The machine learning approach: clustering words and classifying documents with R

Iñaki Inza, Borja Calvo

Once the document-term matrix with an specific sparseness value is elected and depending on the interests of the NLP expert, different types of machine learning models can be learned. This tutorial covers the *clustering of words with similar patterns of occurrences across documents* and the *classification of documents*. We continue working with the document-term matrices built in the previous tutorial, "A short introduction to the `tm (text mining)` package in `R`: text processing".

> You have already realized that `R` works by creating "objects" and consulting their attributes. Objects are of different types: and this constrains the type of operations that can be applied to each object. For example, text preprocessing operations can only be applied to an object of `VCorpus` type. On the other hand, machine learning operations (e.g. learning classifiers) can not be applied to a `VCorpus` type object: it has to be transformed (i.e. "casting") to a data-matrix or data-frame type.

Clustering of words with similar patterns of occurrences across documents

We try to find clusters of words with hierarchical clustering, a popular clustering techniques which builds a dendogram to iteratively group pairs of similar objects. To do so, a matrix which has removed sparse is needed: the starting point is the 0.8 sparseness-value matrix. After the application of the matrix-casting operator, number of occurrences are scaled: first column (term) mean is substracted, and then this is divided by its standard deviation. It is needed to calculate the distance between pairs of objects (terms): these are saved in `distMatrix`. The `dist` operator performs this calculation between pairs of rows of the provided matrix. As terms appear in the columns of the document-term matrix (`sci.rel.dtm.80`), it is needed to be transposed by means of the `t` operator. The clustering-dendogram is built with the `hclust` operator. It needs as input the calculated distance matrix between pairs of terms and a criteria to decide which pair of clusters to be consecutively joined in the bottom-up dendogram. In this case, the "complete" criteria takes into account the maximum distance between any pair of objects (terms) of both clusters to be merged. Heigth in the dendogram denotes the *distance* between a merged pair of clusters. Figure 1 shows the resulting dendogram.

```
distMatrix=dist(t(scale(as.matrix(sci.rel.dtm.80))))
termClustering=hclust(distMatrix,method="complete")
plot(termClustering)
```

> Instead of hierarchical clustering and following a similar set of functions, the `fpc` package allows to construct a `k-means` clustering.

> Another type of popular NLP machine-learning analysis is to construct clusters of similar documents based on the frequencies of word occurrences.

Año: EMLCT Master
Profesor(a): Iñaki Inza, Borja Calvo
Teléfono: 943 01 50 26
E-mail: inaki.inza@ehu.es

GIPUZKOAKO CAMPUSA
CAMPUS DE GUIPUZCOA
Paseo Manuel de Lardizabal, 1
20018 Donostia (Gipuzkoa)

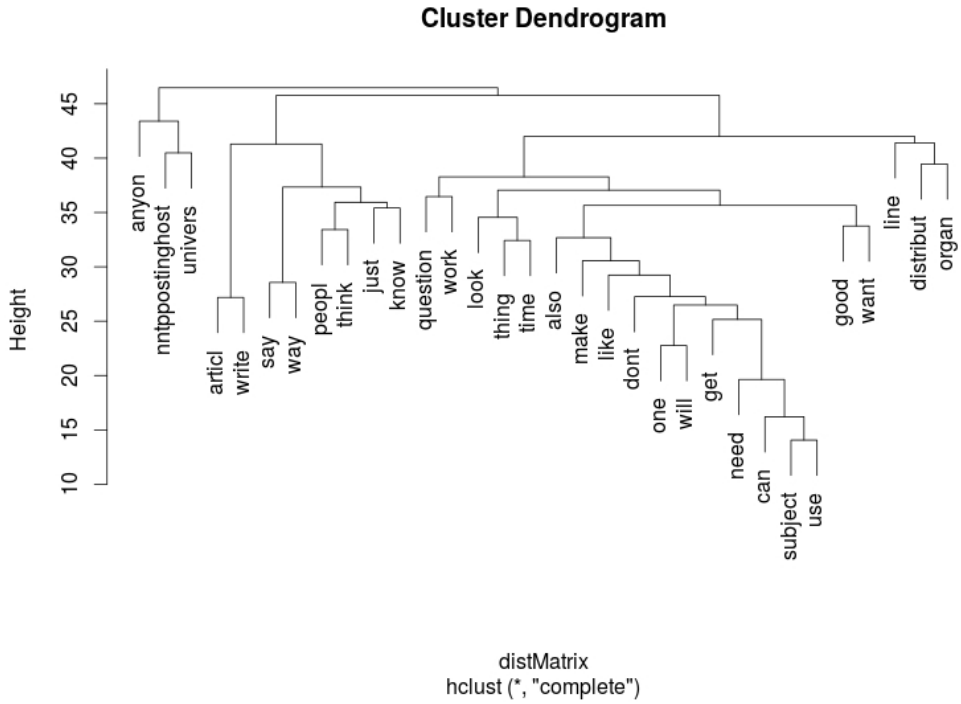

Figure 1: Cluster dendogram which groups terms by their frequency of occurrence

Classification of documents

Our objective is to learn a classifier-model which, based on terms occurrences, predicts the type-topic ("science-electronics" or "religion") of future documents (post-new, in the case of newsgroups). We have a two-class problem.

