An introduction to the Caret package

AUTHOR: Francesco Vetere

E-MAIL: francesco.vetere@studenti.unipr.it

Contents

1 Introduction

Caret (short for **C**lassification **A**nd **RE**gression **T**raining) is a comprehensive framework for building machine learning models in R.

R is a language and environment for statistical computing and graphics, widely used in AI and ML applications.
It is open-source, provides many statistical techniques (such as statistical tests, classification, clustering, ...), and has many packages that can be used to solve different problems.

Sometimes the syntax and the way to implement ML algorithms differ across packages: **Caret** is an R package that provides a uniform interface to the various existing modeling functions.
In particular, it offers tools for data splitting, data pre-processing, model creation and tuning, and many more.

The aim of this paper is to introduce the developer to the Caret package: starting from the installation process, it will be then shown how to use Caret to build machine learning models, focusing in particular on Multi-Layer Perceptron.
Finally, some practical examples will be presented.

2 Installation and prerequisites

R is a multi-platform environment, available for various OS.
In this paper, the installation process will be shown for Ubuntu 20.04 LTS
(however, it is very similar for other Linux distros).

Precompiled binaries of R are available for various OS on **CRAN** (`https://cloud.r-project.org/`),
a network of ftp and web servers around the world that store identical, up-to-
date, versions of code and documentation for R.

However, with Ubuntu it is also possible to install it directly from terminal with
`sudo`, or as a root user:

```
# update indices
apt update -qq

# install two helper packages we need
apt install --no-install-recommends software-properties-common dirmngr

# import the signing key (by Michael Rutter) for these repo
apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
    E298A3A825C0D65DFD57CBB651716619E084DAB9

# add the R 4.0 repo from CRAN -- adjust 'focal' to 'groovy' or 'bionic' as
    needed
add-apt-repository "deb https://cloud.r-project.org/bin/linux/ubuntu $(
    lsb_release -cs)-cran40/"

# install R and its dependencies
apt install --no-install-recommends r-base
```

Now, R and its dependecies are installed: typing the command `R` in the terminal
will launch the R interpreter.

In order to write R code, a very popular IDE is **RStudio**, available for many
platforms (`https://www.rstudio.com/products/rstudio/download/`).

Once the correct version of the software has been chosen, a `.deb` package will
be downloaded: it is useful to install it with `gdebi`, a command that will ensure
that all additional prerequisites for RStudio are fullfilled (such as `clang` and
others).

```
# install RStudio and its dependencies
gdebi rstudio-1.4.1717-amd64.deb
```

In order to install the Caret package, it's sufficient to open RStudio (or directly the R interpeter) and execute the following line of code:

```
install.packages('caret', dependencies = TRUE)
```

Caret will be downloaded from CRAN, together with its dependencies.

Once the process is ended, Caret is ready to be included in a normal R program with the following directive:

```
library('caret')
```

3 Core elements of Caret

3.1 Data preprocessing

3.1.1 createDataPartition()

Once a dataset has been loaded with standard R functions (i.e.: `read.csv()`), the first step is to split it into a training set and a test set.
(Note: when we talk about test set, in this circumstance, we are always referring to it as a validation set).

In order to do this, Caret offers a handy `createDataPartition` function.

Syntax

```
createDataPartition(
  y,
  times = 1,
  p = 0.5,
  list = TRUE
)
```

Arguments

y: a vector of outcomes
`times`: the number of partitions to create
p: the percentage of data that goes to training
`list`: TRUE if the results should be in a list;
     FALSE if the results should be in a matrix of dimensions
     `[floor(p * length(y)), times]`

Return value

A list or matrix of row position integers corresponding to the training data.

Example

```
# Load the caret package
library(caret)

# Import dataset
dataset <- read.csv('../datasets/iris-dataset.csv')

# Create the training and test datasets
set.seed(100)

# Step 1: Get row numbers for the training data
trainRowNumbers <- createDataPartition(dataset$Species, p=0.8, list=FALSE)

# Step 2: Create the training dataset
trainData <- dataset[trainRowNumbers,]

# Step 3: Create the test dataset
testData <- dataset[-trainRowNumbers,]

print(nrow(dataset))   # 150
print(nrow(trainData)) # 120
print(nrow(testData))  # 30
```

3.1.2 preProcess()

Often some kind of data transformation can be useful (i.e.: normalization).
Caret make this process easy, providing the `preProcess` function.

Syntax

```
preProcess(
  x,
  method = c("center", "scale")
  ...
)
```

Arguments

`x`: a matrix or data frame
`method`: a character vector specifying the type of processing.
      Possible values are:
      "range": Normalize values so it ranges between 0 and 1
      "center": Subtract Mean
      "scale": Divide by standard deviation
      and many more...
...

Return value

A list of various statistics, which will be transormed in the desired data frame calling the `predict` function, as shown in the example below.

Example

```
print("before\n")
print(head(dataset))

# We want to normalize our dataset
preProcess_range_model <- preProcess(dataset, method='range')
dataset <- predict(preProcess_range_model, newdata = dataset)

print("after\n")
print(head(dataset))
```

3.2   Model training and tuning

3.2.1   train()

Once data are ready, the next step is to build the machine learning model, choosing its hyperparameters and its training control strategies.

Caret provides a huge list of possible models, currently 238. (https://topepo.github.io/caret/available-models.html).

