intro_to_caret_part2

Sheng Ming

April 10, 2018

In part 1 of the tutorial, we mainly discussed the data processing and splitting methods. Now let's talk about the modeling process.

3. Model training and tuning

3.0 R Markdown Setup

Before we write any codes, we could customize the options for R code chunks globally. You can overwrite local chunk options by modifying the chunk headers (inside the {r}).

For the second half of the tutorial, we'll build some complicated models which are computational intensive and time consuming (compared to the `ISLR` textbook examples). Every time we knit the rmd, by default it will run all the codes in the file. For this tutorial, it'll probably take you more than 5 minutes to generate the pdf. By setting cache = TRUE, the results of every code chunk is stored locally after you execute it for the first time. So you don't run the codes again every time you knit unless you modify them.

```
knitr::opts_chunk$set(cache = TRUE, message = FALSE, warning = FALSE)
```

3.1 What models should we consider?

At the very beginning of the modeling part, this is always the first question we should think about. Different models have different power and interpretability, need different computational costs and involve different feature selection and parameter tuning processes. Some of the models work specially well for certain data types (i.e. convolutional neural network for computer vision problems).

For the titanic data, there are many dummy variables and higher-order interactions between different variables may be worth to consider. Therefore tree-based models are considered for their abilities to explain non-linear relations and some interpretability (with the help of the variable importance metrics, which will be covered below).

In the tutorial, I tried Random Forest and eXtreme Gradient Boosting Trees (known as `xgboost`, more details in https://cran.r-project.org/web/packages/xgboost/xgboost.pdf). `xgboost` was covered in another DSO 530 R tutorial. You should also try other models that you think might work well for this problem.

Now, let's first load the processed data and fitted model objects which we saved to our local drive in the end of part 1 of the tutorial to our workspace:

```
dummy.vars = readRDS("dummies.rds")
pre.scale = readRDS("scale.rds")
pre.impute = readRDS("impute.rds")

train.scaled = read.csv('train_complete.csv')
titanic.train = read.csv('titanic_train.csv')
titanic.test = read.csv('titanic_test.csv')
train.scaled$Survived = as.factor(train.scaled$Survived)
titanic.train$Survived = as.factor(titanic.train$Survived)
titanic.test$Survived = as.factor(titanic.test$Survived)
```

3.2 Feature Selection

Are we good to go now? Yes. But there is still one thing we can do, feature selection. You may ask, haven't we already done that in section 1? Not really. What we did in section 1 is called feature pre-screening, where we focus on the data quality and the relations between the predictors. Here, we want to focus on the relation between predictors and the Survived class. Why bother? First, many of the features may not be very informative or even they are just noise variables. Including those features in our model will increase variance and lead to overfitting. Second, we want our models to be interpretable. Keeping it simple can help you build actions in real life and reason your model to the other business partners.

There are mainly three categories of feature selection methods:

- **Built-in methods**: many models produce prediction equations that do not necessarily use all the predictors. These models are thought to have built-in feature selection. (i.e. lasso, tree-based models, etc.)
- **Filter methods**: filter methods evaluate the relevance of the predictors outside of the predictive models and subsequently model only the predictors that pass some criterion. (i.e. correlation, mutual information, etc.) The marginal screening we learned in class belongs to this category.
- **Wrapper methods**: wrapper methods evaluate multiple models using procedures that add and/or remove predictors to find the optimal combination that maximizes model performance. (i.e. forward selection, backward selection, etc.)

Note: such a division of methods is not necessarily a universally adopted way. For example, the `ISLR` textbook does not use this language.

Now let's see examples for these three methods.

