Intro to `Caret` with the Titanic data (Part I)

Preface

This tutorial guides you to build a practical machine learning model in R with the `caret` library. The focus is on the rationale rather than walking you through every detail of the process. For those who are interested in the details, some deeper topics with **RED** titles are also included. This tutorial is written by USC Marshall Masters of Business Analytics (MBSA) alumnus Sheng Ming (https://www.linkedin.com/in/ShengMing26) for Dr. Xin Tong's (https://sites.google.com/site/xintonghomepage/) DSO 530 class *Machine Learning Methods*.

Caveats:

1. This tutorial covers the whole picture and is divided into two parts. Part I focuses on data processing and part II focuses on modeling.
2. You'll see many new concepts, terminologies, and methods you haven't learned yet in DSO 530. If you find certain materials too difficult to understand, just skip them. A good portion of the materials is **NOT** mandatory by the class.
3. The Titanic data analysis I demonstrate is one of many possibilities to handle this data for prediction. I do not claim "optimality" in each implementation step. That means, I can only argue what I do make sense, but these arguments do not exclude other alternative means.

Acknowledgements

Most of the tutorial materials are adapted from a Youtube tutorial (https://www.youtube.com/watch?v=z8PRU46I3NY) and the official caret documentation (http://topepo.github.io/caret/index.html). Whenever you have a question, you can refer to the official documentation.

Introduction of `caret`

- `Caret` is an acronym for [**C**]lassification [**A**]nd [**RE**]gression [**T**]raining.
- Written and maintained by Max Kuhn, `caret` provides a consistent syntax for more than 200 models that are implemented by other machine learning R packages.
- As a wrapper package, `caret` focuses on streamlining the whole modeling process. If you want to learn or customize the details of your model, you might need to refer to more specialized packages such as `e1071`, `randomForest` and etc..

As its name suggests, `caret` is especially mighty in classification and regression modeling. However, if you are looking for solutions for an unsupervised problem, `caret` is still a good choice but you'll not see as much uplift as it has for supervised problems.

Caret tutorial with the Titanic example

In the following, we use the famous Titanic data. Please go to https://www.kaggle.com/c/titanic/data for description of the data. Basically, the task here is to predict whether a passenger on the Titanic survived based on a few features. This should be formulated as a binary classification problem. Let's download the `train.csv` and `test.csv` and put them into your working directory.

1. Data Processing

Believe it or not, in a typical machine learning project, you usually spend most of your time (may up to 70-80%) on collecting, retrieving, organizing and cleaning your data. Unfortunately, the fun parts that we are mostly interested in: the feature engineering and modeling itself, may just be a small portion of your work, and you'll also spend a good amount of time on productionizing and deploying your model.

For the Titanic data and most projects you've done so far, the data quality is fairly good and you don't need to worry about your data pipeline stuff. But please keep in mind this is NOT usual in the real life.

First, let's install the caret package (I commented the install command out because I've already installed the package):

```
#install.packages(caret)
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

1.1 Feature Pre-screening

Load the training data:

```
train = read.csv('train.csv', stringsAsFactors = FALSE)
# Take a look to the data
str(train)
```

```
## 'data.frame':    891 obs. of  12 variables:
##  $ PassengerId: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Survived   : int  0 1 1 1 0 0 0 0 1 1 ...
##  $ Pclass     : int  3 1 3 1 3 3 1 3 3 2 ...
##  $ Name       : chr  "Braund, Mr. Owen Harris" "Cumings, Mrs. John Bradley (Florence Briggs Thayer)"
##  $ Sex        : chr  "male" "female" "female" "female" ...
##  $ Age        : num  22 38 26 35 35 NA 54 2 27 14 ...
##  $ SibSp      : int  1 1 0 1 0 0 0 3 0 1 ...
##  $ Parch      : int  0 0 0 0 0 0 0 1 2 0 ...
##  $ Ticket     : chr  "A/5 21171" "PC 17599" "STON/O2. 3101282" "113803" ...
##  $ Fare       : num  7.25 71.28 7.92 53.1 8.05 ...
##  $ Cabin      : chr  "" "C85" "" "C123" ...
##  $ Embarked   : chr  "S" "C" "S" "S" ...
```

