

Applied Machine Learning

Text Retrieval, Association Rules and Decision Tree Learning using R

Adrian Cuyugan / Information Analytics, GAMBI

Contents

1	Introduction	2
2	Scope	2
2.1	Techniques	2
2.2	Before starting	3
3	Data	3
3.1	Source and Goal	3
3.2	Bag of Words	3
3.3	Data fields	3
4	Text Retrieval	3
4.1	Load the dataset	3
4.2	Build corpus	4
4.3	Sparsity	5
5	Association Rules	5
5.1	Negative reviews	5
5.2	Random sampling	5
5.3	A Priori algorithm	6
5.4	Subset prune	7
5.5	Custom stopwords	8
5.6	Graph	8
6	Decision Tree Learning	10
6.1	Frequent words	10
6.2	Bit-vector weight	11
6.3	Training and testing sets	11
6.4	Build and visualize tree	11
6.5	Complexity parameter	14
6.6	Model evaluation	14

7 Citations	15
7.1 References	15
7.2 Tools	16

1 Introduction

Usually, Machine Learning stops after you understand how the algorithm runs. Data Mining, on the other, is the real-life application of how these complex algorithms is put into practice with very noisy data - recognizing any patterns, discovering outliers and estimating coefficients which and can later be developed to produce accurate predictions.

2 Scope

2.1 Techniques

This document assumes that you have basic knowledge on the following subjects:

1. **Information Retrieval** - the lesson stops at text retrieval since the data is often offline and does not necessarily require a constant query function, $f(q, d)$. For this activity, [tm](#) package is used to create a corpus and build the sparse matrix. Also, n-gram bag of words is used to construct our sparse matrix using the raw term frequency count for the association rules and a binary-term frequency weighting for decision tree learning.

$$VSM(q, d) = \sum_{i=1}^n x_1 y_1 \dots + \dots x_n y_n$$

2. **Association Rules** - finding frequent patterns based on the text corpus we have built from [tm](#). [arules](#) and [arulesViz](#) are used for this. Also, [Gephi](#) visualizes the network of rules created from the Apriori algorithm. *Corpus* is a latin word which is a collection of plain text that can be read by the computer when doing Natural Language Processing - linguistics, building sparse matrix, frequency count, etc.
3. **Decision Tree** - the simplest Classification and Regression Tree (CART) algorithm computes for entropy based the response variable that we set, *sentiment*. Remember that entropy is:

$$E = - \sum_{i=1}^k \frac{C_i}{n} \log_2 \frac{C_i}{n}$$

[rpart](#) builds CART and estimates the model because it contains additional features like cross-validation within training set. Also, [rattle](#) is used to visualize the tree.

For beginners, [rattle](#) is another data mining tool in R which has a GUI if you are not yet familiar with R coding.

Warning: There are so many data analysts who can use the tool but does not know the algorithm behind it. This is a danger zone because you can interpret the results incorrectly.

2.2 Before starting

You can use `?help` to search what are the description and available parameters that you can use in this example. For example, if you want to know what does `head()` do, type `?head` in R console.

The scripts are coded as simple as possible with attached comments so that you can understand what happens within each line of code.

The scripts do not include `install.packages()` as it assumes that you have already installed the packages. If you have not, please refer to the above techniques as the names of packages are highlighted with their corresponding documentation from CRAN directory.

3 Data

3.1 Source and Goal

A current knowledge-based competition (bragging rights award) is being held at [Kaggle](#). One of the available data set we could use is [Google's Word2Vec's](#). For this exercise, Natural Language Processing (parsing of words) is not done because it requires further understanding on the subject matter, only bag of words.

Although model evaluation is not discussed (though I highly suggest that you make further reading on this), model accuracy and other quantitative measures are included. There is a big difference between building the model and interpreting the results than doing a prediction that leads to prescribing recommendations as these models are tested against another source of data set which is usually the testing set and/or cross-validation set.

3.2 Bag of Words

Unigram *bag of words* is used to create the vector of words based on all 25,000 movie reviews. These are also labelled so that we can calculate the estimates of probability of a word producing a positive or negative review. Sampling is also done to produce the association rules as it requires more memory to run the algorithm.

3.3 Data fields

1. **id** - Unique ID of each review. This is excluded from the dataset but is very helpful in identifying the unique records of review. `rowname()` function in R accomplishes this task.
2. **sentiment** - The response variable. This is the sentiment of the review; 1 for positive reviews and 0 for negative reviews. This is ignored when doing unsupervised learning.
3. **review** - This is a free-form text field where the users wrote their reviews. This is converted into a *Document Term Matrix* as a result of the text retrieval process.