3.2.1 Built-in feature selection

Here, we use the very powerful **train** function in `caret` to train a random forests model (method = "rf") with 5-fold cross-validation repeated 5 times. In the `ISLR` textbook and our DSO 530 class, we talked about 5-fold CV once. It does not hurt to repeat the process multiple times to increase stability, as long as computation power can handle. The **trainControl** function is used to control the computational nuances of the **train** function. Here we specify "repeatedcv" as the resampling method for repeated cross validation. You can also set other resampling methods like bootstrapping and leave-one-out cross-validation. **Resampling** is one of the key terms in `caret` because how you resample the data will also affect your model performance. To help you understand, for instance, each iteration in the 5-fold cv can be regarded as one resampling iteration. Therefore, a 5-fold cv repeated 5 times has 25 resampling iterations.

One thing to mention here. For the feature selection examples below, I only use the random forest model for the purpose of simplification. As I mentioned, different models could have different feature selection procedures so you might want to do this separately for different models.

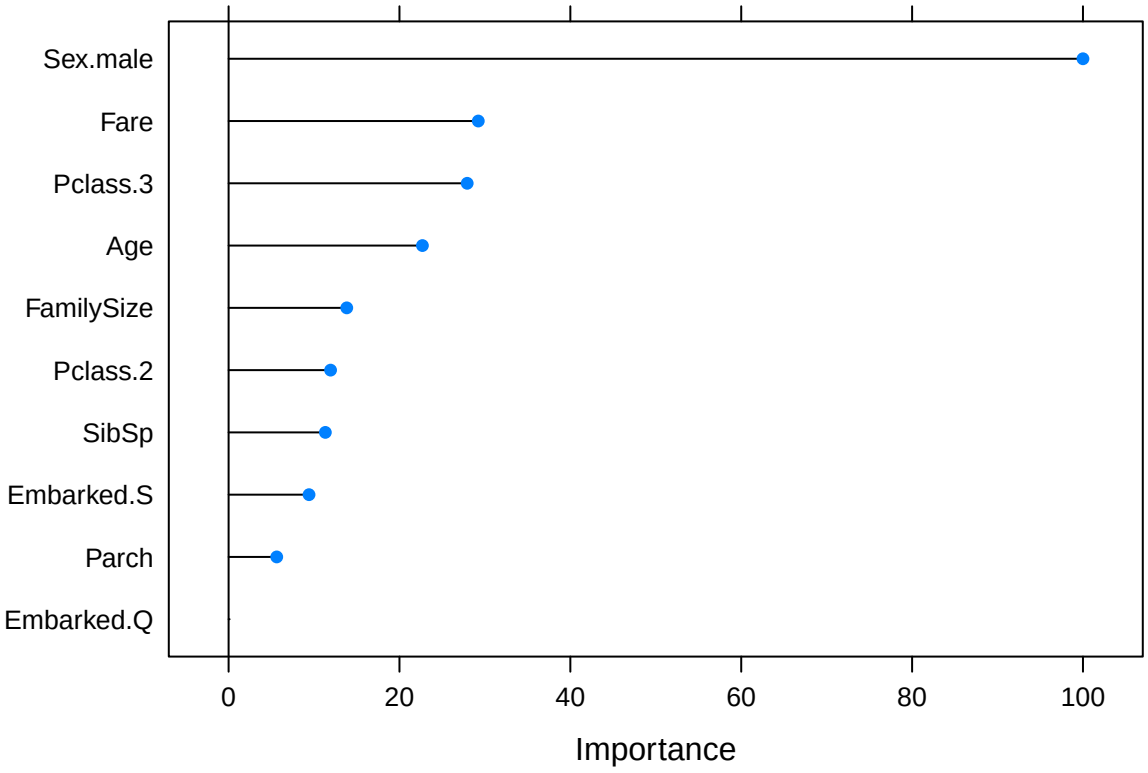
The **varImp** function returns the variable importance metric given by the training result. Again, different models have different ways to measure the variable importance. See details at http://topepo.github.io/caret/variable-importance.html.

```
# Need to have randomForest installed to call 'rf' method in caret
#install.packages("randomForest")
library(caret)
rfCtrl1 = trainControl(method='repeatedcv', number = 5, repeats = 5)
set.seed(555)
rf = train(Survived ~ .,
           data = titanic.train,
           method = 'rf',
           trControl = rfCtrl1,
```

```
           importance = TRUE)
print(varImp(rf))
```

```
## rf variable importance
##
##             Importance
## Sex.male       100.000
## Fare            29.224
## Pclass.3        27.928
## Age             22.685
## FamilySize      13.831
## Pclass.2        11.936
## SibSp           11.326
## Embarked.S       9.416
## Parch            5.634
## Embarked.Q       0.000
```

```
plot(varImp(rf))
```



We see that the top 4 variables Sex.male, Fare, Pclass.3 and Age significantly outweigh other variables (the varImp function by default applies a min-max normalization).