The 0.9 sparseness value document-term matrix is our starting point. We first need to append the class (document type) vector as the last column of the matrix: the first 591 documents cover the "science-electronics" newgroup.

```
dim(sci.rel.dtm.90)
type=c(rep("science",591),rep("religion",377)) # create the type vector
sci.rel.dtm.90=cbind(sci.rel.dtm.90,type) # append
dim(sci.rel.dtm.90) # consult the updated number of columns
```

This new matrix is the starting point for any software specialized on supervised classification. However, it is needed to concatenate "matrix" and "data.frame" casting operations. The name of the last column is updated.

```
sci.rel.dtm.90.ML.matrix=as.data.frame(as.matrix(sci.rel.dtm.90))
colnames(sci.rel.dtm.90.ML.matrix)[119]="type"
```

> The different columns of an object can be accessed by typing the tab after the object name and the $ symbol.

The `caret` [2, 1] package is the reference tool for building supervised classification and regression models in `R`. The following shows the current top machine learning packages in `R`. `caret` package covers all the steps of a

classic pipeline: data preprocessing, model building, accuracy estimation, prediction of the type of new samples, and statistical comparision between the performance of different models. Another similar package is `mlr3`. If you are interested, you can find an interesting tutorial.

> Very useful: the cheatsheet of `caret`. Its principal functions illustrated in a single page.

Before learning a classification model it is needed to define the subsets of samples (documents) to train and test the it. The `createDataPartition` produces a train-test partition of our corpus of 968 documents. This will be maintained during the whole pipeline of analysis. Test samples won't be used for any modeling decision: only to predict their class and create a confusion matrix. Consult the parameters of `createDataPartition`, as well as other two functions with similar purposes, `createFolds` and `createResample`. A list of randomly sampled numbers (object `inTrain`), as index numbers, is used to partition the whole corpus in two `R` objects.

```
library(caret)
set.seed(107) # a random seed to enable reproducibility
inTrain <- createDataPartition(y=sci.rel.dtm.90.ML.matrix$type,p=.75,list=FALSE)
str(inTrain)
training <- sci.rel.dtm.90.ML.matrix[inTrain,]
testing <- sci.rel.dtm.90.ML.matrix[-inTrain,]
nrow(training)
```

We now can start training and testing different supervised classification models. `train` function implements the building process. Check its parameters: among them, we highlight the following:

- `preProcess` parameter defines the preprocessing steps to be applied. They are popular with classic numeric variables, such as imputation of missing values, centering and scaling, etc. As it was shown in the previous tutorial, NLP datasets have their own preprocessing tools. They are not going to be applied in our dataset;
- `trControl` parameter defines the method to estimate the error of the classifier. It is defined by means of the application of the `trainControl` function. This allows the use of different performance estimation procedures such as k-fold cross-validation, bootstrapping, etc. We apply a 10-fold cross-validation, repeated 3 times;
- `method` parameter fixes the type of classification algorithm to be learned. The wide list of algorithms (and its parameters) covered by `caret` can be found in `https://topepo.github.io/caret/train-models-by-tag.html`. Taking into account the large dimensionality of classic NLP datasets, the use of classifiers capable to deal with this characteristic is highly recommended. In this tutorial, *Linear support vector machine (SVM)* and `k-nearest neighbour (K-NN)` models are learned and validated;
- `metric` parameter fixes the score to assess-validates the goodness of each model. Apart from the ROC metric used in this tutorial, a large set of metrics is offered: Accuracy (percentage of correct classification), kappa, Sens (Sensitivity), Specificity (Spec), RMSE in the case of regression problems... I have not found a list with all the metrics offered by `caret`, but consulting the help of the package will give you a broad idea;

Take into account the following facts about the model-training step:

