Splitting the Sample

Contents

The dataset may be split into a train and test sample using random approaches or non-random approaches. The latter are used in very specific situations, hence are not discussed here. For more on non-random approaches see, Chapter 4, p.67 in Applied Predictive Modeling, by Max Kuhn and Kjell Johnson. There are two approaches to random sampling:

- Simple Random Sampling: Each observation has an equal likelihood of getting picked.
- Stratified Sampling: Simple random sampling is done on subgroups or strata. In predictive modeling situations, stratified sampling is used to ensure similar distribution of the outcome variable in train and test samples.

Data

To demonstrate these two approaches to splitting the sample, we will use the diamonds dataset that ships with ggplot2. We will split this data into a train and test set in the ratio, 70:30, so that 70% of the sample will go to the train sample.

```r
library(ggplot2)
head(diamonds)
```

```
## # A tibble: 6 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 0.23  Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
## 2 0.21  Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
## 3 0.23  Good      E     VS1      56.9    65   327  4.05  4.07  2.31
## 4 0.290 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
## 5 0.31  Good      J     SI2      63.3    58   335  4.34  4.35  2.75
## 6 0.24  Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

Simple Random Sampling

Set the seed to 61710. Choice of seed is arbitrary but is very important as it ensures the random split can be replicated. R has a memory of one line for the set.seed() function, so seed must be set right before the step that runs random sampling function.

Before, we actually split the data, we generate a random sample of row indices from a vector containing all the rows in diamonds `1:nrow(diamonds)`. We get 70% of the row indices by specifying a size that is 70% of the number of rows in diamonds: `0.7*nrow(diamonds)`. Take the individual pieces apart to convince yourself of the underlying process. Also, note by default sample() does sampling without replacement which is what we want.

```
set.seed(61710)
split = sample(x = 1:nrow(diamonds),size = 0.7*nrow(diamonds))
split[1:10]
```

```
##  [1] 40257 48822 14319 28974 15191 23114 38585 50147 35738 40618
```

The split vector contains approximately 70% of the numbers in `1:nrow(diamonds)`

```
length(1:nrow(diamonds))
```

```
## [1] 53940
```

```
length(split)
```

```
## [1] 37758
```

Next, we subset the rows of `diamonds` that have an index matching in split.

```
train = diamonds[split,]
```

Next, we use a clever R trick to create a test sample by *not* including the rows in split. Here, `-split` will not include any row indices that are contained in split.

```
test = diamonds[-split,]
```

We can confirm it all worked well by checking the number of rows in the train and test sets. They must sum to the number of rows in the original dataset, diamonds.

```
nrow(train)
```

```
## [1] 37758
```

```
nrow(test)
```

```
## [1] 16182
```

```
nrow(diamonds)
```

```
## [1] 53940
```

Lastly, we can check to see if the variables in the two samples are similar. Random sampling is going to generate similar samples but they are unlikely to be identical. So, let's check

```
mean(train$price);mean(test$price)
```

```
## [1] 3944.739
```

```
## [1] 3904.942
```

Stratified Random Sampling (with a numeric outcome)

When using data for prediction, it is important that the train and test samples be similar but even more important for the outcome variable to have a similar distribution. For this reason, it is common to sample in such a manner that the outcome variable is approximately equal across train and test samples. We will make use of `createDataPartition()` from the caret package.

As was the case above, we set a seed and run it immediately before the sampling function. Assuming the goal here is to predict price, we setup the sample so that price is kept approximately equal across samples. Since a numeric variable doesn't have natural strata, it is divided into groups based on percentiles. Here, we are using groups=50. We are keeping 70% observations in the train sample.

```
library(caret)
set.seed(61710)
split = createDataPartition(y = diamonds$price, p = 0.7, list = F, groups = 50)
```

Once we get the split vector, we proceed in the same manner as for random sampling.

```
train = diamonds[split,]
test = diamonds[-split,]
```

Verify that the split divided up the diamonds in approximately 70:30 ratio.

```
nrow(train)
```

```
## [1] 37777
```

```
nrow(test)
```

```
## [1] 16163
```

```
nrow(diamonds)
```

```
## [1] 53940
```

Compare the outcome variable, price, across the samples. How does this compare to simple random sampling.