3.2.2 Univariate filtering

Alternatively, each predictor could be individually evaluated to check if there is a plausible relationship between it and the observed classes (this is the marginal screening idea mentioned in Lecture 6 of DSO 530). Only predictors with important relationships would then be included in a classification model. The **sbf** function (selection by filter) handles these operations.

Whlie the function is just one line of code, it actually wraps a sequence of functions which score each

variable individually, filter the scores with a certain threshold and then fit the model based on the selected features. To know the details of how exactly these work, please refer to the caret documentation http://topepo.github.io/caret/feature-selection-using-univariate-filters.html .

In the below example, we performed this univariate feature filtering process with a 5-fold cross-validation repeated 5 times on our training data, which means we'll do the filtering on the entire training data as well as the 25 resamples respectively (see section 3.2.1 to know why it's 25 here).

Note: in the result below, the 6 selected variables are the ones derived from the full training set, not the top 6 selected variables out of all the resampling results. Therefore, if the variance of the resampling results is large, it indicates that the result from the univariate filtering method may not be very reliable.

```
rfCtrl2 = sbfControl(functions = rfSBF, method = 'repeatedcv', number = 5, repeats = 5)
# Perform selection by filter
set.seed(555)
rfWithFilter = sbf(x = titanic.train[, -1],
                   y = titanic.train[, 1],
                   sbfControl = rfCtrl2)
rfWithFilter
```

```
##
## Selection By Filter
##
## Outer resampling method: Cross-Validated (5 fold, repeated 5 times)
##
## Resampling performance:
##
##  Accuracy  Kappa AccuracySD KappaSD
##    0.7933 0.5334    0.02199 0.05452
##
## Using the training set, 6 variables were selected:
##    Pclass.2, Pclass.3, Sex.male, Parch, Fare...
##
## During resampling, the top 5 selected variables (out of a possible 6):
##    Embarked.S (100%), Fare (100%), Pclass.3 (100%), Sex.male (100%), Parch (60%)
##
## On average, 5.2 variables were selected (min = 4, max = 6)
```

```
predictors(rfWithFilter)
```

```
## [1] "Pclass.2"   "Pclass.3"   "Sex.male"   "Parch"      "Fare"
## [6] "Embarked.S"
```

As the result shows, the univariate filtering method selects 6 variables from the entire training set. While there is some variance in the resampling results, the number of predictors selected is around 4-6.

3.2.3 Recursive Feature Elimination

The recursive feature elimination is like a forward or backward feature selection process. You specify the sizes of subsets of predictors you want to try, and the algorithm will keep the most important variables in the model based on the variable importance metric and do the training/validation process over each resampling iteration. It's usually more accurate than the sbf method as it takes into account the possible interactions among the variables but it is also more computational expensive.

The computation structure of rfe with resampling is shown as below:

