Tutorial: 0.433 score with randomForest in R

Hello there,

Ever wondered how many bikes were rented in Washington at the end of each month in 2011 and 2012? Or just want to improve your score in kaggle and learn some R? You came to the right place! This tutorial will provide you with a score around 0.433, which (at the time of writing) would put you in the top 15% of the leaderboard. By following this tutorial you will learn how to do basic data manipulation with R, perform basic Feature Engineering and use the randomForest package in R.

I strongly suggest the use of RStudio, which lets you save R scripts and gives a nice overview of all your plots and data.

This is my first tutorial after I took on my first kaggle competition without following tutorials myself. If you find flaws or factual mistakes in my explanations feel free to point them out. After all I'm just a n00b hobbyist trying to improve.

DISCLAIMER: It just came to my attention that there is a special rule for the competition, which states that only data known at the time can be used to predict each month. This tutorial violates that rule.

Reading the data

Enough talk, let's start: First of all we will have to read in the data from the provided csv files. Download the files from kaggle and put them in a folder. To make life a little easier we set that folder as our working directory. That means for the duration of our R session we can reference files from this folder without stating the full path each time:

```
setwd("/Your/path/here/")
```

Hint: Windows users will have to use \\ instead of /

So, we read in the data from the directory with the readcsv command:

```
train <- read.csv("train.csv")
test <- read.csv("test.csv")
```

Hint: If you just want to mess around or are unsure about the syntax of complex prediction methods like randomForest, use only a few rows of the data until you are sure your script returns usable results. This will save you hours of your life. Use read.csv("train.csv", **nrows=1000**) instead of the above commands.

Now we have converted the csv files to R dataframes, which we can manipulate and interpret in R. See here for more info on data frames: http://www.r-tutor.com/r-introduction/data-frame

Doubleclick on the frames in the data tab or simply type View(train) to take a look at the dataframe.

Getting an overview

If you just want to write the script and get to the code, please skip this section.

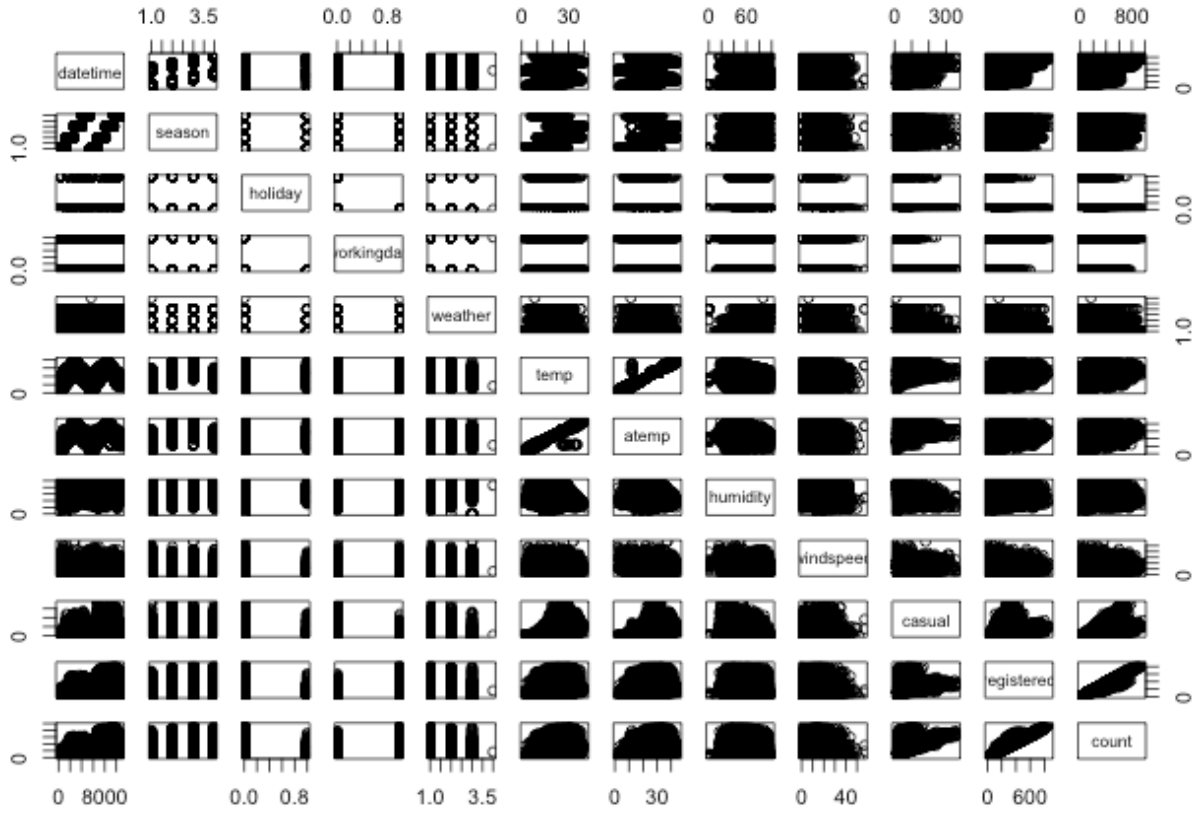
In the train set we find 12 variables. The columns count, casual and registered are our outcome variables. Count represents the total rentals from casual and registered users. This is the value we ultimately want to predict in the test data. In this tutorial we do this by making separate predictions for the casual and the registered rentals and adding them together.