- the expression `type ~` is quite popular in `R` to denote the specific variable to be predicted, followed by the set of predictors. A point indicates that the rest of variables are used as predictors;
- together with the estimation of the percentage recognition (accuracy), the value of the *Kappa statistic* is shown. It is a popular score in NLP studies. Its definition is not trivial in the context of supervised classification. Roughly, this score compares the "observed" accuracy of the learned classifier with respect

to a random classifier: measuring the score difference between our classifier and a random classifier. A larger definition of this metric can be found in `http://stats.stackexchange.com/questions/82162/kappa-statistic-in-plain-english`;

- while the linear SVM classifier does not have parameters, K-NN has the "number of neighbours" (K) key parameter. By default, changing the value of the parameter, `caret` evaluates 3 models (`tuneLength` equal to 3). The `tuneLength` option of the `train` function fixes the number of values of each parameter to be checked. For example, if the classifier has 2 parameters and the `tuneLength` parameter is not changed, $3 \times 3 = 9$ models are evaluated. The `tuneGrid` option offers the possibility to select among a set of values to be tuned-tested;
- `caret` supports more than 150 supervised classification and regression algorithms. A small portion of them are learned by means of the software of the package itself. The majority of the algorithms are learned by other `R` packages which are conveniently accessed by `caret`.

```
# fixing the performance estimation procedure
ctrl <- trainControl(method = "repeatedcv",repeats=3)
svmModel3x10cv <- train (type ~ ., data=training,method="svmLinear",trControl=ctrl)
svmModel3x10cv
knnModel3x10cv <- train (type ~ ., data=training,method="knn",trControl=ctrl)
knnModel3x10cv
```

The training process can still be enriched with extra parameters in the `trainControl` function: `summaryFunction` controls the type of evaluation metrics. In binary classification problems (e.g. "science" versus "religion") the `twoClassSummary` option displays area under the ROC curve, sensitity-recall and specificity metrics. To do so, it is also needed to activate the `classProbs` option which saves the probability that the classifier assigns to each sample belonging to each class-value.

```
install.packages(pROC)
library(pROC)
ctrl <- trainControl(method = "repeatedcv",repeats=3, classProbs=TRUE,
                     summaryFunction=twoClassSummary)
knnModel3x10cvROC <- train (type ~ ., data=training,method="knn",trControl=ctrl,
                            metric="ROC",tuneLength=10)
knnModel3x10cvROC
plot(knnModel3x10cvROC)
```

In order to predict the class value of unseen documents of the test partition `caret` uses the classifier which shows the best accuracy estimation of their parameters. Function `predict` implements this functionality. Consult its parameters. The `type` parameter, by means of its `probs` value, outputs the probability of test each sample belonging to each class ("a-posteriori" probability). On the other hand, the `raw` value outputs the class value with the largest probability. By means of the `raw` option the confusion matrix can be calculated: this crosses, for each test sample, predicted with real class values.

```
svmModelClasses <- predict(svmModel3x10cv, newdata = testing, type = "raw")
confusionMatrix(data=svmModelClasses,testing$type)
```

Can a *statistical comparison* be performed between the 3x10cv validation results of K-NN and SVM? Note than in our case, due to the 3 repetitions of the 10-fold cross-validation process, there are 30 resampling results for each classifier. First, results of both classifiers are crossed using the `resamples` function. As the `set.seed` did not change, the same paired cross-validation subsets of samples were used for both classifiers. This forces to use a *paired* t-test to calculate the significance of the differences between both classifiers. A simple plot is drawn, showing the accuracy differences between both models for each of the 30 cross-validation folds: note

that svm has a better accuracy than knn for all the cross-validation fold results (Figure 2).

Using the `diff` function over the `resamps` object which saves the "crossing" of both classifiers, we can show a rich output of the performed comparison: each number matters. The output shows, for each metric (area under the ROC curve, sensitivity, specificity), the difference of the mean (positive or negative, following the order in the `resamples` function) between both classifiers. The p-value of the associated t-test is also shown.

The interpretation of the p-value has the key. It is related with the risk of erroneously discarding the null-hypothesis of similarity between compared classifiers, when there is no real difference. Roughly speaking, it can also be interpreted as the degree of similarity between both classifiers. A p-value smaller than 0.05 (or 0.1, depending on your interpretation and threshold) alerts about statistically significant differences between both classifiers, `https://en.wikipedia.org/wiki/Statistical_significance`. That is, when the risk of erroneously discarding the hypothesis of similarity between both classifiers is low, we assume that there is a statistically significant difference between classifiers.

```
resamps=resamples(list(knn=knnModel3x10cv,svm=svmModel3x10cv))
summary(resamps)
xyplot(resamps,what="BlandAltman")
diffs<-diff(resamps)
summary(diffs)
```