Algorithm 2: Recursive feature elimination incorporating resampling

2.1 **for** *Each Resampling Iteration* **do**
2.2 Partition data into training and test/hold–back set via resampling
2.3 Tune/train the model on the training set using all predictors
2.4 Predict the held–back samples
2.5 Calculate variable importance or rankings
2.6 **for** *Each subset size* S_i, $i = 1 \ldots S$ **do**
2.7 Keep the S_i most important variables
2.8 [Optional] Pre–process the data
2.9 Tune/train the model on the training set using S_i predictors
2.10 Predict the held–back samples
2.11 [Optional] Recalculate the rankings for each predictor
2.12 **end**
2.13 **end**
2.14 Calculate the performance profile over the S_i using the held–back samples
2.15 Determine the appropriate number of predictors
2.16 Estimate the final list of predictors to keep in the final model
2.17 Fit the final model based on the optimal S_i using the original training set

```
rfctrl3 = rfeControl(functions = rfFuncs,
                     method = "repeatedcv",
                     number = 5,
                     repeats = 5,
                     verbose = FALSE)
# Specify the size of models to be tried
subsets = 4:8
# Perform recursive feature elimination
set.seed(555)
rfWithRfe = rfe(x = titanic.train[, -1],
                y = titanic.train[, 1],
                sizes = subsets,
                rfeControl = rfctrl3)
rfWithRfe
```

```
##
## Recursive feature selection
##
## Outer resampling method: Cross-Validated (5 fold, repeated 5 times)
##
## Resampling performance over subset size:
##
##  Variables Accuracy  Kappa AccuracySD KappaSD Selected
##          4   0.8176 0.5963    0.02894 0.06610        *
##          5   0.8118 0.5849    0.02811 0.06279
##          6   0.8122 0.5853    0.02782 0.06261
```

```
##          7   0.8042 0.5686    0.03176 0.07167
##          8   0.8048 0.5675    0.02572 0.05875
##         10   0.8090 0.5818    0.03194 0.07107
##
## The top 4 variables (out of 4):
##    Sex.male, Fare, Pclass.3, Age
```

```
predictors(rfWithRfe)
```

```
## [1] "Sex.male" "Fare"     "Pclass.3" "Age"
```

A subset size of 4 produces the optimal overall prediction accuracy, The selected predictors are Sex.male, Fare, Pclass.3 and Age.

So what shall we do now? The results from the three feature selection methods are generally consistent with some variation. How you make the decision here depends on the type of model you use and other relevant factors. Here I choose the variables Sex.male, Fare, Age and Pclass.3, but you are free to try on other subsets of predictors.

```
selected = c("Sex.male", "Fare", "Age", "Pclass.3", "Survived")
titanic.train.top4 = titanic.train[, which(colnames(titanic.train) %in% selected)]
```

3.3 Tuning tuning parameters with cross-validations

tuning parameters are those parameters that are not trainable but can play a key role in the model success. Therefore, besides "training" a model, we also need to "tune" it. Grid search is one of the most common and straight-forward tuning parameters searching methods: you list a set of possible values and the algorithm will iterate over every possible combination of tuning parameters and then choose the combination giving the optimal result.

The computation strucure of parameter tuning is shown as below:

```
Define sets of model parameter values to evaluate
for each parameter set do
    for each resampling iteration do
        Hold–out specific samples
        [Optional] Pre–process the data
        Fit the model on the remainder
        Predict the hold–out samples
    end
    Calculate the average performance across hold–out predictions
end
Determine the optimal parameter set
Fit the final model to all the training data using the optimal parameter set
```

3.3.1 Tuning the random forest model

For the default random forest model in caret, there is only one tunable tuning parameter, mtry – number of predictors sampled for spliting at each node. If you want to tune more parameters, you may need to customize your own random forest model from the raw **randomForest** package. Check this post: https://stackoverflow.com/questions/38625493/tuning-two-parameters-for-random-forest-in-caret-package

```
rf.control = trainControl(method = "repeatedcv",
                          number = 5,
```

```
                          repeats = 5,
                          search = "grid")
# List all the possible values you want to try
rf.grid = expand.grid(mtry = 2:8)
```

Train and tune the random forest model on both full predictor set and the top 4 subset. The returned object of class **train** contains the resampling results as well as the best model fitted on the entire training set, which we'll use to evaluate on the test set later.