The rest of the variables are so called predictors or predictor variables, which we want to use for predicting the outcome.

A good start to get a feel for data is the pairs command, which plots all the variables of a data set against each other.

```
pairs(train)
```

The result should look like this:



Hint: In R you can call the documentation for a command anytime by writing ?command (?pairs in our example)

The graphs in the bottom row plot the count variable on the y-axis against each of the other variables on the x-axis. At first glance we can see that there are some loose connections to the weather variables: Extreme weather seems to reduce the bike rentals. But no variable in the graph shows a nice linear connection to the count.

The only linear connections of count are seemingly with the casual and registered variables. But these are also outcomes and not predictors. So unfortunately we can't use them for our model.

Looks like we have to engineer some features ourselves …

Feature Engineering (with a short introduction to functions)

Feature Engineering – in short – means to transform the available data in a manner that represents the problem in a better way to the Machine Learning algorithm.

For a pretty good overview of the topic see this article:
http://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/

First things first: Always remember that any manipulation to the training data has to be done in the exact same manner to the test data. The reason is that the Machine Learning algorithm expects to find the same data structure used for learning when it makes predictions.

Instead of copypasting all our feature engineering commands we want to work efficient by using a function. If you are not familiar with functions: A function takes an input (e.g. a data frame), performs a set of operations and returns an output (e.g. a manipulated data frame). If you want to write efficient programs or scripts, using functions is the first imperative.

See here for more info on functions:
http://www.cs.utah.edu/~germain/PPS/Topics/functions.html

Our goal here is to have a function that performs the same operations on the train and the test data. So we initialize the function:

```
featureEngineer <- function(df) {

  return(df)
}
```

That function takes in a data frame and returns a data frame. Within the boundaries of the function the data frame will be called "df". The code we will write in this section is meant to be put into this function.

In a first step we do some routine work and convert variables with only a certain amount of possible values into factors so they can be easier processed by the Machine Learning algorithm.

```
names <- c("season", "holiday", "workingday", "weather")
df[,names] <- lapply(df[,names], factor)
```

One of kaggles rules is that we can only use the data provided. So we can't add any cool data from external sources. We will have to make do with what we got…

Surely the datetime variable holds some valuable information for us. After all there will probably be more bikes rented to commuting hours than at 2a.m. on a Tuesday morning. For similar reasons we are also interested in looking at rentals per the day of the week.