4 Text Retrieval

4.1 Load the dataset

The CSV file is downloaded for offline use so that this document can be generated faster. The data set could be saved into the working directory. You can check your current working using `getwd()` or manually set it using `setwd()`. For easier viewing of the codes, C drive is used as a working directory.

```

train <- read.csv("C://popcorn-train.csv", head = T, sep = ",") # Load data set

rownames(train) <- train$id # Assigns the ID as row identifier
train <- train[,2:3] # Subsets only Sentiment and Comments
train$sentiment <- factor(train$sentiment,
                          levels = c(1,0),
                          labels = c("Pos", "Neg")) # Converts 0 and 1 into categories
table(train$sentiment) # Count of records per sentiment

```

```

##
##   Pos   Neg
## 12500 12500

```

After loading the data set into R environment, **id** variable is assigned as the row identifier then, it is dropped since it has already been attached. Since R automatically recognizes **sentiment** as a numerical variable, this is changed to categorical by using the **factor()** function.

Note that the proportion of Positive and Negative reviews are equal.

4.2 Build corpus

The corpus could be built from a data frame or a vector format. This example uses data frame since the CSV have been loaded as a data frame. Each line of code converts the corpus into its respective transformation function. For more advanced control of parameters, please refer to `?tm_map`.

```

# Prepare corpus
require(tm)
keyword <- as.data.frame(train$review)
keyword <- Corpus(DataframeSource(keyword)) # Creates corpus
keyword <- tm_map(keyword, removeWords, stopwords(kind = "english")) # Removes stop words
keyword <- tm_map(keyword, stemDocument, language = "porter") # Applies porter stemming
keyword <- tm_map(keyword, stripWhitespace) # Removes whitespace
keyword <- tm_map(keyword, content_transformer(tolower)) # Converts to lowercase
keyword <- tm_map(keyword, removePunctuation) # Removes punctuation marks
keyword <- tm_map(keyword, removeNumbers) # Removes numbers

```

```

# Raw Term Frequency
keyword.term <- DocumentTermMatrix(keyword) # Creates a sparse matrix
keyword.term # Checks for how many terms were produced

```

```

## <<DocumentTermMatrix (documents: 25000, terms: 102984)>>
## Non-/sparse entries: 2459554/2572140446
## Sparsity           : 100%
## Maximal term length: 60
## Weighting           : term frequency (tf)

```

There are 102,984 terms produced by the algorithm. In Information Retrieval, all of these could be included in the model but since it built a very large matrix, dropping sparsed terms that barely have frequency count in the matrix is done to minimize computational load.

4.3 Sparsity

Use `removeSparseTerms` to remove low frequency count terms.

```
keyword.term <- removeSparseTerms(keyword.term, sparse = 0.9)
keyword.term # Checks for sparsity
```

```
## <<DocumentTermMatrix (documents: 25000, terms: 126)>>
## Non-/sparse entries: 606899/2543101
## Sparsity           : 81%
## Maximal term length: 8
## Weighting          : term frequency (tf)
```

After sparsing, non-essential terms is minimized to < 130 or 80%. Although this is still a lot of variables that CART would handle, based on a data mining analyst's judgement, this could still be adjusted to < 0.90 which would lessen the bag of words/variables included in the final model.

As CART requires a response variable, the document term matrix is combined with the sentiment vector.

```
keyword.matrix <- as.data.frame.matrix(keyword.term) # Converts to data frame matrix
keyword.matrix <- cbind(keyword.matrix, sentiment = train$sentiment) # Combines response
rownames(keyword.matrix) <- row.names(train) # Applies row identifier
head(keyword.matrix[150:170,118:127]) # Previews the combined data frame
```

```
##           what whole will without wonder work world year young sentiment
## 11973_4      0      0      0          0          0      0      0      0      0      Neg
## 4541_10      0      0      0          0          0      0      0      0      0      Pos
## 5169_7       0      1      0          0          0      0      0      0      0      Pos
## 6809_1       0      0      0          0          0      0      0      1      0      Neg
## 8125_7       0      0      0          0          0      3      0      1      0      Pos
## 4802_8       0      0      0          0          0      1      0      0      1      Pos
```

5 Association Rules

5.1 Negative reviews

For this activity, only the negative reviews are used to look for frequent patterns. Note that when the matrix is built, raw term frequency is used; the values of the matrix would be totally different if TF-IDF weighting is selected.