Note: in the following model training jobs, I set the same seed 530 for all the models. That's because I want to make sure each model is trained on the same resampling version of data, hence their results are more comparable.

```
# Train on full predictor set
set.seed(530)
rfTrain = train(Survived ~ .,
                data = titanic.train,
                method = "rf",
                tuneGrid = rf.grid,
                trControl = rf.control)
rfTrain
```

```
## Random Forest
##
## 625 samples
##  10 predictor
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 500, 500, 500, 500, 500, 500, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy  Kappa
##   2     0.81152   0.5812157
##   3     0.80992   0.5845186
##   4     0.80992   0.5879667
##   5     0.80416   0.5774710
##   6     0.79424   0.5580287
##   7     0.79040   0.5509982
##   8     0.78784   0.5451550
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 2.
```

```
# Train on top 4 predictor subset
set.seed(530)
rfTrain.top4 = train(Survived ~ .,
                     data = titanic.train.top4,
                     method = "rf",
                     tuneGrid = rf.grid,
                     trControl = rf.control)
rfTrain.top4
```

```
## Random Forest
##
```

```
## 625 samples
##   4 predictor
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 500, 500, 500, 500, 500, 500, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy  Kappa
##   2     0.81568   0.5949910
##   3     0.78592   0.5397864
##   4     0.77408   0.5158030
##   5     0.77952   0.5278379
##   6     0.77664   0.5206021
##   7     0.77632   0.5205987
##   8     0.77760   0.5227345
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 2.
```

For both the RF model with full predictor set and the one with top 4 predictors, mtry = 2 achieves the best average result over all resampling validations.

3.3.2 Tuning the xgboost model

For the eXtreme gradient boost model, there are more tunable tuning parameters (you may want to check out the R xgboost documentation or the xgboost tutorial made by Shunan to see the definition of each parameter below)

```
xgb.control = trainControl(method = "repeatedcv",
                           number = 5,
                           repeats = 5,
                           search = "grid")
xgb.grid = expand.grid(eta = c(0.05, 0.10, 0.15),
                       nrounds = c(50, 75, 100),
                       max_depth = 3:5,
                       min_child_weight = c(2.0, 3.0, 4.0),
                       colsample_bytree = c(0.4, 0.5, 0.6),
                       gamma = 0,
                       subsample = 1)
dim(xgb.grid)
```

```
## [1] 243   7
```

So how many seperate models do we need to train here? Let's do a simple math. We will perform a 5-fold cross validation repeated 5 times over 243 possible combinations of tuning parameters. 5 * 5 * 243 = 6075. We are going to train 6075 seperate xgboost trees! Luckily for the titanic data, as the data volume is very small, it's feasible to run it on your computer. But for a much larger dataset, you may need more powerful computing resources.

Parallel Training

The Good news is, there are some packages in R which can enable caret to train in parallel, which could significantly save the runtime (depending on what kind of model you use). **doSNOW** is one of such packages. While there are many package options in this space (doMC, doParallel, etc.), doSNOW has the advantage of working on both Windows and Mac OS X.

Note: Tune the value of **spec** based on the number of cores/threads available on your machine, otherwise R will report an error (if your machine only has one CPU, you can only set spec = 1, hence no improvement in your runtime)

```
# Need to install the required packages
#install.packages(c("doSNOW", "xgboost"))
library(doSNOW)
cl = makeCluster(spec = 1, type = 'SOCK')
# Register the cluster
registerDoSNOW(cl)
# Keep a count of the runtime of xgb training
# Train on full predictor set
set.seed(530)
system.time(xgbTrain <- train(Survived ~ .,
                  data = titanic.train,
                  method = "xgbTree",
                  tuneGrid = xgb.grid,
                  trControl = xgb.control))
```

```
##    user  system elapsed
##  78.163   0.380  83.025
```

```
# Show the best tuning parameter combination
xgbTrain$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 91      50         3 0.1     0              0.5                2         1
```

```
# Train on top 4 predictor subset
set.seed(530)
system.time(xgbTrain.top4 <- train(Survived ~ .,
                      data = titanic.train.top4,
                      method = "xgbTree",
                      tuneGrid = xgb.grid,
                      trControl = xgb.control))
```

```
##    user  system elapsed
##  60.185   0.378  67.354
```

```
# Show the best tuning parameter combination
xgbTrain.top4$bestTune
```

```
##    nrounds max_depth  eta gamma colsample_bytree min_child_weight
## 10      50         3 0.05     0              0.5                2
##    subsample
## 10         1
```

```
# Stop the cluster
stopCluster(cl)
```