```
head(train, 20)
```

```
##    PassengerId Survived Pclass
## 1            1        0      3
## 2            2        1      1
## 3            3        1      3
## 4            4        1      1
## 5            5        0      3
## 6            6        0      3
## 7            7        0      1
## 8            8        0      3
## 9            9        1      3
## 10          10        1      2
## 11          11        1      3
## 12          12        1      1
```

```
## 13          13        0      3
## 14          14        0      3
## 15          15        0      3
## 16          16        1      2
## 17          17        0      3
## 18          18        1      2
## 19          19        0      3
## 20          20        1      3
##                                                     Name    Sex Age
## 1                                  Braund, Mr. Owen Harris   male  22
## 2      Cumings, Mrs. John Bradley (Florence Briggs Thayer) female  38
## 3                                   Heikkinen, Miss. Laina female  26
## 4             Futrelle, Mrs. Jacques Heath (Lily May Peel) female  35
## 5                                 Allen, Mr. William Henry   male  35
## 6                                         Moran, Mr. James   male  NA
## 7                                  McCarthy, Mr. Timothy J   male  54
## 8                           Palsson, Master. Gosta Leonard   male   2
## 9         Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg) female  27
## 10                      Nasser, Mrs. Nicholas (Adele Achem) female  14
## 11                          Sandstrom, Miss. Marguerite Rut female   4
## 12                                Bonnell, Miss. Elizabeth female  58
## 13                           Saundercock, Mr. William Henry   male  20
## 14                              Andersson, Mr. Anders Johan   male  39
## 15                     Vestrom, Miss. Hulda Amanda Adolfina female  14
## 16                         Hewlett, Mrs. (Mary D Kingcome)  female  55
## 17                                     Rice, Master. Eugene   male   2
## 18                                Williams, Mr. Charles Eugene   male  NA
## 19 Vander Planke, Mrs. Julius (Emelia Maria Vandemoortele) female  31
## 20                                  Masselmani, Mrs. Fatima female  NA
##    SibSp Parch           Ticket    Fare Cabin Embarked
## 1      1     0        A/5 21171  7.2500              S
## 2      1     0         PC 17599 71.2833   C85        C
## 3      0     0 STON/O2. 3101282  7.9250              S
## 4      1     0           113803 53.1000  C123        S
## 5      0     0           373450  8.0500              S
## 6      0     0           330877  8.4583              Q
## 7      0     0            17463 51.8625   E46        S
## 8      3     1           349909 21.0750              S
## 9      0     2           347742 11.1333              S
## 10     1     0           237736 30.0708              C
## 11     1     1          PP 9549 16.7000    G6        S
## 12     0     0           113783 26.5500  C103        S
## 13     0     0        A/5. 2151  8.0500              S
## 14     1     5           347082 31.2750              S
## 15     0     0           350406  7.8542              S
## 16     0     0           248706 16.0000              S
## 17     4     1           382652 29.1250              Q
## 18     0     0           244373 13.0000              S
## 19     1     0           345763 18.0000              S
## 20     0     0             2649  7.2250              C
```

OK, there are 891 rows and 12 variables. As we know, Survived is the class we need to predict. For all the other variables, are they all useful and should be put into the model? Apparently not. First, let's exclude the PassengerId since it just serves as the row index and we're not going to join any other tables based on it.

Cabin should also be removed as it's very messy and with too many missing values. What about Name and Ticket? It's possible for us to dig out some important information based on certain prefix of names or ticket numbers (i.e. Master. and Dr. may have higher survival rate). However, it's difficult to do so and the effort you spend on this may not be worthwhile. So let's keep it simple and remove these four columns.