```
mean(train$price);
```

```
## [1] 3932.281
```

```
mean(test$price)
```

```
## [1] 3934.012
```

Stratified Random Sampling (with a categorical outcome)

A categorical outcome (e.g., email response) has strata that are defined by the levels of the variable (e.g., for email response: respond and not respond). In this case, simple random samples will be drawn from each strata and then combined.

First, let's create a categorical outcome from the diamonds dataset. The new variable, price_hilo has two levels, high and low.

```
diamonds$price_hilo = ifelse(diamonds$price>mean(diamonds$price),'High','Low')
head(diamonds)
```

```
## # A tibble: 6 x 11
##   carat cut    color clarity depth table price     x     y     z price_hilo
##   <dbl> <ord>  <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <chr>
## 1 0.23  Ideal  E     SI2      61.5    55   326  3.95  3.98  2.43 Low
## 2 0.21  Premi~ E     SI1      59.8    61   326  3.89  3.84  2.31 Low
## 3 0.23  Good   E     VS1      56.9    65   327  4.05  4.07  2.31 Low
## 4 0.290 Premi~ I     VS2      62.4    58   334  4.2   4.23  2.63 Low
## 5 0.31  Good   J     SI2      63.3    58   335  4.34  4.35  2.75 Low
## 6 0.24  Very ~ J     VVS2     62.8    57   336  3.94  3.96  2.48 Low
```

We are now, going to conduct stratified random sampling to ensure the breakdown of high and low in price_hilo is the same across train and test samples. Set seed and run createDataPartition with price_hilo.

```
set.seed(61710)
split = createDataPartition(y = diamonds$price_hilo,p = 0.7,list = F)
```

Create train and test samples using split.

```
train = diamonds[split,]
test = diamonds[-split,]
```

Verify that the split divided up the diamonds in approximately 70:30 ratio.

```
nrow(train)
```

```
## [1] 37759
```

```
nrow(test)
```

```
## [1] 16181
```

```
nrow(diamonds)
```

```
## [1] 53940
```

Since outcome is a categorical variable, we look at counts rather than average.

```
table(train$price_hilo)
```

```
##
##  High   Low
## 13760 23999
```

```
table(test$price_hilo)
```

```
##
##  High   Low
##  5897 10284
```

To compare, proportion of high and low prices in each sample, we examine proportions.

```
prop.table(rbind(train = table(train$price_hilo),
      test = table(test$price_hilo)),
      margin = 1)
```

```
##            High       Low
## train 0.3644164 0.6355836
## test  0.3644398 0.6355602
```

Stratified Random Sampling (with a categorical outcome) using caTools

Another package that can accomplish stratified sampling when the outcome variable is categorical is caTools. The process is quite similar. We are doing a 70:30 split, stratifying by price_hilo.

```
library(caTools)
set.seed(61710)
split = sample.split(Y = diamonds$price_hilo, SplitRatio = 0.7)
```

The key difference of sample.split() from previous sampling functions is that it generates a logical, *not* a vector of numbers.

```
table(split)
```

```
## split
## FALSE  TRUE
## 16182 37758
```

Since sample.split() generates a logical, to subset, we will have to make a subtle but important change. Rather than using `-`, we will using `!` operator for the test sample.

```
train = diamonds[split,]
test = diamonds[!split,]
```

```
nrow(train)
```

```
## [1] 37758
```

```r
nrow(test)
```

```
## [1] 16182
```

Since outcome is a categorical variable, we look at counts rather than average.

```r
table(train$price_hilo)
```

```
##
##  High   Low
## 13760 23998
```

```r
table(test$price_hilo)
```

```
##
##  High   Low
##  5897 10285
```

To compare, proportion of high and low prices in each sample, we examine proportions.

```r
prop.table(rbind(train = table(train$price_hilo),
      test = table(test$price_hilo)),
      margin = 1)
```

```
##            High       Low
## train 0.3644261 0.6355739
## test  0.3644173 0.6355827
```

Compare the results of createDataPartition() to sample.split(). You will note, the results are similar but not identical.

In Conclusion

For simple random sampling, use `sample()`. For stratified sampling with a numeric outcome variable, use `caret::createDataPartition`. For stratified sampling with a categorical outcome, use either `caret::createDataPartition()` or `caTools::sample.split()`.