To extract the day of the week, we have to convert the datetime variable into the datatype POSIXlt, which can tell you the day of the week for any given date among other cool things:

```
df$datetime <- as.character(df$datetime)
df$datetime <- strptime(df$datetime, format="%Y-%m-%d %T", tz="EST")
```

Now we can parse the hour of day from the datetime variable and convert it into a new factor with 24 levels:

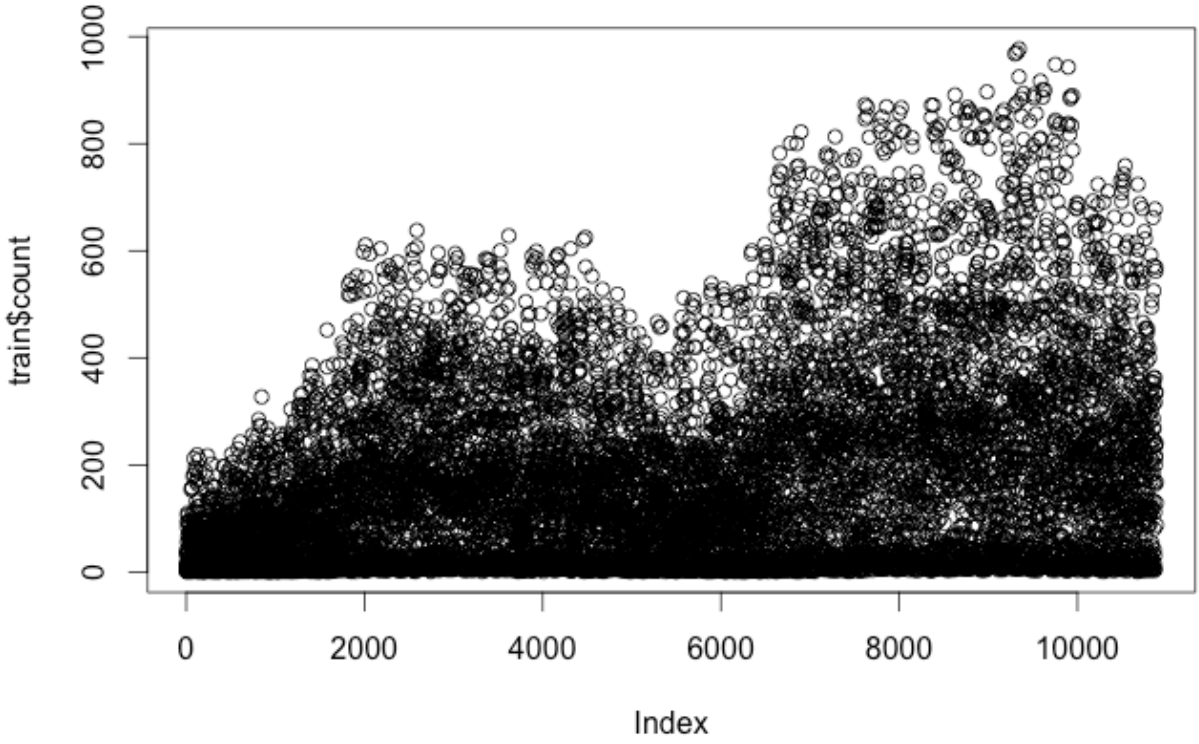
```
df$hour <- as.integer(substr(df$datetime, 12,13))
df$hour <- as.factor(df$hour)
```

To get the weekday for each date would be a tedious affair. Fortunately R can do this for us if we use the weekdays function, which can extract the day of the week from a POSIXlt object. For better readability we will re-name the 7 factor levels and order them:

```
df$weekday <- as.factor(weekdays(df$datetime))
 df$weekday <- factor(df$weekday, levels = c("Montag", "Dienstag", "Mittwoch",
"Donnerstag", "Freitag", "Samstag", "Sonntag"))
```

What else can we extract from the available data? Let's take a look how the number of rented bikes "looks" over the whole timeframe:

```
plot(train$count)
```



Hint: Using basic plots to get a feel for the data works well if you are a visual person like me. Other methods for this include using key metrics such as table summaries. Try and find something that you are comfortable with on an intuitive level.