```
colsExd = c('PassengerId', 'Cabin', 'Name', 'Ticket')
train.fltrd = train[, -which(colnames(train) %in% colsExd)]
head(train.fltrd)
```

```
##   Survived Pclass    Sex Age SibSp Parch    Fare Embarked
## 1        0      3   male  22     1     0  7.2500        S
## 2        1      1 female  38     1     0 71.2833        C
## 3        1      3 female  26     0     0  7.9250        S
## 4        1      1 female  35     1     0 53.1000        S
## 5        0      3   male  35     0     0  8.0500        S
## 6        0      3   male  NA     0     0  8.4583        Q
```

1.2 Checking the data quality

We see from the data that there are some 'NA' values in Age and remember that we saw a lot of blank cells in Cabin, there could also be blank values in other columns. Let's check both:

```
# Whether missing values exist
colSums(is.na(train.fltrd) > 0)
```

```
## Survived   Pclass      Sex      Age    SibSp    Parch     Fare Embarked
##        0        0        0      177        0        0        0        0
```

```
# Whether blank cells exist
colSums(train.fltrd == '', na.rm = T)
```

```
## Survived   Pclass      Sex      Age    SibSp    Parch     Fare Embarked
##        0        0        0        0        0        0        0        2
```

Notice that in some business practices, you might need to make a detailed data quality report for each variable (DQR, yeah you should have done this if you're taking DSO 562). But here I skip this step as it's not our focus.

Ok. Now we see 177 missing values in Sex and 2 blank values in Embarked. What should we do now? Shall we remove the entire rows? Obviously not. Since all the other variables look fine. Should we impute the missing values right now? Hold on a second. Some of the advanced imputing methods are very computational intensive, (i.e. we can build KNN or tree-based models to impute the missing values). But we might not want to do this right now because we don't even know whether we should include this column in our model! For example, if Age were highly correlated with Fare, we may want to eliminate the Age column (I know it's not a very good example here since apparently Age is a key factor).

1.3 Dealing with categorical variables

Now, let's perform dummy encoding to the categorical variables, that is, converting a categorical variable to a series of binary variables. Why doing this? First, most models (except tree-based models) require all predictors to be numeric; Second, dummy encoding could help to discover the interaction among the underlying distribution of variables. Third, dummy encoding enables the calculation of correlation so as to help detect multicolineary.

Note: in R you don't have to do dummy encoding for categorical variables for tree-based models (same for regression etc.). But if you're using `scikit-learn` in Python, it requires you to convert everything to numeric even for tree-based models.

Now let's learn the syntax in `caret` to do this. It's special in this machine learning package that it treats data transformation the way as it builds a 'model', it first 'trains' the parameters and then 'predicts' on the data you want to transform. You'll see such syntax throughout this tutorial. You may feel it counter-intuitive in the first place. The most important reason is this is more efficient. Imagine you want to normalize the data, now you just need to compute the mean for every column once in the training data, then every time new data comes in, you can directly use the computed means to transform the new records with the `predict` function.

Here the `dummyVars` function trains a "model" to create all those dummy variables. The "~." expression ensures that all the columns are transformed and the `dummyVars` function is smart enough to keep all the numeric columns untouched.

```
# Convert categorical predictors to factors first
train.fltrd$Pclass = as.factor(train.fltrd$Pclass)
train.fltrd$Sex = as.factor(train.fltrd$Sex)
train.fltrd$Embarked = as.factor(train.fltrd$Embarked)
# Set fullRank to be TRUE to avoid linear dependency, that is,
# if the categorical variable has n values, you just create n-1 dummy variables
dummy.vars = dummyVars(~ ., data = train.fltrd[, -1], fullRank = TRUE)
head(train.fltrd)
```

```
##   Survived Pclass    Sex Age SibSp Parch    Fare Embarked
## 1        0      3   male  22     1     0  7.2500        S
## 2        1      1 female  38     1     0 71.2833        C
## 3        1      3 female  26     0     0  7.9250        S
## 4        1      1 female  35     1     0 53.1000        S
## 5        0      3   male  35     0     0  8.0500        S
## 6        0      3   male  NA     0     0  8.4583        Q
```

```
train.dummy = cbind(Survived = train.fltrd$Survived,
                    predict(dummy.vars, newdata = train.fltrd))
head(train.dummy)
```

```
##   Survived Pclass.2 Pclass.3 Sex.male Age SibSp Parch    Fare Embarked.C
## 1        0        0        1        1  22     1     0  7.2500          0
## 2        1        0        0        0  38     1     0 71.2833          1
## 3        1        0        1        0  26     0     0  7.9250          0
## 4        1        0        0        0  35     1     0 53.1000          0
## 5        0        0        1        1  35     0     0  8.0500          0
## 6        0        0        1        1  NA     0     0  8.4583          0
##   Embarked.Q Embarked.S
## 1          0          1
## 2          0          0
## 3          0          1
## 4          0          1
## 5          0          1
## 6          1          0
```

Wait, it seems something is wrong here. If you check the data dictionary, there are only three classes for Embarked: C, Q, S. As we've set fullRank = TRUE, why are they all converted to dummy variables?

Remember we saw 2 blank values in Embarked in the result of Section 1.2? Apparently R regarded the blank value as another class. So let's impute the blank values.

