.9974214     0.9947883     0.9994442    0.2844037               0
2  0.9954105 0.9941954     0.9913057     0.9978993    0.2835288               0
3  0.9943906 0.9929057     0.9899855     0.9971966    0.2835288               0
4  0.9959225 0.9948426     0.9919816     0.9982380    0.2844037               0
5  0.9969403 0.9961300     0.9933524     0.9988764    0.2845487               0
6  0.9949058 0.9935555     0.9906515     0.9975545    0.2857871               0
7  0.9928681 0.9909841     0.9880627     0.9960956    0.2801834               0
8  0.9969450 0.9961369     0.9933625     0.9988781    0.2825866               0
9  0.9969419 0.9961323     0.9933558     0.9988769    0.2838940               0
10 0.9969435 0.9961341     0.9933591     0.9988775    0.2842588               0
```

3. Expected out of sample error

The possibility of error depends on the complement to the perfect accuracy. The worst case would be the probability of all our cross validation accuracy measurements to fail, which will imply a **4%** possibility of error on the worst case scenario (under a hypothetical case on where all outliers were present):

```
1 - ( 0.9979613 * 0.9954105 *  0.9943906 *  0.9959225 * 0.9969403 * 0.9949058 * 0.9928681 * 0.9969450 *
```

```
## [1] 0.04004088
```

Confusion Matrix on Training Dataset to Predict Out Sample Error When we apply our full-trained model to the completed training data, we can produce the confusion matrix that quickly shows the number of values that were predicted incorrectly. On this case only **4 out of 19,622 cases** were incorrectly qualified, making us to believe that our model will turn to a **0.2%** of sample error can be produced:

```
Confusion Matrix and Statistics
--------------------------------

          Reference
Prediction    A    B    C    D    E
         A 5579    1    0    0    0
         B    1 3795    1    0    0
         C    0    0 3422    0    0
         D    0    0    1 3215    0
         E    0    0    0    0 3607
```

After this was evaluated, I took the whole training set (already cleaned and completed) and passed through the training process, that lead to the results I applied on the submission.

Statistics to produce Sample Errors

The Negative Predictive Value will portray our probability of produce results out of the expected value. You will see that in our case, that power is almost 100%.

```
                     Class: A Class: B Class: C Class: D Class: E
Sensitivity            0.9998   0.9997   0.9994   1.0000   1.0000
Specificity            0.9999   0.9999   1.0000   0.9999   1.0000
Pos Pred Value         0.9998   0.9995   1.0000   0.9997   1.0000
```

```
Neg Pred Value         0.9999   0.9999   0.9999   1.0000   1.0000
Prevalence             0.2844   0.1935   0.1745   0.1638   0.1838
Detection Rate         0.2843   0.1934   0.1744   0.1638   0.1838
Detection Prevalence   0.2844   0.1935   0.1744   0.1639   0.1838
Balanced Accuracy      0.9999   0.9998   0.9997   1.0000   1.0000
```

The low error resulted from applying our prediction model algorithm to the data set comes from the reality that our training process was performed with the most reduced number of classes possible, getting only those who really represented some variance.

4. Why the choices were made this way

Based on four main principles:
1. Maximizing Accuracy
2. Minimizing Processing Time
3. Reducing Expected Error
4. Systematic evaluation of alternatives

And observing for the final "production" testing exactly same rules:

```
# Model being created again with the cleaned Training dataset
message( "Predicting Full Training Set" )
modFit <- caret::train( classe ~ ., method = "treebag", data = training.complete,
                        na.action = na.roughfix, verbose=FALSE ) ;

# Model being predicted with the cleaned Testing dataset
modPredict <- predict( modFit, newdata = testing.complete, na.action = na.pass )
modPredict

# These are the results per test case
[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
```

As commented at the beginning, all source code will be available on the repository for your review (or amusement). I was unable to place it fully in here because of space restrictions, and because I preferred to explain the hows and whys of the process.

The best of lucks with your own project, thanks!

- Measured number of words: 1774 before this line, hoping I didn't bore you guys with all this.