5.2 Random sampling

Building models takes a lot of memory resources and when you hit the ceiling, your computer will definitely crash. To minimize the computational load, the negative reviews is sampled without replacement to 5,000 records.

```
popcorn.neg <- subset(keyword.matrix, sentiment == "Neg") # Subsets negative reviews
popcorn.neg <- popcorn.neg[, !(colnames(popcorn.neg) %in% "sentiment")] # Drops response
# Random sampling
```

```
set.seed(14344) # For reproducibility
popcorn.neg <- popcorn.neg[sample(1:nrow(popcorn.neg), 5000, replace=FALSE),] # Samples
popcorn.neg <- as.matrix(popcorn.neg) # Converts data frame to matrix
```

For the results to be fully reproducible, `set.seed()` is used so that when you generate the resampling function, the same records are selected.

5.3 A Priori algorithm

One of the most common algorithms in doing association rules is a priori. Before running the function, the data set needs to be converted from matrix/data frame into a transactions class. For example:

ID	Transactions/Bag of Words
1	example, film, interested, just, like, look, movie, scene, will
2	because, beyond, director, movie, know, like, movie, want
3	because, better, forget, get, know, look, movie, not, will, work
4	better, interested, movie, same, something, work

```
require(arules)
popcorn.neg <- as(object = popcorn.neg, Class = "transactions") # Converts to transaction
popcorn.neg
```

```
## transactions in sparse format with
## 5000 transactions (rows) and
## 126 items (columns)
```

After sampling, a reproducible random sample of 5,000 records with 127 bag of words is converted into a transactions matrix format like above.

```
neg.arules <- apriori(popcorn.neg, parameter=list(support=0.1, conf=0.1)) # Runs algorithm

##
## Parameter specification:
## confidence minval smax arem aval originalSupport support minlen maxlen
##      0.1      0.1      1 none FALSE          TRUE      0.1      1      10
## target  ext
## rules FALSE
##
## Algorithmic control:
## filter tree heap memopt load sort verbose
##    0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## apriori - find association rules with the apriori algorithm
## version 4.21 (2004.05.09)      (c) 1996-2004  Christian Borgelt
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[126 item(s), 5000 transaction(s)] done [0.01s].
## sorting and recoding items ... [113 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 done [0.01s].
```

```
## writing ... [3099 rule(s)] done [0.00s].
## creating S4 object ... done [0.00s].
```

```
neg.arules <- sort(neg.arules, decreasing=T, by="lift") # Sorts based on lift values
```

The *minsup* (relative support) and *minconf* (conditional probability) values totally depend on the analyst's judgement; also, expertise in the domain helps. Confidence is computed as follows:

$$\text{conf}(X \Rightarrow Y) = \frac{\text{sup}(X \cup Y)}{\text{sup}(x)} = P(Y|X) = \frac{P(Y \cap X)}{P(X)}$$

Find time to look at the output of the model. Based on the *minsup* = 10% and *minconf* = 10%, there are 3,099 rules that are discovered. The *minlen* could also be tweaked.

Then, the rules are sorted based on the magnitude of lift values. Lift is computed as:

$$\text{lift}(X, Y) = \frac{\text{conf}(X \Rightarrow Y)}{\text{sup}(Y)} = \frac{\text{sup}(X \cup Y)}{\text{sup}(X) * \text{sup}(Y)}$$

5.4 Subset prune

One property of frequent patterns is the downward closure where an itemset subset is also frequent if the parent itemset is also frequent. Pruning resolves this to remove ambiguity in the rules.

```
neg.prune <- is.subset(x = neg.arules, y = neg.arules) # Finds subsets
neg.prune[lower.tri(x = neg.prune, diag = TRUE)] <- NA # Discards lower triangle of matrix
neg.redundant <- colSums(neg.prune, na.rm = TRUE) >= 1 # Computes for sums of terms
neg.prune <- neg.arules[!neg.redundant] # Removes redundancies based on subsets

# Converts rules into data frame
# elements is a sub-class of the rule, just like subsetting
neg.arules.df = data.frame(
  lhs = labels(lhs(neg.prune))$elements,
  rhs = labels(rhs(neg.prune))$elements,
  neg.prune@quality)

head(neg.arules.df, 20) # Views top 20 rules based on lift
```