What we see here (among other things like seasonal fluctuation) is that the overall trend heads up from 2011 to 2012. So the year is an important predictor for us. We parse it from the data like we did with the hour:

```
df$year <- as.integer(substr(df$datetime, 1,4))
df$year <- as.factor(df$year)
```

Our featureEngineer function is now complete. The last line of the function should be

```
return(df)
```

Which returns the modified data frame to the caller of the function.

Now we can transform our initial data frames by simply calling the featureEngineer function for each one and we are ready to go to the next section:

```
train <- featureEngineer(train)
test <- featureEngineer(test)
```

Hint: If you don't want to re-read the original csv files each time you change the function, simply assign new names to the modified train and test data. Like so: trainNew <- featureEngineer(train)

Using the randomForest Package

randomForest is a Machine Learning library for R. In short a Random Forest uses the given variables to build multiple decision trees (an "ensemble") and gives us back the mean prediction of the individual trees (a "fit"). We can then apply these predictions to a test data set.

Find more information on Random Forests in general here: http://en.wikipedia.org/wiki/Random_forest

The documentation for R's randomForest package can be found here: http://cran.r-project.org/web/packages/randomForest/randomForest.pdf

In order to use a Random Forest to predict the bike rental numbers we first need to install the necessary package in R, since this is a non-standard feature.

Go to the bottom right window, select the ribbon "packages" and click "Install". Type "random" and select "randomForest" from the autosuggest list. Make sure "install dependencies" is checked and leave all other options default. Hit "Install" and let R do the work. The success message will look something like "The downloaded binary packages are in …."

Now we need to load the package into our R session using the following command:

```
library(randomForest)
```

Hint: While R is a great tool by itself, its best "feature" is the huge ecosystem of external packages. An overview can be found here: http://cran.r-project.org/web/packages/available_packages_by_name.html

Before we start let's break apart the main function "randomForest". The documentations gives us the cryptic description that we should use the function like so:

```
randomForest(x, y=NULL,  xtest=NULL, ytest=NULL,
ntree=500,
             mtry=if (!is.null(y) && !is.factor(y))
             max(floor(ncol(x)/3), 1) else
floor(sqrt(ncol(x))),
             replace=TRUE, classwt=NULL, cutoff, strata,
             sampsize = if (replace) nrow(x) else
ceiling(.632*nrow(x)),
             nodesize = if (!is.null(y) && !is.factor(y))
5 else 1,
             maxnodes = NULL,
             importance=FALSE, localImp=FALSE, nPerm=1,
             proximity, oob.prox=proximity,
             norm.votes=TRUE, do.trace=FALSE,
             keep.forest=!is.null(y) && is.null(xtest),
corr.bias=FALSE,
             keep.inbag=FALSE, ...)
```

So, yeah… If you are new to all this and just want to produce something you can put on kaggle: Don't faint now! The above shows us the randomForest function we want to use to make a prediction with all its possible parameters. But for the practical purposes of this tutorial we only need very few of these parameters. All the other parameters are either optional or will be set to their default value if we don't call on them explicitly.

Interesting to us are the values ntree, mtry and importance. Since we make a separate fit for casual and registered we should control these parameters per separate variables. So we don't have to change everything twice when we toy around with the parameters.

The variable ntree sets how large your Random Forest is going to be. Or in other words, how many trees should be contained in your ensemble. We use 500 here to strike some balance between fitness and computation time.

```
myNtree = 500
```

The variable mtry controls the number of variables randomly sampled at each split. The random value here is one third of all the variables given to randomForest. In our example we use the value 5:

```
myMtry = 5
```

Hint: More trees and more variables don't always mean "better fit". But they *do* always mean a lot of additional computation time. Try and play around with the values but don't panic when your R session gets busy for 10 – 20 minutes. This stuff is intense!