Alternating performance metrics

Now we've finished tuning all the tuning parameters. But there is still one most important component we could adjust – the performance metric (the metric we train the model to optimize). By default, the training job is optimized on RMSE for regression models and accuracy for classification models in caret. But you can change or even customize it depending on what you want. For instance, if a false negative will incurs a

severe cost for your modeling (imagine you're building a model for critical disease diagnosis), you may want to change your optimized metric from accuracy to sensitivity (true positive rate).

Here, we use the random forest model as example. Let's say we want to use the ROC as the optimized metric. We need to do some minor changes in our code. Check http://topepo.github.io/caret/model-training-and-tuning.html#customizing-the-tuning-process to see more details.

```
# Need to convert class value to non-numeric to avoid naming error
titanic.train$Survived = as.factor(ifelse(titanic.train$Survived == "1", "Yes", "No"))
rf.control = trainControl(method = "repeatedcv",
                          number = 5,
                          repeats = 5,
                          search = "grid",
                          ## Estimate class probabilities
                          classProbs = TRUE,
                          ## Evaluate performance using
                          ## the following function
                          summaryFunction = twoClassSummary)
# List all the possible values you want to try
rf.grid = expand.grid(mtry = 2:8)
# Train on top 4 predictor subset
set.seed(530)
rfTrain.roc = train(Survived ~ .,
                    data = titanic.train,
                    method = "rf",
                    trControl = rf.control,
                    tuneGrid = rf.grid,
                    verbose = FALSE,
                    ## Specify which metric to optimize
                    metric = "ROC")
rfTrain.roc
```

```
## Random Forest
##
## 625 samples
##  10 predictor
##   2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 500, 500, 500, 500, 500, 500, ...
## Resampling results across tuning parameters:
##
##   mtry  ROC        Sens       Spec
##   2     0.8458766  0.9262338  0.6275000
##   3     0.8459361  0.8987013  0.6675000
##   4     0.8433820  0.8857143  0.6883333
##   5     0.8392749  0.8737662  0.6925000
##   6     0.8360011  0.8587013  0.6908333
##   7     0.8332251  0.8509091  0.6933333
##   8     0.8306548  0.8493506  0.6891667
##
## ROC was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 3.
```

3.4 Choosing the best model

Now we've trained and tuned four models: random forest on full predictor set, random forest on top 4 predictors, xgb trees on full predictor set and xgb trees on top 4 predictors. Let's compare the results of these four models. The **resamples** function helps to collect all the resampling results. Since models are fit on the same versions of the training data, it makes sense to make inferences on the differences between models. the **diff** function helps to compute the differences, then use a simple t-test to evaluate the null hypothesis that there is no difference between the models.

```
resamps = resamples(list(RF1 = rfTrain,
                         RF2 = rfTrain.top4,
                         XGB1 = xgbTrain,
                         XGB2 = xgbTrain.top4))
# Execute the line below to see the details of the resampling results
# resamps$values
summary(resamps)
```

```
##
## Call:
## summary.resamples(object = resamps)
##
## Models: RF1, RF2, XGB1, XGB2
## Number of resamples: 25
##
## Accuracy
##       Min. 1st Qu. Median   Mean 3rd Qu.  Max. NA's
## RF1  0.760   0.792  0.808 0.8115   0.824 0.880    0
## RF2  0.760   0.800  0.816 0.8157   0.832 0.872    0
## XGB1 0.768   0.800  0.816 0.8150   0.824 0.856    0
## XGB2 0.712   0.760  0.768 0.7680   0.776 0.816    0
##
## Kappa
##        Min. 1st Qu. Median   Mean 3rd Qu.   Max. NA's
## RF1  0.4763  0.5432 0.5776 0.5812  0.6096 0.7392    0
## RF2  0.4587  0.5569 0.6001 0.5950  0.6258 0.7251    0
## XGB1 0.4998  0.5706 0.5935 0.5928  0.6159 0.6806    0
## XGB2 0.3766  0.4466 0.4846 0.4731  0.4938 0.5725    0
```

```
difValues = diff(resamps)
summary(difValues)
```

```
##
## Call:
## summary.diff.resamples(object = difValues)
##
## p-value adjustment: bonferroni
## Upper diagonal: estimates of the difference
## Lower diagonal: p-value for H0: difference = 0
##
## Accuracy
##      RF1       RF2       XGB1      XGB2
## RF1            -0.00416  -0.00352   0.04352
## RF2  1                    0.00064   0.04768
## XGB1 1         1                    0.04704
## XGB2 1.207e-06 2.669e-09 9.010e-11
```

```
##
## Kappa
##       RF1        RF2        XGB1       XGB2
## RF1             -0.01378  -0.01163   0.10808
## RF2  1                     0.00215   0.12186
## XGB1 1          1                    0.11971
## XGB2 1.567e-08 9.095e-10 1.234e-11
```