##	lhs	rhs	support	confidence	lift
## 2758	{just,like,movi}	{movie}	0.1068	0.5297619	1.576672
## 2726	{just,like,the}	{look}	0.1038	0.4556629	1.546717
## 1834	{movi,watch}	{movie}	0.1272	0.5170732	1.538908
## 2754	{even,movi,the}	{movie}	0.1044	0.5158103	1.535150
## 2766	{just,movi,the}	{movie}	0.1182	0.5025510	1.495688
## 1726	{like,make}	{look}	0.1084	0.4395783	1.492119
## 2770	{like,movi,the}	{movie}	0.1320	0.4988662	1.484721
## 1846	{good,movi}	{movie}	0.1244	0.4987971	1.484515
## 1873	{even,movi}	{movie}	0.1276	0.4984375	1.483445
## 1858	{get,movi}	{movie}	0.1214	0.4979491	1.481991
## 1825	{can,movi}	{movie}	0.1014	0.4970588	1.479342
## 2738	{just,like,the}	{realli}	0.1016	0.4460053	1.478797
## 1840	{movi,see}	{movie}	0.1218	0.4963325	1.477180
## 1828	{bad,movi}	{movie}	0.1206	0.4930499	1.467410

```
## 1852      {movi,time}  {movie}  0.1208  0.4918567  1.463859
## 1738      {just,like}  {look}   0.1246  0.4281787  1.453424
## 1900      {like,movi}  {movie}  0.1696  0.4870764  1.449632
## 2994  {just,make,the}  {even}   0.1020  0.5723906  1.447624
## 1891      {just,movi}  {movie}  0.1512  0.4861736  1.446945
## 482       {want}      {see}    0.1184  0.5262222  1.435412
```

```
neg.prune # Total rules after pruning
```

```
## set of 1249 rules
```

You can use `inspect()` to view the rules from the `arules` package. Unfortunately, since the `tm` package is loaded in the environment and it uses the same function for a different purpose, this masks `inspect()` from `arules` package. That's why the rules are converted into a data frame so that it can be easily exported or viewed.

There are a total of 1,249 rules after pruning.

Going back to the top 20 rules, assume interpretations of patterns found by the a priori algorithm.

5.5 Custom stopwords

Now, the goal is to construct associations based on the rules found. As the rules that merit high lift values do not make sense (or maybe they do, again, domain expertise is needed), *just*, *the*, *really*, these can be added to a list of custom stop words. Then, another run of pattern mining could be done to find more meaningful words. For this exercise, let's assume that these words make sense to construct meaningful assumptions or **if** \Rightarrow **then** statements.

Another approach is to use TF-IDF because common words such as *movi* and *movie* score low using this weighting. Remember that a normalized TF-IDF with smoothing is computed as:

$$TF_{IDF} = \log\left[\frac{(M+1)}{k}\right]$$

5.6 Graph

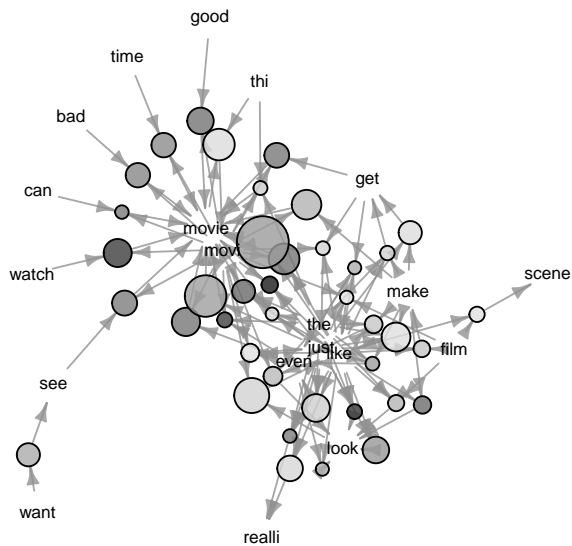
The first 40 rules ranked by the magnitude of the lift value are visualized to show their association. The size of the bubbles is the *support* and the color density of it is the *lift*. Precision is set to show upto tenth decimal and `cex` is a numerical value giving the amount by which plotting text and symbols should be magnified/demagnified relative to the default.

```
require(arulesViz)
par(mfrow=c(1,2)) # Displays two plots in one graphics
plot(head(neg.prune,40), method="graph",
      control=list(type="items", precision=1, cex=.6,
                   main="Negative Reviews Items")) # Plots items rules
plot(head(neg.prune,40), method="graph",
      control=list(type="itemsets", precision=1, cex=.6,
                   main="Negative Reviews Itemsets")) # Plots itemsets
```

```
par(mfrow=c(1,1)) # Resets graphic device
```