Should importance of predictors be assessed? Yes, please, since we aren't so sure about that ourselves.

```
myImportance = TRUE
```

And at last we set a seed for the random sampling generator so we make our results comparable and not tainted by different random samples (Feel free to use any integer value you like here).

```
set.seed(415)
```

Now we can start making sense of the data using randomForest.

One last hint before we start: The execution of randomForest takes some real time and R should be left to itself while doing it. Else R will crash and you will get no result at all. As a rough benchmark: The casual and registered fit below take some 5-10 minutes on a Macbook Pro, i5, 8GB RAM. The test fits with all variables take 10-15 minutes.

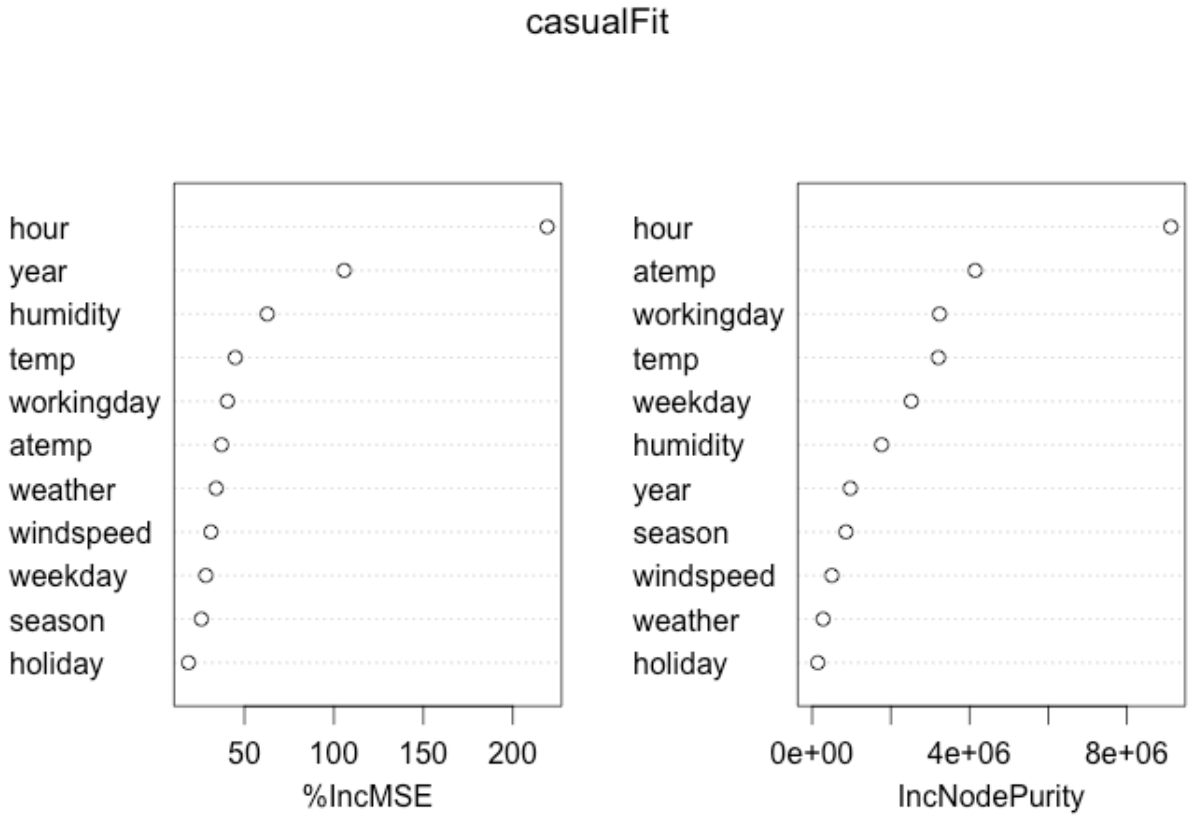
We start by building a random forest with all predictors available and let R plot us the importance of the different predictors. We start with the casual outcome. We remove datetime because it can't be used by randomForest. Also, unparsed this column is about as helpful a predictor as the row number would be (because it has as many values as the data set has rows). Besides, we already extracted all the necessary information from