The lower diagnonal triangle from the above result displays the p-values from the pairwise t-test and there is no significant difference among the first three models, while the last one, the xgboost model on top 4 predictors is significantly inferior. Why did that happen? Remember in section 3.1 and 3.2, I mentioned that different models could have different feature selection process and different best subset of features. In the tutorial, we selected the features in favor of the random forest model. Therefore, it's no surprising to see the xgboost model underperforms with the selected features.

You may choose any of the first three models as your final model since they have almost equivalent performance (while the second and third ones seem to be slightly better), or even you can ensemble the results of the three. In practice, a less complex model is usually preferred. Here the random forest model on top 4 predictors is the simplest one, let's use it as our final model.

4. Evaluating on the test set

So far we've used cross-validations on the training set to determine the optimal hyparameters for each model and the best model. But we still want to evaluate the result on an "unseen" dataset. Why bother? That's because the learned models in the cross-validation are correlated as they share a same portion of the training samples. Therefore, the best model you derive from the cv could still be somewhat overfitting the training data and it's necessary to use an independent dataset to derive an unbiased estimate of the model.

Note: if you used a separate validation set instead of cross-validation, you'll still need another independent test set for the similar reason. You selected the best model based on the validation set, which means you "optimized" your model based on both the training and validation data, hence there could be an overfitting issue.

We use the model fitted on the training set to predict the Survived class in the test set and use the **confusionMatrix** function to estimate the effectiveness.

```
# Show the fitted model with the best tuning parameter
rfTrain.top4$finalModel
```

```
##
## Call:
##  randomForest(x = x, y = y, mtry = param$mtry)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 17.92%
## Confusion matrix:
##     0   1 class.error
## 0 354  31  0.08051948
## 1  81 159  0.33750000
```

```
preds.rf = predict(rfTrain.top4, titanic.test)
confusionMatrix(preds.rf, titanic.test$Survived)
```

```
## Confusion Matrix and Statistics
##
```

```
##           Reference
## Prediction   0   1
##          0 152  25
##          1  12  77
##
##                Accuracy : 0.8609
##                  95% CI : (0.8134, 0.9001)
##     No Information Rate : 0.6165
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.6986
##  Mcnemar's Test P-Value : 0.04852
##
##             Sensitivity : 0.9268
##             Specificity : 0.7549
##          Pos Pred Value : 0.8588
##          Neg Pred Value : 0.8652
##              Prevalence : 0.6165
##          Detection Rate : 0.5714
##    Detection Prevalence : 0.6654
##       Balanced Accuracy : 0.8409
##
##        'Positive' Class : 0
##
```

We see the prediction accuracy on the test set is pretty high, even higher than what we have for most of the training resampling iterations. That's very unusual in reality because as mentioined, we optimized the parameters based on the training and validation data. However, due to the small size of the test set for our titanic example, the variance is expected to be larger.