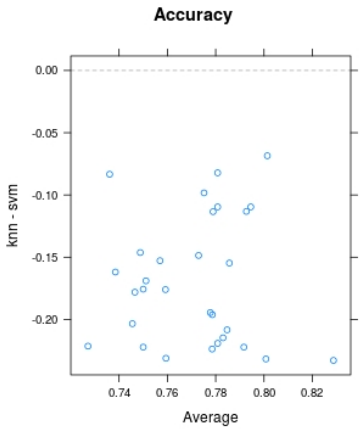

Figure 2: SVMLinear versus K-NN. "Bland-Altman" plot. SVM outperforms K-NN in the 30 resampling estimations

Evaluation exercise

Together with the explanations of the previous `tm` tutorial, create a corpus and a document-term matrix which characterises a **supervised classification** problem of your interest: 2 different types of documents, different types of sentiments, different types of text difficulties... that is, the corpus has to be annotated, at least with two different labels, in order to proceed with supervised classification tasks.

As the output of the exercise, we hope a document similar to this tutorial: that is, a **notebook** which mixes code and the explanations of this code and your decisions. A rich document. The document has to alternate explanations about your decision and the `R` code that implements it: colloquially, do "do your own tutorial-notebook", with your chosen application-corpus. Specifically, the document that you have read has been edited with the `knitr` package and the `RStudio` software: `Rnw` format. You can of course edit with this technology or another of your convenience: R-markdown, jupyter notebooks, etc.

Your notebook-tutorial has to cover the following items:

- a short description of your dataset and NLP problem. The nature of the problem and the objective of the modelization, the class variables to be predicted and its possible values, how can the models be evaluated, the predictive features and another concept which helps in the description of the domain;
- over the whole corpus, apply one of the **outlier detection** techniques exposed in class and consider deleting extreme-outlier documents;
- show a **wordcloud** based on your corpus bag-of-words;
- the way to **partition the corpus in train-test** does not have to be the same of the tutorial. There are other possibilities such as `createFolds()`, `createResample()`, etc.
- choose two other **classification algorithms** not used in this tutorial. For your selection, **describe** briefly their behaviour and its parameters. Take into account the effect of the exposed `tuneLength` option: use the `tubeLength` and-or `tuneGrid` options to **tune the parameters of chosen classification algorithms**;
- remember that the `trainControl()` function allows to select the options to validate the classifier. Consult its options and do **variations** with respect to my selection;
- if class-label distributions are **unbalanced** in your corpus, test a 'resampling' method which will try to improve the recovery rate in the minority class. Which is the class-label distribution of your corpus? Show it. Study the options of the `sampling` option in `trainControl`: enlace. `caret` has a brief and intuitive tutorial about the topic: check the first sentences of this link. Its use is also explained in this link;
- while in my tutorial the **chosen metric** to validate the models has been ROC, you can of course select another metric of your convenience: accuracy, sensitivity, F_1, kappa, etc. If you do not have prior information, the default and common-sense score metric to choose in your supervised classification application is *accuracy*;
- test the **feature selection** options offered by `caret`. At least, apply one of them over your corpus. Can the feature subset to learn the final classifier be optimized? `caret` offers genetic algorithms by wrapper, simmulated annealing by wrapper, recursive backward elimination by wrapper. Or univariate filters that rank independently the variables with respect to their relevance-correlation with respect to the annotation-column;
- previous item was 'feature selection'. Let's now practise with **feature extraction**, where a set of features is 'constructed' from original ones: commonly, linear combinations of original ones. In this area, it is likely that you know algorithms such as principal component analysis - PCA, singular value decomposition, etc. It is easy to learn a PCA in `R` with the `prcomp` function and visualize in a 2-D graph two first components (i.e. those that save larger variability of original data): trying to find an intuitive separation of problem-classes. Try it and comment the results;

- remember that the `predict()` functions allows to select the options to **predict the class-label of the samples of the test partition**. Consult its options and do variations with respect to my selection. Do class-label predictions over new-unseen-unnanotated documents;
- to do the final **statistical comparison** between your pair of selected classifiers, it is needed to understand the use and output of `resamples()`, `summary()`, `diff()` functions. The output is long but rich in information to extract conclusions of the comparison. Interpret the calculated **p-value** to analyze the significance of the differences between compared classifiers. In its current implementation, `caret` applies a *t-test* to obtain the significance of the differences between the pair of compared classifiers.

References

[1] Max Kuhn. Contributions from Jed Wing, Steve Weston, Andre Williams, Chris Keefer, Allan Engelhardt, Tony Cooper, Zachary Mayer, and the R Core Team. *caret: Classification and Regression Training*, 2014. R package version 6.0-35.

[2] M. Kuhn and K. Johnson. *Applied Predictive Modeling*. Springer, 2013.