Machine Learning course
University of Parma - A.Y. 2020/2021

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 tests, classification, clustering, etc.), 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.
Finally, some practical examples of Multi-Layer Perceptrons will be presented.

2 Installation and prerequisites

2.1 R

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 stores 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.

2.2 RStudio

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
```

2.3 Caret

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, useful because it allows Caret to preserve
the proportion of the categories specified in this parameter.
`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 preprocessing over our dataset can be useful, like normalization or standardization.
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.
See `https://topepo.github.io/caret/available-models.html` for further details.

Each model may be implemented using a different backend library, but Caret's interface remains the same, simplifying the developer's work.

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 + ...` 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 that specifies the 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)
```

Looking at the output generated, we can observe that not only a confusion matrix has been generated, but also some relevant statistics.

From the confusion matrix, we can observe how exactly 1 instance of Iris-virignica has been misclassified as Iris-versicolor.
This led to a global accuracy of 96.67%.

Also, for each class some separate metrics are computed, such as sensitivity, specificity, precision, recall, etc.

```
  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
  Precision                     1.0000                 1.0000             0.9091
  Recall                        1.0000                 0.9000             1.0000
  ...
```

4 Examples

In this final section, some complete examples of MLPs are presented.
Caret offers various models in order to build MLPs, such as `'nnet'`, `'mlp'`, `'mlpML'`, `'mlpKerasDropout'`, `'mlpKerasDropoutCost'` and many more.

Each of them provides a different set of tuning parameters that need to be assigned in the `tuneGrid` parameter of the `train` function.

4.1 Classification with MLP (1 layer)

In this first example, the objective is to perform a classification task over the well known iris dataset, using a Multi-Layer Perceptron with a single layer of 3 neurons.
For this task, the `nnet` model is used.

```
# 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 a MLP with 1 layers with 3 neurons
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 istances have been classified with an accuracy of 96.67%.

Looking at the confusion matrix, we note that only 1 instance has been misclassified.

Of course, a lot of metrics have been generated because we selected `'everything'` as value for the `mode` parameter of the confusion matrix.

```
 [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
...
```

4.2 Classification with MLP (3 layers)

In this example, we want to adapt the previous problem using a 3-layers MLP.

This can be done using the `mlpML` model, which allows to specify up to 3 layers as tuning parameters (other models like `mlpSGD` provide also more sophisticated tuning parameters, such as `learn_rate` and `momentum`).

```
# 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 a MLP with 3 layers with 4, 2 and 2 neurons respectively
model <- train(form=Species ~ .,   # outcome ~ predictors
             data=trainData,
             method='mlpML',
             trControl=trainControl(method="repeatedcv", number=10, repeats=10),
             tuneGrid=expand.grid(layer1=3, layer2=2, layer3=1),
             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 we obtained an accuracy of 56.67%.

This accuracy, much lower than the one we obtained with a simpler MLP, indicates that this model is not so adequate for the task: it is too complex, and thus it overfits data.

Moreover, not only accuracy is worse, but almost every other computed metric.

```
 [1] Iris-setosa   Iris-setosa   Iris-setosa   Iris-setosa   Iris-setosa
      Iris-setosa   Iris-setosa
 [8] Iris-setosa   Iris-setosa   Iris-setosa   Iris-versicolor Iris-versicolor
      Iris-setosa Iris-versicolor
[15] Iris-versicolor Iris-setosa Iris-setosa   Iris-versicolor Iris-versicolor
     Iris-versicolor Iris-versicolor
[22] Iris-versicolor Iris-versicolor Iris-versicolor Iris-versicolor Iris-
     versicolor Iris-versicolor Iris-versicolor
[29] Iris-versicolor Iris-versicolor

Levels: Iris-setosa Iris-versicolor Iris-virginica

Confusion Matrix and Statistics

                 Reference
Prediction        Iris-setosa Iris-versicolor Iris-virginica
  Iris-setosa             10               3              0
  Iris-versicolor          0               7             10
  Iris-virginica           0               0              0

Overall Statistics

               Accuracy : 0.5667
               ...

Statistics by Class:

                     Class: Iris-setosa Class: Iris-versicolor Class: Iris-
                          virginica
Sensitivity                   1.0000                 0.7000                0.0000
Specificity                   0.8500                 0.5000                1.0000
...
```

4.3 Regression with MLP

In this final example, the task is to predict the value of a potential car sale (i.e. how much a particular person will spend on buying a car) for a customer on the basis of the following attributes: age, gender, average miles driven per day, personal debt, monthly income.