Since no significant overfitting we see from the test result, we can finally make the judge call that our model is well behaved and could be used for future prediction.

5. Making predictions on unlabeled data

The final step is to make predictions on the unlabeled data, that is the test.csv you've already downloaded. We'll revisit the code in chapter one to process the data. Save the result in a file and you could then submit it on the Kaggle titanic website https://www.kaggle.com/c/titanic/submit to see the accuracy of your predictions.

```
titanic.new = read.csv('test.csv', stringsAsFactors = FALSE)
str(titanic.new)
```

```
## 'data.frame':    418 obs. of  11 variables:
##  $ PassengerId: int  892 893 894 895 896 897 898 899 900 901 ...
##  $ Pclass     : int  3 3 2 3 3 3 3 2 3 3 ...
##  $ Name       : chr  "Kelly, Mr. James" "Wilkes, Mrs. James (Ellen Needs)" "Myles, Mr. Thomas Francis
##  $ Sex        : chr  "male" "female" "male" "male" ...
##  $ Age        : num  34.5 47 62 27 22 14 30 26 18 21 ...
##  $ SibSp      : int  0 1 0 0 1 0 0 1 0 2 ...
##  $ Parch      : int  0 0 0 0 1 0 0 1 0 0 ...
##  $ Ticket     : chr  "330911" "363272" "240276" "315154" ...
##  $ Fare       : num  7.83 7 9.69 8.66 12.29 ...
##  $ Cabin      : chr  "" "" "" "" ...
##  $ Embarked   : chr  "Q" "S" "Q" "S" ...
```

```
# Feature pre-screening
colsExd = c('PassengerId', 'Cabin', 'Name', 'Ticket')
titanic.fltrd = titanic.new[, -which(colnames(titanic.new) %in% colsExd)]
# Perform dummy encoding
titanic.fltrd$Pclass = as.factor(titanic.fltrd$Pclass)
titanic.fltrd$Sex = as.factor(titanic.fltrd$Sex)
titanic.fltrd$Embarked = as.factor(titanic.fltrd$Embarked)
titanic.dummy = predict(dummy.vars, newdata = titanic.fltrd)
# Impute the missing value
titanic.imputed = as.data.frame(predict(pre.impute, titanic.dummy))
# Create new variable
titanic.imputed$FamilySize = 1 + titanic.imputed$SibSp + titanic.imputed$Parch
# Standardize the features
titanic.scaled = predict(pre.scale, titanic.imputed)
# Make predictions
titanic.preds = predict(rfTrain.top4, titanic.scaled)
# Add the predicted values to the data
titanic.new$Survived = titanic.preds
# Save the results for Kaggle submission
write.csv(titanic.new[, c('PassengerId', 'Survived')],
          'test_with_predictions.csv',
          row.names = FALSE)
```

Re-train the model on the entire training data

Empirically, it's usually preferred to re-train the model with the tuned tuning parameters over all the training samples you have (here it means including all the samples in the train.csv and forget the train/test split) before making the predictions on new unseen data as generally the more data you have the better model you get. However, you should note that:

1. The re-trained model doesn't have a validation set since you've already used all the labeled data you have.
2. For some models the optimal tuning parameters are dependent on the sample size. For example, the optimal min_samples_leaf (minimum number of samples at a leaf node) for decision tree should be larger as the sample size increases.

```
rfTrain.full = train(Survived ~ .,
                     # Change the data to the full training set
                     data = train.scaled,
                     method = "rf",
                     tuneGrid = data.frame(mtry = 2),
                     trControl = trainControl(method = "none"))
titanic.preds.full = predict(rfTrain.full, titanic.scaled)
titanic.new$Survived = titanic.preds.full
write.csv(titanic.new[, c('PassengerId', 'Survived')],
          'test_with_predictions2.csv',
          row.names = FALSE)
```