Negative Reviews Items

size: support (0.1 – 0.2)
color: lift (1.4 – 1.6)



Negative Reviews Itemsets

width: support (0.1 – 0.2)
color: lift (1.4 – 1.6)

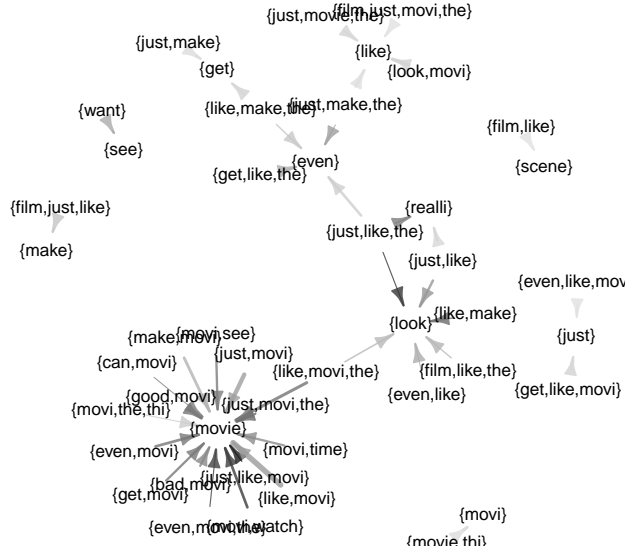


Figure 1: Items and itemsets

Using some basic understanding in graph theory (which is another topic - centrality, degrees, clustering, etc), the words could be clustered based on their association with one another. I would let you interpret the graphs for deeper understanding.

Another approach is to perform community modularity clustering based on all rules found in the model. [Gephi](#) is particularly effective in this and it is open-source. Figure 2 is an example.

Basic NLP could be performed by just looking for patterns using *syntactic parsing* (nouns vs verbs) compared with positive reviews. By parsing the bag of words using bigram to n-gram models, more advanced techniques such as *Latent Dirichlet Allocation* to perform [Topic Modeling](#) could be done. This is very helpful in classifying words specially when analyst's domain expertise is very limited.