This can be done for example using a `mlp` model, specifying `RMSE` as the chosen metric.

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

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

set.seed(100)

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

# print(nrow(trainData)) # 771
# print(nrow(testData)) # 192

model = train(form=Car_price ~ .,
             data=trainData,
             method="mlp",
             preProc=c("center", "scale"), # Preprocessing can be done also here
             trControl=trainControl(method="repeatedcv", number=3, repeats=3),
             tuneGrid=expand.grid(size=3),
             metric="RMSE",    # Root mean squared error as metric
             trace=FALSE)

# Predict "Car_price" attribute for data in testData using the model trained
    before
prediction <- predict(model, testData[-6])
print(prediction) # 192 predictions are made, because nrow(testData) = 192

# Print, among other things, the final RMSE
print(model)
```

This is the output generated, from which we can observe that the 192 test istances have been predicted with a final RMSE of 9296.64.

```
6            16           19           25           27
8.551736e+03 1.946015e+04 8.551736e+03 1.605100e+04 1.946015e+04

...

920          929          944          952          955
8.551736e+03 1.090841e+04 1.605100e+04 8.551736e+03 2.460274e+04

Multi-Layer Perceptron

963 samples
  5 predictor

Pre-processing: centered (5), scaled (5)
Resampling: Cross-Validated (3 fold, repeated 3 times)
Summary of sample sizes: 642, 643, 641, 642, 642, 642, ...
Resampling results:

  RMSE      Rsquared   MAE
  9296.637  0.2369883  7287.627
```

Of course, an RMSE value is not so immediate to be interpreted, unlike an accuracy value.
To have a better understanding of how the model performs, we can plot the real outputs vs the predicted outputs, for every instance of the test set.

We can see that the model is not so precise in its predictions: in this case, it underfits its data.

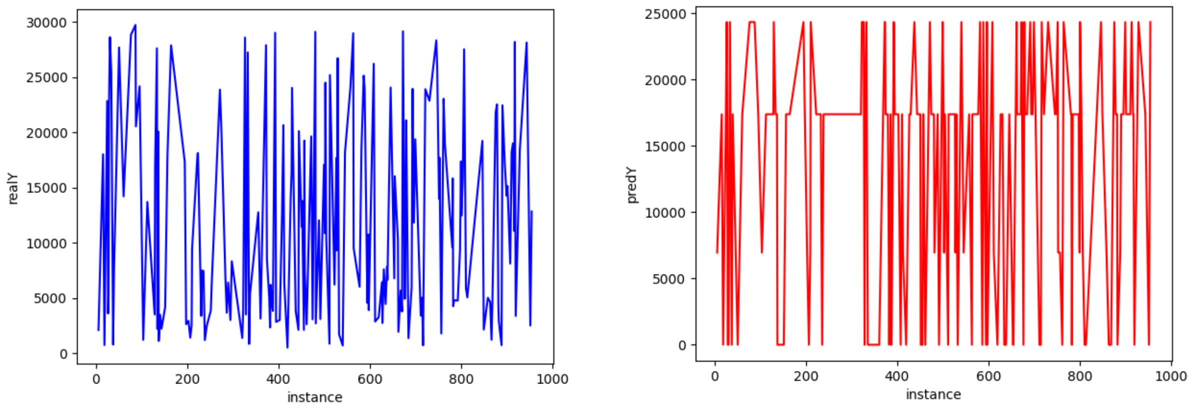

Figure 1: real outputs vs predicted outputs

References

[1] *The R Project*
`https://www.r-project.org/`

[2] *CRAN - The Comprehensive R Archive Network*
`https://cloud.r-project.org/`

[3] *R Tutorial*
`https://www.w3schools.com/r/default.asp`

[4] *RStudio*
`https://www.rstudio.com/products/rstudio/`

[5] *Caret*
`https://topepo.github.io/caret/`

[6] *Caret package on CRAN*
`https://cran.r-project.org/web/packages/caret/`