Each model may be implemented using a different backend library, but Caret's interface remains the same (that's the big advantage).

In particular, Caret provides a `train` function, that performs all the necessary work in order to train and tune a specific model, that can be used transparently both for regression and classification tasks.

Syntax

```
train(form,
      data,
      method='rf',
      trControl=trainControl(),
      tuneGrid=NULL,
      ...
     )
```

Arguments

`form`: A formula of the form $y \sim x1 + x2 + \ldots$ for dividing outcome from predictors

`data`: Data frame from which variables specified in `form` are taken.

`method`: A string specifying which classification or regression model to use.
Possible values are found using `names(getModelInfo())`

`trControl`: A function defining the training control strategy.

`tuneGrid`: A data frame with possible tuning values for the chosen `method`.

...

Return value

A list which describes the model.

Example

```
# Train using a neural network (SLP) with 3 neurons, no weight decay,
# and K-fold cross-validation (K=10) as training control
model <- train(form=Species ~ .,   # outcome ~ predictors
               data=trainData,
               method='nnet',
               trControl=trainControl(method="repeatedcv",number=10,repeats=10),
               tuneGrid=expand.grid(size=3,decay=0),
               trace=FALSE)          # avoids verbose output
```

3.3   Predictions and evaluations

3.3.1   predict()

When the model has been created, trained and tuned, we can start predict new outcomes from test data.
This is done by `predict` function.

Syntax

```
predict(
  object,
  newdata = NULL,
  type = "raw",
  ...
)
```

Arguments

`object`: A model created with `train`

`newdata`: An optional set of data to predict on.
If NULL, then the original training data are used.

`type`: either "raw" or "prob", for the number/class predictions
or class probabilities, respectively.
Class probabilities are not available for all classification models.

...

Return value

A vector of predictions if `type` = "raw", or a data frame of class probabilities for `type` = "prob".

Example

```
# Predict "Species" attribute for data in testData using the model trained before
prediction <- predict(model, testData[-5])
```

3.3.2 confusionMatrix()

Once the predictions are made, it is possible to compare the predictions versus the actual data, generating also some evaluation metrics.
This is done by `confusionMatrix` function.

Syntax

```
confusionMatrix(
  data,
  reference,
  mode = "sens_spec",
  ...
)
```

Arguments

`data`: A factor of predicted classes.
`reference`: A factor of classes to be used as the true results.
`mode`: A single character string either "sens_spec", "prec_recall",
      or "everything"
...

Return value

A list containing the table representing the confusion matrix and the various evaluation metrics selected with `mode`.

Example

```
# Create a confusion matrix with all possible evaluation metrics
cm <- confusionMatrix(reference = as.factor(testData$Species),
      data = prediction, mode='everything')
print(cm)
```

4 Examples of MLPs

In this final section, some complete examples of MLPs are presented.

4.1 SLP

In this first example, the objective is to perform a classification task over the iris dataset, using a Single Layer Perceptron with 3 neurons.

```
# Load the caret package
library(caret)

# Import dataset
dataset <- read.csv('../datasets/iris-dataset.csv')

set.seed(100)

# Split data
trainRowNumbers <- createDataPartition(dataset$Species, p=0.8, list=FALSE)
trainData <- dataset[trainRowNumbers,]
testData <- dataset[-trainRowNumbers,]

# Train using a neural network (SLP) with 3 neurons, no weight decay,
# and K-fold cross-validation (K=10) as training control
model <- train(form=Species ~ .,   # outcome ~ predictors
               data=trainData,
               method='nnet',
               trControl=trainControl(method="repeatedcv", number=10, repeats=10),
               tuneGrid=expand.grid(size=3, decay=0),
               trace=FALSE)   # avoids verbose output

# Predict "Species" attribute for data in testData using the model trained before
prediction <- predict(model, testData[-5])
print(prediction) # 30 predictions are made, because nrow(testData) = 30

# Create a confusion matrix with all possible evaluation metrics
cm <- confusionMatrix(reference = as.factor(testData$Species), data = prediction,
      mode='everything')
print(cm)
```

This is the output generated, from which we can observe that the 30 test instances have been classified with an accuracy of 96.67%.

```
[1] Iris-setosa    Iris-setosa    Iris-setosa    Iris-setosa    Iris-setosa    Iris
    -setosa
[7] Iris-setosa    Iris-setosa    Iris-setosa    Iris-setosa    Iris-versicolor
    Iris-versicolor
[13] Iris-versicolor Iris-versicolor Iris-versicolor Iris-versicolor Iris-
    versicolor Iris-virginica
[19] Iris-versicolor Iris-versicolor Iris-virginica Iris-virginica Iris-virginica
     Iris-virginica
[25] Iris-virginica Iris-virginica Iris-virginica Iris-virginica Iris-virginica
    Iris-virginica

Levels: Iris-setosa Iris-versicolor Iris-virginica

Confusion Matrix and Statistics

              Reference
Prediction       Iris-setosa Iris-versicolor Iris-virginica
  Iris-setosa           10               0               0
  Iris-versicolor        0               9               0
  Iris-virginica         0               1              10

Overall Statistics

           Accuracy : 0.9667
           ...

Statistics by Class:

                 Class: Iris-setosa Class: Iris-versicolor Class: Iris-
                     virginica
Sensitivity                1.0000              0.9000              1.0000
Specificity                1.0000              1.0000              0.9500
...
```