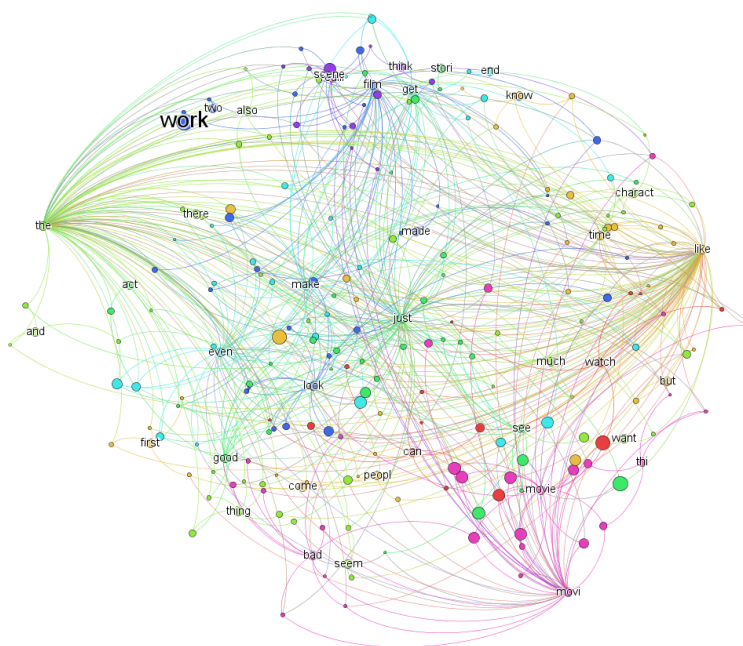


Figure 2: Text Network Analysis using Gephi

6 Decision Tree Learning

6.1 Frequent words

As mentioned, a typical CART is a greedy algorithm that prefers to look for lower entropy in certain variables that contain higher proportion compared to the rest. Another option before building the model is to look at the most frequent words that are found from the term matrix.

```
findFreqTerms(keyword.term, lowfreq = 5000) # Which words are most frequent?
```

```
## [1] "act"      "actor"    "also"     "and"      "bad"      "best"     "better"
## [8] "but"      "can"      "character" "come"     "end"      "even"     "ever"
```

```
## [15] "film"      "find"      "first"     "get"       "give"      "good"      "great"
## [22] "just"      "know"      "life"      "like"      "littl"     "look"      "love"
## [29] "made"      "make"      "man"       "mani"      "movie"     "movi"      "much"
## [36] "never"     "one"       "peopl"     "plai"      "plot"      "realli"    "sai"
## [43] "scene"     "see"       "seem"      "seen"      "show"      "still"     "stori"
## [50] "take"      "the"       "there"     "thi"       "thing"     "think"     "time"
## [57] "two"       "wai"       "want"      "watch"     "well"      "will"      "work"
## [64] "year"
```

6.2 Bit-vector weight

Assume that you are not familiar with the domain (analyzing movie reviews from forum, for this case), all of the the variables could be used as predictor.

```
keyword.binary <- weightBin(keyword.term) # Applies binary transformation
keyword.binary <- as.data.frame.matrix(keyword.binary) # Converts to data frame
keyword.binary <- cbind(keyword.binary, sentiment = train$sentiment) # Combines response
keyword.binary <- as.data.frame(sapply(keyword.binary,
                                       FUN = as.factor)) # Converts columns to categorical
rownames(keyword.binary) <- row.names(train) # Applies unique ID
```

To further simplify the model building, bit-vector weighting is used. As the matrix contains 1 and 0, all predictors are converted to categorical variables to signify missing and present in the document.

6.3 Training and testing sets

Using complex algorithms to perform descriptive statistics tends to be useless if this cannot be used to predict future state. The simplest way to test accuracy of the model is to divide the data set into two: training and test sets; typically, it is divided into 80-20 splits.

```
set.seed(14344) # For reproducibility
split <- sample(seq_len(nrow(keyword.binary)), # Counts records from 1 to n
               size = 0.8 * nrow(keyword.binary), # Computes for 80% for training set
               replace = FALSE)

popcorn.train <- keyword.binary[split, ] # Creates training set
popcorn.test <- keyword.binary[-split, ] # Created testing set
```

Other methods like bootstrapping and k-fold validations could be used to make sure that the model is iteratively tested with the lowest standard error as possible.

6.4 Build and visualize tree

```
require(rpart)
require(rattle)
popcorn.tree <- rpart(formula = sentiment ~ ., # tests all predictors against response
                     data = popcorn.train, # From the training set
                     method = "class", # Tells model it is a classification tree
                     parms = list(split = "information"), # Uses information gain
                     model = T) # Retains model information
```

The line that indicates “*” denotes a terminal node of the tree (i.e., a leaf node - the tree is not split any further at that node). The legend is also printed. I highly recommend that you interpret the results based on the output with the visualized tree. You can start from the root (top-down approach) or from the leaf (bottom-up approach.)

Information gain is used to a control parameter when the algorithm converges. This is represented as:

$$IG = E_{\mu}(T) - E_{\mu}(T, a)$$

```
# Prints nodes and leaves
```

```
print(popcorn.tree)
```

```
## n= 20000
##
## node), split, n, loss, yval, (yprob)
##      * denotes terminal node
##
##  1) root 20000 9992 Pos (0.4996000 0.5004000)
##    2) bad=1 4583 1149 Neg (0.7492909 0.2507091) *
##    3) bad=0 15417 6558 Pos (0.4253746 0.5746254)
##      6) great=0 11498 5535 Pos (0.4813881 0.5186119)
##        12) love=0 9065 4362 Neg (0.5188086 0.4811914)
##          24) noth=1 983  276 Neg (0.7192269 0.2807731) *
##          25) noth=0 8082 3996 Pos (0.4944321 0.5055679)
##            50) best=0 6758 3213 Neg (0.5245635 0.4754365)
##              100) plot=1 1204  419 Neg (0.6519934 0.3480066) *
##              101) plot=0 5554 2760 Pos (0.4969391 0.5030609)
##                202) just=1 1814  765 Neg (0.5782800 0.4217200) *
##                203) just=0 3740 1711 Pos (0.4574866 0.5425134) *
##          51) best=1 1324  451 Pos (0.3406344 0.6593656) *
##    13) love=1 2433  832 Pos (0.3419647 0.6580353) *
##    7) great=1 3919 1023 Pos (0.2610360 0.7389640) *
```

```
# Sets side by side layout of the two plots
```

```
layout(matrix(c(1,2,3,4), nrow = 1, ncol = 2, byrow = TRUE), widths=c(2.5,2))
```

```
# Plots the model
```

```
fancyRpartPlot(model = popcorn.tree, main = NULL, sub = NULL)
```