datetime. Further we remove the columns registered and count since they aren't predictor variables. Then we create a random Forest for the whole dataset.

```
testTrain <- subset(train, select = -c(datetime, count, registered))
testFit <- randomForest(casual ~ ., data=testTrain, ntree=myNtree, mtry=myMtry, importance=myImportance)
```

Now we want to see how important each of the variables were for randomForest. We use the function varImpPlot to give us two scores: The %incMSE, which lists (in short) how much each variable improves the prediction of its tree compared to the exact same tree without the variable. The IncNodePurity compares the node purity of the trees with and without the variable.

The graph should look something like this:



Now we do the same with the registered users and the result should be a similar graph.

```
testTrain <- subset(train, select = -c(datetime, count, casual))
testFit <- randomForest(registered ~ ., data=testTrain, ntree=myNtree, mtry=myMtry, importance=myImportance)
```

Hint: I excluded the code for the testfits from the example R file. But you can just copypaste them into your script and execute them.

In an attempt to keep the number of predictors down we include only the most important ones per fit. An empirical observation of mine is that less predictors (but not too few!) lead to a better fit, which in turn leads to a better kaggle score. So for each fit,

casual and registered, we take the top 5-6 scores from each score from each graph and write them down. We end up with lists like this:

- **casual:** hour (x2), year, humidity (x2), temp (x2), atemp (x2), workingday (x2), weekday
- **registered:** hour (x2), year (x2), season (x2), weather, workingday (x2), humidity, weekday, atemp

And now it's time to finally get to the point and do some Machine Learning. These are our real fits:

```
casualFit <- randomForest(casual ~ hour + year + humidity + temp + atemp + workingday + weekday, data=train, ntree=myNtree, mtry=myMtry, importance=myImportance)
```

```
registeredFit <- randomForest(registered ~ hour + year + season + weather + workingday + humidity + weekday + atemp, data=train, ntree=myNtree, mtry=myMtry, importance=myImportance)
```

Hint: Execute these commands one at a time and wait until R is done with them. Each takes about 5-10 Minutes and you will crash R if you meddle with your session in between.

Our machine has learned and saved the results to randomForest.formula objects, which allow us to make predictions off them. We apply them to the test data via the generic predict function. R will then take the new data frame, process the variables according to the randomForest.formula and give out a result for each row of the new data frame. We save the results in a new column, which we create during the prediction.

```
test$casual <- predict(casualFit, test)
```

```
test$registered <- predict(registeredFit, test)
```

And finally we create a count column in the test data, which consists of the sum of the casual and the registered fit. And since you can't rent one tenth of a bike, we round this value to 0 digits:

```
test$count <- round(test$casual + test$registered, 0)
```

That's it.