```
# Look at the distribution of Embarked
table(train.fltrd$Embarked)
```

```
##
##       C   Q   S
```

```
##   2 168  77 644
```

We see that 'S' dominates this column. We could adopt the advanced technologies to model the missing values. But here there are just two missing values. For simplicity, we replace the blank cells with 'S'.

```
train.fltrd$Embarked[train.fltrd$Embarked == ''] = 'S'
# Although we've replaced the blank value, it still exists as a level of factor.
# Use the droplevels function to drop the unused level
train.fltrd$Embarked = droplevels(train.fltrd$Embarked)
# Let's do the dummy transformation again
dummy.vars = dummyVars(~ ., data = train.fltrd[, -1], fullRank = TRUE)
train.dummy = cbind(Survived = train.fltrd$Survived,
                    predict(dummy.vars, newdata = train.fltrd))
head(train.dummy)
```

```
##   Survived Pclass.2 Pclass.3 Sex.male Age SibSp Parch    Fare Embarked.Q
## 1        0        0        1        1  22     1     0  7.2500          0
## 2        1        0        0        0  38     1     0 71.2833          0
## 3        1        0        1        0  26     0     0  7.9250          0
## 4        1        0        0        0  35     1     0 53.1000          0
## 5        0        0        1        1  35     0     0  8.0500          0
## 6        0        0        1        1  NA     0     0  8.4583          1
##   Embarked.S
## 1          1
## 2          0
## 3          1
## 4          1
## 5          1
## 6          0
```

Categorical variables with too many levels?

What if some categorical variables contain too many levels, like zipcode? If we think we should keep this column in our model, then we might want to reduce the levels before we apply the dummy encoding; otherwise the feature dimension may explode. There is no one best way to do this and it may require some domain knowledge. Here is an article discussing such approaches: https://www.analyticsvidhya.com/blog/2015/11/easy-methods-deal-categorical-variables-predictive-modeling/

Categorical table encoding

Another alternative to deal with too many levels is to build a categorical table, that is, to assign a value directly associated with the dependent variable (i.e. the average of the dependent variable) for each possible category. After you encode your factors in this way, you then join the categorical table to the original data to append this calculated value as a new feature representing the categorical variable itself.

The pros of the categorical table encoding is that there is no dimensionality expansion as each categorical variable becomes one continuous variable, while the cons of doing so is the loss of interaction. For instance, red apples and green apples may have very different prices while that doesn't hold true for red grapes and green grapes. If we encode the attributes "color" and "type of fruit" with a categorical table, it will lose this interaction. So be **CAREFUL** to use the categorical table because sometimes it will harm your model performance a lot as you lose a great amount of information!

Below I demonstrate some simple dplyr and SQL code (the data.table package is much more efficient in data manipulation and I'll illustrate this in another tutorial) that generate the categorical table for the Embarked factor in the titanic data by calculating the average survival rate for each class of Embarked.

Note: For categorical table encoding, the right order is to implement this on the training portion of data after you do the training/test splitting because you are not supposed to know the true Y in testing. While if you want to re-train your model on the full training data after evaluation, you should calculate the categorical table again on all the data with true Y you have.