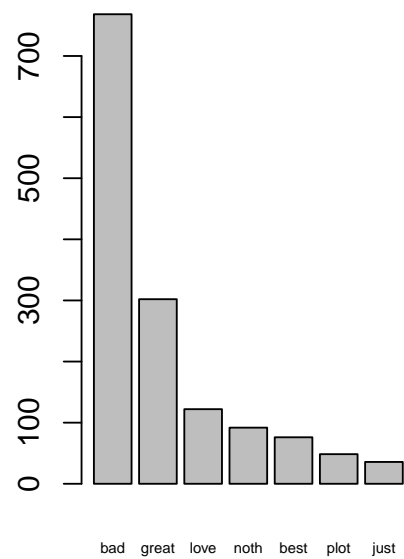
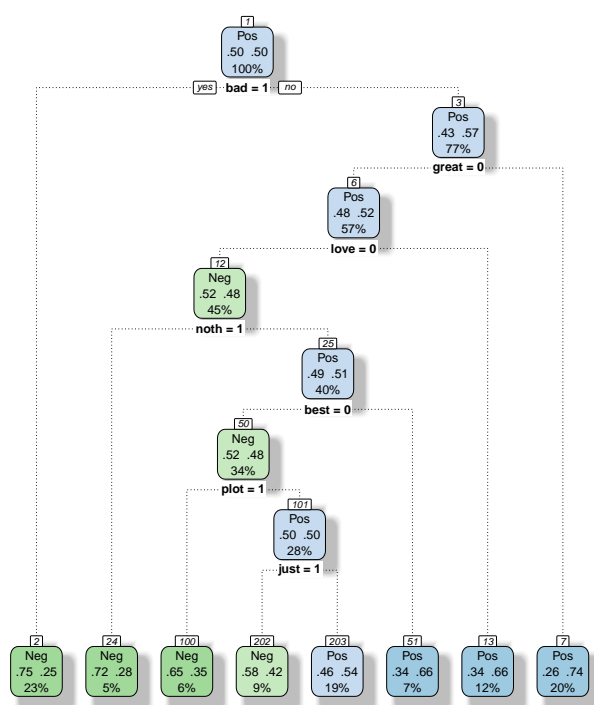
```
# Plots variable importance
```

```
barplot(popcorn.tree$variable.importance,
        cex.names = 0.5,
        sub = "Variable Importance")
```

```
popcorn.tree$variable.importance
```

```
##      bad      great      love      noth      best      plot      just
## 768.29715 302.00945 122.08563  91.86011  76.09715  48.46538  35.75477
```

Now, interpret the tree. This is explained in the active training sessions.



Variable Importance

Figure 3: Decision Tree

6.5 Complexity parameter

The complexity parameter α specifies how the cost of a tree $C(T)$ is penalized by the number of terminal nodes $|T|$, resulting in a regularized cost $C\alpha(T)$. This is not discussed the previous lessons as this is only applicable when you perform k-fold cross-validations. The package performs 10 folds as a default. This further tests the accuracy of the model even without a testing set. Note the misclassification error after cross-validation with the error rate when we evaluate against the testing set later.

This is represented as:

$$C\alpha(T) = C(T) + \alpha |T|$$

```
printcp(popcorn.tree) # Prints CP

##
## Classification tree:
## rpart(formula = sentiment ~ ., data = popcorn.train, method = "class",
##       model = T, parms = list(split = "information"))
##
## Variables actually used in tree construction:
## [1] bad    best  great just  love noth  plot
##
## Root node error: 9992/20000 = 0.4996
##
## n= 20000
##
##          CP nsplit rel error  xerror      xstd
## 1 0.228683      0  1.00000 1.01902 0.0070756
## 2 0.017064      1  0.77132 0.77132 0.0068882
## 3 0.015913      5  0.69496 0.73569 0.0068239
## 4 0.010000      7  0.66313 0.67394 0.0066887
```

In a nutshell the: misclassification error (33.13%) = root node error (49.96%) * relative error of lowest CP (66.313%). More statistics are available in `summary(popcorn.tree)`.