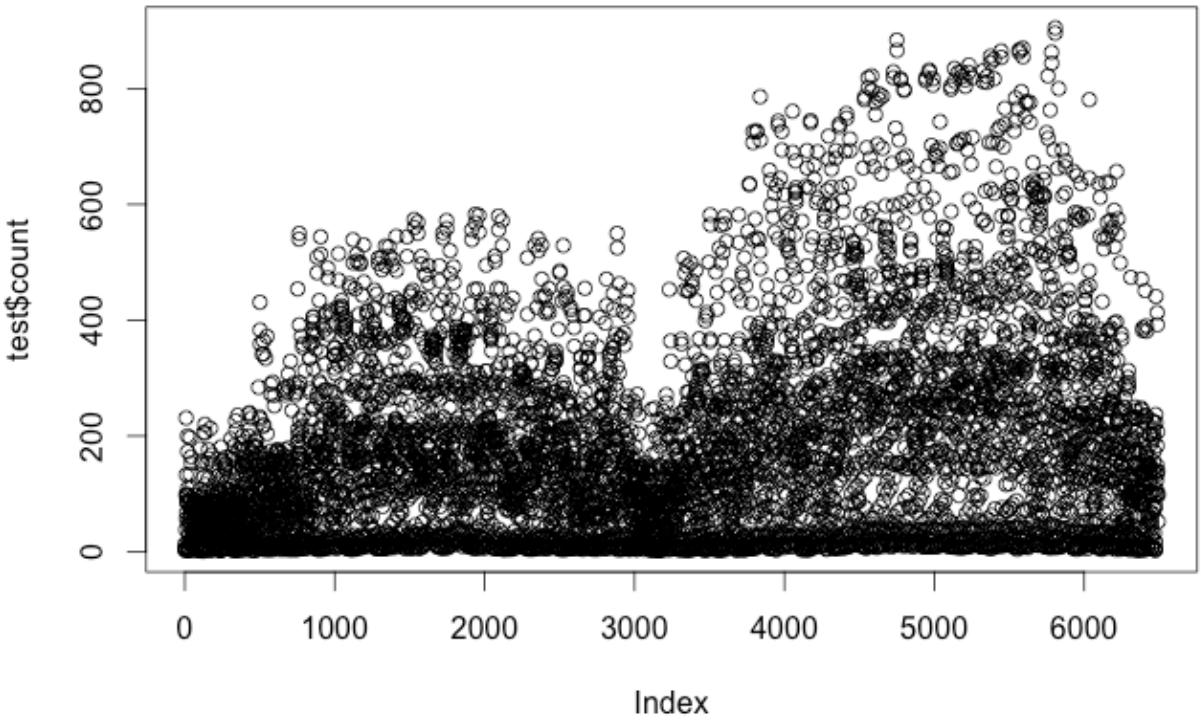
Validating our Prediction

Before we submit our data, we should take a look at our predictions and perform a quick check if they could result in a good score. We take the easiest route here by taking a simple and quick eyeball test.

Remember the `plot(train$count)` plot from above? If you skipped that section you should create that graph now. It shows the number of rentals throughout 2011 and 2012.

Since our predictions are made for the same timeframe (we predicted the last 10 days for each month of 2011 and 2012) we can make the exact same plot for our predicted rentals in the test data.

```
plot(test$count)
```



We can see the same basic structure as in the train data: The seasonal fluctuation, the dense population at the bottom of the y-axis, the thinner population at the top, and the ascending overall trend from 2011 to 2012. That looks pretty good!

Hint: There are many more advanced (and actually scientific) techniques to validate and grade out your predictions. At some point you should make yourself familiar with them. A couple of good pointers can be found here: http://www.quora.com/What-are-ways-to-prevent-over-fitting-your-training-set-data

Writing the output File

All that's left to do now is writing an output file that we can upload to kaggle. The constraints for this file are that it is csv format and contains only two values: The datetime and the predicted count.

I will make this short: Create a stripped down data frame from test data and write it in a csv file using write.csv. Since we have set the working directory in the beginning of our session, the csv file will be written into that directory:

```
submit <- data.frame (datetime = test$datetime, count = test$count)
write.csv(submit, file = "randomForest_Prediction.csv", row.names=FALSE)
```

Hit the “make a submission”-button on kaggle, upload the csv file and watch yourself climb the leaderboard.

Conclusion

There you go. A pretty good prediction in less than 70 lines of code (excluding the randomForest code :-P). I hope you had as much fun as I had writing this and got a good jumpstart into Machine Learning. Play a little with the variables and the parameters and see if you can improve on our score.

Download the R file under:
https://github.com/Bruschkov/kaggle_bikesharing/blob/master/randomForest_TUTORIAL.R

Feel free to re-use and copy this content without limits. But please give credit where it’s due.

Let me know if you have any feedback, corrections or just a general urge to mail somebody under jan.brusch@web.de