```
# Use dplyr to implement categorical table encoding
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
embarked = train.fltrd %>%
  select(Embarked, Survived) %>%
  group_by(Embarked) %>%
  summarise(Embarked.Survived = sum(Survived)/n())
embarked
```

```
## # A tibble: 3 x 2
##   Embarked Embarked.Survived
##     <fctr>             <dbl>
## 1        C         0.5535714
## 2        Q         0.3896104
## 3        S         0.3390093
```

```
train.ctgr = inner_join(train.fltrd, embarked)
```

```
## Joining, by = "Embarked"
```

```
head(train.ctgr)
```

```
##   Survived Pclass    Sex Age SibSp Parch    Fare Embarked
## 1        0      3   male  22     1     0  7.2500        S
## 2        1      1 female  38     1     0 71.2833        C
## 3        1      3 female  26     0     0  7.9250        S
## 4        1      1 female  35     1     0 53.1000        S
## 5        0      3   male  35     0     0  8.0500        S
## 6        0      3   male  NA     0     0  8.4583        Q
##   Embarked.Survived
## 1         0.3390093
## 2         0.5535714
## 3         0.3390093
## 4         0.3390093
## 5         0.3390093
## 6         0.3896104
```

```
# Use sqldf to implement categorical table encoding
library(sqldf)
```

```
## Loading required package: gsubfn
```

```
## Loading required package: proto
```

```
## Loading required package: RSQLite
```

```
## Loading required package: DBI
```

```
embarked = sqldf('
SELECT embarked, CAST(SUM(survived) AS real)/COUNT(survived) AS "Embarked.Survived"
FROM "train.fltrd"
GROUP BY embarked
')
```

```
## Loading required package: tcltk
```

```
embarked
```

```
##   Embarked Embarked.Survived
## 1        C         0.5535714
## 2        Q         0.3896104
## 3        S         0.3390093
```

```
train.ctgr = sqldf('
SELECT f.*, "Embarked.Survived"
FROM "train.fltrd" f LEFT JOIN embarked e
ON f.embarked = e.embarked
')
head(train.ctgr)
```

```
##   Survived Pclass    Sex Age SibSp Parch    Fare Embarked
## 1        0      3   male  22     1     0  7.2500        S
## 2        1      1 female  38     1     0 71.2833        C
## 3        1      3 female  26     0     0  7.9250        S
## 4        1      1 female  35     1     0 53.1000        S
## 5        0      3   male  35     0     0  8.0500        S
## 6        0      3   male  NA     0     0  8.4583        Q
##   Embarked.Survived
## 1         0.3390093
## 2         0.5535714
## 3         0.3390093
## 4         0.3390093
## 5         0.3390093
## 6         0.3896104
```

Removing zero-variance or near zero-variance variables

In some situations, the data generating mechanism can create predictors that only have a single unique value (zero-variance predictor) or similarly predictors that have only a handful of unique values that occur with very low frequencies. Recall that in a previous DSO 530 test, a quesion asked about whether a `gender` variable that only takes "male" value in the dataset is useful as a predictor. For another example:

```
data(mdrr)
data.frame(table(mdrrDescr$nR11))
```

```
##   Var1 Freq
## 1    0  501
## 2    1    4
## 3    2   23
```

The problem is these predictors may become zero-variance predictors when the data are split into cross-validation/bootstrap sub-samples or that a few samples may have an undue influence on the model. So

these predictors may be considered to be eliminated before modeling. See detail explanation at http://topepo.github.io/caret/pre-processing.html. But we should have a word of caution here. Some near-zero variance variables might be actually very useful when combining with other predictors, so whether you really want to get rid of them is a case by case study.

```
#nzv = nearZeroVar(train, saveMetrics = TRUE)
nzv = nearZeroVar(train.dummy, saveMetrics = TRUE)
nzv
```

```
##              freqRatio percentUnique zeroVar   nzv
## Survived      1.605263     0.2244669   FALSE FALSE
## Pclass.2      3.842391     0.2244669   FALSE FALSE
## Pclass.3      1.227500     0.2244669   FALSE FALSE
## Sex.male      1.837580     0.2244669   FALSE FALSE
## Age           1.111111     9.8765432   FALSE FALSE
## SibSp         2.909091     0.7856341   FALSE FALSE
## Parch         5.745763     0.7856341   FALSE FALSE
## Fare          1.023810    27.8338945   FALSE FALSE
## Embarked.Q   10.571429     0.2244669   FALSE FALSE
## Embarked.S    2.636735     0.2244669   FALSE FALSE
```

Should we do this step before or after dummy encoding? You can do it in either way. Personally I prefer to do this after dummy encoding.

1.4 Identifying correlated predictors

The next step is to identify correlated predictors. Linear models like the linear regression are extremely sensitive to those correlated predictors, the multicolinearity will invalidate your coefficient estimates. On the other hand, tree-based models like random forest and boosted trees are more robust to multicolinearity. Note that, when all you care about is prediction (as opposed to inference), identifying correlated predictors are not that important.

The `findCorrelation` in caret is one of the many functions in R to automatically handles the collinearity elimination. Here you only need to input a threshold of correlation to eliminate from.