6.6 Model evaluation

With a good accuracy rate, typically above 80%, the model can be considered to perform good predictions. In machine learning and diagnostics, **F1-score** is considered as one the basic quantitative measures to describe how good the model is in terms of trade-off between precision and recall; this is calculated as:

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

To expand the formula to show precision and recall:

$$F_1 = 2 \frac{\frac{TP}{(TP+FP)} * \frac{TP}{(TP+FN)}}{(\frac{TP}{(TP+FP)} + \frac{TP}{(TP+FN)})}$$

In a classification model evaluation, precision and recall always come together, hence the F-measure. Wikipedia has a good example of this in [information retrieval](#) domain.

```

popcorn.prediction <- predict(object = popcorn.tree, # Tests model
                             newdata = popcorn.test, # with Testing set
                             type = "class") # Tells it is a classification prediction

# Build binary classification confusion matrix
popcorn.confusion <- table(Actual = popcorn.test$sentiment,
                           Predicted = popcorn.prediction)

tp <- popcorn.confusion[1,1] # True Positive
tn <- popcorn.confusion[2,2] # True Negative
fp <- popcorn.confusion[2,1] # False Positive / Type I error
fn <- popcorn.confusion[1,2] # False Negative / Type II error
n <- sum(popcorn.confusion) # Total records of testing set

popcorn.accuracy <- (tp + tn) / n # Accuracy rate = 66.78%
popcorn.error <- (fp + fn) / n # Error rate = 33.22%
popcorn.precision <- tp / (tp + fp) # Precision / Sensitivity = 69.50%
popcorn.recall <- tn / (tn + fn) # Recall / Specificity = 60.21%
popcorn.f1 <- 2 * popcorn.precision * popcorn.recall /
  (popcorn.precision + popcorn.recall) # F1-score / Harmonic mean
popcorn.oddsratio <- (tp * tn) / (fp * fn) # Odds ratio = 4 folds

```

Remember the misclassification rate from the cross-validation? It is very close to the error rate when the model is tested using the testing set. I encourage you to interpret the other measures.

7 Citations

7.1 References

- Jiawei Han, Micheleine Kamber and Jian Pei, *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 3rd ed., 2011
- Chary Aggarwal and Jiawei Hand (eds.), *Frequent Patter Mining*, Springer 2014
- Chris Manning and Hinrich Schütze, *Foundations of Statistical Natural Language Processing*, MIT Press. Cambridge, MA: May 1999.
- Mark Sanderson. *Test Collection Based Evaluation of Information Retrieval Systems*. Foundations and Trends in Information Retrieval 4(4): 247-375 (2010).
- Zhai, ChengXiang. *Statistical Language Models for Information Retrieval. Synthesis Lectures Series on Human Language Technologies*. Morgan & Claypool Publishers, 2008.
- Wil van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*, Springer Verlag, 2011, ISBN 978-3-642-19344-6
- Angela Chapman. Word2Vec - Google Code. Kaggle's competition for using Google's word2vec package for sentiment analysis.
- Xie Y (2013). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1482203530, .
- Xie Y (2014). "knitr: A Comprehensive Tool for Reproducible Research in R." In Stodden V, Leisch F and Peng RD (eds.), *Implementing Reproducible Computational Research*. Chapman and Hall/CRC. ISBN 978-1466561595, .

Williams GJ (2011). *Data Mining with Rattle and R: The art of excavating data for knowledge discovery*, series Use R! Springer. .

Hahsler M, Gruen B and Hornik K (2005). “arules - A Computational Environment for Mining Association Rules and Frequent Item Sets.” *Journal of Statistical Software*, 14(15), pp. 1-25. ISSN 1548-7660, .

Feinerer I, Hornik K and Meyer D (2008). “Text Mining Infrastructure in R.” *Journal of Statistical Software*, 25(5), pp. 1-54. .

7.2 Tools

R Core Team (2015). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. .

Xie Y (2015). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.10, .

Hahsler M, Buchta C, Gruen B and Hornik K (2014). *arules: Mining Association Rules and Frequent Itemsets*. R package version 1.1-6, .

Hahsler M and Chelluboina S (2014). *arulesViz: Visualizing Association Rules and Frequent Itemsets*. R package version 1.0-0, .

Therneau T, Atkinson B and Ripley B (2015). *rpart: Recursive Partitioning and Regression Trees*. R package version 4.1-9, .

Feinerer I and Hornik K (2014). *tm: Text Mining Package*. R package version 0.6, .

Bastian M., Heymann S., Jacomy M. (2009). Gephi: an open source software for exploring and manipulating networks. International AAAI Conference on Weblogs and Social Media.