```
# Set use = 'complete.obs' to deal with the missing values
colCor = cor(train.dummy[, -1], use = 'complete.obs')
highCorCols = findCorrelation(colCor, cutoff = 0.75)
if (length(highCorCols) > 0) {
  train.dummy = train.dummy[, -which(colnames(train.dummy) %in% colnames(colCor)[highCorCols])]
}
```

1.5 Data imputing

Now we have finished the feature pre-screening part. The next step, as mentioned above, is to replace the missing values in Age. This is what we called data imputing or interpolation. There are a variety of methods to approach this from naively imputing the mean to fitting a seperate model to predict.

Fotunately, the function `preProcess` can handle everything for us. This function can perform various types of data transformation (boxcox, scale, pca, etc.). It provides three built-in imputing methods: medianImpute, knnImpute and bagImpute, each one is more flexible than the previous one but incurs higher computational cost. Imputation via bagging fits a bagged tree model for each predictor (as a function of all the others). Since the size of Titanic data is very small, we may want to try out this most expensive method.

Note: the imputing methods in `preProcess` require all predictors to be numeric. If you don't want to do the dummy transformation and want to keep all the categorical variables as factors, you can create a copy of

the data, do the dummy transformation and imputing, then replace the corresponding column in the original data frame with the new imputed column.

```
set.seed(1234)
# Exclude Survived as we should only include all predictors for imputing
pre.impute = preProcess(train.dummy[, -1], method = "bagImpute")
train.imputed = predict(pre.impute, train.dummy)
head(train.imputed)
```

```
##   Survived Pclass.2 Pclass.3 Sex.male      Age SibSp Parch    Fare
## 1        0        0        1        1 22.00000     1     0  7.2500
## 2        1        0        0        0 38.00000     1     0 71.2833
## 3        1        0        1        0 26.00000     0     0  7.9250
## 4        1        0        0        0 35.00000     1     0 53.1000
## 5        0        0        1        1 35.00000     0     0  8.0500
## 6        0        0        1        1 27.26783     0     0  8.4583
##   Embarked.Q Embarked.S
## 1          0          1
## 2          0          0
## 3          0          1
## 4          0          1
## 5          0          1
## 6          1          0
```

1.6 Feature engineering

Congratulations! You've completed the most boring but non-trivial part of data processing. Now let's explore a little bit of the feature engineering part. This is actually my favorite part of modeling as it allows you to apply domain knowledge and business savvy to do something creative. Unfortunately for the Titanic data, there is not much room for feature engineering since the data is quite straight-forward. But there is always something you can do about the features. You see Sibsp is the # of siblings / spouses aboard the Titanic and Parch is the # of parents / children aboard the Titanic. How about to build a new variable which is the sum of these two variables and indicating the family size? Let's do this.

One thing to notice here is both `dummyVars` and `preProcess` would convert the data.frame to a matrix for computation, hence we need to convert the data back to a data.frame to alter the features.

```
train.imputed = as.data.frame(train.imputed)
train.imputed$FamilySize = 1 + train.imputed$SibSp + train.imputed$Parch
```

1.7 Standardizing

The final step of the data preprocessing part is to standardize (or normalize) all features to the same scale. This common practice is preferred in most modeling cases especially when the data ranges vary a lot from column to column or you want to build your model on some distance measures (i.e. knn). The drawback for scaling the data is it loses some interpretability. For example, after the scaling below, the dummy variables are not 0 or 1 any more, but become the measures of standard deviation. How would you explain this to your boss with no statistical background?

```
pre.scale = preProcess(train.imputed[, -1], method = c("center", "scale"))
train.scaled = predict(pre.scale, train.imputed)
head(train.scaled)
```

```
##   Survived   Pclass.2   Pclass.3  Sex.male        Age      SibSp
## 1        0 -0.5098652  0.9020807  0.737281 -0.5566902  0.4325504
```

```
## 2        1 -0.5098652 -1.1073041 -1.354813  0.6318693  0.4325504
## 3        1 -0.5098652  0.9020807 -1.354813 -0.2595503 -0.4742788
## 4        1 -0.5098652 -1.1073041 -1.354813  0.4090144  0.4325504
## 5        0 -0.5098652  0.9020807  0.737281  0.4090144 -0.4742788
## 6        0 -0.5098652  0.9020807  0.737281 -0.1653696 -0.4742788
##        Parch       Fare Embarked.Q Embarked.S  FamilySize
## 1 -0.4734077 -0.5021631 -0.3073897  0.6154927  0.05912667
## 2 -0.4734077  0.7864036 -0.3073897 -1.6228911  0.05912667
## 3 -0.4734077 -0.4885799 -0.3073897  0.6154927 -0.56065994
## 4 -0.4734077  0.4204941 -0.3073897  0.6154927  0.05912667
## 5 -0.4734077 -0.4860644 -0.3073897  0.6154927 -0.56065994
## 6 -0.4734077 -0.4778481  3.2495483 -1.6228911 -0.56065994
```

Whether should I normalize the dummies?

Some of you may feel uncomfortable to scale the binary dummy variables. You will find different opinions for this question if you search it online. Here is one of the discussion https://stats.stackexchange.com/questions/69568/whether-to-rescale-indicator-binary-dummy-predictors-for-lasso.

2. Split the data

Just one more step before we could officially start the modeling part. Remember that all the manipulation we've done so far is based on the train.csv data. We'll need to split it into two parts: the training set and the test set. In part 2 of the tutorial, we'll train different models and perform cross-validation on the training set and evaluate the model performance on the test set. Note that the `test.csv` file you downloaded can only be used for submission to the Kaggle platform, but cannot be used to evaluate model performance, because the instances in this file do not have the survival status labels.

One important thing is the distribution of the objective class (a.k.a, response, outcomes, Y variable etc.) could be imbalanced. For example, what if more than 90% of passengers died? To have a representative distribution of the class in both training and test sets, we could perform a stratified sampling by specifying train$Survived in the `createDataPartition` function.

Note: here I split the data into two parts: the training set and the test set and cross-validations will be performed in the training set in the future modeling process. If you don't want to do any cross-validation, it's better to split the data to three parts: training, validation and test (i.e. 60/20/20%) by calling the `createDataPartition` function twice.

```
set.seed(4321)
train.scaled$Survived = as.factor(train.scaled$Survived)
indexes = createDataPartition(train.scaled$Survived,
                              times = 1,
                              # We use a common 70/30% partition but
                              # there is no universally agreed split ratio
                              p = 0.7,
                              list = FALSE)
titanic.train = train.scaled[indexes,]
titanic.test = train.scaled[-indexes,]
```

Examine the proportions of the Survived class label across the datasets

```
prop.table(table(train.scaled$Survived))
```

```
##
##         0         1
```

```
## 0.6161616 0.3838384
```

```
prop.table(table(titanic.train$Survived))
```

```
##
##     0     1
## 0.616 0.384
```

```
prop.table(table(titanic.test$Survived))
```

```
##
##         0         1
## 0.6165414 0.3834586
```

We see both the training set and test set have consistent distribution of the class Survived.

In part 2 of the tutorial, I'll introduce how you can do feature selection, model fitting, hyperparameter tuning and performance evaluation using caret. Now let's saved the trained models and the processed data for future uses.

```
# save the trained models
saveRDS(dummy.vars, "dummies.rds")
saveRDS(pre.scale, "scale.rds")
saveRDS(pre.impute, "impute.rds")

# save the processed data
write.csv(train.scaled, 'train_complete.csv', row.names = FALSE)
write.csv(titanic.train, 'titanic_train.csv', row.names = FALSE)
write.csv(titanic.test, 'titanic_test.csv', row.names = FALSE)
```